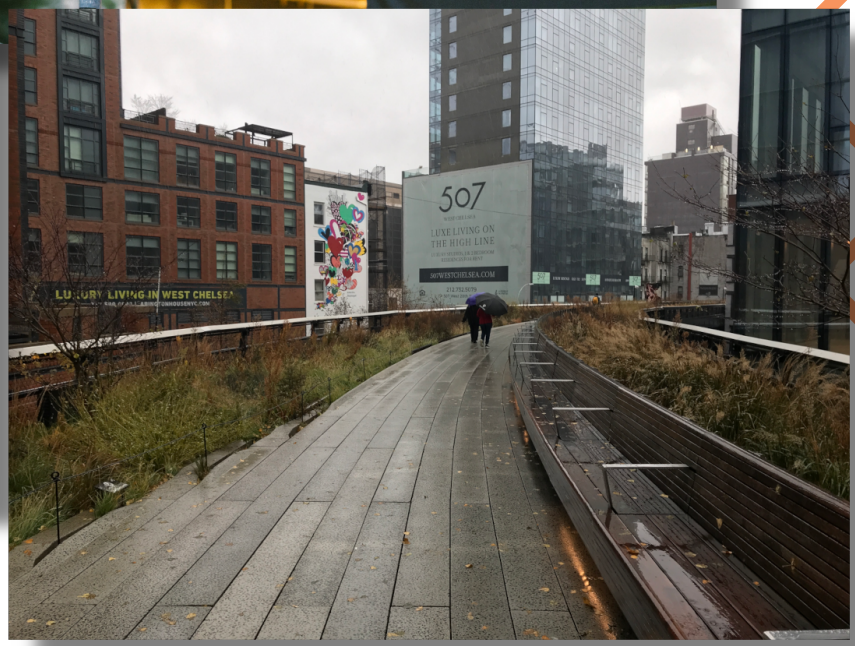
Quneiform 2025

User Manual

Internationalization & Localization Analysis System



Blake Madden

Quneiform 2025

User Manual

Blake Madden

Quneiform 2025

User Manual

Published in the United States

Table of contents

Chapter 1

Overview

`Quneiform` is a command-line utility and graphical user interface that scans source and resource files to check for various i18n and l10n issues. Additionally, the GUI version provides pseudo-translation support for *gettext* catalogs (*.po files).

1.1 File Support

`Quneiform` supports static analysis for the following:

- C code
- C++ code ('98 and modern C++)
- Windows resource files (*.rc)
- macOS application Info.plist files

Static analysis and pseudo-translation are available for:

- GNU *gettext* translation files (*.po and *.pot)

Additionally, it offers specialized support for the following frameworks:

- wxWidgets
- Qt
- KDE
- GTK$^+$
- Win32
- MFC

1.2 Static Analysis

The command line and GUI versions provide the following checks:

- Strings exposed for translation[1] that probably should not be. This includes (but not limited to):
  - Filenames
  - Strings only containing `printf()` commands
  - Numbers
  - Regular expressions
  - Strings inside of debug functions
  - Formulas
  - Code (used for code generators)
  - Strings that contain URLs or email addresses
- Strings not available for translation that probably should be.
- Localizable strings being concatenated at runtime.
- Use of deprecated text macros (e.g., `wxT()` in wxWidgets, `_T()` in Win32).
- Use of deprecated string functions (e.g., `_tcsncpy()` in Win32).
- Suspect usage of i18n functions.
- Strings with malformed syntax (e.g., malformed HTML tags).
- Files that contain extended ASCII characters, but are not UTF-8 encoded. (It is recommended that files be UTF-8 encoded for portability between compilers.)
- Strings that contain extended ASCII characters that are not encoded. ("Danke schön" instead of "Danke sch\U000000F6n".) Encoding extended ASCII characters is recommended for best portability between compilers.
- UTF-8 encoded files which start with a BOM/UTF-8 signature. It is recommended to save without the file signature for best compiler portability.
- `printf()`-like functions being used to just format an integer to a string. It is recommended to use `std::to_string()` to do this instead.
- Pluralization and pronoun issues.
- Usage of half-width characters.
- Literal, localizable strings being used in search and comparison functions.
- `printf()` command mismatches between source and translation strings.
- Accelerator mismatches between source and translation strings.
- Number mismatches between source and translation strings.
- Complicated strings that should have translator comments added.
- Font issues in Windows resource files (Microsoft, STRINGTABLE resource):
  - Dialogs not using "MS Shell Dlg" or "MS Shell Dlg 2."
  - Dialogs with non-standard font sizes.
- Info.plist files not including any `CFBundleLocalizations` keys.

Code formatting and other issues can also be checked for, such as:

- Trailing spaces at the end of a line.
- Tabs (instead of spaces).
- Lines longer than 120 characters.
- Spaces missing between "//" and their comments.
- ID variable[2] assignment issues:
  - The same value being assigned to different ID variables in the same source file (e.g., "wxID_HIGHEST + 1" being assigned to two menu ID constants).
  - Hard-coded numbers being assigned to ID variables.
  - Out-of-range values being assigned to MFC IDs (TN020: ID Naming and Numbering Conventions).

[1] Strings are considered translatable if inside of gettext, wxWidgets, Qt, or KDE (ki18n) i18n functions. This includes functions and macros such as `gettext()`, `_()`, `tr()`, `translate()`, `QT_TR_NOOP()`, `wxTRANSLATE()`, `i18n()`, etc.

[2] Variables are determined to be ID variables if they are integral types with the whole word "ID" in their name.

1.3 Pseudo-translation

(available in the GUI version)

Pseudo-translation includes features such as:

- Multiple methods for character replacement (e.g., replacing characters with accented variations or upper casing them).
- Increasing the width of the translations. This is useful for ensuring that strings are not being truncated at runtime.
- Wrapping the translations in braces. This is useful for ensuring that strings are not being pieced together at runtime.
- Appending a unique ID to each translation. This is useful for finding where a translation is being loaded from.

When pseudo-translating, a copy of all string catalogs will be created and have their translations filled with mutations of their respective source strings. These files (which will have a "pseudo_" prefix) can then be compiled and loaded by your application for integration testing.

> **Note**
> After processing a folder, the **Analysis Log** tab of the bottom window will display a list of all pseudo-translated resource files that were generated.

Chapter 2

Building

`Quneiform` requires a C++20 compiler and can be built on Windows, macOS, and Linux.

Cmake scripts are included for building both the command-line and GUI (graphical user interface) versions. *Cmake* 3.25 or higher is required.

If compiling with GCC, GCC 12.2.1 or higher is required.

For the GUI version, wxWidgets 3.3 or higher is required.

2.1 Command-line Utility

`Quneiform` can be configured and built with *Cmake*.

On Unix:

Listing 2.1 `Terminal`

```
cmake . -DCMAKE_BUILD_TYPE=Release
cmake --build . --target all -j $(nproc) --config Release
```

On Windows, “CMakeLists.txt” can be opened and built directly in Visual Studio.

After building, “quneiform” will be available in the “bin” folder.

2.2 GUI Version

wxWidgets 3.3 or higher is required for building the graphical user interface version.

Download wxWidgets, placing it at the same folder level as this project. After building *wxWidgets*, *Quneiform* can be configured and built with *Cmake*.

On Linux/macOS:

Listing 2.2 `Terminal`

```
# download and build wxWidgets one folder above
cd ..
git clone https://github.com/wxWidgets/wxWidgets.git --recurse-submodules
cd wxWidgets
cmake . -DCMAKE_INSTALL_PREFIX=./wxlib -DwxBUILD_SHARED=OFF \
    -D"CMAKE_OSX_ARCHITECTURES:STRING=arm64;x86_64" \
    -DCMAKE_OSX_DEPLOYMENT_TARGET=10.15 \
    -DwxBUILD_OPTIMISE=ON -DwxBUILD_STRIPPED_RELEASE=ON \
    -DCMAKE_BUILD_TYPE=Release
cmake --build . --target install --config Release

# go back into the project folder and build the GUI version
cd ..
cd quneiform/gui
cmake . -DCMAKE_BUILD_TYPE=Release
cmake --build . --target all --config Release
```

> **Note**
> On macOS, a universal binary 2 (containing arm64 and x86_64 binaries) will be built with the above commands.

On Windows with Visual Studio, build wxWidgets with the defaults, except `wxBUILD_SHARED` should be set to "OFF" (and `MAKE_BUILD_TYPE` set to "Release" for release builds).

