Expressões lógicas, expressões condicionais, prioridades e operadores, base binária, operadores de bits

Equivalência entre valores lógicos e aritméticos

Quando uma expressão lógica é calculada dentro do programa, é feita a seguinte associação numérica com o valor da expressão:

verdadeiro – 1
falso – 0

O valor zero está associado ao valor lógico `falso`, enquanto que qualquer valor diferente de zero está associado ao valor lógico `verdadeiro`.

Assim, uma comparação como, por exemplo, `a > b`, quando calculada, tem associado a si o valor 1 se `a > b` (verdadeira) e 0 se `a <= b` (falsa).

Da mesma forma uma expressão aritmética como, por exemplo, `a + b`, pode ser considerada falsa ou verdadeira, conforme seu valor seja zero ou diferente de zero.

O programa abaixo mostra algumas variações e exemplos da utilização de expressões aritméticas como lógicas e vice-versa. Verifique o que será impresso abaixo:

```
#include <stdio.h>
int main() {
 char a;
 int b;
 int x = 0, y = 1;
 /* os valores atribuidos a a e b são zero e um, dependendo se a
    expressão é verdadeira ou falsa */
 a = x < y;
 printf ("\n (1) valor de a = %5d",a);
 a = x < y && y > 0;
 printf ("\n (2) valor de a = %5d",a);
 a = x > y;
 printf ("\n (3) valor de a = %5d",a);
 b = x+y < x*y || x == y;
 printf ("\n (4) valor de b = %5d",b);

 /* comparações */
 if (1) printf ("\n (5) sempre verdadeiro");
 if (-5) printf ("\n (6) sempre verdadeiro");
 if (0) ;
 else printf ("\n (7) sempre falso");
 if (1 == 1) printf ("\n (8) sempre verdadeiro");
 if (0 == 0) printf ("\n (9) sempre verdadeiro");

 /* a também pode receber valores aritméticos entre 0 e 255 */
 a = x * y;
 if (a) printf ("\n (10) verdadeiro quando a diferente de zero - a =
%2d", a);
 else   printf ("\n (10) falso quando a igual a zero - a = %2d", a);
```

```
 /* a e b como valores lógicos */
 a = x * y;
 b = x + y;
 if (a && b)
        printf ("\n (11) verdadeiro se a e b nao zero: a = %2d b =
%2d",a,b);
 else printf ("\n (11) falso se a ou b sao zero: a = %2d b = %2d",a,b);

 b = 0;
 /* a repetição será infinita */
 while (1) {
    if (b++ >  4) break; /* força a saída */
    printf("\n (%2d) valor de b = %2d", 11+b, b);
 }

 /* outro exemplo de while */
 a = 1;
 b = 0;
 while (a) {
    printf("\n (%2d) valor de a = %2d valor de b = %2d", 18+b, a, b);
    a = (b++) < 3;
 }
 printf("\n (%2d) valor final de a = %2d e de b = %2d", 18+b, a, b);
}
```

```
 (1) valor de a =     1
 (2) valor de a =     1
 (3) valor de a =     0
 (4) valor de a =     0
 (5) sempre verdadeiro
 (6) sempre verdadeiro
 (7) sempre falso
 (8) sempre verdadeiro
 (9) sempre verdadeiro
 (10) falso quando a igual a zero - a =  0
 (11) falso se a ou b sao zero: a =  0 b =  1
 (12) valor de b =  1
 (13) valor de b =  2
 (14) valor de b =  3
 (15) valor de b =  4
 (16) valor de b =  5
 (18) valor de a =  1 valor de b =  0
 (19) valor de a =  1 valor de b =  1
 (20) valor de a =  1 valor de b =  2
 (21) valor de a =  1 valor de b =  3
 (22) valor final de a =  0 e de b =  4
```

Supondo **`int x,y`**; as seguintes expressões lógicas são equivalentes:

```
(x && y)    (x != 0 && y != 0)    !(x == 0 || y == 0)

(x || y)    (x  != 0 ||  y!= 0)   !(x == 0 && y == 0)
```

Lembram-se daquele problema que dada uma seqüência terminada por zero, calcular a soma dos elementos da seqüência?

```
int soma, x;
:
:
x = 1; soma = 0;
while (x) {
   scanf("%d", &x);
   soma = soma + x;
}
printf ("o valor da soma = %d", soma);
```

Otimização do cálculo de expressões lógicas

Quando se usa uma expressão composta, com os operadores `&&` e `||`, o compilador otimiza o cálculo das mesmas, quando o valor já está definido no meio do cálculo.
Por exemplo, considere a expressão `(a > b && c > d)`. Se `a > b` já é falso, não é necessário verificar se `c > d`, pois o resultado da expressão já será falso. O mesmo ocorre com `(a > b || c > d)`. Se `a > b` é verdadeiro, a expressão é verdadeira independente se `c > d` ou não.

