1.10. Побитовые операции

Friday, July 3, 2020 17:34

Как мы уже говорили, язык С предусматривает системное программирование. Помимо указателей, при помощи которых можно легко получать доступ к нужным участкам памяти, может также понадобиться изменять отдельные биты в некоторых регистрах или переменных.

Основы работы с битами

В предыдущем уроке мы упоминали понятие **байта**. Байт состоит из 8 элементов, называемых **битами**. У бита может быть одно из двух значений – `0`, либо `1`. Так, если мы обратимся к какому-либо адресу памяти через указатель, то получим последовательность из 8 бит, например:

```
01100100
```

Самый **правый** бит в байте называется **младшим** битом (*least significant (LSB) or low-order bit*), а самый **левый** бит называется **старшим** битом (*most significant (MSB) or high-order bit*). Если представить себе последовательность бит в виде числа, то счет будет идти с самого правого, младшего бита. Первый бит в десятичной форме равен $2^0 = 1$, бит идущий слева от него $2^1 = 2$, следующий бит $2^2 = 4$.
Так, у нас 3, 6 и 7 биты равны 1. Тогда в десятичном виде это будет равно $2^2 + 2^5 + 2^6 = 100$.

Отрицательные числа представляются немного иначе. Большинство компьютерных систем представляет при помощи **дополнительного кода** (*twos complement*). В таком представлении самый левый бит используется для хранения **знака** числа. Если этот бит равен `1`, значит число отрицательное, а если `0`, то положительное. Остальные биты хранят собственно число. Представленное в дополнительном коде число `-1` будет выглядеть так:

```
11111111
```

Удобный способ преобразовать отрицательное десятичное число в двоичный код – прибавить к нему 1, преобразовать в обычный, беззнаковый двоичный код и затем инвертировать биты.
Для примера преобразуем число –5:

- –5 + 1 = –4
- 4 в двоичном виде будет 00000100
- Инвертируем биты: 11111011.

Чтобы преобразовать двоичное число обратно в десятичное, нужно произвести те же действия в обратном порядке.

Получается, что наибольшее представленное в дополнительном коде положительное число, хранимое в `n` битах, будет равно $2^{n-1} - 1$. Например, в 8-битном числе максимальное положительное значение будет $2^7 - 1 = 127$.

А наименьшим отрицательным числом будет -2^{n-1}, т.е. –128. *Подумайте, почему не –127*.

В большинстве современных систем под целые числа выделяется 32 бита. Наиболее максимальным значением будет *2,147,483,647*, а наименьшим *–2,147,483,648*. В уроке 1.2 (*типы переменных*) мы также знакомились с беззнаковыми переменными (модификатор `unsigned`). Беззнаковые числа бывают только положительными, а за счет того, что больше не нужно хранить информацию о знаке числа, освободившийся бит удваивает допустимый диапазон, точнее, теперь максимальным значением будет 2^{n-1}. То есть, при 32 битах диапазон будет от 0 до 4,294,967,296.

Операции над битами

В таблице ниже приведены операторы для работы с битами.

Symbol	Operation
&	Bitwise AND
|	Bitwise Inclusive-OR
^	Bitwise Exclusive-OR
~	Ones complement
<<	Left shift
>>	Right shift

Битовые операторы

Все операторы в таблице кроме оператора инверсии (~) требуют два операнда. Побитовые операции могут проводиться над любыми **целыми** числами, будь это `int`, `short`, `long long`, `signed/unsigned` – неважно. Над числами с плавающей точкой (`float`) побитовые операции проводить **нельзя**.

