

Traiter de l'Unicode dans différents langages

Stéphane Bortzmeyer

<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 21 Février 2007. Dernière mise à jour le 16 Janvier 2008

<http://www.bortzmeyer.org/unicode-en-divers-langages.html>

Pour montrer les différences du support d'Unicode dans divers langages de programmation, voici le même cahier des charges mis en œuvre dans plusieurs langages.

Le programme à écrire doit accomplir les tâches suivantes :

- Lire les noms des fichiers à traiter sur la ligne de commande (ces fichiers sont tous codés en UTF-8),
- Pour chaque fichier, afficher le nombre de caractères Unicode qu'il contient (et surtout pas le nombre d'octets), le terminateur à la fin des lignes compte,
- Pour chaque fichier, créer un fichier de même nom terminé par `.reverse` et contenant les mêmes caractères mais où ils sont inversés pour chaque ligne, les lignes elles-mêmes étant maintenues dans leur ordre initial, le saut de ligne restant à sa place.

Par exemple, le fichier suivant :

```
Ils s'étaient réveillés  
dans une Citroën niçoise...
```

sera transformé en :

```
séllievér tneiaté's sli  
...esioçin nëortic enu snad
```

Ce programme permet de tester les entrées-sorties en UTF-8 mais aussi la manipulation de chaînes de caractère en Unicode, bien plus facile si le langage permet de manipuler des chaînes de caractères Unicode, et pas simplement des chaînes d'octets.

Comme me le fait remarquer Bertrand Petit, ce test est **peu fiable**. En effet, si la chaîne Unicode comprend des caractères Unicode combinants comme le U+0301 (accent aigu combinant), l'inversion va l'attacher à une autre lettre. En outre, inverser les chaînes de caractères ne sert à rien, c'est juste un exercice d'algorithmique classique.

Mais oublions ces problèmes et commençons par les langages où c'est le plus facile. Pour Python, tout est fourni dans le langage :

```

#!/usr/bin/python
# -*- coding: UTF-8 -*-

# Python a deux sortes de chaînes de caractères, les traditionnelles
# et les Unicode (elles devraient fusionner en Python 3000). Nous
# n'utilisons ici que les Unicode.
# http://evanjones.ca/python-utf8.html

# On note que seuls des modules standard, distribués avec Python, sont
# utilisés.
import sys
import codecs

def fatal(msg=None, code=1):
    if msg:
        sys.stderr.write("Fatal error: %s\n" % msg)
        sys.exit(code)

def usage():
    sys.stderr.write("Usage: %s file(s)\n" % sys.argv[0])

def reverse(str):
    """ Inverse une chaîne de caractères """
    # Sans doute pas efficace mais simple et joli
    if len(str) <= 1:
        return str
    else:
        return str[-1] + reverse(str[0:-1])

if __name__ == '__main__':
    if len(sys.argv) <= 1:
        usage()
        fatal()
    for filename in sys.argv[1:]:
        inputfile = codecs.open(filename, 'r', "utf-8")
        outputfile = codecs.open(filename + ".reverse", 'w', "utf-8")
        total = 0
        for line in inputfile:
            # Une autre solution si on n'a pas utilisé le module "codecs" :
            # line = unicode(rawline, "utf-8")
            total = total + len(line)
            # Ne pas oublier de ne *pas* inverser le saut-de-ligne final
            outputfile.write("%s\n" % reverse(line[:-1]))
        inputfile.close()
        outputfile.close()
        # TODO: print ne convertit pas si mon terminal n'est pas UTF-8
        # (alors que ça marche en Perl)
        print "%i caractères Unicode dans %s" % (total, filename)

```

On peut trouver d'autres textes intéressants sur le support d'Unicode en Python comme "*Python and Unicode*" (<http://downloads.egenix.com/python/Unicode-EPC2002-Talk.pdf>) de l'expert Marc-André Lemburg ou bien le "*Unicode HOWTO*" (<http://www.amk.ca/python/howto/unicode>).

En Perl, c'est tout aussi facile :

```

#!/usr/bin/perl -w

# man perlunicode

# On note que seuls des modules standard, distribués avec Perl, sont
# utilisés.

```

```

use strict;
use utf8;

sub usage {
    print STDERR "Usage: $0 file(s)\n";
}

sub fatal {
    my $msg = shift;
    if ($msg) {
        print STDERR "$msg\n";
    }
    exit(1);
}

if ($#ARGV < 0) {
    usage();
    fatal("");
}

my $filename;
foreach $filename (@ARGV) {
    open(INPUT, "<:utf8", $filename) or die "Cannot open $filename: $!";
    open(OUTPUT, ">:utf8", "$filename.reverse") or
        die "Cannot open $filename.reverse: $!";
    my $total;
    my $line;
    foreach $line (<INPUT>) {
        $total += length($line);
        chomp $line;
        my $rline = reverse $line;
        print OUTPUT $rline, "\n";
    }
    close(INPUT);
    close(OUTPUT);
    print "$total caractères Unicode dans $filename\n";
}

```

Dans les deux cas, on note qu'on n'utilise directement les constructions du langage (comme `reverse`), on n'a pas besoin de sous-programmes spécialisés en Unicode.

Pour Haskell, si le langage dispose en standard d'un type de chaîne de caractères Unicode (c'est même l'unique type), en revanche, il n'y a pas de possibilité standard de lire et écrire l'UTF-8. J'ai donc utilisé l'excellente bibliothèque `Streams` (<http://www.haskell.org/haskellwiki/Library/Streams>), ce qui donne le résultat suivant :

```

import qualified System

-- Bibliothèque Streams en http://www.haskell.org/haskellwiki/Library/Streams
import qualified System.Stream as Streams

getLinesFromFile handle = do
    over <- Streams.vIsEOF handle
    if over then
        return ([])
    else do
        line <- Streams.vGetLine handle -- Saute le saut de ligne final, d'où le +1
        -- dans l'appel à foldr, qui n'est pas idéal puisque, sur MS-Windows,
        -- c'est deux caractères.
        rest <- getLinesFromFile handle
        if rest == [] then
            return ([line])
        else

```

<http://www.bortzmeyer.org/unicode-en-divers-langages.html>

```

        return ([line] ++ rest)
    }
}

processFile filename = do
  input <- Streams.openFD filename Streams.ReadMode
  >>= Streams.withEncoding Streams.utf8
  lines <- getLinesFromFile input
  let total = foldr (\l -> \ total -> total + (length l) + 1) 0 lines
  putStrLn ((show total) ++ " caractères Unicode dans " ++ filename)
  Streams.vClose input
  let reversedLines = map reverse lines
  output <- Streams.openFD (filename ++ ".reverse") Streams.WriteMode
  >>= Streams.withEncoding Streams.utf8
  mapM (Streams.vPutStrLn output) reversedLines
  Streams.vClose output

main = do
  filenames <- System.getArgs
  mapM processFile filenames

```

Une autre solution pour lire l'UTF-8 aurait été celle décrite par Eric Kow (<http://www.haskell.org/haskellwiki/UTF-8>).

Pour Java, le type `String` est également un type Unicode, donc c'est assez simple. Les entrées-sorties du `FileReader` se font avec l'encodage spécifié par la locale donc attention à vérifier qu'elle est bien en UTF-8. Avec le compilateur `gcj`, par exemple :

```

% LC_CTYPE=fr_FR.utf8 gcj --main=CAS --output=cas-java CAS.java
% LC_CTYPE=fr_FR.utf8 ./cas-java test1.txt test2.txt
Le fichier test1.txt contient 52 caractères Unicode
Le fichier test2.txt contient 5 caractères Unicode

```

Kim Minh Kaplan est l'auteur de ce code :

```

import java.io.BufferedReader;

public class CAS
{
    static public void main(String args[]) throws java.io.IOException
    {
        for (int i = 0; i < args.length; i++) {
            BufferedReader entrée = null;
            try {
                entrée = new BufferedReader(new java.io.FileReader(args[i]));
                java.io.Writer sortie = null;
                try {
                    sortie = new java.io.FileWriter(args[i] + ".reverse");
                    int compteur = 0;
                    String ligne;
                    while ((ligne = entrée.readLine()) != null) {
                        int j = ligne.length();
                        // BOURDE - Si le fichier n'est pas terminé
                        // par un caractère de fin de ligne, ça en
                        // ajoute un.
                        compteur += j + 1;
                        while (j-- > 0)
                            sortie.write(ligne.charAt(j));
                        sortie.write('\n');
                    }
                }
            }
        }
    }
}

```

<http://www.bortzmeyer.org/unicode-en-divers-langages.html>

```
    }
    System.out.println("Le fichier " + args[i] + " contient " +
                      compteur + " caractères Unicode");
}
finally {
    if (sortie != null)
        sortie.close();
}
    }
    finally {
if (entrée != null)
    entrée.close();
}
}
}
}
```

