LocaleLookup

Problem Statement

Modern operating systems provide support for dozens or even hundreds locales encoded in various codesets. The set of locales and codesets installed on a computer is typically determined by the system administrator at the time the operating system is installed. Although there are standards and conventions in place to establish a common set of locale names, due to historical reasons both locale and codeset names tend to vary from one implementation to another. Operating systems may provide the standard names as well as the traditional ones, with the former simply being aliases for the latter.

The stdcxx test suite contains tests that exercise the behavior of the localization library. Since the set of installed locales may vary from server to server and since their names need not be consistent across different operating systems, the stdcxx test driver provides mechanisms to determine the names of all locales known to a system. For simplicity, many tests exercise the localization library using all these locale names. Other tests do so in an effort to exercise different code paths taken based on whether a locale uses a single-byte or multi-byte encoding. On systems with many installed locales running these tests may take a considerable amount of time and use up valuable system resources. For example, on AIX systems with all available locales installed running each test can take as much as an hour. In addition, since many of the locale names reference the same locale exercising all of them is wasteful. In addition, since many locales differ only in very minor details (e.g., the values of punctuator characters), exhaustively testing all of them ends up repeatedly executing the same code paths and is unnecessary.

Objective

The objective of this project is to provide an interface to make it easy to write localization tests without the knowledge of platform-specific details (such as locale names) that provide sufficient code coverage and that complete in a reasonable amount of time (ideally seconds as opposed to minutes). The interface must make it easy to query the system for locales that satisfy the specific requirements of each test. For example, most tests that currently use all installed locales (e.g., the set of tests for the `std::ctype` facet) only need to exercise a representative sample of the installed locales without using the same locale more than once. Thus the interface will need to make it possible to specify such a sample. Another example is tests that attempt to exercise locales in multibyte encodings whose `MB_CUR_MAX` ranges from 1 to 6 (some of the `std::codecvt` facet tests). The new interface will need to make it easy to specify such a set of locales without explicitly naming them, and it will need to retrieve such locales without returning duplicates.

Use Cases

The existing locale tests select locales based on a few different criteria. Below is a list of locales tests and the criteria used for locale selection within those tests.

Test	Criteria
22.locale.codecvt.mt.cpp	*1,+
22.locale.codecvt.out.cpp	*10
22.locale.cons.mt.cpp	*1,+
22.locale.ctype.cpp	*2
22.locale.ctype.is.cpp	*2
22.locale.ctype.mt.cpp	*1,+
22.locale.ctype.narrow.cpp	*2
22.locale.ctype.scan.cpp	*2
22.locale.ctype.tolower.cpp	*2
22.locale.ctype.toupper.cpp	*2
22.locale.globals.mt.cpp	*8,+
22.locale.messages.cpp	*7
22.locale.money.get.mt.cpp	*1,+
22.locale.money.put.mt.cpp	*1,+
22.locale.moneypunct.cpp	*4
22.locale.moneypunct.mt.cpp	*1,+
22.locale.num.get.cpp	*9
22.locale.num.get.mt.cpp	*1,+
22.locale.num.put.cpp	*9
22.locale.num.put.mt.cpp	*1,+

22.locale.numpunct.mt.cpp	*1,+
22.locale.statics.mt.cpp	*4,+
22.locale.time.get.cpp	*5,6
22.locale.time.get.mt.cpp	*1,+
22.locale.time.put.mt.cpp	*1,+

1. Any locale for which setlocale (LC_ALL, name) will succeed.
2. Any locale for which setlocale (LC_CTYPE, name) will succeed.
3. Any locale for which setlocale (LC_NUMERIC, name) will succeed.
4. All installed locales.
5. First locale matching a specific name.
6. First locale matching a regular expression.
7. First locale that is not an alias for the C/POSIX locale.
8. Any locale for which setlocale (LC_ALL, name) will succeed, list includes C/POSIX locale.
9. Any locale for which setlocale (LC_NUMERIC, name) will succeed and decimal_point is not '.'
10. Locale with largest MB_CUR_LEN value.

+ Test limits the number of locales tested.

Note: Most of the MT tests limit the number of locales to 32, so the test failure is not a matter of running against to many locales, it is an issue of running to many iterations per thread. The 'solution' discussed in this document doesn't seem to address the actual problem for these tests.
Note: Most of the tests simply run against all locales that have a specified category. We need to decide how to further reduce the number of locales tested.

