# Strings, Text and Formatting

**Working with strings, formatting, tokenizing, regular expressions and text encodings.**

# Overview

- String Functions
- Formatting and Parsing Numbers
- Tokenizing Strings
- Regular Expressions
- Text Encodings

# String Functions

- POCO provides a bunch of function templates for frequently used string operations:
  - trimming (whitespace removal)
  - case conversion
  - case-insensitive comparison
  - character translation and substring replacement
  - concatenation
- `#include "Poco/String.h"`
- These work with `std::string` and `std::wstring`.

# String Functions (cont'd)

- Many functions come in two variants:
  - a function that returns a new string and leaves the original string unmodified,
  - a function that directly modifies the original string.
- The latter variants are called "in place" functions and have inPlace appended to the function name.
- All functions are in the Poco namespace.

# Trimming

- std::[w]string trimLeft(const std::[w]string& str)
  returns a copy of str with all leading whitespace removed
- std::[w]string& trimLeftInPlace(std::[w]string& str)
  removes all leading whitespace from str and returns a reference to it
- std::[w]string trimRight(const std::[w]string& str)
  removes a copy of str with all trailing whitespace removed
- std::[w]string trimRightInPlace(std::[w]string& str)
  removes all trailing whitespace from str and returns a reference to it

# Trimming (cont'd)

- std::[w]string trim(const std::[w]string& str)
  returns a copy of str with all leading and trailing whitespace removed
- std::[w]string trimInPlace(std::[w]string& str)
  returns all leading and trailing whitespace from str and returns a reference to it

```
#include <Poco/String.h>

using Poco::trim;
using Poco::trimLeft;
using Poco::trimRight;
using Poco::trimRightInPlace;


int main(int argc, char** argv)
{
    std::string hello("  Hello, world!  ");

    std::string s1(trimLeft(hello)); // "Hello, world!  "
    trimRightInPlace(s1);            // "Hello, world!"
    std::string s2(trim(hello));     // "Hello, world!"

    return 0;
}
```

# Case Conversion

- std::[w]string toUpper(const std::[w]string& str)
  std::[w]string toLower(const std::[w]string& str)
  returns a copy of str with all characters converted to upper-/lowercase.

- std::[w]string& toUpperInPlace(std::[w]string& str)
  std::[w]string& toLowerInPlace(std::[w]string& str)
  you get the idea...

- Warning: These function do not work with UTF-8 strings. See Poco::UTF8 for an UTF-8 capable replacement.

# Case-insensitive Comparison

- int icompare(const std::[w]string& str1, const std::[w]string& str2)
  compares str1 to str2, and returns
  - 0 if str1 == str2
  - -1 if str1 < str2
  - +1 if str1 > str2
- There are different variants taking substrings, iterators, C strings, etc. Please refer to the reference documentation for details.
- **Warning**: This does not work with UTF-8 strings.
  Use the Poco::UTF8String class for UTF-8 encoded strings.

```cpp
#include "Poco/String.h"

using Poco::toUpper;
using Poco::toLower;
using Poco::toLowerInPlace;
using Poco::icompare;

int main(int argc, char** argv)
{
	std::string hello("Hello, world!");

	std::string s1(toUpper(hello)); // "HELLO, WORLD!"
	toLowerInPlace(s1);             // "hello, world!"

	int rc = icompare(hello, s1);             // 0
	rc = icompare(hello, "Hello, Universe!"); // 1

	return 0;
}
```

# I18N and L10N

- Internally, the case conversion and case-insensitive comparison functions use C++ locales.
  - ASCII characters will always work
  - whether non-ASCII characters (e.g., "Umlauts") work, depends on your particular C++ library implementation (and your locale settings)
- For reliable multilingual case conversion (and collation, etc.), it's best to use a specialized third-party library, e.g. ICU (http://www.ibm.com/software/globalization/icu/).

