Programação de Computadores

Aula #14

Operações com Bits

O que são *bitfields* em C? Como funcionam as operações *bitwise*?
O que são *bitmaps* em C?

Ciência da Computação – BCC3 – 2024/01
Prof. Vinícius Fülber Garcia

A Memória

Tipos de dados abstraem determinadas porções de memória usadas com um objetivo específico.

Em C, o tipo de dados nativo com menor “tamanho” em memória é o *char*, que contém um byte (ou seja, oito bits).

De forma geral, manipulamos a memória principal ao nível de bytes... porém, e se quiséssemos **manipular os bits** de um determinado byte individualmente?

Sim! Isso é possível!

A Memória

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

Antes de mais nada, vamos analisar um contexto!

O processo de alocação de memória da estrutura ao lado requer um **total de 24 bytes!**

CADA INT, QUATRO BYTES

A Memória

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

Porém, dado que existe um número finito de meses, dias, horas, minutos e segundos, podemos optar por tipos de dados “mais baratos” para armazenar os mesmos.

Agora a estrutura requer 10 bytes!

10 BYTES??????!!!!!

A Memória

Mas mesmo com essa alteração, ainda estamos desperdiçando a memória!

- Meses: necessita de doze símbolos ([1, 12])
- Dias: necessita de trinta e um símbolos ([1, 31])
- Horas: necessita de vinte e quatro símbolos ([0, 23])
- Minutos: necessita de sessenta símbolos ([0, 59])
- Segundos: necessita de sessenta símbolos ([0, 59])

Como podemos “enxugar” ainda mais a estrutura?

BITFIELDS

Bitfields

A ideia por trás de *bitfields* é bastante simples, mas também poderosa:

Utilizar porções específicas de um ou mais bytes para representar dados

Ou seja, podemos “fragmentar” bytes em porções de bits específicas conforme a quantidade de símbolos que precisam ser representáveis.

Porém, é importante ressaltar que **ainda lemos e escrevemos bytes na memória**, não bits.

Bitfields

***Bitfields* são criados no contexto de estruturas em C**; a notação para a criação destes é a seguinte:

```
tipo nome_variavel:quantidade_bits;
```

Vale a pena lembrar: as variáveis presentes em uma estrutura são alocadas em um espaço contínuo de memória! É isso que permite a utilização eficiente de *bitfields*.

Bitfields

Voltando ao exemplo inicial, podemos determinar a quantidade de bits necessária para representar cada dado da estrutura:

- Meses: necessita de doze símbolos ([1, 12]) – **REQUER 4 BITS**
- Dias: necessita de trinta e um símbolos ([1, 31]) – **REQUER 5 BITS**
- Horas: necessita de vinte e quatro símbolos ([0, 23]) – **REQUER 5 BITS**
- Minutos: necessita de sessenta símbolos ([0, 59]) – **REQUER 6 BITS**
- Segundos: necessita de sessenta símbolos ([0, 59]) – **REQUER 6 BITS**

Bitfields

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

Agora, usando *bitfields*, a estrutura necessita de **um total de 8 bytes** para ser instanciada em memória:

Um inteiro de 4 bytes e 3,25 (4, na prática) bytes de variáveis do tipo *char*.

Economia de 2 bytes por instância!

Bitfields

E o que acontece se...

```
unsigned char bit:1 = 2;
```

Se houver memória alocada sequencialmente em posse do processo, **o valor será gravado utilizando dois bits**, tornando o valor da variável “bit” igual a 0 e ocupando um bit que não é originalmente desta variável.

Caso um acesso inválido à memória ocorra, um ***segmentation fault*** ocorrerá.

Bitfields

Outros cuidados importantes são:

- 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

Bitwise

Além de definirmos *bitfields* em estruturas, de maneira mais geral, podemos utilizar operadores para manipular bits específicos de um ou de um conjunto de bytes.

OPERADORES *BITWISE*

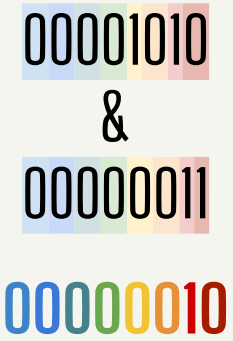
Os operadores *bitwise* consistem em operadores lógicos aplicados bit-a-bit em um determinado dado:

not, and, or, xor, right shift, left shift

Bitwise

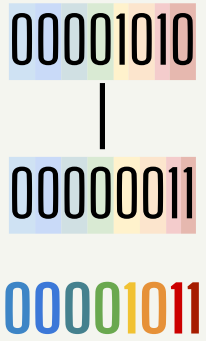
Conjunção (operador &):

```
unsigned char a = 10;
unsigned char b = 3;
printf("%u", (unsigned char) (a | b));
```



Disjunção inclusiva (operador |):

```
unsigned char a = 10;
unsigned char b = 3;
printf("%u", (unsigned char) (a | b));
```



Bitwise

Disjunção exclusiva (operador ^):

```
unsigned char a = 10;
unsigned char b = 3;
printf("%u", (unsigned char) (a ^ b));
```

Negação (operador ~):

```
unsigned char a = 10;
printf("%u", (unsigned char) ~a);
```

Bitwise

Deslocamento dir. (operador >>):

```
unsigned char a = 10;
printf("%u", (unsigned char) a >> 2);
```

(2x) >> 00001010

00000010

Deslocamento esq. (operador <<):

```
unsigned char a = 10;
printf("%u", (unsigned char) a << 2);
```

(2x) << 00001010

00101000

Bitwise

```
unsigned short a = 10;
unsigned char b = 3;
printf("%u", (unsigned char) (a | b));
```

0000000000001010
&
00000011

00000000000001011

Operadores *bitwise* podem ser aplicados entre operandos com tipos de dados diferentes, que requerem quantidades de bytes diferentes.

Nesse caso, os bits de menor valor são alinhados e as operações são realizadas bit-a-bit enquanto houver pares a serem processados.

Bitmaps

Uma utilidade bastante comum que requer a manipulação de bits consiste na criação de mapas de bit (*bitmaps*).

A ideia básica de um bitmap é **demonstrar a presença ou a ausência** de um determinado dado uma vez conhecida a posição no mapa que o representa.

- Se a posição contém o valor 1, dado presente/válido/verdadeiro
- Se a posição contém o valor 0, dado ausente/inválido/falso

Bitmaps

Por exemplo, se temos um vetor com mil posições pré-inicializadas e precisamos mapear quais posições foram modificadas durante a execução do programa.

Podemos utilizar um *bitmask* com 1000 bits ao invés de guardar um short com cada posição modificada!

Uma vez que **1000 bits equivalem a 125 bytes e 1 short requer 2 bytes**, se 63 posições do vetor forem modificadas, neste cenário, o *bitmask* já se torna vantajoso.

Bitmaps

```
#include <limits.h>

// auxiliar macros
#define BITMASK(b)      ( 1 << ((b) %
CHAR_BIT))
#define BITSLOT(b)      ((b)  / CHAR_BIT)
#define BITNSLOTS(nb)   ((nb  + CHAR_BIT  - 1) /
CHAR_BIT)

// bit operations
#define BITSET(a, b)    ((a)[BITSLOT(b)]  |=
BITMASK(b))
#define BITCLEAR(a, b)  ((a)[BITSLOT(b)]  &=
~BITMASK(b))
#define BITTEST(a, b)   ((a)[BITSLOT(b)]  &
BITMASK(b))
#define BITTOGGLE(a, b) ((a)[BITSLOT(b)]  ^=
BITMASK(b))
```

Uma possível implementação de bitmap é dada pelo código de preprocessador ao lado (implementação comp.lang.c).

Vamos analisar cada uma das operações providas

Exercício #14

Considere uma lista de usuários cujos nomes estão armazenados em números inteiros:

- As letras foram armazenadas da primeira à última, nos bits menos significativos para os mais significativos
- Se o nome tem mais de quatro caracteres, inteiros extras são usados
- Bytes não utilizados são indicados por um valor nulo (\0)

Decifre os nomes a seguir:

```
1634624844          1819631952 111          1700949842 7304306
```

Obrigado!

Vinícius Fülber Garcia
inf.ufpr.br/vinicius/
viniciusfulber@ufpr.br