Containers and Strings
Why the Implementation Matters

Barbara Geller & Ansel Sermersheim

CppNow - May 2017

Introduction

- Overview / Biography
- String terminology
- How other libraries handle Strings
- CsString to the rescue
- CsString integrated with CopperSpice
- DoxyPress improved String handling

Overview / Biography

- CopperSpice
- DoxyPress
- CsSignal library
- libGuarded library
- CsString library ⇦ *you are here*

Overview / Biography

- CopperSpice
    - initial release - May 2014
    - run time counter registration
        - replaces moc and improves introspection
        - allows reflections of templated classes
    - build system - autotools or cmake
    - improved the signal / slot system
    - contains a set of String classes inherited from Qt
        - QString, QByteArray, QLatin1String, QChar

Overview / Biography

- CopperSpice - Documentation
  - DoxyPress was used to generate all CS documentation
    - improved readability and accuracy
    - full API documentation with class diagrams
    - overview documentation
      - build instructions
      - how to migrate from Qt
      - setting up a CS project
      - CS development timeline

Overview / Biography

- CopperSpice - Containers
  - implementation matters more in library design
  - the mistakes may need to be supported
  - a redesign can be painful for the library developers and the users

- Example
  - all of the containers inherited by CS were custom classes
  - implementing containers by hand requires an enormous amount of continuous maintenance
    - move semantics
    - variadic templates
    - ranges?

- CopperSpice - Containers
  - removed legacy sequential containers, reimplemented using the STL containers
  - chose composition instead of inheritance
  - easy to add full support for move semantics
  - maintained and extended the CopperSpice API
  - added support for the STL API
    - append() vs push_back()
    - isEmpty() vs empty()

- CopperSpice - Containers
  - QList extremely inefficient and error prone
  - recommended by Qt developers to avoid, use QVector
  - QVector and QList implementation used “copy on write”

  - as of CopperSpice 1.4.0
    - QVector uses std:vector
    - QList uses std::deque

```
// qregion.cpp, oldRects is never used anywhere
// undocumented, looks like dead code
QVector<QRect> oldRects = dest.rects;
```

Overview / Biography

- DoxyPress
  - initial release - November 2015
  - documentation tool
  - various output formats are available
  - option to parse C++ source code using clang
  - written in C++
  - uses the CopperSpice String classes
  - processes a great deal of text

Overview / Biography

- Diamond
  - programmers editor
  - written in C++
  - uses the CopperSpice String classes
  - processes a great deal of text

Overview / Biography

- CopperSpice
  - CsSignal library
    - initial release May 2016
    - uses libGuarded library
    - fully integrated with CopperSpice
  - CsString library
    - initial release May 2017
    - partially integrated with CopperSpice

String Terminology

- Part II

String Terminology

- Character Set
  - collection of symbols
  - the set does not associate any values to these symbols
  - unordered list

  - Examples
    - Latin character set is used in English and most European languages
    - Greek character set is used only by the Greek language

String Terminology

- Character Encoding
  - the values associated with a character set
  - confusing terminology
  - better term is *Character Map*

String Terminology

- Coded Character Set
  - combination of a character set and a character map
  - Example
    - ASCII is a coded character set
    - ISO-8859-1 is a coded character set
      - latin script, used extensively in western Europe
    - KOI8-R is a coded character set
      - cyrillic script, used extensively in Russia

String Terminology

- Code Point or Code Position
    - character encoding terminology which refers to the numerical values defined by the Unicode standard
    - code points and characters are not the same
    - working with strings you need to think in terms of code points and not characters

    - atomic unit of text
    - 32-bit integer data type
    - lower 21-bits represent a valid code point and the upper 11-bits are zero

String Terminology

- Code unit or Storage unit
  - describes the unit of storage for an encoded code point
  - in UTF-8 the code unit is 8-bits
  - in UTF-16 the code unit is 16-bits

- Basic Multilingual Plane (BMP)
  - first 64k code points in Unicode
  - set of characters which fit into 2 bytes in UTF-16
  - contains characters for almost all modern languages and a large number of symbols

String Terminology

- ASCII
  - 7-bit coded character set finalized in 1968
  - 128 characters from 00 to 7F which match the corresponding Unicode code points

  - ASCII is often incorrectly used to refer to various 8-bit coded character sets which just happen to include the ASCII characters in the first 128 code points

String Terminology

- Latin-1
  - Latin Alphabet Number 1, also known as ISO-8859-1
  - 8-bit coded character set published in 1987
  - 191 characters from the Latin script
  - later used in the first 256 code points of Unicode
  - Latin-1 is a superset of the ASCII standard
  - used in the US, Western Europe, much of Africa

