

Основы алгоритмизации и программирования.
Битовые операции
Лекция 10

Привезенцев Д.Г.

Муромский институт Владимирского государственного университета
Очная форма обучения

25 ноября 2021 г.

Что такое битовые операции

Битовые операции – это тестирование, установка или сдвиг битов в байте или слове, которые соответствуют стандартным типам языка C **char** и **int**.

Битовые операторы не могут использоваться с **float**, **double**, **long double**, **void** и другими сложными типами.

Оператор	Действие	Обозначение
&	И	AND
	ИЛИ	OR
~	Дополнение (НЕ)	NOT
^	Исключающее ИЛИ	XOR
>>	Сдвиг вправо	-
<<	Сдвиг влево	-

Таблицы истинности

Битовые операции изучаются в дискретной математике, а также лежат в основе цифровой техники, так как на них основана логика работы логических вентилей — базовых элементов цифровых схем. В дискретной математике, как и в цифровой технике, для описания их работы используются таблицы истинности.

x	y	NOT x	x AND y	x OR Y	x XOR y
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Побитовое И (&)

Побитовое И (оператор &) берёт два числа и логически умножает соответствующие биты. Например, если логически умножить 3 на 8, то получим 0

$$3_{10} = 00000011_2$$

$$8_{10} = 00001000_2$$

Тогда их побитовое произведение - операция И - будет представлена следующим образом

0	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0
↓	↓	↓	↓	↓	↓	↓	↓
0	0	0	0	0	0	0	0

Побитовое И (&)

Соответственно, побитовое произведение чисел 31 и 17 даст 17, так как

$$31_{10} = 00011111_2$$

$$17_{10} = 00010001_2$$

0	0	0	1	1	1	1	1
0	0	0	1	0	0	0	1
↓	↓	↓	↓	↓	↓	↓	↓
0	0	0	1	0	0	0	1

Побитовое И (&)

Побитовое произведение чисел 35 и 15 равно 3.

$$35_{10} = 00100011_2$$

$$15_{10} = 00001111_2$$

0	0	1	0	0	0	1	1
0	0	0	0	1	1	1	1
↓	↓	↓	↓	↓	↓	↓	↓
0	0	0	0	0	0	1	1

Побитовое ИЛИ (|)

Побитовое ИЛИ (оператор |) берёт два числа и логически складывает соответствующие биты без переноса.

Например, если логически сложить 15 и 11, то получим 15

$$15_{10} = 00001111_2$$

$$11_{10} = 00001111_2$$

0	0	0	0	1	1	1	1
0	0	0	0	1	0	1	1
↓	↓	↓	↓	↓	↓	↓	↓
0	0	0	0	1	1	1	1

Побитовое ИЛИ (|)

Побитовое ИЛИ для чисел 33 и 11 вернёт 4, так как

$$33_{10} = 00100001_2$$

$$11_{10} = 00001111_2$$

0	0	1	0	0	0	0	1
0	0	0	0	1	0	1	1
↓	↓	↓	↓	↓	↓	↓	↓
0	0	1	0	1	0	1	1

Побитовое отрицание (оператор \sim)

Побитовое отрицание (оператор \sim) работает не для отдельного бита, а для всего числа целиком. Оператор инверсии меняет ложь на истину, а истину на ложь, для каждого бита.

Так, например, побитовое отрицание числа 65 - это число -66.

$$65_{10} = 01000001_2$$

0	1	0	0	0	0	0	1
↓	↓	↓	↓	↓	↓	↓	↓
1	0	1	1	1	1	1	0

Да, но: $10111110_2 = 190_{10}$! Почему ответ -66?

Побитовое отрицание (оператор \sim)

Ответ: однобайтовое знаковое число в Си (тип `char`) представляется обратным дополнительным кодом.

Поэтому если старший бит равен 1, то число отрицательное. Чтобы получить его модуль нужно сделать отрицание и прибавить единицу:

Побитовое отрицание (оператор \sim)

Ответ: однобайтовое знаковое число в Си (тип `char`) представляется обратным дополнительным кодом.

