GTK+/GLibの文字符号化方式

岩本一樹

概要

GTK+およびGLibが扱う文字列はUTF-8であるが、GTK+とGLibの一部のAPIはUTF-8以外の文字列を扱う。また一部のAPIは暗にUTF-8を他の文字符号化方式に変換する。本論文ではGTK+とGLibのAPIの文字列の文字符号化方式(Character Eencoding)を検証し、file name encodingを中心にGTK+およびGLibでプログラムを組む上で注意すべき点をまとめる。

1.ファイルやパスを表す文字列

ほとんどのAPIでは文字列はUTF-8であるが、ファイルやパスを表す文字列はUTF-8以外の場合がある。GLib Reference ManualのCharacter Set Conversionではそれをfile name encodingと言っている。UTF-8とfile name encodingの関係はGLib Reference ManualのCharacter Set Conversionで説明されている。openやstat、readdirのような元々GTK+やGLibと関係ないAPIがUTF-8ではないのは言うまでもないことである。しかしGTK+やGLibのAPIであっても、主に前述のopenやstat、readdirのようなAPIに使われるファイルやパスを扱うGTK+やGLibのAPIはUTF-8以外の時がある。

例えばGtkFileChooserで選択されたファイル名はgtk_file_chooser_get_filenameで取得できるが、このAPIが返す文字列はfile name encodingである。またGtkFileChooserにファイル名を設定するgtk_file_chooser_set_filenameの引数の文字列はfile name encodingである。

実際のfile name encodingの文字符号化方式はGLib-2.4以降は環境変数G_FILENAME_ENCODINGで指定される。GLib-2.6以降ならば環境変数G_FILENAME_ENCODINGは複数の文字符号化方式を「,」で区切って指定できる(GLib-2.4であっても「,」で区切る記述は認められるが、先頭の文字符号化方式のみ有効となる)。

例
G_FILENAME_ENCODING=ISO-8859-1

例
G_FILENAME_ENCODING=EUC-JP,CP932

文字符号化方式に@localeを指定した時にはそれはロケールの文字符号化方式になる。ロケールの文字符号化方式とは、優先順位が高い順にnl_langinfoの返値、setlocaleの返値、環境変数のLC_ALL、LC_CTYPE、LANGの値で示されるいずれか1つの文字符号化方式のことである。WindowsではGetACPの返値が元になる。

例
G_FILENAME_ENCODING=@locale,CP932

もし環境変数G_FILENAME_ENCODINGが定義されていないか、GLib-2.4より前で環境変数G_BROKEN_FILENAMESが定義されているならば、file name encodingの実体はg_get_charsetの返値になる。すなわち現在のロケールの文字符号化方式になる。

環境変数G_FILENAME_ENCODINGとG_BROKEN_FILENAMESの両方が定義されていない時には、file name encodingの実体はUTF-8になる。

バージョン	G_FILENAME_ENCODING	G_BROKEN_FILENAMES	文字符号化方式
2.2 以前	無視される	あり	ロケールの文字符号化方式
		なし	UTF-8
2.4	あり	無視される	G_FILENAME_ENCODING の 文字符号化方式(単一)
	なし	あり	ロケールの文字符号化方式
		なし	UTF-8
2.6 以降	あり	無視される	G_FILENAME_ENCODING の 文字符号化方式(複数可)
	なし	あり	ロケールの文字符号化方式
		なし	UTF-8

なお Windows ではどのバージョンも環境変数 G_FILENAME_ENCODING と G_BROKEN_FILENAMES には対応しておらず、これらは無視される。Windows における file name encoding の実装は後に述べる。

