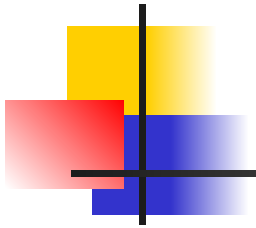
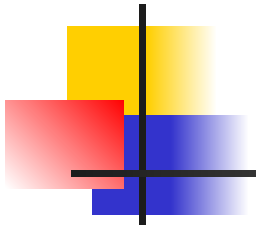


Глава 3

Операторы и выражения



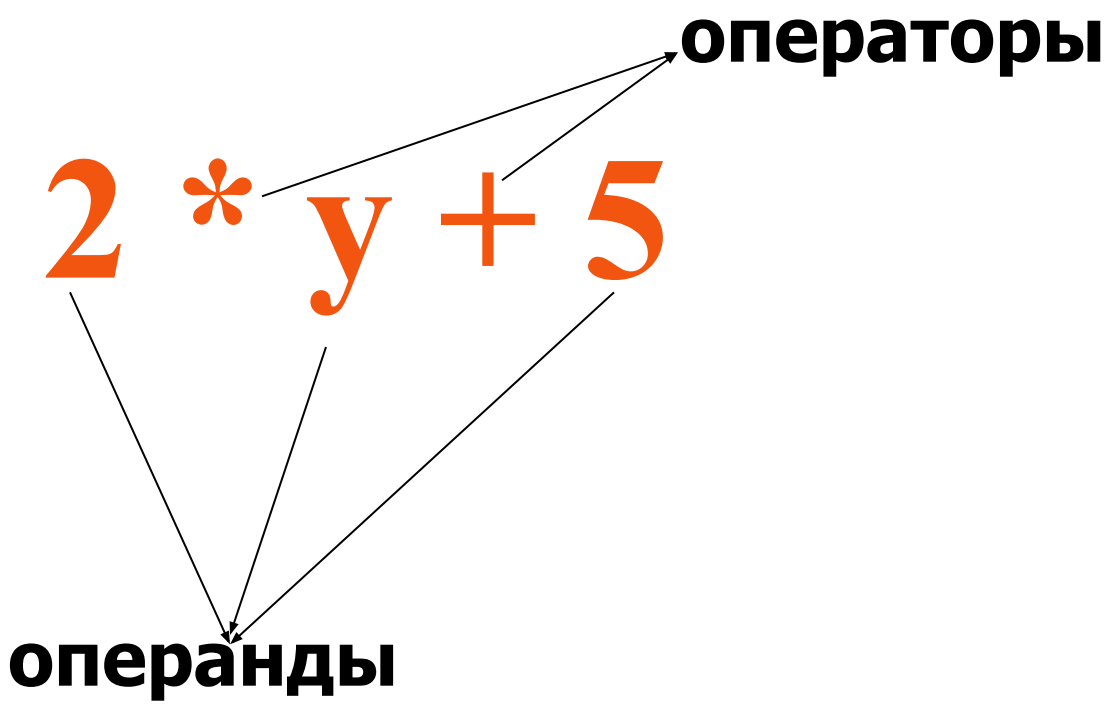
Задачи:

- Уметь объяснять операторы присвоения
- Понимать арифметические выражения
- Уметь понимать логические и операторы сравнения
- Понимать логические операторы и выражения
- Понимать приоритет операторов