En D, si le langage tient de son ancêtre C des notions de bas niveau (comme le fait que les trois types caractères sont décrits en terme de leur encodage), le type `dchar` est un moyen pratique de manipuler l'Unicode, en n'utilisant que des fonctions fournies avec le langage (comme la méthode `reverse` sur les chaînes de caractères :

```
// We use only standard libraries

// Various IO and file handling stuff
import std.stdio;
import std.file;
import std.stream;

// Unicode stuff
import std.utf;

int main(char[][] argv)
{
    int argc = argv.length;
    char[] filename;
    char[] data; // Uninterpreted data, to be read from the file and transformed
                // to Unicode later. Must be UTF-8 code units.
    dchar[] line; // dchar == Unicode character (encoded as UTF-32)
    File ifile, ofile;
    if (argc <= 1) {
        writefln("Usage: %s file ...", argv[0]);
        return 1;
    }
    for (int i = 1; i < argc; i++) {
        filename = argv[i];
        int length = 0;
        try {
            ifile = new File(filename, FileMode.In);
            ofile = new File(filename ~ ".reverse", FileMode.Out);
            do {
                data = ifile.readLine();
                line = toUTF32(data);
                length = length + line.length;
                ofile.writefln(line.reverse);
            } while (!ifile.eof);
            writefln("%s: %d Unicode characters", filename, length);
        }
        catch (Exception fe) {
            writefln("Problem \"%s\" on %s, the file probably does not exist",
                    fe.toString(), filename); // FileException does not catch non-existing files :-()
        }
    }
}
```

```

return 0;
}

```

Pour Emacs Lisp, Kim Minh Kaplan a écrit ce code, en ajoutant que c'est « un des langages les plus adaptés » :

```

(defun kmk-count-and-swap (filename)
  (interactive "fCount and swap file: ")
  (with-temp-buffer
    (insert-file-contents-as-coding-system 'utf-8 filename)
    (while (not (eobp))
      (apply
        'insert
        (nreverse
          (string-to-list
            (progl
              (buffer-substring-no-properties (point-at-bol) (point-at-eol))
              (delete-region (point-at-bol) (point-at-eol))))))
      (forward-line))
    (let ((coding-system-for-write 'utf-8))
      (write-file (concat filename ".reverse")))
    (message "Sauvé %s.reverse : %d caractères"
      filename (- (point-max) (point-min)))))

```

Pour Common Lisp, Samuel Tardieu me propose ce code (partiel) que je n'ai pas encore pu tester (Kim Minh Kaplan y a apporté un changement, sur la délicate gestion du saut de ligne final). On voit que, là aussi, seuls des primitives du langage sont utilisées (il faut apparemment un Common Lisp qui supporte l'UTF-8 en standard (comme sbcl).

```

(defun revuni (filename)
  (with-open-file (in filename :external-format :utf-8)
    (with-open-file (out (concatenate 'string filename ".reverse")
      :external-format :utf-8 :direction :output
      :if-exists :supersede)
      (loop for (line missing-nl-p) = (multiple-value-list (read-line in nil))
        while line
          sum (length line)
          do (write-string (nreverse line) out)
          unless missing-nl-p
            do (write-char #\newline out)
            and sum 1))))

(defun revfile (filename)
  (format t "Le fichier ~a contient ~a caractères unicode~%"
    filename (revuni filename)))

```

Pour C, où il n'existe pas de type de données standard pour les caractères Unicode, Erwan David suggère le code (partiel, il manque la fonction main) suivant, que je n'ai pas encore testé :

```

/* Erwan David */

#include <string.h>
#include <stdio.h>

