**Did you hear the one about the infinite recursion which walked into bar, and the bartender says "Did you hear about infinite recursion which walk into the bar, and the bartender says...**

David Notkin ● Autumn 2009 ● CSE303 Lecture 23

## A puzzle...

- A king wishes to throw a grand party tomorrow in his castle. He has purchased 1000 bottles of wine to serve to his many guests.
- However, a thief has been caught breaking into the wine cellar! He poisoned a single bottle. The poison is lethal at even the smallest dose; it causes death within approximately 12-15 hours.
  - The king wants to find out which bottle has been poisoned and throw it out so that his guests will not be harmed.
- The king has over 1000 servants to help him, and a few dozen prisoners in his dungeon, but he does not want to risk servant lives if possible. The prisoners are vermin and may be sacrificed.
  - How should the king find the poisoned bottle?
- Hint: First solve it with 4 bottles of wine and 2 prisoners.

## The answer

- Number each bottle from 1 to 1000.
  - Convert the bottle numbers to ten-digit binary numbers, from 1 (00000001) to 1000 (1111101000)
- Consider each of the 10 prisoners to represent one of the ten bits
- Each prisoner will drink from multiple bottles.
  - Prisoner i will drink every bottle for which bit i is 1.
- The pattern of dead prisoners tells you which bottle was poisoned.
  - If prisoners 1, 3, and 7 die, bottle # (512 + 128 + 8) = 648 was bad.
- Moral : Tightly packed data can be a good thing to avoid waste.

## Motivation

- C was developed with systems programming in mind
  - lean, mean, fast, powerful, unsafe
  - pointers provide direct access to memory
- C is often used in resource-constrained situations
  - devices without much memory
  - devices with slow processors
  - devices with slow network connections
- it is sometimes necessary to manipulate individual bits of data
  - "twiddle with bits"
  - "bit packing"

## Terms

- **nibble**: 4 bits
- **byte**: 8 bits (also sometimes called an "octet")
- **word**: size of an integer on a given machine (often 32 bits)
- **hword**: 16 bits ("half word")
- **dword**: two words long ("double word", "long word")
  - How many unique values can be stored in a bit? A nibble? A byte?
  - How many unique values can be stored using N bits?

## Bases, number systems

- decimal (base-10) `int x1 = 42;`
  - most natural to most humans
- binary (base-2)
  - how the computer stores data
- hexadecimal (base-16) `int x2 = 0x2a;`
  - memory addresses
  - each digit maps to 4 bits; concise
- octal (base-8) `int x3 = 052;`
  - chmod permissions
  - each digit maps directly to 3 bits; no special number symbols used

## Binary representations

- ints are stored as 32-bit (4-byte) integers
  - **42**

| 00000000 | 00000000 | 00000000 | 00101010 |
|---|---|---|---|

  - **int y = 1+128+256+4096+32768+131072;**

| 00000000 | 00000010 | 10010001 | 10000001 |
|---|---|---|---|

  - the maximum positive int value that can be stored is $2^{31} - 1$
    - **int z = 2147483647;**

| 01111111 | 11111111 | 11111111 | 11111111 |
|---|---|---|---|

## Negative binary numbers

- left most bit is the "sign bit"; 0 for positive, 1 for negative
  - all 1s represents -1 ; subsequent negatives grow "downward" -- *two's complement representation*

| 11111111 | 11111111 | 11111111 | 11111111 |
|---|---|---|---|

| 11111111 | 11111111 | 11111111 | 11111110 |
|---|---|---|---|

| 11111111 | 11111111 | 11111111 | 11111101 |
|---|---|---|---|

  a single 1 followed by all zeros represents $-2^{31}$
  - int z = -2147483648; // largest negative value

| 10000000 | 00000000 | 00000000 | 00000000 |
|---|---|---|---|

## Negating in binary

- Negating a binary number
  - "ones complement" : flip the bits
  - **twos complement : flip the bits, add 1**
- Converting a negative number from decimal to binary and back
  - add 1, then convert absolute value to binary, then flip bits
  - binary to decimal: flip bits, convert to decimal, subtract 1

```
int x = -27;    // -27 + 1       = -26
                //  26 base 2    = 11010
                //  flip         = 00101
```

| 11111111 | 11111111 | 11111111 | 11100101 |
|---|---|---|---|

## Bitwise operators

| expression | description |
|---|---|
| ***a* & *b*** | AND ; all bits that are set to 1 in both ***a*** and ***b*** |
| ***a* \| *b*** | OR ; all bits that are set to 1 in ***a*** or in ***b*** or both |
| ***a* ^ *b*** | XOR ; all bits that are set to 1 in ***a*** or in ***b*** but not in both |
| ***~a*** | NOT ; the "ones complement" of the bits of ***a*** (all bits flipped) |
| ***a* << *n*** | LEFT SHIFT ; moves all digits to the left by n places; same as multiplying ***a*** * $2^n$ |
| ***a* >> *n*** | RIGHT SHIFT ; moves all digits to the right by n places; same as dividing ***a*** / $2^n$ |

