Potext Developer Guide 0.1

Chris Ahlstrom
(`ahlstromcj@gmail.com`)

March 22, 2024

πωτεχ

Pseudo-Greek Transliteration

Contents

List of Figures

1 Introduction

The *potext* libraries adopt, refactor, and greatly extend the *tinygettext* library ([3]). The purpose of that library is to provide a lightweight mechanism to do translations using the *Portable Object* translation files, `*.po`, directly, rather than *Machine Object* files, `*.mo`.

The purpose of the *Potext* library is to provide `C++` functions that mirror the many functions of *gettext*, including `textdomain()`, `bindtextdomain()`, `bind_textdomain_codeset()`, `gettext()`, `dgettext()`, `ngettext()`, and others. In addition, some of the details of *tinygettext* are wrapped so that, other than marking the text to be translated, the translation setup is done by calling a single function in `main()`.

Our main goal is to make it easy and lightweight to internationalize an application while sticking with *GNU Gettext* conventions.

The *GNU Gettext* manual ([2]) is an important resource used in writing *Potext*. Also important is the source code at `savannah.gnu.org` ([1]).

Note that *Potext* requires the usage of `C++17` and above. It support builds using the *GNU* and *Clang* compilers. *Windows* support will be provided, but it is not ready yet.

1.1 Additions to Tinygettext

- Re-implementations of *gettext/libintl* functions as a module (collection of functions) in the `po` namespace.
- Integration of the functions above with the dictionary-manager class.
- Support for selecting a domain during the run.
- A new class, `nlsbindings`, to supplement the `language` class.
- An additional test program, test `.po` files, and upgrades to existing tests.
- Full documentation of architecture and usage.
- Meson `.wrap` files to use Meson as a subproject. A sample application project is stored in the tar-file noted below.

With these additions, *Potext* should be relatively straightforward to use in a new `C++` application.

`extras/code/mini-potext-test.tar.xz` contains a simple application. Unpack the tar-file into its own directory and build it by running the `work.sh` script.

1.2 Deletions from GNU Gettext

- Locking has been removed from some `gettext()`-like functions; locking can be put back once testing reveals it necessary for the use-cases that *Potext* supports.
- Determining if the binary is running SUID root.

Currently *Potext* is meant to be set up once during the run of an application. It currently does not detect changes to the environment variables related to localization and character conversions. It assumes that once the setup is made, no localization changes will be made. Keeps it simple.

1.3 Code Changes

- Changed the coding style and naming conventions.
- Use of initializer lists for initializing various containers.
- Use of the `auto` keyword in declarations and `for`-loops.
- Many additional `using` directives.

These changes are meant to make the code easier to read and understand.

1.4 Future Work

- Hammer on this code in *Windows*.
- Work out the installation process; including `.po` file installation and copying to the user's configuration directory.
- Handle capitals, punctuation, etc. without additional `.po` entries.
- Add parsing of `*.mo` files for more complete compatibility with *GNU Gettext*.
- Get the handling of categories (e.g. `LC_TIME`) to work. However, note that the category is almost always `LC_MESSAGES`.
- Handle the "C" locale as discussed below.
- Allow for the user to override the character set via the `OUTPUT_CHARSET` environment variable.
- Add support to assist a *Potext*-using project with the installation of the `.po` files.

Ignore `LANGUAGE` and its system-dependent analog if the locale is set to "C" because:

1. "C" locale uses the ASCII encoding; most international messages use non-ASCII characters, which get displayed as question marks or as invalid 8-bit characters.
2. The precise output of some programs in the "C" locale is specified by POSIX and should not depend on environment variables like `LANGUAGE`. Such programs can use `gettext()`.

Also ignore `LANGUAGE` and its system-dependent analog if the locale is `C.UTF-8` or `C.<encoding>`; that's the by-design behaviour for `glibc`, see `https://sourceware.org/glibc/wiki/Proposals/C.UTF-8`. Also look in `/usr/lib/locale/C.utf8`.

1.5 Naming Conventions

Potext uses some conventions for naming things in this document.

- `$prefix`. The base location for installation of the application and its ancillary data files on *UNIX/Linux/BSD*:
  - `/usr/`
  - `/usr/local/`
- `$winprefix`. The base location for installation of the application and its ancillary data files on *Windows*.
  - `C:/Program Files/`
  - `C:/Program Files (x86)/`
- `$podir`. The installed location of the `*.po` files. The directory `share` (*Linux*) or `data` (*Windows*), the package-name of the application (`PACKAGE`), and `po` are concatenated, and again the conventions differ between operating systems.
  - `/usr/share/PACKAGE/po/`
  - `/usr/local/share/PACKAGE/po/`
  - `C:/Program Files/PACKAGE/data/po/`
  - `C:/Program Files (x86)/PACKAGE/data/po/`
- `$home`. The alternate installed location of the `*.po` files. Not to be confused with `$HOME`, this is the standard location for configuration files. On a UNIX-style system, it would be `$HOME/.config/appname`. The files would be put into a `po` subdirectory here.
- `$localedir`. The installed location of the `*.mo` files. The directory `share` (*Linux*) or `data` (*Windows*), the package-name of the application (`PACKAGE`), and `locale` are concatenated. The conventions for *Linux* versus *Windows* differ as a matter of historical interest:
  - `/usr/share/PACKAGE/locale/`
  - `/usr/local/share/PACKAGE/locale/`
  - `C:/Program Files/PACKAGE/data/locale/`
  - `C:/Program Files (x86)/PACKAGE/data/locale/`

  At present, *Potext* does not support directories of `.mo` files. It might, in the future.
- `LC_MESSAGES`. A more common convention for `*.mo` files on *UNIX* is to put them in
  - `/usr/share/locale/<language>/LC_MESSAGES/PACKAGE.mo`
  - `/usr/local/share/locale/<language>/LC_MESSAGES/PACKAGE.mo`.

Currently, the *Potext* library uses only the `*.po` files. In the future it might also handle `*.mo` files. Also note that various applications differ in the exact location of their translation files.

1.6 Home Potext Configuration

The *Potext* library also supports installing the `*.po` translation files in the user's configuration area. The conventions we use are:

- `$home`. The location where `PACKAGE` installs, creates, or copies its configuration files. Do not confuse it with `$HOME`, although `$home` is in `$HOME/.config/PACKAGE`. The `*.po` files are stored in `$HOME/.config/PACKAGE/po`.
- `$winhome`. This location is different for *Windows*: `C:/Users/user/AppData/Local/PACKAGE`. Again, the `*.po` files are in a subdirectory called `po`.

Also, for reference, we mention some of the files used by *GNU Gettext*.

- `PACKAGE.pot`, created by `xgettext`.
- `LANG.po`, created by `msgmerge`, copying `PACKAGE.pot`, or by editing.
- `LANG.gmo`, created by `msgfmt`.
- For installed packages, see `$prefix/locale/LANG/PACKAGE.mo`.
- Or see `$prefix/locale-langpack`. `LC_category` (e.g. `LC_NUMERIC`).
- `LANG/PACKAGE.po`, reverse engineered from `PACKAGE.mo` by `msgunfmt`.

Also refer to the *Python* packages *polib* and *poedit*.

2 Potext Usage in Applications

This section briefly covers the usage of *Potext* in an application. A real sample is included in `library/tests/hellopotext.cpp` (see section 5.1 "Hello Potext" on page 22). A small sample application showing the usage of *Potext* as a *Meson* subproject is contained in:

```
extras/code/mini-potext-test.tar.xz
```

Unpack this file in its own directory and check it out.

2.1 Main Module Using Potext

The first thing is to add the following header file to the module defining the `main()` function.

```
#include "po/potext.hpp"      // includes "po/gettext.hpp"
```

For clarity, `potext.hpp` is better, but it does include an extra header file.

If *Potext* support is optional for the project, then do something like this; `PROJECT_USE_POTEXT` is a macro optionally defined when configuring the project build.

```
#if defined PROJECT_USE_POTEXT
#include "po/potext.hpp"   // includes "po/gettext.hpp"
else
#define _(str)             (str)
#define N_(str)            str
#endif
```

An application using "gettext" internationaliztion generally needs to call `setlocale()`, `textdomain()`, and `bindtextdomain()`. The following function wraps up these functions in one call.

```
std::string init_app_locale
(
    const std::string & arg0,
    const std::string & pkgname,
    const std::string & domainname,
    const std::string & dirname,
    int category
)
```

`arg0`. The path-name by which the program was called. This information can determine more precisely where installed `.po` files might be stored.