Выражения

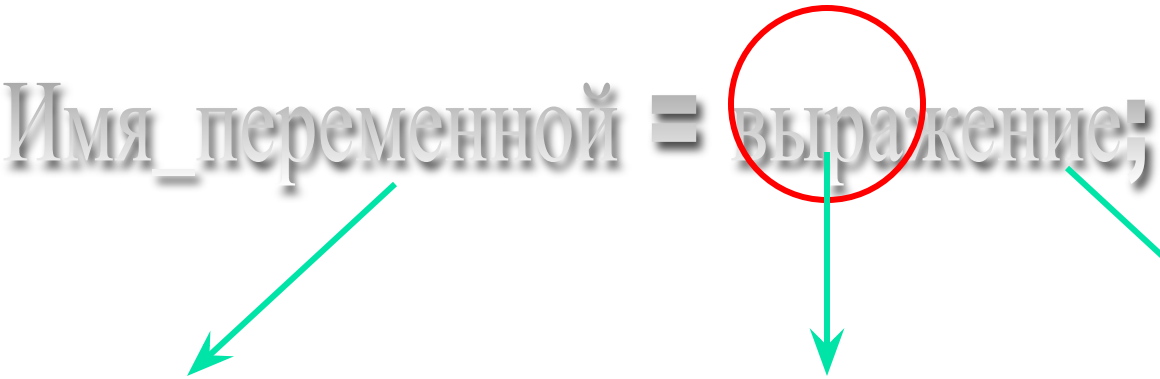
Комбинация операторов и операндов

Пример:



Оператор присваивания

Оператор присваивания(=) может быть использован с любым правильным выражением в С



lvalue

(левое значение)

Assignment operator

(Оператор присваивания)

rvalue

(правое значение)

Multiple Assignment

В одном и том же утверждении можно несколько раз переприсваивать одно и тоже значение нескольким переменным

