Internazionalizzazione dei programmi – i18n

[Questo articolo analizza il procedimento di internazionalizzazione e quello di localizzazione di un programma dal punto di vista del programmatore ed è dedicato a programmatori C con almeno le conoscenze di base del linguaggio.]

Avevo già anticipato in un precedente articolo, dedicato alla localizzazione dei programmi, i concetti di base relativi a internazionalizzazione (i18n) e localizzazione (l10n). Ritengo opportuno evitare di ripetere qui tali concetti di base e passare direttamente ad analizzare il punto di vista del programmatore.
Va premesso che il presente non vuole essere un trattato esaustivo sull'i18n (non sarei in grado di arrivare a tanto) ma una semplice introduzione per accompagnare un programmatore, a digiuno di tali concetti, un po' più addentro nell'argomento.

Sommario

Introduzione

Ritengo importante ribadire l'utilità del processo di globalizzazione dei programmi quale strumento per permettere agli stessi di essere fruibili in un numero maggiore di diversi paesi e quindi al maggior numero possibile di persone. Questa possibilità è ancora più importante nel mondo dell'open source dove raggiungere un ampio numero di sviluppatori o di testers può rappresentare per un programma la possibilità di sopravvivere.
A ciò si aggiunge la considerazione che molti meccanismi vengono implementati automaticamente dalle librerie standard o dai tools di programmazione e quindi, spesso, lo sforzo richiesto al programmatore è solo quello di predisporre il programma ad essere localizzato.
Sono comunque ben lontano dall'affermare che l'i18n sia semplice, non a caso il capitolo 22 del manuale della libc, dedicato all'argomento, si apre con una frase che, tradotta, suona più o meno così:
"Una domanda molto comune nei newsgroup e mailing list è 'Come posso fare <operazione qualsiasi di modifica a una stringa?' I programmatori più coscienziosi fanno seguito alla domanda con '...e come faccio a rendere tale codice portatile?' Povero innocente programmatore non hai idea della profondità dei guai nei quali ti stai gettando. Sarebbe meglio per la tua salute mentale infilare il problema in un cestino per la carta e dimenticartene..."

Ben lontano dal volervi spaventare ribadisco che cercherò di introdurvi all'argomento a piccole dosi per cui farete sempre in tempo ad abbandonare il tutto...andiamo quindi ad iniziare.

Il concetto di *locale*

Le regole guida dell' i18n richiedono che i programmatori scrivano un programma in modo da evitare di codificare al suo interno informazioni o strutture che non siano poi in grado di passare un processo di l10n. Per facilitare tale operazione si ha a disposizione per ciascuna lingua una tabella di regole e costrutti detta *locale* atta a descrivere delle specifiche convenzioni culturali. Alcuni aspetti di tali convenzioni possono essere ad esempio:

- La direzione della scrittura (molte lingue arabe od orientali prevedono la scrittura da destra a sinistra o dall'alto in basso)
- La classificazione dei caratteri (quali sono alfabetici, quali numerici...)
- Formato dei numeri, della data/ora, della valuta
- L'ordine dei caratteri ed il metodo di confronto

Purtroppo nel tempo la definizione di locale è stata oggetto di numerosi tentativi di standardizzazione che hanno portato a vari metodi di codifica e che prevedono differenti sistemi di locale come ad es. il POSIX locale per le convenzioni culturali di base, l' ISO/IEC 14652 che è una estensione del POSIX, l'X locale per le applicazioni Xwindow etc...
Le uniche locale che sono garantite essere sempre disponibili in ogni sistema sono la "C" e la "POSIX" che garantiscono un comportamento delle varie funzioni conforme alle specifiche standard del linguaggio C.

