Writing Portable Programs

Prof. David August

COS 217

Goals of Today's Class

- Writing portable programs in C
  - Sources of heterogeneity
  - Data types, evaluation order, byte order, char set, …
- Reading period and final exam
  - Important dates
  - Practice exams
- Lessons from COS 217
  - Course wrap-up

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 byte orderings
  - Little endian
  - Big endian

Portability

- Goal: run program on any other system
  - Do not require any modifications to the program at all
    - Simply recompile the program, and run
  - Program should continue to perform correctly
    - Ideally, the program should perform well, too.
- Portability is hard to achieve
  - Wide variation in computing platforms
  - Patches and releases are frequent operations
- Normally, portability is difficult to achieve
  - Still, good to make programs as portable as possible
  - This requires extra care in writing and testing code

Programming Language

- Stick to the standard
  - Program in a high-level language and stay within the language standard
  - However, the standard may be incomplete
    - E.g., `char` type in C and C++ may be signed or unsigned
- Program in the mainstream
  - Mainstream implies the established style and use
    - Program enough to know what compilers commonly do
    - Difficult for large languages such as C++
- Beware of language trouble spots
  - Some features are intentionally undefined to give compiler implementers flexibility

Size of Data Types

- What are the sizes of `char, short, int, long, float and double` in C and C++?
  - `char` has at least 8 bits, `short` and `int` at least 16 bits
  - `sizeof(char)` ≤ `sizeof(short)` ≤ `sizeof(int)` ≤ `sizeof(long)`
  - `sizeof(float)` ≤ `sizeof(double)`
- In Java, sizes are defined
  - `byte`: 8 bits
  - `char`: 16 bits
  - `short`: 16 bits
  - `int`: 32 bits
  - `long`: 64 bits
- **Our advice: always use `sizeof()` to be safe**

Order of Evaluation

- Order of evaluation may be ambiguous
  - `strings[i] = names[++i];`
    - `i` can be incremented before or after indexing `strings`!
  - `printf("%c %c\n", getchar(), getchar());`
    - The second character in `stdin` can be printed first!
- What are the rules in C and C++?
  - Side effects and function calls must be completed at "**;**"
  - && and || execute left to right, only as far as necessary
- What about Java?
  - Expressions including side effects evaluated left to right
- **Our advice: do not depend on the order of evaluation in an expression**

Characters Signed or Unsigned?

- Char type may be signed or unsigned
  - Either a 7-bit or an 8-bit character
- Code that is *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
  - **`s[i]`** is **`255`**, but **`EOF`** is **`-1`**
  - Hence, the program will hang!

Portable Version Using Integers

- Solution
  - Use an integer to store the output of getchar()
- Portable C code

```
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';
```

Other C Language Issues

- Arithmetic or logical shift
  - C: signed quantities with >> may be arithmetic or logical
    - What is "**`-3 >> 1`**"?
    - Does it shift-in a sign bit (i.e., a 1) or a 0?
  - Java: >> for arithmetic right shift, and >>> for logical
- Byte order
  - Byte order within **`short`**, **`int,`** and **`long`** is not defined

Alignment of Structures and Unions

- Structure consisting of multiple elements

```
struct Foo {
   char x;
   int y;
}
```

- Items are laid out in the order of declaration
- But, the alignment is undefined
  - There might be holes between the elements
  - E.g., **`y`** may be 2, 4, or 8 bytes from **`x`**

Use Standard Libraries

- Pre-ANSI C may have calls not supported in ANSI C
  - Program will break if you continue use them
  - Header files can pollute the name space
- Consider the signals defined
  - ANSI C defines 6 signals
  - POSIX defines 19 signals
  - Most UNIX defines 32 or more
- Take a look at **`/usr/include/*.h`** to see the conditional definitions

Avoid Conditional Compilation

- Writing platform-specific code is possible

```
…
some common code
#ifdef MAC
…
#else
#ifdef WINDOWSXP
…
#endif
#endif
```

- But, #ifdef code is difficult to manage
  - Platform-specific code may be all over the place
  - Plus, each part requires separate testing

Isolation

- Common feature may not always work: Life is hard
- Localize system dependencies in separate files
  - Separate file to wrap the interface calls for each system
  - Example: unix.c, windows.c, mac.c, …
- Hide system dependencies behind interfaces
  - Abstraction can serve as the boundary between portable and non-portable components
- Java goes one big step further
  - Virtual machine which abstracts the entire machine
  - Independent of operating systems and the hardware

