Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

Character Codes and Error Detection

Homework #1 Comments

- Grades returned via Sakai
- **Question: "Do I *have* to show my work?"**
- **Answer: No, but…**
- My instructions to grader
  - If(correct) then points=100%
  - If(incorrect && no-work) then points=0%
  - If(incorrect && work-shown) then points=50-99%

Homework #1 Comments

- 50-word Problem
- *"Processors spend an increasing amount of time idle while waiting for instructions and data to be fetched from memory because the Von Neumann bottleneck allows the system to only retrieve one element at a time."*

Character Codes

Character Codes

- Goal of signed/unsigned/floating-point systems
  - Store numbers for **computer** processing
- Goal of character codes
  - Store numbers/characters in the machine, but for **humans**
- Character codes have evolved alongside computers
  - Larger memories and storage devices permit richer character codes
  - The earliest computer coding systems used six bits
    - Fewest bits possible to represent digits (0-9), capital letters (A-Z), symbols

Character Codes - BCD

- Only 4 bits
- Sparse or **packed**
  - Sparse – Put one BCD digit in a byte / pad with 0's
  - Packed – Put two BCD digits in one byte

Digit	BCD (4 bits)
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
Unsigned Number	1111
Positive Number	1100
Negative Number	1101

Character Codes - EBCDIC

- Extended Binary-Coded Decimal Interchange Code (EBCDIC)
  - Introduced in 1964
  - Extension of BCD – 8 bits instead of 4
- EBCDIC was support amazing new features!
  - Upper **and** lowercase alphabetic characters!
  - Punctuation!
  - Control characters!
- EBCDIC and BCD are still in use by IBM mainframes today

Character Codes - ASCII

- American Standard Code for Information Interchange (ASCII)
  - Chosen by *non-IBM* companies
  - 7 bits – 8th bit could be used for **parity**
    - Even # of bits = 0, odd # of bits = 1
  - See table 2.7 in textbook (page 90) for conversion between a letter ('a') and its code (97_{10} or 61_{16})
- ASCII was the dominant character code outside the IBM mainframe world
  - … until Unicode finally become widespread

Character Codes - Unicode

- What about **non-latin** characters?
- **Unicode**
  - 16-bits system allows for 65536 unique characters
  - All the Western (Latin, Cyrillic, Greek) characters, 4096 different symbols, 48,000+ Eastern (Chinese/ Japanese/Korean) characters, etc…
- Sufficient for **every** character of **every** language?
  - Not quite – but an optional extension allows for a million extra characters…

Character Codes - Unicode

- The Unicode codespace allocation is shown at the right.
- The lowest-numbered Unicode characters comprise the ASCII code
- The highest provide for user-defined codes

Character Types	Language	Number of Characters	Hexadecimal Values
Alphabets	Latin, Greek, Cyrillic, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation.	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Han Expansion	4096	E000 to EFFF
User Defined		4095	F000 to FFFE

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>

int main()
{
  printf("Data Representation Demo Program\n");

  // Variables to examine in memory
  int myInteger = 255;                        // 0x000000FF (4 bytes)
  int myInteger2 = -255;                      // 0xFFFFFF01 (4 bytes)
  unsigned int myUnsignedInt = 2882400152; // 0xABCDEF98 (4 bytes)
  float myFloat = 3.75;                       // 0x40700000 (4 bytes)
  double myDouble = 3.75;                     // 4000xE000000000000 (8 bytes)
  char myChar = 'a';                          // ASCII(a)=0x61 (1 byte)
  wchar_t myWideChar = L'a';                  // UNICODE(a)=0x00000061

  uint8_t* ptr;
  int i;

  printf("--myInteger--\n");
  ptr=&myInteger;
  for(i=0; i<sizeof(myInteger); i++)
    {
      printf("Mem[%X]=%02X\n", ptr, *ptr);
      ptr++;
    }
    // etc...
```

```
Data Representation Demo Program
--myInteger--
Mem[CD956098]=FF
Mem[CD956099]=00
Mem[CD95609A]=00
Mem[CD95609B]=00
--myInteger2--
Mem[CD956094]=01
Mem[CD956095]=FF
Mem[CD956096]=FF
Mem[CD956097]=FF
--myUnsignedInt--
Mem[CD956090]=98
Mem[CD956091]=EF
Mem[CD956092]=CD
Mem[CD956093]=AB
--myFloat--
Mem[CD95608C]=00
Mem[CD95608D]=00
Mem[CD95608E]=70
Mem[CD95608F]=40
```

Variable being printed

Memory address

Byte at that address

```
--myDouble--
Mem[CD956078]=00
Mem[CD956079]=00
Mem[CD95607A]=00
Mem[CD95607B]=00
Mem[CD95607C]=00
Mem[CD95607D]=00
Mem[CD95607E]=0E
Mem[CD95607F]=40
--myChar--
Mem[CD95609F]=61
--myWideChar--
Mem[CD956088]=61
Mem[CD956089]=00
Mem[CD95608A]=00
Mem[CD95608B]=00
```

Error Detection / Correction

Error Detection and Correction

- **Why worry - aren't modern computers perfectly reliable?**
- 100% perfection is physically impossible for any data recording or transmission medium over its entire expected useful life
- 100% perfection gets harder to achieve as technology improves
  - Smaller bits on hard drive or memory – easier for cosmic rays to flip from 0<->1
  - Faster transmission of data on network – harder to tell the difference between a 0 and 1

Error Detection and Correction

- Check digits / parity bits – appended to the end of a long number – can provide some protection against data input errors
  - The last characters of UPC barcodes and ISBNs are check digits
- Longer data streams require more economical and sophisticated error detection mechanisms
- Cyclic redundancy checking (CRC) codes provide error detection for large blocks of data

Error Detection and Correction

- Data transmission errors are easy to fix once an error is detected
  - Just ask the sender to **re-transmit** the same data again
- In computer memory and data storage, however, this cannot be done
  - Too often the only copy of something important is in memory or on disk
- Thus, to provide data integrity over the long term, **error correcting codes** are required.

Error Detection and Correction

- **Hamming codes** and **Reed-Solomon codes** are two important error correcting codes
- Hamming codes can detect/correct a single bit error, and detect (but not correct) a two-bit error
  - Frequently used in RAM chips where errors are rare (but not rare enough!)
- Reed-Solomon codes are useful in correcting burst errors that occur when a series of adjacent bits are damaged
  - Scratch on a CD/DVD
  - Burst of static on a DSL phone line