ПОБИТОВЫЕ ОПЕРАЦИИ

ЧТО ТАКОЕ ПОБИТОВЫЕ ОПЕРАЦИИ И ЗАЧЕМ ОНИ НУЖНЫ

Побитовые операторы манипулируют отдельными битами в пределах переменной.

В далёком прошлом компьютерной памяти было очень мало и ею сильно дорожили. Это было стимулом максимально разумно использовать каждый доступный бит.

Например, в **логическом типе данных bool** есть всего лишь два возможных значения (true и false), которые могут быть представлены одним битом, но по факту занимают целый байт памяти! А это, в свою очередь, из-за того, что переменные используют уникальные адреса памяти, а они выделяются только в байтах. Переменная bool занимает 1 бит, а другие 7 тратятся впустую.

Используя побитовые операторы, можно создавать функции, которые позволят уместить 8 значений типа bool в переменной размером 1 байт, что значительно сэкономит потребление памяти. В прошлом такой трюк был очень популярен. Но сегодня, по крайней мере, в прикладном программировании, это не так.

ЧТО ТАКОЕ ПОБИТОВЫЕ ОПЕРАЦИИ И ЗАЧЕМ ОНИ НУЖНЫ

С поддерживает все существующие битовые операторы. Поскольку С создавался, чтобы заменить ассемблер, то была необходимость поддержки всех (или по крайней мере большинства) операций, которые может выполнить ассемблер.

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

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

При работе с побитовыми операторами используйте целочисленные типы данных unsigned.

ПОБИТОВЫЕ ОПЕРАЦИИ

Оператор	Символ	Пример	Операция
Побитовый сдвиг влево	<<	x << y	Все биты в x смещаются влево на y бит
Побитовый сдвиг вправо	>>	x >> y	Все биты в x смещаются вправо на y бит
Побитовое НЕ	~	~x	Все биты в x меняются на противоположные
Побитовое И	&	x & y	Каждый бит в x И каждый бит в y
Побитовое ИЛИ	|	x | y	Каждый бит в x ИЛИ каждый бит в y
Побитовое исключающее ИЛИ (XOR)	^	x ^ y	Каждый бит в x XOR каждый бит в y

Правило: При работе с побитовыми операторами используйте целочисленные типы данных unsigned.

& : ПОРАЗРЯДНАЯ КОНЪЮНКЦИЯ

(операция «И» или поразрядное умножение). Возвращает 1, если оба из соответствующих разрядов обоих чисел равны 1.

Рассмотрим выражение 14 & 5:

1 1 1 0 // 14

0 1 0 1 // 5

---

0 1 0 0 // 4

| : ПОРАЗРЯДНАЯ ДИЗЪЮНКЦИЯ

(операция «ИЛИ» или поразрядное сложение). Возвращает 1, если хотя бы один из соответствующих разрядов обоих чисел равен 1.

Рассмотрим выражение 10 | 7:

1 0 1 0 // 10

0 1 1 1 // 7

---

1 1 1 1 // 15

^ : ИСКЛЮЧАЮЩЕЕ “ИЛИ”

Побитовое исключающее ИЛИ (^) (англ. «XOR» от «eXclusive OR«). При обработке двух операндов, исключающее ИЛИ возвращает true (1), только если один и только один из операндов является истинным (1). Если таких нет или все операнды равны 1, то результатом будет false (0).

Рассмотрим выражение 6 ^ 3:

0 1 1 0 // 6

0 0 1 1 // 3

----

0 1 0 1 // 5

~ : ПОРАЗРЯДНОЕ ОТРИЦАНИЕ ИЛИ ИНВЕРСИЯ.

Инвертирует все разряды операнда. Если разряд равен 1, то он становится равен 0, а если он равен 0, то он получает значение 1.

Рассмотрим выражение ~ 9:

1 0 0 1// 9

0 1 1 0 // 6

ОПЕРАЦИИ АРИФМЕТИЧЕСКОГО СДВИГА

Операции битового сдвига могут быть полезны при декодировании информации от внешних устройств и для чтения информации о статусе.

Операторы битового сдвига могут также использоваться для выполнения быстрого умножения и деления целых чисел.

ОПЕРАТОР ПОБИТОВОГО АРИФМЕТИЧЕСКОГО СДВИГА ВПРАВО >>: *A*>> *B*

Оператор **>>** сдвигает вправо биты выражения *A* на количество битов, указанных в выражении *B*.

Для заполнения позиций слева **используется бит знака** значения *A*.

