QAnyStringView
Variant String Views and Why You Care

Meeting C++ 2021 LT, 2021-11-12

Marc Mutz <marc.mutz@qt.io>

- **History**
- Present
- Future

QStringView Diaries: Masters of the Overloads

- Technical Deep Dive:
  - Overload Management

https://www.kdab.com/qstringview-diaries-masters-overloads/

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

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

StringViews, StringViews everywhere (Meeting C++ 2017)

- Heterogeneous Associative Container Lookup
- Technical Deep Dive:
  - Contracts

https://www.youtube.com/watch?v=0QFPKgvLhao

The Most Important API Design Principle? (Meeting C++ 2017 LT)

- Non-Owning Interface Idiom (NOIv1)

https://www.youtube.com/watch?v=JUUID0_NvQI

QStringView: Past, Present and Future (QtWS 2019)

- QStringTokenizer
- QLatin1String as a view
- new multiArg()

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

QStringView Diaries: Zero-Allocation String Splitting

- QStringTokenizer (update)

https://www.kdab.com/qstringview-diaries-zero-allocation-string-splitting/

Coroutines As An API Principle (Meeting C++ 2021)

- Non-Owning Interface Idiom v2 (NOIv2)

- History
- **Present**
- Future

- `qTokenize` / `QStringTokenizer` merged
- QLatin1String has survived
- QUtf8StringView merged
- QAnyStringView merged

QAnyStringView

QAnyStringView is a variant string view type

- can be either
  - QLatin1String
  - QUtf8StringView
  - QStringView
- figures out which one automatically:
  - `const char8_t*` → UTF-8
  - QLatin1String → Latin-1
  - `const char*` / `QByteArray` / etc → UTF-8 (Qt policy, will probably vanish in Qt 7 for u8"")
  - `const char16_t*` / `const QChar*` / etc → UTF-16
- unlike QStringView, can also be constructed from
  - QChar
  - char32_t(!)
  - QStringBuilder expression template(!)

QAnyStringView (cont'd)

Example: QObject::setObjectName()

```
1 void QObject::setObjectName(const QString &); // traditional
2
3 o.setObjectName("o");                  // allocates
4 o.setObjectName(QLatin1String("o"));   // allocates
5 o.setObjectName(QStringLiteral("o")); // doesn't allocate, but still temp QString
```

vs.

```
1 void QObject::setObjectName(QAnyStringView); // didn't make it into Qt 6
2
3 o.setObjectName("o");                  // doesn't allocate (UTF-8)
4 o.setObjectName(QLatin1String("o"));   // doesn't allocate (Latin-1)
5 o.setObjectName(QStringLiteral("o")); // pessimisation now (still temp QString)
```

QAnyStringView (cont'd)

Unlike QStringView

- QAnyStringView doesn't nicely overload with QString

```
1 void foo(const QString &);
2 void foo(QAnyStringView);
3
4 foo("hello"); // ERROR: ambiguous
```

- designed to replace QString functions
  - not overload them
  - making QString/QAnyStringView overloadable would have severely demoted QAnyStringView's flexibility
- QAnyStringView has very limited API
  - no splitting etc
- instead, use visit()
  - think std::visit(<lambda>, std::variant<~~~~>)

QAnyStringView Visitation

Instead of implementing public API as overload sets for QStringView / QChar / QLatin1String / QUtf8StringView

- implement *one* public function taking QAnyStringView
- farm out to L1/UTF-8/UTF-16 in the implementation only

Simple example:

```
1 void QObject::setObjectName(QAnyStringView name) {
2     d->objectName = // a QString
3         name.visit([auto name] { name.toString(); });
4     // conveniently already implemented for you:
5     d->objectName = name.toString();
6 }
```

If all we do is build a QString, why take a QAnyStringView?

Several reasons:

If all we do is build a QString, why take a QAnyStringView?

Several reasons:

- now one QString construction in library code (O(1))
  - vs. one QString construction per call site (O(N))

Why Do I Care?

If all we do is build a QString, why take a QAnyStringView?

Several reasons:

- now one QString construction in library code (O(1))
  - vs. one QString construction per call site (O(N))
- implementation can change independent of callers:

If all we do is build a QString, why take a QAnyStringView?

Several reasons:

- now one QString construction in library code (O(1))
  - vs. one QString construction per call site (O(N))
- implementation can change independent of callers:

```
1 void QObject::setObjectName(QAnyStringView name) {
2     d->objectName = // a std::u16string (for SSO)
3                     // or a QVarLengthArray<char16_t,32>
4         name.visit([](auto name) { ~~~ convert ~~~; });
```

If all we do is build a QString, why take a QAnyStringView?