Categorie

Il linguaggio C ISO standard aderisce da vicino alle convenzioni POSIX per le tabelle locale. Ciascuna locale rappresenta una lingua, un territorio ed un determinato set di caratteri. Per l'identificazione di una locale si utilizza il seguente formato:
lingua_paese.set
La lingua viene identificata da un codice di due caratteri (consigliati minuscoli) secondo la definizione ISO 639:1988 e il paese è un codice di due caratteri (consigliati maiuscoli) definiti dall'ISO 3166-1.
Nell'uso più comune il codeset viene omesso e quindi avremo codici del tipo it_IT (cfr. RFC 1766) che ad esempio indica l'italiano parlato in Italia e le relative regole (come già accennato nel precedente articolo).
I programmi C normalmente ricavano tali informazioni (o meglio le ereditano), al loro avvio, dalla variabile d'ambiente LANG come già spiegato nell' articolo sul l10n. Oltre a questa possono essere definite (con lo stesso utilizzo) le variabili LANGUAGE e NLSPATH mentre LINGUAS è considerata obsoleta.
Ulteriori variabili d'ambiente sono definite in relazione alle *categorie*. Per permettere vari gradi di l10n dei programmi sono infatti state definite delle categorie (come parti di ciascuna tabella locale) ciascuna rappresentante alcuni aspetti specifici della localizzazione. Questo permette di scegliere una locale specifica ed indipendente per ogni categoria (a seconda della necessità). Variabili ritenute valide sono:

LC_ALL	sovrascrive tutte le categorie locali	
LC_COLLATE	per i confronti dei caratteri	
LC_TYPE	classificazione dei caratteri	
LC_MONETARY	informazioni legate alla valuta	
LC_NUMERIC	formattazione dei numeri	
LC_TIME	formattazione data e ora	
LC_MESSAGES	messaggi	

Lo standard ISO/IEC 1465, che è una estensione della locale POSIX, fra le altre cose aggiunge sei categorie a quelle definite dagli standard POSIX:

LC_PAPER	per la gestione delle dimensioni della carta	
LC_NAME	per la gestione dei nomi delle persone	
LC_ADDRESS	per la formattazione degli indirizzi e dei codici postali	
LC_TELEPHONE	per il formato dei numeri telefonici	
LC_MEASUREMENT	per le unità di misura	
LC_VERSIONS	per la gestione delle versioni della locale (le informazioni sono disponibili tramite la funzione nl_langinfo() che vedremo poi).	

Tutti gli standard si riservano di poter definire in futuro ulteriori variabili LC_* di cui i programmi potranno usufruire.

Esempi di utilizzo della locale

Il bello di tutto ciò è che lo standard C implementa alcuni meccanismi di gestione dell'i18n in modo automatico. Quindi prima di spaventarvi con altre nozioni passerei ad un esempio pratico:
Come sopra enunciato i programmi ereditano le impostazioni locali dalle variabili di ambiente, ma tali variabili non impongono automaticamente la locale usata dalle funzioni di libreria. Ogni programma si avvia con presettata la locale 'C'. Per utilizzare la locale specificata dalle variabili d'ambiente si deve utilizzare la funzione **setlocale()** come vedremo...

Utilizzando un editor a vostra scelta create quindi il file *esempio1.C* e copiateci il codice seguente:

```
/*esempio1.c - out sample*/
#include <stdio.h>
#include <locale.h>
int main (int argc, char *argv[]) {
  float price=12.5;
  /*selezione della lingua dall'ambiente       */
  /*utilizzare il comando export LANG=xx_XX    */
  setlocale(LC_ALL,"");
  printf ("formato numerico locale:  %f \n",price);
  return 0;
}
```

compiliamo
`$ gcc esempio1.c -o esempio1`
settiamo la variabile d'ambiente relativa al linguaggio scegliendo l'inglese USA
`$ export LANG="en_US"`
ed eseguiamo il programma

```
$ esempio1
formato numerico locale:  12.500000
```

L'output prodotto è il numero 12,5 in formato numerico anglosassone con il . (punto) come separatore decimale.
Ora selezionamo il formato italiano:

```
$ export LANG="it_IT"
$ esempio1
formato numerico locale:  12,500000
```

L'output prodotto è (ovviamente) il numero 12,5 in formato italiano con la , (virgola) come separatore decimale.
Faccio notare l'uso di LC_ALL all'interno della funzione setlocale() per settare la locale per tutte le categorie e le funzioni associate; specificando un parametro differente, al posto di LC_ALL, l'impostazione viene eseguita solo per la categoria di funzioni relative. Nell'esempio avremmo potuto utilizzare la seguente forma
`setlocale(LC_NUMERIC,"")`
ottenendo lo stesso risultato influenzando solo il comportamento delle funzioni legate al trattamento numerico.

