21.10.2024

Манипулирование битами

Филиппов Михаил Витальевич

m.filippov@g.nsu.ru

89232283872

Императивное программирование, 2024-2025

Новосибирский государственный университет

*НАСТОЯЩАЯ НАУКА

Давайте познакомимся

Филиппов Михаил Витальевич

- Окончил магистратуру ФФ НГУ
- Окончил аспирантуру ИТ СО РАН
- Являюсь м.н.с. ИТ СО РАН
- 7+ лет опыт в программировании C/C++

Адженда

Биты и только биты

90 минут

Двоичные числа, биты и байты

Пример 2157 в 10 записи:

2 x 1000 + 1 x 100 + 5 x 10 + 7 x 1

$2 \times 10^3 + 1 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$

Пример 1101 в 2 записи:

$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

В 10 записи:

1 x 8 + 1 x 4 + 0 x 2 + 1 x 1 = 13

Пример .101 в 2 записи:

1/2 + 0/4 + 1/8

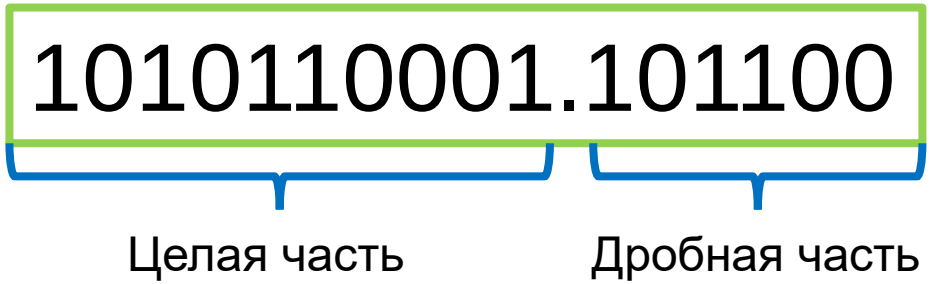
В 10 записи:

0.50 + 0.00 + 0.125 = 0.625.

Номер бита	7	6	5	4	3	2	1	0
	0	1	0	0	1	0	0	1
Значение бита	128	64	32	16	8	4	2	1

Что такое фиксированная точка

Числа с фиксированной точкой — это двоичные числа, у которых ограничен размер их целой и дробной части. Например, если число состоит из 16 битов, то мы можем выделить для целой части первые 10 битов и оставшиеся шесть — для дробной части.



689.6875

Что такое фиксированная точка

- ✓ они всегда представляются в виде конечного числа
- ✓ скорость обработки компьютером

- ✗ Маленький диапазон значений
- ✗ Дробный минимальный шаг
- ✗ Низкая точность

$$1111111111.111111_2 = 1023.983475_{10}$$
$$1111111111.111110_2 = 1023.96875_{10}$$

Что такое плавающая точка

Число с плавающей точкой (или число с плавающей запятой) — это численное представление вещественного числа в программировании. Оно является его приближённым значением.

- **Знаковый бит** указывает, положительное число или отрицательное.
- **Экспонента** показывает, на какое число нужно умножать мантиссу.
- **Мантисса** — это фиксированное количество битов, которое выражает точность числа.

Представление числа в научной нотации.

- **Знак числа** указывает, какое это число: положительное или отрицательное.
- **Коэффициент** — это основная часть десятичного числа, записанного обычно в диапазоне от 1 до 9.
- **Мантисса** — это дробная часть коэффициента.
- **Экспонента** — это то, на что мы умножаем коэффициент.

$$1.21 \times 10^{-7} = (+1) \times 1.21 \times 10^{-7}$$

Знаковый бит | Мантисса | Экспонента

- +1 — знак числа (положительный),
- 0.21 — мантисса,
- −8 — экспонента.

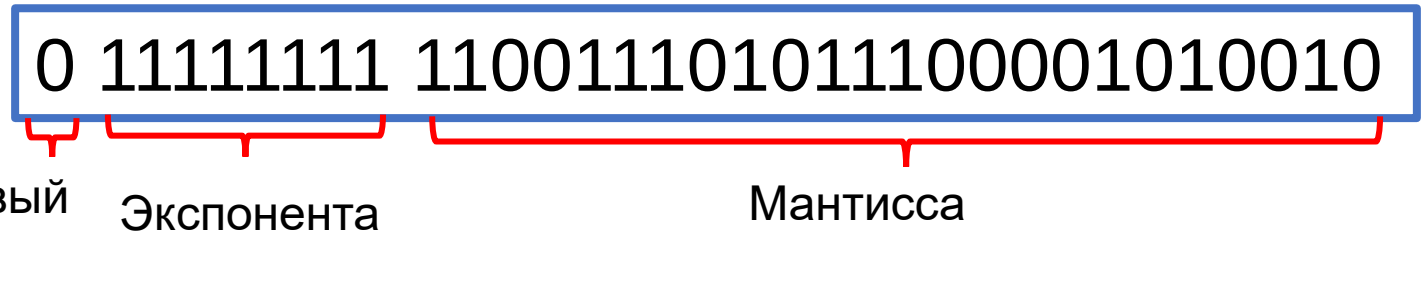
Как представляют числа с плавающей точкой

Стандарт IEEE 754 — это набор правил, которые описывают, как вещественные числа представляются в компьютере. Этот формат стал самым распространённым в программировании, когда дело доходит до арифметики чисел с плавающей точкой.