Definitions

canonical language: The `<language>` field is two lowercase characters that represent the language as defined by ISO-639.

canonical country: The `<COUNTRY>` field is two uppercase letters that represent the country as defined by ISO-3166.

canonical codeset: The `<CODESET>` field is a string describing the encoding character set. For our purposes, the codeset is the preferred MIME name of the codeset as defined by IANA.

Plan

This page relates to the issue described in STDCXX-608. There has been some discussion both on and off the dev@ list about how to proceed. This page is here to document what has been discussed.

The plan to meet the Objective is to provide an interface to query the set of installed locales based on a set of a small number of essential parameters used by the localization tests. The interface should make it easy to express conjunction, disjunction, and negation of the terms (parameters) and support (a perhaps simplified version of) Basic Regular Expression syntax. We've decided to use shell brace expansion as a means of expressing logical conjunction between terms: a valid brace expression is expanded to obtain a set of terms implicitly connected by a logical AND. Individual ('\n'-separated) lines of the query string are taken to be implicitly connected by a logical OR. This approach models the grep interface with each line loosely corresponding to the argument of the `-e` option to `grep`.

Part 1 (STDCXX-714)

The first thing that we needed was to write the function for doing Basic Regular Expression name matching and add it to the test suite.. Martin has already added an implementation of rw_fnmatch(), so that is done. `rw_fnmatch()` is a simplified implementation of the POSIX fnmatch() function which supports a simplified and modified form of BRE used in filename globbing. This is sufficient for what we need in term of regular expression support.

The second thing that we needed was a function to do brace expansion. After much discussion, it was decided that the csh brace expansion rules made the most sense. Travis provided an implementation of a function for doing brace expansion. The function rw_shell_expand() does whitespace tokenization and collapse, and then does brace expansion on each token, much like the behavior you would see from the csh shell.

Just for illustration, consider the following string.

```
a-{1,2}-b
```

If you passed this to `rw_shell_expand()` (with ' ' as the seperator), the result would be

```
a-1-b a-2-b
```

Part 2 (STDCXX-715)

Every platform has a unique list of locales available. For example, Windows sytems use `English` as a language name, but most *nix systems the canonical `en` or in some cases `EN`. This problem exists for all fields of the locale name.

To deal with this, we need to provide a mapping between the native names and the canonical names that we plan to support in the query string. The plan is to provide these mappings in data files. We would need at least three different mappings, one each for language, country and codeset. We would need one additional mapping if we wanted to map from a canonical language code to a default country code. This would be necessary so that we can map locale names like `russian` or `ru` to an appropriate territory code.

The format of these files is simple. Here is a grammar

```
native-name-list ::= <native-name> | <native-name> ',' <native-name-list> | '\n' <ws> <native-name-list>
line         ::= '#' <comment> | <canonical-name> <native-name-list>
line-list    ::= <line> | <line> '\n' <line-list>
```

The grammar is comma delimited, so the strings are not to be quoted. Here is an example to illustrate.

```
# this is a comment line

 # _not_ a comment line
# the above maps '_not_ a comment line' to the value '#'

# map 'English' to 'en'
en        English

# map 'Albanian', 'alb' and 'sqi' to 'sq'
sq    Albanian, alb, sqi

# similar to above, except that mapping is multiline
cu    Church Slavic, Old Slavonic, Church Slavonic,
      Old Bulgarian, Old Church Slavonic, chu
```

Part 3 (STDCXX-716)

The proposed interface to all of this is a single public function named rw_query_locales(). The signature would be...

```
char* rw_query_locales (int loc_cat, const char* query, size_t count);
```

The `loc_cat` parameter is the locale category to get locales for, just like `rw_locales()` does in its current implementation. The `query` parameter will be the query string. The `count` parameter is the maximum number of locales to return. This allows you to easily limit the number of locales returned and eventually tested.