Chiaramente non è necessario selezionare interattivamente dalla shell la lingua dell'output desiderato ma è possibile selezionarla da programma, indipendentemente dalle impostazioni delle variabili d'ambiente, come nell'esempio che segue:

```c
/*esempio2.c - locale sample*/
#include <stdio.h>
#include <locale.h>
int main (int argc, char *argv[]) {
   float price=12.5;
   //selezione del locale USA da programma
   setlocale(LC_ALL,"us_US");
   printf("formato numerico US:  %f\n",price);
   //selezione del locale Italia da programma
   setlocale(LC_ALL,"it_IT");
   printf("Formato numerico It:  %f\n",price);
   return 0;
}
```

passiamo a compilazione ed esecuzione:
$ gcc esempio2.c -o esempio2

```
$ esempio2
formato numerico US:  12.500000
Formato numerico It:  12,500000
```

Cambiando la locale da codice è comunque opportuno salvare la locale originale del sistema in una variabile in modo da poterla ripristinare. Ecco uno spezzone di codice che ottiene lo scopo:

```
.....
char *p, *save_locale;
/*leggo la variabile d'ambiente*/
p=setlocale(LC_ALL,"");
/*la salvo*/
save_locale = (char *)malloc(strlen(p) +1);
strcpy(save_locale, p);
......
/*eventuali modifiche*/
.........
/*ripristino*/
p = setlocale(LC_ALL, save_locale);
free(save_locale);
```

in questo modo abbiamo riportato il sistema alle condizioni iniziali.

Trattamento di valuta

A questo punto si potrebbe pensare che se tutto funziona automaticamente non c'è in realtà molto lavoro da fare. Per non deludere nessuno complichiamo un po' le cose con un programmino che mostra il valore dell'esempio precedente in formato valutario.

```
/* esempio3.c */
#include <stdio.h>
#include <libintl.h>
#include <locale.h>
int main (int argc, char *argv[]) {
   float price=12.5;
   /*setto la lingua inglese*/
   setlocale(LC_ALL,"en_US");
   /*estraggo il simbolo della valuta*/
   const char *sig=localeconv()->currency_symbol;
   printf("formato monetario US: %s  %f\n",sig,price);
   /* come sopra ma per formato italiano*/
   setlocale(LC_ALL,"it_IT");
   const char *sig2= localeconv()->currency_symbol;
   printf("formato monetario IT: %s  %f\n",sig2,price);
   return 0;
}
```

passiamo ancora alla compilazione e all'output

```
$ gcc esempio3.c -o esempio3
$ esempio3
formato monetario USA:    $ 12.500000
formato monetario Italia: L. 12,500000
```

Come si vede la funzione **localeconv()** ha prelevato i simboli corretti dalla locale prescelta e nel caso della lira ha anche inserito il punto del simbolo. Nel caso qualcuno si stia chiedendo perchè il formato italiano mostra la Lira come simbolo valutario al posto dell' Euro il motivo è che il programma è stato eseguito in una vecchia distribuzione di GNU/Linux nella quale le tabelle dei simboli non sono state aggiornate.

L'esempio appena mostrato però non tiene conto del fatto che non in tutti i paesi il simbolo valutario viene mostrato prima del valore. Un esempio più completo dovrebbe infatti tenere conto di questo come segue:

```
/* esempio4.c */
#include <libintl.h>
#include <locale.h>
int main (int argc, char *argv[]) {
   float price=12.5;
   /*setto la locale per la lingua tedesca*/
   setlocale(LC_ALL,"de_DE");
   printf("formato monetario Germania: ");
   /*verifico se il simbolo valutario precede il valore ed eventualmente lo stampo*/
   printf("%s ", (((int) localeconv()->p_cs_precedes)? localeconv()-
>currency_symbol:""));
   /*stampo il valore */
   printf("%f",price);
   /*verifico se il simbolo valutario segue il valore ed eventualmente lo stampo*/
   printf("%s ", (!((int) localeconv()->p_cs_precedes)? localeconv()-
>currency_symbol:""));
   printf("\n");
   /*come sopra ma per la lingua italiana*/
   setlocale(LC_ALL,"it_IT");
   printf("formato monetario Italia:   " );
   printf("%s ", (((int) localeconv()->p_cs_precedes)? localeconv()-
>currency_symbol:""));
   printf("%f",price);
   printf("%s ", (!((int) localeconv()->p_cs_precedes)? localeconv()-
>currency_symbol:""));
   printf("\n");
   return 0;
}
```

