Оглавление

Краткие сведения из курса двоичной арифметики. Использование нескольких систем исчисления

Двоичная система исчисления.

Мы с Вами давно уже знаем о том, что компьютер может различить только нулевое и единичное состояние бита, и работает компьютер в системе исчисления с основанием 2 или в ***двоичной системе.***

Примечание: Бит получил свое название от английского ***Binary digit (двоичная цифра).***

Пора нам познакомиться с правилами выполнения действий с двоичными числами. Чем мы, собственно и займемся в этом уроке.

Сочетанием двоичных цифр (битов) можно представить любое значение. Значение двоичного числа определяется относительной позицией каждого бита и наличием единичных битов. Ниже показано восьмибитовое число, содержащее все единичные биты:

```
значения:  128  64  32  16  8  4  2  1
биты:    1   1   1   1   1  1  1  1
```

Самая правая цифра имеет весовое значение 1, следующая цифра влево - 2, следующая - 4 и т.д. Общая сумма для восьми единичных битов в данном случае составит 255:

```
(1+2+4+8+16+32+64+128=255)
```

Сложение

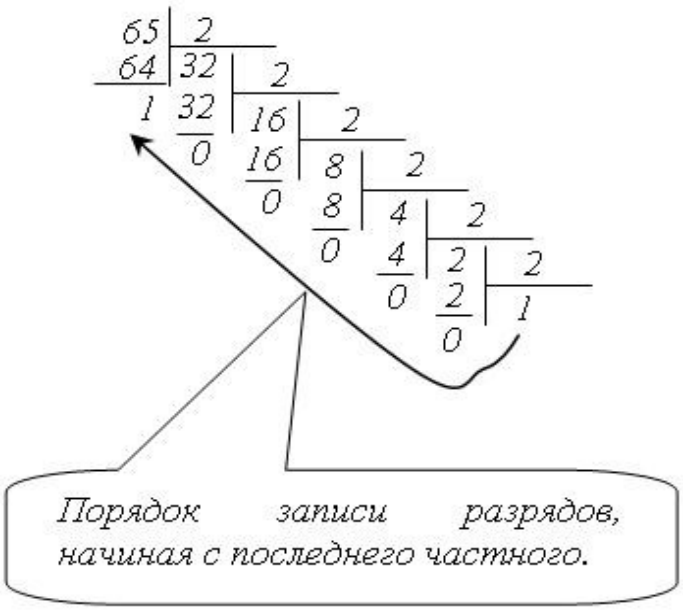
Компьютер выполняет арифметические действия только в двоичном формате. Поэтому, необходимо знать правила сложения в двоичной системе исчисления. Напомним их:

```
0 + 0 = 0
1 + 0 = 1
1 + 1 = 10
```

Давайте рассмотрим использование этих правил на конкретном примере.

Пример: сложить числа 65 и 42, представленные в двоичной системе исчисления.

В десятичной системе исчисления все осуществляется достаточно просто: 65+42=107. Для сложения этих чисел в двоичной системе исчисления нужно сначала перевести их в эту систему, например, как показано на рисунке:



Таким образом, получаем: 6510 = 010000012. Обратите внимание на то, что ведущий ноль в двоичном представлении числа добавлен для дополнения двоичного представления до восьми бит. Аналогично: 4210 = 001010102. Выполним сложение этих чисел:

```
    01000001
+
    00101010
    --------
    01101011
```

Можете перепроверить и убедиться, что 011010112=10710:

```
0*27+1*26+1*25+0*24+1*23+0*22+1*21+1*20 = 64+32+8+2+1 = 107
```

Вычитание

Только что, мы рассмотрели сложение чисел в компьютере. А как же осуществляется вычитание? Для выполнения операции вычитания последнее заменяется сложением, а в качестве второго слагаемого берется ***противоположное*** число. Например, пусть надо выполнить вычитание: 65 - 42. Заменим его сложением: 65 + (-42). Но как получить соответствующее отрицательное двоичное число, спросите вы? Этот вопрос мы сейчас и рассмотрим.

Все представленные выше двоичные числа имеют положительные значения, что обозначается ***нулевым значением самого левого (старшего) разряда***. Отрицательные двоичные числа содержат ***единичный бит в старшем разряде***. Для получения отрицательного двоичного числа можно использовать следующий алгоритм:

1. взять соответствующее положительное число и инвертировать его биты (1 заменить на 0 и наоборот)
2. к полученному числу прибавить 1

Пример использования рассмотренного алгоритма.

Пример 1. Получить двоичное представление числа -65.