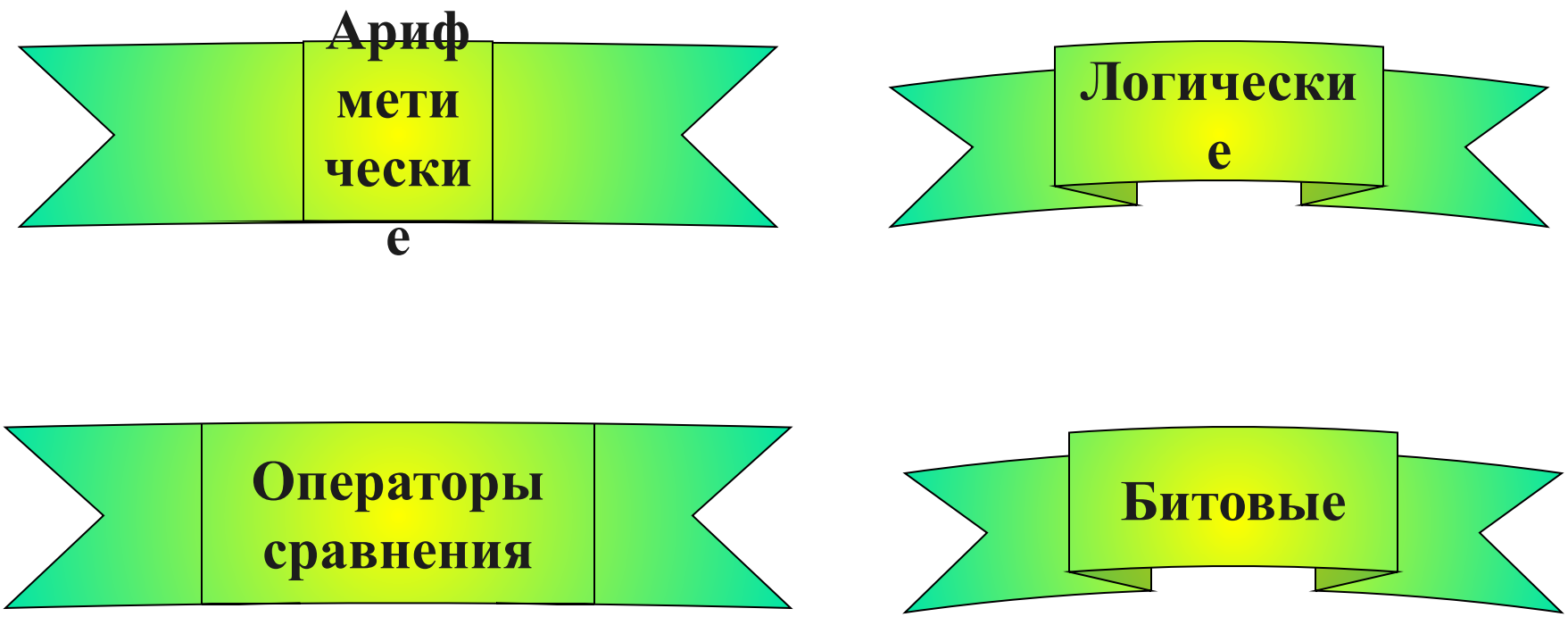
```
a = b = c = 10;
```

А вот так делать нельзя:

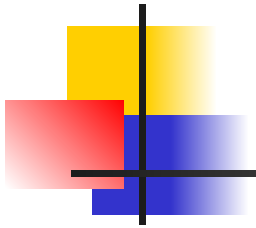
```
int a = int b = int b = int c = 10
```

Операторы

4 Типа



Арифметические выражения



Математические выражения могут быть представлены в C, используя арифметические операторы

Примеры:

++i % 7

5 + (c = 3 + 8)

a * (b + c/d)22

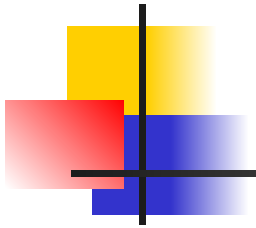
Арифметические операторы

Оператор	Символ	Пример	Операция
Сложение	+	x + y	x плюс y
Вычитание	–	x – y	x минус y
Умножение	*	x * y	x умножить на y
Деление	/	x / y	x разделить на y
Деление с остатком	%	x % y	Остаток от деления x на y

Арифметические операторы присваивания

Оператор	Символ	Пример	Операция
Присваивание	=	x = y	Присваиваем значение y переменной x
Сложение с присваиванием	+=	x += y	Добавляем y к x
Вычитание с присваиванием	−=	x −= y	Вычитаем y из x
Умножение с присваиванием	*=	x *= y	Умножаем x на y
Деление с присваиванием	/=	x /= y	Делим x на y
Деление с остатком и с присваиванием	%=	x %= y	Присваиваем остаток от деления x на y переменной x

Инкремент и декремент



Оператор	Символ	Пример	Операция
Префиксный инкремент (пре-инкремент)	++	++x	Инкремент x, затем вычисление x
Префиксный декремент (пре-декремент)	−−	−−x	Декремент x, затем вычисление x
Постфиксный инкремент (пост-инкремент)	++	x++	Вычисление x, затем инкремент x
Постфиксный декремент (пост-декремент)	−−	x−−	Вычисление x, затем декремент x

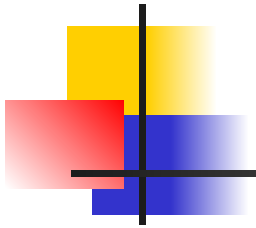
Инкремент и декремент

С операторами инкремента/декремента версии префикс всё просто. Значение переменной сначала увеличивается/уменьшается, а затем уже вычисляется.

Например

```
1  int x = 5;
2  int y = ++x; // x = 6 и 6 присваивается переменной y
```

Инкремент и декремент



А вот с операторами инкремента/декремента версии постфикс несколько сложнее. Компилятор создает временную копию переменной x, увеличивает или уменьшает оригинальный x (не копию), а затем возвращает копию. Только после возврата копия x удаляется. Например:

```
1  int x = 5;
2  int y = x++; // x = 6, но переменной y присваивается 5
```

Сравнительные & логические операторы

Используются для……

Сравнить значения в двух переменных, или между переменной и константой

Операторы сравнения:

Operator	Relational Operators Action
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
==	Equal
!=	Not equal

Сравнительные & логические операторы-2

Логические операторы это символы используемые для соединения или отрицания операторов сравнения