Open "gui/CMakeLists.txt" in Visual Studio, setting the *CMake* setting's configuration type to "Release" for a release build.

After building, "quneiform" will be available in the "bin" folder.

Part I

i18n & l10n Issues

Chapter 3

Internationalization Issues

Internationalization (i18n) occurs during the development stage of software, which includes (but not limited to) the following areas:

- Preparing the program to display numbers and dates using the client's regional settings.
- Exposing strings for translation.
- Ensuring that strings not meant for translation are not exposed to translators.

I18N issues include any problems during this stage which will later affect either translation efforts or regional display issues for clients. This chapter will discuss these various issues and provide recommendations for how to detect and correct them using *Quneiform*.

3.1 Embedded Strings

Embedded strings are strings that are hard coded in source code and not made available for translation. It is recommended to expose these strings to translators via mechanisms such as *gettext* functions, string tables, *.resx files, etc.

In *Quneiform*, checking the option **Strings not exposed for translation** will warn about any embedded strings in the source code.

Additionally, resource catalogs can also be pseudo-translated to assist with finding UI strings that are not available to translators.

3.2 Concatenated Strings

When multiple strings are pieced together at runtime, this creates a concatenated string. Consider the following code sample:

```
wxString TOO_DIFFICULT_DESCRIPTION =
        _(L"Text is too difficult to be classified to a specific grade level because "
           "it contains too many ");
TOO_DIFFICULT_DESCRIPTION += fryGraph->GetScores().front().IsWordsHard() ?
                            _(L"high syllable words.") :
                            _(L"long sentences.");
```

Here, we see a string "...it contains too many" being concatenated with either "high syllable words." or "long sentences.". This is an issue because it assumes that translations will follow the same word ordering and grammar as the source language. Also, it fails to provide context to the translator as to what all is being

combined into this message. Rather than seeing a single message, translators are faced with multiple string resources without knowing that they even relate to each other.

In this example, the solution is to create two full (albeit lengthy) string resources for both variations of the possible message.

Checking the option **Concatenated strings** will warn about any possible concatenated strings.

3.3 Articles and Dynamic Content

An article (e.g., "an", "the") proceeding dynamic content in a string presents grammatical issues for translators. For example, consider this string:

"Find the {}"

In the above example, the "{}" placeholder is probably being used in multiple contexts. The issue with this is that many languages have multiple articles that change based on the noun they relate to (Calek 21). In the case of German, translating "the" as "die," "der," or "das" is bound to be incorrect for some of the instances where this string is used.

This can even be an issue for English. For example, with this resource:

"A %s is required"

'A' will be grammatically incorrect if "%s" is replaced with a word starting with a vowel sound.

It is better to have multiple string resources, one for each noun variation, rather than relying on placeholders. This ensures grammatically correct translations, as well as removing ambiguity for the translators.

3.4 Pronouns and Dynamic Content

Strings that consist only of a pronoun (e.g., "him," "she") will present grammatical problems if being formatted into larger messages. It is recommended to use multiple string resources with each pronoun variation instead replacing placeholders with these values. Another option is to use gender-neutral language instead of relying on a list of dynamic pronouns (Pérez and Sáenz 33).

As an example, assume we have these string resources:

"her"

"him"

"them"

And we have another message:

"Provide %s the reset link"

which dynamically replaces "%s" with one of the above values.

First, we should remove all three pronoun strings. Then we can provide three versions of the full message, replacing the "%s" with each of the pronouns. This will ensure that translators will have full control over how to translate the message when the gender changes.

Another option is to replace the message with gender-neutral language:

"Provide **the user** the reset link"

3.5 Pluralization

The same message being used for both singular and plural situations presents issues because grammar issues between the source and target languages.

These type of strings are generally used in two situations:

- When a quantity is dynamically formatted into the message.
- When a quantity is not known and the message is simply conveying that one or more items can be involved.

For the former, it is preferred to have two (or more) separate resource strings; one for singular and one (or more) for plural variations. Some frameworks provide pluralization functions (`ngettext()`, `wxPLURAL()`) which can be used to provide separate strings for these variations. As an example:

"%zu item(s)"

Here a number is being formatted into this message at runtime. It is better to pass singular and plural variations to a pluralizing function instead. For example:

```
wxPLURAL("%zu item", "%zu item", items.size())
```

For the latter, it is preferred to reword the string (removing the "(s)"). As an example:

"Variable(s) must be selected for the '%s' list."

This could be reworded to:

"At least one variable must be selected for the '%s' list."

or

"Variables must be selected for the '%s' list."

Checking the option **Strings used for both singular and plural** will warn about this issue.

3.6 Exposed Internal Strings

Strings that should not be translated should never be exposed to translators, even if context and translator comments are provided to warn them. In other words, if it shouldn't be translated, then it shouldn't be in the resources.

This can include internal strings being used to build syntax (e.g., HTML code), GUIDs, filenames, font names, debug messages, etc. This can also include URLs and email addresses, unless translators need to provide different contact information (for regional offices, for example).

Checking the options **Translatable strings that shouldn't be**, **Translatable strings being used in debug functions**, and **Translatable strings that contain URLs, email addresses, or phone numbers** will help with detecting these issues.

3.7 Comparing Translator Strings

When find and comparison functions contain a literal, localizable string, this usually indicates that this string is exposed for translation in multiple places. This can be a problem for a couple of reasons. The first is that it assumes translators will localize these strings the same way. Although tools like *gettext* will usually combine multiple instances of the same string, it won't if any of them have a unique context added to them. By design, these different instances will be presented separately and translators may translate them differently. Consider the following example:

```
...
else if (selectedTest == _(L"Dolch Sight Words"))
    {
    wxGetApp().GetMainFrame()->DisplayHelp(L"online/reviewing-standard-projects.html");
    }
```

The phrase "Dolch Sight Words" will be extracted as a localizable string from this code. Most likely, there is another instance of this string exposed for translation elsewhere. The issue is, if that other instance has context or translator comments added to it, then it will be treated as a separate resource. When the catalog is translated, if the two entries for this string are translated differently, then this comparison will fail at runtime.

The second issue is that this assumes all instances of a string will always match in the program. If any instance changes, however, then it will break not only localized products, but the source product as well. Consider the following example:

```
static wxString FormatClozeValuesLabel(const wxString& testName)
    {
    return testName + L" " + _(L"(predicted cloze scores)");
    }
...
static wxString StripClozeValuesLabel(const wxString& testName)
    {
    wxString begin;
    if (testName.EndsWith(_(L"(pred. cloze scores)"), &begin))
        {
        begin.Trim();
        return begin;
        }
```

Here we can see that a function is looking for the phrase "(pred. cloze scores)" and removing it if it finds it. There is another function that builds a string and presumably appends "(pred. cloze scores)" to it. Note, however, that this other function must have been changed and now appends "(predicted cloze scores)". Because of this, the source (e.g., English) version of the program (along with localized versions) will all break.

The recommended solution is to make this an extractable string from only one location. Then, this one resource can be used from all other locations in the program. This will ensure that if this string needs to be changed or have context added to it, it won't negatively impact the program. For example, this string could be returned from a common function:

```
static wxString GetPredClozeLabel()
    { return _(L"(predicted cloze scores)"); }
...
    return testName + L" " + GetPredClozeLabel();
...
    if (testName.EndsWith(GetPredClozeLabel(), &begin))
```

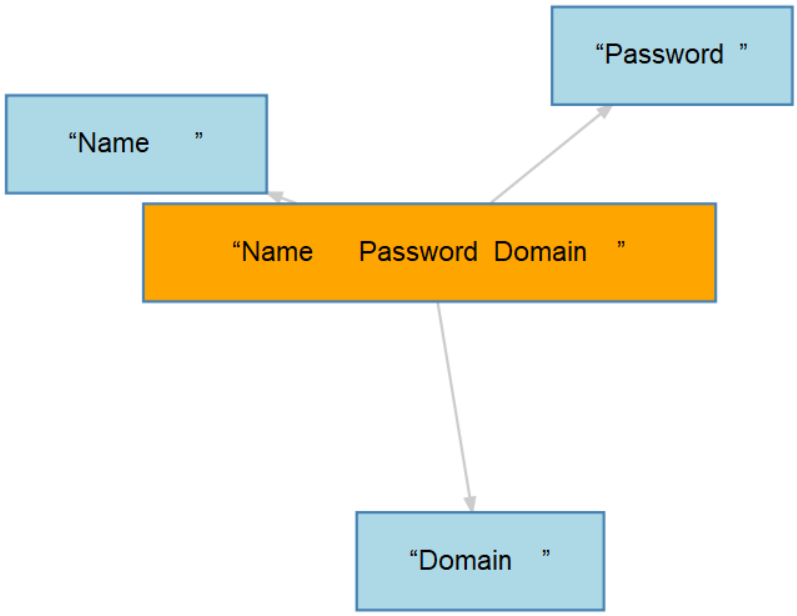
3.8 Multipart Strings

Multipart strings (or "mega strings") are single string resources that are split at runtime into smaller strings (Madden 65-66).

