

## Лекция 7

**Операторы.  
Преобразования типов данных.  
Приоритеты операций.**

# Операторы

Знак операции – это один или более символов, определяющих действие над операндами.

Внутри знака операции пробелы не допускаются.

Делятся:

- унарные;            `// i++;`
- бинарные;        `// i+j;`
- тернарную.        `// (a>b)?a:b;`

Один и тот же знак может интерпретироваться по-разному в зависимости от контекста.

`x=-y;` // унарный минус

`x-=y;` // бинарный минус  $\rightarrow$  `x=x-y;`

`x--=-y;` //?

Все знаки, за исключением `[]`, `()` и `?:` представляют собой отдельные лексемы.

# Унарные операторы

Операция	Краткое описание
<code>++</code>	увеличение на 1 (инкремент)
<code>--</code>	уменьшение на 1 (декремент)
<code><u>sizeof</u></code>	размер
<code>~</code>	поразрядное отрицание
<code>!</code>	логическое отрицание
<code>-</code>	арифметическое отрицание (унарный минус)
<code>+</code>	унарный плюс
<code>&amp;</code>	взятие адреса
<code>*</code>	<u>разадресация</u>
<code>new</code>	выделение памяти
<code>delete</code>	освобождение памяти
<code>(type)</code>	преобразование типа

# Бинарные операторы

Операция	Краткое описание
*	умножение
/	деление
%	остаток от деления
+	сложение
-	вычитание
<<	сдвиг влево
>>	сдвиг вправо
<	меньше
<=	меньше или равно
>	больше
>=	больше или равно
==	равно
!=	не равно
&	поразрядная конъюнкция (И)
^	поразрядное исключающее ИЛИ
	поразрядная дизъюнкция (ИЛИ)
&&	логическое и
	логическое или

# Тернарная и бинарные операторы

Операция	Краткое описание
?:	условная операция (тернарная)
=	присваивание
*=	умножение с присваиванием
/=	деление с присваиванием
%=	остаток деления с присваиванием
+=	сложение с присваиванием
-=	вычитание с присваиванием
<<=	сдвиг влево с присваиванием
>>=	сдвиг вправо с присваиванием
&=	поразрядное <u>И</u> с присваиванием
=	<u>поразрядное ИЛИ</u> с присваиванием
^=	<u>поразрядное</u> <u>исключающее ИЛИ</u> с присваиванием
,	последовательное вычисление

# Операции инкремента и декремента

Операции увеличения и уменьшения на 1 (**++** и **--**). Эти операции, имеют две формы записи – префиксную, когда операция записывается перед операндом, и постфиксную. В префиксной форме сначала изменяется операнд, а затем его значение становится результирующим значением выражения, а в постфиксной форме значением выражения является исходное значение операнда, после чего он изменяется.

**++n1**    увеличить до использования;

**n1++**    увеличить после использования;

**--n1**    уменьшить до использования;

**n1--**    уменьшить после использования.

Пусть **n=5** ;

**x=n++** ;

**?x=    ?n=**

**x=++n** ;

**?x=    ?n=**

Инкрементные и декрементные операторы можно применять только к переменным.

**(i+j)++    /\*неверно\*/**

Если требуется только увеличить или уменьшить значение переменной, то безразлично какой оператор использовать, как например:

**if (c == '\n') n1++;**

# Операция sizeof

Операция определения размера **sizeof** предназначена для вычисления размера объекта или типа в байтах, и имеет две формы:

**sizeof выражение**

**sizeof ( тип )**

Пример:

```
#include<iostream>
using namespace std;
int main() {
```

```
    float x = 1;
```

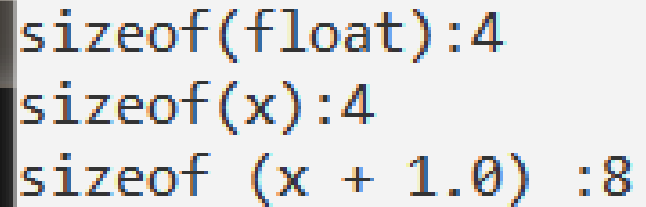
```
    cout<<"sizeof(float) : "<<sizeof(float) ;
```

```
    cout<<"\nsizeof(x) : "<<sizeof x;
```

```
    cout<<"\nsizeof (x + 1.0) : "<<sizeof(x +1.0) ;
```

```
    return 0;
```

```
}
```



```
sizeof(float):4
sizeof(x):4
sizeof (x + 1.0) :8
```

Последний результат связан с тем, что вещественные константы по умолчанию имеют тип **double**, к которому, как к более длинному, приводится тип переменной **x** и всего выражения. Скобки необходимы для того чтобы выражение, стоящее в них, вычислялось раньше операции приведения типа, имеющей больший приоритет, чем сложение.