# Character Translation

- std::[w]string translate(const std::[w]string& str,
  const std::[w]string& from, const std::[w]string& to)
  returns a copy of str with all characters in from replaced with the corresponding (by position) characters in to.
  If there is no corresponding character in to, the character is removed.
- std::[w]string& translateInPlace(std::[w]string& str,
  const std::[w]string& from, const std::[w]string& to)
- from and to can also be old-style C strings.

```
#include "Poco/String.h"

using Poco::translateInPlace;

int main(int argc, char** argv)
{
    std::string s("Eiffel Tower");

    translateInPlace(s, "Eelo", "3310"); // "3iff31 T0w3r"

    return 0;
}
```

# Substring Replacement

- std::[w]string replace(const std::[w]string& str,
  const std::[w]string& from, const std::[w]string& to)
  returns a copy of str with all occurences of the substring given in from replaced with the string given in to.
- std::[w]string& replaceInPlace(std::[w]string& str,
  const std::[w]string& from, const std::[w]string& to)
- from and to can also be old-style C strings.

```
#include "Poco/String.h"

using Poco::replace;

int main(int argc, char** argv)
{
    std::string s("aabbcc");

    std::string r(replace(s, "aa", "AA")); // "AAbbcc"
    r = replace(s, "bb", "xxx");           // "aaxxxcc"
    r = replace(s, "bbcc", "");            // "aa"

    return 0;
}
```

# String Concatenation

- std::[w]string cat(const std::[w]string& s1,
  const std::[w]string& s2 [, const std::[w]string& s3 [,...]])
  concatenates up to six strings and returns the result

- template <class S, class It>
  S cat(const S& delimiter, const It& begin, const It& end)
  concatenates all string in the range [It, end), delimited by delimiter

- cat() is more efficient than operator + of std::string

```cpp
#include "Poco/String.h"
#include <vector>

using Poco::cat;

int main(int argc, char** argv)
{
	std::vector<std::string> colors;
	colors.push_back("red");
	colors.push_back("green");
	colors.push_back("blue");

	std::string s;
	s = cat(std::string(", "), colors.begin(), colors.end());
	    // "red, green, blue"

	return 0;
}
```

# Formatting Numbers

- Poco::NumberFormatter provides static methods to format numbers into strings.
- #include "Poco/NumberFormatter.h"
- All format methods are available for int, unsigned int, long, unsigned long, Int64 and UInt64.
- Internally, the methods use std::sprintf().

# The NumberFormatter Class

- std::string format(int value)
  formats an integer value in decimal notation, using as little space as possible

- std::string format(int value, int width)
  formats an integer value in decimal notation, right justified in a field of at least width characters

- std::string format0(int value, int width)
  formats an integer value in decimal notation, right justified and zero-padded in a field of at least width characters

# The NumberFormatter Class (cont'd)

- std::string formatHex(int value)
  formats an integer value in hexadecimal notation, using as little space as possible

- std::string formatHex(int value, int width)
  formats an integer value in hexadecimal notation, right justified and zero-padded in a field of at least width characters

- std::string format(const void* ptr)
  formats the pointer ptr in an eight (32-bit pointer) or 16 (64-bit pointer) characters wide field, in hexadecimal notation

# The NumberFormatter Class (cont'd)

- std::string format(double value)
  formats a floating-point value in decimal floating-point notation, with a precision of eight fractional digits

- std::string format(double value, int precision)
  formats a floating-point value in decimal floating-point notation,with precision fractional digits

- std::string format(double value, int width, int precision)
  formats a floating-point value in decimal floating-point notation, right justified in a field of the specified width, with precision fractional digits