Числа представляются фиксированным количеством битов, каждый из которых отвечает своим задачам.

В IEEE 754 обычно используют 32 бита. Они делятся на всё те же категории:

- один знаковый бит
- 8 битов для экспоненты (то, на что мы умножаем мантиссу)
- 23 бита для мантиссы (она выражает точность числа)



Перевод из битового представления в число: $(-\mathbf{1})^{З} \times (\mathbf{1} + М) \times \mathbf{2}^{Э-П}$.Здесь З — это знаковый бит, М — мантисса, Э — экспонента, П - порядок.

https://drive.google.com/file/d/15kGy_VNGJAarsoujM_yYYxUNHcEamZFR/view?usp=sharing

Как представляют числа с плавающей точкой

Точность	Размер	Мантисса	Экспонента	Диапазон	Погрешность
половинная	16 бит	10 бит	5 бит	$10^{-4} \ldots 10^{4}$	3 знака
одинарная	32 бита	23 бита	8 бит	$10^{-38} \ldots 10^{38}$	7 знаков
двойная	64 бита	52 бита	11 бит	$10^{-307} \ldots 10^{307}$	16 знаков
четверная	128 бит	112 бит	15 бит	$10^{-4931} \ldots 10^{4931}$	34 знаков

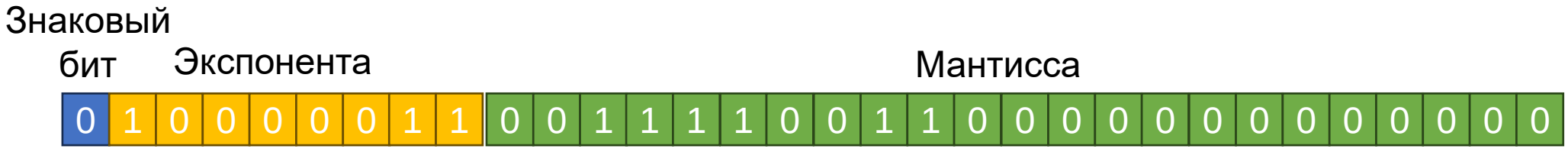
Ноль со знаком

Бесконечность со знаком

Неопределенность (NaN)

Пример

19.59375 в 32 бита



1 шаг – число положительное или отрицательное (0 или 1)
1 бит = 0

2 шаг – представление числа в двоичном формате
10011.10011

3 шаг – нормализация двоичного представления
1.001110011×2^4
4 шаг – прибавить порядок для нахождения экспоненты
$4 + 127 = 131_{10} = 10000011_2$
5 шаг – удалить ведущую 1 из мантиссы
1.001110011 -> 0011110011

Побитовые операции

Язык С предлагает два вида побитовых операций:

- логические операции
- операции сдвига.

В действительной программе вы будете применять целочисленные переменные или константы в обычных формах. Например, вместо 00011001 будет использоваться запись 25, 031 или 0x19.

Побитовые логические операции - отрицание

Унарная операция ~ - преобразует каждую единицу в ноль, а каждый ноль в единицу, как показано в следующем примере:
~(10011010) // выражение
(01100101) // результат

```c
#include <stdio.h>

int main(void)
{
    unsigned char i = 2; // 00000010
    printf("%hhu", ~i);  // 11111101
    return 0;
}
// 253
```

Побитовые логические операции – операция И

Двоичная операция & создает новое значение за счет выполнения побитового сравнения двух операндов. Для каждой позиции результирующий бит будет равен 1, только если оба соответствующих бита в операндах равны 1.

Пример:

(10010011) & (00111101) = (00010001)

В C также имеется операция “И”, объединенная с присваиванием: &=.

Оператор

val &= 0377;

дает такой же результат, как и следующий оператор:

val = val & 0377;

Побитовые логические операции – операция ИЛИ

Двоичная операция | создает новое значение за счет выполнения побитового сравнения двух операндов. Для каждой позиции бит будет равен 1, если любой из соответствующих битов в операндах равен 1.

Пример:

(10010011) | (00111101) = (10111111)

В C также существует операция “ИЛИ”, объединенная с присваиванием: |=.

Оператор

val |= 0377;

даст тот же результат, что и следующий оператор:

val = val | 0377;

Побитовые логические операции – операция исключающее ИЛИ

Двоичная операция ^ выполняет побитовое сравнение двух операндов. Для каждой позиции результирующий бит будет равен 1, если один или другой (но не оба) из соответствующих битов в операндах равен 1.

Пример:

(10010011) ^ (00111101) = (10101110)

В языке С также имеется операция “исключающее ИЛИ”, объединенная с присваиванием: ^=.

