Portability

Professor Jennifer Rexford

http://www.cs.princeton.edu/~jrex

The material for this lecture is drawn, in part, from
The Practice of Programming (Kernighan & Pike) Chapter 8

1

Goals of this Lecture

- Learn to write code that works with multiple:
  - Hardware platforms
  - Operating systems
  - Compilers
  - Human cultures
- Why?
  - Moving existing code to a new context is easier/cheaper than writing new code for the new context
  - Code that is portable is (by definition) easier to move; portability reduces software costs
  - Relative to other high-level languages (e.g., Java), C is notoriously non-portable

2

The Real World is Heterogeneous

- Multiple kinds of hardware
  - 32-bit Intel Architecture
  - 64-bit IA, PowerPC, Sparc, MIPS, Arms, …
- Multiple operating systems
  - Linux
  - Windows, Mac, Sun, AIX, …
- Multiple character sets
  - ASCII
  - Latin-1, Unicode, …
- Multiple human alphabets and languages

3

Portability

- Goal: Run program on any system
  - No modifications to source code required
  - Program continues to perform correctly
    - Ideally, the program performs well too

4

C is Notoriously Non-Portable

- Recall C design goals…
  - Create Unix operating system and associated software
  - Reasonably "high level", but…
  - Close to the hardware for efficiency
- So C90 is underspecified
  - Compiler designer has freedom to reflect the design of the underlying hardware
- But hardware systems differ!
  - So C compilers differ
- Extra care is required to write portable C code

5

General Heuristics

Some general portability heuristics…

6

Intersection

(1) Program to the intersection

- Use only features that are common to all target environments
- I.e., program to the ***intersection*** of features, not the ***union***

- When that's not possible…

Encapsulation

(2) Encapsulate

- Localize and encapsulate features that are not in the intersection
- Use parallel ***source code*** files -- so non-intersection code can be chosen at ***link-time***
- Use parallel ***data*** files – so non-intersection data (e.g. textual messages) can be chosen at ***run-time***

- When that's not possible, as a last resort…

Conditional Compilation

(3) Use conditional compilation

```
#ifdef __UNIX__
  /* Unix-specific code */
#endif
…
#ifdef __WINDOWS__
   /* MS Windows-specific code */
#endif
…
```

- And above all…

Test!!!

(4) Test the program with multiple:

- Hardware (Intel, MIPS, SPARC, …)
- Operating systems (Linux, Solaris, MS Windows, …)
- Compilers (GNU, MS Visual Studio, …)
- Cultures (United States, Europe, Asia, …)

Hardware Differences

- Some **hardware** differences, and corresponding portability heuristics…

11

Natural Word Size

- Obstacle: Natural word size
  - In some systems, natural word size is 4 bytes
  - In some (esp. older) systems, natural word size is 2 bytes
  - In some (esp. newer) systems, natural word size is 8 bytes
- C90 intentionally does not specify sizeof(int); depends upon natural word size of underlying hardware

12

Natural Word Size (cont.)

(5) Don't assume data type sizes

- Not portable:

```
int *p;
…
p = malloc(4);
…
```

- Portable:

```
int *p;
…
p = malloc(sizeof(int));
…
```

13

Right Shift

- Obstacle: Right shift operation
  - In some systems, right shift operation is **logical**
    - Right shift of a negative signed int fills with **zeroes**
  - In some systems, right shift operation is **arithmetic**
    - Right shift of a negative signed int fills with **ones**
- C90 intentionally does not specify semantics of right shift; depends upon right shift operator of underlying hardware

14

Right Shift (cont.)

(6) Don't right-shift signed ints

- Not portable:

```
…
-3 >> 1
…
```

Logical shift => 2147483646
Arithmetic shift => -2

- Portable:

```
…
/* Don't do that!!! */
…
```

15

Byte Order

- Obstacle: Byte order
  - Some systems (e.g. Intel) use little endian byte order
    - Least significant byte of a multi-byte entity is stored at lowest memory address

The int 5 at address 1000:

1000	00000101
1001	00000000
1002	00000000
1003	00000000

  - Some systems (e.g. SPARC) use big endian byte order
    - Most significant byte of a multi-byte entity is stored at lowest memory address

The int 5 at address 1000:

1000	00000000
1001	00000000
1002	00000000
1003	00000101

16

Byte Order (cont.)

