Битовые операторы. Логические побитовые операции и операции сдвига. Средства выравнивания. Примеры.

Один из методов *манипулирования битами*, предполагает создание в структуре *битовых полей*. Это набор соседствующих битов внутри значения типа ***signed int*** или ***unsigned int***.
Битовое поле создаётся в структуре при помощи **оператора :** при этом должно быть размечено каждое поле и определен его размер.

```
struct{
unsigned int autfd : 1;
unsigned int bldfc : 1;
unsigned int undln : 1;
unsigned int itals : 1;
} prnt;
prnt.itals = 0;
prnt.undln = 1;
```

Пример:

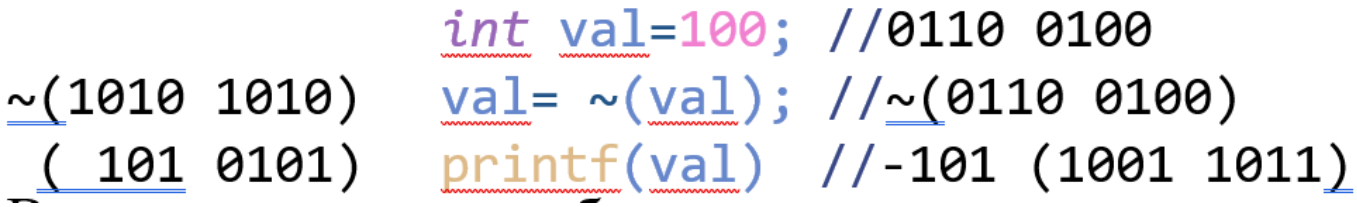
```
//создание структуры
typedef struct bitlaptop
{
    unsigned int cpuType: 2;
    unsigned int ramV: 3;
    unsigned int hasGPU: 1;
    unsigned int osSet: 2;
    unsigned int storT: 2;
    unsigned int storV: 4;
    unsigned int screenS: 2;
} BITLAPTOP; //структура 2 байта

static char *cpu[] = {"Intel", "AMD", "Apple Silicon"};
static const int ramVol[] = {8, 16, 32, 64};
static const char *os[] = {"Linux", "macOS", "Windows"};
static const char *storageT[] = {"HDD", "SSD", "SSD+NVMe"};
static const int storageV[] = {128, 256, 512, 1024, 2048};
static const _char_ *screen[] = {"<14''",">14'' && <=15''",">15'' &&
<=16''",">=17''"};
```

```
void displayLaptopSpecs(const BITLAPTOP * bl) {
printf("Тип процессора: %s\n", bl->cpuType < 3 ? cpu[bl->cpuType] :
"Неизвестно");
printf("Объем оперативной памяти: %d ГБ\n", bl->ramV < 5 ? ramVol[bl->ramV] :
0);
printf("Наличие дискретного графического ускорителя: %s\n", bl->hasGPU ?
"Есть" : "Нет");
printf("Семейство ОС: %s\n", bl->osSet < 3 ? os[bl->osSet] : "Неизвестно");
printf("Тип накопителя: %s\n", bl->storT < 3 ? storageT[bl->storT] :
"Неизвестно");
printf("Размер накопителя: %d ГБ\n", bl->storV < 5 ? storageV[bl->storV] : 0);
printf("Размер экрана: %s дюймов\n", screen[bl->screenS]);}
```

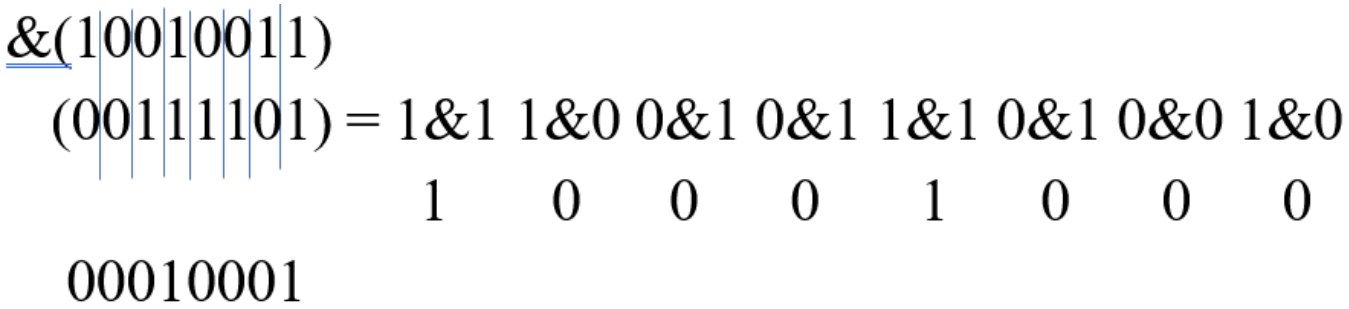
Группы логических операций в языке Си разделены на две группы:

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

Операция дополнение до единицы или **побитовое отрицание: ~** (тильда)
Унарный оператор ~ преобразует каждую единицу в ноль, а каждый ноль в единицу.