# Операторы

**Арифметическое отрицание** (унарный минус -) изменяет знак операнда целого или вещественного типа на противоположный.

**Логическое отрицание** (!) дает в результате значение 0, если операнд есть истина (не нуль), и значение 1, если операнд равен нулю. Операнд должен быть целого или вещественного типа, а может иметь также тип указатель. **Поразрядное отрицание** (~), часто называемое побитовым, инвертирует каждый разряд в двоичном представлении целочисленного операнда.

**Деление** (/) и **остаток от деления** (%). Операция деления применима к операндам арифметического типа. Если оба операнда целочисленные, результат операции округляется до целого числа, в противном случае тип результата определяется правилами преобразования. Операция остатка от деления применяется только к целочисленным операндам. Знак результата зависит от реализации.



# Операторы

Пример:

```
#include<stdio.h>
```

```
int main() {
```

```
    int x=10, y=3;
```

```
    float z = 3.0f;
```

```
    printf("x/y=%d x/z=%f", x/y, x/z);
```

```
    printf("\n %d", x%y);
```

```
    return 0;
```

```
}
```

```
x/y=3 x/z=3.333333
1
```

```
#include<iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    cout<<10/3<<endl;
```

```
    cout<<10/3.0<<endl;
```

```
    cout<<10.0f/3<<endl;
```

```
    cout<<sizeof(10/3.0)<<endl;
```

```
    cout<<sizeof(10.0f/3)<<endl;
```

```
    return 0;
```

```
}
```

```
3
3.33333
3.33333
8
4
```

# Операторы

## Относительные приоритеты:

- 1) унарные + и -;
- 2) \*, /, %
- 3) +, -

## Операторы отношений и логические операторы

Относительные приоритеты:

- 4) >, >=, <=, <
- 5) ==, !=
- 6) &&
- 7) ||

## Унарный оператор ! (not)

if (!f) эквивалентно if f==0

**i < lim-1 && (c=getchar()) != '\n' && c != EOF**

↑     ↑     ↑     ↑                     ↑     ↑     ↑

2     1     6     3                     4     7     5

# Побитовые операторы

В Си имеются шесть операторов для манипулирования с битами. Их можно применять только к целочисленным операндам, т. е. к **char, short, int и long**, знаковым и беззнаковым:

<b>&amp;</b>	- побитовое И
<b> </b>	- побитовое ИЛИ
<b>^</b>	- побитовое исключающее ИЛИ
<b>&lt;&lt;</b>	- сдвиг влево
<b>&gt;&gt;</b>	- сдвиг вправо
<b>~</b>	- побитовое отрицание (унарный)

**Оператор &** (побитовое И) часто используется для обнуления некоторой группы разрядов.

Например:

**n = n & 0177;**

обнуляет в **n** все разряды, кроме младших семи.

(001 111 111)

# Побитовые операторы

**Оператор  $|$**  (побитовое ИЛИ) применяют для установки разрядов; так,

```
#define SET_ON 0177
```

```
x = x | SET_ON;
```

устанавливает единицы в тех разрядах  $x$ , которым соответствуют единицы в **SET\_ON**.

**Оператор  $\wedge$**  (побитовое исключающее ИЛИ) в каждом разряде установит 1, если соответствующие разряды операндов имеют различные значения, и 0, когда они совпадают:

$$\begin{array}{r} \wedge \quad 10101 \\ \quad 11111 \\ \hline \quad 01010 \end{array}$$

# Побитовые операторы

**Операторы << и >>** сдвигают влево или вправо свой левый операнд на число битовых позиций, задаваемое правым операндом, который должен быть неотрицательным.

```
x << 2;      /*x=x*04  1011 1011→ 1110 1100*/
```

```
unsigned char y;
```

```
y>>2;        /* 1011 1100 →0010 1111*/
```

```
signed char z;
```

```
z>>2;        /* 1011 1100 →1110 1111 */
```

**Унарный оператор ~** поразрядно "обращает" целое, т. е. превращает каждый единичный бит в нулевой и наоборот.

```
x = x & ~077;    /*обнуляет в x последние 6 разрядов*/
```

```
/* getbits: получает n бит, начиная с p-й позиции */
```

```
unsigned getbits(unsigned x, int p, int n) {
```

```
    return (x>> (p+1-n)) & ~(~0<< n);
```

```
}
```

