Manipulação de bits (*bit twiddling*) em C

Prof. Marcelo Veiga Neves
marcelo.neves@pucrs.br

Manipulação de bits

- O que é manipulação de bits (em inglês: "bit twiddling")?
- Da Wikipedia:
  - *"Bit manipulation is the act of algorithmically manipulating bits or other pieces of data shorter than a word. Programming tasks that require bit manipulation include low-level device control, error detection and correction algorithms, data compression, encryption algorithms, and optimization"*

Revisando bases numéricas

- Dados numéricos podem ser representados através de várias bases:
  - Decimal
  - Binária
  - Octal
  - Hexadecimal
- Uma base numérica é apenas uma forma de interpretar os dados, mas isso não altera o valor armazenado
  - Não existe um número "em base 10" que é diferente de um número "em base 2"!

Revisando bases numéricas

- Decimal (base 10)
  - Bom para humanos, ruim para programadores
  - Experimente converter de decimal... para qualquer outra base!
- Binário (base 2)
  - Bom para computadores, ruim para todo o resto
  - Nos permite ver claramente como um computador armazena de fato um determinado valor
  - Mas você trabalharia com valores como 11110000101011100001010011101100?
- Um pouco de terminologia:
  - LSB: least significant bit, i.e. o bit que contribui menos para o valor (menos significativo, bit 0)
  - MSB: most significant bit, i.e. o bit que contribui mais para o valor (mais significativo, bit 31, por exemplo)

Revisando bases numéricas

- Hexadecimal (base 16)
  - Oferece um bom equilíbrio entre a facilidade do decimal e a visão direta do binário
  - É uma forma compacta de representar números de grande magnitude
  - Base 16 é uma potência de 2 (2^4), portanto a conversão entre binário e hex é simples:
    - A partir do bit mais significativo, lê-se 4 bits de cada vez:
    - `bin: 1111 0000 1010 1110 0001 0100 1110 1100`
    - `hex:    F    0    A    E    1    4    E    C`
    - O mesmo número em hex é: F0AE14EC

Usando bases numéricas em C

- Em um programa C, você pode usar exatamente a mesma notação[1]:

```
unsigned int var;

var = 254;          // decimal
var = 0xfe;         // mesmo em hex
var = 0b11111110;   // mesmo em binário
```

- Para exibir a saída em hex, use o modificador *%x* na função *printf* (o 04 apresentado no formato indica para serem utilizados 4 dígitos):

```
printf("%04x\n", var);
```

- Não existe modificador para exibir números em binário!

[1] O suporte para binário está disponível no GCC 4.7+ e alguns outros compiladores (não é padrão)

Relembrando o armazenamento de dados em C

- Podemos usar o operador *sizeof* para obter o tamanho em bytes de qualquer variável ou valor
- Por exemplo, supondo o seguinte programa:

```
int main()
{
    printf("sizeof(int)    = %d\n", sizeof(int));
    printf("sizeof(char)   = %d\n", sizeof(char));
    printf("sizeof(float)  = %d\n", sizeof(float));
    printf("sizeof(double) = %d\n", sizeof(double));
}
```

- Você pode explicar a saída?
- *sizeof* é essencial para manipulação de bits, de forma que se possa saber qual é o bit mais significativo

Operadores *bitwise*

- Operadores *bitwise* são aqueles que manipulam individualmente cada bit dos operandos
- Podem ser usados com valores inteiros: *int*, *char*, *short int*, *long int*, de preferência *unsigned*
- Dois tipos:
  - Operadores lógicos
    - Aplicam operações lógicas usuais, mas bit a bit
    - *and*, *or*, *not*, *xor*
  - Operadores de deslocamento
    - Recebem este nome porque deslocam os bits para a direita ou para a esquerda
    - Espaços "vazios" são preenchidos com zeros ou ums
    - São particularmente úteis quando combinados com operadores lógicos (a seguir)

Operadores *bitwise* - AND

- Em C, o operador AND é representado por &
- Exemplo:

```
int main()
{
    unsigned int x = 0b01001010;
    unsigned int y = 0b10010010;

    printf("%08x\n", x & y);
}
```

- Qual o resultado da operação?