- left shift pads remaining right digits with 0
- right shift pads w/ 0 or value of ***a***'s leftmost (most significant) bit
- most operators can be used with =, such as &=, ~=, >>=
- what is the difference between & and && ? ~ and ! ?

## AND, OR, XOR, NOT

| bit1 | bit2 | bit1 & bit2 | bit1 \| bit2 | bit1 ^ bit2 | bit1 & ~bit2 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

| | *64* | *32* | *16* | *8* | *4* | *2* | *1* |
|---|---|---|---|---|---|---|---|
| **25** | **0** | **0** | **1** | **1** | **0** | **0** | **1** |
| **77** | **1** | **0** | **0** | **1** | **1** | **0** | **1** |
| | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

- What is 25 & 77 ?
- What is 25 | 77 ?
- What is 25 ^ 77 ?
- What is 25 & ~77 ?

## Bit shifting

- Shifting left is like multiplying by powers of 2:
  - **int x = 42;  // 101010**
  - **int y = x << 1;  // 1010100 ( 84 = 42 * 2)**
  - **int z = x << 3;  // 1010100000 (336 = 42 * 8)**
  - **int w = x << 31;  // 0 (why?)**
- Shifting right is like dividing by powers of 2:
  - **int x = 42;  // 101010**
  - **int y = x >> 1;  // 10101 (21)**
  - **x = -42;  // 111111...010110**
  - **int z = x >> 1;  // 1111111...01011 (-21)**

## Exercises

- Write functions to do the following tasks:
  - print an integer in binary
  - rotate bits by n places
  - get/set a given bit from a given integer
  - get/set a given range of bits from a given integer
  - invert a given bit(s) of a given integer
  - Should these be functions or preprocessor macros?

## Unsigned integers

- **`unsigned int x = 42u;`**
- changes interpretation of meaning of bits; no negatives allowed
- maximum is twice as high (leftmost bit not used to represent sign)
- right-shift behavior not same (pads w/ 0 instead of sign bit)
- seen in some libraries (size_t, malloc, etc.)
- often used with bit-packing because we don't care about sign
- why not use unsigned more often?
- really, it's all just bits in the end...

## Bit packing

- bit packing: storing multiple values in the same word of memory
  - example: storing a student's id, year, and exam score in a single int
- boolean (bool) values could really be just 1 bit (0 or 1)
  - "bit flags"
  - but a bool is actually a 1-byte integer value (Why?)
- integers known to be small could use fewer than 32 bits
  - example: student IDs, 7 digits (how many bits?)
  - example: homework/exam scores, up to 100 (how many bits?)

## Bit flags

```
#define REGISTERED 0x1
#define FULLTIME 0x2
#define PAIDTUITION 0x4
#define ACADEMICPROBATION 0x8
#define HONORROLL 0x10   // 16
#define DEANSLIST 0x20   // 32
...

int student1 = 0;

// set student to be registered and on honor roll
student1 = student1 | REGISTERED | HONORROLL;

// make sure student isn't on probation
student1 = student1 & ~ACADEMICPROBATION;
```

## Bit fields

```
typedef struct name {
    unsigned name1 : bitsWide;
    ...
    unsigned name2 : bitsWide;
} name;
```

- declares a field that occupies exactly **`bitsWide`** bits
- can be declared only inside a **`struct`**
- exact ordering of bits is compiler-dependent
- can't make pointers to them; not directly addressable

## Binary data I/O

| function | description |
|---|---|
| `size_t fwrite(void* ptr, size_t size, size_t count, FILE* file)` | writes given number of elements from given array/buffer to file *(size_t means unsigned int)* |
| `size_t fread(void* ptr, size_t size, size_t count, FILE* file)` | reads given number of elements to given array/buffer from file |

```
// writing binary data to a file
int values[5] = {10, 20, 30, 40, 50};
FILE* f = fopen("saved.dat", "w");
fwrite(values, sizeof(int), 5, f);

// reading binary data from a file
int values[5];
FILE* f = fopen("saved.dat", "r");
fread(values, sizeof(int), 5, f);
```

## Questions?