Напомним, что **6510 = 010000012**. Инвертируем: **10111110**. К полученному числу прибавим единицу: **10111110+1=10111111**. Убедимся в правильности представления. Сумма +65 и -65 должна составить нуль:

```
    01000001
  +
    10111111
    --------
(1) 00000000
```

Все восемь бит имеют нулевое значение. Пока будем считать, что полученная единица, перенесенная влево, потеряна. Это правило позволяет выполнять вычитание чисел в двоичной системе исчисления:

Вычитание заменяется сложением и в качестве второго слагаемого берется отрицательное число.

Пример 2. Вычесть из 65 число 42.

Двоичное представление для 42 - это 00101010, а для -42 двоичное представление будет следующим - 11010110:

```
     65 = 01000001
+
   (-42) = 11010110
   -----   --------
     23 (1)00010111
```

Пример 3. Какое значение необходимо прибавить к двоичному числу 00000001, чтобы получить число 00000000?

В терминах десятичного исчисления ответом будет число -1. Для двоичного исчисления это число 11111111:

```
    00000001
  +
    11111111
    --------
(1) 00000000
```

В заключение приведем фрагмент уменьшающегося ряда чисел в двоичном представлении:

```
 .    .    .    .
 3     00000011
 2     00000010
 1     00000001
 0     00000000
-1     11111111
-2     11111110
-3     11111101
 .    .    .    .
```

Шестнадцатеричная система исчисления.

Представим, что необходимо просмотреть содержимое некоторых байтов в памяти

Требуется определить содержимое четырех последовательных байтов (двух слов), которые имеют двоичные значения. Для более краткого представления таких длинных чисел был разработан специальный метод представления двоичных данных, по которому каждый байт делится пополам и, каждые полбайта выражаются соответствующим значением. Рассмотрим следующие четыре байта:

```
Двоичное представление:       0101 1001 0011 0101 1011 1001 1100 1110
Десятичное представление:      5    9    3    5   11    9   12   14
```

Так как здесь для представления некоторых чисел требуется две цифры, то расширим систему исчисления так, чтобы

```
10=A, 11=B, 12=C, 13=D, 14=E, 15=F
```

Таким образом, получим более сокращенную форму, которая представляет содержимое вышеуказанных байтов:

```
59    35    B9    CE
```

Такая система исчисления включает "цифры" от 0 до F, и, так как таких цифр 16, она называется ***шестнадцатеричным представлением***. В таблице ниже мы привели двоичные, десятичные и шестнадцатеричные значения чисел от 0 до 15.

Системы исчисления		
двоичная	десятичная	шестнадцатеричная
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Рассмотрим несколько простых примеров шестнадцатеричной арифметики. Следует помнить, что после шестнадцатеричного числа **F** следует шестнадцатеричное 10, что равно десятичному числу 16:

```
6+4=A
5+8=D
F+1=10
F+F=1E
10+10=20
FF+1=100
```

Битовые операции

Только что мы с Вами разобрались с общей теорией, теперь пообщаемся с битовой арифметикой с точки зрения языка С.

В языке С++ существует ряд операций, выполняющихся над разрядами. Они носят название ***битовые операции:***

- **унарная операция:**
  - **инверсия битов (~);**
- **бинарные операции:**
  - **битовое "И" (&);**
  - **битовое "ИЛИ" (|);**
  - **битовое исключающее "ИЛИ" (^);**
  - **сдвиг влево (<<);**
  - **сдвиг вправо (>>);**

Остановимся на данных операциях более подробно.

1. Инверсия битов (поразрядное отрицание, дополнение до единицы) инвертирует биты, т.е. каждый бит со значением 1 получает значение 0 и наоборот.

2.Битовое "И" сравнивает последовательно разряд за разрядом два операнда. Для каждого разряда результат равен 1, тогда и только тогда, когда оба соответствующих разряда операндов равны 1. Так, например,

```
10010011 & 00111101 = 00010001
```

потому что только нулевой и четвертый разряды обоих операндов содержат 1.

3. Битовое "ИЛИ" сравнивает последовательно разряд за разрядом два своих операндов. Для каждого разряда результат равен 1 тогда и только тогда, когда любой из соответствующих разрядов операндов равны 1. Так, например,

```
10010011 | 00111101 = 10111111
```

потому что все разряды (кроме шестого) в одном из двух операндов имеют значение 1.

4. Битовое исключающее "ИЛИ" сравнивает последовательно разряд за разрядом два своих операндов. Для каждого разряда результат равен 1, если один из двух (но не оба) соответствующих разрядов операндов равен 1. Так, например,

```
10010011 ^ 00111101 = 10101110
```

Заметим, что, поскольку нулевой разряд в обоих операндах имеет значение 1, нулевой разряд результата имеет значение 0.