Operadores *bitwise* - OR

- Em C, o operador OR é representado por |
- Exemplo:

```
int main()
{
    unsigned int x = 0b01001010;
    unsigned int y = 0b10010010;

    printf("%08x\n", x | y);
}
```

- Qual o resultado da operação?

Operadores *bitwise* - XOR

- Em C, o operador XOR é representado por $\wedge$
- Exemplo:

```
int main()
{
    unsigned int x = 0b01001010;
    unsigned int y = 0b10010010;

    printf("%08x\n", x ^ y);
}
```

- Qual o resultado da operação?

Operadores *bitwise* - NOT

- Em C, o operador NOT é representado por $\sim$
- Exemplo:

```
int main()
{
    unsigned int x = 0b01001010;

    printf("%08x\n", ~x);
}
```

- Qual o resultado da operação?
- Esse valor muda de acordo com o tamanho do inteiro (*int*, *short*, ...)?
- E como fica, considerando o exemplo abaixo?

```
int main()
{
    unsigned int x = 0b01001010;

    printf("%08x\n", ~x & 0xff);
}
```

Operadores de deslocamento - *shift left*

- Em C, o operador de deslocamento à esquerda é representado por $<<$
- Argumentos: valor a ser deslocado e quantidade de bits para deslocar para a esquerda (o valor original fica inalterado)
- Exemplo: considere x armazenando 23 (0x17):

```
unsigned char x = 0x17;      // 23 em decimal
unsigned char y = x << 1;    // agora y armazenará 46 (0x2e)
```

- Deslocando vários bits de cada vez:

```
y = x << 2;                  // y armazenará 92  (0x5C)
y = x << 3;                  // y armazenará 184 (0xB8)
```

Operadores de deslocamento - *shift right*

- Em C, o operador de deslocamento à direita é representado por >>
- Argumentos: valor a ser deslocado e quantidade de bits para deslocar para a direita (o valor original fica inalterado)
- Exemplo: considere x armazenando 23 (0x17):

```
unsigned char x = 0x17;      // 23 em decimal
unsigned char y = x >> 1;    // agora y armazenará 11 (0x0B)
```

- Deslocando vários bits de cada vez:

```
y = x >> 2;                  // y armazenará 5 (0x5)
y = x >> 3;                  // y armazenará 2 (0x2)
```

- No deslocamento à direita, o que acontece quando o bit mais significativo for 1, e o tipo for *signed* (exemplo: *int*)?

Operadores de deslocamento - Exercícios

- Construa uma função para cada uma das operações abaixo (a função deve retornar o valor alterado):
  1. Zera todos os bits: *unsigned int clear(unsigned int val)*
  2. Seta um único bit: *unsigned int setbit (unsigned int x, int bit)*
  3. Resseta um único bit: *unsigned int clearbit (unsigned int x, int bit)*
  4. Inverte o valor de um único bit: *unsigned int invertBit (unsigned int x, int bit)*
  5. Retorna o estado de um determinado bit: *int testBit (unsigned int x, int bit)* (retorna 0 ou 1)
- Implemente um programa para testar essas funções!

Compos de bit (*bit fields*)

- Denomina-se campo de bit ou palavra parcial (*bit field*, *partial word*) quando extraímos apenas um grupo de bits de uma palavra
- Supondo que desejamos armazenar uma cor através dos seus componentes RGB (*red*, *green*, *blue*), onde cada componente armazena um valor de 0 a 1023 (1024 valores diferentes)
  - A solução mais simples é criar uma struct:

```
struct rgb_s {
    int red;
    int green;
    int blue;
};
```

  - Qual o problema?

Primeira solução

- Quanto espaço precisamos para essa struct?

```
struct rgb_s {
    int red;
    int green;
    int blue;
};
```

- 3 variáveis *int* * sizeof(int) = 3 * 4 = 12 bytes
- Mas isso é apenas UM ponto - e se tiveremos uma matriz de... 1024 x 768 pontos? (1024 * 768 * 12) = 9Mb
- Dependendo da aplicação e do hardware, pode ser demais!