  - many other ISO Latin character sets which support Central Europe, Greek, Hebrew, and other languages

What is Character Encoding

- Example: latin capital letter A
  - symbol A code point value of U+0041
  - UTF-8 this is represented by one byte
  - UTF-16 this is represented by two bytes
  - one code point, one storage unit in either character encoding

- Example: rightwards arrow with corner downwards
  - symbol ↴ code point value of U+21B4
  - UTF-8 this is represented by three bytes, three storage units
  - UTF-16 this is represented by two bytes, one storage unit
  - always one code point, variable number of storage units

What is Character Encoding

- Example: musical symbol eighth note
  - symbol ♪ code point value of U+1d160
  - UTF-8 this is represented by four bytes, four storage units
  - UTF-16 this is represented by four bytes, two storage units
  - always one code point, variable number of storage units
  - outside the BMP

What is Unicode

- Unicode code points are by definition 32-bits
  - working with Unicode code points there is no choice, everything is a 32-bit value
  - Unicode Consortium realized the majority of the romance languages use the Latin alphabet and most of these symbols can be represented using 8-bits
  - the remainder of the symbols need 16-bits or 32-bits
  - it did not make sense to expect everyone to use a 32-bit character encoding when most text can be represented in 8-bits or 16-bits

What is Unicode

- UTF-8
  - variable length encoding
  - better encoding since there are numerous code points which only require one byte instead of two bytes in UTF-16
  - since the storage units are individual bytes there is no concept of big-endian versus little-endian

  - implementing UTF-8 requires a mechanism to calculate how many bytes comprise a single code point
  - this process is simpler than in UTF-16

What is Unicode

- UTF-16
  - variable length encoding
  - it is misleading to say Unicode can be represented in a 16-bit format
  - creates a lot of confusion and rarely implemented correctly

  - implementing UTF-16 requires a mechanism to calculate how many bytes comprise a single code point
  - more difficult to test for correctness
  - poor choice for encoding since it is both too narrow for many code points and too wide for the basic Latin character set

What is Unicode

- Companies like Microsoft may have selected a text encoding without really thinking things through, they elected to adopt UTF-16 as the native encoding for Unicode on Windows

- Languages like Java and Qt followed suit

- The 16-bit encoding seemed attractive and the correct choice at that time

- Languages, operating systems, and application developers learned from the struggles of existing string implementations and realized UTF-8 was the better option

Most Important Fact about Encodings

- Quote from “Joel On Software” in 2003
  - “It does not make sense to have a string without knowing what encoding it uses. You can no longer stick your head in the sand and pretend that ‘plain’ text is ASCII.”

  - strings many not be dazzling or feel cutting edge but they are a major part of nearly every application
  - you really need to know what encoding an email is in or you simply can not interpret or display it correctly
  - searching can be impossible if you are unable to decipher a string correctly

Unicode Timeline

- 1991
  - release UCS-2, 16-bit storage (2 bytes, fixed width)
- 1992
  - MFC Version 1.0 release
    - CString uses UCS-2
    - Microsoft moved to UTF-16 with Windows XP
- 1993
  - release UCS-4, 32-bit storage (4 bytes, fixed width)
- 1995
  - Java version 1.0 string class uses UCS-2

Unicode Timeline

- 1996
  - release UTF-8 (1-4 bytes, variable width)
  - release UTF-16 (2 or 4 bytes, variable width)

- 1999
  - TrollTech releases Qt 2.0
    - QString is the native string class, uses UTF-16
    - characters above 64k are stored using two 16-bit QChars which the user must “glue” together

- 2001
  - release UTF-32

Unicode Timeline

- 2005
  - Java Version 5.0 string class uses UTF-16

- 2017
  - release CsString
    - full Unicode aware string library
    - support for UTF-8 and UTF-16
    - additional encodings can smoothly and easily be added

How other libraries handle Strings

- Part III

How other libraries handle Strings

- What prompted development of CopperSpice
  - where Qt could be improved
    - build systems
    - templates
    - atomics
    - containers
    - signals / slots
    - threading
    - modern C++
    - unicode strings ⇦ *you are here*

How other libraries handle Strings

- What prompted development of DoxyPress
  - where Doxygen could be improved
    - templates
    - containers
    - readable, maintainable, modular
    - modern C++
    - unicode strings ⇦ *you are here*

