Internacionalización (I18N) y Localización (L10N)

Desarrollo Colaborativo de Aplicaciones

Introduccion

Un ejemplo ilustrativo

```
 1 void mostrar_error(const std::string& archivo) {
 2     std::cout << "Error: No se pudo abrir el archivo "
 3               << archivo << std::endl;
 4     std::cout << "Verifica que el archivo existe y tienes permisos."
 5               << std::endl;
 6 }
 7
 8 void mostrar_estadisticas(int total, int procesados) {
 9     std::cout << "Procesados " << procesados
10               << " de " << total << " archivos." << std::endl;
11     double porcentaje = (procesados * 100.0) / total;
12     std::cout << "Progreso: " << porcentaje << "%" << std::endl;
13 }
```

¿Se puede distribuir esta aplicación en internet?

Un ejemplo ilustrativo

El programa **no se puede distribuir porque**:

- Contiene cadenas en español *hardcodeadas*.
- Es imposible cambiar a inglés sin recompilar.
- El formato de números asume convención española.

¿Por qué internacionalizar?

Razones de negocio

- Alcance global
- Ventaja competitiva
- Cumplimiento normativo
- Mejor experiencia de usuario

Realidad del mercado

- Internet: **5.000M** usuarios
- **75%** no hablan inglés
- **72%** prefieren su idioma
- **40%** no compra en otros idiomas

¿Por qué internacionalizar?

La internacionalización no es un lujo, es una **necesidad competitiva** para garantizar el éxito de un producto que se distribuye en internet.

La realidad de la internacionalización

Queda claro que necesitamos **internacionalizar** nuestras aplicaciones. Sin embargo…

¿Ese trabajo nos corresponde a nosotros?

La realidad de la internacionalización

A no ser que tengamos dos grados (informática y traducción e interpretación), la internacionalización es una tarea que implica a especialistas de **la traducción** y **la localización** de textos.

¿Qué sucede con esto?

La realidad de la internacionalización

No debemos asumir **nunca** que los equipos interdisciplinares externos saben de informática y programación. Por lo tanto, **debemos trabajar bajo sistemas centralizados donde todos colaboren de forma cómoda y accesible**

Sobre esta sesión

Por lo tanto, en esta lección no solo aprenderemos a **cómo implementar un sistema de internacionalización**, sino a **cómo diseñar y trabajar con herramientas que permitan la intervención de equipos externos no especialistas**.

I18N y L10N

Internacionalización (I18N)

I18N = I + 18 letras + N

Proceso de **diseño y desarrollo** de *software* de manera que pueda **adaptarse fácilmente** a diferentes idiomas y regiones **sin requerir cambios en el código fuente**.

Internacionalización (I18N)

Sus principios fundamentales son:

1. **Separación de contenido**: Texto separado del código.
2. **Independencia cultural**: No asumir convenciones específicas.
3. **Soporte *Unicode***: Manejar cualquier sistema de escritura.
4. **Diseño flexible**: UI adaptable a diferentes longitudes de texto.

Internacionalización (I18N)

> **Nota**
>
> Los **desarrolladores se responsabilizan del I18N** durante todo el desarrollo de la aplicación.

Localización (L10N)

L10N = L + 10 letras + N

Proceso de **adaptar** una aplicación internacionalizada a un **idioma y cultura específicos**, incluyendo traducciones, formatos, convenciones culturales y requisitos legales.

Localización (L10N)

Los componentes principales de la localización

1. **Traducción**: Adaptar textos al idioma destino
2. **Formatos**: Números, fechas, monedas, medidas
3. **Cultura**: Colores, iconos, imágenes apropiadas
4. **Legal**: Cumplimiento normativo local
5. **UX**: Ajustar flujos a expectativas culturales

¿Traducir = Localizar?

> **Localizar no implica traducción literal**, sino reconocer esos referentes y adaptarlos adecuadamente a la cultura y el idioma propios del receptor.

¿Traducir = Localizar?

