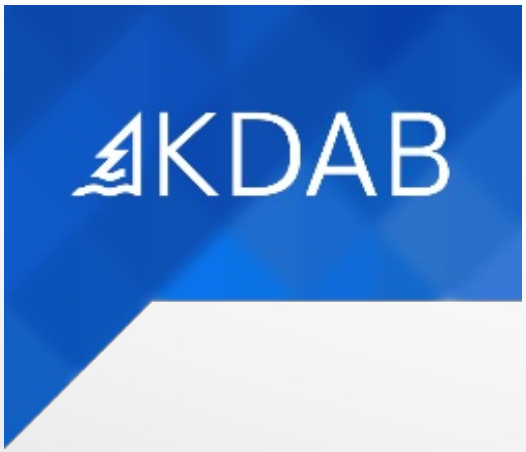
Qt World Summit 2019

5-6 November | Berlin, Germany

QStringView
Past, Present, Future

Marc Mutz, Senior Software Engineer at KDAB



- **Past**
- Present
- Future

In a QtWS far, far away...

QStringView, QStringView everywhere (QtWS 2017)

- QString vs. QStringView
- Why QStringView?
- Using QStringView
  - QStringView as an Interface Type
  - API Patterns for QStringView
  - Heterogeneous Associative Container Lookup
- Technical Deep Dives:
  - Managing Overloads
  - Contracts

https://www.youtube.com/watch?v=_9g5nYrCles

My First QStringView Function

```
 1 bool isValidFirstChar(QChar c)     { return c == '_' || c.isLetter(); }
 2 bool isValidFollowupChar(QChar c) { return c == '_' || c.isLetterOrNumber(
 3
 4 bool isValidIdentifier(const QString &s) noexcept {
 5     auto it = s.begin();
 6     const auto end = s.end();
 7     if (it == end)
 8         return false;
 9     if (!isValidFirstChar(*it++))
10         return false;
11     while (it != end)
12         if (!isValidFollowupChar(*it++))
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("_1"); // allocates
22     std::cout << std::endl;
23 }
```

My First QStringView Function

```
 1 bool isValidFirstChar(QChar c)     { return c == '_' || c.isLetter(); }
 2 bool isValidFollowupChar(QChar c) { return c == '_' || c.isLetterOrNumber(
 3
 4 bool isValidIdentifier(QStringView s) noexcept {
 5     auto it = s.begin();
 6     const auto end = s.end();
 7     if (it == end)
 8         return false;
 9     if (!isValidFirstChar(*it++))
10         return false;
11     while (it != end)
12         if (!isValidFollowupChar(*it++))
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"); // no longer allocates
22     std::cout << std::endl;
23 }
```

My First QStringView Function

```
 1 bool isValidFirstChar(QChar c)     { return c == '_' || c.isLetter(); }
 2 bool isValidFollowupChar(QChar c) { return c == '_' || c.isLetterOrNumber(
 3
 4 bool isValidIdentifier(QStringView s) noexcept {
 5     auto it = s.begin() + 1;
 6     const auto end = s.end();
 7     if (s.isEmpty())
 8         return false;
 9     if (!isValidFirstChar(s.front()))
10         return false;
11     while (it != end)
12         if (!isValidFollowupChar(*it++))
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"); // no longer allocates
22     std::cout << std::endl;
23 }
```

My First QStringView Function

```
 1 bool isValidFirstChar(QChar c)     { return c == '_' || c.isLetter(); }
 2 bool isValidFollowupChar(QChar c) { return c == '_' || c.isLetterOrNumber(
 3
 4 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)) // totally cheap
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"); // no longer allocates
22     std::cout << std::endl;
23 }
```

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 }
```

- Past
- **Present**
- Future

- most const QString functions are now available
- QLatin1String has the same features as QStringView
- multi-arg() can take any number and type of arguments
  - as long as convertible to QStringView, QLatin1String, QChar

New Multi-Arg

arg(str, ..., str) now available on QStringView/QLatin1String

- always returns QString
- arg(num, opts...) intentionally unsupported (awaits QFormattedNumber)

```
1 QString("%1, %2!").arg(QLatin1String("Hello"), QLatin1String("World"));
2 QStringLiteral("%1, %2!").arg(QLatin1String("Hello"), QLatin1String("World"));
3
```

New Multi-Arg

arg(str, ..., str) now available on QStringView/QLatin1String

- always returns QString
- arg(num, opts...) intentionally unsupported (awaits QFormattedNumber)

