SUPPORT OF LOCALE-DEPENDENT BEHAVIOR IN C++, JAVA AND .NET PLATFORM

Miroslav Virius
Faculty of Nuclear Sciences and Physical Engineering
Czech technical University in Prague
Trojanova 13, 120 00 Praha 2
virius@km1.fjfi.cvut.cz

ABSTRACT:
This article treats the idea o locale-dependent behavior (i.e. the localization and the internationalization) of programs. It compares the support of the locale-dependent behavior in some most common programming languages and describes the trend of the evolution of the programming languages and environments in this area.

KEY WORDS:
Locale, Alphabetic ordering, Date format, Number format, Time format, Localization, Internationalization

WHAT IS LOCALE-DEPENDENT BEHAVIOR?
Common user of computer programs expects, that programs he uses respect conventions of the region where he or she lives. This means for many programmers, that the program displays in the menu "Nápověda" instead of "Help" or "?" etc. But displaying texts in user's mother tongue (of course, using correct font) is only a very small part of locale dependent behavior – or of the questions of localization and internationalization of software products. A short list of some other problems related to locale conventions follows:

- Correct encoding of input and output text files.
- Correspondence of lowercase and uppercase characters.
- Alphabetic ordering of character strings.
- Hyphenation.
- Number and currency format.
- Formats of the date, time, time zone, era etc.
- Names and abbreviations of the names of months and days in the week.
- Counting days in the a week and weeks in the a year.
- Date of the acceptance of the Gregorian calendar.
- The possibility to use native alphabet in the identifiers in programs.

Some conventions depend on the language (e.g. alphabetic ordering), some conventions depend on the country (e.g. number format) without regard to the language.
In this article, we will restrict ourselves to languages using vocal writing like Latin, Greek or Cyrillic alphabet or their extensions. Let us look at some of the above mentioned problems closer.

Encoding
Number of the characters, even in all of the Latin based alphabets only, widely exceeds the number of possible 8-bit characters. (This is not only due to the accents, but even due to some special characters, as the German "ß" – the "sharp s" – or æ used in Danish, among other languages.) The first way to deal this overhead of characters were the so called code pages – different (locale dependent) meaning of the code numbers above 127. This was hardly applica-

Naformátováno: Vpravo: 0,63 cm

ble to multilingual texts and it was of course unusable for the languages like Chinese using some thousands of ideographic characters.
This is not enough: Different 8-bit encodings for the same language were used in different platforms. At least five different 8-bit encodings have been used for Czech language, for example.
In the late 80's, the 16-bit encoding known as *Unicode* was proposed; for all practical purposes, this encoding is equivalent to the encoding described in the international standard [1]. It is now widely accepted.
Note that 8-bit encodings are used for text output in the Microsoft Windows operating systems by default.

Lowercase and Uppercase Characters
The above mentioned alphabets distinguish lowercase and uppercase characters. You can meet two major problems:

- Given a lowercase character, there is no corresponding uppercase character. A good example is the German character "ß" (sharp "s"). It is usually replaced by "SS" in uppercase texts.
- Accents used in lowercase texts may be omitted in uppercase texts. This is the case e.g. in the French.

In such cases, there is no unique correspondence between the lowercase and uppercase texts.

Alphabetic Ordering
Alphabetic ordering is probably one the most apparent example of the locale dependent behavior. First, let us say a few words about terminology: Alphabetic ordering is often confused with lexicographical ordering, but these are two different algorithms.
Lexicographical ordering is ordering (of n-tuples) based on lexicographical comparison. This is defined as follows: Given two n-tuples, $a = (a_1, a_2, \ldots, a_n)$ and $b = (b_1, b_2, \ldots, b_n)$, we say, that $a > b$, if and only if one of the following conditions hold:
$a_1 > b_1$ or
$a_1 = b_1$ and $a_2 > b_2$ or
…
$a_i = b_i$ for all $i = 1, \ldots, n - 1$ and $a_n > b_n$.
On the other hand, the alphabetic comparison is much more complicated and widely depends on language conventions. Main points are:

- Different characters may be considered to be equal. E.g., uppercase and lowercase letters with the same meaning, such as "a" and "A", are in Czech (and many other languages) considered equal for the string collation. In Finnish, "v" and "w" are considered to be equal for purpose of the alphabetic ordering. (So, the following names of American states, Washington, Virginia and Wisconsin, are given here in correct alphabetic order according to Finnish conventions.)
- A group of characters is considered as one character. This is the case of "ch", which is in Czech considered as one letter greater than "h" and less than "i", while in Spanish it is considered as one letter, too, but it is ordered between "c" and "d". Another example is "ll" in Spanish or "dž" in Croatian.
- One character is – for the purpose of ordering – considered as a group of characters. This is the case of German umlauts, that are ordered as if they were itemized – e.g. "ü" is ordered according to German rules as a group of two characters "ue".
- Some characters with accents are first considered as the corresponding characters without accent. Only if the order of the entries cannot be settled this way, the accents are taken into account (and characters with accents are usually considered to be greater

Naformátováno: Vpravo: 0,63 cm

than corresponding characters without accents). Such a rule is usually accompanied with some rules stating the ordering of the characters differing only in the accent (e.g "é" and "ě" in Czech). This is the case of some (not all) Czech letters with accents; similar rules can be found e.g. in French.