> Si en un juego hay expresiones de Reddit y 4chan, pues se adaptan culturalmente a, por ejemplo, expresiones de Forocoches o MeriStation. Hay que saber dónde están los límites y jugar con el idioma para realizar un buen trabajo, con textos naturales y cercanos, pero sin pasarse al extremo de que aparezca Carmen Sevilla en una serie de ambientación estadounidense.

¿Traducir = Localizar?

> Básicamente, no existen juegos que no deban o no puedan ser localizados. Y si el problema es con la imagen, ahí entra la colaboración con los desarrolladores para modificar las texturas o lo que sea y conseguir que funcione (algo que se hace mucho del japonés al inglés y viceversa, por ejemplo)

Ramón Méndez (2018)

Ejemplo de buena localización

Paper Mario: La puerta milenaria (2004)

Ejemplo de buena localización

Paper Mario: La puerta milenaria (2004)

Localización (L10N)

> **Nota**
>
> Los **traductores e intérpretes se responsabilizan del L10N** durante todo el desarrollo de la aplicación.

Localización (L10N)

> ⚠ **Advertencia**
>
> Los **desarrolladores pueden participar en el L10N**, pero un rol meramente técnico, como asistente de integración.

Integrando I18N y L10N

Integrando I18N y L10N

> **Nota**
>
> **I18N es un prerequisito para L10N**. No puedes localizar una aplicación efectivamente sin tener su versión internacionalizada primero.

La pregunta del millón

¿Cómo configuramos y diseñamos una aplicación para lograr esta simbiosis?

El sistema locale

¿Qué es un locale?

Un **locale** es un conjunto de **parámetros y convenciones** que definen:

- El idioma del usuario.
- La región geográfica/cultural.
- Las convenciones de formato (números, fechas, moneda).
- El sistema de escritura.
- Las reglas de ordenación.
- La codificación de caracteres.

¿Qué es un locale?

Un locale **encapsula el contexto del idioma** necesario para que el software interprete y presente información de manera apropiada para el usuario.

Anatomía de un locale: formato POSIX

Se sigue un formto estándar:

```
idioma[_TERRITORIO][.codificación]
[@modificador]
```

Anatomía de un locale: formato POSIX

```
es_ES.UTF-8@andalucía
|  |  |      |
|  |  |      └─ Modificador (opcional): Variante dialectal
|  |  └──────── Codificación: Cómo se representan los caracteres
|  └─────────── Territorio: País/región específica
└────────────── Idioma: Código ISO 639
```

Anatomía de un locale: formato POSIX

Algunos ejemplos...

```
es_ES.UTF-8      # Español de España, UTF-8
es_MX.UTF-8      # Español de México, UTF-8
en_US.UTF-8      # Inglés americano, UTF-8
en_GB.UTF-8      # Inglés británico, UTF-8
pt_BR.UTF-8      # Portugués brasileño
pt_PT.UTF-8      # Portugués de Portugal
zh_CN.UTF-8      # Chino simplificado (China continental)
zh_TW.UTF-8      # Chino tradicional (Taiwan)
de_DE.UTF-8      # Alemán de Alemania
de_CH.UTF-8      # Alemán de Suiza
```

Destripando el locale: código de idioma

Todos los códigos de idioma principales siguen el estándar **ISO 639-1**, el cual establece **dos letras** para definir un idioma.

Ejemplos de idiomas: `zh`: Chino, `en`: Inglés, `es`: Español, `hi`: Hindi, `fr`: Francés, `pt`: Portugués

> Actualmente, el **ISO 639-1** soporta **184** idiomas principales.

Destripando el locale: código de idioma

> **Nota**
>
> También existe ISO 639-2 (tres letras) para idiomas menos comunes: `cat` (catalán), `glg` (gallego), `eus` (euskera). Actualmente, soporta 460 lenguas no principales.