Operator	Logical Operators Action
&&	AND
||	OR
!	NOT

Пример: **if (a>10) && (a<20)**

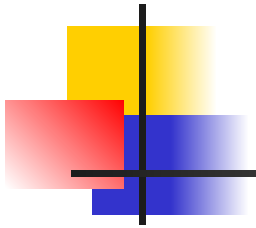
Выражения, в которых используются логические операторы возвращают 0 в случае ложного высказывания, и 1 в случае истинного

Логические операторы (Битовые)

Обрабатывают данные после представления чисел в их бинарную форму.(Битовое представление)

AND (NUM1 & NUM2)	Возвращает 1 если оба операнда 1
OR (NUM1 | NUM2)	Возвращает 1 если биты любого из двух операндов 1
NOT (~ NUM1)	Отрицание бита своего операнда (from 0 to 1 and 1 to 0)
XOR (NUM1 ^ NUM2)	Возвращает 1 если любой из битов в операнде 1 но не оба

Логические операторы Bitwise -2



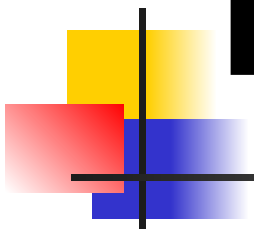
Пример:

10 & 15□ 1010 & 1111□1010 □ 10

10 | 15□ 1010 | 1111□1111 □ 15

10 ^ 15□ 1010 ^ 1111□0101 □ 5

~ 10 □ ~1010 □1011 □ -11



Преобразование типов

Процесс конвертации значений из одного типа данных в другой называется **преобразованием типов**. Преобразование типов может выполняться в следующих случаях:

Преобразование типов

Случай №1: Присваивание или инициализация переменной значением другого типа данных:

double k(4); // инициализация переменной типа double целым числом 4

k = 7; // присваиваем переменной типа double целое число 7

Преобразование типов

Случай №2: Передача значения в функцию, где тип параметра — другой:

```
void doSomething(long l)
{
}

doSomething(4); // передача числа 4 (тип int) в функцию с параметром типа long
```

Преобразование типов

Случай №3: Возврат из функции, где тип возвращаемого значения — другой:

```
float doSomething()
{
    return 4.0; // передача значения 4.0 (тип double) из функции, которая возвращает float
}
```

Преобразование типов

Случай №4: Использование бинарного оператора с операндами разных типов:

double division = **5.0 / 4**; // операция деления со значениями типов double и int

Преобразование типов

Есть 2 основных способа преобразования типов:

Неявное преобразование типов, когда компилятор автоматически конвертирует один фундаментальный тип данных в другой.

Явное преобразование типов, когда разработчик использует один из операторов явного преобразования для выполнения конвертации объекта из одного типа данных в другой.

Преобразование типов

Неявное преобразование типов (или «автоматическое преобразование типов») выполняется всякий раз, когда требуется один фундаментальный тип данных, но предоставляется другой, и пользователь не указывает компилятору, как выполнить конвертацию

Есть 2 основных способа неявного преобразования типов:

числовое расширение;
числовая конверсия.

Преобразование типов

Числовое расширение
Когда значение из одного типа данных конвертируется в другой тип данных **побольше** (по размеру и по диапазону значений), то это называется числовым расширением.

Например, тип int может быть расширен в тип long, а тип float может быть расширен в тип double:

long l(65); // расширяем значение типа int (65) в тип long
double d(0.11); // расширяем значение типа float (0.11) в тип double

Преобразование типов

В языке C++ есть два варианта расширений:

Интегральное расширение (или «**целочисленное расширение**»). Включает в себя преобразование целочисленных типов, меньших, чем int (bool, char, unsigned char, signed char, unsigned short, signed short) в int (если это возможно) или unsigned int.

Расширение типа с плавающей точкой. Конвертация из типа float в тип double.

Интегральное расширение и расширение типа с плавающей точкой используются для преобразования «меньших по размеру» типов данных в типы int/unsigned int или double (они наиболее эффективны для выполнения разных операций).