При вызове `getbits(x, 4, 3);` `x=1001 1001 1011 1101`,

то рез-т равен 7

# Побитовые операторы

```
unsigned getbits (unsigned x, int p, int n) {  
    return (x>>(p+1-n)) & ~ (~0<<n) ; }
```

getbits(x, 4, 3); → если x=1001 1001 1011 1101  
p=4 n=3 (p+1-n)=4+1-3=2

x	1001	1001	1011	1101
x>>(p+1-n)	00	1001	1001	1011 11

0	0000	0000	0000	0000
~0	1111	1111	1111	1111
~0<<n	1111	1111	1111	1000
~ (~0<<n)	0000	0000	0000	0111

( ) & ~ ( )	0010	0110	0110	1111
&	0000	0000	0000	0111

-----  
0000 0000 0000 0111 → 7

# Операторы присваивания

Присваивание `i = i + 2`, можно написать в сжатом виде:

`i += 2;`

Имеются следующие операторы присваивания:

`op=`, где `<op> ::= + | - | * | / | % | << | >> | & | ^ | |` (побит. или)

`i*=2;        /*    i=i*2    */`

`x*=y+1;     /*    x=x*(y+1)   */`

**Присваивание может применяться в выражениях!:**

`while ((c=getchar()) != EOF)`

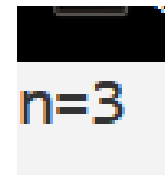
`int bitcount(unsigned x) { // bitcount: подсчет единиц в x`

`int b;`

`for (b=0; x!=0; x>>=1)`

`if (x & 01) b++;`

`return b;}`

A small screenshot of a terminal window. It has a black title bar and a light gray background. The text 'n=3' is displayed in a monospaced font, with 'n=' in black and '3' in a reddish-orange color.

`int main ( ){`

`unsigned x=7;    int n=bitcount(x);`

`printf("n=%d",n);`

`}`

# Операторы присваивания

Операции присваивания могут использоваться в программе как законченные операторы. Формат операции простого присваивания:

**<значение> = <выражение>;**

Сначала вычисляется выражение, стоящее в правой части операции, а потом его результат записывается в область памяти, указанную в левой части (мнемоническое правило: «присваивание – это передача данных "налево"»). То, что ранее хранилось в этой области памяти, естественно, теряется.

```
int main() {  
    int a=3,b=5,c=7;  
    a=b;b=a;c=c+1;  
    printf("%d %d %d",a,b,c);  
    return 0;  
}  
a=? b=? c=?
```

a 3 b 5 c 7



# Преобразования типов при вычислении выражений

Если операнды оператора принадлежат к разным типам, то они **приводятся к некоторому общему типу**. Обычно автоматически производятся лишь те преобразования, которые без какой-либо потери информации превращают операнды с меньшим диапазоном значений в операнды с большим диапазоном (целое -> вещественное). Возможны преобразования с потерей информации.

Например:

**long int** → **int**; **float** → **int**;

Однако при этом могут выдаваться предупреждения.

Значения типа **char** — это *малые целые*, и их можно использовать в арифметических выражениях.

# Преобразования типов

Заголовочный файл **<ctype.h>** определяет семейство стандартных функций для преобразования символов. Например:

**tolower(c)** дает код строчной буквы **c**;

**isdigit(c)** выполняет проверку **c >= '0' && c <= '9'**.

Существует одна тонкость: язык не определяет, являются ли переменные типа **char** знаковыми или беззнаковыми. Поэтому преобразование **char** с единичным старшим битом в **int** на одних машинах будет превращено в отрицательное целое (посредством "распространения знака"). На других – преобразование **char** в **int** осуществляется добавлением нулей слева, и, таким образом, получаемое значение всегда положительно.

Гарантируется, что любой отображаемый символ из стандартной кодовой таблицы никогда не будет отрицательным числом.

*Поэтому переменные типа **char**, в которых хранятся числа, следует специфицировать явно как **signed** или **unsigned**.*

# Преобразования типов

**В общем случае**, когда бинарный оператор имеет разнотипные операнды, прежде чем операция начнет выполняться, **"низший" тип повышается до "высшего"**. Результат будет иметь высший тип.

**Набор неформальных правил** для преобразования типов в **выражениях**, когда нет беззнаковых операндов:

1) если один из операндов принадлежит типу **long double**, то и другой приводится к **long double**.

2) иначе, если один из операндов принадлежит типу **double**, то и другой приводится к **double**.

3) иначе, если один из операндов принадлежит типу **float**, то и другой приводится к **float**.