Hyphenation
The breaking the word on the end of the line (hyphenation) uses various rules in different languages. Algorithms for hyphenation are rather complex and aren't part of locale settings on common computers now. As far as I am aware, this problem is solved in text processors by specialized program components only. We will not discuss hyphenation in this article.

Number Format
There are two main differences in number format in European countries:

- Comma or point is used as decimal separator. (As far as I am aware, people use decimal point only in Greece and Switzerland on the European continent; and of course on British Islands. Note that all the four nations living in Switzerland use the decimal point, even though people speaking the same language in surrounding countries use the decimal comma. This is an example of country dependent convention that is not language dependent.)
- Many different characters are used as group ("thousand") separator; e.g. Czech and French convention is to use a space, German convention is to use the point, Swiss convention is to use an apostrophe (') etc.
- In some non European countries, the digits need not be written in groups of three. E.g., one million is written as 10.00.000 in Nepal.

Currency
In addition to the differences in number formatting, the following locale specific differences concern the notation of the currency amount:

- There are of course different symbols of the local currency.
- The currency symbol is written to the left or to the right of the amount.
- There is or is not a space between the amount and the currency sign.
- Negative amount is written with the minus sign or is marked in some other way (e.g. with letters "CR" for "credit").

Date
Date consists of three items: day, month and year. But the format of the date varies – nearly any order of these three items is somewhere acceptable. The usual order in the Czech environment is *dd. mm. yyyy*, where *dd* stays for the day, *mm* for the number designing the month and *yyyy* for the year; the order *mm dd yyyy* is used in the United States etc. Here are some other problems.

- What are the month and day names and their abbreviations?
- Are the Roman numerals used for month in short date format?
- Which character(s) to use as a separator of this three date components?

Time and Time Zone
It is usual to write the time in 24 hours cycle in many countries; some other countries use two 12 hours cycles per day, some use both. In 12 hours cycle, it is necessary to add, whether it is before or after the noon (in local format). There are of course many other problems:

- Which character is used as a separator in time notation? Quite common is the colon.
- What abbreviations are used for hours, minutes, seconds and time AM or PM?

Naformátováno: Vpravo: 0,63 cm

- In 24 hours cycle, is the midnight time 0:00:00 or 24:00:00?
- In 12 hours cycle, is the noon time 12:00:00 or 0:00:00?
- What is the difference between local and universal time (Greenwich mean time)?
- What is the local name of the time zone (such as Central European Time, which is GMT + 1 hour)?
- Is the daylight saving scheme used in the given country? Is it used just now?

Era

For some mysterious reasons, there is no year zero in our calendar and it is not usual to speak about the year -127; instead, we use the era name, something like 127 BC. This gives two pieces of information:

- What is the origin of the era (e.g. "AH" would refer to the Islamic era).
- Whether is the date before or after the origin.
- Note that the names of era need not be widely accepted – e.g "př.n.l." is widely used. in Czech, but some people prefer older "př.Kr". Some people prefer the "BCE" ("Before Common Era") to the "BC" (Before Christ) in English speaking countries.

Counting the days and weeks

The week begins usually with the Monday in European countries; in the United States, the first day in the week is the Sunday.

The international standard ISO 8601 [2] provides the rule to determine whether the week that contains both last day of one year and the first day of the following year should be taken as last week of the ending year or as the first week of the beginning year. Of course, not all the countries follow this standard.

Gregorian Calendar and Julian Day Number

European countries, United States and many other countries in the world use Gregorian calendar now. If you intend to perform some historical computation, you will have to take into account even the date of the acceptance of the Gregorian calendar reform. This reform was announced by the pope Gregor XIII at February 24, 1582, and it ordered to leave 10 days in the current year beginning with October 5 (so the next day after October 4, 1582, was October 15). This reform was accepted in different countries in different years [3]:

- Gregorian calendar was accepted in 1582 in the very Catholic countries, as in prevailing part of Italy, in Spain etc., and in the Netherlands, but the days recommended by the pope were not necessarily kept.
- It was accepted in the Czech countries in 1584 (but different days were omitted in Bohemia, different in Moravia and different in Silesia).
- It was accepted in 1700 in prevailing part of the protestant countries, as in Germany.
- It was accepted in 1752 in England.
- Some alternative way to approach to the right time counting was tried in Sweden for some time (in 1700 – 1712). In 1712, Sweden returned to Julian calendar and accepted the Gregorian calendar in 1753.
- Orthodox countries accepted Gregorian calendar in the 20th century (e.g. Russia in 1918, Greece in 1923).