Важно: Числовые расширения всегда безопасны и не приводят к потере данных.

Преобразование типов

Числовые конверсии
Когда мы конвертируем значение из более крупного типа данных в аналогичный, но более **мелкий тип** данных, или конвертация происходит между **разными типами данных**, то это называется числовой конверсией.

Например:

```
double d = 4; // конвертируем 4 (тип int) в double
short s = 3; // конвертируем 3 (тип int) в short
```

В отличие от расширений, которые всегда безопасны, конверсии могут (но не всегда) привести к потере данных.

Преобразование типов

Во всех случаях, когда происходит конвертация значения из одного типа данных в другой, который не имеет достаточного диапазона для хранения конвертируемого значения, результаты будут неожиданные. Поэтому делать так не рекомендуется. Например, рассмотрим следующую программу:

```
#include <iostream>
 int main()
{
    int i = 30000;
    char c = i;
     std::cout << static_cast<int>(c);
     return 0;
}
```

В этом примере мы присвоили огромное целочисленное значение типа int переменной типа char (диапазон которого составляет от -128 до 127). Это приведет к переполнению и следующему результату: **48**

Преобразование типов

Однако, если число подходит по диапазону, конвертация пройдет успешно. Например:

```
#include <iostream>
int main()
{
	int i = 3;
	short s = i; // конвертируем значение типа int в тип short
	std::cout << s << std::endl;
	double d = 0.1234;
	float f = d; // конвертируем значение типа double в тип float
	std::cout << f << std::endl;
	return 0;
}
```

Здесь мы получим ожидаемый результат: **3** и **0.1234**

Преобразование типов

В случаях со значениями типа с плавающей точкой могут произойти округления из-за худшей точности в меньших типах.
Например:

```
#include <iostream>
#include <iomanip> // для std::setprecision()
 int main()
{
	float f = 0.123456789; // значение типа double - 0.123456789 имеет 9 значащих цифр, но float может хранить только 7
	std::cout << std::setprecision(9) << f; // std::setprecision определен в заголовочном файле iomanip
	return 0;
}
```

В этом случае мы наблюдаем потерю в точности, так как точность типа float меньше, чем типа double: **0.1234567**91

Преобразование типов

Обработка арифметических выражений

При обработке выражений компилятор разбивает каждое выражение на отдельные подвыражения. Арифметические операторы требуют, чтобы их операнды были **одного типа данных**. Чтобы это гарантировать, компилятор использует следующие правила:

Если операндом является **целое число меньше** (по размеру/диапазону) **типа int,** то оно подвергается **интегральному расширению** в int или в unsigned int.

Если операнды **разных типов данных**, то компилятор вычисляет **операнд с наивысшим приоритетом** и неявно **конвертирует тип другого операнда** в такой же тип, как у первого.

Преобразование типов

Приоритет типов операндов:

long double (самый высокий);
double;
float;
unsigned long long;
long long;
unsigned long;
long;
unsigned int;
int (самый низкий).

Преобразование типов

Автоматическое преобразование типов приведено ниже:

Char и short преобразуются в int, а float в double

Если один операнд типа double и второй типа double, то и результат будет типа double

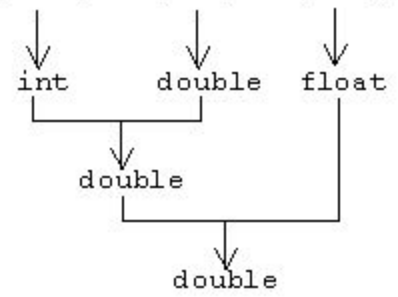
Если один операнд типа long и второй преобразуется к типу long, то и результат будет типа double

Если операнд неопределенного типа и второй также преобразуется к неопределенному типу, то и результат будет такого же типа

