Internationalization

Rex Jaeschke

Edition: 1.0
Printing: November 24, 1999

The training materials associated with this book are available for license. These materials include overhead transparencies, instructor guide, and lab solutions in electronic format. Interested parties should contact the author at the address below.

Please address comments, corrections, and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023
+1 (703) 860-0091
+1 (703) 860-3008 (fax)
rex@RexJaeschke.com

This book was typeset by the author using the TEX typesetting package, LATEX macros, and Makeindex indexing tool.

Contents

Chapter 1

An Overview

In this chapter, we will be introduced to the idea of internationalization with regard to software development.

1.1 Introduction

In recent years there has been an increasing interest in and demand for serious and standardized support for culture-specific data processing. This is happening particularly within countries having multiple languages and cultures (such as Canada, Belgium, and Switzerland). And as national economies become linked there is a steadily increasing demand for software export. However, users prefer to deal with program interfaces using their own cultural conventions, and rightly so.

1.2 What is Internationalization?

An *internationalized* program is one that has *no* dependency on *any* culture. The term internationalization is often referred to as I18N, since there are 18 letters between the i and n.

For most programs to be useful, they must perform some form of I/O. (Even a seemingly cultural-independent program like an operating system has to accept commands and produce messages.) This requirement gives rise to *localization* (L10N); the process of adapting an internationalized program to a specific cultural environment.

1.3 Why Internationalization?

The two main reasons for internationalizing software are:

1. To support a multi-cultural organization (which *need not* necessarily be multinational).
2. To allow the development of software for export to countries having a different culture. (Make no mistake, software export is a growing market.)

1.4 A Sampling of Cultural Differences

Many of us live almost exclusively in a one-culture world. However, once we start looking for differences in cultures, we find an astounding array. Here are the most common ones:

1. Alphabet and collating sequence.
2. What constitutes a letter, a number, a punctuation mark, and white space (if any).
3. Is the concept of upper- and lowercase letters supported? If so, do all uppercase letters have a lowercase counterpart and vice versa?
4. Display width and height of characters.
5. Length of translated text and size of display area.

6. What hyphenation rules (if any) exist?
7. Date and time formats, month names and abbreviations, time zones, Gregorian vs. Era years.
8. How are fractional numbers formatted? How is rounding handled? Negative numbers?
9. How are currency numbers formatted? What are the national and international currency symbols? How is rounding handled? Negative numbers?
10. Length of first and/or last names. Does the concept of first and last names and middle initial even exist?
11. Writing direction: left-to-right/right-to-left, top-to-bottom/bottom-to-top. A carriage return doesn't always go down one line and to the left margin! Can multiple writing directions be used together?
12. Address formats.
13. Titles and forms of address.
14. Telephone number formats.
15. Spelling of country and place names (i.e., Germany vs. Deutschland, Munich vs. München, and London vs. Londres).
16. Measurement system (metric vs. imperial).

Other requirements brought on by a change in culture are:

1. On-line help screens and printed documentation need to be translated. Remember, different dialects of the same language often use different vocabularies and colloquial terms present in one might be completely unacceptable in another.
2. On-screen forms and prompt and error messages need to be translated.
3. Voice messages need to be translated. Music and other audio sounds might have to be changed.
4. In applications written for English-speaking users it is common to use short-cut key combinations such as ALT-F to select a 'File' operations' menu. Of course this would not be intuitive to the user once the text 'File' has been translated to something not containg the letter F.

1.5 The Key to Achieving Internationalization

The three most important things to remember are:

ABSTRACTION

ABSTRACTION

ABSTRACTION

Abstraction involves the separation of a logical interface from a physical implementation or representation. The programming world is full of abstractions; for example we have variables, procedures, keywords, and named constants. Imagine programming without them! (One reason C and C++ have become so popular is that they provide more support for abstraction than do most other mainstream languages.)

Successful abstraction requires isolation of implementation-defined aspects. For example, since text messages are written in some particular language, they should not be hard-coded directly in program source.