`pkgname`. The name of the PACKAGE, which can be the short name for the program, such as "helloworld".

`domainname`. The base name of a message catalog, such as "en_US". It must consist of characters legal in filenames. An application might want to use its package name, such as "helloworld", for the domain name. If empty, then the environment variable `TEXTDOMAIN` is used. If that's empty, then `LC_ALL` is used. If that's empty, then `LC_MESSAGE` is used. Lastly, if that's empty, then `LANG` is used.

`dirname`. Provides the name of the `LOCALEDIR`. The standard search directory is `/user/share/locale`. If empty, then the environment variable `TEXTDOMAINDIR` is used. If the name is "user", then the `.po` files are searched for in `/home/user/.config/package/` or `C:/Users/user/AppData/Local`, instead of some place in the system.

`category`. The area that is covered, such as `LC_ALL`, `LC_MONETARY`, and `LC_NUMERIC`. The default value selects `LC_ALL`.

The following calls are made for the setup:

1. `std::setlocale()` sets the application's current category and locale. The category is `LC_ALL` by default, and the locale is empty, so that the locale parts are modified according to environment variables.
2. `po::set_locale_info()` sets up the domain name and the locale directory name. If empty, the environment variables discussed above are used. In addition, it is determined if the locale directory is a system directory, the user's configuration directory, or some arbitrary directory containing `.po` files.
3. `po::init_lib_locale()` first asks the dictionary-manager (see section 3 "Potext Architecture" on page 10) to add all of the dictionaries (po files) in that directory to the list of selectable dictionaries, making one of them the default dictionary. Then the reimplementation of the `bindtextdomain()` is called to create a new domain-to-directory binding, and it is inserted into a container. This container supports looking up the locale directory associated with a domain.
4. `po::textdomain()` This function sets the current domain for the dictionary manager to use.

2.2 Marking a Module for Translation

The basic usage to *Potext* is essentially identical to that of *GNU Gettext*, except that (currently) only `.po` files are used directly.

Add the following header file.

```
#include "po/potext.hpp"      // includes "po/gettext.hpp"
```

Mark each translatable string as usual, using the `_()` macro:

```
std::string errmsg = _("Unknown system error");
```

That macro hides a call to `po::gettext()`. Additional "get-text" functions are listed in the table in the following section: section 4.2 "Gettext Module" on page 19.

2.3 Creating the .po Files

After marking the files that will provide translations, they must be processed to extract the marked strings for translation. For example:

```
$ xgettext test_helpers.cpp --keyword="_" --output="es.po"
```

The result is an untranslated template, `es.po`.

```
      # SOME DESCRIPTIVE TITLE.
      # Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
      # This file is distributed under the same license as the PACKAGE package.
      # FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
      #
      #, fuzzy
      msgid ""
      msgstr ""
      "Project-Id-Version: PACKAGE VERSION\n"
      "Report-Msgid-Bugs-To: \n"
      "POT-Creation-Date: 2024-03-20 06:53-0400\n"
      "PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
      "Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
      "Language-Team: LANGUAGE <LL@li.org>\n"
      "Language: \n"
      "MIME-Version: 1.0\n"
      "Content-Type: text/plain; charset=CHARSET\n"
      "Content-Transfer-Encoding: 8bit\n"

#: test_helpers.cpp:79
      msgid "output"
      msgstr ""
       . . .
```

The next step is to edit this file appropriately, as in the following snippet:

```
# Mensajes en español para test_helpers.
# Copyright (C) 2024 Potext Software Foundation Inc.
# This file is distributed under the same license as the test_helpers package.
# Chris Ahlstrom <ahlstromcj@gmail.com>, 2024.
msgid ""
msgstr ""
"Project-Id-Version: Potext test_helpers 0.1.0\n"
"Report-Msgid-Bugs-To: ahlstromcj@gmail.com\n"
"POT-Creation-Date: 2023-09-18 22:55+0200\n"
"PO-Revision-Date: 2024-03-20 17:08+0200\n"
"Last-Translator: Google Translate <translate.google.com>\n"
"Language-Team: Spanish <es@tp.org.es>\n"
"Language: es\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8-bit\n"
"X-Bugs: Report translation errors to the Language-Team address.\n"
"Plural-Forms: nplurals=2; plural=(n != 1);\n"

#: test_helpers.cpp:79
msgid "output"
msgstr "producción"
 . . .
```

In the project tree, create a `po` directory and move the `.po` file to it.

Note that the *GNU Gettext* manual ([2]) (in chapter 6) describes many more facets (heh heh) to the creation and manipulation of `.po` files.

2.4 Installing the .po Files

The installation process for the project should include installing the `.po` files. Where to install them in the system, if desired? It does not quite make sense to store them in a place like

```
/usr/local/share/locale/LL/LC\_MESSAGES}
```

because that contains `.mo` files (and some *Qt* `.qm` files!).

We would suggest something like `/usr/local/share/po/PACKAGE`. We should provide support in *Potext* for that. Once *Potext* supports parsing `.mo` files, the usual processes and location can be used. Future stuff.

The project, once installed, can also, if desired, copy the relevant language file to the user's configuration directory, `/home/user/.config/package/` or `C:/Users/user/AppData/Local` at the first run, and use it there.

3 Potext Architecture

This section provides a walk-through of the architecture of the *Potext* library. Much of the architecture is similar to *Tinygettext* ([3]), but there are some important changes and additions. Some notes about the classes and documentation:

- All classes and free functions are wrapped in the `po` namespace.
- The macro `_()` that normally wraps *GNU* function `gettext()` here wraps the *Potext* function `po::gettext()`.
- The related "gettext" functions are redefined in terms of *Potext* functions.
- In the diagrams, for the function parameters, we use "std::string", rather than "const std::string &" for brevity in the diagrams.
- Not every attribute or function is described. Some groups of items include `_xxxx_` to represent a number of similar functions.
- The copy constructors, principal assignment operators, and destructors are not described. See the header files to see if they are `default`, `delete`, or defined.

First, the big picture.