Цифры, сдвинутые за пределы диапазона, удаляются.

Тип данных, возвращаемых данным оператором, определяется типом данных выражения *A*.

Например:

```
short int temp
temp = -14 >> 2
```

после вычисления этого кода переменная temp имеет значение -4, поскольку при сдвиге значения -14 (11110010 в двоичном выражении) на два бита в право получается значение -4 (11111100 в двоичном выражении).

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
код	Знак	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01
Прям.	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0

Получили дополнительный код отрицательного числа. Проделаем обратную процедуру, чтобы получить прямой код числа и применим позиционную формулу для получения десятичного числа
Обратный код: 1’111111111111011: Прямой код: 1’000000000000100 = - 4_{10}

ОПЕРАТОР ПОБИТОВОГО АРИФМЕТИЧЕСКОГО СДВИГА ВЛЕВО <<: *A<<B*

Оператор **<<** сдвигает влево биты выражения *A* на количество битов, указанных в выражении *B*.

“Выталкиваемые наружу” биты пропадают, **освобождающиеся биты заполняются нулями**.

Тип данных, возвращаемых данным оператором, определяется типом данных выражения *A*.

Например:

```
short int temp
temp = -14 << 2
```

после вычисления этого кода переменная temp имеет значение -56, поскольку при сдвиге значения -14 (11110010 в двоичном выражении) на два бита влево получается значение -56 (10111000 в двоичном выражении).

ОСОБЕННОСТИ ПРИМЕНЕНИЯ СДВИГИ

Операторы битового сдвига могут также использоваться для выполнения быстрого умножения и деления целых чисел. Сдвиг влево равносилен умножению на 2, а сдвиг вправо - делению на 2 (четных чисел).

Сдвинутые биты теряются, а с другого конца появляются нули. В том случае, если вправо сдвигается отрицательное число, слева появляются единицы (поддерживается знаковый бит).

ПРИМЕР

	Битовое представление x после выполнения каждого оператора	Значение x
char x;		
x = 7;	00000111	7
x = x << 1;	00001110	14
x = x << 3;	01110000	112
x = x << 2;	11000000	192
x = x >> 1;	01100000	96
x = x >> 2;	00011000	24

Каждый сдвиг влево приводит к умножению на 2. Обратим внимание, что после сдвига x << 2 информация теряется, поскольку биты сдвигаются за конец байта.

Каждый сдвиг вправо приводит к делению на 2. Обратим внимание, что деление не вернуло потерянные при умножении биты.