How other libraries handle Strings

- What STL does not support
  - std::string
    - uses 8-bit storage
    - no mechanism to specify encoding
  - std::wstring
    - uses 16-bit or 32-bit storage
    - no mechanism to specify encoding
  - unicode strings ⇦ *you are here*

How other libraries handle Strings

- What prompted development of CsString
  - Unicode
    - ASCII, Latin-1, UCS-2, UCS-4, UTF-8, UTF-16, UTF-32
  - MFC
    - UCS-2, UTF-16
  - Java
    - UCS-2, UTF-16
  - std::string
    - no encoding
  - QString
    - UTF-16
  - C#
    - UTF-16

How other libraries handle Strings

- DoxyPress
  - original code for text processing
    - used Qt 1.9 QCString, QGString, or const char *
    - QCString and QGString used 8-bit storage, no encoding
    - both string classes roughly equivalent to std::string, they have an implicit conversion to char *
  - refactored every QCString and QGString to use a CopperSpice QString (UTF-16)

How other libraries handle Strings

- DoxyPress - Problem 1
  - QCString returns a null character when accessing past the end of the string
  - switching to QString resulted in many run time crashes, debugging was a nightmare

- DoxyPress - Problem 2

```
QString text = “List of Overloaded Public and Protected Methods”;
m1(text.toUtf8().constData());


void m1(const char * data) {
    m2(data);
}

void m2(const QString & phrase) {
    printOut(phrase);
}
```

How other libraries handle Strings

- DoxyPress - Problem 2
  - “List of Overloaded Public and Protected Methods”
  - German

Liste der überladenen öffentlichen und geschützten Methoden
?

  - Russian

Список перегруженных общедоступных и защищенных методов
?

How other libraries handle Strings

- DoxyPress - Problem 2
  - “List of Overloaded Public and Protected Methods”

  - German

Liste der überladenen öffentlichen und geschützten Methoden
Liste der Ã□Â¼berladenen Ã□Â¶ffentlichen und geschÃ□Â¼tzten Methoden

  - Russian

Список перегруженных общедоступных и защищенных методов
Ð¡Ð¿Ð¸Ñ□Ð¾Ðº Ð¿ÐµÑ□ÐµÐ³Ñ□Ñ□Ð¶ÐµÐ½Ð½Ñ□Ñ
Ð¾Ð±Ñ□ÐµÐ´Ð¾Ñ□Ñ□Ñ□Ð¿Ð½Ñ□Ñ
Ð¸ Ð·Ð°Ñ□Ð¸Ñ□ÐµÐ½Ð½Ñ□Ñ
Ð¼ÐµÑ□Ð¾Ð´Ð¾Ð²

- DoxyPress - Problem 2

```
QString text = “List of Overloaded Public and Protected Methods”;
m1(text.toUtf8().constData());

void m1(const char * data) {
    // data points to a UTF-8 encoded string, but does not know it
    m2(data);
}

void m2(const QString & phrase) {
    // QString(const char *) constructor assumes Latin-1
    printOut(phrase);
}
```

- Any text would be corrupted if it contains code points past 7F

CsString to the Rescue

- Part IV

CsString to the Rescue

- What should we retain?
  - 8-bit storage is more useful and versatile than 16 bit
- What should we change?
  - add a way to specify an encoding format
  - encoding format needs to adhere to Unicode
  - provide a mechanism to add a new encoding format without having to change the base string class

CsString to the Rescue

- CsBasicString
  - foundation class
  - templated class <typename E, typename A>
    - encoding
    - allocator

  - consists of a sequence of code points where each one is represented by a single 32-bit CsChar
  - implements a safe subset of std::string methods
  - supports conversion between existing encodings

CsString to the Rescue

- CsBasicString
  - commonly used instantiations

```
using CsString       = CsBasicString<utf8>;
using CsString_utf8  = CsBasicString<utf8>;
using CsString_utf16 = CsBasicString<utf16>;
```

CsString to the Rescue

- What is a String?
  - const char *
  - std::string
  - std::wstring
  - std::vector<char>
  - boost::string_ref
  - quoted text
  - quoted string
  - string literal
  - array of characters

CsString to the Rescue

- What data types do you see?
    - Example 1
        - const char * str1 = “abc”;
        - CsString str2 = str1;
    - Example 2
        - CsString str3 = “abc”;

CsString to the Rescue

- What data types do you see?
  - (Ex 1) const char * str1 = “abc”;
    - C Style String, initialized with a string literal
  - CsString str2 = str1;
    - unsafe
  - (Ex 2) CsString str3 = “abc”;
    - CsString, initialized with a string literal
  - a string literal is an expression
  - the data type for “abc” is “array of 4 chars”