Destripando el locale: código de territorio

Se utiliza el estándar **ISO 3166-1 alpha-2**, el cual **establece el código de un país por dos letras.**

> **Nota**
>
> El propósito es ser capaces de distinguir variantes regionales del mismo idioma

Destripando el locale: código de territorio

Español

- `es_ES` - España (vosotros, ordenador)
- `es_MX` - México (ustedes, computadora)
- `es_AR` - Argentina (vos, computadora)
- `es_PE` - Perú (ustedes, CPU)

Inglés

- `en_US` - EE.UU. (color, elevator)
- `en_GB` - Reino Unido (colour, lift)
- `en_AU` - Australia (slang único)
- `en_CA` - Canadá (mezcla US/UK)

Destripando el locale: código de territorio

> ⚠ **Advertencia**
>
> **Error común**: Asumir que el código de idioma es suficiente. ¡Las diferencias lingüísticas entre países y regiones son significativas!

Destripando el locale: codificación de caracteres

Es la codificación por la que las cadenas de texto se almacenan en **bytes**. **Cada codificación determina sus propias reglas**.

Destripando el locale: codificación de caracteres

Codificaciones conocidas:

Destripando el locale: codificación de caracteres

UTF-8 se posiciona como el estándar moderno para codificar cadenas de caracteres

Carácter	UTF-8 (hex)	Bytes
'A'	41	1 byte
'ñ'	C3 B1	2 bytes
'€'	E2 82 AC	3 bytes
'🚀'	F0 9F 9A 80	4 bytes
'你'	E4 BD A0	3 bytes

Destripando el locale: codificación de caracteres

Tip

Intenta usar, siempre que puedas, **UTF-8** en tus proyectos. Es compatible con ASCII y soporta todos los idiomas del mundo.

Destripando el locale: codificación de caracteres

Ejemplo:

Texto original: `"Niño comió paella con el señor López"`

```
Codificación correcta (UTF-8):
Niño comió paella con el señor López

Interpretado como ISO-8859-1:
Niño comió paella con el señor López

Interpretado como Windows-1252:
NiÃ±o comiÃ³ paella con el seÃ±or LÃ³pez

Interpretado como ASCII (sin soporte):
Ni?o comi? paella con el se?or L?pez
```

¿Dónde internacionalizar?

Una cuestión que queda por resolver es dónde aplicar la internacionalización con el sistema locale.

Las seis categorías POSIX

El sistema locale POSIX divide las convenciones culturales en **6 categorías independientes**:

`LC_COLLATE`: Ordenación ortográfica de cadenas

> ¿ñ va después de n?

Las seis categorías POSIX

El sistema locale POSIX divide las convenciones culturales en **6 categorías independientes**:

`LC_CTYPE`: Clasificación de caracteres

> ¿Es ñ una letra? ¿Es mayúscula o minúscula?

Las seis categorías POSIX

El sistema locale POSIX divide las convenciones culturales en **6 categorías independientes**:

`LC_MESSAGES`: Mensajes y salidas del sistema

> “Error” vs “Erreur” vs “Fehler”

Las seis categorías POSIX

El sistema locale POSIX divide las convenciones culturales en **6 categorías independientes**:

`LC_MONETARY`: Formato monetario

> “1.234,56 €” vs “$1,234.56”

Las seis categorías POSIX

El sistema locale POSIX divide las convenciones culturales en **6 categorías independientes**:

`LC_NUMERIC`: Formato numérico

> “3,14” vs “3.14”

Las seis categorías POSIX

El sistema locale POSIX divide las convenciones culturales en **6 categorías independientes**:

`LC_TIME`: Formato de fecha/hora

> “07/12/2024” vs “12/07/2024”

Las seis categorías POSIX

> **Tip**
>
> Si quieres internacionalizar toda la aplicación, mejor usa `LC_ALL`, que es una meta-categoría que establece todas a la vez.

I18N en C++

GNU gettext

GNU gettext es el sistema estándar de internacionalización para software de código abierto y muchos proyectos comerciales. Desarrollado por el proyecto GNU en 1995.

GNU gettext

¿Por qué es el estándar?