For example, consider a resource string such as:

"Name Password Domain "

While this is one string, it contains suspicious looking blocks of spaces in between each word. As it turns out, an application may be splitting this string into 10-character sections at runtime:



Then, it will use each of these sections as a separate resource. This is a bad practice found in legacy software, and can be thought of as the opposite of concatenating strings (another bad i18n practice).

The issue with this practice is that it forces translations to a constrained length (10-characters for each word in this example). Next, it is difficult to translate, as the translator must count the characters and spaces for each word segment, not just the entire string. (And this is assuming that the behavior of this string was even communicated to the translators.) Finally, l10n quality assurance tools are not designed to check unusual strings like this. Validating translations for these resources requires custom tools, which creates unnecessary technical debt.

If a translator doesn't format each segment of this string perfectly, then at best the translations will overlap or be clipped at runtime. At worst, the program will crash in situations where the full string is less than 30 characters.

It is preferred to use separate resources for each string. This ensures that the translations aren't constrained to arbitrary lengths and standard QA tools will be able to review the resources.

Checking the option **Multipart strings ("mega strings")** will warn about any possible multipart strings.

Part II

Command-line Utility

Chapter 4

Options

`Quneiform` accepts the following arguments:

input

The folder (or file) to analyze.

--enable

The following checks can be toggled via the `--enable` argument.

- `allI18N`

Perform all internationalization checks (the default).

- `suspectL10NString`

Check for translatable strings that should not be (e.g., numbers, keywords, `printf()` commands).

- `suspectL10NUsage`

Check for translatable strings being used in internal contexts (e.g., debugging and console functions).

- `suspectI18NUsage`

Check for suspect usage of i18n functions.

- `urlInL10NString`

Check for translatable strings that contain URLs, email addresses, or phone numbers.

It is recommended to dynamically format these into the string so that translators do not have to manage them.

- `excessiveNonL10NContent`

Check for translatable strings that contain large blocks on non-translatable content.

- `concatenatedStrings`

Check for strings that may be concatenated at runtime, which does not follow best i18n practices. This is generally accomplished by detecting strings that have leading or trailing spaces. (This is limited to actual space characters, and does not include tabs or newlines.)

> **Note**
> An exception is made for strings ending with ":"; here, it is assumed that this is formatting tabular data, rather than piecing a message together.

- `literalL10NStringCompare`

Check for literal, localizable strings being compared or searched for.

- `L10NStringNeedsContext`

Check for ambiguous strings that lack a translator comment.

Translator comments can be added to *gettext* functions by either inserting a tagged comment in front of them or by using a variant with a context parameter (e.g., `wxTRANSLATE_IN_CONTEXT()`). For example:

```
ListFiles(wxString::Format(
        /* TRANSLATORS: %s is folder name */ _(L"Searching %s"),
        get_selected_dir()));
```

By default, *gettext* will extract comments beginning with "TRANSLATORS:" in front of a resource function as a comment for the translator.

Strings are considered to be ambiguous if they meet any of the following criteria:

- A single word that contains 32 characters or more
- A single word that is uppercased (e.g., "FACTOR_VAR")
  - Punctuation is ignored for this check
- A single word that contains more than one punctuation mark (e.g., "*Print*")
  - Note that hyphens, slashes, periods, and ampersands are ignored for this check
- Contains three or more `printf` commands
- Contains at least one `printf` command and is less than 16 characters
- `notL10NAvailable`

Check for strings not exposed for translation.

- `deprecatedMacro`

Check for deprecated text macros and functions.

This will detect the usage of functions that are no longer relevant, and provide a suggested replacement.

For example, the `TCHAR` functions and macros (Microsoft, Working with Strings; Dunn) used in Win32 programming (e.g., `_TEXT`, `_tcscmp`) to help target Windows 98 and NT are no longer necessary. `Quneiform` will recommend how to remove or replace these.

- `nonUTF8File`

Check that files containing extended ASCII characters are UTF-8 encoded.

UTF-8 is recommended for compiler portability.

- `UTF8FileWithBOM`

Check for UTF-8 encoded files which start with a BOM/UTF-8 signature.

It is recommended to save without the file signature for best compiler portability.

This is turned off by default.

- `unencodedExtASCII`

Check for strings containing extended ASCII characters that are not encoded.

This is turned off by default.

- `printfSingleNumber`

Check for `printf()`-like functions being used to just format a number.

In these situations, it is recommended to use the more modern `std::to_[w]string()` function.

This is limited to integral values; `printf()` commands with floating-point precision will be ignored.

- `dupValAssignedToIds`

Check for the same value being assigned to different ID variables.

This check is performed per file; the same ID being assigned multiple times, but within separate files, will be ignored.

This is turned off by default.

- `numberAssignedToId`

Check for ID variables being assigned a hard-coded number.

It may be preferred to assign framework-defined constants (e.g., `wxID_HIGHEST`) to IDs.

This is turned off by default.

- `malformedString`

Check for malformed syntax in strings (e.g., malformed HTML tags).

- `fontIssue`

Check for font issues.

This is performed on Windows *.rc files and checks for dialogs that use unusual font sizes or are not using 'MS Shell Dlg'.(Microsoft, Using MS Shell Dlg and MS Shell Dlg 2)

- `allL10N`

Perform all localization checks (the default).

- `printfMismatch`

Check for mismatching `printf()` commands between source and translation strings.

This is performed on *gettext* *.po files and will analyze format strings for the following languages:

- C/C++

> **⚠ Warning**
> The checks performed here are strict; all `printf()` commands in translations must match their source counterpart exactly. For example, "*%lu*" vs. "*%l*" will emit a warning. Questionable commands such as "*% s*" (space is only meant for numeric formatting) will also emit a warning.

- `acceleratorMismatch`

Check for mismatching keyboard accelerators between source and translation strings. This is performed by checking that both strings contain one '&' followed by an alphanumeric character.

This is performed on *gettext* *.po files.

- `transInconsistency`

Check for inconsistent casing or trailing punctuation, spaces, or newlines between source and translation strings.

This is performed on *gettext* *.po files.

- `numberInconsistency`

Check for mismatching numbers between the source and target strings.

This is performed on *gettext* *.po files.

- `halfWidth`

Check for halfwidth Kanas, Hanguls, and punctuation in source and target strings.

This is performed on *gettext* *.po files.

- `multipartString`

Check for strings that appear to contain multiple parts that are being sliced at runtime.

- `pluralization`

Check for strings being used for both singular and plural that should be use different variations.

- `articleOrPronoun`

Check for strings with an article (e.g., "the," "a") in front of a formatting placeholder. Also checks for pronouns being used as individual strings.

- `lengthInconsistency`

Check for suspect lengths of translations compared to their source strings.

This is performed on *gettext* *.po files.

- `allCodeFormatting`

Check all code formatting issues (see below).

> **Note**
> These are not enabled by default.

- `trailingSpaces`

Check for trailing spaces at the end of each line.

- `tabs`

Check for tabs. (Spaces are recommended as tabs may appear differently between editors.)

- `wideLine`

Check for overly long lines.

- `commentMissingSpace`

Check that there is a space at the start of a comment.

--disable

Which checks to not perform. (Refer to options available above.) This will override any options passed to `--enable`.

--log-l10n-allowed

Whether it is acceptable to pass translatable strings to logging functions. Setting this to `false` will emit warnings when a translatable string is passed to functions such as `wxLogMessage`, `g_message`, or `qCWarning`.

(Default is `true`.)

--punct-l10n-allowed

Whether it is acceptable for punctuation only strings to be translatable.

Setting this to `true` will suppress warnings about strings such as "*?* - *?*" being available for localization.