Data Exchange

- Use ASCII text
  - Binary is often not portable
- Still need to be careful
  - But, even with text, not all systems are the same
    - Windows systems use '**\r**' or '**\n**' to terminate a line
    - UNIX uses only '**\n**'
  - Example
    - Use Microsoft Word and Emacs to edit files
    - CVS assumes all lines have been changed and will merge incorrectly
  - Use standard interfaces which will deal CRLF (carriage-return and line feed) and newline in a consistent manner

Byte Order: Big and Little Endian

- Example interaction between two machines
  - One process writes a short to outbound socket:

```
unsigned short x;
x = 0x1000;
...
write(sockOut, &x, sizeof(x));
```

  - Later, another process reads it from inbound socket:

```
unsigned short x;
...
read(sockIn, &x, sizeof(x));
```

- What is the value of **x** after reading?

Byte Order Solutions

- Fix the byte order for data exchange
  - Sender:

```
unsigned short x;
putchar(x >> 8);     /* high-order byte */
putchar(x & 0xFF);   /* low-order byte */
```

  - Receiver:

```
unsigned short x;
x = getchar() << 8;     /* high-order */
x |= getchar() & 0xFF;  /* low-order */
```

- Extremely important for network protocols

More on Byte Order

- Language solution
  - Java has a serializable interface that defines how data items are packed
  - C and C++ require programmers to deal with the byte order
- Binary files vs. text files
  - Binary mode for text files
    - No problem on UNIX
    - Windows will terminate reading once it sees Ctrl-Z as input

Internationalization

- Don’t assume ASCII
  - Many countries do not use English
  - Asian languages use 16 bits per character
- Standardizations
  - Latin-1 augments ASCII by using all 8 bits
  - Unicode uses 16 bits per character
  - Java uses Unicode as its native character set for strings
- Issues with Unicode
  - Byte order issue!
  - Solution: use UTF-8 as an intermediate representation or define the byte order for each character

Summary on Portability

- Language
  - Don’t assume **`char`** signed or unsigned
  - Always use **`sizeof()`** to compute the size of types
  - Don’t depend on the order of evaluation of an expression
  - Beware of right shifting a signed value
  - Make sure that the data type is big enough
- Use standard interfaces
  - Use the common features where possible
  - Provide as much isolation as possible
- Byte order
  - Fix byte order for data exchange
- Internationalization
  - Don’t assume ASCII and English

Important Dates

- Tuesday January 16 (Dean’s Date)
  - Execution Profiler Assignment due
- Final Exam
  - **DATE:** 01/25/2007
  - **START TIME:** 9:00 AM
  - **LOCATION:** Friend Center 101
  - Open books, notes, slides, mind, etc.
  - A little secret…

Practice Final Exams

- Go online for old exams and answers
- We recommend you take some practice exams
  - And then look at the answers afterwards
  - Note that some material differs from term to term
- Also, ask questions about the practice exams
  - On the listserv
  - To me or preceptor, in person
  - To each other

Wrap Up: Goals of COS 217

- Understand boundary between code and computer
  - Machine architecture
  - Operating systems
  - Compilers

- Learn C and the Unix development tools
  - C is widely used for programming low-level systems
  - Unix has a rich development environment
  - Unix is open and well-specified, good for study & research
- Improve your programming skills
  - More experience in programming
  - Challenging and interesting programming assignments
  - Emphasis on modularity and debugging

Relationship to Other Courses

- Machine architecture
  - Logic design (306) and computer architecture (471)
  - COS 217: assembly language and basic architecture
- Operating systems
  - Operating systems (318)
  - COS 217: virtual memory, system calls, and signals
- Compilers
  - Compiling techniques (320)
  - COS 217: compilation process, symbol tables, assembly and machine language
- Software systems
  - Numerous courses, independent work, etc.
  - COS 217: programming skills, UNIX tools, and ADTs

Lessons About Computer Science

- Modularity
  - Well-defined interfaces between components
  - Allows changing the implementation of one component without changing another
  - The key to managing complexity in large systems
- Resource sharing
  - Time sharing of the CPU by multiple processes
  - Sharing of the physical memory by multiple processes
- Indirection
  - Representing address space with virtual memory
  - Manipulating data via pointers (or addresses)

Lessons Continued

- Hierarchy
  - Memory: registers, cache, main memory, disk, tape, …
  - Balancing the trade-off between fast/small and slow/big
- Bits can mean anything
  - Code, addresses, characters, pixels, money, grades, …
  - Arithmetic is just a lot of logic operations
  - The meaning of the bits depends entirely on how they are accessed, used, and manipulated
- Capturing a human’s intent is really hard
  - Precise specification of a problem is challenging
  - Correct and efficient implementation of a solution is, too