Побитовое И (*bitwise AND*)

Когда мы проводим операцию `AND` над двумя значениями, происходит сравнивание индивидуальных бит этих значений. Пусть `b1` и `b2` – соответствующие биты из двух сравниваемых значений, тогда таблица истинности для операции `AND` будет выглядеть следующим образом:

```
b1      b2      b1 & b2
————————————————————————
0       0         0
0       1         0
1       0         0
1       1         1
```

Разберем подробнее на примере. Допустим, у нас есть числа `w1` и `w2` типа `short int` (16 бит), и пусть `w1 = 25, w2 = 77`. Тогда после операции

```
    w3 = w1 & w2;
```

переменной `w3` будет присвоено 9. Почему именно 9, станет понятнее, если представить числа в двоичном виде:

```
w1    0000000000011001      25
w2    0000000001001101   & 77
      ------------------------
w3    0000000000001001       9
```

Мы уже, кстати, пользовались логическим `AND ( && )`, работающий точно так же, но по отношению к логическим выражениям, а не к переменным. Вспомните логику работы этого оператора, возможно так будет проще понять побитовый `AND`.

Побитовый `AND` также можно использовать для создания **масок**, при помощи которых можно, например, обнулять какие-то отдельные биты. Например, выражение

```
    w1 = w1 & 3;
```

обнулит все биты в `w1` кроме первых двух. Первые два бита останутся неизменными.

Как и с арифметическими бинарными операторами, при операциях присваивания можно точно так же использовать побитовые операторы. Так, выражение

```
    word &= 15;
```

выполняет ту же функцию, что и выражение

```
word = word & 15;
```

т.е. обнуляет все биты `word`, кроме первых четырех.

При работе с отдельными битами, обычно используют не десятичные числа, как мы делали выше, а **шестнадцатиричные**, так как обычно приходится работать с 32-битными значениями, а 32 нацело делится на 4 (количество бит в шестнадцатиричной цифре). Напишем программу, иллюстрирующую работу с битами и использующую шестнадцатиричное представление чисел.

```
1   // Program to demonstrate the bitwise AND operator
2   #include <stdio.h>
3
4   int main (void)
5   {
6       unsigned int  word1 = 63u, word2 = 104u, word3 = 136u;
7
8       printf ("word1 & word2 = 0x%x\n", word1 & word2);
9       printf ("word1 & word1 = 0x%x\n", word1 & word1);
10      printf ("word1 & word2 & word3 = 0x%x\n", word1 & word2 & word3);
11      printf ("word1 & 1 = 0x%x\n", word1 & 1);
12
13      return 0;
14  }
```

Программа выдаст:

```
word1 & word2 = 0x28
word1 & word1 = 0x3f
word1 & word2 & word3 = 0x8
word1 & 1 = 0x1
```

Вспомните, что для того, чтоб задать беззнаковое значение, нужно дописать к числу `u` или `U`.

Побитовое ИЛИ (*bitwise OR*)

Таблица истинности для `OR` выглядит следующим образом:

b1	b2	b1 | b2
0	0	0
0	1	1
1	0	1
1	1	1

Побитовый `OR` используют для установки определенных бит в `1`. Так, выражение

```
w1 = w1 | 0x7;
```

установит первым трем битам `w1` значение `1`, вне зависимости от их предыдущего значения. Через оператор присваивания тоже можно:

```
w1 |= 0x7;
```

Иллюстрирующую программу покажем позже.

Побитовое исключающее ИЛИ (*bitwise exclusive OR*)

Таблица истинности для `XOR` выглядит следующим образом:

b1	b2	b1 ^ b2
0	0	0
0	1	1
1	0	1
1	1	0

Одним интересным свойством `XOR`-а является то, что `XOR` одинаковых значений даст 0. Это свойство иногда используется в ассемблерных программах для проверки равенства двух значений, но в языке C так делать нежелательно, потому что это не дает никаких преимуществ в плане скорости и только усложняет чтение кода.

Еще одно полезное свойство `XOR` – с его помощью можно обменять значения двух переменных без использования буферной переменной. Это особенно полезно в микроконтроллерах с малым количеством оперативной памяти. Традиционный код для обмена значений переменны выглядит так:

```
temp = i1;
i1 = i2;
i2 = temp;
```

Альтернативная реализация через `XOR` не нуждается в переменной `temp`:

```
i1 ^= i2;
i2 ^= i1;
i1 ^= i2;
```

Оператор битовой инверсии (*The Ones Complement Operator*)

Оператор `NOT` – это унарный оператор, который инвертирует биты операнда. То есть, все `1` заменяются на `0`, а все `0` на `1`. Таблица истинности выглядит следующим образом:

b1	~b1
0	1
1	0

Оператор битовой инверсии (`~`) не следует путать с арифметическим минусом (`–`) и с оператором логического отрицания (`!`). Так, если у нас есть `int w1 = 0`, то побитовая инверсия превратит все нули в числе в единицы, что даст –1. Арифметический минус даст 0, очевидно. А логическое отрицание вернет `True` (1), т.к. воспримет 0 как `False`.