(Default is `false`.)

--exceptions-l10n-required

Whether to verify that exception messages are available for translation.

Setting this to `true` will emit warnings when untranslatable strings are passed to various exception constructors or functions (e.g., `AfxThrowOleDispatchException`).

(Default is `true`.)

--min-l10n-wordcount

The minimum number of words that a string must have to be considered translatable.

Higher values for this will result in less strings being classified as `notL10NAvailable` warnings.

(Default is `2`.)

> **Note**
> Words in this context are values that contain at least one letter; numbers, punctuation, and syntax commands are ignored. For example, "Printing %s," "Player 1," and "Add +" will be seen as one-word strings.

--cpp-version

The C++ standard that should be assumed when issuing deprecated macro warnings (`deprecatedMacro`).

For example, setting this to `2017` or higher (along with enabling the `verbose` flag) will issue warnings for `WXUNUSED`, as it can be replaced with `[[maybe_unused]]`. If setting this to `2014`, however, `WXUNUSED` will be ignored since `[[maybe_unused]]` requires C++17.

(Default is `2014`.)

--fuzzy

Whether to review fuzzy translations.

(Default is `false`.)

-i,--ignore

Folders and files to ignore (can be used multiple times).

> **Note**
> Folder and file paths must be absolute or relative to the folder being analyzed.

-o,--output

The output report path, which can be either a CSV or tab-delimited text file. (Format is deteremined by the file extention that you provide.)

Can either be a full path, or a file name within the current working directory.

-q,--quiet

Only print errors and the final output.

-v,--verbose

Perform additional checks and display debug information and display debug information.

-h,--help

Print usage.

Chapter 5

Examples

The following example will analyze the folder "wxWidgets/src" (but ignore the subfolders "expat" and "zlib"). It will only check for suspect translatable strings, and then send the output to "results.txt".

Listing 5.1 `Terminal`

```
quneiform wxWidgets/src -i expat,zlib --enable=suspectL10NString -o results.txt
```

This example will only check for `suspectL10NUsage` and `suspectL10NString` and not show any progress messages.

Listing 5.2 `Terminal`

```
quneiform wxWidgets/samples -q --enable=suspectL10NUsage,suspectL10NString
```

This example will ignore multiple folders (and files) and output the results to "results.txt."

Listing 5.3 `Terminal`

```
quneiform src --ignore=easyexif,base/colors.cpp,base/colors.h -o results.txt
```

Chapter 6

Reviewing Output

After building, go into the "bin" folder and run this command to analyze the sample file:

Listing 6.1 `Terminal`

```
$ quneiform ../samples -o results.txt
```

This will produce a "results.txt" file in the "bin" folder which contains tabular results. This file will contain warnings such as:

- `suspectL10NString`: indicates that the string "*GreaterThanOrEqualTo*" is inside of a `_()` macro, making it available for translation. This does not appear to be something appropriate for translation; hence the warning.
- `suspectL10NUsage`: indicates that the string "*Invalid dataset passed to column filter.*" is being used in a call to `wxASSERT_MSG()`, which is a debug assert function. Asserts normally should not appear in production releases and shouldn't be seen by end users; therefore, they should not be translated.
- `notL10NAvailable`: indicates that the string "*'%s': string value not found for '%s' column filter.*" is not wrapped in a `_()` macro and not available for localization.
- `deprecatedMacro`: indicates that the text-wrapping macro `wxT()` should be removed.
- `nonUTF8File`: indicates that the file contains extended ASCII characters, but is not encoded as UTF-8. It is generally recommended to encode files as UTF-8, making them portable between compilers and other tools.
- `unencodedExtASCII`: indicates that the file contains hard-coded extended ASCII characters. It is recommended that these characters be encoded in hexadecimal format to avoid character-encoding issues between compilers.

To look only for suspect strings that are exposed for translation and show the results in the console window:

Listing 6.2 `Terminal`

```
$ quneiform ../samples --enable=suspectL10NString,suspectL10NUsage
```

To look for all issues except for deprecated macros:

Listing 6.3 `Terminal`

```
$ quneiform ../samples --disable=deprecatedMacros
```

By default, `Quneiform` will assume that messages inside of various exceptions should be translatable. If these messages are not exposed for localization, then a warning will be issued.

To consider exception messages as internal (and suppress warnings about their messages not being localizable) do the following:

Listing 6.4 `Terminal`

```
$ quneiform ../samples --exceptions-l10n-required=false
```

Similarity, `Quneiform` will also consider messages inside of various logging functions to be allowable for translation. The difference is that it will not warn if a message is not exposed for translation. This is because log messages can serve a dual role of user-facing messages and internal messages meant for developers.

To consider all log messages to never be appropriate for translation, do the following:

Listing 6.5 `Terminal`

```
$ quneiform ../samples --log-l10n-allowed=false
```

To display any code-formatting issues, enable them explicitly:

Listing 6.6 `Terminal`

```
$ quneiform ../samples --enable=trailingSpaces,tabs,wideLine

or

$ quneiform ../samples --enable=allCodeFormatting
```

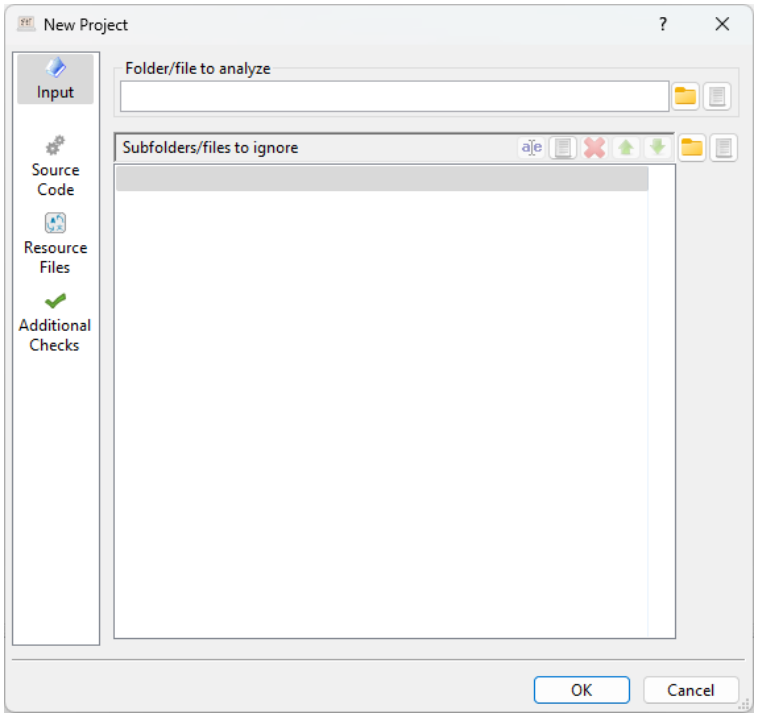
Part III

User Interface

Chapter 7

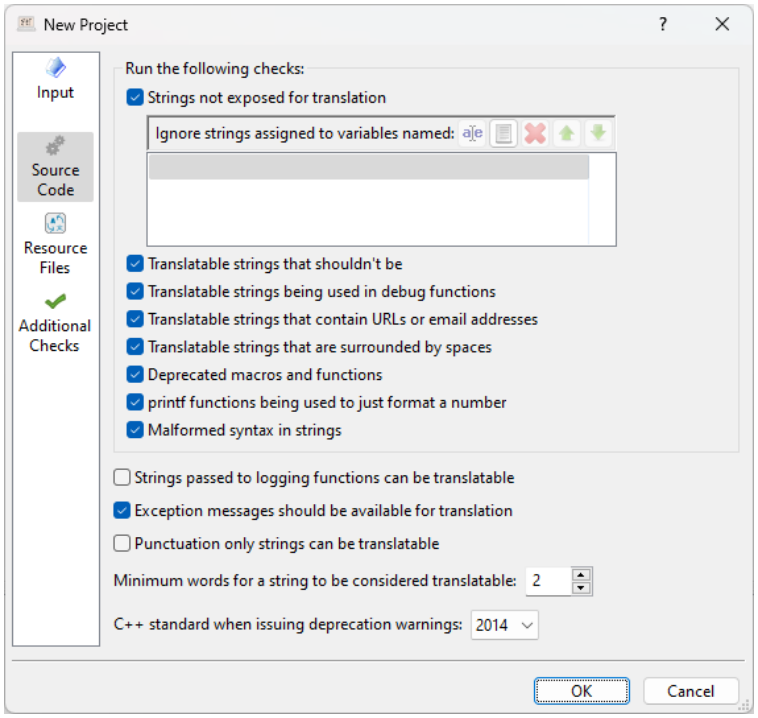
Working with Projects

To create a new project, click the New button on the ribbon. The **New Project** dialog will appear:

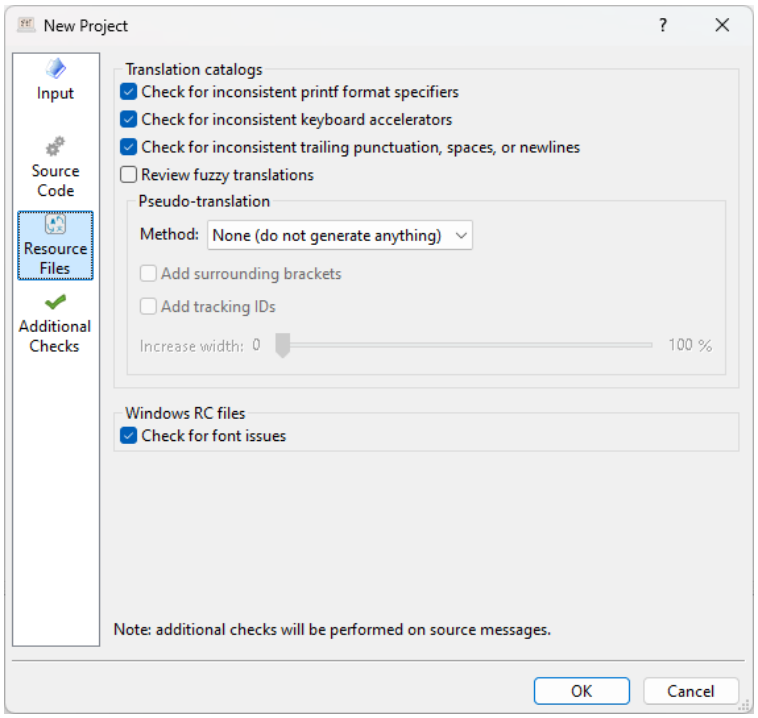


First, select which folder (or file) you wish to analysis. If analyzing a folder, you can also optionally select subfolders and files that should be ignored.

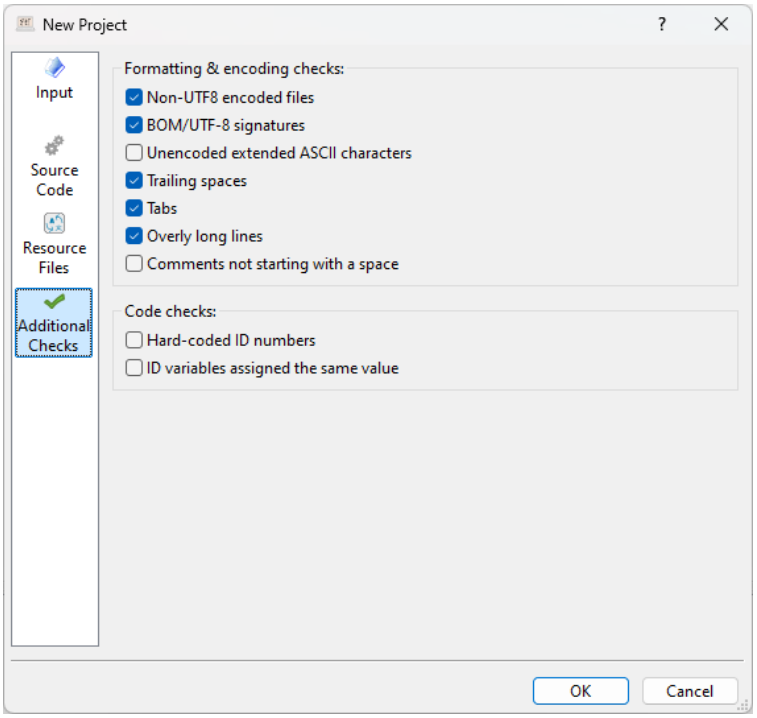
Next, select which source code checks to perform from the **Source Code** page.



Next, options for performing checks on translation catalogs and creating pseudo-translated resources are available on the **Resource Files** page:



Finally, the **Additional Checks** page provides checks unrelated to i18n or l10n, such as code formatting and file encoding issues:



Once finished selecting your options, click OK to create the project.

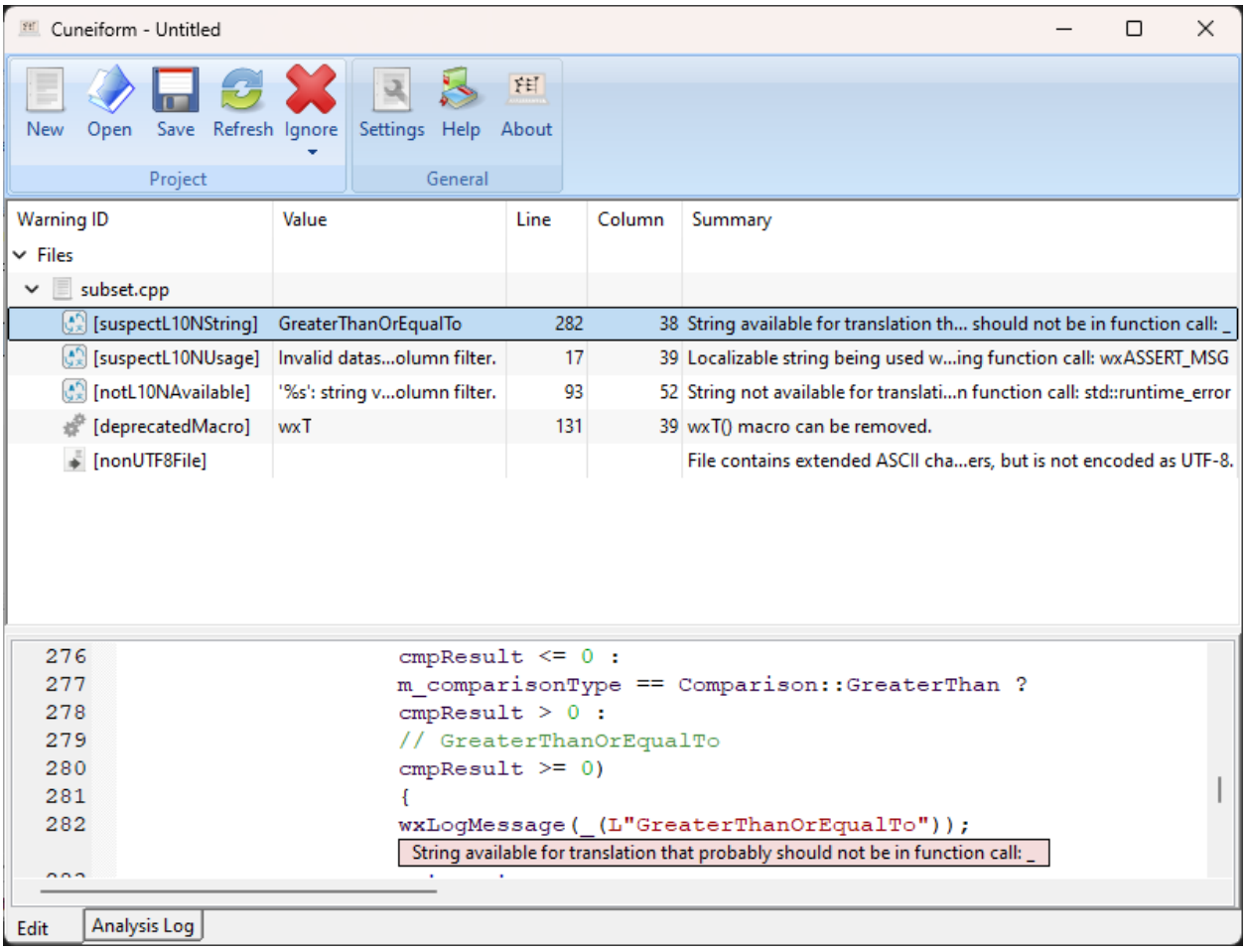
After opening or creating a project, you can reanalyze the folder at any time by clicking Refresh on the ribbon. The **Edit Project** dialog will appear, where you will be able to change any option before re-running the analysis.