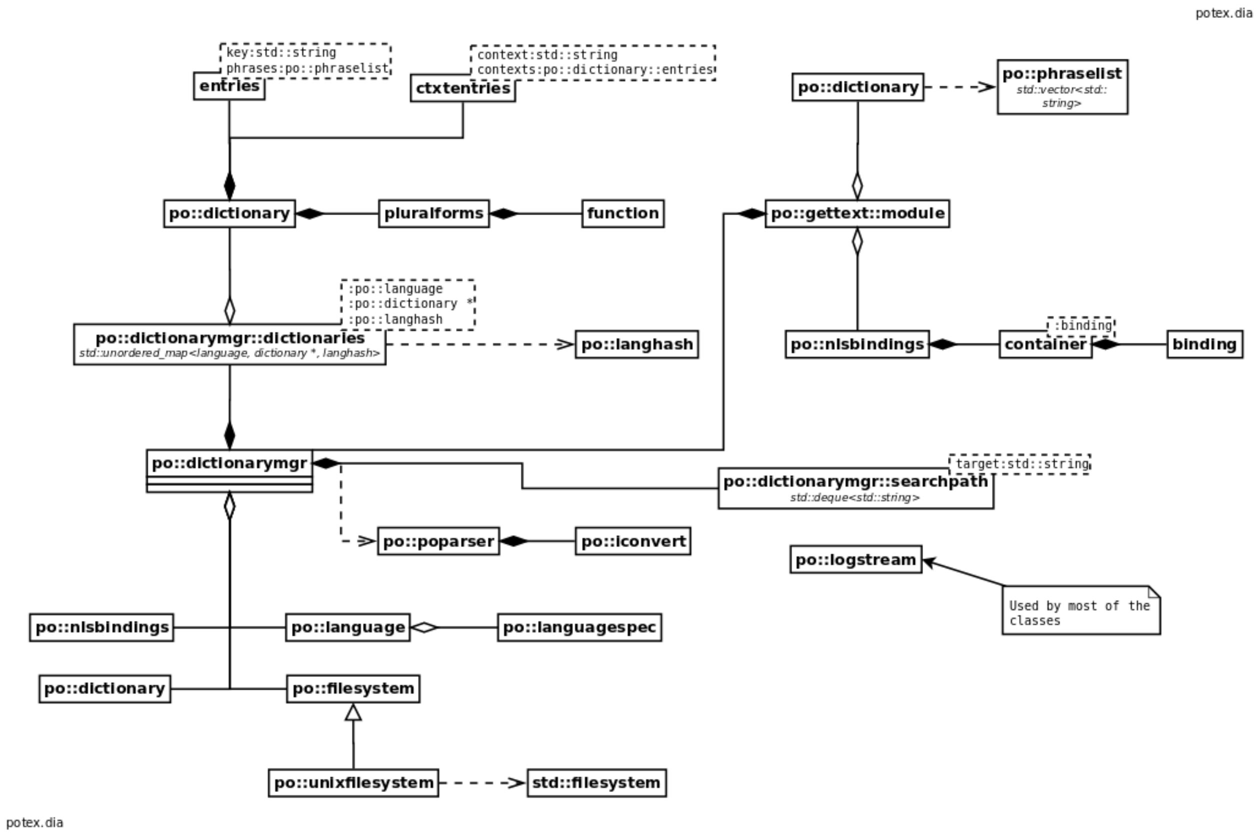


Figure 1: Potext "Big Picture" Architecture

The most important part of *Potext* is the `dictionary` class. It is filled by the `po::poparser::parse()` function, which takes a `.po` file and fills a dictionary object with a set of message strings and their translations. It also includes plural forms from the `.po` file to translate plural messages using "plural" functions. Each dictionary includes the message entries and context entries. For a description of the `.po` file, see the *GNU Gettext* manual ([2]).

The `dictionarymgr` class handles one or more dictionaries. It has been augmented to hold a new `nlsbindings` class to provided support for `bindtextdomain` and `textdomain`, which are *not* provided by *Tinygettext*.

The `dictionarymgr`'s additional member functions are used to implement the free functions in the `gettext` module, such as `gettext()` and `dgettext()`, etc. The `gettext` module is an addition to *Potext* to make it easy to switch from *Gettext* to *Potext*.

The `language` class supports the various parts of a domain name: language, country, modifier, long name, and the long name localized.

Parsing `.po` files is facilitated by the various file-system classes. Each `.po` file results in the creation of a dictionary.

The `poparser` class supports reading and parsing a `.po` file to create a `dictionary`.

The `iconvert` class supports converting the translations to another codeset (besides `UTF-8`) when creating the dictionaries.

The `logstream` class supports internal error logging, but can also be used by an application.

These classes are discussed in more detail in the following sections.

3.1 Logstream Class

The `po::logstream` class is a reimplementation of the `tinygettext::Log` class.

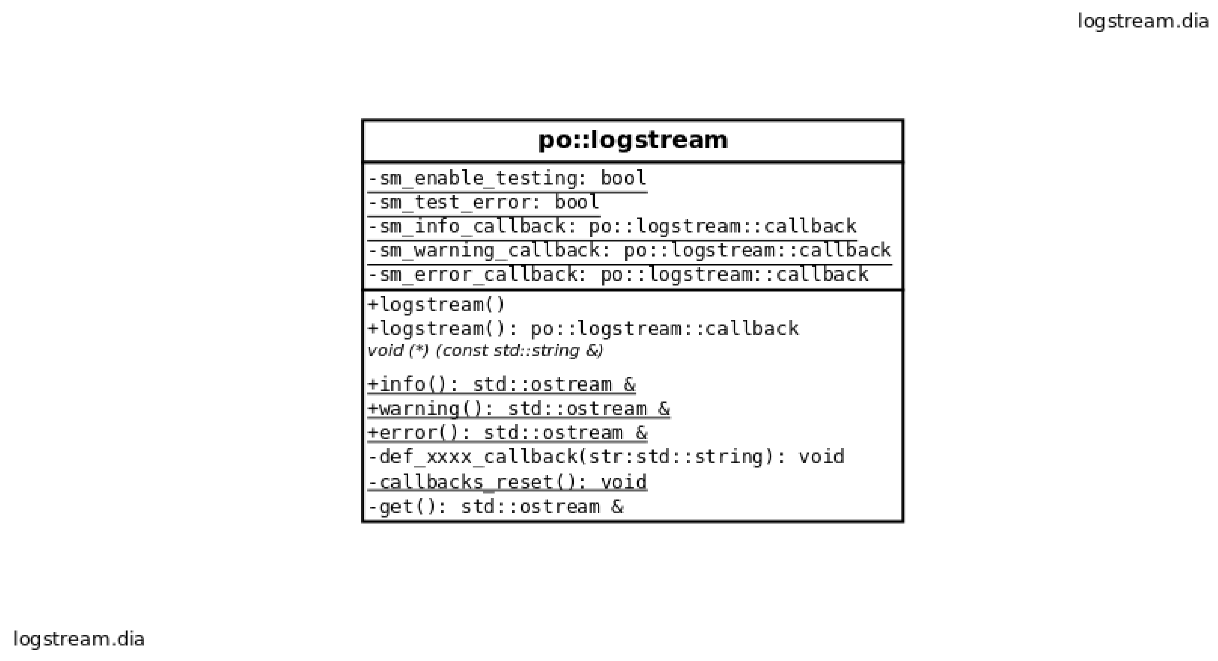

logstream.dia
po::logstream
-sm_enable_testing: bool
-sm_test_error: bool
-sm_info_callback: po::logstream::callback
-sm_warning_callback: po::logstream::callback
-sm_error_callback: po::logstream::callback
+logstream()
+logstream(): po::logstream::callback
void (*) (const std::string &)
+info(): std::ostream &
+warning(): std::ostream &
+error(): std::ostream &
-def_xxxx_callback(str:std::string): void
-callbacks_reset(): void
-get(): std::ostream &
logstream.dia

Figure 2: Log Stream (po::logstream)

The `po::logstream` class provides `std::ostream` objects for emitting errors, warnings, and information messages. It also provides the ability to set a callback function to change how the messages are emitted. It is used internally for writing status to the console. It can also be used by an application, but ...

... An interesting issue that we have not yet figured out is illustrated by the test application `hellopotext`. When run, all of the messages written to `std::cout` appear first, including the final message "SUCCESS". Then all of the messages logged during setup and translation in the *Potext* library appear when `hellopotext` is *exiting*.

3.2 Dictionary Manager Class

The `po::dictionarymgr` class is a reimplementation of the `tinygettext::dictionary_manager` class. It contains an `std::unordered_map` of shared pointers to `dictionary` objects, keyed by `language` objects which are searched using `operator ()` hash function in a `po::langhash` structure.

Currently, *Potext* does not do anything special with the `searchpath` deque.