```
1 QString("%1, %2!").arg(QLatin1String("Hello"), QLatin1String("World"));
2 QStringLiteral("%1, %2!").arg(QLatin1String("Hello"), QLatin1String("World"));
3
4 QStringView(u"%1, %2!").arg(QLatin1String("Hello"), QLatin1String("World"))
5 QLatin1String("%1, %2!").arg(QLatin1String("Hello"), QLatin1String("World"));
```

- No allocation to create QString first
  - -or- no space waste d/t QStringLiteral use

New Multi-Arg (cont'd)

arg() now accepts more types than just QStrings

New Multi-Arg (cont'd)

arg() now accepts more types than just QStrings

- QLatin1String
- QStringView
- QChar

New Multi-Arg (cont'd)

arg() now accepts more types than just QStrings

- QLatin1String
- QStringView
- QChar
- and anything implicitly convertible to one of the above

New Multi-Arg (cont'd)

arg() now accepts more types than just QStrings

- QLatin1String
- QStringView
- QChar
- and anything implicitly convertible to one of the above
  - e.g. std::u16string!
  - facilitated by arg() being a template

New Multi-Arg (cont'd)

arg() now accepts more types than just QStrings

- QLatin1String
- QStringView
- QChar
- and anything implicitly convertible to one of the above
  - e.g. std::u16string!
  - facilitated by arg() being a template
- no conversion to QString for passing

New Multi-Arg (cont'd)

arg() now accepts more types than just QStrings

- QLatin1String
- QStringView
- QChar
- and anything implicitly convertible to one of the above
  - e.g. std::u16string!
  - facilitated by arg() being a template
- no conversion to QString for passing

```
QStringView(u"%1, %2!").arg(QLatin1String("Hello"), QLatin1String("World"))
```

New Multi-Arg (cont'd)

arg() now accepts more types than just QStrings

- QLatin1String
- QStringView
- QChar
- and anything implicitly convertible to one of the above
  - e.g. std::u16string!
  - facilitated by arg() being a template
- no conversion to QString for passing

```
1 QStringView(u"%1, %2!").arg(QLatin1String("Hello"), QLatin1String("World"))
2
3 using namespace std::string_literals;
4 QStringView(u"%1, %2%3").arg(u"Hello", u"World"s, QLatin1Char('!'));
```

New Multi-Arg (cont'd)

arg() can now take more than nine parameters

- implemented as a variadic template
  - on QString, too

New Multi-Arg (cont'd)

arg() can now take more than nine parameters

New Multi-Arg (cont'd)

arg() can now take more than nine parameters

- implemented as a variadic template

New Multi-Arg (cont'd)

arg() can now take more than nine parameters

- implemented as a variadic template
  - on QString, too

New Multi-Arg (cont'd)

arg() can now take more than nine parameters

- implemented as a variadic template
  - on QString, too

```
1 QStringView(u"%1 %2 %3 %4 %5 %6 %7 %8 %9 %10 %11 %12")
2     .arg(u"one", u"two", u"three", u"four", u"five", u"six",
3          u"seven", u"eight", u"nine", u"ten", u"eleven", u"twelve")
```

What About Unary Arg?

But my code needs str.arg(num)...

- Intentionally unsupported
- Want to get rid of the special treatment of numbers vs. strings

```
1 tr("%1 files downloaded, %2 MiB total")
2     .arg(numFiles, numBytes); // OOPS
```

What About Unary Arg?

But my code needs str.arg(num)...

- Intentionally unsupported
- Want to get rid of the special treatment of numbers vs. strings