When you are finished reviewing a project, click Save to save it. The project file will remember the folder you were reviewing, along with your current settings. To reopen this project, click the Open button on the ribbon and select the project file.

Chapter 8

Reviewing Output

After opening a project, select any warning message in the top window and the associated file will be displayed beneath it. The line where the issue was detected will be scrolled to, along with the warning shown underneath it.



From here, you can edit the file in this window to correct any issues. Once you are finished editing, either select another file's warning in the top window or click the Save button on the ribbon. (The latter will also save the project if it has unsaved changes.)

8.1 Ignoring Output

Specific warnings and files can be ignored from the project window.

To ignore a file, select it in the top window and click the Ignore button on the ribbon. From the menu that appears, select the **Ignore '*fileName*'** option (where *fileName* is the file name) and this file will be removed from the project.

To ignore a class of warnings, select a warning for any file in top window and click the Ignore button on the ribbon. From the menu that appears, select the **Ignore '*warningId*'** option (where *warningId* is the warning, such as "[notL10NAvailable]"). All instances of this warning will be removed from all files. Also, this check will no longer be performed when you re-run the analysis.

> **Note**
> These options are also available by right clicking any warning and selecting them from the popup menu.

8.2 Exporting Results

To export all warnings, click the Save button on the ribbon and select **Export Results**. From here, you can export the results to either a CSV or tab-delimited text file.

Chapter 9

Editing

When a source file in the loaded in the editor and the editor is selected, the **Edit** section of the ribbon will become enabled. From here, you can edit the source file in the **Editor** and make use of these options from the ribbon.

9.1 Basic Operations

Clipboard operations (e.g., Cut, Copy, and Paste), Undo, Redo, and Select All are all available from this section.

9.2 Insert

From the Insert button, options are available for inserting and converting content within the editor to fix various warnings.

Translator Comment

To fix warnings about strings needing context, place the cursor into the editor in front of a resource loading call (e.g., `_("TABLE")`). Next, click the Insert button on the ribbon and select **Translator Comment...**. Then, select which translator comment style that you wish to use. (*gettext* and *Qt* styles are available.) Enter a comment that will explain to the translators what "TABLE" means in this context, providing guidance for how they should translate it. Click OK and your comment with the comment style that you selected (e.g., "// TRANSLATORS:") will be inserted in front of the function call.

> **Note**
> This warning is emitted when the option **Check for ambiguous strings that lack a translator comment** (from the **Source Code** settings) is checked.

Encode Extended ASCII Characters

To fix warnings about extended ASCII characters needing to be encoded, highlight the text in the editor which contains these characters. Next, click the Insert button on the ribbon and select **Encode Extended ASCII Characters...**. You will then be prompted about how the selection will be re-encoded. Click OK and the selection will be replaced with the re-encoded content.

As an example, if you have the string “Błąd” (Polish for “Error”) in a source file, some legacy compilers may have difficulty processing it (even if encoded as UTF-8). By following the aforementioned steps, this will be re-encoded to “B\U00000142\U00000105d.”

> **Note**
> This warning is emitted when the option **Unencoded extended ASCII characters** (from the **Additional Checks** settings) is checked.

Make Selection Translatable

To make a string translatable, highlight the string in the editor and select **Make Selection Translatable...** from the Insert button. From the **Make Selection Translatable** dialog, options are available for choosing which translation function to use, along with providing a domain or context (if applicable). The translation marking functions available are compatible with *gettext*, *wxWidgets*, *Qt*, and *KDE*.

Mark Selection as Non-translatable

To mark a string as non-translatable, highlight the string in the editor and select **Mark Selection as Non-translatable...** from the Insert button. From the **Mark Selection as Non-translatable** dialog, options are available for which function to wrap the string in, along with providing an optional context and comment. These functions will mark the string as non-translatable and **Quneiform** will ignore them during an analysis. Additionally, these functions and comments decorating a string will inform developers that the string is not meant for translation and why.

Suppress Warnings for Selection

To suppress warnings for the selected block of code, highlight the string in the editor and select **Suppress Warnings for Selection...** from the Insert button. From the **Suppress Warnings for Selection** dialog, options are available for which analysis program to suppress warnings from, along with which specific warnings to suppress (if relevant). After selecting the application to suppress and clicking OK, the selected code in the editor will be wrapped in the appropriate suppression comments.

Along with *Quneiform*, warning suppression comments are also available for other linting tools such as *Clang-format*, *Clang-tidy*, and *Cpp-check*.

Chapter 10

Tools

A selection of tools are available for examining and converting text from the **Tools** section of the ribbon.

10.1 String Info

The **String Info** dialog provides information about a string. This includes information such as string length and which (if any) extended ASCII chararcters it contains.

10.2 Convert String

The **Convert String** dialog provides the ability to convert portions of a string to different character sets. As an example, this can be useful for translating Latin-1 (i.e, 7-bit ASCII) into Japanese (i.e., full-width) numbers.

> **Note**
> If text is selected in the editor, then that will be used as the initial source string when opening these dialogs.

Chapter 11

Settings

11.1 Input

The following options are available when creating or editing a project.

Folder/file to analyze: enter into here the folder or file to analyze.

Subfolders/files to ignore: enter into this list the folders and files to ignore.

> **Tip**
> Folder and file paths being ignored must be absolute or relative to the folder being analyzed.

11.2 Source Code

The following options are available for C/C++ source files.

Run the following checks

Strings not exposed for translation: select this to check for strings hard coded in source code that not exposed for translation (i.e., embedded strings).

Ignore strings assigned to variables named: when finding strings not exposed for translation, strings can be ignored if assigned to variables from this list. For example, if a list of color names are assigned to a variable named `colorMode` that you wish to ignore, then add "colorMode" to this list. You can enter variable names (e.g., "`colorMode`") or regular expressions (e.g., "`(RELEASE|DEBUG)check[0-9]?`") here.

> **Note**
> Adding a new entry here for a project will also add it to the global settings. This will ensure that you don't have to re-enter these values for future projects.

Refer to Section 3.1 for an overview of this issue.

Translatable strings that shouldn't be: select this to check for translatable strings that should not be (e.g., numbers, keywords, `printf()` commands).

Refer to Section 3.6 for an overview of this issue.

Translatable strings being used in debug functions: select this to check for translatable strings being used in internal contexts (e.g., debugging and console functions).

Translatable strings that contain URLs, email addresses, or phone numbers: select this to check for translatable strings that contain URLs, email addresses, or phone numbers. For situations such as a regional offices needing to use their own contact information in a localized product, it is recommended to leave this unchecked. Otherwise, it is recommended perform this check and to dynamically format these items into the string so that translators do not have to manage them.

Translatable strings that contain an excessive amount of non-translatable content: select this to check for translatable strings that contain large blocks on non-translatable content. As an example, this will detect strings with large blocks of HTML tags that may make the string complicated to translate. In this case, it may be recommended to format the string, replacing the HTML content with placeholders (e.g., `printf` commands). Another option is to concatenate the HTML tags with the localizable string at runtime.

> **Tip**
> Adding a context to the string resource (e.g., a translator comment) will suppress this warning.
>
> For example, a resource such as "%d of %zu" will emit this warning. Although not much can be done about the majority of this string being `printf` commands, a translator comment would help with explaining what they represent. By adding a comment, this warning (along with any ambiguity warnings) will be removed.

Run the following checks (*continued...*)

Concatenated strings: select this to check for translatable strings that start or begin with spaces. (This is limited to actual space characters, and does not include tabs or newlines.)

These may be strings that are concatenated at runtime, which does not follow best i18n practices. It is recommended to format separate values into a single string via a `printf` formatting function, rather than piecing them together.

> **Note**
> An exception is made for strings that end with a colon followed by a space (":"). It is assumed that this part of tabular data being concatenated and will be ignored.

Refer to Section 3.2 for an overview of this issue.

Literal, localizable strings being compared or searched for: select this to check for literal, localizable strings being compared or searched for. This usually indicates a string being marked for localization in multiple places and the program assuming they will all be translated the same way. This is a dangerous assumption and can likely break the program.

Instead, it is recommended to expose the string for localization from one location and use that throughout the program. For example, the string could be defined as a constant variable or returned from a common function.

Refer to Section 3.7 for an overview of this issue.