Normalmente os compiladores da linguagem C usam esta regra como padrão, sendo possível alterá-la através de uma opção de compilação para que o cálculo seja completo.

Em alguns casos, há efeitos colaterais provocados pelo cálculo da expressão. O uso de expressões que pressupõe a otimização de cálculo deve ser feito com muito cuidado, para evitar tais efeitos colaterais.

Considere por exemplo a seguinte solução para verificar se **`x`** é igual a algum elemento de um vetor **`a`** de **`n`** elementos. Lembre-se que os elementos são `a[0],a[1],...,a[n-1]`.

```
i = 0;
while (i < n && a[i] != x) i++;
```

Considerando o caso em que x não é igual a nenhum dos elementos, o programa sai do while quando `i = n` e não realiza a comparação `a[i] != x`. Se comparasse estaria errado, pois o elemento `a[n]` está fora dos `n` elementos considerados.

Invertendo agora a expressão acima:

```
i = 0;
while (a[i] != x && i < n) i++;
```

Embora a operação `&&` seja comutativa, a solução acima está errada, pois quando `i = n` a comparação `a[n] != x` será efetuada, pois vem antes da comparação `i < n`.

Expressões condicionais

Considere o comando abaixo:

```
/* atribui a c o maior entre a e b */
if (a > b) c = a;
else c = b;
```

O mesmo resultado pode ser obtido usando-se uma expressão condicional com o operador ternário **?:** da seguinte forma:

```
c = (a > b) ? a : b;
```

Como muitas construções em C, o uso do operador **?:** é compacta demais (pouco clara), porém às vezes facilita.

A forma geral de uma expressão condicional é:
`expressão1? expressão2: expressão3`

A `expressão1` é calculada. Se seu valor for verdadeiro, é calculado o valor de `expressão2`, senão é calculado o valor de `expressão3`. Somente um dos valores `expressão1` ou `expressão2` é calculado.

Considere o seguinte programa abaixo que imprime os números de 0 a 99, dez deles por linha. Após cada número, com exceção do último de cada linha, é impresso "-":

```
#include <stdio.h>
int main() {
   int i;
   for (i = 0; i < 100; i++)
       printf("%5d%1c", i, (i%10 == 9) ? '\n' : '-');
}
```

```
    0-    1-    2-    3-    4-    5-    6-    7-    8-    9
   10-   11-   12-   13-   14-   15-   16-   17-   18-   19
   20-   21-   22-   23-   24-   25-   26-   27-   28-   29
   30-   31-   32-   33-   34-   35-   36-   37-   38-   39
   40-   41-   42-   43-   44-   45-   46-   47-   48-   49
   50-   51-   52-   53-   54-   55-   56-   57-   58-   59
   60-   61-   62-   63-   64-   65-   66-   67-   68-   69
   70-   71-   72-   73-   74-   75-   76-   77-   78-   79
   80-   81-   82-   83-   84-   85-   86-   87-   88-   89
   90-   91-   92-   93-   94-   95-   96-   97-   98-   99
```

Tabela de operadores e suas prioridades de avaliação

A tabela abaixo mostra os operadores do c, em ordem decrescente de prioridade de cálculo:

Prioridade	Operadores	Obs:
1	() [] -> .	esquerda para direita
2	! ~ ++ -- + - (tipo) * & sizeof	direita para esquerda
3	* / %	esquerda para direita
4	+ -	esquerda para direita

5	<< >>	esquerda para direita
6	< <= > >=	esquerda para direita
7	== !=	esquerda para direita
8	&	esquerda para direita
9	^	esquerda para direita
10	|	esquerda para direita
11	&&	esquerda para direita
12	||	esquerda para direita
13	?:	direita para esquerda
14	= += -= *= /= %= <<= >>= &= |= ^=	direita para esquerda
15	,	esquerda para direita

Os unários +, − e * (linha 2) tem maiores prioridades que os binários correspondentes. Os operadores −> e . são usados em `structs` . O operador `sizeof(tipo)` devolve o tamanho em bytes do `tipo`.

