

# Portability

CPSC 315 – Programming Studio  
Fall 2009

Material from The Practice  
of Programming, by Pike and Kernighan

# Portability

- Ability of software to run in more than one environment
  - Run the same with differing compilers
  - Run the same on different operating systems
- “Portable” often means it is easier to modify existing code than rewrite from scratch

## Why Focus on Portability?

- Some drawbacks to portability:
  - Known requirements don't specify it
  - Less efficient than less portable code
- But, requirements change
  - People will want to run successful programs in new places and ways
- Environments change
  - OS gets “upgraded” – we want the code to improve, also
- Code itself could be ported!
  - Java to C/C++
- Portability tends to reflect good programming

## General Principles

- Will never have “fully” portable code, but you can improve portability
- Try to use only the intersection of standards, interfaces, environments that it must support
- Don't add special code to handle new situations, instead adjust code to fit
- Abstraction and encapsulation help

## Language Issues

- Stick to Language Standards
  - Many languages aren't standardized, and no language is fully specified
  - Even such languages have very common usage patterns
- Program in the mainstream
  - Stick to language constructs that are well-understood
  - Don't use unusual language features or new language additions
  - Requires some familiarity with what "mainstream" is.

## Trouble Spots in Languages

- Sizes of data types
  - int, long, pointers can vary
  - Don't assume length, beyond very well established standards
    - e.g. 8 bits in a byte

## Trouble Spots in Languages

- Order of Evaluation
  - Often not clearly specified, or implemented differently anyway

```
ptr[count] = name[++count]
```

  - count could be incremented before or after used to increment ptr
  - Avoid reliance on specific order, even when the language specifies
    - Could port code, or compiler treat differently

## Trouble Spots in Languages

- "Sign" of a char
  - Could run -128 to 127, or 0 to 255
- Arithmetic and logical shifts
  - How is sign bit handled? shifted or not?
- Byte order
  - Big vs. Little endian

## Trouble Spots in Languages

- Alignment of structures and class members
  - Never assume that elements of a structure occupy contiguous memory.
  - Lots of machine-specific issues
    - e.g. n-byte types must start on n-byte boundaries
  - e.g. i could be 2, 4, or 8 bytes from start:

```
struct X {  
    char c;  
    int i;  
}
```

## Dealing with Language Issues

- General Rules of Thumb:
  - Don't use side effects
  - Compute, don't assume sizes of types/objects
  - Don't (right) shift signed values
  - Make sure data type is big enough for the range of values you will store
- Try several compilers

## Headers and Libraries

- Use standard libraries when available
  - Realize that these are not necessarily universal, though
  - Different implementations may have different “features”
- Careful about using lots of `#ifdefs` to catch language/environment changes
  - Easily leads to convoluted header files that are difficult to maintain
- Choose widely-used and well-established standards
  - networking interfaces
  - graphics interfaces

## Program Organization

- Use only features that are available in all target systems
- Avoid conditional compilation (`#ifdefs`)
  - Especially bad to mix compile-time with run-time commands
  - Makes it difficult to test on different systems, since changes actual program!

## Isolation

- Localize system dependencies in different files
  - e.g. single file to capture unix vs. Windows system calls.
  - Sometimes these system files can have a life/usefulness of their own
- Hide system dependencies behind interfaces
  - Good encapsulation, anyway
  - Java does this fully with virtual machine

## Data Exchange

- Text tends to provide good data exchange
  - Much more portable than binary
  - Still an issue of Carriage Return vs. Carriage Return and Line Feed
- Byte Order matters
  - Big vs. Little Endian is a real issue
  - Be careful in how you rely on it
- Use a fixed byte order for data exchange
  - Write bytes rather than larger formats

## Upgrading with Portability In Mind

- If function specification changes, change the function name
  - e.g.: The sum function (for checksum to see if files were transferred correctly) in Unix has changed implementations, making it nearly useless sometimes
- Maintain compatibility with earlier programs and data
  - Provide a write function, not just a read function for earlier data formats
  - Make sure there is a way to replicate the old function
- Consider whether “improvement” is worth it in terms of portability cost
  - Don’t “upgrade” function if it will provide only limited benefit, but cause portability problems.

## Internationalization

- International standards vary
- Don’t assume ASCII
  - Some character sets require thousands of characters
  - 8-bit vs. 16-bit characters
  - Unicode helps
- Careful about culture/language issues
  - Date and time format
  - Text field lengths
  - Idioms and slang
  - Icons