Julian day number is an useful way to circumvent the problems with calendar changes and other irregularities in date computing. It is the number of the days elapsed since the noon of the Monday, January 1, 4713 BCE (computed according to Julian calendar extended back in the time). If necessary, fractions of the day can be used to express the exact time.

Note that the name "Julian day counting" has nothing to do with Julian calendar and with Julius Caesar.

Naformátováno: Vpravo: 0,63 cm

Identifiers

Any program should be a model of the solved problem. It is clear that the programmer should concern on the problem he solves, not on the names of its components, and the names – identifiers – should be as descriptive as possible. This should be the names from the problem domain, of course. So it might be better to use programmer's mother tongue in the identifiers, not only a restricted set of it or mangled names without accents.

SUPPORT OF LOCALE-DEPENDENT BEHAVIOR

A few words about the history

One the first programming language with elements of support of locale-dependent behavior is Cobol 60, that contained the `DECIMAL POINT IS COMMA` statement for changing the role of the point and comma characters in number literals. But this was rare phenomenon up to the early 90's – usual programming languages and environments were English based, because most computers and compilers were produced in the United States. Even early versions of the C and C++ languages contained no support for locale specific behavior.

Expansion of personal computers changed the situation dramatically. The tools for the support of the locale dependent conventions emerged in many compilers as proprietary extensions first. Locale support is in some extent a part standard of many programming languages and environments since the middle of 90's. This support is usually based on locale settings embedded in the operating system, so its implementation is platform dependent.

The C and C++ programming languages

The `setlocale()` function and the constants defining categories of locale settings were described in the first international standard of the C programming language [4], but it was not very useful in the most common implementations; often, only the "C" locale – i.e. standard conventions of the C language – was available. In fact, this version of the C language supported only 8-bit characters (even though it contained the `wchar_t` type for the representation of the multibyte characters). Functions supporting Unicode and other multibyte characters were introduced into the C programming language standard in the amendment ISO/IEC 9899/AMD1:1995.

The locale settings in the C language hold for the whole program, it cannot be set independently for various parts of the program.

The C++ programming language, as described in the international standard [5], allows the programmer to use all the tools of the C language; moreover, it brings the `locale` class that is much more flexible and various `locale`s may be independently applied to different parts of the program; you can use different `locale` for the input stream and different `locale` for the output stream, for example.

The `locale` class is a container for the so called "facets". Facets are instances of the template classes derived from the `facet` class; every facet encapsulates one part – a facet – of the locale dependent behavior; e.g. the `ctype<charT>` facet contains the tools for the classification of the characters (whether the given character is a digit, a letter etc.), tools for the conversion between uppercase and lowercase characters etc. The `collate<charT>` class contains the tools for the alphabetic ordering of character strings. Other facets contain formatting tools for the numbers, date, currency etc. The user can define and use his own facets.

On the other hand, this support in not complete, because the C++ language does not contain corresponding data types – there is no class representing the date or the time in the standard library, for example, so the facets are in fact useless now. You can write your own tools based on these facets, but their use will be slightly different from e.g. the number formatting.

The support of different encodings is not given by the standard. If you use the PC with the Microsoft Windows, it is embedded into the locale class.

Naformátováno: Vpravo: 0,63 cm

The most serious problem of the locale support in the C++ is, that this feature is not well implemented in many cases. Some implementations do not support it at all, some others support it correctly in some versions only.
It is not unusual, that some features, as the character transformations, are properly implemented only for wide character type even in otherwise correct implementations.
The C++ standard admits the use of Unicode characters classified as letters in identifiers; nevertheless, it is rare phenomenon in the compilers up to now.

Java
The Java programming language was released in 1995. It supports the locale dependent behavior since the very beginning; it is Unicode based (it uses the UTF-8 encoding internally) and provides the programmer tools for an easy transfer of the program source code to different code pages (the native2ascii program).
The locale dependent behavior support is based on the `java.util.Locale` class. This class affects the behavior the class `java.text.Format`, that serves as an interface for the locale dependent text formatting.
This support includes the dealing with uppercase and lowercase letters, alphabetic ordering, number and time formatting, the names of the days and months including their abbreviations, date of acceptance if the Gregorian calendar etc. Java also allows to compute the Julian day numbers.
It also allows using the Unicode characters in the identifiers.
The Locale class allows not only to get the list of all supported locales, but even to change some settings oh the locale for current program. E.g., in the Czech locale is by default the "po Kr." string set as the name of the current era; you easily can change it to more common "n.l.".
Algorithm for alphabetic ordering sorts correctly text containing national characters of the selected locale; it may behave incorrectly, if the text contains characters of other languages. In the case of the Czech locale, the sorting (collating) algorithm distinguishes lowercase and uppercase letters – if the only difference of two entries is that one contains an uppercase letter and the second does not so, it orders the entry containing the capital letter behind the entry that does not contain it. This is deviation from the Czech standard for alphabetic ordering [6], that states, that corresponding uppercase and lowercase characters should be considered equal.
Dealing with different locale-dependent fonts is provided by the configuration files of the Java virtual machine (the Java runtime environment).
Algorithm for the transformation to uppercase characters changes the "ß" to "SS" in German locale settings.
The encoding is not included in the locale specification. If necessary, it is passed as a string parameter to the some methods.
This programming language fully supports the use of the Unicode characters in the identifiers.

Microsoft .NET Framework
We will consider the whole .NET Framework in this subsection, because its infrastructure – including all the libraries – is common for all the languages compiled for this platform. As well as Java, this platform is Unicode based. It also supports the Unicode characters in identifiers.
The main tool to achieve the locale dependent behavior in this platform is the `Culture-Info` class. This class provides the information necessary for string comparison (i.e. alphabetic ordering), date and time formatting etc. in the extent very similar to Java.
The .NET Framework is primarily intended for the Microsoft Windows operating systems. Hence, the .NET SDK does not provide the tools for the transfer of the source code to differ-

Naformátováno: Vpravo: 0,63 cm

ent encoding. On the other hand, the integrated programming environments usually offer the possibility to select the encoding of opened and stored files.
Note that unlike in C++ and Java, the default behavior is using the locale dependent string comparison in .NET Framework. (In fact, this is the only meaningful behavior; ordering according the ASCII code numbers is easy to implement, but has no benefit for the user or the program.) The `ToUpper()` method leaves the "ß" character unchanged in German locale settings. The default symbol for the current era in the Czech locale is – unlike in Java – the "n. l." abbreviation.
The support of the Julian day numbers is not the standard part of the .NET Framework.
The .NET platform supports locale dependent resource libraries. An assembly may have assigned the "culture"; when loading a library, the .NET platform automatically looks for the version with the given culture. This enables the coexistence of different locale dependent versions of the same library.
Another very important support for localization and internationalization of programs is embedded in the most common development environment. If you set the `Localizable` property of the form (in the Windows forms application) to `true`, a large amount of the properties of the form and all the components in the form will be stored as resources. This includes not only texts, but even the dimensions of the controls, because these dimensions are locale dependent, too. (Consider the text sizes of the "Nápověda", "Help" or "?" captions in the same menu command in different language versions.)

CONCLUSION AND PERSONAL EXPERIENCE

C++, Java and the programming languages for the .NET platform are not the only languages supporting in some extent the internationalization and the localization of applications. In the PHP, these features are based on the C locale, for example.
Support of localization and internationalization, or of the locale-dependent behavior, became a standard part of the programming languages and environments recently. Honoring national standards and locale conventions starts to be required. This is supported by national standards, as well as international standards – good example is the Unicode [1] and standard for international date and format [2]. The support for the alphabetic ordering of strings brings the template of an algorithm for string comparison [7].
Surprisingly enough, the main power of internationalization and localization seem to be transnational corporations. Local software houses ignore these problems often and sometimes even declare that "this cannot be done".
Individual Czech programmers – mainly inexperienced ones – like to declare that "the computer should communicate in English only", or even "I would forbid all this nonsense". Well, this is often indolence; but it may be result of the fact that this topics is ignored usually in the textbooks and programming courses, too.

ACKNOWLEDGEMENT

This work was supported by Ministry of Education, Youth and Sports of the Czech Republic under grant no. LA 08010.

NOTE

Author regrets to announce this paper is published in English. It is caused by the National research and development policy issued by the Research and Developement Council of the Czech Republic, that discriminates Czech language.

REFERENCES

Naformátováno: Vpravo: 0,63 cm

1. International Standard ISO/IEC 10646:2003 Universal Multiple-Octet Coded Character Set (UCS). ISO, Genève 2003.
2. International Standard ISO/IEC 8601:2004. Representation of dates and times. ISO, Genève 2003.
3. Wikipedia – Gregoriánský kalendář: http://www.cs.wikipedia.org.
4. International Standard ISO/IEC 9899:1990. Programming Languages – C. ISO, Genève 1990.
5. International Standard ISO/IEC 14882:2003. Programming Languages – C++. ISO, Genève 2003.
6. ČSN 97 6030. Abecední řazení. Český normalizační institut, Praha 1993.
7. ISO/IEC 14651:2007. International string ordering and comparison – Method for comparing character strings and description of the common template tailorable ordering. ISO, Genève 2007.

Naformátováno: Vpravo: 0,63 cm