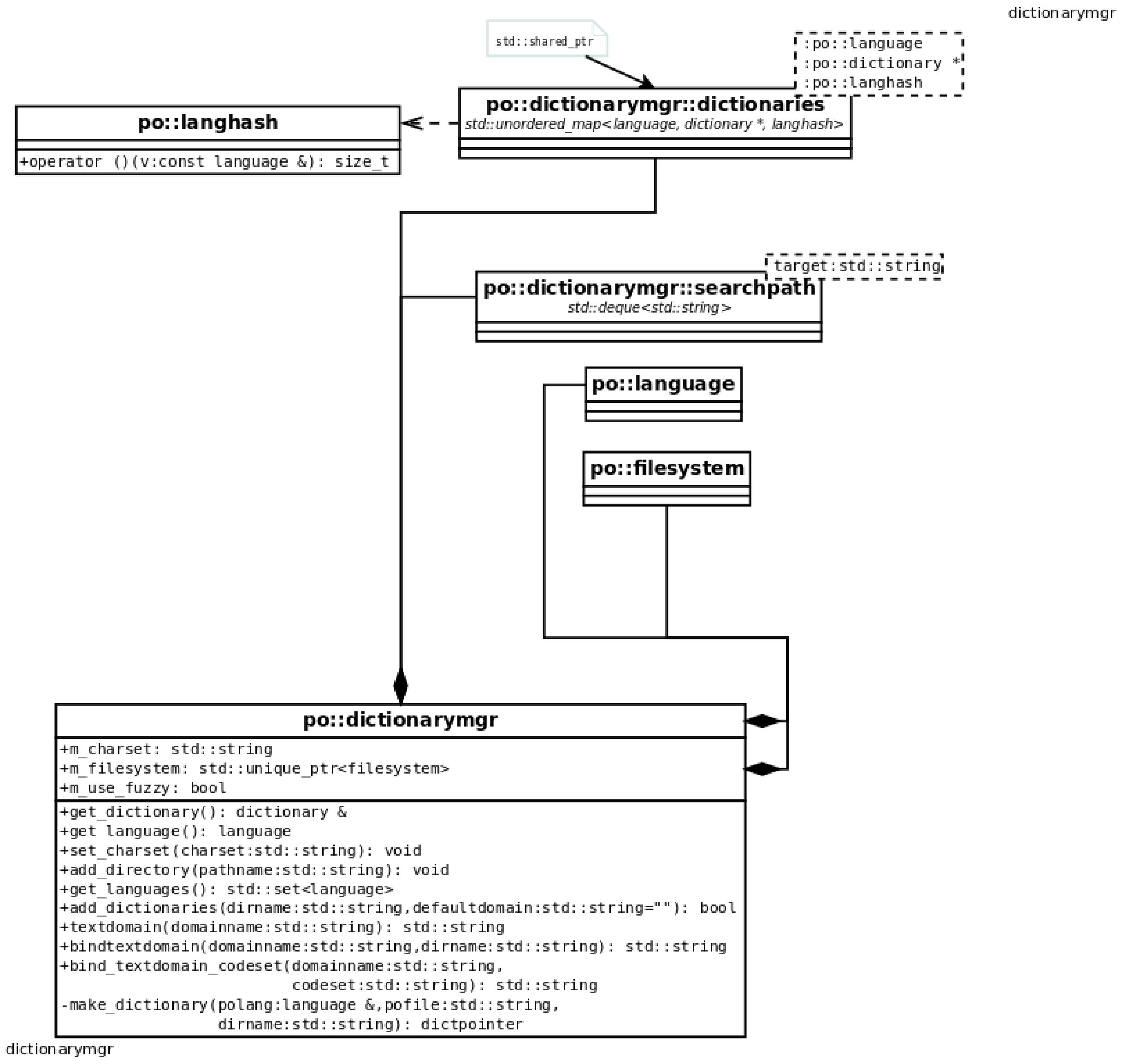

dictionarymgr
std::shared_ptr
:po::language
:po::dictionary *
:po::langhash
po::langhash
+operator ()(v:const language &): size_t
po::dictionarymgr::dictionaries
std::unordered_map<language, dictionary *, langhash>
target:std::string
po::dictionarymgr::searchpath
std::deque<std::string>
po::language
po::filesystem
po::dictionarymgr
+m_charset: std::string
+m_filesystem: std::unique_ptr<filesystem>
+m_use_fuzzy: bool
+get_dictionary(): dictionary &
+get language(): language
+set_charset(charset:std::string): void
+add_directory(pathname:std::string): void
+get_languages(): std::set<language>
+add_dictionaries(dirname:std::string,defaultdomain:std::string=""): bool
+textdomain(domainname:std::string): std::string
+bindtextdomain(domainname:std::string,dirname:std::string): std::string
+bind_textdomain_codeset(domainname:std::string, codeset:std::string): std::string
-make_dictionary(polang:language &,pofile:std::string, dirname:std::string): dictpointer
dictionarymgr

Figure 3: Dictionary Manager (po::dictionarymgr)

In *Tinygettext*, the `dictionary_manager` class coordinated multiple locale directories and the selection of a particular `dictionary` for a translation.

Potext's `dictionarymgr` currently handles only one directory, but it adds support for domain-binding and for actually using translation functions that accept a domain parameter. The new functions are described next.

- `get_bindings()`. This function returns an `nlsbindings` class reference that contains a list of domain names with the names of the directory and the character-set for each domain. The `nlsbindings` class provides the set-binding functions needed by the following new functions.
- `add_dictionaries()`. This function scans a directory for `.po` files and creates a `dictionary` for each one.
- `make_dictionary()`. This helper function opens a file using `std::filesystem` It then creates

an `std::shared_ptr` for a new `dictionary` and calls the static function `po::poparser::parse_po_file()`. Then it calls `po::nlsbindings::set_binding_values()` to create a corresponding binding object.

- `textdomain()`. This function sets the current domain to the given domain name. It is used in the `gettext` module to implement the `textdomain()` function.
- `bindtextdomain()`. This function associates a domain name with a locale directory in which to find the `.po` file. It is used in the `gettext` module to implement the `bindtextdomain()` function.
- `bind_textdomain_codeset()`. This function associates a domain name with a character-set to use in converting messages. It is used in the `gettext` module to implement the `bind_textdomain_codeset()` function.
- `get_dictionary()`.

3.3 Poparser Class

The `po::poparser` class is a reimplementation of the `tinygettext::POParser` class.

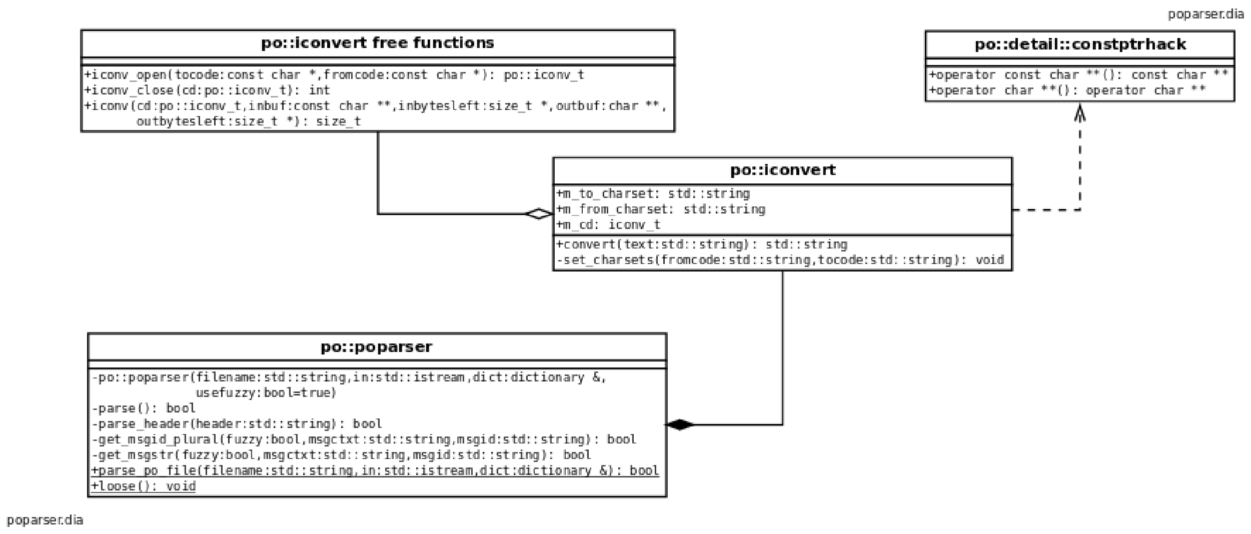

poparser.dia
po::iconvert free functions
+iconv_open(tocode:const char *,fromcode:const char *): po::iconv_t
+iconv_close(cd:po::iconv_t): int
+iconv(cd:po::iconv_t,inbuf:const char **,inbytesleft:size_t *,outbuf:char **, outbytesleft:size_t *): size_t
po::detail::constptrhack
+operator const char **(): const char **
+operator char **(): operator char **
po::iconvert
+m_to_charset: std::string
+m_from_charset: std::string
+m_cd: iconv_t
+convert(text:std::string): std::string
-set_charsets(fromcode:std::string,tocode:std::string): void
po::poparser
-po::poparser(filename:std::string,in:std::istream,dict:dictionary &, usefuzzy:bool=true)
-parse(): bool
-parse_header(header:std::string): bool
-get_msgid_plural(fuzzy:bool,msgctxt:std::string,msgid:std::string): bool
-get_msgstr(fuzzy:bool,msgctxt:std::string,msgid:std::string): bool
+parse_po_file(filename:std::string,in:std::istream,dict:dictionary &): bool
+loose(): void
poparser.dia

Figure 4: PO Parser (po::poparser)

The `poparser` "connects" a `.po` file, an input stream, and a `dictionary` object in order to populate the dictionary with plural forms, set the character-set, and use it (if needed) to convert the translated message string to the character-set, The static function `po::poparser::parse_po_file()` is called to create a temporary `poparser` and use it to read a file and fill in an empty `dictionary`.

The `po::poparser::get_string_line()` function handles the main task of parsing a line from the `.po` file and deciding what to do with it.

The `po::poparser::get_msgstr` function adds a message (which might be converted to a specified character-set) to the dictionary.