Поэтому если старший бит равен 1, то число отрицательное. Чтобы получить его модуль нужно сделать отрицание и прибавить единицу:

$$\textcircled{1} \sim 10111110_2 = 1000001_2$$

Побитовое отрицание (оператор \sim)

Ответ: однобайтовое знаковое число в Си (тип `char`) представляется обратным дополнительным кодом.

Поэтому если старший бит равен 1, то число отрицательное. Чтобы получить его модуль нужно сделать отрицание и прибавить единицу:

$$\textcircled{1} \sim 10111110_2 = 1000001_2$$

$$\textcircled{2} 1000001_2 + 1_2 = 1000010_2$$

Побитовое отрицание (оператор \sim)

Ответ: однобайтовое знаковое число в Си (тип `char`) представляется обратным дополнительным кодом.

Поэтому если старший бит равен 1, то число отрицательное. Чтобы получить его модуль нужно сделать отрицание и прибавить единицу:

$$\textcircled{1} \sim 10111110_2 = 1000001_2$$

$$\textcircled{2} 1000001_2 + 1_2 = 1000010_2$$

$$\textcircled{3} 1000010_2 = 66_{10}$$

Побитовое отрицание (оператор ~)

Листинг 1: Алгоритм получения отрицательного числа

```
#include <stdio.h>
int main()
{
    printf("Введите число (0..127): ");
    char a;
    scanf("%d", &a);

    char b = ~a + 1;

    printf("Его отрицательное число: %d", b);
    getchar();
    getchar();
    return 0;
}
```

Побитовое исключающее ИЛИ (\wedge)

Исключающее ИЛИ (оператор \wedge) применяет побитово операцию XOR.

Например, для чисел 12 и 85, то получим 89

$$12_{10} = 00001100_2$$

$$85_{10} = 00001111_2$$

0	0	0	0	1	1	0	0
0	1	0	1	0	1	0	1
↓	↓	↓	↓	↓	↓	↓	↓
0	1	0	1	1	0	0	1

Логические и побитовые операторы

Иногда логические операторы `&&` и `||` путают с операторами `&` и `|`. Такие ошибки могут существовать в коде достаточно долго, потому что такой код в ряде случаев будет работать. Например, для чисел 1 и 0. Но так как в Си истиной является любое ненулевое значение, то побитовое умножение чисел 3 и 4 вернёт 0, хотя логическое умножение должно вернуть истину.

Листинг 2: Сравнение логических операторов с битовыми

```
#include <stdio.h>
int main()
{
    int a = 3;
    int b = 4;
    printf("a & b = %d\n", a & b);    //выведет 0
    printf("a && b = %d\n", a && b); //выведет не 0 (конкретнее, 1)
    getchar();
    return 0;
}
```


Операции побитового сдвига

Операций сдвига две – битовый сдвиг влево (оператор \ll) и битовый сдвиг вправо (оператор \gg). Битовый сдвиг вправо сдвигает биты числа вправо, дописывая слева нули. Битовый сдвиг влево делает противоположное: сдвигает биты влево, дописывая справа нули. Вышедшие за пределы числа биты отбрасываются.

Например, сдвиг числа 5 влево на 2 позиции

$$00000101_2 \ll 2 == 00010100_2 \quad (5_{10} \ll 2 = 20_{10})$$

Сдвиг числа 19 вправо на 3 позиции

$$00010011_2 \gg 3 == 00000010_2 \quad (19_{10} \gg 3 = 2_{10})$$

Операции побитового сдвига

Числа в двоичном виде представляются слева направо, от более значащего бита к менее значащему. Побитовый сдвиг принимает два операнда – число, над которым необходимо произвести сдвиг, и число бит, на которое необходимо произвести сдвиг.