# The NumberFormatter Class (cont'd)

- All format() member functions have append() counterparts that append the number to an existing string.
- Use of append() can greatly improve performance.
- Example: void append(std::string& str, int value);
- format() is implemented using append().
- **WARNING**: The exact result of the conversion is dependent on the current C locale.

```
#include "Poco/NumberFormatter.h"

using Poco::NumberFormatter;

int main(int argc, char** argv)
{
	std::string s;
	s = NumberFormatter::format(123);		// "123"
	s = NumberFormatter::format(123, 5);	// "  123"
	s = NumberFormatter::format(-123, 5);	// " -123"
	s = NumberFormatter::format(12345, 3);	// "12345"
	s = NumberFormatter::format0(123, 5);	// "00123"

	s = NumberFormatter::formatHex(123);	// "7B"
	s = NumberFormatter::formatHex(123, 4); // "007B"

	s = NumberFormatter::format(1.5);		// "1.5"
	s = NumberFormatter::format(1.5, 2);	// "1.50"
	s = NumberFormatter::format(1.5, 5, 2); // " 1.50"

	s = NumberFormatter::format(&s);		// "00235F7D"

	return 0;
}
```

# Typesafe Printf-style Formatting

- POCO provides a formatting function similar to sprintf, but for std::string and typesafe.
- #include "Poco/Format.h"
- std::string format(const std::string& format,
  const Any& value1[, const Any& value2[, ...]])
- void format(std::string& result, const std::string& format,
  const Any& value1[, const Any& value2[, ...]])

# Typesafe Printf-style Formatting (cont'd)

- The format string is largely compatible with printf() and friends (but there are differences!)
  For details, please refer to the reference documentation.
- Up to six parameters are supported.
- A value that does not match the format specifier type results in output of [ERRFMT].
- If there are more format specifiers than values, the superfluous specifiers are copied verbatim to the result.
- If there are more values than format specifiers, the superfluous values are simply ignored.

```cpp
#include "Poco/Format.h"

using Poco::format;

int main(int argc, char** argv)
{
	int n = 42;
	std::string s;
	format(s, "The answer to life, the universe and everything is %d", n);

	s = format("%d + %d = %d", 2, 2, 4); // "2 + 2 = 4"
	s = format("%4d", 42);               // "  42"
	s = format("%-4d", 42);              // "42  "

	format(s, "%d", std::string("foo")); // "[ERRFMT]"

	return 0;
}
```

# Extracting Numbers From Strings

- The static member functions of the Poco::NumberParser class can be used to parse numbers from strings.
- #include "Poco/NumberParser.h"
- int parse(const std::string& str)
  Parses an integer value in decimal notation from str. Throws a SyntaxException if the string does not contain a valid number.
- bool tryParse(const std::string& str, int& value)
  Parses an integer value in decimal notation from str, and stores it in value. Returns true if the number is valid.
  Returns false if not; value is undefined in this case.

# Extracting Numbers From Strings (cont'd)

- unsigned parseUnsigned(const std::string& str)
- bool tryParseUnsigned(const std::string& str, unsigned& value)
- unsigned parseHex(const std::string& str)
- bool tryParseHex(const std::string& str, unsigned& value)
- Int64 parse64(const std::string& str)
- bool tryParse64(const std::string& str Int64& value)
- UInt64 parseUnsigned64(const std::string& str)
- bool tryParseUnsigned64(const std::string& str UInt64& value)

# Extracting Numbers From Strings (cont'd)

- UInt64 parseHex64(const std::string& str)
- bool tryParseHex64(const std::string& str UInt64& value)
- double parseFloat(const std::string& str)
- bool tryParseFloat(const std::string& str, double& value)

# Tokenizing Strings

- Poco::StringTokenizer can be used to split a string into tokens. Tokens have to be separated by separator characters.
- #include "Poco/StringTokenizer.h"
- The string to be tokenized, the separator characters, and options must be passed to the StringTokenizer constructor.
- The StringTokenizer internally holds a vector containing the extracted tokens.