ovvero abbiamo verificato se il simbolo precede o segue il valore ed agito di conseguenza.
Possiamo verificare il tutto nell'output prodotto:

```
$ esempio4
formato monetario Germania: 12,500000DM
formato monetario Italia:   L.12,500000
```

Non abbiamo tenuto ancora conto degli spazi che precedono o seguono il simbolo ed altro ma penso che comunque i concetti siano chiari.
Come si è visto la funzione localeconv() può ritornare un buon numero di informazioni tramite la struttura **lconv** da essa restituita. Il modo migliore per verificare quali esse possano essere è leggere direttamente il file include locale.h relativo alla libreria locale. Questo file è oltretutto commentato in maniera molto chiara e ritengo inutile farlo a mia volta.

Se siete confusi provvedo subito a complicare le cose: vista la complessità nell'utilizzo delle funzioni standard del C gli autori degli standard X/Open hanno deciso di venire in nostro aiuto con una funzione atta a maneggiare direttamente i formati monetari (giusto per rimanere in tema) ma non solo. Tale funzione infatti può accedere a tutti i singoli elementi delle tabelle locale.
La funzione in questione è
ssize_t strfmon(char *s, size_t max, const char *format,...); ed è contenuta in *monetary.h*
un esempio di codice congruente con quelli già presentati potrebbe essere il presente:

```
/* esempio6.c */
#include <stdio.h>
#include <libintl.h>
#include <locale.h>
#include <monetary.h>
int main (int argc, char *argv[]) {
   double price=12.5;
   char buf[BUFSIZ];
   setlocale(LC_ALL,"de_DE");
   printf("formato monetario Germania: ");
   if (strfmon(buf,sizeof(buf),"%i",price) > 0 )   printf("%s \n ",buf);
   return 0;
}
```

Eseguendo il programma si ottiene:

```
$./esempio6
formato monetario Germania: 12,50 EUR
```

Come si vede le cose sono enormemente semplificate in quanto la funzione produce già un output correttamente formattato. Confrontando i due metodi la funzione strfmon() sembra essere nettamente vincente per semplicità d'uso ma bisogna tenere presente che la localeconv() ha una maggiore portabilità facendo parte delle librerie standard del C.

Trattamento di caratteri

In maniera analoga, a quanto visto per produrre un corretto output numerico, possiamo verificare quanto sia importante usare le corrette funzioni per la validazione dell'input quando esso comprenda delle lettere come nell'esempio che segue:

```
/* esempio7.c */
#include <libintl.h>
#include <locale.h>
#include <ctype.h>
int main (int argc, char *argv[]) {
  /*vengono settate tre variabili diverse, solo per comodità*/
  /*di lettura del programma, contenenti le lettere: a,b,à  */
  char first[]="a";
  char second[]="b";
  char third[]="à";
  /*creo un buffer per l'output*/
  char *a=0;
   setlocale(LC_ALL,"en_EN");
   printf("In Inghilterra: \n" );
   a= isalpha((int)first[0])?" è una lettera":" non è una lettera";
   printf("[%s]%s \n",first,a);
   a= isalpha((int)second[0])?" è una lettera":" non è una lettera";
   printf("[%s]%s \n",second,a);
   a= isalpha((int)third[0])?" è una lettera":" non è una lettera";
```

```
   printf("[%s]%s \n",third,a);
  setlocale(LC_ALL,"it_IT");
  printf("In Italia:   \n" );
  a= isalpha((int)first[0])?" è una lettera":" non è una lettera";
   printf("[%s]%s \n",first,a);
   a= isalpha((int)second[0])?" è una lettera":" non è una lettera";
   printf("[%s]%s \n",second,a);
   a= isalpha((int)third[0])?" è una lettera":" non è una lettera";
   printf("[%s]%s \n",third,a);
   return 0;
}
```

il programma produce il seguente output:

```
In Inghilterra:
[a] è una lettera
[b] è una lettera
[à] non è una lettera
In Italia:
[a] è una lettera
[b] è una lettera
[à] è una lettera
```