Multipart strings ("mega strings"): select this to check for strings that appear to contain multiple parts that are being sliced into separate strings at runtime.

Refer to Section 3.8 for an overview of this issue.

This check should be ran when modernizing legacy software.

Strings used for both singular and plural: select this to check for strings being used for both singular and plural variations.

Refer to Section 3.5 for an overview of this issue.

Articles/pronouns mixed with dynamic content: select this to check for strings with an article (e.g., "the," "a") in front of a formatting placeholder. Additionally, this will check for pronouns being used as individual strings.

Refer to Section 3.3 and Section 3.4 for an overview of this issue.

Run the following checks (*continued...*)

Suspect i18n usage: select this to check for suspect usage of i18n functions and project settings.

This will check for issues such as:

- Usage of the Win32 function `::LoadString()`.
  It is recommended to use other functions (e.g., `CString::LoadString()`) that are not limited to fixed buffer sizes; otherwise, translated strings may become truncated.
- Context arguments passed to l10n functions (e.g., `tr()`) that are longer than 32 characters.
  This may indicate that the context and translatable string arguments are reversed.
- String literals being passed to `_()` and `wxPLURAL()`.
- Possible strings meant for translation being passed to ID loading functions (e.g., `qtTrId()`).
- Using two-digit year specifiers ('%g', '%y', and '%C') in `strftime`-like functions.
  This causes ambiguity between years and months/days for end users and is also not Y2K compliant.
- macOS Info.plist files not including any `CFBundleLocalizations` keys.
  Without these keys, translations will not be loaded by the application at runtime.
- *wxWidgets* projects not calling `wxUILocale::UseDefault()`.
- *wxWidgets* projects not constructing a `wxLocale` object with the user's locale.
  `wxLocale` objects are only necessary if your project relies on C runtime functions to use the user's locale. This is only detected if **Include verbose warnings** is checked.

Check for ambiguous strings that lack a translator comment: select this to check for possibly ambiguous strings that lack a translator comment. *gettext*- and *Qt*-style comments are searched for, which includes formats such as "// TRANSLATORS:" and "//:".

A string will be considered ambiguous if:

- A single word in all caps.
- A single word with multiple punctuation marks embedded in it.
- It contains placeholder text (e.g., "###").
- It contains more than one abbreviation. (Abbreviations should be avoided, or at least the full words should be shown in a translator comment.)
- A short string with a `printf` or positional argument (e.g., "%2").
- A longer string with multiple `printf` or positional arguments.

> **Note**
> `printf` or positional arguments will be ignored if following a colon and space. For example, a string such as "Name: %s\nDate: %s\Location: %s" may not need a comment. It is assumed that the label in front of the colon serves as a description for the following "%s"; thus, a warning will not be emitted.

Deprecated macros and functions: select this to check for deprecated text macros and functions. This will detect the usage of functions that are no longer relevant, and provide a suggested replacement.

For example, the `TCHAR` functions and macros used in Win32 programming (e.g., `_TEXT`, `_tcscmp`) to help target Windows 98 and NT are no longer necessary. `Quneiform` will recommend how to remove or replace these.

printf functions being used to just format a number: select this to check for `printf()`-like functions being used to just format a number. In these situations, it is recommended to use the more modern `std::to_[w]string()` function. This is limited to integral values; `printf()` commands with floating-point precision will be ignored.

Malformed syntax in strings: select this to check for malformed syntax in strings (e.g., malformed HTML tags).

Strings passed to logging functions can be translatable: select this if it should be acceptable to pass translatable strings to logging functions. Setting this to `false` will emit warnings when a translatable string is passed to functions such as `wxLogMessage`, `g_message`, or `qCWarning`.

Exception messages should be available for translation: select this to verify that exception messages are available for translation. Setting this to `true` will emit warnings when untranslatable strings are passed to various exception constructors or functions (e.g., `AfxThrowOleDispatchException`).

Punctuation only strings can be translatable: select this if it should be acceptable for punctuation only strings to be translatable. Setting this to `true` will suppress warnings about strings such as "*?* - *?*" being available for localization.

Minimum words for a string to be considered translatable: enter into here the minimum number of words that a string must have to be considered translatable. Higher values for this will result in less strings being classified as `notL10NAvailable` warnings.

> **Note**
> Words in this context are values that contain at least one letter; numbers, punctuation, and syntax commands are ignored. For example, "Printing %s," "Player 1," and "Add +" will be seen as one-word strings.

C++ standard when issuing deprecation warnings: enter into here the C++ standard that should be assumed when issuing deprecated macro warnings (`deprecatedMacro`). For example, setting this to `2017` or higher (along with selecting the **Include verbose warnings** option) will issue warnings for `WXUNUSED`, as it can be replaced with `[[maybe_unused]]`. If setting this to `2014`, however, `WXUNUSED` will be ignored since `[[maybe_unused]]` requires C++17.

Include verbose warnings: select this to perform additional checks, including those that may not be related to i18n/l10n issues. This will also include additional information in the **Analysis Log** window.

11.3 Resource Files

The following options are available for resource files (i.e., *.po and *.rc).

Translation catalogs

Check for inconsistent printf & positional format specifiers: select this to check for mismatching `printf()` and positional commands between source and translation strings. (This will review "c-format," "kde-format," and "qt-format" strings in *gettext* files.)

> **Warning**
> The checks performed here are strict; all `printf()` commands in translations must match their source counterpart exactly. For example, "*%lu*" vs. "*%l*" will emit a warning. Questionable commands such as "*% s*" (space is only meant for numeric formatting) will also emit a warning.

Check for inconsistent keyboard accelerators: select this to check for mismatching keyboard accelerators between source and translation strings. This is performed by checking that both strings contain one '&' followed by an alphanumeric character.

Check for inconsistent casing, separators, trailing punctuation, spaces, or newlines: select this to check for the following between source and translation strings:

- Mismatching trailing punctuation, spaces, or newlines,
- The source string starting with an uppercase letter and the translation starting with a lowercase letter.
- Mismatching number of "\t" and "|" symbols.

In regards to punctuation, terminating characters such as periods, exclamation points, question marks, and colons are reviewed. For these, warnings will only be emitted if the source string has a terminating mark and the translation does not have either a terminating mark or closing parenthesis. All types of terminating marks are treated interchangeably, as not all languages commonly use question and exclamation marks as much as English.

Check for inconsistent numbers: select this to check for mismatching numbers between the source and target strings.

Devanagari and full-width numbers will be converted to 7-bit ASCII numbers during the comparison. This ensures that "8859-1" and "`８８５９-１`" will be seen as the same number sequence.

> **Note**
> Numbers will have all commas, periods, and non-breaking spaces removed when being compared. This is done to normalize any differences in locale-specific formatting and provide more accurate comparisons. Because of this, the list of numbers shown in these warnings will not include any decimal or thousands separators.
>
> Also, the sequence of mismatching numbers in the warning message will be displayed in lexicographical order, not the order that they appear in the strings. It is assumed that the ordering of numbers can change in the translation; thus, the ordering of numbers is normalized and shown in this way in the warning.

Translation catalogs (*continued...*)

Check for halfwidth characters: select this to check for halfwidth Kanas, Hanguls, and punctuation in source and target strings.

Halfwidth characters are generally only needed in legacy applications, where translations are limited to certain lengths. For example, applications that require a translation to not exceed its source string's length may need to resort to halfwidth Kanas (e.g., "ｼ") to accomplish this. For modern applications, it is preferred to use the fullwidth counterparts (e.g., "シ"), as it appears more professional.

Including this check can be useful when updating translations for legacy applications undergoing modernization.

Check for suspect translation lengths: select this to check for translations that are suspiciously longer than their source strings.

The default is 400%, meaning that a translation can be four times longer than the source string before emitting a warning. This value can be changed by adjustsing the **How much translations can expand** slider.

Tip
Setting this to 0% will instruct the program to warn if any translation exceeds the length of its source string by even one character. This is useful for legacy software where resource-loading routines are using the source strings' hardcoded lengths.

Review fuzzy translations: select this to review fuzzy translations. This should only be selected if you intend to review translations that still require approval and are likely unfinished.

Translation catalogs (*continued...*)

Pseudo-translation

While analyzing translation catalogs, copies of them can also be created and be pseudo-translated. Later, these catalogs can be loaded by your application for integration testing.

Method

This option specifies how to generate pseudo-translations.

None (do not generate anything): instructs the program to not generate any files.