Оператор

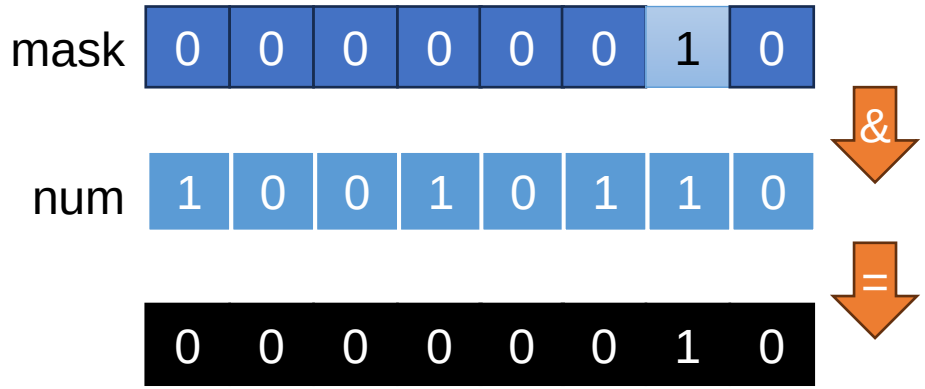
val ^= 0377;

дает тот же результат, что и следующий оператор:

val = val ^ 0377;

Случай применения: маски

Маска — это комбинация битов, в которой некоторые биты включены (1), а некоторые выключены (0).
Для примера предположим, что вы определили символическую константу MASK как 2 (т.е. 00000010), у которой ненулевым является только бит с номером 1. Тогда оператор
num = num & MASK;
приведет к установке всех битов flags (кроме первого) в 0, т.к. любой бит, объединяемый с 0 посредством операции “И”, дает 0. Бит номер 1 переменной остается неизменным.
Распространенный случай: ch &= 0xff; или ch &= 0377;



Случай применения: включение (установка) битов

Иногда требуется включить отдельные биты в значении, оставив остальные без изменений. Например, компьютер IBM PC управляет оборудованием, отправляя нужные значения в порты. Для активизации, скажем, динамика, не обходимо включить бит 1, а остальные биты оставить неизменными. Этого можно сделать с помощью побитовой операции “ИЛИ”. Например, пусть имеется константа MASK, в которой бит 1 установлен в 1. Тогда оператор

```
flags = flags | MASK;
```

Случай применения: выключение (очистка) битов

Точно так же, как удобно иметь возможность включать отдельные биты, не затрагивая остальные, не менее удобно располагать возможностью их выключения. Предположим, что требуется отключить бит номер 1 в переменной flags. И снова MASK имеет включенный только бит 1. Можно воспользоваться следующим оператором:

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

Например, пусть flags равно 00001111 и MASK — 10110110. Выражение

```
flags &~ MASK
```

становится

```
(00001111) &~ (10110110)
// выражение
```

и после вычисления дает следующий результат:

```
(00001001)
```

Случай применения: переключение битов

Переключение бита означает его выключение, если он включен, и включение, если выключен. Для переключения битов можно применять побитовую операцию исключающего “ИЛИ”. Идея в том, что если b — это установленное состояние бита (1 или 0), то 1 ^ b равно 0, когда b равно 1, и 1, когда b равно 0. Кроме того, выражение 0 ^ b дает b независимо от значения b. Следовательно, в результате объединения значения с маской с использованием операции ^ биты, соответствующие 1 в маске, переключаются, а биты, соответствующие 0 в маске, останутся неизмененными. Чтобы переключить бит 1 переменной flags, можно выполнить одно из следующих действий:
flags ^= MASK;
Например, пусть flags равно 00001111 и MASK — 10110110. (00001111) ^ (10110110) = (10111001).

Случай применения: проверка значения бита

Вы уже видели, как изменять значения битов. Предположим, что вместо этого нужно проверить значение какого-нибудь бита. Например, установлен ли в 1 бит 1 в flags?

Чтобы выполнить сравнение только бита 1 в flags с MASK, необходимо сначала замаскировать остальные биты flags:

```
if ((flags & MASK) == MASK)
    puts ("Совпадает!");
```

Побитовые операции имеют приоритет ниже, чем у операции ==, поэтому выражение flags & MASK должно быть заключено в скобки.

Во избежание неполного охвата информации, битовая маска должна иметь ширину не меньше, чем у маскируемого значения.

Побитовые операции сдвига: Сдвиг влево: <<

Операция сдвига влево (<<) сдвигает биты значения левого операнда влево на количество позиций, заданное правым операндом. Освобождаемые позиции заполняются 0, а биты, выходящие за пределы значения левого операнда, теряются. В следующем примере каждый бит сдвигается на две позиции влево:
(10001010) << 2 // выражение
(00101000) // результат
Эта операция выдаст новое битовое значение, но не изменяет операнды. Для примера предположим, что переменная stonk имеет значение 1. Выражение stonk<<2 дает 4, но значением stonk по-прежнему является 1. Чтобы изменить значение переменной, можно воспользоваться операцией сдвига влево с присваиванием (<<=). Эта операция сдвигает биты переменной влево на количество позиций, указанное в правом операнде. Вот пример:

```
int stonk = 1;
int onkoo;
onkoo = stonk << 2; /* присваивает 4 переменной onkoo */
stonk <<= 2; /* изменяет значение stonk на 4 */
```

