Filesystem V3

And what it can teach us about Unicode string classes

Beman Dawes

May 7, 2009

With Corrections

Overview

- Filesystem V3 Motivation
- V3 `class path`
- Lessons Learned
- How to apply to Unicode string design
- Use cases
- Unicode string classes with fixed encodings
- Unicode string class with adaptive encoding
- Interoperability
- Lessons Learned Review

Motivation: V2 Problem

```
class logger
{
public:
  logger(const boost::filesystem::path & p);
  ...
};
...
logger english_log("english");
logger japanese_log(L"日本語");
...
```

Motivation: V2 Problem

```
class logger
{
public:
  logger(const boost::filesystem::path & p);
  ...
};
...
logger english_log("english");
logger japanese_log(L"日本語"); // error!
...
```

V2 class path

```
template <class String, class Traits>
class basic_path
{
public:

  typedef String string_type;
  typedef typename String::value_type value_type;

  basic_path( const string_type & s );
  basic_path( const value_type * s );
...
};

typedef basic_path< std::string, path_traits > path;
typedef basic_path< std::wstring, wpath_traits > wpath;
```

How can we fix filesystem so that:

```
logger japanese_log(L"日本語");
```

will work?

Why am I not asking how to change class logger?

V3 class path

```
class path {
public:
  typedef std::string/wstring string_type;  // POSIX/Windows

  template <class String>
    path( const & String s )
      { path_traits::append( s, m_path ); }

  template <class CharT>
    path( const * CharT s )
      { path_traits::append( s, m_path ); }

  ...

  template <class String>
    String string() const
      { return path_traits::convert<String>( m_path ); }

private:
  string_type m_path;
};
```

V3 Motivation: V3 Works!

```
class logger
{
public:
  logger(const boost::filesystem::path & p);
  ...
};
...
logger english_log("english");
logger japanese_log(L"日本語"); // OK!
...
```

What have we learned?

Lessons Learned

- A single class type simplifies use; users don't have to provide overloads for every possible string type or combinations of types.
- The class member function templates provide:
  - Generic, compile-time polymorphic flexibility.
  - Efficient operation including inline functions and no virtual functions.
  - Interoperability with many types, including user-defined types.

All we need to know about Unicode encodings, in one slide

- There are multiple possible encodings of a Unicode “character”. (i.e. codepoint)
- Utf-8 uses one to four 8-bit characters to encode a Unicode “character”.
- Utf-16 uses one or two 16-bit characters to encode a Unicode “character”.
- Utf-32 uses one 32-bit character to encode a Unicode “character”.
- There are some other encodings that we don’t care about.

Aside

- The heart of a C++ Unicode library will be its algorithms and iterators.
- Unicode string classes will achieve effects and results by calling those algorithms and using those iterators.
- But... today we are talking about Unicode strings, so won't talk further about Unicode algorithms and iterators.

Scenario 1

- Best internal encoding is known at compile time. Use cases:
  - Mostly Western Characters, memory is concern, efficient random access not a concern, so UTF-8 encoding desired.
  - Mostly Asian characters or random access critical, so UTF-16 encoding desired.

Suggested solution to the scenario 1 use cases?

Scenario 1

- Best internal encoding is known at compile time. Use cases:
  - Mostly Western Characters, memory is concern, efficient random access not a concern, so UTF-8 encoding desired.
  - Mostly Asian characters or random access critical, so UTF-16 encoding desired
- Solution: Provided string types utf8_string, utf16_string, and utf32_string with fixed internal encoding.

utf8_string

```
class utf8_string {

private:
  std::string m_string;  // utf-8 encoding
};
```

utf8_string

```
class utf8_string {
public:

  template <class String>
    utf8_string( const & String s )
      { utf_traits::append( s, m_string ); }

  template <class CharT>
    utf_8( const * CharT s )
      { utf_traits::append( s, m_string ); }

private:
  std::string m_string;  // utf-8 encoding
};
```

utf8_string

```
class utf8_string {
public:

  template <class String>
    utf8_string( const & String s )
      { utf_traits::append( s, m_string ); }

  template <class CharT>
    utf_8( const * CharT s )
      { utf_traits::append( s, m_string ); }

  template <class String>
    String string() const
      { return path_traits::convert<String>( m_string ); }

private:
  std::string m_string;  // utf-8 encoding
};
```

utf8_string

```
class utf8_string {
public:

  template <class String>
    utf8_string( const & String s )
      { utf_traits::append( s, m_string ); }

  template <class CharT>
    utf8_string( const * CharT s )
      { utf_traits::append( s, m_string ); }

  template <class String>
    String string() const
      { return path_traits::convert<String>( m_string ); }

  std::size_t size() const { return m_string.size(); }

private:
  std::string m_string;  // utf-8 encoding
};
```

Scenario 2

- Best internal encoding is unknown at compile time:
  - Use of the same string shifts back and forth at runtime between mostly Western and mostly Asian character sets.
  - Memory is sometime important, sometimes not. Other criteria also vary at runtime.
- Solution: utf_string class, with encoding that adapts to the desired encoding.

utf_string

```
class utf_string {
public:

  template <class String>
    utf_string( const & String s )
      { utf_traits::append( s, m_string ); }

  template <class CharT>
    utf_string( const * CharT s )
      { utf_traits::append( s, m_string ); }

  template <class String>
    String string() const
      { return path_traits::convert<String>( m_string ); }

  std::size_t size() const { return m_string.size(); }
  std::size_t length() const { return m_length; }

private:
  implementation-defined m_string;  // adaptive encoding
};
```

utf_string

- What is m_string's type likely to look like? *Some form of discriminated union.*
- What is `iterator` value_type's encoding? *UTF-32 a strong candidate.*
- What concept does `iterator` model? *Not clear.*
- What is the return type of operator[]? *Wrong question. Since operator[] is unsafe, should it exist at all? Should it have a different name?*

Commingled Use Cases

- Compile-time and runtime strings will sometimes be needed in the same application.
- Thus compile-time strings utf8_string, etc. will have to interoperate not only with each other but also with the runtime string, utf_string.
- And, like class path, they all must interoperate with std::basic_string, C-arrays, and user-defined string classes.

Commingled Uses Just Work!

- With the proposed design, there is little that has to be done for interoperability between utf8_string , utf16_string, utf32_string, and utf_string to just work.
- Just provide the necessary traits for conversions.

Lessons Learned

- A single class type simplifies use; users don’t have to provide overloads for every possible string type or combinations of types.
- The class member function templates provide:
  - Generic, compile-time polymorphic flexibility.
  - Efficient operation including inline functions and no virtual functions.
  - Interoperability with many types, including user-defined types.