2.file name encoding の変換

これらの変換のために GLib の API には g_filename_to_utf8 と g_filename_from_utf8、GLib-2.6 以降ならば g_filename_display_name、g_filename_display_basename がある。g_filename_display_basename は引数のファイル名のディレクトリを取り除いた名前部分だけを UTF-8 に変換して返す。

g_filename_to_utf8 と g_filename_display_name の違いは変換を試みる文字符号化方式にある。GLib-2.6 以降には g_get_filename_charsets という API があり、これは file name encoding の実際の文字符号化方式を返す。もし環境変数 G_FILENAME_ENCODING で複数の文字符号化方式を指定していたならば、この API は複数の文字符号化方式を返す。

g_filename_to_utf8 は g_get_filename_charsets が複数の文字符号化方式を返しても最初の文字符号化方式からのみ UTF-8 への変換を試みる。それに対して g_filename_display_name は複数の文字符号化方式が返された時には順番に変換が成功するまで変換を試みる。ゆえに g_filename_display_name の方が UTF-8 に変換できる可能性が高いと言える。しかし g_filename_display_name で UTF-8 に変換されたファイル名は、どの文字符号化方式で変換されたのかわからないため file name encoding に戻す変換はできない。g_filename_to_utf8 で変換されたファイル名は g_filename_from_utf8 で UTF-8 から file name encoding への変換ができる。しかし g_filename_to_utf8 と g_filename_from_utf8 を使った変換では元の同じ文字に変換できるという保証はない。4 節で詳しく述べる。

3.ファイル名はプログラム内部で file name encoding のまま保持する

1 つのプログラムで UTF-8 と file name encoding の 2 種類の文字符号化方式が混在するのは面倒なので、プログラマは gtk_file_chooser_get_filename や g_dir_read_name で文字列を取得したならば、直ちに UTF-8 に変換して file name encoding の文字列をその直後に破棄したくなるかもしれない。file name encoding が必要ならば、そのときに UTF-8 から file name encoding に変換して、不要になったらすぐに file name encoding の文字列を破棄すればプログラムの中のすべての文字列は UTF-8 になる。このアイデアはシンプルで良いように見えるが、問題を含んでいる。file name

encodingとUTF-8の変換が失敗した時、プログラムが正常に動作しなくなる。
　file name encodingをg_filename_display_nameやg_filename_to_utf8でUTF-8に変換できなかったとしても、そのファイル名が間違っているというわけではない。変換は失敗するかもしれないが、だからといってgtk_file_chooser_get_filenameやg_dir_read_nameが返した文字列をopenやstatのようなAPIで使えないわけではない。プログラム内部ではfile name encodingのファイル名を保持し続け、openやstatのようなAPIを呼び出す時には元々のfile name encodingのファイル名を使うようにする。もし変換ができなかった時には、例えばファイル名の代わりに「Unknown file name」のような文字列を表示する。file name encodingとUTF-8の変換に失敗する原因は

1.環境変数G_FILENAME_ENCODINGとG_BROKEN_FILENAMESが正しくない
2.ファイル名の文字符号化方式がUTF-8との変換に対応していない
3.対応する文字がない

が考えられる。

4.libiconvのマッピング問題

　g_filename_to_utf8でfile name encodingの文字列をUTF-8に変換してからUTF-8の文字列をg_filename_from_utf8でfile name encodingの文字列に戻した時に、最初の文字列と同じ文字列に戻ることは保証されない。g_filename_to_utf8とg_filename_from_utf8の動作はiconvに依存する。例えばiconvではCP932→UTF-8→CP932で変換を行った場合に元の文字に戻らない場合がある。

例
≒:87 90 → E2 89 92 → 81 E0
僴:ED 61 → E5 83 98 → FA 7D
兗:ED 62 → E5 85 8A → FA 7E
黑:EE EC → E9 BB 91 → FC 4B
∵:FA 5B → E2 88 B5 → 81 E6

