Лекция 18. Побитовые операции (bitwise operation) Часть 2.

Ревун Артем Леонидович
ст. преп. кафедры вычислительных систем

Курс «Программирование», весенний семестр, 2024
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)

Содержание

- Операции: ~, &, |, ^, >> , <<, &=, |=, ^=, >>=, <<=
- Обзор двоичной, восьмеричной и шестнадцатеричной систем счисления
- Два средства языка С для обработки отдельных битов значения: побитовые операции и битовые поля
- Ключевые слова: _Alignas, _Alignof

Двоичная СС – BIN

Пример: 1234

$$1000 + 200 + 30 + 4$$

$$\Rightarrow 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

Число 1234 *по основанию 10*

Пример: 11011_2

$$10000 + 1000 + 10 + 1$$

$$\Rightarrow 1 \times 2^4 + 1 \times 2^3 + \cancel{0 \times 2^2} + 1 \times 2^1 + 1 \times 2^0$$

Число 11011 по основанию 2 (двоичное число)

Восьмеричная СС – ОСТ

Восьмеричная цифра	Двоичный эквивалент
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Шестнадцатеричная СС - HEX

Десятичное число	Шестнадцатеричная цифра	Двоичный эквивалент
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Побитовые операции (1)

Группы логических операций в языке Си разделены на две группы:

Логические побитовые операции

Сдвиговые побитовые операции

Для большего понимания, все примеры операций будут представлены в двоичной системе счисления, т.к. это наглядно показывает результат выполнения операции

Побитовые операции (2)

Четыре логические побитовые операции работают с целочисленными типами данных (char, short, int, long).

Название ***побитовый***, обусловлено тем, что операция в числе выполняется ***над каждым битом*** числа в независимости от бита находящегося слева или справа.

Побитовые операции (3)

Операция дополнение до единицы или **побитовое отрицание: ~** (тильда)

Унарный оператор ~ преобразует каждую единицу в ноль, а каждый ноль в единицу.

```
                    int val=100; //0110 0100
~(1010 1010)        val= ~(val); //~(0110 0100)
 ( 101 0101)        printf(val)  //-101 (1001 1011)
```

Вспоминаем про преобразование знаковых и беззнаковых чисел.

Побитовые операции (4)

Побитовая операция "И": & (амперсант)

Бинарный оператор & выполняет побитовое сравнение двух операндов. Бит в каждой позиции числа (справа налево) будет равен единице, тогда и только тогда, когда оба соответствующих бита в операндах равны 1.

&(10010011)

(00111101) = 1&1 1&0 0&1 0&1 1&1 0&1 0&0 1&0

1 0 0 0 1 0 0 0

00010001

Побитовые операции (5)

Побитовая операция "И": & (амперсант)

```
int num1 = 100; //01100100
int num2 = 192; //11000000
printf("%d",num1&num2); // 64
```

0	1	1	0	0	1	0	0
1	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0

Побитовые операции (6)

Побитовая операция "ИЛИ": | (терминатор)

Бинарный оператор | выполняет побитовое сравнение двух операндов. Для каждой позиции бит будет равен 1, если хотя бы один из двух битов будет равен 1.

|(10010011)

(00111101) = 1|1 1|0 0|1 0|1 1|1 0|1 0|0 1|0

1 1 1 1 1 1 0 1

10111111

Побитовые операции (7)

Побитовая операция "ИЛИ": | (терминатор)

```
int num1 = 100; //01100100
int num2 = 153; //10011001
printf("%d",num1|num2); // 253
```

0	1	1	0	0	1	0	0
1	0	0	1	1	0	0	1
1	1	1	1	1	1	0	1

Побитовые операции (8)

Побитовая операция исключающее "ИЛИ": ^ (циркумфлекс)

Бинарный оператор ^ выполняет побитовое сравнение двух операндов. Для каждой позиции бит будет равен 1, если один (но не оба) из соответствующих битов в операндах равен 1.

^(10010011)

(00111101) = 1^1 1^0 0^1 0^1 1^1 0^1 0^0 1^0

0 1 1 1 0 1 0 1

10101110

Побитовые операции (9)

Побитовая операция исключающее "ИЛИ": ^ (циркумфлекс)

```
int num1 = 116; //01110100
int num2 = 249; //11111001
printf("%d",num1^num2); // 141
```

0	1	1	1	0	1	0	0
1	1	1	1	1	0	0	1
1	0	0	0	1	1	0	1

Побитовые операции (10)

Для всех бинарных побитовых операций в языке Си доступна сокращенная запись с оператором присваивания.

```
val &= 0377;
val |= 0377;
val ^= 0377;
```

Маски на побитовых операциях (1)

Под **маской** понимается некоторая последовательность битов, в которой часть битов включены (1), а часть выключены(0). **Маска** – это значение заданное программистом с целью *скрыть* или *проявить* биты какого-либо значения.