Описанные выше операции часто используются для установки некоторых битов, причем другие биты остаются неизменными. Они удобны для фильтрации или маскирования битов.

5. Сдвиг влево сдвигает разряды левого операнда влево на число позиций, указанное правым операндом. Освобождающиеся позиции заполняются нулями, а разряды, сдвигаемые за левый предел левого операнда, теряются. Поэтому, например,

```
10001010 << 2  =  00101000 ,
(каждый разряд сдвинулся на две позиции влево).
```

Таким образом, ***x<<2*** сдвигает ***x*** влево на 2 позиции, заполняя освобождающиеся позиции нулями (эквивалентно умножению на 4).

Для значений без знака имеем

```
10001010  >> 2  = 00100010 ,
(каждый разряд сдвинулся на две позиции).
```

Эти две операции выполняют сдвиг, а также эффективное умножение и деление на степени числа 2.

Пример.

```
#include <iostream>
using namespace std;
void main (){

      int y=02,x=03,z=01,k;

      k = x|y&z;
        cout<<k<<" ";  /* Операция  1 */

      k = x|y&~z;
        cout<<k<<" ";  /* Операция  2 */

      k = x^y&~z;
        cout<<k<<" ";  /* Операция  3 */

      k = x&y&&z;
          cout<<k<<" ";  /* Операция  4 */

      x = 1;
        y = -1;

      k = !x|x;
        cout<<k<<" ";  /* Операции  5 */

      k = -x|x;
        cout<<k<<" ";    /* Операции  6 */

      k = x^x;
        cout<<k<<" ";    /* Операции  7 */
```

```
        x <<= 3;
          cout<<x<<" ";     /* Операции  8 */

        y <<= 3;
          cout<<y<<" ";     /* Операции  9 */

        y >>= 3;
          cout<<y<<" ";     /* Операции 10 */
}
Результат работы программы:
3 3 1 1 1 -1 0 8 -8 -1
```

Примечание: Здесь мы с Вами знакомимся с еще одной системой исчисления. Целые константы, начинающиеся с цифры 0, являются ***восьмеричными числами***. Восьмеричное представление целых чисел особенно удобно, когда приходится работать с поразрядными операциями, так как восьмеричные числа легко переводятся в двоичные. В этой задаче числа 01,02,03 соответствуют числам 1, 2 и 3, так что появление восьмеричных чисел служит намеком, что программа рассматривает значения x, y и z как последовательности двоичных цифр.

Комментарии к коду.

```
     Операция 1.

   x = 03; y = 02; z = 01;
   k = x|y&z;
Вследствие приоритетов операций: k = (x|(y&z));.
Самое внутреннее выражение вычисляется первым.
   k = (x|(02&01));
   02 & 01 = 00000010 & 00000001 = 00000000 = 0
   k = (x|00);
   03 | 00 = 00000011 | 00000000 = 00000011 = 03
   03
```

```
   Операция 2.
   x = 03; y = 02; z = 01;
   k = x|y&~z;
   k = (x|(y&(~z)));

   ~00000001 =  11111110
   02 & 11111110 =  000000010 & 11111110 = 000000010 = 02
   03 | 02 = 00000011 | 000000010 = 00000011 = 03
   3
```

```
Операции 3.
x = 03; y = 02; z = 01;
k = x^y&~z;

k = (03^02);
1
```

```
Операции 4.
x = 03; y = 02; z = 01;
k = x&y&&z;

k = ((x&y)&&z);
k = ((03&02)&&z);
k = (02&&z);
k = (true&&z);
k = (&&01);
k = (true&&true)
true или 1
```

```
Операции 5.
x = 01;
k = !x|x;

k = ((!x)|x);
k = ((!true)|x);
k = (false|01);
k = (0|01);
1
```

```
Операции 6.
x = 01;
k = -x|x;

k = ((-x)|x);
k = (-01|01);
-01 | 01  = 11111111 | 00000001 =  11111111 = -1
-1
```

```
Операции 7.
x = 01;
k = x^x;

k = (01^01);
```

```
    0
```

```
Операции 8.
   x = 01;
   x <<= 3;

   x = 01<<3;
   01 << 3 = 000000001 << 3 =  00001000 = 8
   x = 8;
   8
```

```
Операции 9.
   y = -01;
   y <<= 3;

   y = -01<<3
   -01 << 3 = 11111111 << 3 =  11111000 =  -8
   y = -8;
   -8
```

```
Операции 10.
   y = -08;
   y >>= 3;

   y = -08>>3;
   -1
```

Примечание: В некоторых случаях вместо -1 может получиться другой результат (8191). Появление этого значения объясняется тем, что на некоторых компьютерах при операции сдвига знак числа может не сохраниться. Не все трансляторы языка С гарантируют, что операция сдвига арифметически корректна, поэтому в любом случае более ясным способом деления на 8 было бы явное деление ***y=y/8***.

