Introduction to C

Performance

Instructor: Yin Lou

02/07/2011

Optimizing

- When speed really matters, C is the language to use
  - Can be 10-100x faster than Java or Matlab

Optimizing

- When speed really matters, C is the language to use
  - Can be 10-100x faster than Java or Matlab
- But writing something in C doesn't guarantee speed
  - It's up to you to write efficient code

Optimizing

- When speed really matters, C is the language to use
  - Can be 10-100x faster than Java or Matlab
- But writing something in C doesn't guarantee speed
  - It's up to you to write efficient code
- There are two general strategies for optimization
  - Use a better algorithm or different data structures
  - Write code that implements the algorithm more efficiently

Bitwise Operations

```
a = 0x00FF
b = 0xF000

a & b  = 0x0000 // bitwise and
a | b  = 0xF0FF // bitwise or
~a     = 0xFF00 // bitwise not
a ^ b  = 0xF0FF // bitwise xor
a << 4 = 0x0FF0 // left shift by 4 bits
a >> 4 = 0x000F // right shift by 4 bits
```

Right Shift

- Arithmetic right shift: Copy the left most bit
- Logical right shift: Place 0s in the left

Right Shift

- Arithmetic right shift: Copy the left most bit
- Logical right shift: Place 0s in the left

Example

```
int a = 0xF000;

a >> 4 = 0xFF00 // Arithmetic right shift
a >> 4 = 0x0F00 // Logical right shift
```

Right Shift

- Arithmetic right shift: Copy the left most bit
- Logical right shift: Place 0s in the left

Example

```
int a = 0xF000;

a >> 4 = 0xFF00 // Arithmetic right shift
a >> 4 = 0x0F00 // Logical right shift
```

Unfortunately, for `signed` data, there's no standard which one to use, but for almost all machines, arithmetic right shift is used.

Don't Get Confused!

```
x = 0x66, y = 0x93

x & y    = 0x02      x && y = 0x01
x | y    = 0xF7      x || y = 0x01
~x | ~y  = 0xFD    !x || !y = 0x00
x & !y   = 0x00     x && ~y = 0x01
```

Bitwise Operations

- Bitwise operations are fast.

Bitwise Operations

- Bitwise operations are fast.
  - x * 256 $\Leftrightarrow$ x << 8
  - x % 256 $\Leftrightarrow$?

Struct and Union

```
struct node
{
    struct node *left;
    struct node *right;
    double data; // only leaf node has data
};
```

Each node needs 16 bytes, but almost half of them are wasted.

Struct and Union

```
struct node
{
    struct node *left;
    struct node *right;
    double data; // only leaf node has data
};
```

Each node needs 16 bytes, but almost half of them are wasted.

```
union node
{
    struct
    {
        union node *left;
        union node *right;
    } internal;
    double data;
};
```

Struct and Union

But now we have no way to tell which node is leaf.

```
struct node
{
    int is_leaf;
    union
    {
        struct
        {
            union node *left;
            union node *right;
        } internal;
        double data;
    } info;
};
```

Struct and Union

But now we have no way to tell which node is leaf.

```
struct node
{
    int is_leaf;
    union
    {
        struct
        {
            union node *left;
            union node *right;
        } internal;
        double data;
    } info;
};
```

Now each node only needs 12 bytes.

Inefficient Loop

```
void to_lower(char *s)
{
    int i;
    for (i = 0; i < strlen(s); ++i)
    {
        if ('A' <= s[i] && s[i] <= 'Z')
        {
            s[i] -= ('A' - 'a');
        }
    }
}
```

Inefficient Loop

```
void to_lower(char *s)
{
    int i;
    for (i = 0; i < strlen(s); ++i)
    {
        if ('A' <= s[i] && s[i] <= 'Z')
        {
            s[i] -= ('A' - 'a');
        }
    }
}
```

strlen(s) will be computed for each run of the loop. The function seems to be $O(n)$ but it is actually $O(n^2)$.

Code Motion

```
void to_lower(char *s)
{
    int i;
    int len = strlen(s);
    for (i = 0; i < len; ++i)
    {
        if ('A' <= s[i] && s[i] <= 'Z')
        {
            s[i] -= ('A' - 'a');
        }
    }
}
```

Cache

- A subset of data in main memory
- Faster access than main memory
- You can view main memory as a cache to the hard disk

Cache-friendly Code

Comparing these two codes

```
// cache-friendly
int i, j;
for (i = 0; i < 10; ++i)
{
    for (j = 0; j < 20; ++j)
    {
        a[i][j] += 10;
    }
}

// not cache-friendly
int i, j;
for (j = 0; j < 20; ++j)
{
    for (i = 0; i < 10; ++i)
    {
        a[i][j] += 10;
    }
}
```