(7) Don't rely on byte order in code

- Not portable:

```
int i = 5;
char c;
…
c = *(char*)&i; /* Silly, but legal */
```

Little endian:
  c = 5
Big endian:
  c = 0;

- Portable:

```
int i = 5;
char c;
…
/* Don't do that! Or... */
c = (char)i;
```

17

Byte Order (cont.)

(8) Use text for data exchange

- Not portable:

Run on a ***little*** endian computer

```
unsigned short s = 5;
FILE *f = fopen("myfile", "w");
fwrite(&s, sizeof(unsigned short), 1, f);
```

`fwrite()` writes raw data to a file

myfile

```
00000101 00000000
```

`fread()` reads raw data from a file

Run on a ***big*** endian computer:
Reads 1280!!!

```
unsigned short s;
FILE *f = fopen("myfile", "r");
fread(&s, sizeof(unsigned short), 1, f);
```

18

Byte Order (cont.)

- Portable:

Run on a ***big or little*** endian computer

```
unsigned short s = 5;
FILE *f = fopen("myfile", "w");
fprintf(f, "%hu", s);
```

`fprintf()` converts raw data to ASCII text

myfile `00110101` ASCII code for '5'

`fscanf()` reads ASCII text and converts to raw data

Run on a ***big or little*** endian computer: Reads 5

```
unsigned short s;
FILE *f = fopen("myfile", "r");
fscanf(f, "%hu", &s);
```

19

Byte Order (cont.)

If you must exchange raw data…

(9) Write and read one byte at a time

Run on a ***big or little*** endian computer

```
unsigned short s = 5;
FILE *f = fopen("myfile", "w");
fputc(s >> 8, f);    /* high-order byte */
fputc(s & 0xFF, f); /* low-order byte */
```

Decide on big-endian data exchange format

myfile `00000000 00000101`

Run on a ***big or little*** endian computer: Reads 5

```
unsigned short s;
FILE *f = fopen("myfile", "r");
s = fgetc(f) << 8;     /* high-order byte */
s |= fgetc(f) & 0xFF; /* low-order byte */
```

20

OS Differences

- Some **operating system** differences, and corresponding portability heuristics…

End-of-Line Characters

- Obstacle: Representation of “end-of-line”
  - **Unix** (including Mac OS/X) represents end-of-line as 1 byte: **00001010** (binary)
  - **Mac OS/9** represents end-of-line as 1 byte: **00001101** (binary)
  - **MS Windows** represents end-of-line as 2 bytes: **00001101 00001010** (binary)

End-of-Line Characters (cont.)

(10) Use binary mode for textual data exchange

- Not portable:

Open the file in ordinary ***text*** mode

```
FILE *f = fopen("myfile", "w");
fputc('\n', f);
```

Run on Unix | Run on Mac OS/9 | Run on MS Windows

`00001010` — `\n`

`00001101` — `\r`

`00001101 00001010` — `\r` `\n`

- Trouble if read via fgetc() on "wrong" operating system

23

End-of-Line Characters (cont.)

- Portable:

Open the file in ***binary*** mode

```
FILE *f = fopen("myfile", "wb");
fputc('\n', f);
```

Run on Unix, Mac OS/9, or MS Windows

`00001010` — `\n`

- No problem if read via fgetc() in binary mode on "wrong" operating system
- I.e., there is no "wrong" operating system!

24

Data Alignment

- Obstacle: Data alignment
  - Some hardware requires data to be aligned on particular boundaries
  - Some operating systems impose additional constraints:

OS	char	short	int	double
Linux	1	2	4	4
MS Windows	1	2	4	8

Start address must be evenly divisible by:

  - Moreover…
  - If a structure must begin on an x-byte boundary, then it also must end on an x-byte boundary
    - Implication: Some structures must contain padding

25

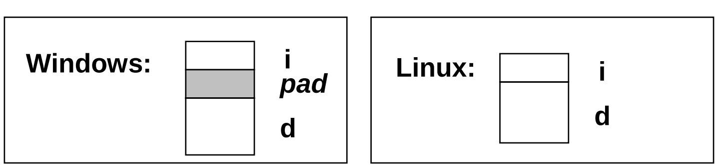
Data Alignment (cont.)