# Tokenizing Strings (cont'd)

- StringTokenizer supports the following options:
  - TOK_IGNORE_EMPTY
    empty tokens are ignored
  - TOK_TRIM
    remove leading and trailing whitespace from tokens

```
#include "Poco/StringTokenizer.h"
#include "Poco/String.h" // for cat

using Poco::StringTokenizer;
using Poco::cat;

int main(int argc, char** argv)
{
    StringTokenizer t1("red, green, blue", ",");
        // "red", " green", " blue" (note the whitespace)
    StringTokenizer t2("red,green,,blue", ",");
        // "red", "green", "", "blue"
    StringTokenizer t3("red; green, blue", ",;",
        StringTokenizer::TOK_TRIM);
        // "red", "green", "blue"
    StringTokenizer t4("red; green,, blue", ",;",
        StringTokenizer::TOK_TRIM | StringTokenizer::TOK_IGNORE_EMPTY);
        // "red", "green", "blue"

    std::string s(cat(std::string("; "), t4.begin(), t4.end()));
        // "red; green; blue"

    return 0;
}
```

# Regular Expressions

- Support for regular expressions in POCO is available through the Poco::RegularExpression class.
- #include "Poco/RegularExpression.h"
- Internally, Poco::RegularExpression uses the PCRE library (Perl Compatible Regular Expressions).
- This means that regular expressions in POCO are largely compatible to those in Perl.

# The RegularExpression Class

- Poco::RegularExpression supports:
  - matching a string against a regular expression (match),
  - extracting substrings using regular expressions (extract and split),
  - replacing substrings using regular expressions (subst).
- Various options control the exact matching behavior. Please refer to the reference documentation for details.

# The RegularExpression Class (cont'd)

- Some useful options:
  - RE_CASELESS
    perform case-insensitive matching (/i option in Perl)
  - RE_ANCHORED
    treat pattern as if it starts with a ^
  - RE_NOTEMPTY
    the empty string never matches
  - RE_UTF8
    assume pattern and subject is UTF-8 encoded

```
#include "Poco/RegularExpression.h"
#include <vector>

using Poco::RegularExpression;

int main(int argc, char** argv)
{
    RegularExpression re1("[0-9]+");
    bool match = re1.match("123");  // true
    match = re1.match("abc");       // false
    match = re1.match("abc123", 3); // true

    RegularExpression::Match pos;
    re1.match("123", 0, pos);    // pos.offset == 0, pos.length == 3
    re1.match("ab12de", 0, pos);  // pos.offset == 2, pos.length == 2
    re1.match("abcd", 0, pos);   // pos.offset == std::string::npos

    RegularExpression re2("([0-9]+) ([0-9]+)");
    RegularExpression::MatchVec posVec;
    re2.match("123 456", 0, posVec);
        // posVec[0].offset == 0, posVec[0].length == 7
        // posVec[1].offset == 0, posVec[1].length == 3
        // posVec[2].offset == 4, posVec[2].length == 3
```

```
std::string s;
int n = re1.extract("123", s);   // n == 1, s == "123"
n = re1.extract("ab12de", 0, s); // n == 1, s == "12"
n = re1.extract("abcd", 0, s);   // n == 0, s == ""

std::vector<std::string> vec;
re2.split("123 456", 0, vec);
    // vec[0] == "123"
    // vec[1] == "456"

s = "123";
re1.subst(s, "ABC");   // s == "ABC"

    s = "123 456";
    re2.subst(s, "$2 $1"); // s == "456 123"

    RegularExpression re3("ABC");
    RegularExpression re4("ABC", RegularExpression::RE_CASELESS);
    match = re3.match("abc", 0); // false
    match = re4.match("abc", 0); // true

    return 0;
}
```

# Text Encodings

- POCO provides some support for using different character encodings with std::string and I/O streams.
- Strings and characters written to a stream can be converted between different encodings.
- A special iterator class can be used to iterate over all characters in a multbyte-encoded string.
- The following encodings are currently supported:
  ASCII, Latin-1, Latin-9, Windows-1252, UTF-8 and UTF-16.
- The recommended encoding with POCO is UTF-8.