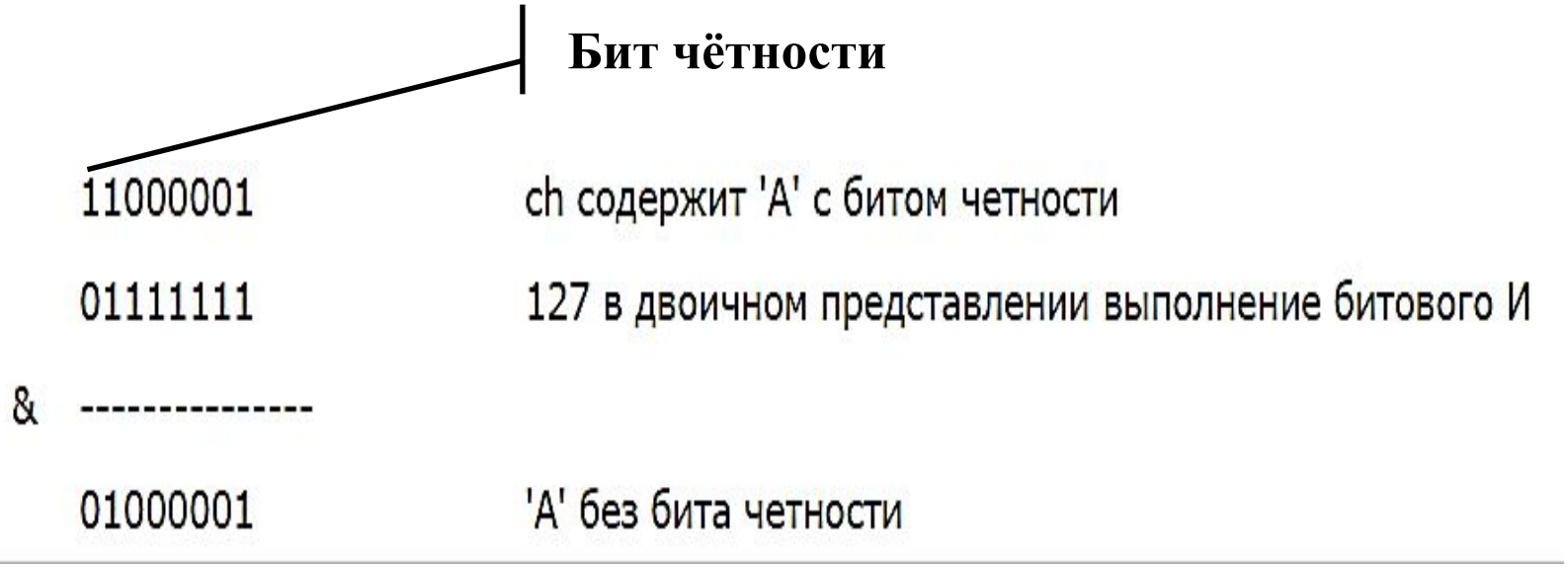
ДЛЯ ЧЕГО ПРИМЕНЯЮТСЯ БИТОВЫЕ ОПЕРАЦИИ

Битовое «И» чаще всего используется для выключения битов: любой бит, установленный в 0, вызывает установку соответствующего бита в другом операнде также в 0. Например, следующий фрагмент программы читает символы, вводимые с консоли, и сбрасывает бит четности в 0:

```
char ch, ch1;
   cin >> ch;
   ch1 = ch & 127;
   cout << ch1<<endl;
```

В последовательной передаче данных часто используется формат 7 бит данных, бит чётности, один или два стоповых бита. Такой формат аккуратно размещает все 7-битные ASCII символы в удобный 8-битный байт.

В следующем примере показано, как работает данный фрагмент программы с битами. В нём предполагается, что ch имеет символ 'A' и имеет бит четности:



В результате работы программы чётность, отображаемая восьмым битом, устанавливается в 0 с помощью битового «И», поскольку биты с номерами от 1 до 7 установлены в 1, а бит с номером 8 — в 0. Выражение ch & 127 означает, что выполняется битовая операция «И» между битами переменной ch и битами числа 127.

В результате получим ch со сброшенным старшим битом.

ДЛЯ ЧЕГО ПРИМЕНЯЮТСЯ БИТОВЫЕ ОПЕРАЦИИ

Битовое «ИЛИ» может использоваться для установки битов: любой бит, установленный в любом операнде, вызывает установку соответствующего бита в другом операнде. Например, в результате операции 128 | 3 получаем:

	10000000	128 в двоичном представлении
	00000011	3 в двоичном представлении
|	----------	битовое ИЛИ
	10000011	результат

ДЛЯ ЧЕГО ПРИМЕНЯЮТСЯ БИТОВЫЕ ОПЕРАЦИИ

Исключающее ИЛИ (XOR) *) устанавливает бит, если соответствующие биты в операндах отличаются.*
Например, в результате операции 127 ^ 120 получаем:

```
  01111111          127 в двоичном представлении
  01111000          120 к двоичном представлении битовое исключающее
^ --------------    ИЛИ
  00000111          результат
```

ОПЕРАЦИИ

Оператор *битового дополнения* ~ инвертирует состояние каждого бита указанной переменной, то есть 1 устанавливается в 0, а 0 — в 1.

Битовые операторы часто используются в процедурах шифрования. Если есть желание сделать дисковый файл нечитабельным, можно выполнить над ним битовую операцию.

Одним из простейших методов является использование битового дополнения для инверсии каждого бита в байте, как показано ниже:

Исходный байт	00101100
После первого дополнения	11010011
После второго дополнения	00101100

Следует обратить внимание, что в результате выполнения двух битовых дополнений получаем исходное число. Следовательно, первое дополнение будет создавать кодированную версию байта, а второе будет декодировать.

ПОБИТОВЫЕ ОПЕРАТОРЫ ПРИСВАИВАНИЯ

Оператор	Символ	Пример	Операция
Присваивание с побитовым сдвигом влево	<<=	x <<= y	Сдвигаем биты в x влево на y бит
Присваивание с побитовым сдвигом вправо	>>=	x >>= y	Сдвигаем биты в x вправо на y бит
Присваивание с побитовой операцией ИЛИ	|=	x |= y	Присваивание результата выражения x | y переменной x
Присваивание с побитовой операцией И	&=	x &= y	Присваивание результата выражения x & y переменной x
Присваивание с побитовой операцией исключающего ИЛИ	^=	x ^= y	Присваивание результата выражения x ^ y переменной x

Все битовые операции выполняются слева направо. В следующей строке приведены битовые операции в порядке уменьшения их приоритета.