Одно из полезных применений оператора битовой инверсии – когда точно неизвестно количество бит, выделенных под переменную. Оператор помогает упростить портативность кода, т.е. сделать программу менее зависимой от конкретной системы, на которой она работает. Например, нам нужно обнулить первый бит в переменной. Классический подход будет через `AND` и число со всеми единицами кроме первого бита:

```
w1 &= 0xFFFFFFFE;
```

но такой подход будет работать только на системах, где под `int` выделяется 32 бита. Если же это выражение заменить на

```
w1 &= ~1;
```

то это будет работать на любой системе, вне зависимости занимаемого переменной количества бит, потому что `~1` будет вычислен самой системой или компилятором под эту систему.

Также стоит упомянуть, приоритет различных операторов. Все побитовые операторы имеют более низкий приоритет по сравнению с любыми арифметическими операторами и операторами сравнения. При этом приоритеты самих операторов расставлены следующим образом: `NOT > AND > XOR > OR`. У оператора `NOT` приоритет выше, чем у всех бинарных (т.е. принимающих два значения) побитовых операторов.

Операторы побитового сдвига

Когда мы выполняем битовый сдвиг, биты буквально сдвигаются в указанном направлении. Старшие биты, которые вследствие сдвига выходят за пределы выделенного под переменную места, будут **потеряны**. В освободившемся месте под младшие биты будут записаны **нули**.

Допустим, у нас есть переменная `w1 = 3`. Тогда выражение

```
w1 = w1 << 1;
```

которое также может быть записано как

```
w1 <<= 1;
```

сместит 3 на один бит влево, и теперь `w1` будет равна 6.
Рассмотрим подробнее:

```
w1       ... 0011    3
w1 << 1  ... 0110    6
```

Сдвинем еще на один бит влево и получим 12:

```
w1       ... 0110    6
w1 << 1  ... 1100    12
```

Как видим, каждый битовый сдвиг умножает сдвигаемое значение на 2. Компиляторы С часто используют это свойство для умножения на степень двойки, потому что битовый сдвиг почти всегда быстрее, чем умножение, особенно на микроконтроллерах.

Биты можно двигать и вправо. Это делается через оператор правого сдвига (`>>`). Выдвинутые за границы младшие биты опять же теряются. Если мы сдвигаем биты в беззнаковом числе, то в старших битах в освободившихся местах будут нули. Если это беззнаковое число, то поведение битового сдвига будет зависеть от системы. Если знаковый бит – ноль, то будут добавлены нули в любом случае. Если же там единица, то некоторые системы добавят единицы, а некоторые нули. Первый вариант называется **арифметическим** правым сдвигом, а второй **логическим**. Всегда уточняйте, какой тип сдвига использует система, для которой вы разрабатываете программу.

Также стоит упомянуть, что в языке С точно не определено, что произойдет, если произвести левый или правый сдвиг на большее число бит, чем хранит переменная. Если сдвигать на отрицательное значение, результат также будет неопределенным.

Теперь давайте напишем программу, осуществляющую левый и правый сдвиг. Некоторые компьютеры имеют специальную инструкцию, которая сдвигает биты на `n` влево, если `n > 0` и на `–n` влево, если `n` отрицательное. Реализуем эту инструкцию на С.

