

# DECIMAL

---

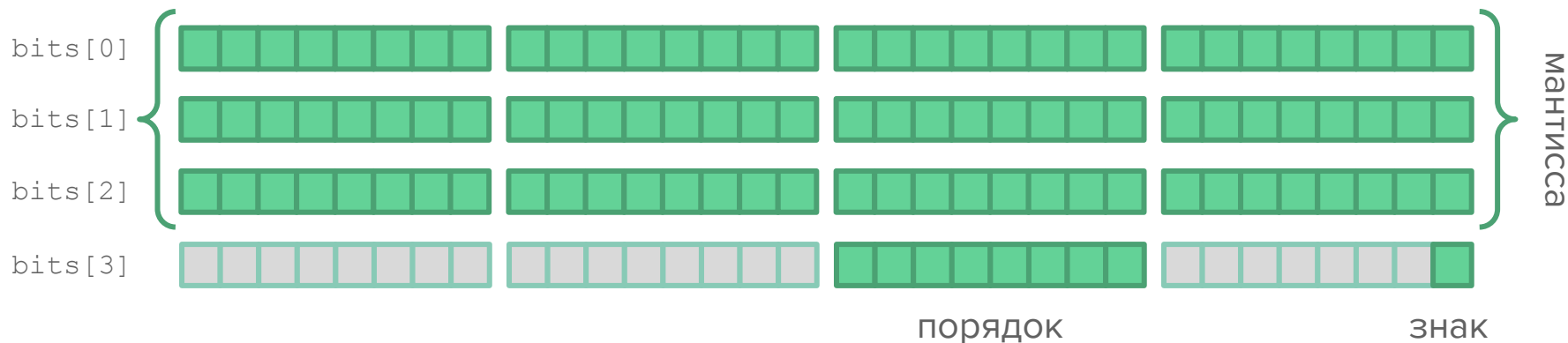
by AMMOSHRI

# СТРУКТУРА

---

# S21\_DECIMAL

$-79,228,162,514,264,337,593,543,950,335 \leq s21\_decimal \leq 79,228,162,514,264,337,593,543,950,335$



от *define type* - определить тип

```
typedef struct{  
    int bits[4];  
} s21_decimal;
```

Знак: 0 (для положительного), 1 (для отрицательного)  
Порядок: от 0 до 28 (степень 10, на которую делится целое число)  
Мантисса: от  $-2^{96}$  до  $2^{96}$

# ОСНОВНЫЕ ПРОБЛЕМЫ ТИПА DECIMAL

## СИСТЕМА СЧИСЛЕНИЯ

В мантиссе нужно использовать все доступные биты для достижения нужной вместимости (для этого нужно работать с числом в двоичном виде), а масштаб хранится в виде степени десяти.

## OVERFLOW

Нужно следить не только за вместимостью каждой отдельной ячейки памяти, но и s21\_decimal в целом. Это особенно актуально при умножении.

## ПРЕДСТАВЛЕНИЕ ЧИСЛА

Одно и тоже число может представляться несколькими разными способами. Например:

знак	порядок	мантисса
0	2	1200
0	1	120
0	0	12

# ПРЕОБРАЗОВАНИЕ ВО ВРЕМЕННЫЙ ТИП

## BIG DECIMAL

Тот же decimal, но для хранения используется `unsigned bits[8]`.

+

- Удобная конвертация
- Легко проверять overflow

-

- Работа в двоичной системе не интуитивна

## BINARY-CODED DECIMAL

Массив ячеек из 4 битов, каждая кодирует один разряд (единицы, десятки, ...)