Объединения

Структура данных ***объединение*** подобна ***структуре***, однако в каждый момент времени может использоваться (является активным) только ***один из его компонентов***. Шаблон объединения может задаваться записью вида:

```
union
  {
   <имя типа1> <компонента1>;
   <имя типа2> <компонента2>;
        .     .      .
   <имя типаN> <компонентаN>;
  };
```

Поля структуры размещаются в оперативной памяти ***одно за другим в той последовательности, в которой перечислены в описании***. Поля объединений размещаются, начиная с ***одного места в памяти*** и, следовательно, накладываются друг на друга.

Доступ к компонентам объединения осуществляется тем же способом, что и к компонентам структур.

Пример.

```
#include <iostream>
using namespace std;


union Test
{
   int a;
   char b;

}kkk;

void main ()
{
  kkk.a = 65;
  cout<<kkk.a<<" "; // число 65
  cout<<kkk.b;// символ A (соответствующий этому числу)
}
```

В качестве более осмысленного примера объекта типа ***union*** рассмотрим объединение ***geom_fig[1]***:

```
union
  {
   int radius; // Окружность.
   int a[2]; // Прямоугольник.
   int b[3]; // Треугольник.
  } geom_fig;
```

В этом примере обрабатывается только активный компонент, то есть компонент, который последним получает свое значение. Например, после присваивания значения компоненту ***radius*** не имеет смысла обращение к массиву ***b***.

Примечание: Обратите внимание на то, что на одних компьютерах поля битов размещаются слева направо, на других - справа налево. Это значит, что при всей полезности работы с ними, если формат данных, с которыми мы имеем дело, дан нам свыше, то необходимо самым тщательным образом исследовать порядок расположения полей; программы, зависящие от такого рода вещей, не переносимы.

Выводы

Объединения применяются для:

- минимизации используемого объема памяти, если в каждый момент времени только один объект из многих является активным;
- интерпретации основного представления объекта одного типа, как если бы этому объекту был присвоен другой тип.

Таким образом, после задания рассмотренной структуры данных в программе будет находиться переменная, которая на законных основаниях может хранить "в себе" значения нескольких типов.

Битовые поля

В прошлом уроке мы с Вами рассматривали понятие **структуры**. Полями структур могут быть не только переменные, но и другие образования, в частности, ***поля битов***. Хотя правила языка не имеют ограничений на характер этих полей, кроме требования, чтобы они помещались в объеме машинного слова, в типичных применениях поля битов служат для хранения целых данных (чаще типа **unsigned**).

Описание поля битов состоит из описания типа поля, его имени и указанного после двоеточия размера поля в битах, например: **unsigned status: 6;**.

Если имя поля опущено, то создается скрытое поле. Если размер поля битов представлен числом 0, то следующее поле битов начнется с границы машинного слова.

Пример.

```
#include <iostream>
using namespace std;
void Binary(unsigned);
void main()
{
   struct Bits
   {
     unsigned bit1: 3;
     unsigned bit2: 2;
     unsigned bit3: 3;
   } Good;
```

```
    Good.bit1 = 4;
    Good.bit2 = 3;
    Good.bit3 = 6;
    cout<<"Show: "<<Good.bit1<<" ";
    cout<<Good.bit2<<" ";
    cout<<Good.bit3<<"\n\n";
    cout << "Sum: ";
    Binary(Good.bit1 + Good.bit2 + Good.bit3);
}
// Функция выводит на экран двоичное представление числа A.
void Binary (unsigned A)
{
    int i,N;
    if(A>255)
                N = 15;
    else
                N = 7;
    for (i=N; i >= 0; i--)
    {
        cout<<((A>>i)&1);
                if(i==8)
                cout<<" ";
    }
    cout<<"\n\n";
}
```

Домашнее задание

1. Написать программу для хранения в битовом поле информации о конфигурации компьютера. Например: Корпус AT – 0, ATX – 1; Видео на борту – 0, карта – 1 и так далее.

2. Написать программу учета сдачи зачетов при помощи битовых полей. Структура содержит поля: фамилия, группа, зачеты (битовое поле). Предусмотреть вывод списков сдавших все зачеты и должников по группам и в алфавитном порядке.

3. Создать битовое поле для хранения времени (часы, минуты, секунды, миллисекунды). Написать функции для установки и получения времени в(из) битовое(-го) поле(-я).