- Implementation
  - CsString, initialized with a C style String
  - CsString, initialized with a string literal

```
template <typename T,
   typename = typename std::enable_if<
      std::is_same<T, const char *>::value ||
      std::is_same<T, char *>::value>::type>
CsBasicString(const T &str);


template <int N>
CsBasicString(const char (&str)[N]);
```

CsString to the Rescue

- API requirement for CsString
  - a string library should seamlessly support string literals
  - CsString must provide constructors and methods like operator!= and operator+= which take a string literal

```
CsString str(“xyz”);

if (str != “abc”) {
   return false;
}


str += “123”;
```

CsString to the Rescue

- Another type of string literal
  - how do you construct a string with non ASCII characters?

```
// sample code
CsString data(U"ABCD↴");
```

```
// constructor
CsBasicString(const char32_t * str);
```

CsString to the Rescue

- Another type of string literal
  - UTF-8 string literal (unsupported, has consequences)
    - u8"ABCD⤵"
    - const char[]
  - UTF-16 string literal (unsupported, may implement)
    - u"ABCD⤵"
    - const char16_t[]
  - UTF-32 string literal (currently supported)
    - U"ABCD⤵"
    - const char32_t[]

CsString to the Rescue

- Passing a multi-byte string literal
  - not safe at present, code produces a warning
  - data assumed to be Latin-1, which clearly may not be true
  - alternative implementations are under consideration

```
// sample code
CsString data("↴");
```

```
// output
code points in data : e2 86 b4
contents of data     : â†´          (mangled, 86 is non printable)
```

CsString to the Rescue

- Design of CsBasicString
  - has a private container which stores the data
  - currently using std::vector
    - (future) implement small_vector for efficiency

- CsBasicString<utf8>
  - utf8 is a data type which implements the UTF-8 encoding

- CsBasicString<utf16>
  - utf16 is a data type which implements the UTF-16 encoding

CsString to the Rescue

- Design of CsBasicString<utf8>
  - std::vector contains the raw UTF-8 data
  - m_string is the private data member
  - m_string.begin() and m_string.end() are std::vector iterators
  - these iterators and the data for m_string are private

- Accessing the data
  - since the values in m_string represent code points how do you walk through the vector, only seeing whole code points
  - how do you expose iterators to a CsBasicString

CsString to the Rescue

- Design of CsBasicString<utf8>

	A	B	C	↴			\0	
std::vector	0	1	2	3	4	5	6	7
CsString	0	1	2	3	x	y	4	

CsString to the Rescue

- Encoding.h

```
class utf8 {
  public:
    using storage_unit = uint8_t;

    template <typename Container>
    static typename Container::const_iterator insert( ... )

    static int walk( ... )

    static CsChar getCodePoint( ... )

  private:
    static int numOfBytes( ... )
}
```

CsString to the Rescue

```
template <typename Container>
static typename Container::const_iterator insert( Container &str1,
      typename Container::const_iterator iter, CsChar c, int count = 1)  {

uint32_t value = c.unicode();

for (int x = 0; x < size; ++x) {
  if (value <= 0x007F) {
    iter = str1.insert(iter, value);

  } else if (value <= 0x07FF) {
    iter = str1.insert(iter, ((value) & 0x3F) | 0x80);
    iter = str1.insert(iter, ((value >> 6) & 0x1F) | 0xC0);
  }

  ( continued . . . )
```

CsString to the Rescue

```
      } else if (value <= 0xFFFF) {
         iter = str1.insert(iter, ((value ) & 0x3F) | 0x80);
         iter = str1.insert(iter, ((value >> 6 ) & 0x3F) | 0x80);
         iter = str1.insert(iter, ((value >> 12) & 0x0F) | 0xE0);

      } else {
         iter = str1.insert(iter, ((value ) & 0x3F) | 0x80);
         iter = str1.insert(iter, ((value >> 6 ) & 0x3F) | 0x80);
         iter = str1.insert(iter, ((value >> 12) & 0x3F) | 0x80);
         iter = str1.insert(iter, ((value >> 18) & 0x07) | 0xF0);
      }
   }

   return iter;
}
```

CsString to the Rescue

- Testing
  - twelve usage tests containing 85 test points
  - included with CsString, test folder
  - human readable output

  - unit test containing 1200 test points
  - development testing
  - every test performed on UTF-8 and UTF-16
  - all tests are validated against std::string, when possible