Muito cuidado deve ser tomado com a mistura excessiva de operadores numa expressão, devido à definição das prioridades. Na dúvida use parêntesis.

Por exemplo:

`if (x & 5 == 0)` é diferente de `if( (x & 5) == 0)`. Veja a tabela acima.

Além disso, na linguagem C não é especificada (definida) a ordem de avaliação dos operandos de um operador. Assim compiladores diferentes podem levar a resultados diferentes. Portanto não é uma boa prática de programação, usar comandos cujo resultado depende da ordem de avaliação dos seus operandos.

Exemplo:

```
a = exp1 + exp2 (quem é calculado primeiro exp1 ou exp2?)
```

Da mesma forma a ordem de cálculo dos parâmetros de função não está definido no C. Por exemplo, se n é 5:

func(n, n++); pode ser func(5, 6) ou func(6,6)

Números na base 2

Para o próximo assunto é bom uma recordação de como se escreve números na base binária:

Um número na base 2 possui apenas os algarismos 0 e 1, da mesma forma que um número na base 10 possui algarismos de 0 a 9.

O número 1235 na base 10 significa 5 + 3*10 + 2*100 + 1*1000.
O número 10101 na base 2 significa 1 + 0*2 +1*4 + 0*8 + 1*16 = 21.

Alguns exemplos de números na base 2.

base 10	base 2
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Dado um número na base 10, escrevê-lo na base 2. Lembram-se do algoritmo? Basta ir dividindo o número por 2 até acabar e ir pegando os restos na ordem inversa.

Quociente	resto da divisão por 2
25	1
12	0
6	0
3	1
1	1
0	

25 na base 10 = 11001 na base 2

Quociente	resto da divisão por 2
67	1
33	1
16	0
8	0
4	0
2	0
1	1
0	

67 na base 10 = 1000011 na base 2

Este mesmo método pode ser usado para escrever um número em qualquer base. Vejamos por exemplo na base 7.

Quociente	resto da divisão por 7

276	3
39	4
5	5
0	

276 na base 10 = 543 na base 7 (de fato 3 + 4*7 + 5*49 = 276)

O programa abaixo, dado um número n, escreve-o na base 2, porém os dígitos saem na ordem inversa. Veja a saída.

```
#include <stdio.h>
/* dado n>=0 escrever n na base 2
   os dígitos vão sair ao contrário */
int main() {
 int n;

 /* ler o n */
 printf("digite o valor de n:");
 scanf("%d", &n);
 /* imprima o resto e divida n por 2
    o número binário vai estar ao contrário */
 while (n > 0) {
     printf("%1d", n%2);
     n = n / 2;
 }
}
```

```
digite o valor de n:76
0011001
```

O programa abaixo faz a mesma coisa, só que faz com que os dígitos sejam impressos na ordem correta.

```
#include <stdio.h>
/* dado n>=0 escrever n na base 2 */
int main() {
 int n,
     d;
 /* ler o n */
 printf("digite o valor de n:");
 scanf("%d", &n);
 /* descubra a potência de 2 imediatamente superior a n */
 d = 1;
 while (d <= n) d = d * 2;
 /* retrocede uma vez */
 d = d / 2;
 /* ache o quociente pelas potências de 2 sucessivas */
 while (d > 0) {
      printf("%1d", n / d);
      n = n % d;
      d = d / 2;

 }
}
```

```
digite o valor de n:125
1111101
```

Representação interna de números binários

Já vimos acima que variáveis do tipo char ocupam 8 bits, short 16 bits, int 32 bits e long int 64 bits.
Assim, supondo:

```
unsigned char a = 1;
unsigned short b = 2;
unsigned int c = 3;
unsigned long int d = 4;
```

Teriam a seguinte representação interna:

a

00000001

b

00000000	00000010

c

00000000	00000000	00000000	00000011

d

00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000100

Notação complemento de 2 para a representação de números negativos

Os números negativos são representados internamente numa notação denominada "complemento de 2".
Uma primeira idéia para se representar números negativos, seria reservar o bit mais da esquerda para o sinal. Assim em um char (8 bits) poderíamos representar os números –127 a +127 (255 possíveis), isto é, em binário 11111111 e 01111111. O zero possui 2 representações: 00000000 e 10000000.
A notação complemento de 2, permite armazenar um valor a mais (de –128 a +127, ou seja, 256 possíveis), além de outras facilidades que veremos abaixo.