# The TextConverter Class

- Poco::TextConverter converts strings between different character encodings.
- #include "Poco/TextConverter.h"
- The source and target encodings are passed to the constructor.
- The convert method performs the conversion.

```cpp
#include "Poco/TextConverter.h"
#include "Poco/Latin1Encoding.h"
#include "Poco/UTF8Encoding.h"
#include <iostream>

using Poco::TextConverter;
using Poco::Latin1Encoding;
using Poco::UTF8Encoding;

int main(int argc, char** argv)
{
    std::string latin1String("This is Latin-1 encoded text.");
    std::string utf8String;

    Latin1Encoding latin1;
    UTF8Encoding utf8;

    TextConverter converter(latin1, utf8);
    converter.convert(latin1String, utf8String);

    std::cout << utf8String << std::endl;
    return 0;
}
```

# The StreamConverter Classes

- Poco::OutputStreamConverter acts as a filter that converts all characters written to it to another encoding, before passing them on to another stream.
- Poco::InputStreamConverter acts as a filter that reads characters from another stream and converts them to another encoding, before passing them to the reader.

```
#include "Poco/StreamConverter.h"
```

```
#include "Poco/StreamConverter.h"
#include "Poco/Latin1Encoding.h"
#include "Poco/UTF8Encoding.h"
#include <iostream>

using Poco::OutputStreamConverter;
using Poco::Latin1Encoding;
using Poco::UTF8Encoding;

int main(int argc, char** argv)
{
    std::string latin1String("This is Latin-1 encoded text.");

    Latin1Encoding latin1;
    UTF8Encoding utf8;

    OutputStreamConverter converter(std::cout, latin1, utf8);
    converter << latin1String << std::endl; // console output will be
    UTF-8

    return 0;
}
```

# The TextIterator Class

- Poco::TextIterator is used to iterate over the Unicode characters in a std::string.
- #include "Poco/TextIterator.h"
- The string can be encoded in any of the supported encodings. Typically, this will be a multibyte encoding like UTF-8.

```
#include "Poco/TextIterator.h"
#include "Poco/UTF8Encoding.h"

using Poco::TextIterator;
using Poco::UTF8Encoding;

int main(int argc, char** argv)
{
    std::string utf8String("This is UTF-8 encoded text.");

    UTF8Encoding utf8;

    TextIterator it(utf8String, utf8);
    TextIterator end(utf8String);

    for (; it != end; ++it)
    {
        int unicode = *it;
    }

    return 0;
}
```

# POCO and Unicode

- Modern Unix systems (including Linux) support UTF-8 for console I/O and the filesystem.
- UTF-8 can be stored in plain C strings and std::string.
- On Windows, POCO can be built with Unicode support.
  This is the default since 1.3.0.
- For this, the preprocessor macro POCO_WIN32_UTF8 must be defined when building all POCO libraries and client code.
- POCO then calls the Unicode variants of the Windows API functions and converts strings between UTF-8 and UTF-16.

# The Unicode Class

- The Poco::Unicode class provides basic support for Unicode character properties (character category, character type, script).
- void properties(int ch, CharacterProperties& props)
  Returns the properties for the Unicode character given in ch.
- bool isLower(int ch)
  bool isUpper(int ch)
  Check for lower case/upper case character.
- int toLower(int ch)
  int toUpper(int ch)
  Case conversion.

# The UTF8 Class

- The Poco::UTF8 class (#include "Poco/UTF8String.h") provides implementations of icompare, toUpper() and toLower() that work with UTF-8 encoded strings.
- Use the static member functions of Poco::UTF8 instead of the freestanding functions if you know that your strings are UTF-8 encoded.

# applied informatics



www.appinf.com | info@appinf.com
T +43 4253 32596 | F +43 4253 32096