Ragioniamo un attimo sull'outpt: la locale "C" che viene usata per default dalle funzioni avrebbe dato lo stesso risultato del primo caso ovvero non avrebbe riconosciuto la à accentata come una lettera. Se l'esempio fosse stato una routine per la validazione dell'input che doveva accettare i soli caratteri alfabetici, in assenza dell'utilizzo della tabella locale, il programma avrebbe lavorato correttamente in Inghilterra ma non in Italia andando a scartare un input valido. Analoghe attenzioni si dovranno avere nell'ordinamento dei caratteri con la funzione strcoll() in quanto ordinare le stringhe in base al valore ASCII dei caratteri che la compongono porterà problemi con lettere accentate e similari che verranno ordinate in modo improprio.

Traduzione dei testi

La traduzione dei messaggi di testo è la parte, del processo di globalizzazione, che viene percepita per prima dall'utente finale ed è anche quella più complessa da gestire.
Fondamentalmente non esiste alcuno standard POSIX per la traduzione dei messaggi ma ci sono due standards de-facto uno da X/Open Group e l'altro proposto inizialmente da Sun ed ora utilizzato anche nei sistemi GNU. Entrambi si basano sullo stesso principio ovvero sulla costruzione di una tabella di riferimento tramite la quale effettuare la traduzione dinamicamente agganciando a run-time la voce tradotta.

Lo standard X/Open si basa sulla funzione catgets() che funziona sul principio di associare ad ogni frase da tradurre un identificativo numerico in modo da potere attraverso esso recuperare l'equivalente frase tradotta dal file della traduzione.
Per indicare alla funzione catgets() dove è situato il file contenente i messaggi tradotti viene utilizzata funzione **catopen()** la quale, a sua volta, si basa sulla variabile d'ambiente **NLSPATH**.
Non vorrei dilungarmi eccessivamente su questa funzione in quanto in ambiente GNU/Linux sembra essere meno usata, rispetto alla funzione gettext(), proprio per la difficoltà di manutenzione dei files di traduzione e delle tabelle di appoggio.

La Sun Microsystems, a suo tempo, attraverso l'Uniforum group, ha cercato di proporre un approccio differente per la traduzione dei messaggi proponendo la funzione gettext().
La funzione gettext() pur non essendo mai stata ratificata come standard lo è divenuta di fatto in quanto a parere di molti risulta più semplice da usare. A differenza della funzione catgets essa non necessita di un identificatore per la frase da tradurre ma utilizza la frase stessa come identificatore per la ricerca. La funzione viene definita nell'header *libintl.h* e la sua definizione esatta è:

```
char * gettext (const char *msgid)
```

Essa è accompagnata da altre funzioni necessarie per il suo corretto funzionamento:

```
char * textdomain (const char *domainname)
```

questa funzione definisce il dominio di default per tutte le chiamate successive di gettext()

```
char * bindtextdomain (const char *domainname, const char *dirname)
```

permette di specficare la directory ce contiene il file di messaggi tradotti per il dominio indicato (in realtà si tratta dell'indirizzo della gerarchia di directories nelle quali vanno salvati i files tradotti). Bisogna tenere conto che il percorso indicato può anche essere relativo e cautelarsi avendo cura che il programma non utilizzi la funzione `chdir` per cambiare la directory di lavoro

corrente, nel quale caso è opportuno che venga utilizzato per bindtxtdomain() un percorso assoluto.

