Localization, internationalization

- Often want our product to be usable in wide range of countries, regions by diverse groups of clients
- Internationalization deals with designing our product to easily support that
- Localization deals with specifical support for a given region/culture/language (using our internationalized design)
- Can include adaptation of suitable alphabets and symbols, language translation, adapting symbols and images, adapting currency and date displays, necessary cultural adjustments, etc

Requires thought at many levels

- To be effective, should be thinking about this from early in design process
- Interfaces must be easily adaptable to different formats
- Right text, images, audio must be easily inserted for each supported region
- Testing needs to be performed across the different languages and versions

<clocale> in C++

- Library can be #included to help support use of correct currency, alphabet, time aspects, etc
- First capture a user's locale settings:

```
std::locale userloc = std::locale("");
```

- Then look up specific settings, such as currency (e.g. USD)

```
std::string curr = std::use_facet<std::moneypunct<char, true>>(loc).curr_symbol()
```

- Check out reference pages on locale for details on the various settings/options

Alphabets and unicode

- ASCII and EBCDIC each represented characters with a single byte of storage, limiting them to at most 256 possible characters, and each chose their own rules for which bit patterns represented which characters and symbols
- To support wider range of symbols, need agreement on a larger (and expandable) character set and which bit patterns represent which symbols
- unicode is the common choice, supported to different degrees by different programming languages

Unicode and character codes

- As with ascii/ebcdic, each bit pattern (often referred to by corresponding decimal or hex value) is matched to a specific symbol, patterns usually 1, 2, 4, 6, or 8 bytes long, but unicode expandable to larger ranges
- Huge range of alphabets and symbols can thus be supported, tables of which value ranges are used for which purpose (and which value for which symbol) widely published, e.g. see unicode.org/charts/
- Support by languages and browsers tends to lag a bit

Examples: hex code and symbol

- Tens of thousands of symbols assigned so far, just a tiny handful of examples here
- code 77 gives symbol w (codes 0-127 == ascii)
- code 1D120 gives symbol 𝄠
- code 20AC gives symbol €
- code 03A0 gives symbol Π
- code 2620 gives symbol ☠

At the programming language level

- Different languages and frameworks provide different degrees of support for localization and internationalization
- We'll pick a couple of tiny aspects and look at support in bash/C++
- Use of unicode to support wider range of character sets
- Use of libraries like <clocale> to handle text, currency, time

Unicode in bash

- Can use unicode in bash directly on command line by using sequence control-shift-u followed by the desired code, then hit enter
- Most editors also provide a means of typing in unicode chars
- Can display unicode in a bash printf using \\u then code, e.g. printf “\\u263A” # smiley face

Unicode in C++

- We can’t use char or string in C++ if we also want to use unicode, nor libraries like cctype
- Instead we include libraries cwchar and cwctype, and use types wchar_t and wstring, most i/o routines are overloaded with variants like wprintf, wcin, wcout, etc
- Can assign unicode chars by their hex code, e.g.

```
wchar_t c = 0x001f6; // or c = L'\x001f6';
```

- Use different literals for wide chars and strings, prefix with L, e.g to assign a wide string literal to a variable: *s = L”foo”;*

Tiny example

```
#include <iostream>
#include <string>
#include <clocale>
int main() {
   setlocale(LC_ALL, "");
   std::wstring prompt = L"Enter the char Q\n";
   std::wcout << prompt;  // wcin also supported
   wchar_t smiley = 0x0000263A;
   if (c == L'Q') std::wcout << L"good job!" << smiley << endl;
   else std::wcout << L"incorrect!\n";
}
```