Листинг 3: Побитовый дwig как умножение и деление на 2

```
#include <stdio.h>
int main()
{
    int a = 12;
    printf("%d << 1 == %d\n", a, a << 1); // 24
    printf("%d << 2 == %d\n", a, a << 2); // 48
    printf("%d >> 1 == %d\n", a, a >> 1); // 6
    printf("%d >> 2 == %d\n", a, a >> 2); // 3
    getchar();
    return 0;
}
```

Примеры

Функция, которые позволяет изменять определённый бит числа.

Для того, чтобы узнать, какой бит (1 или 0) стоит на позиции n , воспользуемся логическим умножением.

Примеры

Функция, которые позволяет изменять определённый бит числа.

Для того, чтобы узнать, какой бит (1 или 0) стоит на позиции n , воспользуемся логическим умножением.

Пусть имеется число $9_{10} = 00001001_2$

Нужно узнать, выставлен ли бит на позиции 3 (начиная с нуля). Для этого умножим его на число, у которого все биты равны нулю, кроме третьего:

$$00001001_2 \& 00001000_2 = 00001000_2$$

Примеры

Функция, которая позволяет изменять определённый бит числа.

Для того, чтобы узнать, какой бит (1 или 0) стоит на позиции n , воспользуемся логическим умножением.

Пусть имеется число $9_{10} = 00001001_2$

Нужно узнать, выставлен ли бит на позиции 3 (начиная с нуля). Для этого умножим его на число, у которого все биты равны нулю, кроме третьего:

$$00001001_2 \& 00001000_2 = 00001000_2$$

Теперь узнаем значение бита в позиции 6

$$00001001_2 \& 01000000_2 = 00000000_2$$

Примеры

Функция, которая позволяет изменять определённый бит числа.

Для того, чтобы узнать, какой бит (1 или 0) стоит на позиции n , воспользуемся логическим умножением.

Пусть имеется число $9_{10} = 00001001_2$

Нужно узнать, выставлен ли бит на позиции 3 (начиная с нуля). Для этого умножим его на число, у которого все биты равны нулю, кроме третьего:

$$00001001_2 \& 00001000_2 = 00001000_2$$

Теперь узнаем значение бита в позиции 6

$$00001001_2 \& 01000000_2 = 00000000_2$$

Таким образом, если мы получаем ответ, равный нулю, то на искомой позиции находится ноль, иначе единица. Чтобы получить число, состоящее из нулей с одним битом на нужной позиции, сдвинем 1 на нужное число бит влево.

Листинг 4: Функция получения двоичного представления числа

```
#include <stdio.h>
#include <limits.h>
int checkbit(const int value, const int position) {
    if ((value & (1 << position)) == 0) {
        return 0;
    } else {
        return 1;
    }
}
int main() {
    int a;
    printf("Введите число a: ");
    scanf("%d", &a);

    int len = sizeof(int) * CHAR_BIT;
    printf("Двоичное представление этого числа:\n");

    for (int i = len - 1; i >= 0; i--) {
        printf("%d", checkbit(a, i));
    }

    getchar(); getchar();
    return 0;
}
```

Листинг 5: Функция получения двоичного представления числа (упрощенная)

```
#include <stdio.h>
#include <limits.h>
int checkbit(const int value, const int position) {
    return ((value & (1 << position)) != 0);
}

int main() {
    int a;
    printf("Введите число a: ");
    scanf("%d", &a);

    int len = sizeof(int) * CHAR_BIT;
    printf("Двоичное представление этого числа:\n");

    for (int i = len - 1; i >= 0; i--) {
        printf("%d", checkbit(a, i));
    }

    getchar(); getchar();
    return 0;
}
```


Примеры

Функция, которая выставляет бит на n -й позиции в единицу

Известно, что логическое сложение любого бита с 1 будет равно 1.

Так что для установки n -го бита нужно логически сложить число с таким, у которого все биты, кроме нужного, равны нулю.

$$0001011_2 | 0000100_2 = 0001111_2$$

Чтобы получить такую маску, сначала создадим число с нулями и одной единицей.