The `po::poparser::get_msgid_plural()` adds a plural form (see section 3.5 "Pluralforms Class" on page 16) or a contextual translation to the dictionary.

The `po::poparser` class uses the `po::iconvert` class to convert the string translation to the desired character-set. The `po::iconvert` class is a reimplementation of the `tinygettext::IConv` class. Note that it defines the `po::iconv_t` type.

3.4 Dictionary Class

The `po::dictionary` class is a reimplementation of the `tinygettext::Dictionary` class.

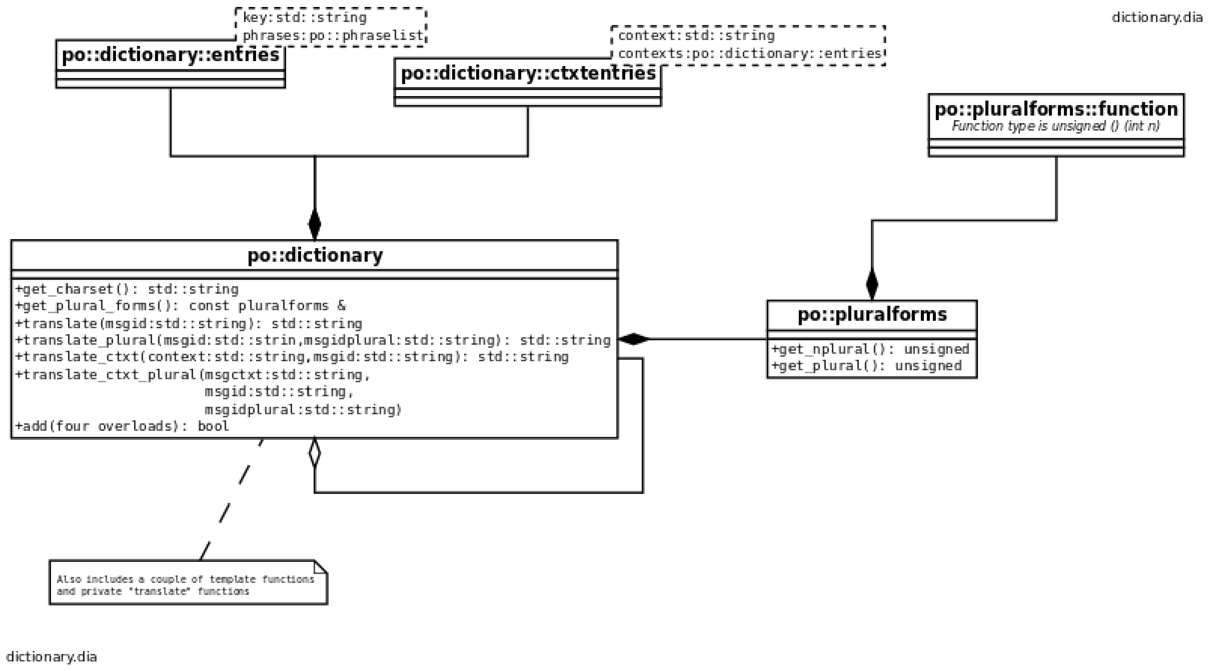

dictionary.dia
key:std::string
phrases:po::phraselist
po::dictionary::entries
context:std::string
contexts:po::dictionary::entries
po::dictionary::ctxtentries
po::pluralforms::function
Function type is unsigned () (int n)
po::dictionary
+get_charset(): std::string
+get_plural_forms(): const pluralforms &
+translate(msgid:std::string): std::string
+translate_plural(msgid:std::strin,msgidplural:std::string): std::string
+translate_ctxt(context:std::string,msgid:std::string): std::string
+translate_ctxt_plural(msgctxt:std::string, msgid:std::string, msgidplural:std::string)
+add(four overloads): bool
po::pluralforms
+get_nplural(): unsigned
+get_plural(): unsigned
Also includes a couple of template functions and private "translate" functions
dictionary.dia

Figure 5: Dictionary (po::dictionary)

The `dictionary` class holds a set of conversions of strings to a list of possible conversions, and another set of lists to support various message contexts.

The dictionary contains `entries`, an `std::unordered_map` of phrases keyed by a message ID string as used in *GNU* `gettext()`. A `phraselist` is simply an `std::vector<std::string>`.

The dictionary also contains `ctxtentries`, an `std::unordered_map` of `entries` keyed by a context string.

The dictionary also contains a `pluralforms` object that can be used to look up the proper plural translation. These functions provide the desired lookups:

- `translate()`.
- `translate_plural()`.
- `translate_ctxt()`.
- `translate_ctxt_plural()`.

Note that there are no functions that use the name of a domain as a parameter. Instead, the domain-using functions in the `gettext` module look up the dictionary associated with the desired domain, and use the appropriate translate function.

The `pluralforms` class deserves its own small section.

3.5 Pluralforms Class

The `po::pluralforms` class is a reimplementation of the `tinygettext::PluralForms` class. Each `.po` file contains a line describing how the translation of plurals is to be handled for each language:

```
Plural-Forms: nplurals=2; plural=n != 1;
```

The `pluralforms` class provides a static function for each possible plural form (and there are quite a number of them). These functions are inserted into an `std::unordered_map` which is keyed by strings like the one shown above. Some of these strings are extremely long. *(We could shorten the keys by ignoring the redundant part of the plural-forms string.)*

The `po::pluralforms::from_string()` function strips the spaces from a string parameter and does a fast lookup to return the appropriate `pluralforms` object.

3.6 Language Class

The `po::language` class is a reimplementation of the `tinygettext::Language` class.

The `po::languagespec` structure is a reimplementation of the `tinygettext::LanguageSpec` structure. This structure now uses `std::string` instead of character pointers.