5.設定ファイルに保存

　プログラムを利用するユーザが開いたファイルや入力した文字列の履歴をファイルに保存し、次に起動した時にもその履歴が生かせるようにするのは珍しいことではない。それはGLibのKey-value file parserを使って実装されるかもしれないし、それ以外の方法で実装されるかもしれない。いずれにしても、設定をテキストファイルに保存するならばUTF-8とfile name encodingの問題が生じる。
　ユーザが入力する文字はUTF-8であるが、ファイル名はfile name encodingである。これらを同じファイルに書き出せば、1つの設定ファイルに異なる文字符号化方式が混在することになる。設定ファイルをエディタで開く場合、UTF-8のテキストファイルとして読み込めばファイル名は文字化けするし、file name encodingのテキストファイルとして読み込めばファイル名以外の文字は文字化けする。
　解決方法の1つとしてUTF-8またはfile name encodingのどちらかの文字列をg_strescapeにより7ビットの範囲に収めてしまう方法が考えられる。この場合はテキストエディタで矛盾なく編集できるが、可読性が低下する。
　もう1つはfile name encodingの文字列をUTF-8に変換して保存する方法である。この場合には設定ファイルをUTF-8のテキストファイルとして開くことができるが、前述の問題が発生する。

設定ファイルにファイル名を保存する場合には、下記の 3 つから選択することになる。

1.文字列を g_strescape/g_strcompress により変換する
2.file name encoding を UTF-8 に変換する
3.テキストファイルとしての編集を諦める

6.Windows のファイル名実装

Windows の API は文字列を ANSI コードページで扱うものと、UNICODE で扱うものの両方が存在する。ANSI コードページでは 1 文字は 1 バイトまたは 2 バイトになる。日本語版 Windows ならば ANSI コードページは CP932 になる。UNICODE と称するものは実際には UTF-16LE のことである(UNICODE と称するのは不適切かもしれないが、Microsoft が発行する文書で UNICODE と書かれているので、以下 UNICODE と記述する)。例えば CreateFile という API は実際には CreateFileA と CreateFileW がある。前者はファイル名を ANSI コードページの文字列で扱うのに対し、後者は UNICODE になる。多くの場合、ソースコードレベルでは CreateFileA や CreateFileW を直接扱うことはなく CreateFile と記述する。ヘッダファイルには UNICODE という名前のマクロが定義されているか否かで、CreateFile は CreateFileA または CreateFileW になる。文字列を表す型は TCHAR や LPTSTR であり、これも UNICODE というマクロで実際の型が決まる。ほとんどの文字列を引数にとる API と文字列をメンバにもつ構造体でこのような仕組みになっている。

Windows NT とそれ以降の OS はファイル名に UNICODE を使うことができる。すなわち ANSI コードページには存在しない文字をファイル名に使うことができる。そのため引数が ANSI コードページな API ではファイルを正しく扱えない。引数が UNICODE な API を呼び出さない限り、ファイルを正しく扱えないのである。

7.Windows における file name encoding の実装

Windows では UTF-8 と file name encoding の関係はさらに複雑になる。ここまでは「file name encoding は open や stat、readdir のような API に使うことができる」という認識であった。GTK+-2.4 までは Windows でもその認識で正しかった。Windows ではどのバージョンも環境変数 G_FILENAME_ENCODING と G_BROKEN_FILENAMES には対応しておらず、GTK+-2.4 以前では file name encoding は ANSI コードページに固定されている。

しかし GTK+-2.6 以降は file name encoding の文字列を open や stat、readdir のような API に使うことができない。GTK+-2.4 以前のように file name encoding を ANSI コードページにしてしまうと、GtkFileChooser や g_dir_read_name で ANSI コードページにはない文字を含むファイル名を取得できなくなる。そこで GTK+-2.6 以降では file name encoding は UTF-8 として実装されている。GtkFileChooser や g_dir_read_name では可能ならば UNICODE 系の Windows の API を呼び出し、UNICODE から UTF-8 への変換を行っている(不可能ならば ANSI コードページ系の Windows の API を呼び出し、ANSI コードページから UTF-8 への変換を行っている)。

これに対応して GLib では g_open や g_stat などの API を新たに用意している。これらは g_open ならば open に、g_stat ならば stat と同じ働きをする。UNIX 系 OS ではこれらは単に名前が変わっただけとみなすことができる。Windows では g_open や g_stat などの API の引数のファイル名はやはり file name encoding であり file name encoding の実体は UTF-8 なので、引数のファイル名は実際には UTF-8 になる。Windows では例えば g_open は引数のファイル名を UNICODE に変換して wopen を呼び出す(UNICODE が利用できないならば引数のファイル名を ANSI コードページに変換して open を呼び出す)。g_open や g_stat などの API を利用すれば、UNIX 系 OS でも