```

```
/* return the number of bytes used by a UTF-8 character, given the
   first character */
```

```
static int numBytes(unsigned char firstByte)
{
    if(firstByte & 0x80 == 0)
    {
        return 1;
    }
    if(firstByte & 0xC0 == 0)
    {
        return 2;
    }
    if(firstByte & 0xE0 == 0)
    {
        return 3;
    }
    if(firstByte & 0xF0 == 0)
    {
        return 4;
    }
    return 0;
}
```

```
#define BUFFER_SIZE 1024
```

```
static int reverseAndCount(FILE * in, FILE * out)
{
    /* assume lines shorter than 1024 bytes */
    unsigned char inBuffer[BUFFER_SIZE];
    unsigned char outBuffer[BUFFER_SIZE];
    int end=0;
    int numRead=0;
    int totalCount=0;
    while(!end)
    {
        int inIndex=0, outIndex=0;
        int numBytesInline;

        while(1)
        {
            numRead=fread(inBuffer+inIndex,1,1,in);
            inIndex++;
            if(numRead=0)
            {
                /* end of file */
                end=1;
                break;
            }
            if(inBuffer[inIndex-1] == '\n') /* end of line */
            {
                break;
            }
        }
        /* reverse line : num bytes will not change */
        numBytesInline=inIndex;
        outBuffer[inIndex-1]='\n';
        totalCount++;
        outIndex=inIndex-1;
        inIndex=0;
        while(outIndex > 0)
        {
            int charLength=numBytes((int)inBuffer+inIndex);

            memcpy(outBuffer+outIndex-charLength,
                inBuffer+inIndex,
                charLength);
            outIndex-=charLength;
            inIndex+=charLength;
            totalCount++;
        }
    }
}
```

```

    }
    fwrite(outBuffer, 1, numBytesInline, out);
}
return totalCount;
}

```

On note l’astuce pour compter le nombre de caractères, en exploitant une propriété d’UTF-8 (les premiers bits nous indiquent la taille totale en octets du caractère). Globalement, cette solution me semble de très bas niveau et peu généralisable à d’autres manipulations.

Une meilleure solution, me semble t-il, est d’utiliser une bibliothèque Unicode comme l’excellente ICU, qui offre des services aux programmeurs C, C++ et Java. Je n’ai pas encore essayé mais un exemple proche figure dans la documentation (<http://icu.sourceforge.net/userguide/utext.html>):

```

int countWords(const char *utf8String) {
    UText      *ut      = NULL;
    UBreakIterator *bi    = NULL;
    int        wordCount = 0;
    UErrorCode  status   = U_ZERO_ERROR;

    ut = utext_openUTF8(ut, utf8String, -1, &status);
    bi = ubrk_open(UBRK_WORD, "en_us", NULL, 0, &status);

    ubrk_setUText(bi, ut, &status);
    while (ubrk_next(bi) != UBRK_DONE) {
        if (ubrk_getRuleStatus(bi) != UBRK_WORD_NONE) {
            /* Count only words and numbers, not spaces or punctuation */
            wordCount++;
        }
    }
    utext_close(ut);
    ubrk_close(ut);
    assert(U_SUCCESS(status));
    return wordCount;
}

```

Le programmeur C++ pourra trouver d’autres informations dans *“Tips on Using Unicode with C/C++”* (<http://linuxgazette.net/147/pfeiffer.html>).

Olivier Robert a fourni le code Ruby. Il nécessite Ruby ≥ 1.9 , l’encodage des chaînes de caractère n’étant apparu qu’avec cette version. On peut trouver d’avantage de détails sur Ruby et Unicode dans *«“Understanding M17n”* (http://blog.grayproductions.net/articles/understanding_m17n) ».

```

# === main
#
def main(argv)
  total = 0
  out = File.open("#{argv[0]}.reverse", "w")
  File.open(argv[0]).each do |line|
    line.force_encoding("UTF-8")
    r_line = line.chomp.reverse
    total += r_line.length
    out.puts(r_line)
  end
  puts "#{total} unicode characters in #{argv[0]}"

```

<http://www.bortzmeyer.org/unicode-en-divers-langages.html>

```
return 0
end # -- main

if __FILE__ == $0 then
  exit(main(ARGV) || 1)
end
```

Lua, lui, n'offre aucune possibilité Unicode, ce qui est paradoxal pour un langage conçu dans un pays non-anglophone. Comme en C, les caractères en Lua ne sont en effet que des octets, dont l'interprétation est laissée au programme. Il n'est même pas possible d'écrire un caractère Unicode quelconque dans un programme, la syntaxe des séquences d'échappement étant trop limitée pour cela. La FAQ de Lua (<http://lua-users.org/wiki/LuaUnicode>) détaille le problème et propose quelques bricolages pour le contourner (pour mesurer la longueur d'une chaîne encodée en UTF-8, le truc est analogue à celui utilisé en C plus haut). Comme pour l'autre langage de bas niveau de cette liste, C, on doit compter sur des bibliothèques extérieures comme `slunicode` (<http://luaforge.net/projects/sln/>).

Et dans les autres langages ? Je vais m'en occuper mais les contributions extérieures sont les bienvenues.