4) иначе операнды типов **char** и **short** приводятся к **int**.

5) и наконец, если один из операндов типа **long**, то и другой приводится к **long**.

# Преобразования типов

**Правила преобразования** усложняются с появлением операндов **типа unsigned**.

Сравнение знаковых и беззнаковых значений зависит от размеров целочисленных типов, которые на разных машинах могут отличаться.

Пусть **int** занимает **16** битов, **long** – **32** бита.

Тогда

1) **-1L < 1U** (**int** приводится к **long**)

2) **-1L > 1UL** (**long** приводится к **unsigned long**)

# Преобразования типов при присвоениях

**Общее правило:** значение правой части присвоения приводится к типу левой части, который и является типом результата.

1. Тип **char** превращается в **int** путем распространения знака или другим описанным выше способом.
2. Тип **long int** преобразуются в **short int** или в значения типа **char** путем **отбрасывания старших разрядов!!!**

```
int i; char c;  
    i = c;      c = i; /* без потерь */  
int i; char c;  
    c = i;      i = c; /* возможны потери */
```

3. Преобразование **float** к **int** сопровождается отбрасыванием дробной части. Если **double** переводится во **float**, то выполняется либо округление, либо отбрасывание дробной части (зависит от реализации)

4. Так как **аргумент функции** копируется, то при его передаче также возможны преобразования типов.

# Преобразования типов

Унарный **оператор явного преобразования** типа:

(имя-типа) выражение

```
int n; double y;
```

```
y=sqrt((double) n); /*явное преобразование типа*/
```

При наличии прототипа функции **sqrt: double sqrt(double);**

при обращении **sqrt(7)** целое **7** будет приведено в **double** автоматически (что собственно и происходит при обычном вызове стандартной функции **sqrt**).

```
#include<iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    cout<<5/2<<endl;
```

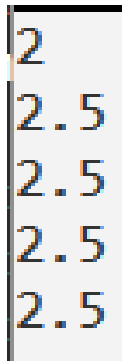
```
    cout<<5/2.0<<endl;
```

```
    cout<<5.0f/2<<endl;
```

```
    cout<<5/(float)2<<endl;
```

```
    cout<<(double)5/2<<endl;
```

```
    return 0; }
```



2
2.5
2.5
2.5
2.5

**Математические функции**, аналогичные собранным в библиотеке **<math.h>**, базируются *на вычислениях с двойной точностью*.

# Приоритеты операций и порядок вычислений

операторы	выполняются
() [] -> .	слева направо
! ~ ++ -- + - * & (тип) sizeof	справа налево
* / %	слева направо
+ -	слева направо
<< >>	слева направо
< <= > >=	слева направо
== !=	слева направо
&	слева направо
^	слева направо
	слева направо
&&	слева направо
	слева направо
?:	справа налево
= += -= *= /= %= &= ^=  = <<= >>=	справа налево
,	слева направо

# Порядок вычислений

Пример1:

```
int x=3, y=4, z=5;  
cout<<x<<y<<z;
```

Операция сдвиг “<<” выполняется слева направо, поэтому после выполнения операции: `cout<<x<<y<<z;`

Сначала будет выведено значение переменной **x**, затем значение переменной **y**, затем значение переменной **z**.

Пример2:

```
int x, y, z;  
x=y=z=5;
```

Операция присваивание “=” выполняется справа налево, поэтому сначала переменная **z** получает значение 5, затем переменной **y** будет присвоено значение переменной **z**, и только потом переменной **x** будет присвоено значение **y**.



# Порядок вычислений

Си не фиксирует очередность вычисления операндов оператора (за исключением `&&`, `||`, `? :` и `,`). Например, в инструкции вида

**`x = f() + g();`**

**`f`** может быть вычислена раньше **`g`** или наоборот.

Очередность вычисления аргументов функции также не определена, поэтому на разных компиляторах вызов

**`printf("%d %d\n", ++n, power(2, n)); /* НЕВЕРНО */`**

может давать несовпадающие результаты. Результат вызова функции зависит от того, когда компилятор сгенерирует команду увеличения **`n`** — до или после обращения к **`power`**.

Правильно:

**`++n;`**

**`printf("%d %d\n", n, power(2, n));`**

# Порядок вычислений

`a[i]=i++;`

Возникает вопрос: массив **a** индексируется старым или измененным значением **i**?

Компиляторы могут по-разному генерировать программу, что проявится в интерпретации данной записи.

**Мораль такова: писать программы нужно так, чтобы они не зависели от очередности вычислений.**