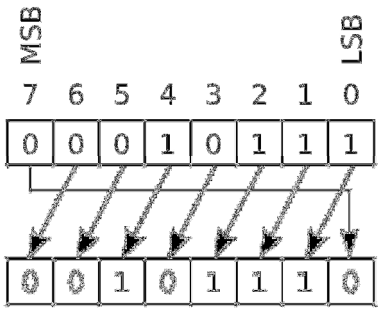
```
1    // Function to shift an unsigned int left if
2    // the count is positive, and right if negative
3
4    #include <stdio.h>
5
6    unsigned int  shift (unsigned int  value, int  n)
7    {
8        if ( n > 0 )     // left shift
```

```
 9            value <<= n;
10        else              // right shift
11            value >>= -n;
12
13        return value;
14    }
15
16    int main (void)
17    {
18        unsigned int  w1 = 0xFFFFu;
19        unsigned int  shift (unsigned int  value, int  n);
20
21        printf ("0x%x\n", shift (w1, 5));
22        printf ("0x%x\n", shift (w1, -6));
23        printf ("0x%x\n", shift (shift (w1, -3), 3));
24
25        return 0;
26    }
```

Третий `printf()` показывает возможность использования вложенных функций, а также потерю бит при сдвиге, т.к. число поменяется после сдвига на 3 влево, а потом на 3 вправо. Такая операция обнулит три младших бита (что все же лучше делать через `w1 &= ~7`).

Циклический сдвиг (*circular shift*)

Циклический сдвиг похож на сдвиг влево/вправо, но здесь уходящий бит не исчезает, а появляется в освободившемся месте на другом конце числа.



Циклический сдвиг влево

Переделаем нашу функцию для битового сдвига под циклический сдвиг. Для этого нужно будет записать ушедшие биты в дополнительную переменную и затем вставить их в освободившееся место. Также нужно будет учитывать количество бит в переменной, – в нашем случае 32 так как мы решили работать с типом `unsigned int`.

```
 1    // Program to illustrate rotation of integers
 2
 3    #include <stdio.h>
 4
 5    int main (void)
 6    {
 7        unsigned int  w1 = 0xabcdef00u, w2 = 0xffff1122u;
 8        unsigned int  rotate (unsigned int  value, int  n);
 9
10        printf ("%x\n", rotate (w1, 8));
11        printf ("%x\n", rotate (w1, -16));
12        printf ("%x\n", rotate (w2, 4));
13        printf ("%x\n", rotate (w2, -2));
14        printf ("%x\n", rotate (w1, 0));
```

```
15          printf ("%x\n", rotate (w1, 44));
16
17          return 0;
18      }
19
20      // Function to rotate an unsigned int left or right
21
22      unsigned int  rotate (unsigned int  value, int  n)
23      {
24          unsigned int  result, bits;
25
26          // scale down the shift count to a defined range
27
28          if  ( n > 0 )
29              n = n % 32;
30          else
31              n = -(-n % 32);
32
33          if  ( n == 0 )
34              result = value;
35          else if ( n > 0 ) {    // left rotate
36              bits = value >> (32 - n);
37              result = value << n  |  bits;
38          }
39          else  {               // right rotate
40              n = -n;
41              bits = value << (32 - n);
42              result = value >> n  |  bits;
43          }
44
45          return result;
46      }
```

Сначала функция проверяет количество сдвигаемых бит `n`. Строки

```
if  ( n > 0 )
   n = n % 32;
else
   n = -(-n % 32);
```

проверяют `n` на положительность. Если `n` положительное, производится сравнение по модулю и результат записывается обратно в `n`. Сравнение по модулю нужно для того, чтобы значение `n` гарантированно оставалось в диапазоне от 0 до 31. Например, `5 % 31` даст 5, т.е. значение сохранится, но `55 % 32` даст 23 и мы все равно не выйдем за допустимые пределы.

Если `n` отрицательное, перед сравнением по модулю его нужно сделать положительным, потому что разные системы по-разному работают с отрицательными аргументами: они могу возвращать как положительное, так и отрицательное значение. Поэтому мы сначала делаем `n` положительным, получаем гарантированно положительный результат и потом этот результат делаем отрицательным.

Если же `n = 0`, функция просто возвращает полученное значение.