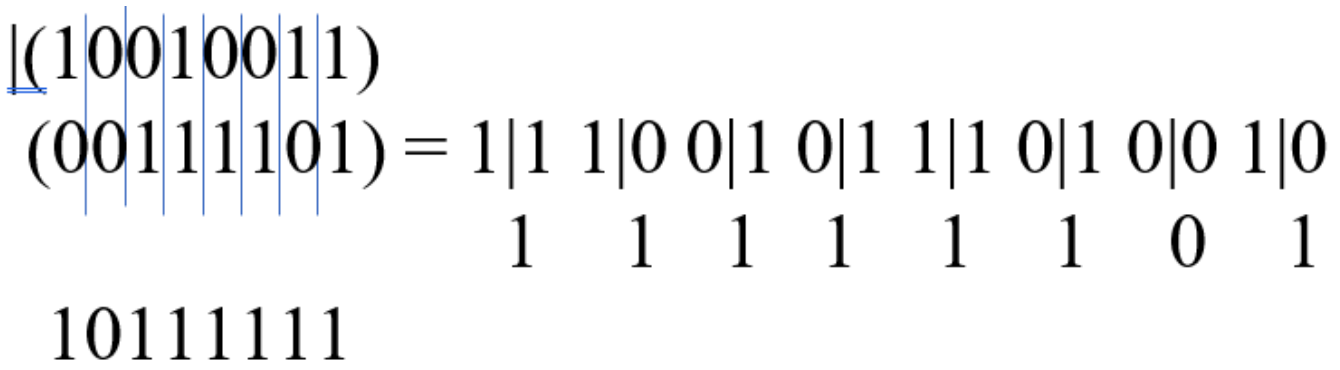


Побитовая операция "И": & (амперсант)
Бинарный оператор & выполняет побитовое сравнение двух операндов. Бит в каждой позиции числа (справа налево) будет равен единице, тогда и только тогда, когда оба соответствующих бита в операндах равны 1.

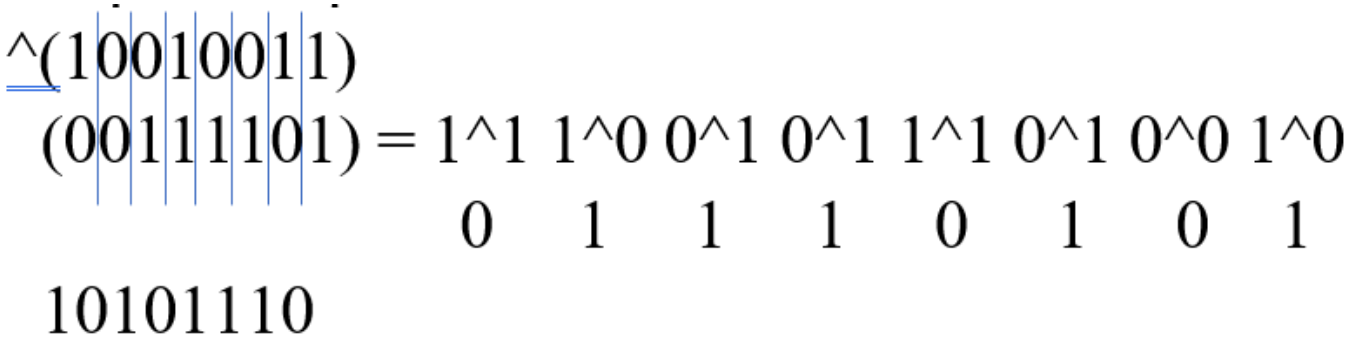


Побитовая операция "ИЛИ": | (терминатор)
Бинарный оператор | выполняет побитовое сравнение двух операндов. Для каждой

позиции бит будет равен 1, если хотя бы один из двух битов будет равен 1.



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.



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

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

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.

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

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

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;
```

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

```
flags ^= MASK;
```

113, 0161, 0x71

6, 06, 0x6

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

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

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

Побитовая бинарная операция сдвиг вправо ">>"

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 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)
  Сокращенные формы записи:

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

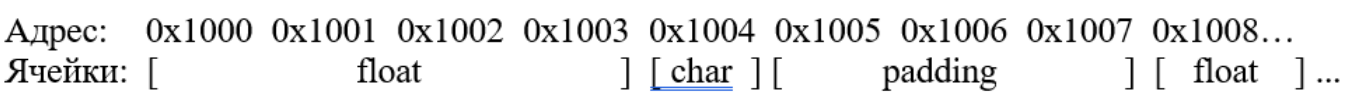
Выравнивание

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.

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);
```




```
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
```

В качестве значения для выравнивания берётся степень двойки, более **высокие** значения имеют более *строгий* или более *жесткий,* в отличии от более **низких значений, в то время как более низкие** значения трактуются, как более *слабые*.
Более строгое вырывание для типа данных или для переменной можно получить при помощи спецификатора _Alignas. При помощи _Alignas можно запрашивать любое выравнивание.
Применение выравнивания чаще всего связано с использованием строгости в при передачи данных в различных **сетевых протоколах**, а также в технологиях связанных с аппаратной реализацией **CPU** и **GPU**.