Segunda solução

- Pensando na quantidade de bits necessária...
  - $1024 = 2^{10}$, então precisamos pelo menos 10 bits por componente de cor
  - 3 componentes (R, G, B) x 10 bits = 30 bits (menos que 4 bytes)
    - red: bits 0 a 9
    - green: bits 10 a 19
    - blue: bits 20 a 29
  - Tudo caberia em um único int (e ainda teríamos 2 bits sem uso!)
  - 00BBBBBBBBBBGGGGGGGGGGRRRRRRRRRR

Segunda solução

- C oferece um recurso chamado campos de bit

```
struct rgb_bits_s {
    int red : 10;
    int green : 10;
    int blue: 10;
};
```

- Com isso, é possível agora alterar ou acessar diretamente apenas os bits necessários:

```
struct rgb_bits_s color;
color.red = 511;
```

Segunda solução

- Solução é simples e eficiente, porém nem sempre pode ser utilizada
- Não se sabe como o compilador irá armazenar os 3 campos na memória
  - Não sabemos a ordem de armazenamento
  - Não sabemos o espaço total ocupado pela struct
- Exemplo: aumentando o tamanho dos campos para 12 bits, quanto espaço ocupa a struct?

```
struct rgb_bits_s {
    int red : 12;
    int green : 12;
    int blue: 12;
};
```

- Existe uma solução para os compiladores GCC, LLVM e outros: declarar um atributo chamado *__attribute__((packed))*.

```
struct rgb_bits_s {
...
} __attribute__((packed));
```

Segunda solução

- Você consegue explicar a saída do programa?

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

struct rgb_bits_s {
    int red : 12;
    int green : 12;
    int blue: 12;
};

struct rgb_bits_s2 {
    int red : 12;
    int green : 12;
    int blue: 12;
} __attribute__((packed));

int main()
{
    struct rgb_bits_s num1;
    struct rgb_bits_s2 num2;

    printf("%ld %ld\n", sizeof(num1), sizeof(num2));

    return 0;
}
```

Terceira solução

- A terceira solução é manipular diretamente os campos de bit, usando os operadores bitwise
- Como fazer?
  - Armazenar um valor específico no campo de bits do componente verde?
  - Extrair o valor dos campos de bits dos componentes vermelho ou azul?
  - ...?

Armazenando um valor em um campo de bits

- Exemplo: supondo que desejamos armazenar o valor 500 no campo de bits da componente verde
  - 500 é 0x1F4 (hex)

```
unsigned int colour = ...;       // algum valor
inicial
unsigned int newgreen = 0x1f4;   // novo valor
para armazenar no verde
```

- Primeiro, precisamos "limpar" (zerar) os 10 bits "verdes"
- Que operação lógica pode ser usada para zerar um conjunto de bits?

Zerando um campo de bits

- Um AND bitwise pode ser usado
- Precisamos de uma *máscara*, onde os bits desejados devem ser zeros
  - 00 1111111111 0000000000 1111111111
- Primeiro criamos uma máscara com 10 bits setados:
  - 00000000000000000000 1111111111
- Agora deslocamos para a esquerda por 10 bits:
  - 00000000000 1111111111 0000000000

```
unsigned int mask = 0x3ff << 10;
```

- Finalmente, precisamos inverter a máscara, pois os bits do meio precisam ser ressetados
  - 00 1111111111 0000000000 1111111111
- Para isso, usamos o operador NOT bitwise:

```
mask = ~mask;
```

Armazenando no campo de bits

- E finalmente, para combinar com o valor atual precisamos:
  - Zerar os bits "verdes" com a máscara:

    ```
    colour = colour & mask;
    ```

  - Modificar os bits "verdes" com o *newgreen* deslocado:

    ```
    colour = colour | newgreen;
    ```

- Se você souber exatamente o que está fazendo, é possível fazer tudo de uma vez só!

  ```
  colour = colour & ~(0x3ff << 10) | (newgreen << 10);
  ```

Exercício

- Crie um programa completo que defina uma varíavel *colour* como um *unsigned int*. Em uma repetição:
  - Mostre o valor atual na tela, em binário e hexa
  - Permite ao usuário escolher o componente de cor desejado (*red*, *green*, *blue*)
  - Pergunte o valor novo e armazene no campo de bits correto

Referências

- Este material foi construído pelo Prof. Sérgio Johann Filho com base nos slides de aula do prof. Marcelo Cohen