~, << >>, &, ^, |

ЦЕЛОЧИСЛЕННЫЕ КОНСТАНТЫ НА С++

Целочисленные данные в языке Си могут быть представлены в одной из следующих систем счисления:

Десятичные	Последовательность цифр (`0 – 9`), которая начинаются с цифры, отличной от нуля. Пример: 1, -29, 385. Исключение — число 0.
Восьмеричные	Последовательность цифр (`0 – 7`), которая всегда начинается с `0`. Пример: 00, 071, -052, -03.
Шестнадцатеричные	Последовательность шестнадцатеричных цифр (`0 – 9` и `A – F`), которой предшествует присутствует `0x` или `0X`. Пример: 0x0, 0x1, -0x2AF, 0x17.

По умолчанию целочисленные константы имеют тип int.

ПРИМЕР ЗАДАЧИ НА УСТАНОВКУ НЕОБХОДИМЫХ БИТОВ.

Написать программу, которая позволит ввести два числа типа unsigned int с клавиатуры, найти и вывести на консоль их сумму, далее, используя битовые операции сделать в ней, чтобы 2-й и 1-й биты были равны 0, 3-й бит - равен 1, а остальные сохранили свои значения, вывести результат.

Наша задача – подобрать такие двоичные константы, которые позволят сделать необходимые операции с указанными битами.

ПОМНИМ: нумерация битов начинается слева и с нуля!

```
int main()     // главная функция программы
{
    unsigned int a, b, sum; /* описание типов переменных */

setlocale(LC_ALL, "rus"); // для вывода русского шрифта в консоль
printf("\nВведите a\n");
scanf_s("%u", &a); /* вводим a */
printf("\nВведите b\n");
scanf_s("%u", &b); /* вводим a */

sum = a + b; /* нашли сумму */
printf("\nСумма равна a и b =%u", sum); /* вывели сумму на
монитор */
sum = sum & 0xfff9; /* установили 2 и 1 биты в 0
                fff9, в двоичной системе
                1111 1111 1111 1111 1111 1111 1111 1001*/
sum |= 0x8;         /* установили 3 бит в 1
              8, в двоичной системе
              0000 0000 0000 0000 0000 0000 0000 1000 */
printf("\nПосле преобразования sum=%u\n", sum);
system("pause");
return 0; // вернулись в среду разработки
}
```

РЕЗУЛЬТАТЫ РАБОТЫ ПРОГРАММЫ

```
Введите a
20

Введите b
2

Сумма равна a и b =22
После преобразования sum=24
Для продолжения нажмите любую клавишу . . .
```

gned int

ю эта константа

2 бита, но

п unsigned int и

ивать.

0000 0000 0001 0000 –1-й и 2-й биты выставлены в 0.

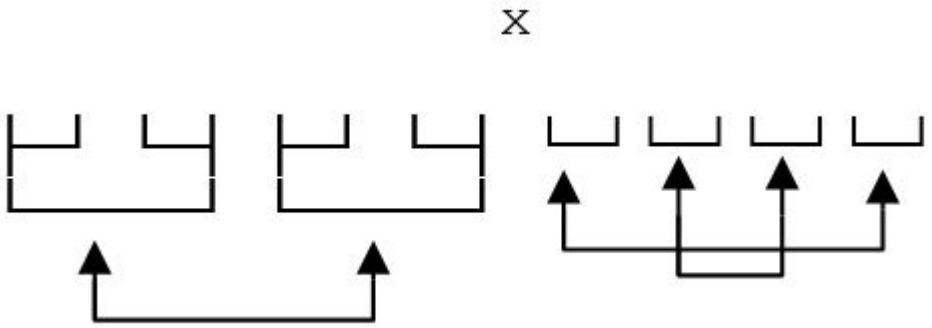
Побитовое ИЛИ полученного числа с числом 0x8 даст следующий результат:

```
0000 0000 0001 0000
0000 0000 0000 1000
---------------------
0000 0000 0001 1000 –3-й бит выставлены в 1.
```