```
1 tr("%1 files downloaded, %2 MiB total")
2     .arg(numFiles, numBytes); // OOPS
3
4 // PROPOSED:
5 tr("%1 files downloaded, %2 total")
6     .arg(locale().toString(numFiles, ~~~),  // returns QFormattedNumber<int>
7          locale().toFileSize(numBytes))     // returns something else
8 // == "12 files downloaded, 2.3 GiB total"
```

- piggy-back onto multi-arg() work
  - to make it easily extensible

- Past
- Present
- **Future**

Going Forward

- Complete Const QString API (Qt 5.15)
  - incl. split()
- QFormattedNumber / QParsedNumber (Qt ??)
- QUtf8String/View (Qt 6)
- QByteArray = array of std::byte (Qt 7? 8?)

Intentionally Unsupported

- mutating functions
  - → out of scope!
- unary arg()
  - → multi-arg() + QFormattedNumber
- number()
  - → fromNumber()
- toInt()/toDouble()/...
  - → toNumber()

Complete Const QString API (Qt 5.15)

- mainly missing:
  - QRegularExpression support
  - split()
    - factory function for QStringTokenizers (slide 15)
    - working implementation exists, didn't make the cut for 5.14
  - number → string → number
    - new API (slide 21)

Remember: QStringView to avoid allocations

KDAB

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

KDAB

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

Solution: QStringTokenizer

QStringTokenizer (cont'd)

QStringTokenizer is a *generator*

- C++20 coroutine, implemented in C++11

QStringTokenizer is a *generator*

- C++20 coroutine, implemented in C++11

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

QStringTokenizer is a *generator*

- C++20 coroutine, implemented in C++11

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

- with C++20:

```
1 auto trimmedValueEquals(const auto &s) {
2     return [s](const auto &e) { e.trimmed() == s; };
3 }
```

KDAB

QStringTokenizer is a *generator*

- C++20 coroutine, implemented in C++11

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

- with C++20:

```
1 auto trimmedValueEquals(const auto &s) {
2     return [s](const auto &e) { e.trimmed() == s; };
3 }
4
5 int countXsInCommaSeparatedList(QStringView s) {
6     return std::ranges::count_if(QStringTokenizer{s, u','}, trimmedValueEquals(u'X'));
7 }
```

QStringTokenizer (cont'd)

Direct use of QStringTokenizer requires C++17 CTAD

QStringTokenizer (cont'd)

Direct use of QStringTokenizer requires C++17 CTAD

- so split() will not go away:
- needed to deduce QStringTokenizer template arguments

QStringTokenizer (cont'd)

Direct use of QStringTokenizer requires C++17 CTAD

- so split() will not go away:
- needed to deduce QStringTokenizer template arguments

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

Object Lifetime

- Problem in QStringBuilder

```
1 QString getString();
```

Object Lifetime

- Problem in QStringBuilder

```
1 QString getString();
2
3 auto res = QStringLiteral("foo") % getString();
4 // OOPS
```

Object Lifetime

- Problem in QStringBuilder

```
1 QString getString();
2
3 auto res = QStringLiteral("foo") % getString();
4 // OOPS
```

- **not** a problem in QStringTokenizer

```
auto res = QStringTokenizer{getString(), u','};
// OK, QStringTokenizer keeps getString()'s result alive
```

Object Lifetime

- Problem in QStringBuilder

```
1 QString getString();
2
3 auto res = QStringLiteral("foo") % getString();
4 // OOPS
```

- **not** a problem in QStringTokenizer

```
auto res = QStringTokenizer{getString(), u','};
// OK, QStringTokenizer keeps getString()'s result alive
```

- should be added to QStringBuilder, too (Qt 6)

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)

KDAB

```
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 QParsedNumber<int> QString::toNumber<int>(~~~);
```

New Number Conversion API (cont'd)

```
1 if (auto r = QString::toInt(~~~)) {
2     use(*r);
3 } else {
4     // handle r.error()
5 }
```

New Number Conversion API (cont'd)

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

New Number Conversion API (cont'd)

```
1 use(QString::toInt(~~~));
```

What is QByteArray?

KDAB

QByteArray semantic overload:

- binary data
- UTF-8-encoded strings

KDAB

What does f(const QByteArray &) expect?

KDAB

(Partial) Solution: QUtf8String(View)

KDAB

QUtf8StringView: Like QLatin1String, but for UTF-8 char8_t*s

KDAB

QUtf8String: Like QString, but for UTF-8 char8_t*s

QLatin1StringView: needed?

KDAB

QString ← QUtf8String: implicit
QUtf8String ← const char8_t*: implicit
QUtf8String ← {QByteArray, const char*}: explicit

QByteArray: QVector

String Types Shuffle

Proposal for Qt string types (probably not Qt 6)

- UTF-16
  - QString: string, owner, char16_t/QChar
  - QStringView: string, observer, char16_t/QChar
- UTF-8
  - QUtf8String: string, owner, char8_t
  - QUtf8StringView: string, observer, char8_t
- Bytes
  - QByteArray: vector, owner, std::byte
  - std::span<std::byte>: array, observer, std::byte
- char/unsigned char/QLatin1String
  - unsupported!

Thank you!

www.kdab.com

marc.mutz@kdab.com