(11) Don't rely on data alignment

- Not portable:

```
struct S {
   int i;
   double d;
}
…
struct S *p;
…
p = (struct S*)malloc(sizeof(int)+sizeof(double));
```

Allocates 12 bytes; too few bytes on MS Windows



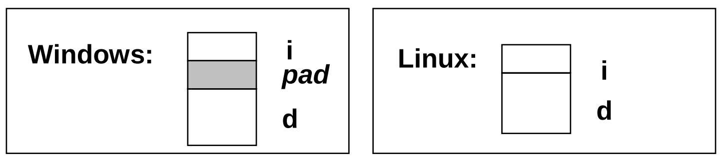
26

Data Alignment (cont.)

- Portable:

```
struct S {
   int i;
   double d;
}
…
struct S *p;
…
p = (struct S*)malloc(sizeof(struct S));
```

Allocates
- 12 bytes on Linux
- 16 bytes on MS Windows



27

Character Codes

- Obstacle: Character codes
  - Some operating systems (e.g. IBM OS/390) use the EBCDIC character code
  - Some systems (e.g. Unix, MS Windows) use the ASCII character code

28

Character Codes (cont.)

(12) Don't assume ASCII

- Not portable:

```
if ((c >= 65) && (c <= 90)) …
```

Assumes ASCII

- A little better:

```
if ((c >= 'A') && (c <= 'Z')) …
```

Assumes that uppercase char codes are contiguous; not true in EBCDIC

- Portable:

```
#include <ctype.h>
…
if (isupper(c)) …
```

For ASCII:
(c >= 'A') && (c <= 'Z')
For EBCDIC:
((c >= 'A') && (c <= 'I'))
|| ((c >= 'J') && (c <= 'R'))
|| ((c >= 'S') && (c <= 'Z'))

29

Compiler Differences

- Compilers may differ because they:
  - Implement underspecified features of the C90 standard in different ways, or
  - Extend the C90 standard
- Some **compiler** differences, and corresponding portability heuristics…

30

Compiler Extensions

- Obstacle: Non-standard extensions
  - Some compilers offer non-standard extensions

31

Compiler Extensions

(13) Stick to the standard language

- For now, stick to C90 (not C99)
- Not portable:

```
…
for (int i = 0; i < 10; i++)
   …
```

Many systems allow definition of loop control variable within **for** statement, but a C90 compiler reports error

- Portable:

```
int i;
…
for (i = 0; i < 10; i++)
   …
```

32

Evaluation Order

- Obstacle: Evaluation order
  - C90 specifies that side effects and function calls must be completed at "**;**"
  - But multiple side effects within the same expression can have unpredictable results

Evaluation Order (cont.)

(14) Don't assume order of evaluation

- Not portable:

```
strings[i] = names[++i];
```

i is incremented before indexing **names**; but has **i** been incremented before indexing **strings**?
C90 doesn't say

- Portable (either of these, as intended):

```
i++;
strings[i] = names[i];
```

```
strings[i] = names[i+1];
i++;
```

Evaluation Order (cont.)

- Not portable:

Which call of **getchar()** is executed first? C90 doesn't say

```
printf("%c %c\n", getchar(), getchar());
```

- Portable (either of these, as intended):

```
i = getchar();
j = getchar();
printf("%c %c\n", i, j);
```

```
i = getchar();
j = getchar();
printf("%c %c\n", j, i);
```

35

Char Signedness

- Obstacle: Char signedness
  - C90 does not specify signedness of char
  - On some systems, char means signed char
  - On other systems, char means unsigned char

36

Char Signedness (cont.)

(15) Don't assume signedness of char

- If necessary, specify "signed char" or "unsigned char"
- Not portable:

```
int a[256];
char c;
c = (char)255;
…
… a[c] …
```

If char is unsigned, then a[c] is a[255]
=> fine
If char is signed, then a[c] is a[-1]
=> out of bounds

- Portable:

```
int a[256];
unsigned char c;
c = 255;
…
… a[c] …
```


Char Signedness (cont.)