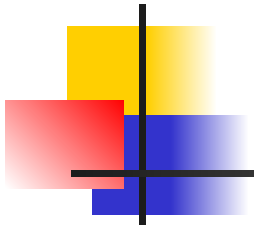
Если операнды слева типа int, то и результат будет типа int

Пример

```
char ch;
int i;
float f;
double d;
result = (ch/i) + (f*d) - (f+i);
```



Операторы явного преобразования типов данных



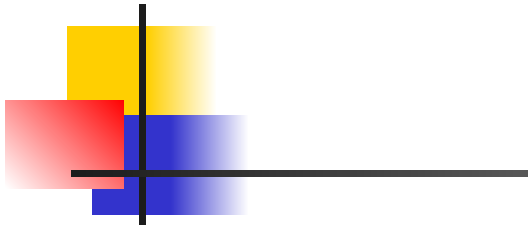
Конвертация C-style

В программировании на языке Си явное преобразование типов данных выполняется с помощью оператора (). Внутри круглых скобок мы пишем тип, в который нужно конвертировать. Этот способ конвертации типов называется конвертацией C-style. Например:

```
int i1 = 11;
int i2 = 3;
float x = (float)i1 / i2;
```

В программе, приведенной выше, мы используем круглые скобки, чтобы сообщить компилятору о необходимости преобразования переменной i1 (типа int) в тип float. Поскольку переменная i1 станет типа float, то i2 также затем автоматически преобразуется в тип float, и выполнится деление типа с плавающей точкой!

Операторы явного преобразования типов данных

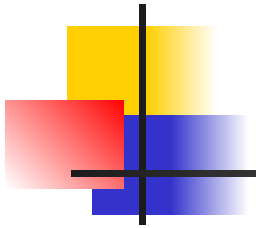


Язык C++ также позволяет использовать этот оператор следующим образом:

```
int i1 = 11;
int i2 = 3;
float x = float(i1) / i2;
```

Конвертация C-style **не проверяется компилятором** во время компиляции, поэтому она может быть неправильно использована, например, при конвертации типов const или изменении типов данных, без учета их диапазонов (что приведет к переполнению).

Операторы явного преобразования типов данных

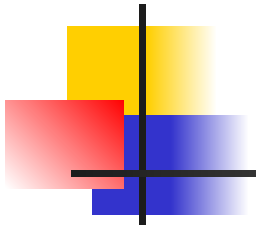


В языке C++ есть еще один оператор явного преобразования типов данных — оператор static_cast. Оператор static_cast лучше всего использовать для конвертации одного фундаментального типа данных в другой:

```
int i1 = 11;
int i2 = 3;
float x = static_cast<float>(i1) / i2;
```

Основным преимуществом оператора static_cast является проверка его выполнения компилятором во время компиляции, что усложняет возможность возникновения непреднамеренных проблем.

Операторы явного преобразования типов данных



Преобразования типов данных следует избегать, если это вообще возможно, поскольку всякий раз, когда выполняется подобное изменение, есть вероятность возникновения непредвиденных проблем.

Но очень часто случаются ситуации, когда этого не избежать. Поэтому в таких случаях лучше использовать оператор **static_cast** вместо конвертации **C-style**.

Приоритет операторов

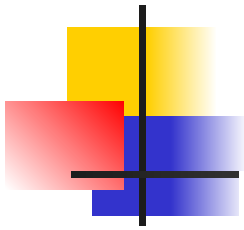
- Приоритет устанавливает иерархию одной группы операторов над другой, когда должно быть вычислено арифметическое выражение
- Это обращает внимание на порядок, в котором вычисляются операторы в C
- Приоритет операторов может быть изменен с помощью заключения нужного выражения в скобки

Класс операторов	Операторы	Ассоциативность
Унарная	- ++ --	Справа налево
Бинарная	^	Слева направо
Бинарная	* / %	Слева направо
Бинарная	+ -	Слева направо
Бинарная	=	Справа налево