Windows でもソースコードの記述を一貫して同じにすることができる。g_open や g_stat などの API の説明は GLib Reference Manual の File Utilities にある。UNIX 系 OS であっても GTK+-2.6 以降ならば g_open や g_stat などの API を使うべきである。

8.file name encoding と"真の"ファイル名の文字符号化方式の関係

GLib Reference Manual の File Utilities にある API のうち g_chmod、g_access、g_creat、g_chdir は GLib-2.8 以降である。GLib-2.6 で chmod、access、creat、chdir を使うならば、file name encoding を ANSI コードページに変換しなければならない。file name encoding から ANSI コードページへの変換は g_locale_from_utf8 で、ANSI コードページから file name encoding への変換は g_locale_to_utf8 でおこなう。通常は file name encoding を UTF-8 とみなすことはできないし、ロケールの文字符号化方式が実際のファイル名の文字符号化方式と同一とは限らない。これは Windows だけの特別な処理となる。また GLib-2.8 以降であっても GLib 以外の対応する API が GLib の File Utilities にはない API を呼ぶ時には、この変換が必要になる。

Glib-2.8 以降ならば g_win32_locale_filename_from_utf8 という API があるので、前述の g_locale_from_utf8 の代わりにこれを用いる。g_win32_locale_filename_from_utf8 は通常は g_locale_from_utf8 と同じだが、もしファイル名を変換できなくてかつ Windows の UNICODE 系の API が使える時には、GetShortPathNameW で短いファイル名を取得して、取得できたならばそれを ANSI コードページの文字列で返す。

いずれにしても Windows では ANSI コードページを引数にとる API のために、g_win32_locale_filename_from_utf8(GLib-2.8 以降)または g_locale_from_utf8(GLib-2.6)で変換する特別な処理が必要になる。

バージョン	file name encoding の実体	file name encoding から fopen などの API 呼び出し時の変換	File Utilities の API
2.4 以前	ANSI コードページ	不要	なし
2.6	UTF-8	g_locale_from_utf8	g_open、g_rename g_mkdir、g_stat、g_lstat g_unlink、g_remove g_rmdir、g_fopen g_freopen
2.8 以降		g_win32_locale_filename_from_utf8	g_chmod、g_access g_creat、g_chdir

9.print 系 API

C 言語のランタイムライブラリにある printf、vprintf、fprintf、vfprintf は GTK+のプログラムで利用できるが、UTF-8 の文字列をこれらの API に渡した場合、ロケールの文字符号化方式が UTF-8 でなければ正しく動作しない。GLib-2.2 以降には g_printf、g_vprintf、g_fprintf、g_vfprintf が用意されている。これらも C 言語のランタイムライブラリにある printf、vprintf、fprintf、vfprintf と同様に UTF-8 の文字列をこれらの API に渡した場合にはロケールの文字符号化方式が UTF-8 でなければ正しく動作しない。これらの API を使う場合には文字符号化方式の変換が必要になる。

しかし GLib Reference Manual の Message Output and Debugging Functions にある g_print は文字符号化方式の変換を行う。この API に UTF-8 の文字列を渡した時にはロケールの文字符号化方式に変換されて出力される。同様に glog と g_logv およびそれを利用するマクロの g_critical、g_debug、g_error、g_message、g_warning も同様に文字符号化方式を変換する。

g_print や g_log では書式制御には C 言語と同様に「%s」が使える。そして C 言語と同様に

「%Ns」(N は数値)で文字の数を指定できる。g_print や g_log では出力する文字列が確定した最後に文字符号化方式の変換を行うので、思ったように動作しないかもしれない。例えばロケールの文字符号化方式が CP932 の時に

g_print ("'%13s'\n", "岩本一樹");

を実行するとスペースは 1 つだけ挿入される。「岩本一樹」は UTF-8 で 12 バイトなので、UTF-8 で数えて不足する 1 バイト分のスペースだけが挿入される。その後で文字列を CP932 に変換する。「岩本一樹」を CP932 に変換すると 8 バイトになるので、「'」の中は全部で 9 バイトにしかならない。文字列を成型されたデータとして扱おうとするならば注意する必要がある。

10.その他の注意点

main の引数はロケールの文字符号化方式である。main の引数からファイルを開く時にはロケールの文字符号化方式から UTF-8 に変換し、その結果を file name encoding に変換しなければならない。具体的には g_locale_to_utf8 と g_filename_from_utf8 を呼び出す。main の引数を GTK+の API で表示する時には g_locale_to_utf8 で UTF-8 に変換する必要がある。

Windows のコンソールアプリケーションの main の引数はロケールの文字符号化方式であり、ロケールの文字符号化方式とは ANSI コードページである。しかし GTK+を利用するプログラムをコンソールアプリケーションとしてコンパイルするのは稀だろう。Windows の GUI プログラムでは WinMain からプログラムが開始する。

GLib には g_getenv、GLib-2.4 以降には g_setenv と g_unsetenv、GLib-2.8 以降には g_listenv という API があり環境変数を扱うことができる。環境変数の名前と値の文字符号化方式は file name encoding である。それは GLib Reference Manual の Miscellaneous Utility Functions で規定されている。もちろん Windows ではこの file name encoding も上記のファイル名の時と同様に GLib-2.4 以前はその実体は ANSI コードページであり、GLib-2.6 以降は UTF-8 である。

g_get_host_name はそのシステムの対応する API が返す文字列をそのまま返す。この API が返す文字列の文字符号化方式はロケールの文字符号化方式と同じと考えて良い。

g_get_user_name と g_get_real_name は Windows 以外ではそのシステムの対応する API が返す文字列をそのまま返すが、Windows では UTF-8 になる。この 2 つの API を利用するプログラムのソースコードは、コンパイルされる環境に応じて分岐が必要になる。Windows 以外ではこの API が返す文字列の文字符号化方式はロケールの文字符号化方式とみなして g_locale_to_utf8 を呼ぶことになり、Windows では文字符号化方式の変換を行わない。

例
```
#ifdef G_OS_WIN32
  display_name = g_strdup (g_get_real_name ());
#else /* not G_OS_WIN32 */
  display_name = g_locale_to_utf8 (g_get_real_name (), -1, NULL, NULL, NULL);
#endif /* not G_OS_WIN32 */
```

11.最後に

統計的な根拠のない私の偏見ではあるが、上級の UNIX 系 OS のユーザになるほどファイルやディレクトリの名前にスペースを含めないのと同様に US-ASCII 以外の文字をファイルやディレクトリ名には使わなくなる傾向にあると思う。US-ASCII の文字しか使わないならば、UTF-8 と file name encoding に間する問題は生じない。よって file name encoding のバグが露見しにくい。

しかしUNIX系OSが一般に普及すれば、ファイルやディレクトリの名前にUS-ASCII以外の文字を使うことが珍しくはなくなる。またGTK+はWindows版やMac OS X版もあり、GTK+を使ったプログラムはマルチプラットフォームとなりうる。せっかくGTK+を使ったのに、file name encodingへの対応が不十分であったためにUNIX系OSでしか正常に動作しないというのでは、ガッカリである。

日本語を母国語とするプログラマは日本語環境のゆえにこの文字符号化方式の問題を理解しやすい立場にあると思う。私自身も自作のプログラム(Video maid、Text maidなど)を通じて文字符号化方式の問題の手本となるように努力したい。

参考文献

GLib Reference Manual http://developer.gnome.org/doc/API/2.0/glib/index.html
GTK+ Reference Manual http://developer.gnome.org/doc/API/2.0/gtk/index.html
Unicode 標準入門 著:Tony Graham 訳:乾和志/海老塚徹 監修:関口正裕 翔泳社
CJKV 日中韓越情報処理 Ken Lunde 著 小松 章, 逆井 克己 訳 オライリー