Vejamos um exemplo para uma palavra de 4 bits apenas, para ficar mais fácil. Os valores representáveis serão:

```
-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2 ,3 ,4, 5, 6, 7
```

Valor	Bits
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000

1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Para se obtermos o tal complemento de 2 de um número negativo, basta pegar o padrão de bits correspondente ao positivo, invertermos todos os bits (zeros por uns e vice-versa) e somar um em binário.

Como a tabela acima é cíclica, somas e subtrações são somas e correspondem a deslocar-se na tabela.

Por exemplo: 2 – 4 é 2 + (-4), ou 0010 + 1100 = 1110, ou seja –2 (veja a tabela).

Outro efeito, dessa notação é quando os valores passam dos limites. Por exemplo:

```
7+2 = -7
5+5 = -6
```

Portanto, cuidado quando usar um char com sinal (não unsigned), para fazer operações aritméticas e o resultado eventualmente não couber em um char. Se quiser armazenar valores maiores o iguais a $128=2^7$, use unsigned char.
O mesmo ocorre com short (maiores ou iguais a 2^{15}) e int (valores maiores ou iguais a 2^{31}).

O que será impresso pelo programa abaixo?

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  char a=150;
  unsigned char b=150;
  printf("a=%d;b=%d",a,b);
  system("PAUSE");
  return 0;
}
```

a=-106;b=150

E pelo programa abaixo?

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

int main() {
  char a=120;
```

```
  unsigned b=120;
  char i=0;
  while (i++<20) printf("\na=%d;b=%d",a++,b++);
  system("PAUSE");
  return 0;
}
```

```
a=120;b=120
a=121;b=121
a=122;b=122
a=123;b=123
a=124;b=124
a=125;b=125
a=126;b=126
a=127;b=127
a=-128;b=128
a=-127;b=129
a=-126;b=130
a=-125;b=131
a=-124;b=132
a=-123;b=133
a=-122;b=134
a=-121;b=135
a=-120;b=136
a=-119;b=137
a=-118;b=138
a=-117;b=139
```

Constantes inteiras decimais, octais e hexadecimais em c

Em C, as constantes inteiras podem ser escritas nas bases decimal, octal ou hexadecimal:

Exemplos:

Decimal	Octal (iniciada por 0)	Hexadecimal (iniciada por 0x ou 0X)
0	00	0x0
1	01	0x1
2	02	0x2
3	03	0x3
4	04	0x4
5	05	0x5
7	07	0x7
8	010	0x8
10	012	0xa ou 0XA
20	024	0x14
32	040	0x20
63	077	0x3f ou 0X3F
100	0144	0x64

127	0177	0x7f ou 0X7F
128	0200	0x80
255	0377	0xff ou 0XFF

Exemplos de comandos:

```
int x;

x = 0xff;

for (x = 077; x < 0777; x++) ...

if (x == 0xfa) .....
```

Além disso, as constantes inteiras podem ter sufixo L para torná-las `long`, ou U para torná-las `unsigned` como nos exemplos abaixo:

```
1L constante 1 decimal long
1U constante 1 decimal unsigned
1UL constante 1 decimal unsigned long
0177L constante 0177 octal long
0x2abUL constante 2ab hexadecimal unsigned long
```

Operações com bits

São os seguintes os operadores que manipulam bits:

a) Deslocamento para a esquerda (shift left) <<
b) Deslocamento para a direira (shift right) >>
c) **E** lógico (and) &
d) **OU** lógico (or) |
e) **OU** exclusivo (xor) ^
f) **NÃO** lógico ou o complemento (not) ~

Exemplos supondo:

```
unsigned char a = 1, b = 255, c;
/* observe que em binário a=00000001 e b=11111111 */
c = a << 2; /* desloca a 2 bits para a esquerda, portanto c fica com 4 */
c = b >> 3; /* desloca c 3 bits para a direita, portanto c fica com 31 */
c = a&b; /* c fica com 1 */
c = a|b; /* c fica com 255 */
c = a^b; /* c fica com 254 */
```

Os operadores aplicam-se a todos os tipos numéricos e inteiros (char, short, int, long int, signed ou unsigned) e opera bit a bit.