CsString Testing

- Unit Test 6

```
CsString::CsString str1("Ending character is 3 bytes");
str1.append(UCHAR('⤵'));
```

```
Insert 2 left arrows at the 7th character
      Ending↵↵ character is 3 bytes ⤵

Insert string literal at the 7th character
      Ending [string literal] ↵↵ character is 3 bytes ⤵

Replace string literal at the 7th character
      Ending { new string text } character is 3 bytes ⤵
```

CsString Testing

- Unit Test 7

```
Original String: ABCD↴¿E♪F

Walk backwards: F
Walk backwards: ♪
Walk backwards: E
Walk backwards: ¿
Walk backwards: ↴
Walk backwards: D
Walk backwards: C
Walk backwards: B
Walk backwards: A
```

Note: musical symbol 8th note is U+1D160, outside BMP

CsString Testing

- Unit Test 7

```
Original String: ABCD↴¿E♪F
Substring beginning at 3, length 4: D↴¿E

Erase A element: BCD↴¿E♪F
Erase B element: CD↴¿E♪F
Erase C element: D↴¿E♪F
Erase D element: ↴¿E♪F
Erase ↴ element: ¿E♪F
Erase ¿ element: E♪F
Erase E element: ♪F
Erase ♪ element: F
Erase F element:
```

CsString Testing

- Unit Test 8

```
Original String (↴ is 3 bytes): ABCD↴

String - size_storage()      : 7
String - size_codePoints()   : 5
String - size()              : 5
String - length()            : 5

Copy original string from begin() + 2 : CD↴
Substring beginning at 3, length 2    : D↴
```

CsString integrated with CopperSpice

- Part V

CsString integrated with CopperSpice

- QString
  - current string class is UTF-16
  - does not fully support code points outside the BMP

- QString8
  - beta release CopperSpice 1.4.1 (released May 1 2017)
  - production release CopperSpice 1.5.0

- QString16
  - pending

CsString integrated with CopperSpice

- QString8 enhancements
  - arg()
    - similar to printf()
    - around 20 different versions
    - refactor using variadic templates
  - remove QLatin1String
    - wrapper for a const char *
    - unnecessary since CsString can decipher between a C style string and a string literal

DoxyPress improved String handling

- Part VI

DoxyPress improved String handling

- Resolved issue
  - during usage testing we discovered ⤵ was not appearing in the html output
  - lex rules were matching a single byte at a time
  - required another rule to decipher when a byte was part of multi-byte code point
  - issue discovered in multiple places

DoxyPress improved String handling

- Switching from QString to QString8
  - will reduce memory usage by 50%
  - continuous conversions between UTF-16 and UTF-8
    - const char * result = data.toUtf8().constData();

Putting it all Together

- What is next?

Putting it all Together

- Piece by piece
  - developing CopperSpice proved we needed to design a standalone Signal / Slot library (CsSignal)
  - deadlocks in CsSignal demanded a threading library
  - unable to document CopperSpice we created DoxyPress and switched parsing from lex to clang for C++
  - mangled text required a Unicode aware string library

  - CsSignal uses libGuarded
  - CopperSpice uses CsSignal and CsString
  - DoxyPress uses CopperSpice

Future Plans

- CsString
  - add ISO-8859-1 encoding (maybe others)
  - implement small string optimization
  - add locale aware comparison using Unicode algorithms
  - add normalization functions

- libGuarded
  - associative containers
  - lock free containers

Future Plans

- CopperSpice
  - complete QString8 and QString16
  - redesign QMap and QHash leveraging STL containers
  - optimize QVariant
  - lambda based indexOf and lastIndexOf, all container classes
  - MSVC using clang front end, if possible

- CsSignal
  - improve move semantics

Future Plans

- DoxyPress
  - add parsing support for clang 3.8 and clang 3.9
  - optimize clang integration used in parsing
  - refactor comment parser
  - improve unicode support

Libraries & Applications

- CopperSpice
  - libraries for developing GUI applications
- PepperMill
  - converts Qt headers to CS standard C++ header files
- CsSignal Library
  - thread aware signal / slot library
- CsString Library
  - unicode aware string support library
- LibGuarded
  - multithreading library for shared data

Libraries & Applications

- KitchenSink
  - one program which contains 30 demos
  - links with almost every CopperSpice library

- Diamond
  - programmers editor which uses the CS libraries

- DoxyPress & DoxyPressApp
  - application for generating documentation for a variety of computer languages in numerous output formats

Where to find our libraries

- www.copperspice.com
- download.copperspice.com
- forum.copperspice.com

- ansel@copperspice.com
- barbara@copperspice.com

- Questions? Comments?