+

- Почти что работа с обычными десятичными числами

-

- Алгоритм конвертации довольно сложный

# СЛУЖЕБНЫЕ ФУНКЦИИ

```
int is_zero (s21_decimal num); // ноль - любое число, где
    bits[0] = bits[1] = bits[2] = 0
void null_decimal (s21_decimal* num);

// привет от ООП. То же самое для s21_big_decimal
int get_sign (s21_decimal num);
void set_sign (s21_decimal num, int sign_value);
int get_scale (s21_decimal num);
void set_scale (s21_decimal* num, int scale_value);
int get_bit (s21_decimal num, int bit); // от 0 до 95!
void set_bit (s21_decimal* num, int bit, unsigned value);
```

# АРИФМЕТИКА

---

# ПОБИТОВЫЕ ОПЕРАЦИИ

& побитовое И	1010 & 0011	0010
побитовое ИЛИ	1010   0011	1011
<< сдвиг влево	01011	10110
>> сдвиг вправо	10111	01011
~ побитовое НЕ	10110	01001

- Сдвиг влево - умножение на 2
- Сдвиг вправо - деление на 2
- $\sim a$  ?  $\sim a$  (инверсия int)
- $\sim (1u \ll a)$  (маска из всех битов, кроме a)



# АРИФМЕТИКА → ПОБИТОВАЯ АРИФМЕТИКА

Вся арифметика делается на `s21_big_decimal`!

$$(+18.04) + (-20.04) = (+1804 / 10^2) + (-2004 / 10^2) = -(2004 - 1804) / 10^2 = -200 / 10^2$$

$$(-36.5) - (-0.75) = (-365 / 10^1) - (-75 / 10^2) = -(3650 + 75) / 10^2 = -3725 / 10^2$$

$$(-0.357) * (-45.1) = (-357 / 10^3) * (-451 / 10^1) = +(357 * 451) / (10^3 * 10^1) = +161007 / 10^4$$

$$(+58.63) / (-45.1) = (+5863 / 10^2) / (-451 / 10^1) = -(5863 / 451) / (10^2 / 10^1) = -13 / 10^1$$

Побитовое сложение:

- Порядок одинаковый
- Знак одинаковый

Побитовое вычитание:

- Порядок одинаковый
- Знак одинаковый
- Первое число больше второго

Побитовое деление:

- Второе число не 0

# ПОБИТОВОЕ СЛОЖЕНИЕ

```
void bitwise_addition (s21_big_decimal value_1, s21_big_decimal value_2,  
    s21_big_decimal *result) {  
    for (int i = 0; i < 32 * 7; ++i) {  
        unsigned result_bit = big_get_bit(value_1, i) +  
                                big_get_bit(value_2, i);  
  
        result_bit %= 2;  
        big_set_bit(result, i, result_bit);  
    }  
}
```

# ПОБИТОВОЕ СЛОЖЕНИЕ

```
void bitwise_addition (s21_big_decimal value_1, s21_big_decimal value_2,
                      s21_big_decimal *result) {
    unsigned memo = 0;
    for (int i = 0; i < 32 * 7; ++i) {
        unsigned result_bit = big_get_bit(value_1, i) + big_get_bit(value_2, i)
                               + memo;

        memo = result_bit / 2;
        result_bit %= 2;
        big_set_bit(result, i, result_bit);
    }
}
```

# ПОБИТОВОЕ СЛОЖЕНИЕ

```
void bitwise_addition (s21_big_decimal value_1, s21_big_decimal value_2,
                      s21_big_decimal *result) {
    unsigned memo = 0;
    // граница цикла через sizeof позволяет легко адаптироваться
    // к изменению типа
    for (int i = 0;
         i < (int)(sizeof(s21_big_decimal) / sizeof(unsigned) - 1) * 32; ++i){
        unsigned result_bit = big_get_bit(value_1, i) + big_get_bit(value_2, i)
                               + memo;

        memo = result_bit / 2;
        result_bit %= 2;
        big_set_bit(result, i, result_bit);
    }
} // Вычитание (bitwise_subtraction) делается аналогично
```

# ПОГОВОРИМ О БОГЕ НАШЕМ SHIFT

```
void shift_left (s21_big_decimal* num) {  
    unsigned temp = num->bits[0];  
    num->bits[0] <<= 1;  
    num->bits[1] |= temp >> (32 - 1);  
}
```

# ПОГОВОРИМ О БОГЕ НАШЕМ SHIFT

```
void shift_left (s21_big_decimal* num) {  
    unsigned memory = 0;  
    unsigned temp = num->bits[0];  
    num->bits[0] <=< 1;  
    memory = temp >> (32 - 1);  
  
    temp = num->bits[1];  
    num->bits[1] <=< 1;  
    num->bits[1] |= memory;  
    memory = temp >> (32 - 1);  
  
    temp = num->bits[2];  
    num->bits[2] <=< 1;  
    num->bits[2] |= memory;  
    memory = temp >> (32 - 1);  
    // и так ещё 4 раза...  
}
```

# ПОГОВОРИМ О БОГЕ НАШЕМ SHIFT

```
void shift_left (s21_big_decimal* num) {  
    unsigned memory = 0;  
    for (int i = 0; i < 7; ++i) {  
        unsigned temp = num->bits[i];  
        num->bits[i] <= 1;  
        num->bits[i] |= memory;  
        memory = temp >> (32 - 1);  
    }  
}
```

# ПОГОВОРИМ О БОГЕ НАШЕМ SHIFT

```
void shift_left (s21_big_decimal* num) {
    unsigned memory = 0;
    for (int i = 0;
        i < (int) (sizeof(s21_big_decimal) / sizeof(unsigned) - 1); ++i) {
        unsigned temp = num->bits[i];
        num->bits[i] <<= 1;
        num->bits[i] |= memory;
        memory = temp >> (32 - 1);
    }
}
```



# ПОГОВОРИМ О БОГЕ НАШЕМ SHIFT

```
void shift_left (s21_big_decimal* num, int shift_value) {
    unsigned memory = 0;
    for (int i = 0;
        i < (int)(sizeof(s21_big_decimal) / sizeof(unsigned) - 1); ++i){
        unsigned temp = num->bits[i];
        num->bits[i] <<= shift_value;
        num->bits[i] |= memory;
        memory = temp >> (32 - shift_value);
    }
}
```

# НОРМАЛИЗАЦИЯ

scale	mantissa
101	101101
10	10111110110
1000	11

## УМНОЖЕНИЕ НА 10

$$a * 10 = a * 8 + a * 2$$

$$a * 10 = a \ll 3 + a \ll 1$$

Для сравнения чисел, а также их сложения и вычитания, должен быть **одинаковый порядок.**

Это достигается за счёт умножения чисел на 10, пока порядок не станет равным.

# ПОБИТОВОЕ УМНОЖЕНИЕ

```
value_1 = 0b101101  // си понимает такую запись
value_2 = 0b1010
```

	ten	operations	result
101101			
* 1010			0
-----	1010	+ 101101	101101
000000		<< 1	1011010
101101 (value_1 << 1)	1010	+ 0	10110100
+ 000000		<< 1	11100001
101101 (value_1 << 3)	1010	+ 101101	111000010
-----		<< 1	111000010
111000010	1010	+ 0	111000010

# ПОБИТОВОЕ ДЕЛЕНИЕ

operat.	remainder	result	sc.
	100010	0	0
+1, << 1	100010 - (1000<<2)	10	0
+0, << 1	<i>10 &lt; (1000&lt;&lt;1)</i>	100	0
+0	<u>10</u> < 1000	100	0
	10*1010	0	1
+1, << 1	10100 - (1000<<1)	10	1
+0	<u>100</u> < 1000	10	1
	100*1010	0	2
+1, << 1	101000 - (1000<<2)	10	2
+0, << 1	<i>1000 &lt; (1000&lt;&lt;1)</i>	100	2
+1	1000 - 1000	101	2

$$\begin{array}{r}
 100010 \quad | 1000 \\
 \underline{-1000} \quad | 100 \\
 10 * 1010
 \end{array}$$

$$\begin{array}{r}
 10100 \quad | 1000 \\
 \underline{-1000} \quad | 10 \\
 100
 \end{array}$$

$$\begin{array}{r}
 101000 \quad | 1000 \\
 \underline{-1000} \quad | 101 \\
 1000 \\
 \underline{-1000} \\
 0
 \end{array}$$

$$\begin{aligned}
 \text{scale} &= 2 \\
 \text{mantissa} &= 100 * (1010^2) + \\
 &+ 10 * 1010 + 101
 \end{aligned}$$

# КОНВЕРТЕРЫ

---

# КОНВЕРТЕРЫ

```
// это делается на раз-два, вы разберетесь...
int s21_from_int_to_decimal(int src, s21_decimal *dst);
// если в decimal есть дробная часть, отбросьте ее
int s21_from_decimal_to_int(s21_decimal src, int *dst);
// сначала копируете целую часть, затем делите на 10 пока scale > 0
int s21_from_decimal_to_float(s21_decimal src, float *dst);
// ну что, как дела у вашего s21_string+?
int s21_from_float_to_decimal(float src, s21_decimal *dst);

#define OK 0
#define CONVERTATION_ERROR 1

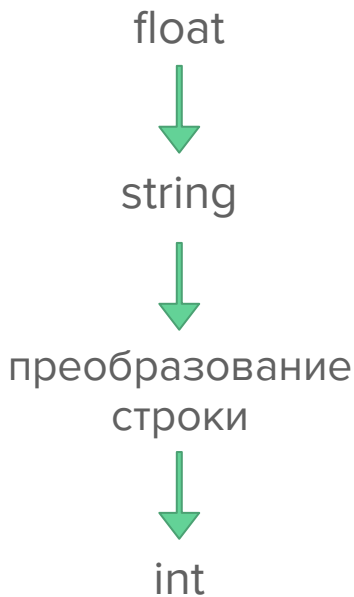
typedef enum returnValues {
    OK = 0,
    CONVERTATION_ERROR = 1,
    CALCULATION_ERROR = 1,
} returnValues;
```

# FROM FLOAT TO DECIMAL

**ПРОБЛЕМА:** представление float и decimal в битовом виде *очень разное*

*“Если [при обработке числа с типом float] [значимых десятичных] цифр больше 7, то значение числа округляется к ближайшему, у которого не больше 7 значимых цифр.”*

— README\_RUS.MD



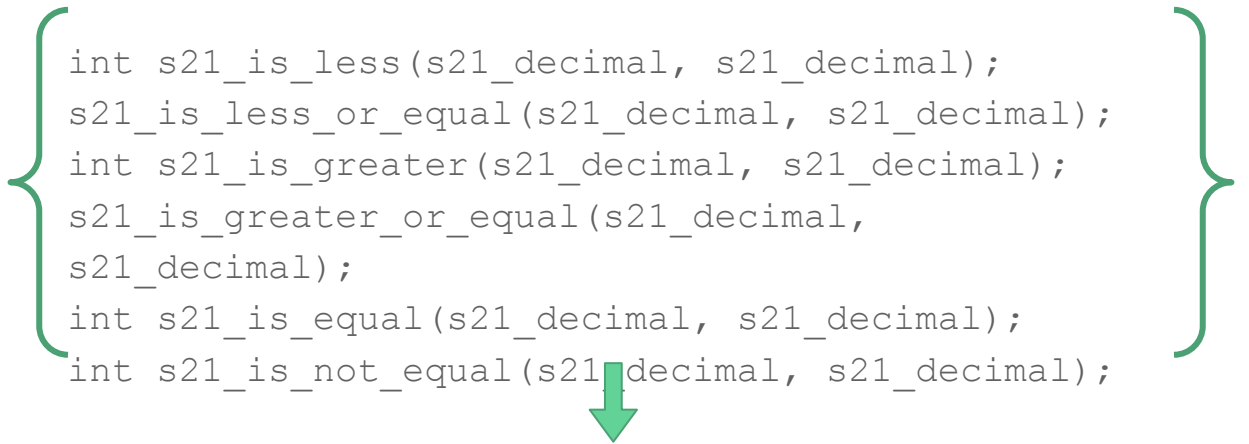
Спецификатор  
форматирования E  
(научная нотация):  
34.543327 → 3.454333E+01

# СРАВНЕНИЕ И ОКРУГЛЕНИЕ

---



# ОПЕРАТОРЫ СРАВНЕНИЯ



```
int s21_is_less(s21_decimal, s21_decimal);  
s21_is_less_or_equal(s21_decimal, s21_decimal);  
int s21_is_greater(s21_decimal, s21_decimal);  
s21_is_greater_or_equal(s21_decimal,  
s21_decimal);  
int s21_is_equal(s21_decimal, s21_decimal);  
int s21_is_not_equal(s21_decimal, s21_decimal);
```

```
int compare(s21_decimal a, s21_decimal b);  
// возвращает: 1 при a>b, 0 при a==b, -1 при a<b  
// побитовое сравнение от старшего бита к младшему  
// ака тысячи, сотни, десятки, единицы  
// помните про знаки и нормализацию
```

# ОКРУГЛЕНИЕ

```
tr_num = truncate(number)
fractional_part = number - tr_num
```

`fractional_part < 0.5`

`fractional_part == 0.5`

`fractional_part > 0.5`

```
res = tr_num
```

```
/* в отличие от обычного
   округления среднее
   арифметическое ошибок
   стремится к нулю */
```

```
// banker's rounding
if (tr_num % 2 == 0)
    res = tr_num
else
    res = tr_num + 1
```

```
res = tr_num + 1
```

# НАПИСАНИЕ ТЕСТОВ

---

# ШАБЛОН (C# НАМ В ПОМОЩЬ)

```
using System;

public class TestGen
{
    protected static void ConsoleWriteVar(decimal variable, string nameVar)
    {
        var tmp = decimal.GetBits(variable);

        Console.WriteLine("s21_decimal {4} = {{ {0}, {1}, {2}, {3} }};", tmp[0],
            tmp[1], tmp[2], tmp[3], nameVar);
    }
    public static void Main(string[] args)
    {
        decimal num_1 = -1034953869456;
        ConsoleWriteVar(num_1, "num_1");
        decimal num_2 = 333425;
        ConsoleWriteVar(num_2, "num_2");
        decimal orig_res = num_1 * num_2;
        ConsoleWriteVar(orig_res, "orig_res");
        ConsoleWriteVar(0, "s21_res");
        Console.WriteLine("int return_s21 = s21_mul(num_1, num_2, &s21_res);");
        for (int i = 0; i < 4; ++i) {
            Console.WriteLine("ck_assert_int_eq(s21_res.bits[{0}], orig_res.bits[{0}]);", i);
        }
        Console.WriteLine("ck_assert_int_eq(return_s21, 0);");
    }
}
```

СПАСИБО ЗА ВНИМАНИЕ!

---

# ПРИЛОЖЕНИЯ

---

# DOUBLE DABBLE и BCD

<https://www.youtube.com/watch?v=eXIfZ1yKFIA>

[https://en.wikipedia.org/wiki/Double\\_dabble](https://en.wikipedia.org/wiki/Double_dabble)

```
typedef struct {  
    unsigned value : 4;  
} nibble_t;
```

```
typedef struct{  
    unsigned sign : 1;  
    unsigned scale : 8;  
    nibble_t nibble[60];  
} bcd_t;
```

# КОММЕНТАРИИ К ФУНКЦИЯМ

```
/*  
    Сумма двух decimal  
    @param value_1 первое число  
    @param value_2 второе число  
    @param *result сумма чисел  
  
    @return 0 - OK \  
    @return 1 - число слишком велико или равно бесконечности \  
    @return 2 - число слишком мало или равно отрицательной бесконечности \  
    @return 3 - деление на 0  
*/  
int s21_add(s21_decimal value_1, s21_decimal value_2, s21_decimal *result);
```

```
/**  
 * Сумма двух decimal  
 *  
 * Parameters:  
 * value_1 - первое число  
 * value_2 - второе число  
 * *result - сумма чисел  
 *  
 * Returns:  
 * 0 - OK  
 * 1 - число слишком велико или равно бесконечности  
 * 2 - число слишком мало или равно отрицательной бесконечности  
 * 3 - деление на 0  
 */  
Code1  
int s21_add(s21_decimal value_1, s21_decimal value_2, s21_decimal *result);
```