Several reasons:

- now one QString construction in library code (O(1))
  - vs. one QString construction per call site (O(N))
- implementation can change independent of callers:

```
1 void QObject::setObjectName(QAnyStringView name) {
2     d->objectName = // a std::u16string (for SSO)
3                     // or a QVarLengthArray<char16_t,32>
4         name.visit([](auto name) { ~~~ convert ~~~; });
```

- maybe you're pre-processing the string anyway, causing a detach:

If all we do is build a QString, why take a QAnyStringView?

Several reasons:

- now one QString construction in library code (O(1))
  - vs. one QString construction per call site (O(N))
- implementation can change independent of callers:

```
1 void QObject::setObjectName(QAnyStringView name) {
2     d->objectName = // a std::u16string (for SSO)
3                     // or a QVarLengthArray<char16_t,32>
4         name.visit([](auto name) { ~~~ convert ~~~; });
```

- maybe you're pre-processing the string anyway, causing a detach:

```
1 QString QSettingsPrivate::actualKey(QAnyStringView key); // could return QVLA now
2 QVariant QSettings::value(QAnyStringView key) const { // tst_qsettings -= 7.3%
3     return d->get(d->actualKey(key));
4 }
```

If all we do is build a QString, why take a QAnyStringView?

Several reasons:

- now one QString construction in library code (O(1))
  - vs. one QString construction per call site (O(N))
- implementation can change independent of callers:

```
1 void QObject::setObjectName(QAnyStringView name) {
2     d->objectName = // a std::u16string (for SSO)
3                     // or a QVarLengthArray<char16_t,32>
4         name.visit([](auto name) { ~~~ convert ~~~; });
```

- maybe you're pre-processing the string anyway, causing a detach:

```
1 QString QSettingsPrivate::actualKey(QAnyStringView key); // could return QVLA now
2 QVariant QSettings::value(QAnyStringView key) const { // tst_qsettings -= 7.3%
3     return d->get(d->actualKey(key));
4 }
```

- maybe you're just parsing, and not storing
  - {QUuid,QVersionNumber,QColor}::fromString(QAnyStringView)

If all we do is build a QString, why take a QAnyStringView?

Several reasons:

- now one QString construction in library code (O(1))
  - vs. one QString construction per call site (O(N))
- implementation can change independent of callers:

```
1 void QObject::setObjectName(QAnyStringView name) {
2     d->objectName = // a std::u16string (for SSO)
3                     // or a QVarLengthArray<char16_t,32>
4         name.visit([](auto name) { ~~~ convert ~~~; });
```

- maybe you're pre-processing the string anyway, causing a detach:

```
1 QString QSettingsPrivate::actualKey(QAnyStringView key); // could return QVLA now
2 QVariant QSettings::value(QAnyStringView key) const { // tst_qsettings -= 7.3%
3     return d->get(d->actualKey(key));
4 }
```

- maybe you're just parsing, and not storing
  - {QUuid,QVersionNumber,QColor}::fromString(QAnyStringView)
- and if Qt couldn't be further from your day work...
  - have you looked at `std::basic_fstream` ctors lately?

- History
- Present
- **Future**

- Make QAnyStringView more widely applicable (ongoing)
- QFormattedNumber / QParsedNumber (Qt ??)
- QUtf8String / QAnyString (Qt ??)
- QByteArray = array of std::byte (Qt 7? 8?)

QAnyStringView Future Directions

- use it in many more places
- add conversions to other than QString
  - will simplify storing in QVLA/u16string
- QString::arg() support
  - will support non-allocating arg("foo")
    - currently still builds a temporary QString
- QStringBuilder support

Instead of me telling you again to wait for QFormattedNumber...

Instead of me telling you again to wait for QFormattedNumber...

... let's look at this cutie here:

Instead of me telling you again to wait for QFormattedNumber...

... let's look at this cutie here:

```
1  namespace detail {
2      template <typename Integral>
3      using array_for = std::array<std::numeric_limits<Integral>::digits10 + 1 /*-*/>;
4  }
5
6  template<typename Integral, std::enable_if_t<std::is_integral_v<Integral>, bool> = true>
7  auto to_l1(Integral x, detail::array_for<Integral> &&buffer = {})
8  {
9      const auto b = buffer.data.data();
10     const auto e = b + buffer.data.size();
11     auto r = std::to_chars(b, e, x);         // C++17
12     Q_ASSUME(r.ec == std::errc{});
13     return QLatin1String(b, r.ptr);
14  }
```

- uses C++17 `to_chars` - blazingly fast
- covers 99% of all `QString::number()` calls
- homework: get rid of the value-initialization of `buffer`

EOF