It is important to consider at what level we want to achieve abstraction. From a software development point of view, there are three possible times at which this can be achieved. They are:

Compile time: It is possible (and generally considered desirable) to have only one source file containing the code for a given function. This is achieved by some form of conditional compilation in which decisions are made on which source lines should be passed through to the compiler. The advantage of this approach is that the executable contains only that code needed for a specific target. The disadvantage of this approach is that the executable contains only that code needed for a specific target; code must be recompiled to support a different target. We must maintain an inventory of target-specific versions. This approach requires that our target be static. That is, a user's target environment stays the same. For most users that will probably be true.

Link time: Consider the case of a sort function. It can be called in many places but, depending on which object library version of that function is provided at link time, the sort might or might not treat upper- and lowercase as being equivalent. Like the compile-time case, the advantage of this approach is that the executable contains only that code needed for a specific target. The disadvantage of this approach is that the executable contains only that code needed for a specific target; code must be relinked to support a different target. We must maintain an inventory of target-specific versions unless we are prepared to ship the different object libraries to customers and allow them to relink programs themselves to support a different target.

Run time: The advantage of this approach is that the executable is ready to handle any possibility. We no longer need to maintain an inventory of target-specific versions. The disadvantage of this approach is that the executable contains *all* the code needed for *all* supported targets. (Note, however, that ways exist to reduce such code bloat. For example, overlays that don't get called, are swapped out to disk so don't require memory.) The impact of supporting multiple environments, one at a time, can be significantly reduced, however, by making things table driven. For example, when the target changes at run time, a disk-based configuration file for that target is read into memory. In this manner, an arbitrary large set of targets can be supported, one at a time, and, given the proper tools, the user can even provide their own custom target. (An interesting question arises about the language in which configuration command names are written.)

All three approaches have their advantages and disadvantages; there is no one correct answer for every application. If we think about it for more than a few moments, we'll realize the following very important point:

A project requiring internationalization is one that involves some form of portability.

We can talk about portability from two points of view: generic and specific. Generically, portability simply means running a program in an environment that is somehow different from the one for which it was originally designed. Since the cost of producing and maintaining software far outweighs that for producing hardware, we have a huge incentive to increase the shelf life of our software beyond the current incarnations of hardware. It simply makes economic sense to do so.

Specific portability involves identifying the individual target environments in which a given program must run and clearly stating how those environments differ. Some examples of port scenarios follow:

1. Moving from one operating system to another on the same machine.
2. Moving from a version of an operating system on one machine to the same operating system on another machine.
3. Moving between variants of the same operating system.
4. Moving between two entirely different hardware and software environments.
5. Moving between different compilers on the same system.
6. Moving from one version of a compiler to another version of the same compiler (or, even more subtly, from one MS-DOS memory model to another).

Since abstraction is the key to portability, and internationalization is just another (albeit major) variable in the portability equation, abstraction is the key to achieving internationalization.

Technically, it is perfectly okay to limit target cultures to a specific subset. For example, we might chose only those whose alphabet can be represented in single-byte characters. This is no better or worse than deciding in a port project, to cater only for machines using the ASCII character set or twos-complement representation of negative integers.

We can't achieve something or measure our progress if we haven't defined it. And if we find we need something outside the framework of our original design, we either have to consider that a design limitation or set about changing

the design goals. Just as it would be naive, and probably not particularly cost effective, to design software to be portable across *all possible* targets that have ever existing or will ever exist, it is unlikely we really want to support *all possible* cultures.

1.6 Don't Reinvent the Wheel

There are two major sources of prior and emerging art:

1. That defined by national, international, or de-facto industry standards.
2. That proprietary to a particular vendor.

If we care about multiple hardware and/or operating system platforms, we'll probably care about standards. If all we care about is Microsoft's Windows current and future releases, for example, we likely won't be at all worried about committing to doing things their way.