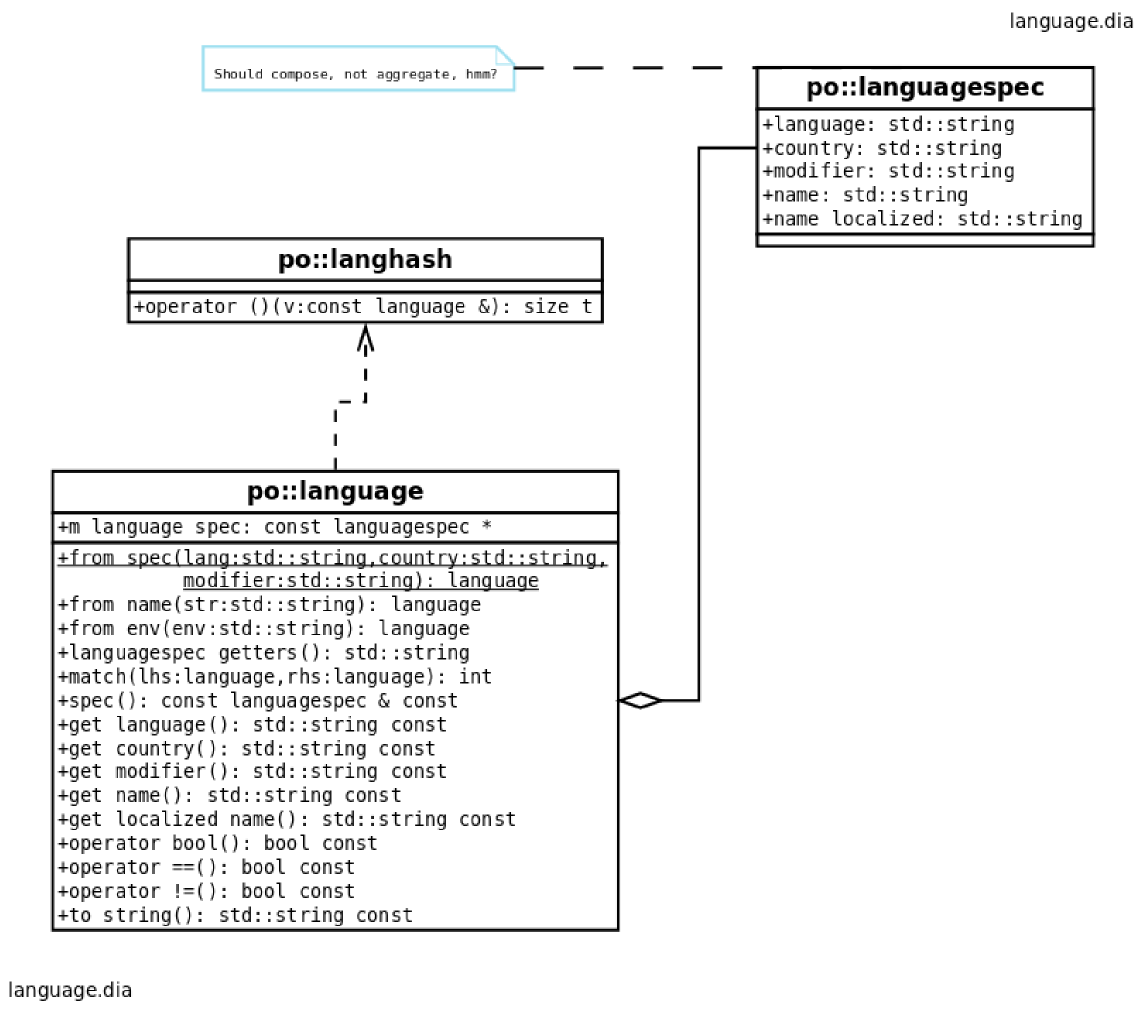

language.dia
Should compose, not aggregate, hmm?
po::languagespec
+language: std::string
+country: std::string
+modifier: std::string
+name: std::string
+name localized: std::string
po::langhash
+operator ()(v:const language &): size t
po::language
+m language spec: const languagespec *
+from spec(lang:std::string,country:std::string, modifier:std::string): language
+from name(str:std::string): language
+from env(env:std::string): language
+languagespec getters(): std::string
+match(lhs:language,rhs:language): int
+spec(): const languagespec & const
+get language(): std::string const
+get country(): std::string const
+get modifier(): std::string const
+get name(): std::string const
+get localized name(): std::string const
+operator bool(): bool const
+operator ==(): bool const
+operator !=(): bool const
+to string(): std::string const
language.dia

Figure 6: Language (po::language)

The `language` class is a wrapper for a `languagespec` structure. As shown in the figure, it provides functions to get and set the components of a language specification, to make comparisons, and converted the specification to a string.

The `dictionarymgr` uses the `language` as a key to look up the desired dictionary, and if not found, to make a new dictionary and add it to the dictionary container.

We still need to understand a little more about this class and its usage.

3.7 NLS Bindings Class

The `nlsbindings` class is an addition for the *Potext* library to support adding text-domain functions akin to those in *GNU Gettext*, but wrapped in the `po` namespace.

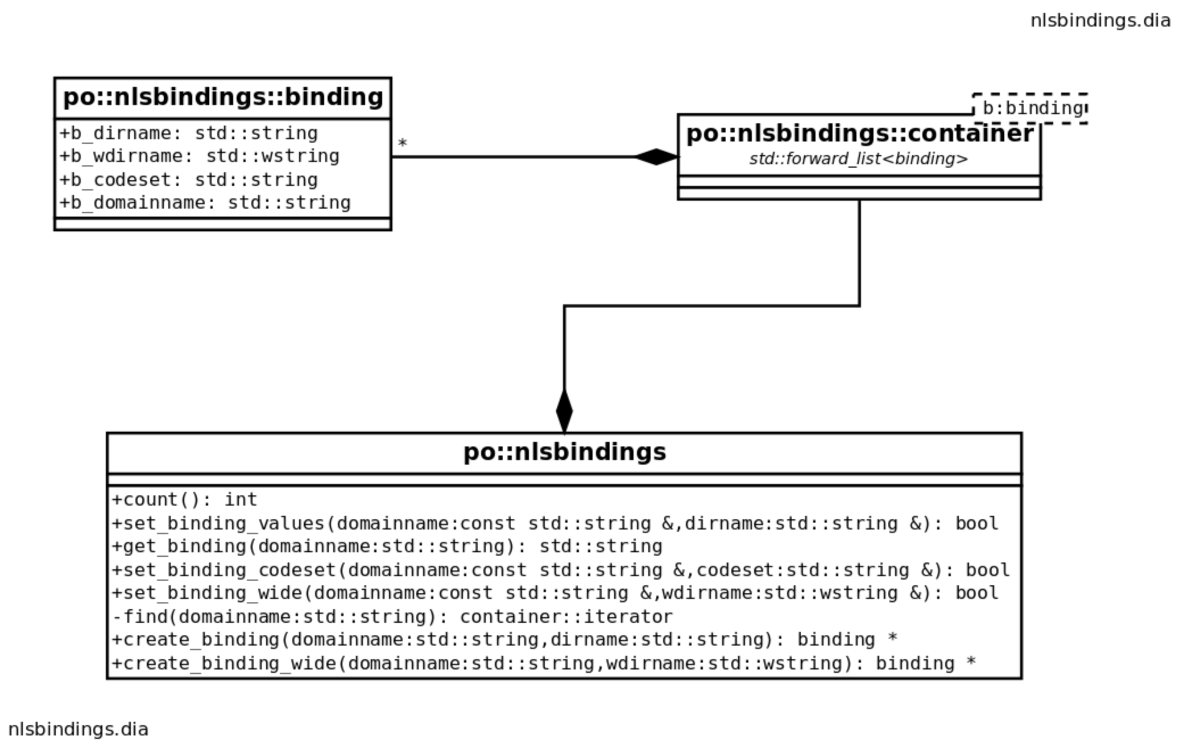

nlsbindings.dia
po::nlsbindings::binding
+b_dirname: std::string
+b_wdirname: std::wstring
+b_codeset: std::string
+b_domainname: std::string
*
b:binding
po::nlsbindings::container
std::forward_list<binding>
po::nlsbindings
+count(): int
+set_binding_values(domainname:const std::string &,dirname:std::string &): bool
+get_binding(domainname:std::string): std::string
+set_binding_codeset(domainname:const std::string &,codeset:std::string &): bool
+set_binding_wide(domainname:const std::string &,wdirname:std::wstring &): bool
-find(domainname:std::string): container::iterator
+create_binding(domainname:std::string,dirname:std::string): binding *
+create_binding_wide(domainname:std::string,wdirname:std::wstring): binding *
nlsbindings.dia

Figure 7: NLS Bindings (po::nlsbindings)

It provides some simplified implementations of *GNU Gettext* functions that lack such niceties as locking and checking for SUID root applications. These can be added later as the need becomes apparent. For details, see the code in the *GNU Gettext* project in it's `gettext-runtime/intl` directory.

3.8 Gettext Module

The *Potext* `gettext` module is discussed in the next section. (See section 4.2 "Gettext Module" on page 19.)

4 GNU Gettext and Its Potext Replacements

This section briefly covers the public functions and macros of *GNU Gettext* and our replacements for them. Here are the main differences:

- All implementations are functions; no macros are used.
- All functions are inside the `po` namespace.
- All character pointers are replaced by `std::string`.
- Lookups are done via `.po` files, at present.
- None of the functions with a "category" parameter are implemented at this time. Those function would seem to need to find an load up another `dictionary` object. Also, by far the most common translation files on a *UNIX* system are in the `LC_MESSAGE` subdirectories.

4.1 GNU Gettext Header File

This section provides a walkthrough of the `gettext.h` header file of the *Potext* library. This is useful in understanding *Gettext* versus *Potext*.