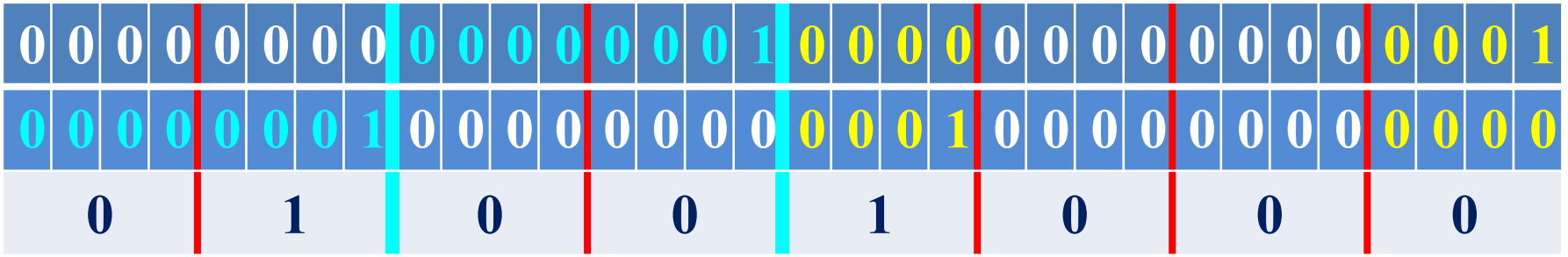
Полученное число $0000\ 0000\ 0001\ 1000_2 = 24_{10}$. Это соответствует результатам работы программы.

ПРИМЕР ЗАДАЧИ НА ПРИМЕНЕНИЕ БИТОВЫХ ОПЕРАЦИЙ

Написать программу, которая позволит ввести число x типа unsigned int с клавиатуры, напечатать его и, используя битовые операции, поменять в нем четверки и восьмерки бит по схеме



КОНТРОЛЬНЫЙ ПРИМЕР ДЛЯ $65537_{10} = 10001_{16}$



ПОЛУЧИЛИ: $1001000_{16} = 16^6 + 16^3 = 16781312_{10}$

```
int main()
{unsigned int lx, l41, l42, l43, l44, l83;
setlocale(LC_ALL, "rus"); // для вывода русского шрифта в консоль
printf("\n Введите число : "); /* поясняющая надпись */
scanf_s("%ld", &lx); /* вводим число */
printf("\n Введено число lx = %ld ( 16-format:  %X) \n", lx,lx); /*
вывели число */
l41 = lx & 0xf; /* нашли первую четверку */
l42 = lx & 0xf0; /* нашли вторую четверку */
l43 = lx & 0xf00; /* нашли третью четверку */
l44 = lx & 0xf000; /* нашли четвертую четверку */
l83 = lx & 0xff0000; /* нашли третью восьмерку */
/* поставили четвертую восьмерку на место третьей
и обнулили младшие шестнадцать и старшие 8 бит */
lx = (lx >> 8) & 0xff0000;
/* поставили все на место */
lx += (l41 << 12) + (l42 << 4) + (l43 >> 4) + (l44 >> 12) + (l83 <<
8);
/* вывели полученное значение на монитор */
printf("\n После преобразования число равно %ld ( 16-format:  %X)\n",
lx, lx);

system("pause");  return 0; }
```

```
Введите число : 65537

Введено число lx = 65537 ( 16-format:  10001)

После преобразования число равно 16781312 ( 16-format:  1001000)
Для продолжения нажмите любую клавишу . . .
```

ПОЛУЧИЛИ: $1001000_{16} = 16^6 + 16^3 = 16781312_{10}$

Пример задачи на применение битовых операций

Дано число k, $0 \le k \le 31$. Не используя арифметические операторы сложения, умножения, вычитания, деления, взятия остатка, вычислить 2^k, применяя побитовые операторы &, |, ~, ^, <<, >>.

Контрольный пример:

$2^0 = 1_{10} = 00000001_2$

$2^1 = 2_{10} = 00000010_2$

$2^2 = 4_{10} = 00000100_2$

.....

Очевидно, что степень k позволяет сместить 1 влево на k позиций

```
int main()
{
unsigned int k, n=1,m; /* описание типов переменных */
setlocale(LC_ALL, "rus"); // для вывода русского шрифта в консоль
printf("\n Введите число k: "); /* поясняющая надпись */
scanf_s("%ld", &k); /* вводим число */
printf("\n Введено число k = %ld  \n", k); /* вывели число */
m = n << k;
printf("\n 2^k = %ld ( 16-format:  %X)\n", m, m);

system("pause");  return 0; // вернулись в операционную систему
}
```

```
Введите число k: 10

Введено число k = 10

2^k = 1024 ( 16-format:  400)
Для продолжения нажмите любую клавишу . . .
```

```
Введите число k: 30

Введено число k = 30

2^k = 1073741824 ( 16-format:  40000000)
Для продолжения нажмите любую клавишу . . .
```