Символически обозначим переменную хранящую значение маски **MASK**.

Маски на побитовых операциях (2)

Побитовая операция "И« (маска сокрытия)

Значение	1	0	0	1	1	1	0	1
Маска (сокрытие)	1	0	0	0	0	0	0	0
Результат	1	-	-	-	-	-	-	-

157, 0235, 0x9D

```
flags &= MASK;
```

128, 0200, 0x80

128, 0200, 0x80

Маски на побитовых операциях (3)

Побитовая операция "ИЛИ“ (Маска включения)

Значение	0	0	1	0	0	1	1	0
Маска (включение)	1	0	0	0	1	0	0	0
Результат	1	0	1	0	1	1	1	0

38, 046, 0x26

```
flags |= MASK;
```

136, 0210, 0x88

174, 0256, 0xAE

Маски на побитовых операциях (4)

Побитовые операции дополнение до единицы и"ИЛИ": (маска очистки битов)

Значение	0	1	1	1	0	1	1	1
Маска (очистка)	0	1	0	1	0	1	0	1
~Маска (очистка)	1	0	1	0	1	0	1	0
Результат	0	0	1	0	0	0	1	0

119, 0167, 0x77

85, 0125, 0x55

34, 042, 0x22

```
flags &= ~MASK;
```

Маски на побитовых операциях (5)

Побитовая операция исключающее "ИЛИ"(маска переключения битов)

Значение	0	1	1	1	0	1	1	1
Маска (переключение)	0	1	1	1	0	0	0	1
Результат	0	0	0	0	0	1	1	0

119, 0167, 0x77

113, 0161, 0x71

6, 06, 0x6

```
flags ^= MASK;
```

Маски на побитовых операциях (6)

Побитовая операция "И" (маска проверки бита)

Значение	0	1	1	1	0	1	1	1
Маска (проверка)	0	1	1	1	0	0	0	1
Результат	0	1	1	1	0	0	0	1

119, 0167, 0x77

`((flags&MASK)==MASK)` 113, 0161, 0x71

True 113, 0161, 0x71

Маски на побитовых операциях (7)

Побитовая операция "И" (маска проверки бита)

Значение	0	1	0	0	0	1	1	1
Маска (проверка)	0	1	1	1	0	0	0	1
Результат	0	1	0	0	0	0	0	1

71, 0107, 0x47

```
((flags&MASK)==MASK)
```

113, 0161, 0x71

False

65, 0101, 0x41

Побитовые операции сдвига (1)

Побитовая бинарная операция сдвиг влево "<<"

До сдвига	0	1	0	0	0	1	1	1
Результат	0	0	0	1	1	1	0	0

```
flags = flags << 2
```

71, 0107, 0x47

28, 034, 0x1C

Побитовые операции сдвига (2)

Побитовая бинарная операция сдвиг влево ">>"

До сдвига	0	1	0	0	0	1	1	1
Результат	0	0	0	1	0	0	0	1

```
flags = flags >> 2
```

71, 0107, 0x47

17, 021, 0x11

Побитовые операции сдвига (3)

Побитовая бинарная операция сдвиг влево ">>"

В зависимости от системы, операция может учитывать ***знаковость*** числа. `flags >>= 2`

Система воспринимает все числа как без знака

До сдвига	1	1	0	1	0	1	0	0
Результат	0	0	1	1	0	1	0	1

Система воспринимает все числа как знаковые

До сдвига	1	1	0	1	0	1	0	0
Результат	1	1	1	1	0	1	0	1

Побитовые операции сдвига (4)

Побитовые операции сдвига могут быть эффективным средством выполнения умножения и деления на степени 2:

number << *n* Умножает number на 2 в степени n

number >> *n* Делит number на 2 в степени n, (n > 0)

`number = 28`	0	0	0	1	1	1	0	0
`number << 2`	0	1	1	1	0	0	0	0 = 112
`number >> 2`	0	0	0	0	0	1	1	1 = 7

Побитовые операции сдвига (4)