1. **Maduro y robusto**: Casi 30 años de desarrollo
2. **Ecosistema completo**: Herramientas, editores, plataformas
3. **Ampliamente adoptado**: GNOME, KDE, WordPress, GIMP, etc.
4. **Soporte completo**: Plurales, contextos, comentarios
5. **Multiplataforma**: Linux, macOS, Windows, BSD

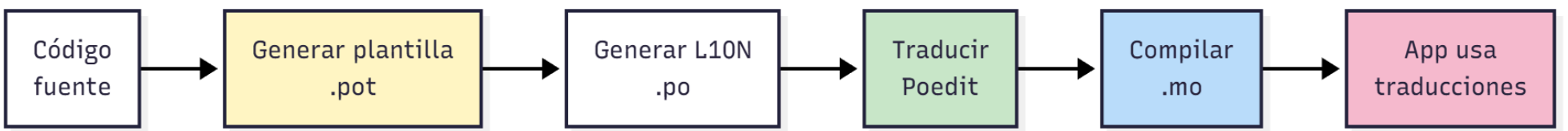
GNU gettext

- Linux desktop (GNOME, KDE, XFCE)
- Aplicaciones GNU (Bash, GCC, Emacs)
- Gestores de contenido (WordPress, Drupal)
- Herramientas de desarrollo (Git, many others)

El enfoque gettext

1. **El código fuente es la fuente de verdad**: Los strings en el código (típicamente inglés) son las claves.
2. **Separación clara**: Desarrolladores escriben código, traductores traducen.
3. **Formato legible**: Archivos PO son texto plano, editables con cualquier editor.
4. **Optimización en runtime**: Archivos MO binarios optimizados para velocidad.
5. **Extracción automática**: Las herramientas extraen strings del código.

Ciclo de gettext



Generación de plantilla

En primer lugar, debemos generar una **plantilla** que los traductores replicarán para localizar la aplicación.

¿Cómo sabemos qué se debe traducir?

A través de la función `gettext`.

> **Nota**
>
> Gettext utiliza un sistema **clave-valor** para hacer las traducciones. El **string original** del código fuente define una clave, la cual se usará para sustituir por traducciones (o usarla por defecto en el idioma original).

Generación de plantilla

¿Cómo identificamos nuestro texto?

A través del **string original** en el código fuente. Ese término se convierte en una **clave**.

```
1 #include <libintl.h>
2 #include <locale.h>
3
4 gettext("Save file");  // "Save file" es el id
```

Generación de plantilla

Debemos definir también el **dominio** de nuestras traducciones (entendámolos como un *namespace*).

```
1 #include <libintl.h>
2 #include <locale.h>
3
4 textdomain("myapp");
5 gettext("Save file");  // "Save file" es el id
```

> ⚠ **Advertencia**
>
> ¡Mucho cuidado! Las traducciones, por defecto, se buscarán en la carpeta `/usr/share/locale/lg_RG/CODE/myapp.po`. Si quieres evitar este comportamiento, deberás reconfigurar el *path* de búsqueda con la función `bindtextdomain`.

Generación de plantilla

Para generar la plantilla, ejecutaremos el comando `xgettext`

```
1 xgettext --output=myapp.pot src/*.cpp
```

Tip

Si quieres evitar usar `gettext` en todo tu código, puedes **definir una macro de preprocesador** con un prefijo. Por ejemplo, _. Si haces eso, deberás añadir el argumento –keyworkd a `xgettext`

Localizando nuestra aplicación

Antes de seguir, debemos conocer los ficheros que genera `gettext`:

- **POT**: Template (plantilla sin traducir).
- **PO**: Portable Object (traducción editable).
- **MO**: Machine Object (binario optimizado).

Localizando nuestra aplicación

Es decir:

- **POT**: Plantilla de la que sacaremos las traducciones.
- **PO**: Fichero que los intérpretes usarán para localizar la aplicación.
- **MO**: Binario que leerá nuestro programa localizado.

Localizando nuestra aplicación

Portable Object Template `.pot`

Es la plantilla que contiene todos los strings **originales extraídos del código** sin ninguna traducción. Es el punto de partida para crear archivos PO de cada idioma.

Localizando nuestra aplicación

- Generado automáticamente por `xgettext`.
- Contiene `msgid` pero `msgstr` vacíos.
- Incluye metadatos: ubicación en código, comentarios, etc.
- Se actualiza cada vez que cambia el código.
- **No se traduce directamente**: Se usa para crear/actualizar ficheros `.po`.