- Not portable:

```
int i;
char s[MAX+1];
for (i = 0; i < MAX; i++)
   if ((s[i] = getchar()) == '\n') || (s[i] == EOF))
      break;
s[i] = '\0';
```

If char is unsigned, then this always is FALSE

- Portable:

```
int c, i;
char s[MAX+1];
for (i = 0; i < MAX; i++) {
   if ((c = getchar()) == '\n') || (c == EOF))
      break;
   s[i] = c;
}
s[i] = '\0';
```

Library Differences

- Some **library** differences, and corresponding portability heuristics…

39

Library Extensions

- Obstacle: Non-standard functions
  - "Standard" libraries bundled with some development environments (e.g. GNU, MS Visual Studio) offer non-standard functions

40

Library Extensions

(16) Stick to the standard library functions

- For now, stick to the C90 standard library functions
- Not portable:

```
char *s = "hello";
char *copy;
…
copy = strdup(s);
…
```

`strdup()` is available in many "standard" libraries, but is not defined in C90

- Portable:

```
char *s = "hello";
char *copy;
…
copy = (char*)malloc(strlen(s) + 1);
strcpy(copy, s);
…
```


Cultural Differences

- Some **cultural** differences, and corresponding portability heuristics…

Character Code Size

- Obstacle: Character code size
  - United States
    - Alphabet requires 7 bits => 1 byte per character
    - Popular character code: ASCII
  - Western Europe
    - Alphabets require 8 bits => 1 byte per character
    - Popular character code: Latin-1
  - China, Japan, Korea, etc.
    - Alphabets require 16 bits => 2 bytes per character
    - Popular character code: Unicode

43

Character Code Size

(17) Don’t assume 1-byte character code size

- Not portable:

```
char c = 'a';
```

- Portable:
  - C90 has ***no good solution***
  - C99 has “wide character” data type, constants, and associated functions

```
#include <stddef.h>
…
wchar_t c = L'\x3B1'; /* Greek lower case alpha */
```

  - But then beware of byte-order portability problems!
  - Future is not promising

44

Human Language

- Obstacle: Humans speak different natural languages!

45

Human Language (cont.)

(18) Don't assume English

- Not portable:

```
/* somefile.c */

…
printf("Bad input");
…
```

- Can't avoid natural language! So…

46

Human Language (cont.)

- Encapsulate code

```
/* somefile.c */

#include "messages.h"
…
printf(getMsg(5));
…
```

- Choose appropriate "message.c" file at ***link-time***

Messages module, with multiple implementations

```
/* messages.h */
char *getMsg(int msgNum);
```

```
/* englishmessages.c */
char *getMsg(int msgNum) {
   switch(msgNum) {
      …
      case 5:
         return "Bad input";
      …
   }
}
```

```
/* spanishmessages.c */
char *getMsg(int msgNum) {
   switch(msgNum) {
      …
      case 5:
         return "Mala entrada";
      …
   }
}
```

47

Human Language (cont.)

- Maybe even better: encapsulate ***data***

Messages module

```
/* messages.h */
char *getMsg(int msgNum);
```

```
/* messages.c */

enum {MSG_COUNT = 100};
char *getMsg(int msgNum) {
   static char *msg[MSG_COUNT];
   static int firstCall = 1;
   if (firstCall) {
      <Read all messages from
       appropriate messages.txt
       file into msg>
      firstCall = 0;
   }
   return msg[msgNum];
}
```

```
/* englishmessages.txt */

…
Bad input
…
```

```
/* spanishmessages.txt */

…
Mala entrada
…
```

- Choose appropriate "message.txt" file at ***run-time***

48

Summary

- General heuristics
  - (1) Program to the intersection
  - (2) Encapsulate
  - (3) Use conditional compilation (as a last resort)
  - (4) Test!!!

49

Summary (cont.)

- Heuristics related to **hardware** differences
  - (5) Don’t assume data type sizes
  - (6) Don’t right-shift signed ints
  - (7) Don’t rely on byte order in code
  - (8) Use text for data exchange
  - (9) Write and read 1 byte at a time
- Heuristics related to **OS** differences
  - (10) Use binary mode for textual data exchange
  - (11) Don’t rely on data alignment
  - (12) Don’t assume ASCII

50

Summary (cont.)

- Heuristics related to **compiler** differences
  - (13) Stick to the standard language
  - (14) Don't assume evaluation order
  - (15) Don't assume signedness of char
- Heuristic related to **library** differences
  - (16) Stick to the standard library
- Heuristics related to **cultural** differences
  - (17) Don't assume 1-byte char code size
  - (18) Don't assume English

51