Примеры

Функция, которая выставляет бит на n-й позиции в единицу

Известно, что логическое сложение любого бита с 1 будет равно 1.

Так что для установки n-го бита нужно логически сложить число с таким, у которого все биты, кроме нужного, равны нулю.

$$0001011_2 | 0000100_2 = 0001111_2$$

Чтобы получить такую маску, сначала создадим число с нулями и одной единицей.

```
int setbit(const int value, const int position) {  
    return (value | (1 << position));  
}
```

Примеры

Функция, которая устанавливает бит на n -й позиции в ноль.

Для этого нужно, чтобы все биты числа, кроме n -го, не изменились. Умножим число на такое, у которого все биты равны единице, кроме бита под номером n .
Например

$$0001011_2 \& 1110111_2 = 0000011_2$$

Чтобы получить такую маску, сначала создадим число с нулями и одной единицей, а потом инвертируем его.

Примеры

Функция, которая устанавливает бит на n -й позиции в ноль.

Для этого нужно, чтобы все биты числа, кроме n -го, не изменились. Умножим число на такое, у которого все биты равны единице, кроме бита под номером n .
Например

$$0001011_2 \& 1110111_2 = 0000011_2$$

Чтобы получить такую маску, сначала создадим число с нулями и одной единицей, а потом инвертируем его.

```
int unsetbit(const int value, const int position) {  
    return (value & ~(1 << position));  
}
```

Примеры

Функция, изменяющая значение n-го бита на противоположное.

Для этого воспользуемся функцией исключающего или: применим операцию XOR к числу, которое состоит из одних нулей и одной единицы на месте нужного бита

$$0001011_2 \wedge 0010000_2 = 0011011_2$$

Чтобы получить такую маску, сначала создадим число с нулями и одной единицей, а потом инвертируем его.

Примеры

Функция, изменяющая значение n-го бита на противоположное.

Для этого воспользуемся функцией исключающего или: применим операцию XOR к числу, которое состоит из одних нулей и одной единицы на месте нужного бита

$$0001011_2 \wedge 0010000_2 = 0011011_2$$

Чтобы получить такую маску, сначала создадим число с нулями и одной единицей, а потом инвертируем его.

```
int switchbit(const int value, const int position) {  
    return (value ^ (1 << position));  
}
```

Листинг 6. Программа изменения битов числа

```

1  #include <stdio.h>
2  #include <limits.h>
3
4  int checkbit(const int value, const int
      position) {
5      return ((value & (1 << position)) != 0);
6  }
7  int setbit(const int value, const int
      position) {
8      return (value | (1 << position));
9  }
10 int unsetbit(const int value, const int
      position) {
11     return (value & ~(1 << position));
12 }
13 int switchbit(const int value, const int
      position) {
14     return (value ^ (1 << position));
15 }
16
17 void printbits(int n) {
18     int len = sizeof(int)* CHAR_BIT;
19     for (int i = len-1; i >= 0; i--) {
20         printf("%d", checkbit(n, i));
21     }
22     printf("\n");
23 }
24
25 int main() {
26     int a;
27     printf("Введите число a: ");
28     scanf("%d", &a);
29
30     printbits(a);
31     a = setbit(a, 5);
32     printbits(a);
33     a = unsetbit(a, 5);
34     printbits(a);
35     a = switchbit(a, 11);
36     printbits(a);
37     a = switchbit(a, 11);
38     printbits(a);
39
40     getchar();getchar();
41 }

```

Битовые флаги

Листинг 7. Проверка трех флагов

```
1  #include <stdio.h>
2
3  int main() {
4      unsigned char a = 1, b = 0, c = 0;
5
6      if (a) {
7          if (b) {
8              if (c) {
9                  printf("true true true");
10             } else {
11                 printf("true true false");
12             }
13         } else {
14             if (c) {
15                 printf("true false true");
16             } else {
17                 printf("true false false");
18             }
19         }
20     } else {
21         if (b) {
22             if (c) {
23                 printf("false true true");
24             }
25             else {
26                 printf("false true false");
27             }
28         }
29         else {
30             if (c) {
31                 printf("false false true");
32             }
33             else {
34                 printf("false false false");
35             }
36         }
37     }
38
39     getchar();
40     return 0;
41 }
```