Поворот бит влево состоит из трех шагов. Сначала `n` старших бит извлекаются в переменную `bits` при помощи сдвига на 32 – `n`. Здесь 32 – это количество бит в переданной в функцию переменной. Далее выполняется левый сдвиг бит value и производится операция OR между полученным значением и извлеченными битами из `bits`. Операция `OR` идентична операции сложения; так как у нас в освободившемся месте в `value` будут нули, а в `bits` в том же месте будут ушедшие из левой части биты, то в `value` будут записаны только единицы, содержащиеся в `bits`.

При правом сдвиге выполняются те же действия, только сдвиги производятся в обратную сторону.

Битовые поля (*Bit Fields*)

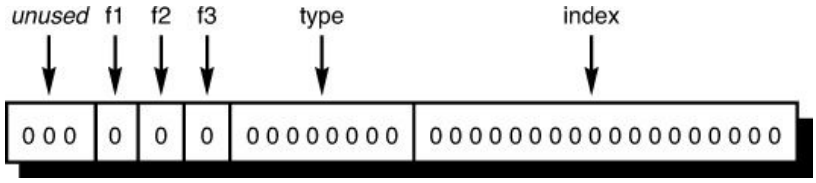
Операции с отдельными битами часто используются, когда одна переменная используется для хранения нескольких элементов данных. Например, когда мы объявляем переменную `bool`, она хранит всего один бит, но в памяти компьютера она будет занимать как минимум 8 бит (вспомните, что указатели на адрес памяти указывают на байты, которые состоят из 8 бит; на отдельные биты они указывать не могут). При недостатке памяти намного рациональнее будет записывать значения таких переменных в виде отдельных бит одной переменных. Например, в `unsigned int` можно хранить 32 таких значения. Доступ к отдельным битам осуществляется при помощи **битового сдвига**.

Допустим, нам нужно упаковать 5 отдельных значений в одну переменную, потому что эти значения небольшие, но их очень много, и нужно экономить память. Допустим, у нас есть 3 переменных `f1, f2, f3` типа `bool`, целочисленная переменная `type`, которая может быть в пределах от 1 до 255, и, наконец, есть знаковая переменная `index`, которая может быть в пределах от 0 до 100000.

Для хранения `f1, f2` и `f3` нужно всего 3 бита. Для `type` нужно 8 бит (потому что $2^8 = 255$). Наконец, для `index` нужно 18 бит. В сумме все 5 переменных занимают 29 бит, поэтому мы можем их хранить в одной 32-битной переменной:

```
unsigned int  packed_data;
```

и затем присвоить значения переменных `f1, f2, f3, type и index` определенным полям в переменной `packed_data`. На рисунке ниже показано, как это будет выглядеть.



Битовые поля в `packed_data`

Обратите внимание, что осталось три неиспользованных бита. Теперь рассмотрим, как записывать и извлекать значения переменных из битовых полей. Например, можно полю `type` присвоить значение `n`, сдвинув его на нужное количество бит и проведя операцию `OR` (помня о том, что n должно быть в диапазоне от 0 до 255):

```
packed_data |= n << 18;
```

Естественно, так можно делать только когда поле `type` обнулено. Чтобы обнулить `type`, нужно через `AND` на `packed_data` наложить маску, из единиц кроме того места, где находится `type`:

```
packed_data &= 0xfc03ffff;
```

Можно еще проще: сдвинуть 8 нулей туда где хранится `type` и провести операцию `AND`:

```
packed_data &= ~(0xff << 18);
```

С таким способом не нужно будет каждый раз рассчитывать значение маски.

Чтобы диапазон `n` гарантированно находился в диапазоне от 0 до 255, перед операцией присвоения значению `type`, можно провести операцию `n = n & 0xFF`. Это обнулит все биты в `n` кроме первых восьми.

