Operações com bits

Video desta aula



Neste módulo são abordadas técnicas para acessar e manipular bits. Elas são úteis para armazenar grandes quantidades de informação simples em pouco espaço, como vetores de flags ou inteiros com faixas de valores pequenas. O acesso a bits individuais também é útil em operações de baixo nível, envolvendo registradores, pacotes de rede ou portas de entrada/saída.

Bit fields

Ao construir programas que manipulem muitos dados, é importante escolher o tipo mais adequado para cada variável. Por exemplo, podemos definir uma estrutura de dados para armazenar datas/horas da seguinte forma:

```
typedef struct
{
  unsigned int year, month, day, hour, min, sec ;
} date_t ;
```

Usando essa estrutura, cada variável do tipo `date_t` ocupa 24 bytes (6 inteiros de 4 bytes).

No entanto, usar `int` para os campos da estrutura é um desperdício, pois os valores armazenados são pequenos. Podemos usar tipos inteiros menores, como `char` e `short`:

```
typedef struct
{
  unsigned short year ;
  unsigned char  month, day ;
  unsigned char  hour, min, sec ;
} date_t ;
```

Em máquinas com arquitetura de 32 ou 64 bits, cada variável to tipo `date_t` ocupará 8 bytes de memória:

- 5 bytes para os 5 `unsigned char`
- 2 bytes para o `unsigned short`
- 1 byte de *padding* (“enchimento”)

O *padding* é necessário porque as variáveis inteiras deve estar “alinhadas” na memória, ou seja, seu primeiro byte deve estar em um endereço par (*word-size* = 2 bytes), para ser lido corretamente pelo processador. O alinhamento se aplica às variáveis e aos campos internos das estruturas.

Entretanto, há espaço para economizar mais memória, pois os campos da estrutura não precisam usar toda a faixa de valores oferecida por seus tipos:

campo	faixa	bits necessários
year	0 ... 4000[1)]	12
mon	0 ... 11	4
day	0 ... 31	5
hour	0 ... 23	5
min	0 ... 59	6

campo	faixa	bits necessários
sec	0 ... 59	6
TOTAL		38

A linguagem C permite definir variáveis inteiras com um número específico de bits **dentro de structs**, através de uma funcionalidade chamada *bitfield*:

```
typedef struct
{
  unsigned short year:12 ;
  unsigned char  month:4 ;
  unsigned char  day:5 ;
  unsigned char  hour:5 ;
  unsigned char  min:6 ;
  unsigned char  sec:6 ;
} date_t ;
```

Essa nova estrutura demanda 38 bits, ou 4,75 bytes. Devido à necessidade de usar bytes inteiros e do alinhamento, na realidade a estrutura ocupa 6 bytes de memória, ou seja, 2 bytes a menos que no caso anterior.

O código abaixo apresenta os tamanhos das três estruturas:

struct-size.c

```
#include <stdio.h>

typedef struct
{
  unsigned int year, month, day, hour, min, sec ;
} date_t1 ;

typedef struct
{
  unsigned short year ;
  unsigned char  month, day ;
  unsigned char  hour, min, sec ;
} date_t2 ;

typedef struct
{
  unsigned short year:12 ;
  unsigned char  month:4 ;
  unsigned char  day:5 ;
  unsigned char  hour:5 ;
  unsigned char  min:6 ;
  unsigned char  sec:6 ;
} date_t3 ;

int main ()
{
  printf ("date_t1 ocupa %ld bytes\n", sizeof (date_t1)) ;
  printf ("date_t2 ocupa %ld bytes\n", sizeof (date_t2)) ;
  printf ("date_t3 ocupa %ld bytes\n", sizeof (date_t3)) ;
}
```

Bitfields são muito úteis quando é necessário ler ou manipular bits individuais na memória. Uma aplicação frequente é no acesso a estruturas de dados de baixo nível, em drivers de acesso ao hardware. Por exemplo, o *struct* abaixo representa um registrador de 32 bits da interface de um controlador de disco rígido:

```
struct DISK_REGISTER  {
     unsigned ready:1 ;
     unsigned error_occured:1 ;
     unsigned disk_spinning:1 ;
     unsigned write_protect:1 ;
     unsigned head_loaded:1 ;
     unsigned error_code:8 ;
     unsigned track:9 ;
     unsigned sector:5 ;
     unsigned command:5 ;
} ;
```

Outro exemplo muito interessante de uso de *bitfields* pode ser encontrado no arquivo `ieee754.h` do código-fonte do Linux. Esse arquivo define a estrutura em memória dos números de ponto flutuante conforme o padrão IEEE 754.

Formato do `float`:

- sinal (1 bit)
- expoente (8 bits)
- mantissa (23 bits)

```
// extraído (e simplificado) de /usr/include/x86_64-linux-gnu/ieee754.h

union ieee754_float
{
  float f;

  /* This is the IEEE 754 single-precision format.  */
  struct
  {
    #if __BYTE_ORDER == __BIG_ENDIAN
    unsigned int negative:1;
    unsigned int exponent:8;
    unsigned int mantissa:23;
    #endif                          /* Big endian.  */
    #if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int mantissa:23;
    unsigned int exponent:8;
    unsigned int negative:1;
    #endif                          /* Little endian.  */
  } ieee;
};
```

Cuidados a tomar no uso de *bitfields*:

- elementos de *bitfield* não podem ser endereçados por ponteiros, pois podem não começar no início de um byte de memória;
- muitos compiladores limitam o tamanho de um bitfield ao tamanho máximo de um inteiro (16, 32 ou 64 bits);
- vetores de *bitfields* não são permitidos.
- o uso de *bitfields* pode tornar o código não-portável entre máquinas com configuração little/big endian distintas.

Acesso a bits individuais

Para testar um bit específico em uma variável inteira, podemos efetuar um AND bit a bit entre essa variável e uma máscara de bits (*bitmask*), na qual somente o bit a ser testado é verdadeiro.

Por exemplo: para verificar se o 4° bit de `value` está ativo, usa-se a máscara `0x8`:

```
if ( value & 0x8 )        // 0x8 = 0000 1000 em binário
{
   ...
}
```

Para ativar um bit específico, efetua-se um OR bit-a-bit entre a variável e a máscara de bits correspondente.

Por exemplo: para ativar o 3° bit de `value`, usa-se a máscara `0x4`:

```
value = value | 0x4 ;      // 0x4 = 0000 0100 em binário
value |= 0x4 ;           // versão compacta
```

Similarmente, para desativar o 3° bit da variável `value`:

```
value = value & ~ 0x4 ;     // explique!
```

A máscara de bits também pode ser gerada por deslocamentos: