17. Operações de Bit e Bitfields

17.1 Operadores bit-a-bit

- Os operadores de bit operam em lógica binária retornando um escalar. São aplicados aos tipos inteiros (short, unsigned, long, ...)

Operador	Descrição
|	OR binário
^	XOR binário (ou exclusivo)
~	Not – complemento de 1
&	AND binário
<<	Deslocamento à esquerda
>>	Deslocamento à direita

Exemplo1: uso destes operadores.

```
#include <stdio.h>
int main()
{
  int x=1,y=1,z=1;          // x =0001, y = 00001, z = 0001
  printf("~x=%i\n",~x);  //  ~10 = 10 = -2
  x = 1 << 1;               // 01 desloca 1 a esquerda 10 = 2
  y = 10 & 3;               // 1010 "e bit a bit" 0110 = 0010 =2
  z = 10 | 3;               // 1010 "ou bit a bit" 0110 = 1110 = 11
  printf("x=%i, y=%i e z=%i\n", x, y, z);
}
```

resulta em:
~x=-2
x=2, y=2 e z=11

- quando se efetua uma operação binária envolvendo, por exemplo, uma variável inteira sem sinal com 32 *bits*, é necessário considerar os efeitos da operação sobre os 32 *bits* que compõem a variável.

- número de *bits* da variável `int` é dado por `sizeof(int)* 8,` dado que esse operador devolve o tamanho em octetos (*bytes*) da variável ou do tipo de dados que lhe é passado como parâmetro

- pode-se acessar bits específicos de uma variável inteira usando os operadores de bit e uma máscara de bits na qual somente o bit a ser acessado é verdadeiro. Para facilitar, máscaras podem ser definidas como macros.

Exemplo2: verifica se o terceiro bit de um inteiro k é verdadeiro.

```
int main ()
{
   printf ("Digite um numero:");
   scanf ("%d", &k);
   if (k & 0x4)   // 0x4 = 0000 0100 em binário, somente o 3º. bit = 1.
   {
     printf ("\n o terceiro bit de %d e verdadeiro\n\", k);
   }
   else
   {
     printf ("\n o terceiro bit de %d nao e verdadeiro\n\n", k);
     k = k | 0x4; // torna o tercerio bit verdadeiro com um ou bit a bit
     if (k & 0x4)
       printf (" agora o terceiro bit de %d e verdadeiro\n\n", k);
    }
}
```

17.2 Bitfields (campos de bits)

A principal diferença entre o uso de um bitfield (campo de *bits)* e o uso de uma variável inteira é que uma operação binária numa variável inteira envolve todos os *bits* da variável, ao passo que num campo de *bits*, apenas são afetados os *bits* especificados nos campos de bits.

Exemplo 1: sintaxe

```
struct nometipo {
   <tipo1> var1: [nrbytes1] ;
   <tipo2> var2: [nrbytes2] ;
    ...
   <tipon> varn: [nrbytesn] ;
} vbitfield1,vbitfield2;
struct nometipo outravar;
```

exemplo:

```
struct  dadosProduto{
        int cor: 3;
        int eh_seda: 1;
};


struct dadosProdutto prod;
```

- Os tipos (tipo1, ...tipon) devem ser char, int, signed int, unsigned int…

Os inteiros nrbytes1 .. nrbytesn, fornecidos na estrutura, representam o número de bits a ser armazenado para cada campo.

- O bitfield dadosProduto possui um campo eh_seda que tem 1 bit (podendo representar 0 ou 1 somente), e cor que tem tem 3 bits, o que permite representar até 8 cores diferentes (000, 001, 010, 011, 100, 101, 110 e 111, que correspondem aos inteiros de 0 a 7.

- Cabe ao programador saber o que cada bit representa e evitar efeitos indesejados.

- A vantagem é economia de memória pois com o bitfield 4 bits são usados, se campos inteiros fossem utilizados 64 bits seriam necessários.

- O acesso aos campos do bitfield é similar ao da estrutura. Mas não se usa apontadores para os seus membros.

```
prod.ehSeda = 1;    // representa verdadeiro
prod.ehSeda = 0;    //  representa falso
prod.cor = 5     // representa cor verde
prod.cor = 4     // representa cor azul
prod.cor = 10   // pode dar erro.
```

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.

Exemplo 2: com enumerado

```
typedef  enum tipoCor {BRANCO, VERMELH0, AMARELO,
CINZA , AZUL, VERDE, PRETO, ROSA};
struct  dadosProduto{
     int tipoCor: 3;
     int eh_seda: 1;
};
prod.cor = VERDE    // representa cor verde 5
prod.cor = AZUL    // representa cor azul   4
```

Exemplo 3: uma estrutura para manipular bits com um bitfield.

```
typedef struct {
   unsigned char value : 1;
} bit;

int main() {
   bit bits[8];
   int bits_i;
   const int bits_length = (sizeof(bits) / sizeof(bit));
   for (bits_i = 0; bits_i < bits_length; bits_i++)
   {
      if (bits_i % 2)
         bits[bits_i].value = 0;
      else
        bits[bits_i].value = 1;
      printf("%d", bits[bits_i].value);
   }
   printf("\n");
}
```