Помимо хранения бит в виде обычного числа, их также можно хранить в виде **битовых полей**. При таком подходе используется особый синтаксис структуры, который позволяет определить битовые поля: задать их размеры и название. Создадим аналог переменной `packed_data` через битовые поля:

```
struct  packed_struct
{
    unsigned int  :3;
    unsigned int  f1:1;
    unsigned int  f2:1;
    unsigned int  f3:1;
    unsigned int  type:8;
    unsigned int  index:18;
};
```

Обращение к битовым полям происходит как к обычным элементам структуры. Например, если у нас объявлена переменная

```
struct packed_struct  packed_data;
```

то можно легко установить полю `type` значение `n`:

```
packed_data.type = n;
```

Здесь нам уже не нужно думать о том, что `n` может не влезть в `type` и не нужно заморачиваться с масками и побитовыми операциями.

Битовые поля можно использовать в обычных выражениях и в них они будут автоматически сконвертированы в целочисленные значения. То есть, выражение

```
i = packed_data.index / 5 + 1;
```

вполне будет работать, как и

```
if  ( packed_data.f2 )
        ...
```

Единственное, на что нужно обратить внимание, в битовых полях нет никаких гарантий насчет расположения полей друг относительно друга. Если мы запускаем код на одной и той же системе, проблем с этим быть не должно, но при работе с разными системами, нужно уточнять, как эти системы организуют битовые поля. Например, некоторые системы собирают переменную битового поля справа налево и в таком случае нужно будет объявить поля в обратном порядке:

```
struct  packed_struct
{
    unsigned int  index:9;
    unsigned int  type:4;
    unsigned int  f3:1;
    unsigned int  f2:1;
    unsigned int  f1:1;
    unsigned int  :3;
};
```

Ничто также не мешает внутри структуры помимо битовых полей объявлять и другие типы данных:

```
struct  table_entry
{
    int           count;
    char          c;
    unsigned int  f1:1;
    unsigned int  f2:1;
};
```

Битовые поля могут быть только типа `_Bool` и `int`. Если при объявлении просто на писать `int`, то это не даст гарантий, будет ли это знаковая или беззнаковая переменная, это будет зависеть от компилятора. Лучше всегда явно указывать желаемый тип: `signed int` или `unsigned int`. Также нельзя делать массивы битовых полей и делать указатели на битовые поля.

Битовые поля упаковываются в **блоки** *(units)*, размер которых зависит от компилятора, но обычно это тип `word`, размер которого равен разрядности процессора. В 8-битных процессорах это будет 8 бит, в 64-битных – 64.

Компилятор С **никогда** не меняет порядок битовых полей в попытках оптимизировать занимаемую память.

Также стоит упомянуть **нулевые битовые поля** (*Zero-Length Bitfields*). Как мы уже говорили, в разных системах битовые поля могут быть по-разному расположены друг относительно друга. Нулевое поле используется для того, чтобы два соседних битовых поля гарантированно шли друг за другом:

```
struct foo {
  int    a:3;
```

```
    int    b:2;
    int     :0; // Force alignment to next boundary.
    int    c:4;
    int    d:3;
};
```

Как видно из примера, нулевое поле задается как обычное поле, но с нулевой длиной и **без названия**.

Источники:

1. Stephen Kochan – Programming in C (4th Edition); chapter 11
2. Bit Twiddling Hacks – https://graphics.stanford.edu/~seander/bithacks.html
3. Fast inverse square root – https://en.wikipedia.org/wiki/Fast_inverse_square_root
4. What is zero-width bit field – https://stackoverflow.com/questions/13802728/what-is-zero-width-bit-field
5. How to print binary number via printf – https://stackoverflow.com/questions/6373093/how-to-print-binary-number-via-printf/6373450
6. Is there a bit-equivalent of sizeof() in C? – https://stackoverflow.com/questions/3319717/is-there-a-bit-equivalent-of-sizeof-in-c

Упражнения

2. *Write a program that determines whether your particular computer performs an arithmetic or a logical right shift.*

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

int main (void)
{
     void print_bin(int n);

     int  length, w1 = -3; // Create a negative int

     length = CHAR_BIT * sizeof (int); // Determine int size

     if (w1 >> length)
          printf("System performs arithmetic right shift.\n");
     else
          printf("System performs logical right shift.\n");

     return 0;
}
```

3. *Given that the expression `~0` produces an integer that contains all 1s, write a function called `int_size()` that returns the number of bits contained in an int on your particular machine.*

```
int int_size(void)
{
     int count = 0, i = ~0;
     while (i) {
          i <<= 1;
          count++;
     }
     return count;
}
```

4. *Using the result obtained in exercise 3, modify the `rotate()` function so that it no longer makes any assumptions about the size of an int.*

```
unsigned int  rotate (unsigned int  value, int  n)
{
     int int_size(void);
```

```
4
5          unsigned int  length, result, bits;
6
7          length = int_size();
8
9         // scale down the shift count to a defined range
10
11         if  ( n > 0 )
12              n = n % length;
13         else
14              n = -(-n % length);
15
16         if  ( n == 0 )
17              result = value;
18         else if ( n > 0 ) {    // left rotate
19              bits = value >> (length - n);
20              result = value << n  |  bits;
21         }
22         else  {               // right rotate
23              n = -n;
24              bits = value << (length - n);
25              result = value >> n  |  bits;
26         }
27
28         return result;
29    }
```

5. *Write a function called `bit_test()` that takes two arguments: an `unsigned int` and a bit number `n`. Have the function return `1` bit number `n` if it is on inside the word, and `0` if it is off. Also write a corresponding function called `bit_set()` that takes two arguments: an `unsigned int` and a bit number `n`. Have the function return the result of turning bit n on inside the integer.*

```
1     // Function to check if n-th bit is set in given uint32
2     unsigned int bit_test(unsigned int number, unsigned int n)
3     {
4          n = abs(n) % 32; // so n doesn't exceed the int size
5
6          if ((1 << n) & number)
7               return (1);
8          else
9               return (0);
10    }
11
12    // Function to set the n-th bit in given uint32
13    unsigned int bit_set(unsigned int number, unsigned int n)
14    {
15         number |= 1 << n;
16         return number;
17    }
```

6. *Write a function called `bitpat_search()` that looks for the occurrence of a specified pattern of bits inside an unsigned int. Make certain that the function makes no assumptions about the size of an `int` (see exercise 3 in this chapter).*

```
1     int bitpat_search(unsigned int source, unsigned int pattern, int n)
2     {
3          unsigned int i, length, mask = 0;
4          length = int_size();
5          n = n % length;
6
7          // convert n to number of 1-s, i.e. if n=3, mask = 111
8          if (n > 1)
9               for (i = 0; i < n; i++)
10                   mask = mask * 2 + 1;
```

```
11          else
12               return(-1);
13
14          pattern &= mask; // apply mask to keep only bits of interest in pattern
15
16          for (i = 0; i <= (length - n); i++) {
17               if ((source & (mask << i)) == (pattern << i))
18                    return (i);
19          }
20
21          return (-1);
22     }
```

Сделал наоборот: начинаю с начала и вывожу номер бита, с которого начинается искомый паттерн. Такой способ практичнее. Также в книге ошибка: вместо `int` они приводят пример для `short`.

7. *Write a function called `bitpat_get()` to extract a specified set of bits.*

```
1      int bitpat_get(unsigned int source, int start, int n)
2      {
3           unsigned int i, mask = 0;
4           n = n % int_size();
5           start = start % int_size();
6
7           // convert n to number of 1-s, i.e. if n = 3, mask = 111
8           if (n > 1)
9                for (i = 0; i < n; i++)
10                    mask = mask * 2 + 1;
11          else
12               return(0);
13
14          return ((source & (mask << start)) >> (start));
15     }
```

8. *Write a function called `bitpat_set()` to set a specified set of bits to a particular value.*

```
1      void bitpat_set(unsigned int *source, unsigned int pattern, int start, int n)
2      {
3           unsigned int i, mask = 0;
4           n = n % int_size();
5           start = start % int_size();
6
7           // convert n to number of 1-s, i.e. if n=3, mask = 111
8           if (n > 1)
9                for (i = 0; i < n; i++)
10                    mask = mask * 2 + 1;
11          else
12               return;
13
14          pattern &= mask;
15          *source &= ~(mask << start);
16
17          pattern <<= start;
18          *source |= pattern;
19
20          return;
21     }
```