String Theory

Thiago Macieira

Qt Developer Days 2014

Who am I?

How many string classes does Qt have?

- **Present**
  - QString
  - QLatin1String
  - QByteArray
  - QStringLiteral (not a class!)
  - QStringRef
  - QVector<char>
- **Past**
  - QCString / Q3CString
- **Non-Qt**
  - std::string
  - std::wstring
  - std::u16string / std::u32string
  - Character literals ("", L"", u"", U"")

Character types, charsets, and codecs

What's a charset?

USASCII code chart

b_7 b_6 b_5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4	b_3	b_2	b_1	Column → / Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	–	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Legacy encodings

- **6-bit encodings**
- **EBCDIC**
- **UTF-1**

Examples modern encodings

- **Fixed width**
  - US-ASCII (ANSI X.3.4-1986)
  - Most DOS and Windows codepages
  - ISO-8859 family
  - KOI8-R, KOI8-U
  - UCS-2
  - UTF-32 / UCS-4
- **Variable width**
  - UTF-7
  - UTF-8, CESU-8
  - UTF-16
  - GB-18030
- **Stateful**
  - Shift-JIS
  - EUC-JP
  - ISO-2022

Unicode & ISO/IEC 10646

- **Unicode Consortium - http://unicode.org**
- **Character maps, technical reports**
- **The Common Locale Data Repository**

27F0 **Supplemental Arrows-A** **27FF**

	27F
0	⟰ 27F0
1	⟱ 27F1
2	⟲ 27F2
3	⟳ 27F3
4	⟴ 27F4

Arrows

27F0 ⟰ UPWARDS QUADRUPLE ARROW
→ 290A ⤊ upwards triple arrow
→ 2B45 ⬅ leftwards quadruple arrow

27F1 ⟱ DOWNWARDS QUADRUPLE ARROW
→ 290B ⤋ downwards triple arrow

27F2 ⟲ ANTICLOCKWISE GAPPED CIRCLE ARROW
→ 21BA ↺ anticlockwise open circle arrow
→ 2940 ⥀ anticlockwise closed circle arrow

27F3 ⟳ CLOCKWISE GAPPED CIRCLE ARROW
→ 21BB ↻ clockwise open circle arrow
→ 2941 ⥁ clockwise closed circle arrow

27F4 ⟴ RIGHT ARROW WITH CIRCLED PLUS
→ 2B32 ⬲ left arrow with circled plus

Long arrows

The long arrows are used for mapping whereas the short forms would be used in limits. They are also needed for MathML to complete mapping to the AMSA sets.

27F5 ⟵ LONG LEFTWARDS ARROW
→ 2190 ← leftwards arrow

Codec

- **enCOder/DECoder**
- **Usually goes through UTF-32 / UCS-4**

Codecs in your editor / IDE

- **Qt Creator: UTF-8**
- **Unix editors: locale[1]**
- **Visual Studio: locale[2] or UTF-8 with BOM**

1) modern Unix locale is usually UTF-8; it always is for OS X
2) Windows locale is almost never UTF-8

Codecs in Qt

- **Built-in**
  - Unicode: UTF-8, UTF-16, UTF-32 / UCS-4
- **ICU support**

C++ character types

Type	Width	Literals	Encoding
`char`	1 byte	`"Hello"`	arbitrary
		`u8"Hello"`	UTF-8
`wchar_t`	Platform-specific	`L"Hello"`	Platform-specific
`char16_t` (C++11)	At least 16 bits	`u"Hello"`	UTF-16
`char32_t` (C++11)	At least 32 bits	`U"Hello"`	UTF-32

Using non-basic characters in the source code

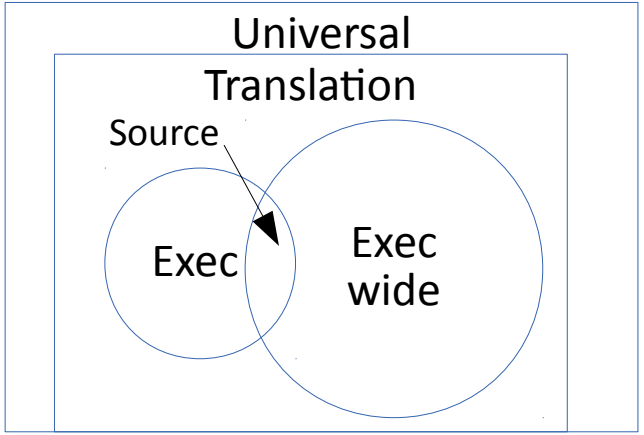
- **Often, bad idea**
  - Compiler-specific behaviour

```
char msg[] = "How are you?\n"
             "¿Como estás?\n"
             "Hvordan går det?\n"
             "お元気ですか？ \n"
             "Как поживаешь?\n"
             "Τι κάνεις;\n";
```

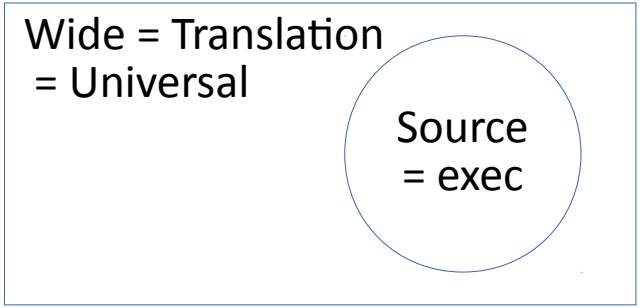
The five C and C++ charsets

- (Basic/Extended) Source character set
- (Basic/Extended) Execution character set
- (Basic/Extended) Execution wide-character set
- Translation character set
- Universal character set

Required



But usually



Writing non-English

- **C++11 Unicode strings**

```
return QStringLiteral(u"Hvordan g\u00E5r det?\n");
```

- **Regular escape sequences**

```
return QLatin1String("Hvordan g\xE5r det?\n") +
        QString::fromUtf8("\xC2\xBFComo est\xC3\xA1s?");
```

Qt support

Recalling Qt string types

- **Main classes**
  - QString
  - QLatin1String
  - QByteArray
- **Other**
  - QStringLiteral
  - QStringRef

Qt string classes in detail

Type	Overhead	Stores	8-bit clean?
`QByteArray`	16 / 24 bytes	`char`	Yes
`QString`	16 / 24 bytes	`QChar`	No (stores 16-bit!)
`QLatin1String`	Non-owning	`char`	N/A
`QStringLiteral`		`Same as QString`	
`QStringRef`	Non-owning	`QString*`	No

Remember your encoding

```
while (file.canReadLine()) {
    QString line = file.readLine();
    doSomething(line);
}
```

QString implicit casting

- **Assumes that char* are UTF-8**
  - Constructor
  - `operator const char*() const`
- **`Use QT_NO_CAST_FROM_ASCII and QT_NO_CAST_TO_ASCII`**

QByteArray

- **Any 8-bit data**
- **Allocates heap, with 16/24 byte overhead**

```
qint64 read(char *data, qint64 maxlen);
QByteArray read(qint64 maxlen);
QByteArray readAll();
qint64 readLine(char *data, qint64 maxlen);
QByteArray readLine(qint64 maxlen = 0);
virtual bool canReadLine() const;
```

QLatin1String

- **Latin 1 (ISO-8859-1) content**
  - Not to be confused with Windows 1252 or ISO-8859-15
- **No heap**

```
bool startsWith(const QString &s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const;
bool startsWith(const QStringRef &s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const;
bool startsWith(QLatin1String s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const;
bool startsWith(QChar c, Qt::CaseSensitivity cs = Qt::CaseSensitive) 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(QLatin1String s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const;
bool endsWith(QChar c, Qt::CaseSensitivity cs = Qt::CaseSensitive) const;
```

QStringLiteral

- **Read-only, shareable UTF-16 data***
- **No heap, but 16/24 byte overhead**

```
#  define QStringLiteral(str) \
    ([]() -> QString { \
        enum { Size = sizeof(QT_UNICODE_LITERAL(str))/2 - 1 }; \
        static const QStaticStringData<Size> qstring_literal = { \
            Q_STATIC_STRING_DATA_HEADER_INITIALIZER(Size), \
            QT_UNICODE_LITERAL(str) }; \
        QStrringDataPtr holder = { qstring_literal.data_ptr() }; \
        const QString s(holder); \
        return s; \
    }())
```

*) Depends on compiler support: best with C++11 Unicode strings

Standard Library types

- **`std::string`**
  - `QString::fromStdString` – `QString::toStdString`
- **`std::wstring`**
  - `QString::fromStdWString` – `QString::toStdWString`
- **`std::u16string` (C++11)**
- **`std::u32string` (C++11)**

C++11 (partial) support

```
static QString fromUtf16(const char16_t *str, int size = -1)
{ return fromUtf16(reinterpret_cast<const ushort *>(str), size); }
static QString fromUcs4(const char32_t *str, int size = -1)
{ return fromUcs4(reinterpret_cast<const uint *>(str), size); }
```

Which one is best? (1)

```
bool startsWith(const QString &s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const;
bool startsWith(const QStringRef &s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const;
bool startsWith(QLatin1String s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const;
bool startsWith(QChar c, Qt::CaseSensitivity cs = Qt::CaseSensitive) const;
```

```
return s.startsWith("Qt Dev Days");
```

```
return s.startsWith(QLatin1String("Qt Dev Days"));
```

```
return s.startsWith(QStringLiteral("Qt Dev Days"));
```

Which one is best? (2)

```
QString message()
{
    return "Qt Dev Days";
}
```

```
QString message()
{
    return QLatin1String("Qt Dev Days");
}
```

```
QString message()
{
    return QStringLiteral("Qt Dev Days");
}
```

Which one is best? (3)

```
QString message()
{
    return "Qt Dev Days " + QDate::currentDate().toString("yyyy");
}
```

```
QString message()
{
    return QLatin1String("Qt Dev Days ") + QDate::currentDate().toString("yyyy");
}
```

```
QString message()
{
    return QStringLiteral("Qt Dev Days ") + QDate::currentDate().toString("yyyy");
}
```

The fast operator +

```
ipv4Addr += number(address >> 24)
            + QLatin1Char('.')
            + number(address >> 16)
            + QLatin1Char('.')
            + number(address >> 8)
            + QLatin1Char('.')
            + number(address);
```

```
ipv4Addr += number(address >> 24);
ipv4Addr += QLatin1Char('.');
ipv4Addr += number(address >> 16);
ipv4Addr += QLatin1Char('.');
ipv4Addr += number(address >> 8);
ipv4Addr += QLatin1Char('.');
ipv4Addr += number(address);
```

- **Use QT_USE_FAST_OPERATOR_PLUS**

Simple rules to use

- **Always know your encoding**
- **Choose the right type:**

1) QByteArray for non-UTF16 text or binary data
2) QString for storage, QStringRef for non-owning substrings
3) QLatin1String if function takes QLatin1String
4) QStringLiteral if you're not about to reallocate

Thiago Macieira
thiago.macieira@intel.com
http://google.com/+ThiagoMacieira