Битовые флаги

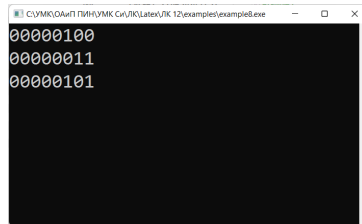
Листинг 8. Сохранение флагов в одну переменную

```
1  #include <stdio.h>
2  #include <limits.h>
3
4  void printbits (int n) {
5      for (int i = CHAR_BIT - 1; i >= 0; i--)
6          printf("%d", (n & (1 << i)) != 0);
7      }
8      printf("\n");
9  }
10
11 int main() {
12     unsigned char a, b, c;
13     unsigned char res;
14
15     a = 1; b = 0; c = 0;
16     res = c | b << 1 | a << 2;
17     printbits(res);
18
19     a = 0; b = 1; c = 1;
20     res = c | b << 1 | a << 2;
21     printbits(res);
22
23     a = 1; b = 0; c = 1;
24     res = c | b << 1 | a << 2;
25     printbits(res);
26
27     getchar();
28     return 0;
29 }
```

Битовые флаги

Листинг 8. Сохранение флагов в одну переменную

```
1  #include <stdio.h>
2  #include <limits.h>
3
4  void printbits (int n) {
5      for (int i = CHAR_BIT - 1; i >= 0; i--)
6          printf("%d", (n & (1 << i)) != 0);
7      printf("\n");
8  }
9
10
11 int main() {
12     unsigned char a, b, c;
13     unsigned char res;
14
15     a = 1; b = 0; c = 0;
16     res = c | b << 1 | a << 2;
17     printbits(res);
18
19     a = 0; b = 1; c = 1;
20     res = c | b << 1 | a << 2;
21     printbits(res);
22
23     a = 1; b = 0; c = 1;
24     res = c | b << 1 | a << 2;
25     printbits(res);
26
27     getchar();
28     return 0;
29 }
```



```
C:\УМК\ОАиП\ПИН\УМК Си\ЛК\Latex\ЛК 12\examples\example8.exe
00000100
00000011
00000101
```

Битовые флаги

Листинг 9. Проверка флагов, занесенных в одну переменную

```
1  #include <stdio.h>
2
3  int main() {
4      unsigned char a, b, c;
5      unsigned char res;
6      a = 1;
7      b = 0;
8      c = 0;
9
10     res = c | b<< 1 | a << 2;
11     switch (res) {
12         case 0b00000000:
13             printf("false false false");
14             break;
15         case 0b00000001:
16             printf("false false true");
17             break;
18         case 0b00000010:
19             printf("false true false");
20             break;
21         case 0b00000011:
22             printf("false true true");
23             break;
24         case 0b00000100:
25             printf("true false false");
26             break;
27         case 0b00000101:
28             printf("true false true");
29             break;
30         case 0b00000110:
31             printf("true true false");
32             break;
33         case 0b00000111:
34             printf("true true true");
35             break;
36     }
37     getchar();
38     return 0;
39 }
40 }
```

Практическое применение битовых флагов

Этот метод очень часто используется для назначения опций функций в разных языках программирования. Каждый флаг принимает своё уникальное название, а их совместное значение как логическая сумма всех используемых флагов.

Например,

```
char *fp = "/home/ec2-user/file.txt";
int flag = O_RDWR | O_CREAT | O_TRUNC | O_APPEND;
int fd = open(fp, flag, 0644);
```

Здесь флаг `O RDWR` равен

[illegible]

О CREAT равен

[illegible]

0 TRUNC равен

$$\overline{00000000000000000000000000}100000000_2 = 512_{10}$$

и 0 APPEND

$$\overline{00000000000000000000000000000000}1000_2 = 8_{10}$$