Побитовые операции сдвига могут быть использованы для извлечения *части* значений из более крупных значений или структур.

```
#define BYTE_MASK 0xff
unsigned long color = 0x002a162f;
unsigned char blue , green, red;
red = color & BYTE_MASK;
green = (color >> 8) & BYTE_MASK;
blue = (color >> 16) & BYTE_MASK;
```

Побитовые операции сдвига (5)

```
#define BYTE_MASK 0xff
```

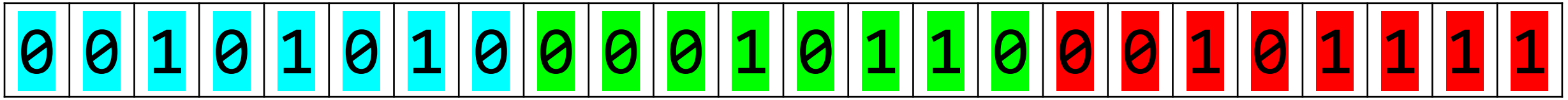
1	1	1	1	1	1	1	1

```
unsigned long color = 0x002a162f;
```

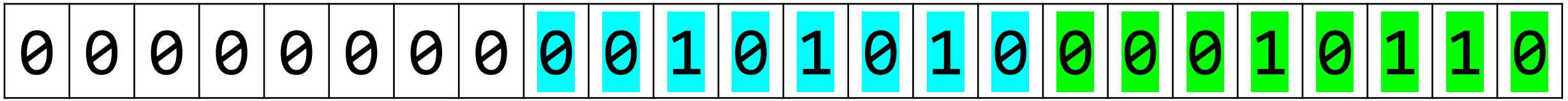
0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0	0	0	1	0	1	1	1	1

Побитовые операции сдвига (6)

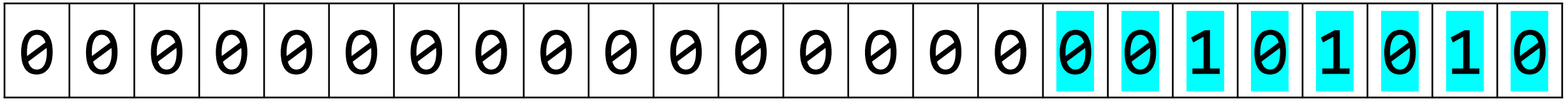
```
unsigned long color = 0x002a162f;
```




```
red = color & BYTE_MASK; //00101111
```




```
green=(color >> 8)&BYTE_MASK; //00010110
```




```
blue=(color >> 16)&BYTE_MASK; //00101010
```

Побитовые операции сдвига (7)

Так же как и для логических побитовых операций для операций сдвига доступны сокращенные формы записи скомбинированные с оператором присваивания.

```
color >>= 8;
color <<= 8;
```

Средств выравнивания - C11 (1)

Средства **выравнивания** по своей природе больше ориентированы на манипулирование ***байтами***, чем ***битами***, но они также отражают возможность языка Си при работе с аппаратной составляющей. В этом контексте ***выравнивание*** относится к тому, как объекты располагаются в памяти.

Пример, для максимальной эффективности система может требовать, чтобы значение типа **double** хранилось в памяти по адресу, кратному 4, но разрешать значению типа **char** храниться по любому адресу.

Средств выравнивания - C11 (2)

Операция `_Alignof` выдает требования к выравниванию указанного типа. Для ее использования необходимо после ключевого слова `_Alignof` поместить имя типа в круглых скобках.

```
size_t f_align = _Alignof(float);
```

Адрес: 0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007 0x1008…
Ячейки: [float] [char] [padding] [float] ...

Средств выравнивания - C11 (3)

```
struct Foo {
    char a;      // 1 байт
    float b;     // 4 байта
    short c;     // 2 байта
};
_Alignof(char); //Выравнивание char: 1 байт
_Alignof(float); //Выравнивание float: 4 байт
_Alignof(short); //Выравнивание short: 2 байт
sizeof(struct Example);
//Размер структуры: 12 байт
```

Средств выравнивания - C11 (3)

Адрес	0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007	0x1008	0x1009	0x100A
Ячейки	a	padding	padding	padding	b				c		padding

Таким образом все переменные будут выравнены в памяти до одинаковой длинны, (в данном случае до типа `float`, все пустоты будут заполнены отступами. И суммарно вся структура будет занимать 12 байт. Как-будто хранится три значения типа `float`.

Зачем?

Продолжение следует…
Спасибо за внимание

Ревун Артем Леонидович
ст. преп. кафедры вычислительных систем

Курс «Программирование», весенний семестр, 2024
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)