The proposed grammar used by the query string is similar to what is used for the xfail.txt `config` string. It is a shell globbed string that has its terms joined with dashes.

```
<match> is a shell globbing pattern in the format below. All fields are required.

iso-country  ::= ISO-639-1 or ISO-639-2 two or three character country code
iso-language ::= ISO-3166 two character language code
iana-codeset ::= IANA codeset name

match        ::= <iso-language-expr> '-' <iso-country-expr> '-' <mb_cur_len-expr> '-' <iana-codeset-expr>
match_list   ::= match | match ' ' match_list
```

So, given a query string

```
*-{CA,US}-1-{ISO-8859-1,UTF-8}
```

this function would internally apply brace expansion to get the following list of expressions

```
*-CA-1-*-ISO-8859-1 *-CA-1-*-UTF-8 *-US-1-*-ISO-8859-1 *-US-1-*-UTF-8
```

⚠️ Notice that I have moved the codeset to be the last match in the query string. That is because the codeset string is allowed to contain dashes. This was done to avoid issues with accidentally mistaking dashes in the codeset name with dashes in the grammar.

After doing the brace expansion, this function will get a list of installed locales and their canonical representation strings. Then, for each of the brace expanded expressions, the function will search for locales whose canonical representation matches the expression. If the name is a match, the native locale name will be appended to a buffer that will be returned to the user. Logic will exist to prevent the same locale from being accepted for more than one matching expression.

⚠️ Perhaps we should consider adding an additional parameter to prepend the C/POSIX locales as there is no way to match them using the canonical locale name matching rules we've laid out above.

The buffer returned by `rw_locale_query()` is owned by that function and is not to be dallocated by the user. This buffer is currently planned to be left in use at program termination. If it is deemed necessary, some additional code can be written to cleanup the buffer before program exit, or we could require the user to deallocate the buffer when they are done with it.

Ideas

I'm wondering why we didn't decide to use a callback system for this. It would allow us to use arbitrary criteria to test a locale. The interface wouldn't always be 'grep-like', but it would be very extensible. Something like this...

```
_TEST_EXPORT const char*
rw_locale_language (const char*);

_TEST_EXPORT const char*
rw_locale_territory (const char*);

_TEST_EXPORT const char*
rw_locale_codeset (const char*);

_TEST_EXPORT void
rw_locale_test (bool (*fun)(const char*, void*), void*);
```

The function `rw_locale_test()` would get a list of all installed locales, then pass the name of those locales and the context pointer p to `fun`. The user function could do whatever it wanted to decide if the locale is acceptable.

This would make it quite simple to select only locales with a specific attribute. For example if we only wanted to select a locale with the largest MB_CUR_LEN value...

```
struct _locale_mb_context
{
  char name [128];
  int cur_len;
};

static bool
_rw_locale_mb_fun (const char* name, void* p)
{
  const char* loc = setlocale (LC_CTYPE, name);
  if (!loc)
  {
    _locale_mb_context* context =
        (_locale_mb_context*)p;

    const int cur_len = MB_CUR_LEN;
    if (context->cur_len < cur_len)
    {
      strcpy (context->name, loc);
      context->cur_len = cur_len;
    }
  }

  return false;
}

static const char*
test_big_mb_locale ()
{
  locale_mb_context ctxt;
  rw_locale_test (_rw_locale_mb_fun, &ctxt);

  // run the test on locale named by ctxt.name
}
```

Or, to get a list of all locales that match brace expansion

```
static bool
_rw_locale_match (const char* name, void* p)
{
  _locale_match_context* context =
    (_locale_match_context*)p;

  const char* language = rw_locale_language (name);
  const char* country  = rw_locale_territory (name);
  const char* codeset  = rw_locale_codeset (name);

  char buf [128];
  sprintf (buf, "%s-%s-%s", language, country, codeset);

  for (const char* s = context->expr;
       *s; s += strlen (s) + 1)
  {
    if (rw_fnmatch (s, name))
    {
      // run the test on locale named by name
    }
  }

  return false;
}

static void
test_all_matches (const char* expr)
{
  char buf [256];

  char* res = rw_shell_expand (expr, 0, buf, sizeof (buf));

  _rw_locale_test (_rw_locale_match, res);

  if (res != buf)
    free (res);
}
```

References

discussion http://www.nabble.com/low-hanging-fruit-while-cleaning-up-test-failures-to13634803.html#a15137334

std-country (ISO-3166) http://www.iso.org/iso/english_country_names_and_code_elements

std-lang (ISO-639) http://www.loc.gov/standards/iso639-2/php/English_list.php

std-codeset http://www.iana.org/assignments/character-sets

AIX 6.1 National Language Support Guide and Reference

HP-UX 11.0 - 11i Internationalization Features White Paper

Locales and Internationalization in The GNU C library

The GNU libc locales mailing list archives

Solaris Locale FAQ

Win32 Locale Identifier Constants and Strings