Veja o programa abaixo e o que será impresso:

```
#include <stdio.h>
int main() {
```

```
   unsigned char a=1, b=255, c, d, i;

   /* observe que
      a = 00000001 ou a = 01h
      b = 11111111 ou b = ffh */

   for (i = 0; i < 9; i++) {
          c = a << i;
          d = b >> i;
          printf("\na deslocado %1d bits para esquerda = %5u", i, c);
          printf("\nb deslocado %1d bits para direita  = %5u", i, d);
   }
}
```

```
a deslocado 0 bits para esquerda =     1
b deslocado 0 bits para direita  =   255
a deslocado 1 bits para esquerda =     2
b deslocado 1 bits para direita  =   127
a deslocado 2 bits para esquerda =     4
b deslocado 2 bits para direita  =    63
a deslocado 3 bits para esquerda =     8
b deslocado 3 bits para direita  =    31
a deslocado 4 bits para esquerda =    16
b deslocado 4 bits para direita  =    15
a deslocado 5 bits para esquerda =    32
b deslocado 5 bits para direita  =     7
a deslocado 6 bits para esquerda =    64
b deslocado 6 bits para direita  =     3
a deslocado 7 bits para esquerda =   128
b deslocado 7 bits para direita  =     1
a deslocado 8 bits para esquerda =     0
b deslocado 8 bits para direita  =     0
```

Idem para o programa abaixo:

```
int main() {
   unsigned char a=1, b=255, i;

   /* observe que
      a = 00000001 ou a = 01h
      b = 11111111 ou b = ffh */

   for (i = 0; i < 9; i++) {
          printf("\na   - decimal = %5u - hexadecimal = %2x"
                 "\nb   - decimal = %5u - hexadecimal = %2x"
                 "\na&b - decimal = %5u - hexadecimal = %2x"
                 "\na|b - decimal = %5u - hexadecimal = %2x"
                 "\na^b - decimal = %5u - hexadecimal = %2x"
                 "\n~a  - decimal = %5u - hexadecimal = %2x\n",
                 a, a,
                 b, b,
                 a&b, a&b,
                 a|b, a|b,
                 a^b, a^b,
                 (unsigned char)(~a), (unsigned char)(~a));
            a = a << 1;
```

```
            b = b >> 1;
    }
  }
```

```
a   - decimal =     1 - hexadecimal =  1
b   - decimal =   255 - hexadecimal = ff
a&b - decimal =     1 - hexadecimal =  1
a|b - decimal =   255 - hexadecimal = ff
a^b - decimal =   254 - hexadecimal = fe
~a  - decimal =   254 - hexadecimal = fe


a   - decimal =     2 - hexadecimal =  2
b   - decimal =   127 - hexadecimal = 7f
a&b - decimal =     2 - hexadecimal =  2
a|b - decimal =   127 - hexadecimal = 7f
a^b - decimal =   125 - hexadecimal = 7d
~a  - decimal =   253 - hexadecimal = fd


a   - decimal =     4 - hexadecimal =  4
b   - decimal =    63 - hexadecimal = 3f
a&b - decimal =     4 - hexadecimal =  4
a|b - decimal =    63 - hexadecimal = 3f
a^b - decimal =    59 - hexadecimal = 3b
~a  - decimal =   251 - hexadecimal = fb


a   - decimal =     8 - hexadecimal =  8
b   - decimal =    31 - hexadecimal = 1f
a&b - decimal =     8 - hexadecimal =  8
a|b - decimal =    31 - hexadecimal = 1f
a^b - decimal =    23 - hexadecimal = 17
~a  - decimal =   247 - hexadecimal = f7


a   - decimal =    16 - hexadecimal = 10
b   - decimal =    15 - hexadecimal =  f
a&b - decimal =     0 - hexadecimal =  0
a|b - decimal =    31 - hexadecimal = 1f
a^b - decimal =    31 - hexadecimal = 1f
~a  - decimal =   239 - hexadecimal = ef


a   - decimal =    32 - hexadecimal = 20
b   - decimal =     7 - hexadecimal =  7
a&b - decimal =     0 - hexadecimal =  0
a|b - decimal =    39 - hexadecimal = 27
a^b - decimal =    39 - hexadecimal = 27
~a  - decimal =   223 - hexadecimal = df

a   - decimal =    64 - hexadecimal = 40
b   - decimal =     3 - hexadecimal =  3
a&b - decimal =     0 - hexadecimal =  0
a|b - decimal =    67 - hexadecimal = 43
a^b - decimal =    67 - hexadecimal = 43
~a  - decimal =   191 - hexadecimal = bf
```

```
a   - decimal =   128 - hexadecimal = 80
b   - decimal =     1 - hexadecimal =  1
a&b - decimal =     0 - hexadecimal =  0
a|b - decimal =   129 - hexadecimal = 81
a^b - decimal =   129 - hexadecimal = 81
~a  - decimal =   127 - hexadecimal = 7f


a   - decimal =     0 - hexadecimal =  0
b   - decimal =     0 - hexadecimal =  0
a&b - decimal =     0 - hexadecimal =  0
a|b - decimal =     0 - hexadecimal =  0
a^b - decimal =     0 - hexadecimal =  0
~a  - decimal =   255 - hexadecimal = ff
```