Vediamo un esempio classico...

```
/*ciao.c*/
#include <locale.h>

#define PACKAGE "ciao"
#define LOCALEDIR "."

int main ()
{
    setlocale (LC_ALL, "");
    /*definisco il dominio (percorsi e nome file) */
    /*per la funzione gettext()                   */
    bindtextdomain (PACKAGE, LOCALEDIR);
    textdomain (PACKAGE);
    /*testo l'output in inglese ed in italiano*/
    /*scegliendo esplicitamente la lingua     */
    setlocale (LC_ALL, "en_US");
    printf (gettext ("Hello world\n"));
    setlocale (LC_ALL, "it_IT");
    printf (gettext ("Hello world\n"));
}
```

Per la creazione del file delle traduzioni vi rimando al precedente articolo, ricordandovi che il file oggetto va posto in una struttura di directory riferita a quella definita dal programma nella macro **LOCALEDIR**. la struttura sarà *codice lingua*/LC_MESSAGES.
Nel nostro caso ci si riferisce alla directory corrente "." dove è contenuto il programma e vanno definite quindi ./ it/LC_MESSAGES e ./us/LC_MESSAGES contenenti ciascuna il file ciao.mo (o ciao.gmo) tradotto rispettivamente in italiano ed in inglese. Una volta preparato il tutto provate ad eseguire il programma ed avrete come output:

```
$./ciao
Hello world
Ciao mondo
```

Tenete comunque presente che in un normale programma non viene scelta da programma la lingua di output ma viene prodotto l'output in base alla lingua corrente di sistema.

Vi sono delle raccomandazioni essenziali contenute nel manuale di gettext() ove si consiglia di:

- Utilizzare la lingua Inglese con uno stile decente.
- Usare frasi intere.
- Spezzare le frasi a livello di paragrafo.
- Utilizzare stringhe formattate piuttosto che concatenate.

L'ultima raccomandazione che non è poi così ovvia da comprendere si lega alla seconda ed alla modalità di lavoro di gettext(). Poichè la funzione utilizza la frase per ottenere la corrispondente frase tradotta l'uso di frasi intere permette di scegliere con minore ambiguità la frase corretta. Inoltre costruendo la frase con parole concatenate noi imponiamo, da codice, un ordine per le parole stesse che non necessariamente trova corrispondenza in una lingua straniera. Prendiamo ad esempio lo spezzone di codice seguente:

```
...
strcpy (s, gettext("The red "));
strcat (s, gettext(" car"));
strcat (s, gettext(" is on "));
strcat (s, gettext(" black"));
strcat (s, gettext(" road."));
...
```

Una volta stampata la stringa "s", in inglese, verrebbe visualizzato: "The red car is on black road.",
che va tradotto, in italiano, come: "L' auto rossa è sulla strada nera."
ma la concatenazione delle stringhe impone una traduzione del tipo: "La rossa auto è sulla nera strada.".
Comprensibile certamente, ma di sicuro non molto gradevole alla lettura; e in altre lingue, inoltre, il risultato potrebbe essere ancora peggiore.

A complicare l'utilizzo della funzione gettext sono le forme plurali...siamo infatti abituati a ragionare al plurale pensando all'esistenza di due forme quella singolare e quella plurale. Questo è vero per la nostra lingua ma esistono lingue nelle quali il plurale può coincidere con il singolare oppure possono esistere due o più forme plurali a seconda del numero di oggetti individuati.
Qualsiasi tentativo da parte del programmatore di risolvere tali forme nel codice porta ad un aumento della complessità del programma senza avvicinarsi alla soluzione del problema stesso. Il programmatore deve quindi evitare di prendersi in carico la gestione del problema ed utilizzare la funzione

```
char * ngettext (const char *msgid1, const char *msgid2, unsigned long int n)
```

che risolve il problema e per il cui utilizzo vi rimando al manuale di gettext nel quale troverete anche i suggerimenti per il miglior utilizzo della funzione.

Conclusioni

Concludo qui questa introduzione all'internazionalizzazione. Il tema avrebbe potuto essere sviluppato per volumi ma penso che i riferimenti in calce potranno soddisfare in modo valido le inevitabili domande che potranno nascere dopo questa lettura.
Sono chiaramente a disposizione di chiunque avesse bisogno di delucidazioni o avesse suggerimenti o segnalazione di inesattezze.

Riferimenti

RFC 1766 - Tags for the identification of Languages
Secure Programming for Linux and Unix HOWTO - capitolo 5.8
Programming for Internationalization FAQ
Thai Locale
Riferimenti alla funzione catgets
Manuale di gettext
Home page di Kbabel
The C++ Programming Language (3rd Edition) - Appendix D by Bjarne Stroustrup

di Rudi Giacomini Pilon