UPPERCASE: translations will be uppercased variations of the source strings.

Fill with 'X'es: letters from the source string will be replaced with either 'x' or 'X' (matching the original casing). This will produce the most altered appearance for the pseudo-translations.

During integration testing, these pseudo-translations will be easier to spot, but will make navigating the UI more difficult. This is recommended for the quickest way to interactively find UI elements that are not being made available for translation.

European: letters and numbers from the source string will be replaced with accented variations from European alphabets. The translation will be clearly different, but still generally readable.

Cherokee: letters and numbers from the source string will be replaced with Cherokee characters. (Some numbers will be replaced with full-width numbers when a similiar looking Cherokee character isn't available.) Like **European**, translations will be clearly different, but somewhat readable.

Add surrounding brackets: select this option to add brackets around the pseudo-translations. This can help with identifying truncation and strings being pieced together at runtime.

Add tracking IDs: select this to add a unique ID number (inside or square brackets) in front of every pseudo-translation. This can help with finding where a particular translation is coming from.

Expand/contract width: select how much wider and shorter (between -50–100%) to make pseudo-translations. For widening, strings are padded with hyphens on both sides. For shortening, as many characters from the translatable sections of the string will be removed to meet the requested percentage.

The default is a 40% increase, as "German and Spanish translations can sometimes take 40% more screen space than English" (Boyero).

Note
If surrounding brackets or tracking IDs are being included, then their lengths will be factored into the increased/decreased width calculation.

Windows RC files

Check for font issues: select this to check for dialogs that use unusual font sizes or are not using 'MS Shell Dlg'.

Note
Some static analysis options from the **Source Files** section will also be used while analyzing the source strings in these resource files.

11.4 Additional Checks

The following additional options are available for C/C++ source files. These options do not relate to internationalization, but are offered as supplemental checks for code formatting and other issues.

Formatting & encoding checks

Non-UTF8 encoded files: select this to check that files containing extended ASCII characters are UTF-8 encoded. UTF-8 is recommended for compiler portability.

BOM/UTF-8 signatures: select this to check for UTF-8 encoded files which start with a BOM/UTF-8 signature. It is recommended to save without the file signature for best compiler portability.

Unencoded extended ASCII characters: select this to check for strings containing extended ASCII characters that are not encoded.

Trailing spaces: select this to check for trailing spaces at the end of each line.

Tabs: select this to check for tabs. (Spaces are recommended as tabs may appear differently between editors.)

Overly long lines: select this to check for overly long lines.

Comments not starting with a space: select this to check that there is a space at the start of a comment.

Code checks

Hard-coded ID numbers: select this to check for ID variables being assigned a hard-coded number. It may be preferred to assign framework-defined constants (e.g., `wxID_HIGHEST`) to IDs.

ID variables assigned the same value: select this to check for the same value being assigned to different ID variables. This check is performed per file; the same ID being assigned multiple times, but within separate files, will be ignored.

For MFC projects, this will also check for MFC resource IDs that are outside of expected ranges. For example, IDs starting with `IDS_` (a string ID) should be between 1 and 0x7FFF.

Part IV

Additional Features

Chapter 12

Inline Suppression

12.1 Code Blocks

Warnings can be suppressed for blocks of source code by placing `quneiform-suppress-begin` and `quneiform-suppress-end` comments around them. For example:

```
// quneiform-suppress-begin
if (_debug && allocFailed)
    AfxMessageBox("Allocation failed!");
// quneiform-suppress-end
```

This will prevent a warning from being emitted about "Allocation failed" not being available for localization.

> **⚠ Warning**
> The comment style of the begin and end tags must match. For example, if using multi-line comments (i.e., `/**/`), then both tags must be inside of `/**/` blocks.

12.2 Individual Strings

To instruct the program that a particular string is not meant for localization, wrap it inside of a `_DT()` or `DONTTRANSLATE()` function call. By doing this, the program will not warn about it not being available for localization. For example, both strings in the following will be ignored:

```
if (allocFailed)
    MessageBox(DONTTRANSLATE("Allocation failed!"));
else
    WriteMessage(_DT("No memory issues"));
```

You can either include the file "donttranslate.h" (included with *Quneiform*) into your project to add these functions, or you can define your own macro:

```
#define DONTTRANSLATE(x) (x)
#define _DT(x) (x)
```

If using the versions provided in "donttranslate.h," additional arguments can be included to explain why strings should not be available for translation. For example:

```
const std::string fileName = "C:\\data\\logreport.txt";

// "open " should not be translated, it's part of a command line
auto command = DONTTRANSLATE("open ") + fileName;
// expands to "open C:\\data\\logreport.txt"

// a more descriptive approach
auto command2 = DONTTRANSLATE("open ", DTExplanation::Command) + fileName;
// also expands to "open C:\\data\\logreport.txt"

// an even more descriptive approach
auto command3 = DONTTRANSLATE("open ",
                              DTExplanation::Command,
                              "This is part of a command line, "
                              "don't expose for translation!") +
                fileName;
// also expands to "open C:\\data\\logreport.txt"

// a shorthand, _DT(), is also available
auto command = _DT("open ") + fileName;
```

Note

`_DT()` and `DONTTRANSLATE()` do not have any effect on the code and would normally be compiled out. Their purpose is only to inform `Quneiform` that their arguments should not be translatable.

Appendix A

Acknowledgements

Open-Source Libraries

The 3rd party, open-source libraries that *Quneiform* includes (or links with) are listed below (along with their respective licenses).

cxxopts

C++11 library for parsing command line options.

Copyright (c) 2014 Jarryd Beck

MIT License

UTF8-CPP

UTF-8 with C++ in a portable way.

Copyright 2006 Nemanja Trifunovic

Boost Software License 1.0

wxWidgets

A stable and powerful open source framework for developing native cross-platform GUI applications in C++.

Copyright (c) 1998-2005 Julian Smart, Robert Roebling et al

wxWindows Library Licence, Version 3.1

Artwork

Photo by Riccardo Trimeloni on Unsplash

Photo by Fionn Große on Unsplash

References

Boyero, Virginia. “Design and evolution of the localization pipeline in Snowdrop”. *MultiLingual*, vol. 31, no. 2, 2020, p. 37.

Calek, Sarah. “What’s in a game translator”. *MultiLingual*, vol. 30, no. 3, 2019, pp. 20–22.

cppreference.com. printf, fprintf, sprintf, snprintf, printf_s, fprintf_s, sprintf_s, snprintf_s. en.cppreference.com/w/c/io/fprintf.

Dunn, Michael. The Complete Guide to C++ Strings, Part I - Win32 Character Encodings. www.codeproject.com/Articles/2995/The-Complete-Guide-to-C-Strings-Part-I-Win32-Chara.

Haible, Bruno, and Daiki Ueno. *GNU gettext*, www.gnu.org/software/gettext/manual/gettext.html.

KDE. Development/Tutorials/Localization/i18n Mistakes. techbase.kde.org/Development/Tutorials/Localization/i18n_Mistakes.

———. Translations / i18n. develop.kde.org/docs/plasma/widget/translations-i18n/.

Madden, Blake. “Localization workarounds for non-internationalized software”. *MultiLingual*, vol. 30, no. 6, 2019, pp. 65–66.

Microsoft. printf_p positional parameters. learn.microsoft.com/en-us/cpp/c-runtime-library/printf-p-positional-parameters?view=msvc-170.

———. STRINGTABLE resource. learn.microsoft.com/en-us/windows/win32/menurc/stringtable-resource.

———. TN020: ID Naming and Numbering Conventions. learn.microsoft.com/en-us/cpp/mfc/tn020-id-naming-and-numbering-conventions?view=msvc-170.

———. Using MS Shell Dlg and MS Shell Dlg 2. learn.microsoft.com/en-us/windows/win32/intl/using-ms-shell-dlg-and-ms-shell-dlg-2.

———. Working with Strings. learn.microsoft.com/en-us/windows/win32/learnwin32/working-with-strings.

Pérez, Cristina, and Leticia Sáenz. “Gender-inclusive languages in games and its localization challenges”. *MultiLingual*, vol. 30, no. 3, 2019, pp. 32–35.

Qt Company, The. Writing Source Code for Translation. doc.qt.io/qt-5/i18n-source-translation.html.

wxWidgets. wxTranslations Class Reference. docs.wxwidgets.org/latest/classwx_translations.html.

Index