Estructura de un archivo POT

Header (Cabecera del archivo)

```
msgid ""
msgstr ""
"Project-Id-Version: myapp 1.0\n"      # Nombre y versión
"Report-Msgid-Bugs-To: bugs@example.com\n"  # Dónde reportar errores
"POT-Creation-Date: 2024-12-08 10:30+0100\n"  # Cuándo se generó
"Content-Type: text/plain; charset=UTF-8\n"   # La codificación
...
```

Estructura de un archivo POT

Entrada simple

```
#: src/main.cpp:42              # Ubicación en código (archivo:línea)
msgid "Welcome"                 # String original
msgstr ""                       # Traducción (vacía en POT)
```

Estructura de un archivo POT

Entrada con contexto

```
#: src/menu.cpp:20
msgctxt "menu"
msgid "File"
msgstr ""
```

Estructura de un archivo POT

Entrada con comentarios

```
1 #. TRANSLATORS: This is shown when the user logs in
2 #: src/auth.cpp:15
3 msgid "Welcome back"
4 msgstr ""
```

Localizando nuestra aplicación

Para generar una localización de nuestro programa, deberemos crear un fichero `.po` a partir del `.pot` generado.

Usaremos el comando `msginit` con ese propósito:

```
1 msginit -i myapp.pot -l en_US -o en_US.po
```

Localizando nuestra aplicación

¿Qué hace msginit?

1. Copia todo el contenido del POT
2. Establece el idioma y región en el header
3. Configura Plural-Forms según el idioma
4. Inicializa msgstr vacíos (listos para traducir)
5. Añade información del traductor (si disponible)

Localizando nuestra aplicación

Tal y como observamos, el fichero `.po` generado es, de raíz, **igual que el fichero de plantilla**.

Sin embargo, este fichero es el que los intérpretes usarán para traducir la aplicación. Para ello, **deberán** rellenar todos los campos `msgstr` con el mensaje que deberá aparecer en esa versión.

Localizando nuestra aplicación

Es decir, en el **fichero** `.po` **debemos esperar:**

```
1 msgid "Settings"
2 msgstr "Configuración"
```

Localizando nuestra aplicación

> **Nota**
>
> Seguramente te estés preguntando: *¡Pero qué demonios va a hacer un intérprete con este fichero y este sistema!* ¡No te preocupes! Existen herramientas con GUI para editar estos ficheros, como **PoEdit** o **Lokalize** para que no sufran.

Localizando nuestra aplicación

Datos importantes de los ficheros po

Las entradas de un fichero `.po` tienen **estados**.

Localizando nuestra aplicación

Datos importantes de los ficheros po

Las entradas de un fichero `.po` tienen **estados**.

Si una entrada está completa, se dice que está **localizada**

```
1 msgid "Save"
2 msgstr "Guardar"
```

Localizando nuestra aplicación

Datos importantes de los ficheros po

Las entradas de un fichero `.po` tienen **estados.**

Si una entrada está in, se dice que está **sin localizar**

```
1 msgid "Save"
2 msgstr ""
```

En este caso, se pondrá el texto **original** por defecto.

Localizando nuestra aplicación

Datos importantes de los ficheros `po`

Las entradas de un fichero `.po` tienen **estados.**

Si lleva un comentario `#, fuzzy`, significa que **es una localización dudosa**

```
1 #, fuzzy
2 msgid "Load file from disk"
3 msgstr "Cargar archivo"
```

`gettext` no usará el `msgid` hasta que no se elimine el *flag*.

Localizando nuestra aplicación

¿Cuándo se pone el *flag* de *fuzzy*?

- El/la intérprete así lo decide por dudas en la localización.
- Cuando el comando `msmerge` detecta que el `msgid` cambió de una versión a otra del `.pot`.

Localizando nuestra aplicación

Sobre `msmerge`...