Let us survey the important functions and macros that are used in the `gettext.h` header file (see `/usr/include/libintl.h`):

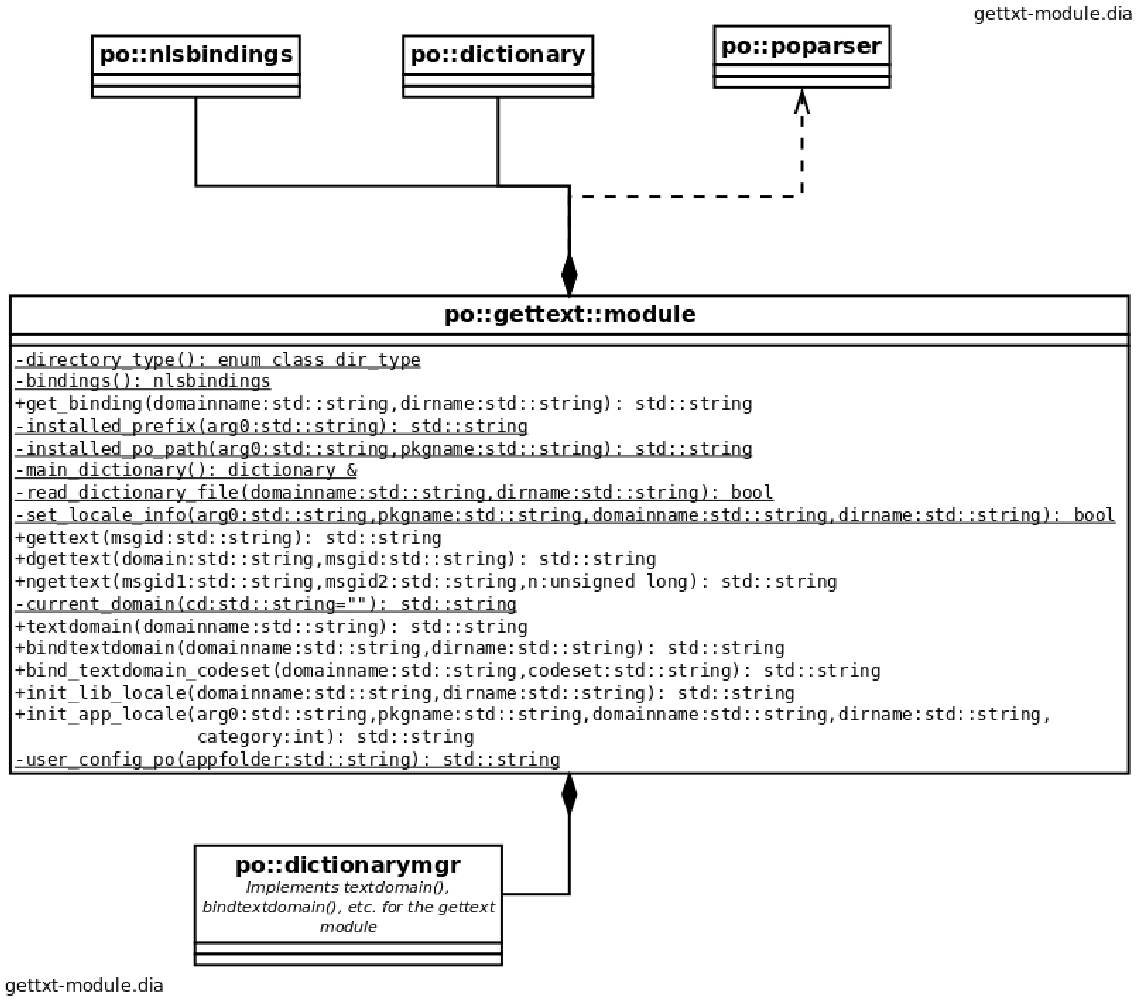
- `ENABLE_NLS`. If defined in a GNU automake project, this includes the `libintl.h` header file, which is not needed in an application using the *Potext* library for translation. NLS can be disabled via `-disable-nls`.
- `DEFAULT_TEXT_DOMAIN`. If `ENABLE_NLS` is defined, this macro causes `gettext` to be defined as `dgettext`, and `ngettext` to be defined as `dngettext`. If `ENABLE_NLS` is *not* defined, then the following "functions" are "voided":
  - `gettext`
  - `dgettext`
  - `dcgettext`
  - `ngettext`
  - `dngettext`
  - `dcngettext`
  - `textdomain`
  - `bindtextdomain`
  - `bind_textdomain_codeset`
- `DEFAULT_TEXT_DOMAIN` revisited. If defined, more macros are defined, for message-context support. These call `pgettext_aux` or `npgettext_aux`
  - `pgettext`
  - `dpgettext`
  - `dcpgettext`
  - `npgettext`
  - `dnpgettext`
  - `dcnpgettext`
- `GNULIB_defined_setlocal`. If defined, uses the `rpl_setlocale` from *gnulib* as `setlocale`.
- `gettext_noop`. A pseudo function that marks code for extraction of messages, but does not call `gettext`.
- `pgettext_expr`. Calls `dcpgettext_expr()`.
- `dpgettext_expr`. Calls `dcnpgettext_expr()`.

Do we want *potext* to be a drop-in replacement for all this stuff? We shall try!

4.2 Gettext Module

The `gettext` module provides a reimplementation of *GNU Gettext* "gettext" functions in the `po` namespace.

Here, we use the term "module" to describe a set of related functions that are not members of a class. All functions in this module are in the `po` namespace, or are `static` and internal to the module.



```
-directory_type(): enum class dir_type
-bindings(): nlsbindings
+get_binding(domainname:std::string,dirname:std::string): std::string
-installed_prefix(arg0:std::string): std::string
-installed_po_path(arg0:std::string,pkgname:std::string): std::string
-main_dictionary(): dictionary &
-read_dictionary_file(domainname:std::string,dirname:std::string): bool
-set_locale_info(arg0:std::string,pkgname:std::string,domainname:std::string,dirname:std::string): bool
+gettext(msgid:std::string): std::string
+dgettext(domain:std::string,msgid:std::string): std::string
+ngettext(msgid1:std::string,msgid2:std::string,n:unsigned long): std::string
-current_domain(cd:std::string=""): std::string
+textdomain(domainname:std::string): std::string
+bindtextdomain(domainname:std::string,dirname:std::string): std::string
+bind_textdomain_codeset(domainname:std::string,codeset:std::string): std::string
+init_lib_locale(domainname:std::string,dirname:std::string): std::string
+init_app_locale(arg0:std::string,pkgname:std::string,domainname:std::string,dirname:std::string,
                 category:int): std::string
-user_config_po(appfolder:std::string): std::string
```

Figure 8: Gettext Module (namespace po)

We are slowly implement the various "gettext" functions shown in that figure, plus some others that are not shown. See the next section for a brief discussion of our copy of the `GNU Gettext` header file.

The following table lists the functions/macros and their purpose and status. The implementations are member functions of `po::dictionary` or `po::dictionarymgr` (*); all dictionaries are contained and referenced in `po::dictionarymgr`, either as the main dictionary or a dictionary selected based on domain. Fall-back functions are noted for some "none" implementations.

]

Table 1: Gettext Functions

Function	**Implementation**	**Purpose**
`textdomain()`	textdomain() *	Set or change the current global domain for the `LC_MESSAGE` category.
`bindtextdomain()`	bindtextdomain() *	Set or change the locale directory for the given domain. `LC_MESSAGE` category. The wide-character `UTF-16` version for *Windows* is not yet implemented.
`bind_textdomain_codeset()`	partial *	Set or change the character-set for the given domain. `LC_MESSAGE` category.
`gettext()`	translate()	Single message ID translation.
`dgettext()`	translate()	Single message ID translation in a specific domain.
`dcgettext()`	none: dgettext()	Single message ID translation in a specific domain and specific language category. For all get-text functions with a category parameter, there is currently no implementation, just a fall-back to the non-category version.
`ngettext()`	translate_plural()	Message ID translation using a specific singular or plural form.
`dngettext()`	translate_plural()	Message ID translation using a specific singular or plural form for a given domain.
`dcngettext()`	none: translate_plural()	Message ID translation using a specific singular or plural form for a given domain and a given locale category.
`pgettext()`	translate_ctxt()	Single message ID translation for a given context (e.g. "console" versus "gui". The 'p' stands for 'particular'.
`dpgettext()`	translate_ctxt()	Single message ID translation for a given context and the given domain.
`dpcgettext()`	none: dpgettext()	Single message ID translation for a given context, given domain, and category other than `LC_Messages`.
`npggettext()`	no	Probaby worth doing.
`dnpggettext()`	no	Probaby worth doing.
`dcnpggettext()`	no	Probaby not worth doing.

5 Potext Tests

This section provides a useful walkthrough of the testing of the *potext* library. They illustrate the various way in which the *Potext* library can be used by a develper.

These tests are not installed; they are in the transitory directory `potext/build/library/tests`. The test `*.po` files are in the directory `potext/library/tests` and its subdirectories. The shell script `library/tests/tests.sh` runs all of the `potext_test` tests listed in the following file, plus a couple more.

```
library/tests/testlines.list
```

It specifies the tests described here, but using only single-word phrases. The reason is that we have not yet figured out how to deal with phrases enclosed in double-quotes in a shell script.