Veja agora outro programa que escreve os números de 0 a n na base 2. Veja também o que será impresso:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int i, j, k, n;
   printf("entre com n:");
   scanf("%d", &n);
   n = abs(n); /* considere positivo */
   for (i = 1; i <= n; i++) {
      k = i;
      printf("\n%d - ",k);
      /* ache a potencia de 2 imediatamente maior que i */
      for (j = 0; ; j++)
          if (k < (1 << j)) break;
      /* portanto, i < 2**j */
      j--;
      while (j >= 0) {
           /* imprime n dividido por 2**j */
           printf("%1d", k / (1 << j));
           k = k % (1 << j);
           j--;
      }
   }
}
```

```
entre com n:15

1 - 1
2 - 10
3 - 11
4 - 100
5 - 101
6 - 110
7 - 111
8 - 1000
9 - 1001
10 - 1010
```

11 - 1011
12 - 1100
13 - 1101
14 - 1110
15 - 1111

Exercício – refaça os comandos abaixo, usando operações >> e << em vez de * e /

```
x = x * 2;
x = x * 4;
x = x * 32;
x = x / 2;
x = x / 16;
x = x * 6;
```

Nos exemplos abaixo, supor que os bits dentro de uma variável estão numerados da direita para a esquerda, isto é:
tipo char – b7, b6, b5, b4, b3, b2, b1, b0
tipo short – b15, b14, b13, ..., b2, b1, b0
tipo int – b31, b30, b29, ..., b2, b1, b0
É uma maneira conveniente de numerar, pois o número do bit coincide com a potência de 2 da base binária.

```
unsigned char x;
int b;

/* verificar se o bit 5 de x é 1 */
if (x & 32 != 0);
if (x & 040 != 0); /* outra forma */
if (x & 0x20 != 0); /* outra forma */

/* verificar se ambos os bits 0 e 7 de x são 1 */
if (x & 129 != 0);
if (x & 0201 != 0); /* outra forma */
if (x & 0x81 != 0); /* outra forma */

/* fazer com que o bit 5 de x fique 1 */
x = x | 32;
x = x | 040; /* outra forma */
x = x | 0x20; /* outra forma */
```

Algumas funções usando operadores de bits

P70a) Função que dado x entre 0 e 30 devolve 2**x

```
long int exp_2_x(int x) {
   if (x<0 || x > 30) return -1;
   return 1 << x;
}
```

P70b) Função int pegabit (x, n) que devolve o valor do bit n do inteiro x. n pode ser 0 a 31. Supor que o bit mais a direita é o bit 0 e o mais a esquerda o bit 31. Isto é, será devolvido 0 ou 1 ou –1 caso n não esteja entre 0 e 31.

```
int pegabit(int x, int n) {
   if (n < 0 || n > 31) return -1;
   return (x & (1 << n) == 0)) return 0;
   return 1;
}
```

P70c) Função setabit(x, n, b) que faz o bit n do inteiro x ficar 0 ou 1 conforme o valor de b seja 0 ou 1. Retorna 1 se tudo bem ou –1 se n está fora dos limites.

```
int setabit (int *x, int n, int b) {
   if (n < 0 || n > 7) return -1;
   *x = *x | (b << n);
   return 1;
}
```

P70d) Função obtembits(x, p, n) que retorna ajustado a direita, o campo de n bits de x que começa na p-ésima posição. Por exemplo obtembits(x, 4, 3) retorna os 3 bits nas posições de bit 4, 3 e 2 , ajustados à direita.
(solução do livro Kernighan&Ritchie)

```
unsigned obtembits(unsigned x, int p, int n){
  return (x >> (p+1-n) & ~(~0 << n);
}
```

P70e) Função contabits (x) que conta o número de bits 1 do inteiro x. (solução do livro Kernighan&Ritchie).

```
int contabits(unsigned x) {
   int b;
   for (b = 0; x != 0; x >>= 1)
       if (x & 01) b++;
   return b;
}
```