Es un comando útil para cuando actualizamos nuestro código fuente y, por lo tanto, **modificamos la plantilla de traducciones.**

```
1 msgmerge --update es_ES.po myapp.pot
```

Localizando nuestra aplicación

Sobre `msmerge`...

1. **Mantiene** traducciones existentes que no cambiaron.
2. **Añade** nuevos msgid (con `msgstr` vacío).
3. **Marca como fuzzy** traducciones de `msgid` modificados.
4. **Mueve a obsoletos** msgid que ya no existen.
5. **Actualiza** referencias de línea en comentarios.

Últimos pasos de la localización

El último paso para localizar nuestra aplicación es **compilar** los ficheros `.po` en ficheros `.mo`.

Últimos pasos de la localización

Machine Object

Archivo **binario** generado desde `.po`, optimizado para:

- Búsqueda ultra-rápida (hash tables).
- Tamaño compacto.
- Carga eficiente en memoria.
- Uso en producción.

Últimos pasos de la localización

- **No legible por humanos**: Formato binario
- **Multiplataforma**: Maneja endianness automáticamente
- **Rápido**: Búsquedas O(1) con hash
- **Compacto**: Sin comentarios, sin espacios extra
- **Solo strings traducidos**: Excluye fuzzy y sin traducir

Últimos pasos de la localización

> ⚠ **Advertencia**
>
> ¡Recuerda! Si no has cambiado con `bindtextdomain` la ubicación de búsqueda tus ficheros `.mo`, entonces `gettext` los buscará en la ruta de `usr/share/locale`.

Últimos pasos de la localización

Para compilar, usaremos el comando de `msgfmt`

```
1 msgfmt es_ES.po -o es_ES.mo
```

Presto! Ya tendremos nuestra aplicación localizada a varios idiomas.

Tip

Para probar en local, puedes cambiar la variable de entorno `LANG` al ejecutar el programa. Esto a veces no funciona, por lo que puedes también utilizar una de mayor nivel como `LANGUAGE`.

El desafío de los plurales

Cada idioma tiene **reglas diferentes** para plurales:

Inglés (2 formas)

```
1 file
2 files
5 files
```

Francés (2 formas)

```
0 fichier
1 fichier
2 fichiers
```

Polaco (3 formas)

```
1 plik
2 pliki
5 plików
22 pliki
```

Esloveno (4 formas)

```
1 datoteka
2 datoteki
3 datoteke
5 datotek
```

El desafío de los plurales

> ⚠ **Advertencia**
>
> **No puedes asumir** que añadir el caracter “s” funciona globalmente

El desafío de los plurales

Gettext gestiona **automáticamente** estas diferencias usando fórmulas matemáticas.

Plurales con gettext

Dentro de la cabecera del fichero `.po`

```
1  "Plural-Forms: nplurals=N; plural=EXPRESIÓN;\n"
```

- **nplurals**: Número de formas plurales
- **plural**: Expresión que devuelve el índice (0, 1, 2...)

Plurales con gettext

En el código, utilizaremos la función `ngettext`:

```
1 int count = get_file_count();
2
3 printf(ngettext(
4     "%d file",
5     "%d files",
6     count
7 ), count);
```

Plurales con gettext

¿Qué hace `ngettext`?

1. Toma el número `count`.
2. Aplica la fórmula Plural-Forms del locale actual.
3. Obtiene el índice (0, 1, 2…).
4. Busca msgstr[índice] en el catálogo.
5. Reemplaza %d con el número.

Plurales con gettext

En el fichero `.po` se generan todas las opciones de plurales que hay en el idioma.

```
1 "Plural-Forms: nplurals=2; plural=(n != 1);\n"
2
3 msgid "%d file"
4 msgid_plural "%d files"
5 msgstr[0] "%d archivo"     # Forma 0: singular (n=1)
6 msgstr[1] "%d archivos"    # Forma 1: plural (n!=1)
```

Plurales con gettext

Tip

GNU `gettext` especifica que, por defecto, contiene las fórmulas de plurales de 140 idiomas, por lo que las obtendrá automáticamente si especificamos bien el *locale* al que vamos a generar el fichero `.po`.