Let us survey the main features of all of the test `po` files:

- `po` directory. This directory contains a small sampling of `.po` files from the *GNU Gettext* project. They have been pared down a bit just to save a few bytes, and a few extra translations have bee added for testing. They are used in the new `hellopotext` test program to test the functions in the `gettext` C++ module.
- `library/tests/de.po`. A basic `po` file with just `msgid` keys and `msgstr` translations in Deutsche (German, Alemania).
- `library/tests/broken.po`. This file has an entry `msgstr[10]`, obviously bad.
- `library/tests/helloworld/de.po`. It has some `msgctxt` sections with `msgid_plural`, `msgstr[0]` for a singular translation and `msgstr[1]` for a plural translation for each context (none, "gui", and "console").
- `library/tests/level/de.po`. Contains basic entries plus a number of entries with a blank message-ID followed by a long description and a message-string with a blank value followed by a long translation. NEED TO FIGURE THAT OUT. Also includes a couple of `printf()` format statements.
- `library/tests/po/de.po`. Another basic file with a number of translations and a weird message-ID called "umlaut".
- `library/tests/po/de_AT.po`. A short file with "umlaut" and a couple of plurals.
- `library/tests/po/fr.po`. Contains one German translation. Wtf?

One thing to watch for in running the tests. The test programs write output to `std::cout` or `std::cerr`, while the *Potext* library internals use the `po::logstream` class functions. What happen is the all of the application output comes first, while the log-stream message are not emitted until test program exit. Not sure why the latter aren't "flushed" immediately.

5.1 Hello Potext

This test is a work in progress. Without arguments, it runs through basic tests of the following functions. The main domain is provided to `po::init_app_locale()` which should normally be called in the applications's `main()` entry point function.

- `_()` and `gettext()`. This function does a message translation lookup using the current domain, which is logged in the `init_app_locale()` function. This smoke test illustrates the most common case we want to cover, which is the translation of a phrase according to the main (or only) domain and locale directory loaded. Also included is a test where the original message is not present in the `.po` file.
- `dgettext`. This function looks up a domain's dictionary and uses it for the translation. This test runs through all of the domains (i.e. `.po` files) in the `po` project directory. Currently tested are the *es*, *fr*, *de*, and a bogus domain named *xx*, which should just return the input message ID.

- `dcgettext`. This test does not do anything. Currently *Potext* does not handle locale categories. The reasons? First, the most common use case is looking up message translations in the `LC_MESSAGES` locale category. Second, this translation would require loading additional locale directories and their dictionaries. With this complication, we will sit on this problem for awhile.
- `ngettext`. This test handles plural forms in the current domain. It deals only the the main domain, `es`, It tests translating the plurals of the following singulars: "File" and "Person". The translations are likely not accurate, as they were provided by *Google Translate*. But they adequately test the process.
- `dngettext`. This test handles plural forms in a specified domain. Currently tested are the *es*, *fr*, *de*, and a bogus domain named *xx*, which should just return the input message ID.
- `dcngettext`. This function is not yet implemented, due to difficulties with selecting the category directory, as discussed above.
- `pgettext`. This test applies only to the domain specified in the `init_app_locale()` function.

Some functions not yet tested because of the implmentation difficulties noted above.

- `dcgettext()`.
- `dcngettext()`.

Additional arguments can change the default domain. We will document these real soon now.

5.2 Potext Test Program

This small application is the best test of *potext* from the perspective of a developer wanting to use it in an application. Running it without any arguments shows a list of 8 tests. These are reflected in the following sections.

5.2.1 Potext Test 'Translate'

These tests are run using a command like the following:

```
$ ./build/library/tests/potext_test translate <...options...>
```

By running `potext_test` without any options, one sees four "translate" commands. The four variations on the "translate" test are described in the following sub-sections.

5.2.1.1 Translate Basic: Po File and Sample Message

This test is a simple translation of a word. The basic "translate" test is run by the following form, which has an argument count of 4.

```
$ ./build/library/tests/potext_test translate <file> <msg>
```

The file is a test `.po` file in the `tests` directory. The message is a phrase present in that file, such as *"F1 - show/hide this help screen"*, translated in `de.po` as *"F1 - Hilfe anzeigen/verstecken"*.

Here is the run:

```
$ ./build/library/tests/potext_test translate \
   library/tests/de.po "F1 - show/hide this help screen"
```

The output is

```
Translation: "F1 - Hilfe anzeigen/verstecken"
```

If only a part of the message is provided, of course there is no match, and the message is

```
Translation: "F1 - Hilfe"
[potext] Couldn't translate: "F1 - Hilfe"
```

This second test shows that any deviation from a supported message causes an warning, and returns the original message. These deviations include missing letters, missing punctuation, additional spaces. *We wonder if we can work around such issues in this library. We shall see.*

5.2.1.2 Translate: Po File, Context, and Sample Message

This test is run by the following form, which has an argument count of 5.

```
$ ./build/library/tests/potext_test translate <file> <context> <msg>
```

This test requires a `po` file with message-context entries such as these three different entries found in `library/test/helloworld/de.po` for the phrase "Hello World":

```
msgctxt ""
msgid "Hello World"
msgctxt "console"
msgid "Hello World"
msgctxt "gui"
msgid "Hello World"
```

Please note that the *GNU* `gettext()` documentation says that an empty message context (`msgctxt ""`) is *not* the same as a missing message context. In the "helloworld" test program, these contexts are provided by the following lines:

```
pgettext("", "Hello World")
pgettext("console", "Hello World")
pgettext("gui", "Hello World")
```

The macros `ngettext` and `npgettext` are also used to provide access to the various plural forms in that `po` file. In any case, we need to use `library/test/helloworld/de.po` for this test. An actual test run:

```
$ ./build/library/tests/potext_test translate \
      library/tests/helloworld/de.po "console" "Hello World"
```

The output is

```
Context 'console' translation: "Hallo Welt (singular) in der Console"
```

If the `<context>` parameter is not found in the `po` file, a message is emitted to indicate the error.

5.2.1.3 Translate: Po File with Singular and Plurals

This test is run by the following form, which has an argument count of 6.

```
$ ./build/library/tests/potext_test translate <file> <singular>
      <plural> <N>
```

The singular and plural parameters are message IDs, such as "Hello World". This command is a bit tricky; the N value is not a C index, but an index that starts at 1. The N parameter ranges from 1 to the last array value in the `po` file. The number of singular/plural translations depends on the language and is specified in the specific `.po` file using a header declaration such as "`Plural-Forms: nplurals=2; plural=(n != 1);`". Look at `pluralforms.cpp` to see all the plural-forms settings and "callbacks" that are support. Some of these forms support Slavic and Arabic languages, and we are not able to test them.

An actual test run:

```
$ ./build/library/tests/potext_test translate \
      library/tests/helloworld/de.po "Hello World" "Hello Worlds" 1
```

The output is

```
TODO
```

5.2.1.4 Translate: Po File with Context, Singular, and Plurals

This test is run by the following form, which has an argument count of 7.

```
$ ./build/library/tests/potext_test translate <file> <context> \
   <singular> <plural> <N>
```

An actual test run:

```
$ ./build/library/tests/potext_test translate \
      library/tests/helloworld/de.po "gui" "Hello World" "Hello Worlds" 0
```

The output is

```
TODO
```

5.2.2 Potext Test 'Directory'

These tests are run using a command like the following:

```
$ ./build/library/tests/potext_test directory <dir> <msg> [<lang>]
```

5.2.3 Potext Test 'Language'

These tests are run using a command like the following:

```
$ ./build/library/tests/potext_test language <lang>
```

5.2.4 Potext Test 'Language Directory'

These tests are run using a command like the following:

```
$ ./build/library/tests/potext_test language-dir <dir>
```

5.2.5 Potext Test 'List Message Strings'

These tests are run using a command like the following:

```
$ ./build/library/tests/potext_test list-msgstrs <file>
```

6 Summary

Contact: If you have ideas about *Potext* or a bug report, please email us (at `mailto:ahlstromcj@gmail.com`).

Remaining issues:

The `*.po` files "msgid" and "msgstr" entries have punctuation marks and trailing spaces that are significant. CAN WE GET AROUND THIS ISSUE? We need to trim these trailing characters in both specifications, and also when translating, and restore them in the translation.

7 References

The *Potext* reference list.

References

[1] GNU Translation Team. *GNU Gettext Code* `https://git.savannah.gnu.org/git/gettext.git`. 2023.

[2] GNU Translation Team. *GNU Gettext Tools manual, version 0.22*. `https://www.gnu.org/software/gettext/manual/gettext.pdf`. 2023.

[3] Tinygettext Team. *Tinygettext on GitHub*. `https://github.com/tinygettext/tinygettext`. 2023.