Побитовые операции сдвига: Сдвиг вправо :>>

Операция сдвига вправо (>>) сдвигает биты значения левого операнда вправо на количество позиций, указанное в правом операнде. Биты, которые выходят за правую границу левого операнда, теряются. Для типов без знака освобождаемые слева позиции заполняются 0. Для типов со знаком данных результат зависит от системы. Освобождаемые позиции могут заполняться 0 либо битом знака:

(10001010) >> 2 // выражение, значение со знаком
(00100010) // результат в одних системах
(10001010) >> 2 // выражение, значение со знаком
(11100010) // результат в других системах

Для значения без знака результат будет следующим:

(10001010) >> 2 // выражение, значение без знака
(00100010) // результат во всех системах

Операция сдвига вправо с присваиванием (>>=) сдвигает вправо биты левого операнда на заданное в правом операнде количество позиций, например:

```
 int sweet = 16;
int ooosw;
ooosw = sweet >> 3; /* ooosw равно 2, sweet по-прежнему 16 */
sweet >>=3;  /* значение sweet изменилось на 2 */
```

Случай применения: побитовые операции сдвига

Побитовые операции сдвига могут служить удобными эффективным (в зависимости от оборудования) 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, если значение number неотрицательно
Эти операции сдвига аналогичны смещению десятичной точки при умножении или делении на 10.

Пример для цветов:

```
#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;
```

В коде посредством операции сдвига вправо 8-битовое значение составляющей цвета перемещается в младший байт. Затем с помощью приема с маской значение младшего байта присваивается желаемой переменной.

Демо битовые операции 1/2

```
char *itobs(int, char *)
void show_bstr(const char *);
int main(void)
{
    char bin_str[CHAR_BIT * sizeof(int) + 1];
    int number;
    puts("Вводите целые числа и просматривайте их двоичные
представления.");
    puts("Нечисловой ввод завершает программу.");
    while (scanf("%d", &number) == 1)
    {
        itobs(number, bin_str);
        printf("%d представляется как ", number);
        show_bstr(bin_str);
        putchar('\n');
    }
    puts("Программа завершена.");
    return 0;
}
```

Демо битовые операции 2/2

```c
#include <stdio.h>
#include <limits.h> // для CHAR_BIT количество битов на символ
char *itobs(int n, char *ps)
{
    int i;
    const static int size = CHAR_BIT * sizeof(int);
    for (i = size - 1; i >= 0; i--, n >>= 1)
        ps[i] = (01 & n) + '0'; // предполагается кодировка ASCII
    ps[size] = '\0';
    return ps;
}
void show_bstr(const char *str)
{ /* отображение двоичной строки блоками по 4 */
    int i = 0;
    while (str[i]) /* пока не будет получен нулевой символ */
    {
        putchar(str[i]);
        if (++i % 4 == 0 && str[i])
            putchar(' ');
    }
}
```

Демо 2 битовые операции 1/2

```c
char *itobs(int, char *);
void show_bstr(const char *);
int invert_end(int num, int bits);
int main(void)
{
    char bin_str[CHAR_BIT * sizeof(int) + 1];
    int number;
    puts("Вводите целые числа и просматривайте их двоичные представления.");
    puts("Нечисловой ввод завершает программу.");
    while (scanf("%d", &number) == 1)
    {
        itobs(number, bin_str);
        printf("%d представляется как\n", number);
        show_bstr(bin_str);
        putchar('\n');
        number = invert_end(number, 4);
        printf("Инвертирование последних 4 битов дает\n");
        show_bstr(itobs(number, bin_str));
        putchar('\n');
    }
    puts("Программа завершена.");
    return 0;
}
```

Демо 2 битовые операции 2/2

```
int invert_end(int num, int bits)
{
    int mask = 0;
    int bitval = 1;
    while (bits-- > 0)
    {
        mask |= bitval;
        bitval <<= 1;
    }
    return num ^ mask;
}
Вводите целые числа и просматривайте их двоичные представления.
Нечисловой ввод завершает программу.
12531
12531 представляется как
0000 0000 0000 0000 0011 0000 1111 0011
Инвертирование последних 4 битов дает
0000 0000 0000 0000 0011 0000 1111 1100
```

Битовые поля

Второй метод манипулирования битами предусматривает ***использование битового поля***, которое представляет собой просто набор соседствующих битов внутри значения типа signed int или unsigned int. (Стандарты C99 и C11 дополнительно разрешают иметь битовые поля типа _Bool.) Битовое поле создается путем объявления структуры, в которой помечено каждое поле и определен его размер. Например, следующее объявление устанавливает четыре однобитовых поля:

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

Такое определение приводит к получению структуры prnt, содержащей четыре однобитовых поля. Теперь для присваивания значений отдельным полям можно применять обычную операцию членства в структуре:

```
prnt.itals = 0;
prnt.undln = 1 ;
```

Поскольку каждое из этих полей — это просто один бит, присваивать можно только значения 1 и 0.

Битовые поля

Структуры с битовыми полями служат удобным средством для отслеживания настроек. Многие настройки, такие как полужирное или пассивное начертание шрифта, сводятся к указанию одной из двух опций: “включено" или “отключено”, “да” или “нет”, “истинно” или “ложно”. Когда нужен одиночный бит, не имеет смысла применять целую переменную. Структура с битовыми полями позволяет хранить множество настроек в одной конструкции.

Временами настройка предусматривает более двух опций, поэтому для представления всех вариантов одного бита оказывается недостаточно. Это не проблема, т.к. размеры нолей не ограничены одни м битом. Структуру можно определить следующим образом:

```
struct {
    unsigned int codel : 2;
    unsigned int code2 : 2;
    unsigned int code3 : 8;
 } prcode;
```

Этот код создает два 2-битовых поля и одно 8-битовое. Теперь возможны следующие присваивания:

prcode.codel = 0; prcode.code2 = 3; prcode.code3 = 102;

Битовые поля

А что, если общее количество объявленных битов превысит размер типа unsigned int? Тогда будет использоваться следующая область для хранения unsigned int. Отдельное поле не должно перекрывать границу между двумя смежными областями unsigned int. Компилятор автоматически сдвигает такое перекрывающее определение поля, чтобы выровнять его по границе unsigned int. Когда это происходит, в первой области unsigned int остается неименованный промежуток.

Структуру полей можно заполнить неименованным и промежутками с применением ширин не именованных полей. Использование не именованного поля шириной 0 приводит к тому, что следующее поле выравнивается по следующей области целочисленного значения:

```
struct {
    unsigned int field1 : 1;
    unsigned int : 0;
    unsigned int field2 : 1;
    unsigned int : 2;
    unsigned int field3 : 1;
} stuff;
```

Здесь между полями stuff.field1 и stuff.field2 имеется 2-битовый промежуток, а поле stuff.field3 хранится в следующей области int.

Пример с битовыми полями

Предположим, что вы решили представить свойства выводимого на экран окна. Окно обладает только перечисленным и ниже свойствами.

- Окно может быть прозрачным или непрозрачным.
- Цвет фона выбирается из следующей палитры: черный, красный, зеленый, желтый, синий, пурпурный, голубой и белый.
- Рамка может быть скрыта или отображена.
- Цвет рамки выбирается из той же палитры, что и цвет фона.
- Для рамки применяются три стиля линии: сплошная, пунктирная и штриховая.

```
struct box_props {
    bool opaque : 1;
    unsigned int fill_color : 3;
    unsigned int : 4;
    bool show_border : 1;
    unsigned int border_color : 3;
    unsigned int border_style : 2;
    unsigned int : 2;
};
```

Комбинация битов	Десятичный эквивалент	Цвет
000	0	Черный
001	1	Красный
010	2	Зеленый
011	3	Желтый
100	4	Синий
101	5	Пурпурный
110	6	Голубой
111	7	Белый

Демо с битовыми полями 1/3

```c
/* fields.c – определение и использование полей */
#include <stdio.h>
#include <stdbool.h> // C99, определение bool, true, false
/* стили линии */
#define SOLID 0
#define DOTTED 1
#define DASHED 2
/*основные цвета */
#define BLUE 4
#define GREEN 2
#define RED 1
/* смешанные цвета */
#define BLACK 0
#define YELLOW (RED | GREEN)
#define MAGENTA (RED | BLUE)
#define CYAN (GREEN | BLUE)
#define WHITE (RED | GREEN | BLUE)
const char *colors[8] = {"черный", "красный", "зеленый", "желтый",
                         "синий", "пурпурный", "голубой", "белый"};
```

Демо с битовыми полями 2/3

```c
struct box_props
{
    bool opaque : 1;
    unsigned int fill_color : 3;
    unsigned int : 4;
    bool show_border : 1;
    unsigned int border_color : 3;
    unsigned int border_style : 2;
    unsigned int : 2;
};
void show_settings(const struct box_props *pb);
int main(void)
{ /* создание и инициализация структуры box_props */
    struct box_props box = {true, YELLOW, true, GREEN, DASHED};
    printf("Исходные настройки окна:\n");
    show_settings(&box);
    box.opaque = false;
    box.fill_color = WHITE;
    box.border_color = MAGENTA;
    box.border_style = SOLID;
    printf("\nИзмененные настройки окна:\n");
    show_settings(&box);
    return 0;
}
```

Демо с битовыми полями 3/3

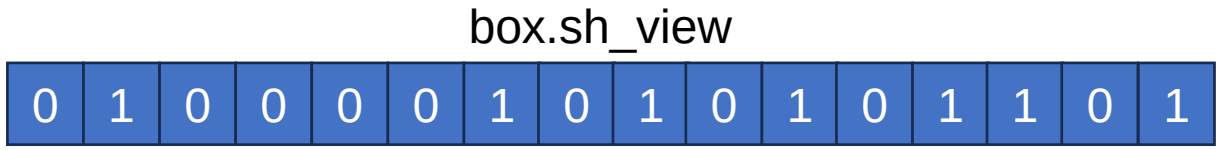
```
void show_settings(const struct box_props *pb)
{
    printf("Окно %s.\n", pb->opaque == true ? "непрозрачно" : "прозрачно");
    printf("Цвет фона %s.\n", colors[pb->fill_color]);
    printf("Рамка %s.\n", pb->show_border == true ? "отображается" : "не
отображается");
    printf("Цвет рамки %s.\n", colors[pb->border_color]);
    printf("Стиль рамки ");
    switch (pb->border_style)
    {
    case SOLID:
        printf("сплошной.\n");
        break;
    case DOTTED:
        printf("пунктирный.\n");
        break;
    case DASHED:
        printf("штриховой.\n");
        break;
    default:
        printf("неизвестного типа.\n");
    }
}
```

Битовые поля и побитовые операции

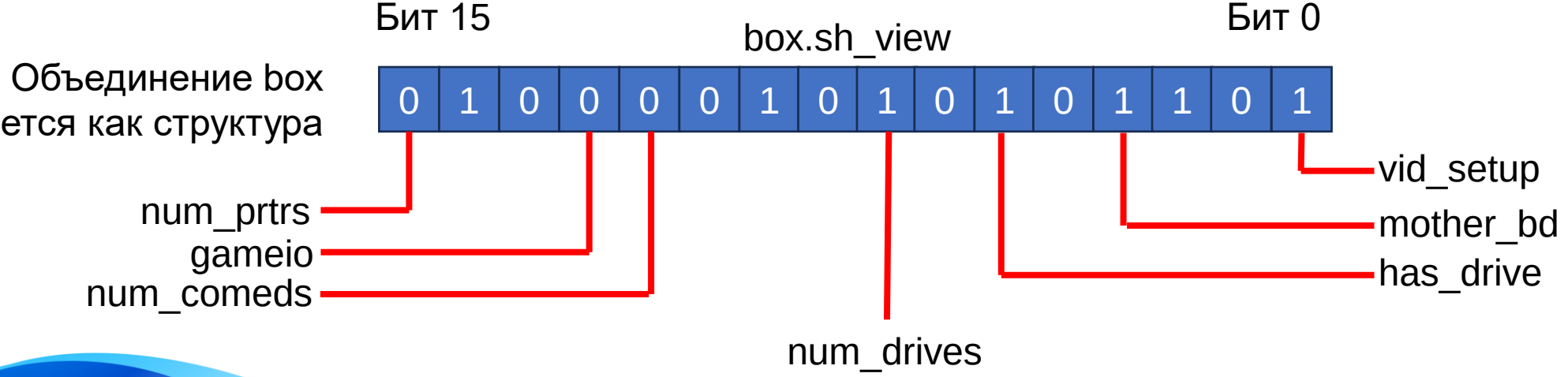
Битовые поля и побитовые операции — это два альтернативных подхода к решению задачи программирования одного и того же типа. Это значит, что часто можно применять любой из подходов. Рассмотрим пример, в котором задействованы оба подхода. В качестве средства комбинирования подхода на основе структуры и подхода на базе побитовых операций можно воспользоваться объединением. Исходя из существующего объявления типа struct box_props, можно объявить следующее объединение:

```
union Views { /* взгляд на данные как на struct или как на unsigned short */
    struct box_props st_view;
    unsigned short us_view;
};
```

Объединение box рассматривается как целое число



Объединение box рассматривается как структура



Демо с битовыми полями 1/6

```c
/* dualview.c -- битовые поля и побитовые операции */
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
/* КОНСТАНТЫ БИТОВЫХ ПОЛЕЙ */
/* стили линии */
#define SOLID 0
#define DOTTED 1
#define DASHED 2
/* основные цвета */
#define BLUE 4
#define GREEN 2
#define RED 1
/* смешанные цвета */
#define BLACK 0
#define YELLOW (RED | GREEN)
#define MAGENTA (RED | BLUE)
#define CYAN (GREEN | BLUE)
#define WHITE (RED | GREEN | BLUE)
```

Демо с битовыми полями 2/6

```
/* ПОБИТОВЫЕ КОНСТАНТЫ */
#define OPAQUE 0x1
#define FILL_BLUE 0x8
#define FILL_GREEN 0x4
#define FILL_RED 0x2
#define FILL_MASK 0xE
#define BORDER 0x100
#define BORDER_BLUE 0x800
#define BORDER_GREEN 0x400
#define BORDER_RED 0x200
#define BORDER_MASK 0xE00
#define B_SOLID 0
#define B_DOTTED 0x1000
#define B_DASHED 0x2000
#define STYLE_MASK 0x3000
const char *colors[8] = {"черный", "красный", "зеленый", "желтый",
                         "синий", "пурпурный", "голубой", "белый"};
```

Демо с битовыми полями 3/6

```
struct box_props
{
    bool opaque : 1;
    unsigned int fill_color : 3;
    unsigned int : 4;
    bool show_border : 1;
    unsigned int border_color : 3;
    unsigned int border_style : 2;
    unsigned int : 2;
};
union Views
{ /* взгляд на данные как на struct или как на unsigned short */
    struct box_props st_view;
    unsigned short us_view;
};
void show_settings(const struct box_props *pb);
void show_settings1(unsigned short);
char *itobs(int n, char *ps);
```

Демо с битовыми полями 4/6

```
int main(void)
{ /* создание объекта Views, инициализация представления в виде структуры */
    union Views box = {{true, YELLOW, true, GREEN, DASHED}};
    char bin_str[8 * sizeof(unsigned int) + 1];
    printf("Исходные настройки окна:\n");
    show_settings(&box.st_view);
    printf("\nНастройки окна с использованием представления unsigned short: \n");
    show_settings1(box.us_view);
    printf("комбинация битов %s\n", itobs(box.us_view, bin_str));
    box.us_view &= ~FILL_MASK;                /* очистить биты фона */
    box.us_view |= (FILL_BLUE | FILL_GREEN); /* переустановить фон */
    box.us_view ^= OPAQUE;                    /* переключить прозрачность */
    box.us_view |= BORDER_RED;                /* ошибочный подход */
    box.us_view &= ~STYLE_MASK;               /* очистить биты стиля */
    box.us_view |= B_DOTTED;                  /* установить пунктирный стиль */
    printf("\nИзмененные настройки окна: \n");
    show_settings(&box.st_view);
    printf("\nНастройки окна с использованием представления unsigned short: \n");
    show_settings1(box.us_view);
    printf("Комбинация битов %s\n", itobs(box.us_view, bin_str));
    return 0;
}
```

Демо с битовыми полями 5/6

```
void show_settings(const struct box_props *pb)
{
    printf("Окно %s.\n", pb->opaque == true ? "непрозрачно" : "прозрачно");
    printf("Цвет фона %s.\n", colors[pb->fill_color]);
    printf("Рамка %s.\n", pb->show_border == true ? "отображается" : "не
отображается");
    printf("Цвет рамки %s.\n", colors[pb->border_color]);
    printf("Стиль рамки ");
    switch (pb->border_style)
    {
    case SOLID:
        printf("сплошной.\n");
        break;
    case DOTTED:
        printf("пунктирный.\n");
        break;
    case DASHED:
        printf("штриховой.\n");
        break;
    default:
        printf("неизвестного типа.\n");
    }
}
```

Демо с битовыми полями 6/6

```
void show_settings1(unsigned short us)
{
    printf("Окно %s.\n", (us & OPAQUE) == OPAQUE ? "непрозрачно" : "прозрачно");
    printf("Цвет фона %s.\n", colors[(us >> 1) & 07]);
    printf("Рамка %s.\n", (us & BORDER) == BORDER ? "отображается" : "не
отображается");
    printf("Стиль рамки ");
    switch (us & STYLE_MASK)
    {
    case B_SOLID:
        printf("сплошной.\n");
        break;
    case B_DOTTED:
        printf("пунктирный.\n");
        break;
    case B_DASHED:
        printf("штриховой.\n");
        break;
    default:
        printf("неизвестного типа.\n");
    }
    printf("Цвет рамки %s.\n", colors[(us >> 9) & 07]);
}
```

Средства выравнивания (C11)

Средства выравнивания C11 по своей природе больше ориентированы на манипулирование байтами, чем битами, но они также отражают возможность языка C иметь дело с оборудованием. В этом контексте выравнивание относится к тому, как объекты располагаются в памяти. Например, 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. Большинству программистов редко когда придется заботиться о выравнивании. Но в некоторых ситуациях контроль над выравниванием позволяет извлечь выгоду, например, при передаче данных из одного физического места в другое либо при вызове инструкций, которые оперируют на множестве элементов данных одновременно.

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

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

Средства выравнивания (C11)

С помощью спецификатора _Alignas можно запрашивать конкретное выравнивание для переменной или типа. Однако вы не должны запрашивать выравнивание, которое слабее фундаментального выравнивания, принятого для типа. Например, если требование к выравниванию для float составляет 4, не запрашивайте значение выравнивания, равное 1 или 2. Этот спецификатор применяется как часть объявления, и за ним следует пара круглых скобок, содержащая либо значение выравнивания, либо тип:

```
_Alignas(double) char cl;
_Alignas(8) char c2;
unsigned char _Alignas(long double) c_arr[sizeof(long double)];
```

Средства выравнивания (C11)

```c
// align.c -- использование _Alignof и _Alignas (C11)
#include <stdio.h>
int main(void)
{
    double dx;
    char ca;
    char cx;
    double dz;
    char eb;
    char _Alignas(double) cz;
    printf("Выравнивание char: %zd\n", _Alignof(char));
    printf("Выравнивание double: %zd\n", _Alignof(double));
    printf("&dx: %p\n", &dx);
    printf("&ca: %p\n", &ca);
    printf("&cx: %p\n", &cx);
    printf("&dz: %p\n", &dz);
    printf("&cb: %p\n", &eb);
    printf("&cz: %p\n", &cz);
    return 0;
}
```

```
Выравнивание char: 1
Выравнивание double: 8
&dx: 0x7fff97789ba8
&ca: 0x7fff97789ba5
&cx: 0x7fff97789ba6
&dz: 0x7fff97789bb0
&cb: 0x7fff97789ba7
&cz: 0x7fff97789ba0
```

Big and little endian

Endianness – это способ, которым компьютеры организуют данные в памяти.

- big-endian начинается с самой большой цифры,
- little-endian начинается с самой маленькой цифры.

Это важно знать, чтобы данные правильно читались на разных устройствах.
Endianness решает проблему универсального понимания данных между различными компьютерными системами. Как если бы все в мире договорились, по каким правилам читать книгу: с начала или с конца. Это упрощает обмен данными и гарантирует, что информация интерпретируется правильно, независимо от того, на каком устройстве она открывается.
Понимание этого концепта упрощает написание программ, которые работают с данными на низком уровне, например, при общении с сетью или работе с файлами. Знание о endianness помогает избежать ошибок, когда данные выглядят "перевернутыми" после передачи между системами с разным порядком байтов.

Big and little endian

```c
#include <stdio.h>
int main()
{
    unsigned int number = 0x12345678;
    unsigned char *numPtr = (unsigned char *)&number;

    printf("Представление числа 0x12345678 в памяти:\n");
    for (int i = 0; i < sizeof(number); i++)
    {
        printf("Байт %d: 0x%X\n", i, numPtr[i]);
    }
    return 0;
}
```

Представление
числа 0x12345678
в памяти:
Байт 0: 0x78
Байт 1: 0x56
Байт 2: 0x34
Байт 3: 0x12

Big-endian и little-endian – это два противоположных способа хранения данных.
Big-endian хранит старший байт числа по меньшему адресу памяти, что интуитивно понятно при чтении слева направо. Например, число 0x12345678 будет храниться как 12 34 56 78.
В то время как little-endian делает обратное, помещая младший байт по меньшему адресу, так что то же число будет храниться как 78 56 34 12. Это различие может казаться незначительным, но оно имеет большое значение при обмене данными между разными системами.


А теперь усложним

Манипулирование битами

Для того чтобы обнулить крайний справа единичный бит, а если такового нет - вернуть 0 (например, 01011000 → 01010000), используется формула

x&(x-1)

Чтобы установить крайний справа нулевой бит равным 1, а если такового нет — установить все биты равными 1 (например, 10100111 → 10101111), можно воспользоваться формулой

x|(x+1)

Приведенная ниже формула позволяет превратить все завершающие значение единичные биты в нулевые, а если таковых нет - просто вернуть исходное значение (например, 10100111 → 10100000):

x&(x+1)

Чтобы превратить все завершающие значение нулевые биты в единичные, а если таковых нет, просто вернуть исходное значение (например, 10101000 → 10101111), можно выполнить следующую операцию:

x|(x-1)

Манипулирование битами

Приведенная далее формула позволяет создать слово с единственным битом 1 в позиции крайнего справа 0 в слове x, а если такового нет, то вернуть значение 0 (например. 10100111→00001000):

~x&(x+1)

Небольшое изменение превращает эту формулу в другую, которая создает слово с единственным битом 0 в позиции крайнего справа единичного бита в слове x, а если такового нет, то возвращающую слово со всеми битами, установленными равными 1 (например, 10101000→11110111):

~x&(x-1)

Для создания слова с единицами в позициях завершающих нулевых битов и с нулевыми битами во всех остальных местах можно воспользоваться одной из приведенных ниже формул (например, 01011000→00000111):

~x&(x-1) или ~(x|-x), или (x&-x)-1

Для создания слова с нулевыми битами в позициях завершающих единичных битов в x и единичными битами во всех остальных местах, или, если завершающих единичных битов нет, то слова, состоящего из единичных битов (например, 10100111 → 11111000):

~x|(x+1)

Манипулирование битами

Формула
x&(-x)
выделяет крайний справа единичный бит, возвращая 0, если такового нет (например. 01011000→00001000).
Формула
x^(x-1)
создает слово, в котором в позициях крайнего справа единичного бита и следующих за ним нулевых битов x стоят единичные биты; если единичного бита нет слово из единичных битов, и целое число, равное 1, если нет завершающих нулей (например, 01011000 → 00001111).
Используйте приведенную далее формулу для создания слова с единичными битами в позициях крайнего справа нулевого бита и следующих за ним единичных битов x; если нулевого бита нет слово из единичных битов, и целое число, равное 1, если нет завершающих единиц (например, 01010111 → 00001111):
x^(x+1)
Для того чтобы обнулить крайнюю справа непрерывную строку единиц (например, 01011100→01000000), можно воспользоваться одной из формул
(((x|(x-1))+1)&x) и ((x&-x)+x)&x

Демо

Вычисление следующего большего числа с тем же количеством единичных битов

```c
#include <stdio.h>
unsigned int snoob(unsigned int x)
{
    unsigned int smallest, ripple, ones; // x = xxx0 1111 0000
    smallest = x & -x;                   //     0000 0001 0000
    ripple = x + smallest;               //     xxx1 0000 0000
    ones = x ^ ripple;                   //     0001 1111 0000
    ones = (ones >> 2) / smallest;       //     0000 0000 0111
    return ripple | ones;                //     xxx1 0000 0111
}
int main()
{
    unsigned int x = 240;
    printf("Итог: %u\n", snoob(x));
    return 0;
}
// Итог: 263
```