



# Ветвления и побитовые операции

Программирование на языке Си  
(базовый уровень)



# Оглавление

<b>Введение</b>	<b>3</b>
Термины, используемые в лекции	3
<b>Типы фиксированного размера</b>	<b>4</b>
<b>Спецификаторы фиксированной длины для функций printf, scanf</b>	<b>5</b>
<b>Представление целых знаковых чисел</b>	<b>6</b>
<b>Приведение типов</b>	<b>8</b>
Явное приведение типов	8
Неявное приведение типов	9
<b>Операции сравнения</b>	<b>11</b>
<b>Логические операции</b>	<b>11</b>
<b>Условный оператор</b>	<b>13</b>
Тернарные операции $a ? b : c$ ;	14
<b>Побитовые операции</b>	<b>15</b>
<b>Операция последовательного вычисления</b>	<b>16</b>
<b>Оператор присваивания и побочный эффект</b>	<b>16</b>
<b>Точка следования</b>	<b>17</b>
<b>Старшинство операций</b>	<b>18</b>
<b>Форматирование кода</b>	<b>21</b>
Структура программы	22
<b>Вычисление корней квадратного уравнения (продолжение)</b>	<b>23</b>
<b>Подведение итогов</b>	<b>25</b>
<b>Домашнее задание</b>	<b>26</b>
<b>Используемые источники</b>	<b>26</b>

# Введение

На предыдущей лекции вы узнали:

- Какие типы данных бывают в языке Си
- Спецификаторы типов (signed, unsigned, short и long)
- Про объявление переменных и их имена
- Спецификаторы классов памяти и квалификатор const
- Что такое константы и литералы
- Про вычисление выражений
- Укороченное присваивание и унарные операции
- Форматный ввод и вывод

На этой лекции вы найдете ответы на такие вопросы как:

- Спецификаторы фиксированной длины в C99
- О представлении целых и знаковых чисел
- Узнаем про явное и неявное приведение типов
- Логические операции
- Условный оператор
- Тернарные операции  $a ? b : c$ ;
- Побитовые операции
- Приоритет выполнения операций
- Что такое оператор присваивания его побочный эффект и точка следования
- Форматирование кода

## Термины, используемые в лекции

**Побитовая операция** – операция над двоичным представлением числа.

**XOR** – побитовая операция исключающее *ИЛИ*.

**Конъюнкция** – логическое *И*.

**Дизъюнкция** – логическое *ИЛИ*.

**Побочный эффект** – это изменение объекта памяти.

## Типы фиксированного размера

Стандарт C99 включает определения нескольких новых целочисленных типов для повышения переносимости программ. Уже доступных базовых целочисленных типов было сочтено недостаточным, поскольку их фактические размеры определяются реализацией и могут различаться в разных системах. Новые типы особенно полезны во встроенных средах (IoT), где оборудование обычно поддерживает только несколько типов и эта поддержка варьируется в зависимости от среды.



**Внимание!** В среде разработки должна быть включена поддержка стандарта C99. См. [Инструкция по установке Geany и компилятора TDM-gcc для Windows](#)

Все новые типы определены в заголовке `<inttypes.h>` (`printf`, `scanf`), а также доступны в заголовке `<stdint.h>`. Целочисленные типы фиксированной ширины, которые гарантированно имеют одинаковое количество бит  $n$  во всех реализациях.

В следующей таблице приведены типы и интерфейс для получения сведений о реализации ( $n$  означает количество битов):

Type category	Signed types			Unsigned types		
	Type	Minimum value	Maximum value	Type	Minimum value	Maximum value
Exact width	<code>intn_t</code>	<code>INTn_MIN</code>	<code>INTn_MAX</code>	<code>uintn_t</code>	0	<code>UINTn_MAX</code>
	<code>int8_t</code>	<code>INT8_MIN</code>	<code>INT8_MAX</code>	<code>uint8_t</code>	0	<code>UINT8_MAX</code>
	<code>int16_t</code>	<code>INT16_MIN</code>	<code>INT16_MAX</code>	<code>uint16_t</code>	0	<code>UINT16_MAX</code>
	<code>int32_t</code>	<code>INT32_MIN</code>	<code>INT32_MAX</code>	<code>uint32_t</code>	0	<code>UINT32_MAX</code>

	int64_t	INT64_MIN	INT64_MAX	uint64_t	0	UINT64_MAX
<b>Pointer</b>	intptr_t	INTPTR_MIN	INTPTR_MAX	uintptr_t	0	UINTPTR_MAX
<b>Maximum width</b>	intmax_t	INTMAX_MIN	INTMAX_MAX	uintmax_t	0	UINTMAX_MAX



Данная таблица позволяет очень просто задавать типы данных нужной длины и знаковости, не прибегая к конструкциям *long*, *short*, *unsigned* и т.д.



Сравните с таблицей типов в предыдущей лекции.

## Спецификаторы фиксированной длины для функций printf, scanf

В C99 файл заголовка `<inttypes.h>` включает ряд макросов для использования в платформенно-независимом кодировании `printf`, `scanf`. Они должны быть вне двойных кавычек, например:

```
int32_t a;
scanf("%" PRId32, &a);
printf("a = %" PRId32 " hex a = %" PRIx32 "\n", a, a);
```

### Примеры макросов:

Macro	Description	32-bit platforms	64-bit platforms
PRIId32	Typically equivalent to <i>I32d</i> (Win32/Win64)	<i>d</i>	<i>d</i>
PRIId64	Typically equivalent to <i>I64d</i> (Win32/Win64)	<i>lld</i>	<i>ld</i>
PRId32	Typically equivalent to <i>I32i</i> (Win32/Win64)	<i>i</i>	<i>i</i>
PRId64	Typically equivalent to <i>I64i</i> (Win32/Win64)	<i>lli</i>	<i>li</i>
PRId32	Typically equivalent to <i>I32u</i> (Win32/Win64)	<i>u</i>	<i>u</i>
PRId64	Typically equivalent to <i>I64u</i> (Win32/Win64)	<i>llu</i>	<i>lu</i>
PRIx32	Typically equivalent to <i>I32x</i> (Win32/Win64)	<i>x</i>	<i>x</i>
PRIx64	Typically equivalent to <i>I64x</i> (Win32/Win64)	<i>llx</i>	<i>lx</i>

# Представление целых знаковых чисел

Все знаковые отрицательные числа хранятся в дополнительном коде. Положительные числа хранятся в прямом коде. Представление отрицательных чисел таким образом, дает возможность проводить арифметические операции сложения и вычитания абсолютно одинаково вне зависимости от знака числа.

**Способ 1.** Вычисление по формуле

$$\langle \text{Дополнительный код} \rangle (X) = 2^k - |X|,$$

где  $k$  число разрядов выделенное под хранение числа  $X$ .

**Способ 2.** Сделать побитовую инверсию всех разрядов и прибавить 1.

Причем самый старший разряд определяет знак числа (0-это плюс, 1-это минус), а оставшиеся младшие разряды выделены под хранение дополнительного кода числа.



...  
+2 = 00002  
+1 = 00001  
0 = 00000  
-1 = 0 - 1 = 99999  
-2 = 0 - 2 = 99998  
-3 = 0 - 3 = 99997  
...

Рассмотрим представление отрицательного числа -5 в дополнительном коде, на примере ячейки размером 1 байт (8 бит).

Общая формула.

$2^8 - |-5| = 256 - 5 = 251$  данное число необходимо перевести в двоичный вид и полученный результат будет размещен в ячейке размером 1 байт.

$$-5_{10} = 251 = FB_{16}$$

Значение	1	1	1	1	1	0	1	1
Разряд	знак	6	5	4	3	2	1	0

Разряды идут от старшего к младшему

Как нетрудно заметить диапазон знаковых чисел будет равен -128 до 127

$$-128 = 1000\ 0000$$

$$127 = 0111\ 1111$$

### Задача.

Представить в дополнительном коде для 8-битного целого числа 0, 1, 127, -1, -4, -11, -127, -128 используя как двоичную так и шестнадцатеричную систему счисления.



**Решение.** Ответы вы найдете в таблице ниже.

Десятичное представление	Двоичное представление (8 бит), код:		
	прямой	обратный	дополнительный
127	0111 1111	0111 1111	0111 1111
1	0000 0001	0000 0001	0000 0001
0	0000 0000	0000 0000	0000 0000
-0	1000 0000	1111 1111	
-1	1000 0001	1111 1110	1111 1111
-2	1000 0010	1111 1101	1111 1110
-3	1000 0011	1111 1100	1111 1101
-4	1000 0100	1111 1011	1111 1100
-5	1000 0101	1111 1010	1111 1011
-6	1000 0110	1111 1001	1111 1010
-7	1000 0111	1111 1000	1111 1001
-8	1000 1000	1111 0111	1111 1000
-9	1000 1001	1111 0110	1111 0111
-10	1000 1010	1111 0101	1111 0110
-11	1000 1011	1111 0100	1111 0101
-127	1111 1111	1000 0000	1000 0001
-128	---	---	1000 0000

# Приведение типов

**Приведение** (преобразование) типа — преобразование значения одного типа в значение другого типа.


Выделяют два способа приведения типов: явные (*explicit*) и неявные (*implicit*).


## Явное приведение типов

Явное приведение задается программистом в тексте программы. При явном приведении перед выражением следует указать в круглых скобках имя типа, к которому необходимо преобразовать исходное значение.

```
int main()
{
    int X;
    int Y = 200;
    char C = 255;    //Чтобы избежать неоднозначностей, рекомендуется явно
                    //указывать знаковость для типа char -1 или 255?
    X = C * 10 + Y; // переменная C приводится к беззнаковому типу char
}
```

Чему будет равен X?

 **X = -1 \* 10 + 200 = 190** *char* по умолчанию знаковый и значение 255 в дополнительном коде трактуется как -1.

 **Внимание!** Выполняйте арифметические операции либо только над знаковыми типами, либо только над беззнаковыми.

```
int main()
{
    int X;
    int Y = 200;
    unsigned char C = 255; //указываем беззнаковый тип
    X = (unsigned char) C * 10 + Y; // переменная C явно приводится к
    беззнаковому типу unsigned char
}
```

Чему теперь будет равен X?





$X = (\text{unsigned char}) C * 10 + Y;$  // переменная *C* приводится к беззнаковому типу *char*,



$X = 255 * 10 + 200 = 2750;$

## Неявное приведение типов

Неявное приведение выполняется компилятором по правилам, описанным в стандарте языка. Рассмотрим эти правила.

Для двухместных арифметических операций приведение типов выполняется по следующим правилам:

1. Если **один** из типов операндов – **вещественный**, то **второй** операнд тоже приводится **к вещественному типу**. Общим типом будет наибольший вещественный тип из типов операндов (т.е. для *float* и *double* – *double*, для *double* и *long double* – *long double* и т.п.).

```
float f;
f = 3 / 2; //целочисленное деление f = 1.0
f = 3./ 2; // вещественное деление f = 1.5
```

2. Если есть операнд целого типа короче, чем *int* (*short* или *char*) и все значения этого типа могут быть представлены как *int*, то он преобразуется к *int*; иначе - к *unsigned int*. Это преобразование называется «целочисленное расширение» (**integer promotion**) и выполняется для избежания потерь точности при вычислениях.

```
short i, x = 5, y = -2;
i = x + y; //переменные x и y сначала расширяются до типа int,
           // выполняется сложение а затем результат помещается в i
```

3. Для операндов целых типов неявное приведение типов управляется целочисленным **рангом** (количество разрядов) приведения типа.

Ранг выбирается таким образом: ранг типа *long long* (64 бита) больше ранга *long* (32 бита), который в свою очередь больше ранга *int* (32 или 16 бит) и т.д. до *char* (8 бит).

Если ранг знакового и беззнакового варианта одного целого типа совпадают, то выбирается беззнаковый тип.

```
unsigned int u = 50;
int i = -500;
int answer = i / u; // чему равен answer?
```

Чему теперь будет равен answer?



answer = 85899335;



Переменная *i* будет приведен к беззнаковому типу и его значение будет равно  $2^{32}-500$  вместо -500, а результат деления будет приведен обратно к знаковому типу *int*



**Внимание!** Выполняйте арифметические операции либо только над знаковыми типами, либо только над беззнаковыми.

Здесь нужно явное приведение типов

```
unsigned int u = 50;
int i = -500;
int answer = i / (int)u; // чему равен answer?
```

**Задача.** Найдите значения *x*

```
int a = 7;
float x;
x = a / 4; // ?
x = 4 / a; // ?
x = float(a) / 4; // ?
x = 1.*a / 4; // ?
```




```
int a = 7;
float x;
x = a / 4; // = 1 - целочисленное деление
x = 4 / a; // = 0 - целочисленное деление
x = float(a) / 4; // = 1.75 - явное приведение типов
x = 1.*a / 4; // = 1.75 - неявное приведение типов (см. Правило 1).
```


## Операции сравнения


Операции сравнения генерируют результат типа *int*: единицу (True), если отношение истинно, и 0(False) в противном случае:


- равно "=", не равно "!="
- больше ">", меньше "<"
- больше или равно ">=", меньше или равно "<=")

```
int a, b;
a = 5 == 3; // a = 0
b = 100 >= 1; // b = 1
```

 **Внимание!** Обратите внимание на различия операций сравнения "**==**", и присваивания "**=**". Это **совершенно** разные операции.

 `if (a = 5)` – всегда будет выполняться, т.к. здесь присваивается переменной *a* значение 5. **Типичная ошибка!**

 `if (a == 5)` – будет выполняться только если *a* имеет значение 5.


 Операции больше/меньше или равно так и записывается. Сначала знак больше (**>=**)/меньше (**<=**) потом знак равно. Наоборот - не правильно!

## Логические операции

Логическими операциями в языке Си являются:

- !** – отрицание
- &&** – конъюнкция
- ||** – дизъюнкция

Результатом этих операций является **единица** (Истинна или *True*), если соответствующая логическая функция истинна, и **ноль** (Ложь или *False*), если ложна.

 При вычислении операндов целочисленных, вещественных или указателей истинным считается любое ненулевое значение, а ложью – лишь ноль.

```
int a = 5, n = 0;
if (a) // Это TRUE
    printf("It's true");
else
    printf("It's false");
```

Благодаря тому, что после вычисления первого операнда операций **&&** и **||** находится точка последовательных вычислений, становится возможным

использования «короткой» логики: операнды вычисляются слева направо, и если результат операции уже известен, вычисление второго операнда не производится.

```
int a, b = 5, n = 0;  
a = ( n != 0 && b / n ); // Нет ошибки - деления на ноль
```

**Задачи.** Написать эквивалентное выражение, не содержащее операции ! – отрицание

1.  $!(a > b)$
2.  $!(2 * a == b + 4)$
3.  $!(a < b \ \&\& \ c < d)$
4.  $!(a < 2 \ || \ a > 5)$
5.  $!(a < 1 \ || \ b < 2 \ \&\& \ c < 3)$



**Решение**

1.  $a \leq b$
2.  $2 * a \neq b + 4$
3.  $a \geq b \ || \ c \geq d$
4.  $a \geq 2 \ \&\& \ a \leq 5$
5.  $a \geq 1 \ \&\& \ (b \geq 2 \ || \ c \geq 3)$ .

## Условный оператор

Условный оператор имеет вид:

`if (expression) statement; else statement;`

Причем нужно отметить, что ветка **else** всегда относится к ближайшему **if**

```
if (expr) stmt;
else if (expr) stmt;
else if (expr) stmt;
else stmt;
```

**Задача.** Необходимо считать два целых числа и вывести наибольшее из них.

<pre>int a,b; scanf("%d%d",&amp;a,&amp;b); if (a&gt;b)     printf("%d",a); else     printf("%d",b);</pre>	<pre>int a,b; scanf("%d%d",&amp;a,&amp;b); if (a&gt;b) {     printf("%d",a); } else {     printf("%d",b); }</pre>
---	---

## Тернарные операции **a ? b : c;**

Тернарный оператор **?:** позволяет сократить определение простейших условных конструкций **if** и имеет следующую форму:

*expression ? statement1 : statement2;*

- сначала вычисляется выражение *expression1*
- **если** его значение отлично от нуля, то результатом всей операции является значение выражения *statement1* :
- **иначе** – значение выражения *statement2*;

<pre>int a, b, max; scanf("%d%d",&amp;a,&amp;b); if (a &gt; b)     max = a; else     max = b; printf("%d",max);</pre>	<pre>int a, b, max; scanf("%d%d",&amp;a,&amp;b); //короткая запись if else //условная операция max = a &gt; b ? a : b; printf("%d",max);</pre>
<pre>int a, b, max; scanf("%d%d",&amp;a,&amp;b); max = a; if (b &gt; max)     max = b; printf("%d",max);</pre>	Можно вот так короче

Минимальное из чисел **a** и **b**:

```
min = (a < b) ? a : b;
```

Тернарная операция может быть использована в ситуации, не связанной с

присваиванием:

```
#include <stdio.h>
const int TV_PAL    = 1;
const int TV_SECAM  = 0;
const int TV_WORK   = 1;
const int TV_TEST   = 0;
int main(int argc, char **argv)
{
    int tv_system = TV_SECAM;
    int tv_input  = TV_TEST;
    printf("TV Diagnostic %s %s",
           tv_system == TV_PAL ? "PAL" : "SECAM", tv_input ? "WORK" : "TEST"
    );
    return 0;
}
```

$a ? b : c ? d : e$  эквивалентно  $(a ? b : (c ? d : e))$

## Побитовые операции

Побитовые операции работают над двоичным представлением числа.

Одноместными побитовыми операциями является:

- “~” - инверсия или побитовое отрицание

Двухместными – побитовыми операциями является:

- “&” - побитовое И
- “|” - побитовое ИЛИ
- “^” - побитовое исключающее ИЛИ (XOR)
- “<<” - побитовый сдвиг влево
- “>>” - побитовый сдвиг вправо.



Сдвиг на отрицательное число бит не определен (**warning: right shift count is negative**)



Сдвиг на число бит, превосходящее размер первого операнда, не определен (**warning: greater than or equal to the length in bits of the promoted left operand**)



Для побитовых вычислений рекомендуется всегда использовать беззнаковые типы.




При сдвиге вправо беззнакового операнда освобождающиеся биты заполняются нулями, тогда как для знаковых операндов возможно заполнение

как нулями так и значением знакового бита – выбор точного поведения зависит от компилятора.

```
int a = 1, b = -1;
printf("a = %d b = %d", a >> 1, b >> -1); // a = 0 b = -1
```

В зависимости от знаковости типа операнда может быть получен различный результат.

 Для побитовых вычислений рекомендуется всегда использовать беззнаковые типы

```
int a = 0xaabbccdd;
a = a >> 4;
// арифметический сдвиг
printf("%x\n", a);
// 0xf aabbccd
```

```
unsigned int u = 0xaabbccdd;
u = u >> 4;
// логический сдвиг
printf("%x\n", u);
// 0x0 aabbccd
```

Арифметический сдвиг аналогичен логическому, но значение слова считается знаковым числом представленному дополнительным кодом. Арифметический сдвиг — сдвигом со знаком (для знаковых типов).

## Операция последовательного вычисления

Оператор запятая "," - операнды вычисляются слева направо, а результатом операции является значение последнего вычисленного выражения.

```
int a, b;
a = (b = 5, b + 2) // переменной b присваивается 5, потом вычисляется b+2
                  // и результат кладется в a
```



Операция запятая (,) позволяет объединить несколько выражений в одно выражение и, таким образом, в Си вводится понятие выражение с запятой, имеющее общий вид записи:

<выражение>, <выражение>, <выражение>, ...;



Например, когда нужно проинициализировать перед началом цикла или увеличить в конце цикла несколько переменных:



```
for (i = 1, j = 2; i < n; j++, i++)
```

Подробнее изучим в разделе Циклы.



`int a, b = 10, c;` Операция запятая (,) при объявлении переменных

## Оператор присваивания и побочный эффект

**Оператор** – команда языка программирования.

**Оператора присваивания** – вычисляемая команда изменения значения переменной. Это наиболее частая операция.

- Левая часть присваивания должна обозначать объект памяти
- Правая – являться выражением

Например,  $a = b + c$ , где  $a, b, c$  – целочисленные переменные.

Результатом присваивания является изменение объекта памяти, т.е. **побочный эффект**.



Обратите внимание, что присваивание является операцией и генерирует значение – результатом присваивания является значение левой части

Следовательно, возможно выписать цепочку присваиваний ( $a = b = c + d$ ), при этом операция присваивания *ассоциируется* справа налево (т.е. сначала вычисляется выражение  $c+d$ , его результат записывается в переменную  $b$ , а потом – в переменную  $a$ , **побочным эффектом** этого выражения является изменение переменных  $a$  и  $b$ ).

```
int main()
{
    int a, b, x, y, z;
    a = 5; //положить целое число 5 в переменную a
    x = a + 20; // x = 25
    y = (x - 15) * (x + a); // y = 300
    z = y = a+1; // y = 6 z = 6
}
```

## Точка следования

Точкой следования (**sequence point**), называется место в программе, в котором все побочные эффекты предыдущих вычислений закончены, а новые – не начаты.

В корректной программе между двумя точками следования изменение значения переменной возможно не более одного раза, при этом старое значение читается только для определения нового. Точками следования в программе являются:

- конец полного выражения

```
int a, b, c, x = 0;  
a = (a + b) * (c + 3); // после вычисления всего выражения, перед равно  
x = x + 2; // благодаря точке следования, это нормальный код с точки  
зрения программирования. С точки зрения математики - оно не верное.
```

- при выполнении оператора “запятая”:  $x, y$  – между вычислением  $x$  и  $y$ ;
- при выполнении операции  $z ? x : y$  – между вычислением  $z$  и вычислением  $x$  либо  $y$ ;
- при вызове функции – перед выполнением ее тела и после вычисления ее аргументов;
- при выполнении логических операций  $x \&\& y$  и  $x || y$  – между вычислением  $x$  и вычислением  $y$ .



### Внимание!

```
int a, b = 10, c;
```



$a = b++ + ++b$ ; **ТАК НЕЛЬЗЯ** дважды модифицируется одна переменная



$a = (c = 3) + (c = 2)$ ; **ТАК НЕЛЬЗЯ** дважды модифицируется одна переменная

Между двумя точками последовательных вычислений изменение значения переменной возможно не более одного раза.

## Старшинство операций

Для того, чтобы повысить качество кода и уменьшить количество дополнительных скобок в выражениях, полезно знать и использовать приоритет выполнения различных арифметических и логических операций.

В языке Си компилятор сам определяет, в каком порядке выполнять операции одинакового приоритета. Это называется ассоциативность (associativity).

Операции	Ассоциативность
$() [] \rightarrow .$	$\rightarrow$

! ~ ++ -- + - sizeof (type) * &	←
* / %	→
+ -	→
<< >>	→
< <= > >=	→
== !=	→
&	→
^	→
	→
&&	→
	→
?:	←
= += -= *= /= %= &= ^=  = <<= >>=	←
,	→

Из таблицы следует, что операнды, представляющие вызов функции **()**, индексное выражение **[ ]**, выражение выбора элемента **->** . и выражение в скобках, имеют наибольший приоритет и ассоциативность слева направо.

Приведение типа **(type)** имеет тот же приоритет и порядок выполнения, что и унарные операции **! ~ ++ -- + - sizeof \* &**.

Выражение может содержать несколько операций одного приоритета. Когда несколько операций одного и того же уровня приоритета появляются в выражении, то они применяются в соответствии с их ассоциативностью — либо справа налево, либо слева направо.

Следует отметить, что в языке Си принят неудачный порядок приоритета для некоторых операций, в частности для операции сдвига и поразрядных операций. Они имеют более низкий приоритет, чем арифметические операции (сложение и др.). Поэтому выражение

```
a = b & 0xFF + 5; // вычисляется как
a = b & (0xFF + 5);
```

а выражение


```
a + c >> 1; //вычисляется как  
(a + c) >> 1;
```

Другой порядок вычислений можно определить скобками:

```
a = (b & 0xFF) + 5;  
a + (c >> 1);
```

Приоритет	Оператор	Описание	Ассоциативность
1	++	Суффиксный инкремент	Слева направо
	--	Суффиксный декремент	
	()	Вызов функции	
	[]	Взятие элемента массива	
	.	Выбор элемента по ссылке	
	->	Выбор элемента по указателю	
2	++	Префиксный инкремент	Справа налево
	--	Префиксный декремент	
	+	Унарный плюс	
	-	Унарный минус	
	!	Логическое НЕ	
	~	Побитовое НЕ	
	(type)	Приведение типа	
	*	Разыменование указателя	
	&	Взятие адреса объекта	
	sizeof	Sizeof (размер)	
3	*	Умножение	Слева направо
	/	Деление	
	%	Получение остатка от деления	
4	+	Сложение	
	-	Вычитание	
5	<<	Побитовый сдвиг влево	
	>>	Побитовый сдвиг вправо	
6	<	Меньше	
	<=	Меньше или равно	
	>	Больше	
	>=	Больше или равно	
7	==	Равенство	

	!=	Неравенство	
8	&	Побитовое И (and)	
9	^	Побитовое исключающее ИЛИ (xor)	
10		Побитовое ИЛИ (or)	
11	&&	Логическое И	
12		Логическое ИЛИ	
13	?:	Тернарная условная операция	Справа налево
	=	Присваивание	
	+=	Сложение, совмещенное с присваиванием	
	-=	Вычитание, совмещенное с присваиванием	
	*=	Умножение, совмещенное с присваиванием	
	/=	Деление, совмещенное с присваиванием	
	%=	Вычисление остатка от деления, совмещенное с присваиванием	
	<<=	Побитовый сдвиг влево, совмещённый с присваиванием	
	>>=	Побитовый сдвиг вправо, совмещённый с присваиванием	
	&=	Побитовое «И», совмещенное с присваиванием	
	=	Побитовое «ИЛИ», совмещенное с присваиванием	
	^=	Побитовое «исключающее ИЛИ» (xor), совмещенное с присваиванием	
15	,	Оператор «запятая»	Слева направо

 Обратите внимание, что тернарная операция '? :' имеет более низкий приоритет, чем сложение, умножение, оператора побитового ИЛИ и так далее

Рассмотрим код:

```
int z = x + (A == B) ? 1 : 2;
```

Фактически, здесь написано:

```
int z = (x + (A == B)) ? 1 : 2;
```

А хотелось:

```
int z = x + (A == B ? 1 : 2);
```

Предлагается написать так:

```
int Z = X;  
Z += A == B ? 1 : 2;
```

## Форматирование кода

Форматирование кода - это оформление его особым образом, чтобы код выглядел единообразно, его было удобно читать и удобно вносить изменения.

Как мы говорили ранее текст программы пишется в первую очередь для прочтения человеком, а уже потом для обработки компьютером.

Хорошо оформленный код легко читается программистом, что позволяет его улучшать, расширять и переиспользовать в будущем. Такой код содержит меньше ошибок и упрощает отладку программы.

При написании программы первоочередная задача обеспечить ее правильное оформление. Если код не оформлен должным образом он не считается написанным, даже если успешно проходит трансляцию и как-то работает.

## Структура программы

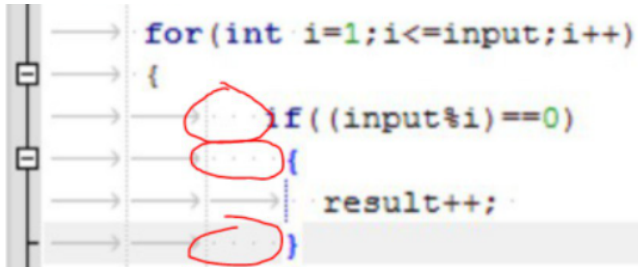
Структура программы формируется по принципу “матрешки”: одни операторы содержат другие.

Отступы позволяют высветить структуру программы показывая уровень вложенности оператора. Отступы делаются при помощи пробелов или табуляции. Рассмотрим на примере программы отыскивающую корень линейного уравнения вида  $ax + b = 0$ .

```
#include <stdio.h>  
  
int main()  
{  
    float a, b;  
    printf("Solve a*x+b=0\n");  
    printf("Enter a and b:\n");  
    scanf("%f%f", &a, &b);  
    if (a == 0)  
    {  
        if (b == 0)  
            printf("True for any x\n");  
        else  
            printf("No roots\n");  
    }  
    else  
        printf("x = %5.5f\n", -b/a);  
    return 0;
```

```
}
```

🔥 Для отступа используют табуляцию **ИЛИ** пробелы. Их смешивание **недопустимо**.



```
for(int i=1;i<=input;i++)  
{  
    if((input%i)==0)  
    {  
        result++;  
    }  
}
```

Если в программе встречается сложный оператор, состоящий из других операторов, то каждый уровень вложенности оператора отмечается отступом.

Рассмотрим форматирование сложного оператора *if*:



```
if (p) res = p -> data;  
else res = 0;
```



```
if (p)  
    res = p -> data;  
else  
    res = 0;
```

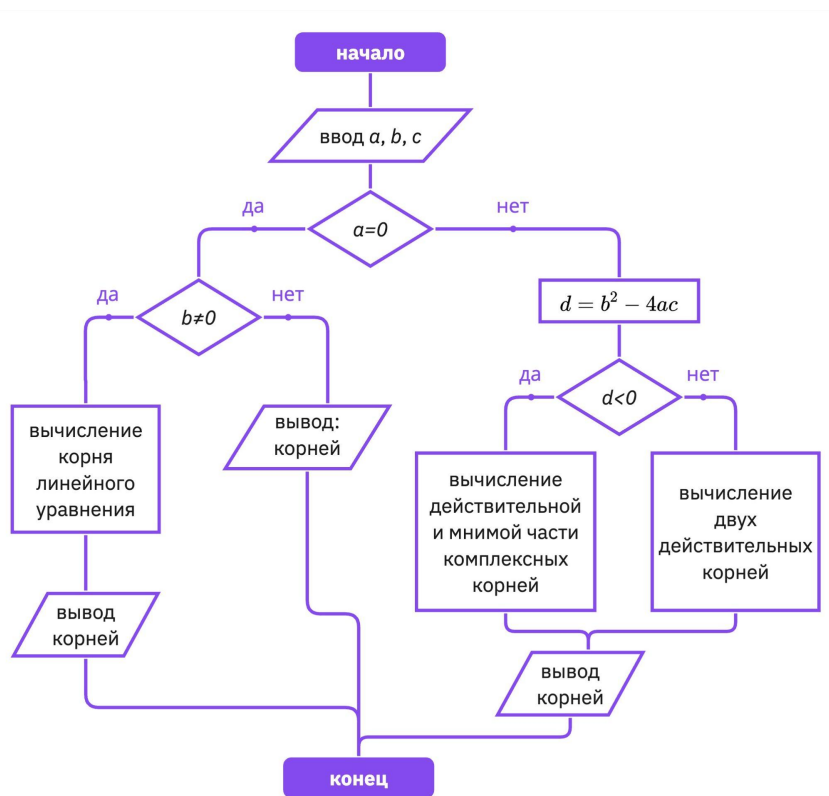


```
if (p)  
{  
    flag = 1;  
    x = p -> data;  
}  
else  
{  
    p = malloc(sizeof(*p));  
    p -> data = x;  
}
```

В инструкции указаны настройки Geany, для отображения знаков табуляции и пробела.

# Вычисление корней квадратного уравнения (продолжение)

По итогам сегодняшней лекции мы уже можем реализовать более сложный алгоритм вычисления корней квадратного уравнения с обработкой различных исключительных ситуаций, таких как деление на 0, отрицательное значение дискриминанта и т.д.



Вычисление корней квадратного уравнения (продолжение) - изменения

```
B = b/2;
if(a!=0)
{
    d = B*B-a*c;
    if(d<0)
    {
        printf("Корни квадратного уравнения комплексные \n");
    }
    else
    {
        printf("Корни квадратного уравнения \n");
        d = sqrtf(d);
        X1 = (-B + d)/a; //2
        printf("X1 = %f \n",X1);
        X2 = (-B - d)/a; //16
        printf("X2 = %f \n",X2);
    }
}
else
```



```

{
    if(b!=0)
    {
        X1 = -c/b;
        printf("Корень линейного уравнения %f\n",X1);
    }
    else
    {
        printf("Корней НЕТ!\n");
    }
}

```

Вычисление корней квадратного уравнения (продолжение) код целиком:

```

#include <stdio.h>
#include <locale.h>
#include <math.h>

int main(int argc, char **argv)
{
    float a,b,c;
    float B,d;
    float X1,X2;
    setlocale(LC_ALL, "Rus");
    printf("Вычисление корней квадратного уравнения \\ \\ \"a*x*x+b*x+c=0\\\"\n");
    printf("Введите a:\n");
    scanf ("%f", &a); //1
    printf("Введите b:\n");
    scanf ("%f", &b); //18
    printf("Введите c:\n");
    scanf ("%f", &c); //32
    B = b/2;
    if(a!=0)
    {
        d = B*B-a*c;
        if(d<0)
        {
            printf("Корни квадратного уравнения комплексные \n");
        }
        else
        {
            printf("Корни квадратного уравнения \n");
            d = sqrtf(d);
            X1 = (-B + d)/a; //2
            printf("X1 = %f \n",X1);
            X2 = (-B - d)/a; //16
            printf("X2 = %f \n",X2);
        }
    }
    else
    {
        if(b!=0)
        {
            X1 = -c/b;
            printf("Корень линейного уравнения %f\n",X1);
        }
        else
        {
            printf("Корней НЕТ!\n");
        }
    }
}

```

```
    return 0;  
}
```

## Подведение итогов

Итак, на этой лекции мы рассмотрели [типы фиксированного размера](#) в рамках стандарта C99, а также [спецификаторы фиксированной длины для функций printf, scanf](#).

Поняли как устроено [представление целых знаковых чисел](#) в дополнительном коде.

Разобрали что такое [явное приведение типов](#) и [неявное приведение типов](#).

Познакомились на примерах с [операциями сравнения](#) и [логическими операциями](#). Изучили [условный оператор](#) if, а также [тернарные операции a ? b : c](#), применяемые для сокращенной записи.

Увидели как работают [лобитовые операции](#) в языке Си

Была рассмотрена [операция последовательного вычисления](#) “,” и [точка следования](#), а также [оператор присваивания и побочный эффект](#).

Обратили внимание на приоритет и ассоциативность [операций](#).

Начали изучение [форматирования кода](#) и разобрались в [структуре программы](#).

Продолжили решение задачи [вычисления корней квадратного уравнения](#) с обработкой различных исключительных условий.

## Домашнее задание

Отработать примеры кода, показанные в лекции.

# Используемые источники

1. [cprogramming.com](http://cprogramming.com) - учебники по [C](#) и [C++](#)
2. [free-programming-books](http://free-programming-books) - ресурс содержащий множество книг и статей по программированию, в том числе по [C](#) и [C++](#) (там же можно найти ссылку на распространяемую бесплатно автором html-версию книги Eckel B. «Thinking in C++»)
3. [tutorialspoint.com](http://tutorialspoint.com) - ещё один ресурс с множеством руководств по изучению различных языков и технологий, в том числе содержит учебники по [C](#)
4. Юричев Д. [«Заметки о языке программирования Си/Си++»](#) - «для тех, кто хочет освежить свои знания по Си/Си++»
5. Статья про MISRA <https://habr.com/ru/company/pvs-studio/blog/482490/>
6. [Онлайн версия «Си для встраиваемых систем»](#)

Дополнительные материалы

7. <https://studfile.net/preview/2919030/page:18/>
8. <https://studfile.net/preview/1743723/page:11/>
9. <https://it.wikireading.ru/35124>
10. [https://www.eusi.ru/lib/bochkov\\_jazyk\\_programmirovania\\_si\\_dlja/4.shtml](https://www.eusi.ru/lib/bochkov_jazyk_programmirovania_si_dlja/4.shtml)
11. [https://translated.turbopages.org/proxy\\_u/en-ru.ru.fca6f0a8-6416f55b-03d5a3fa-74722d776562/https/stackoverflow.com/questions/1082655/conditional-operator-differences-between-c-and-c](https://translated.turbopages.org/proxy_u/en-ru.ru.fca6f0a8-6416f55b-03d5a3fa-74722d776562/https/stackoverflow.com/questions/1082655/conditional-operator-differences-between-c-and-c)
12. <https://it.wikireading.ru/35124>
13. <https://studfile.net/preview/1743723/page:11/>
14. <https://studfile.net/preview/3084791/page:7/>
15. <https://pvs-studio.ru/ru/blog/terms/0064/>

16. <https://it.wikireading.ru/32064>
17. [https://www.tutorialspoint.com/cprogramming/c\\_operators\\_precedence.htm](https://www.tutorialspoint.com/cprogramming/c_operators_precedence.htm)
18. <https://inickel.livejournal.com/54697.html>
19. <http://microsin.net/programming/arm/c-operators-precedence.html>
20. <https://learn.microsoft.com/ru-ru/cpp/c-language/precedence-and-order-of-evaluation?view=msvc-160&viewFallbackFrom=vs-2019>