Приоритет операторов

- Приоритет устанавливает иерархию одной группы операторов над другой, когда должно быть вычислено арифметическое выражение
- Это обращает внимание на порядок, в котором вычисляются операторы в C
- Приоритет операторов может быть изменен с помощью заключения нужного выражения в скобки

Класс операторов	Операторы	Ассоциативность
Унарная	- ++ --	Справа налево
Бинарная	* / %	Слева направо
Бинарная	+ -	Слева направо
Бинарная	=	Справа налево



Приоритет операторов

ПРИМЕР: -8 * 4 % 2 - 3

Порядок	Выполняемая операция	Результат
1.	-8 (минус)	отрицание 8
2.	- 8 * 4	32
3.	- 32 % 2	16
4.	16-3	13

Приоритет между сравнением операторов

Всегда вычисляется слева направо

Приоритет для логических операторов-1

Приоритет	Оператор
1	NOT
2	AND
3	OR

В частных случаях со сложной структурой логические операторы вычисляются справа налево

Приоритет для логических операторов -2

Примите во внимание следующее выражение:

False OR True AND NOT False AND True

Это выражение вычисляется, как показано ниже:

False OR True AND [NOT False] AND True

NOT имеет высший приоритет.

False OR True AND [True AND True]

AND является оператором с высоким приоритетом и операторы с одинаковы
приоритетом вычисляются справа налево

False OR [True AND True]

[False OR True]

True

Приоритет среди разных типов операторов-1

Когда равенство используется для того, чтобы связать боле одного типа операторов, требуется ввести правило приоритетов для различных типов операторов

Приоритет	Тип оператора
1	Арифметический
2	Сравнение
3	Логический

Приоритет среди разных типов операторов -2

Примите во внимание следующий пример:

2*3+4/2 > 3 AND 3<5 OR 10<9

Ход вычислений показан ниже:

[2*3+4/2] > 3 AND 3<5 OR 10<9

Первыми вычисляются арифметические операторы:

[[2*3]+[4/2]] > 3 AND 3<5 OR 10<9

[6+2] >3 AND 3<5 OR 10<9

[8 >3] AND [3<5] OR [10<9]

Приоритет среди разных типов операторов -3

Следующими вычисляются операторы сравнения, а поскольку они все имеют одинаковый приоритет, они вычисляются слева на право

True AND True OR False

Последними вычисляются вычисляются логические операторы. AND имеет более высокий приоритет, чем OR

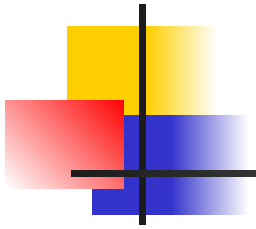
[True AND True] OR False

True OR False

True

Смена приоритета-1

- Скобки ( ) имеют высший приоритет
- Приоритет операторов может быть изменен используя скобки ( )
- Операторы низкого приоритета заключенные в скобки получают высший приоритет и исполняются первыми
- В случае вложенных скобок ((( ))), внутренние скобки будут исполняться первыми
- В случае вложенных скобок вычисления выполняются слева направо



Смена приоритета -2

Примите во внимание следующий пример:

5+9*3^2-4 > 10 AND (2+2^4-8/4 > 6 OR (2<6 AND 10>11))

Решение:

1. 5+9*3^2-4 > 10 AND (2+2^4-8/4 > 6 OR (True AND False))

Внутренние скобки обладают приоритетом над другими операторами, а действия вне скобок выполняются по обычным правилам

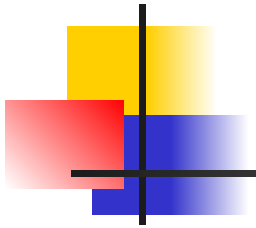
2. 5+9*3^2-4 > 10 AND (2+2^4-8/4 > 6 OR False)

Смена приоритета

3. 5+9*3^2-4 >10 AND (2+16-8/4 > 6 OR False)

   Затем выполняются действия во внешних скобках

4. 5+9*3^2-4 > 10 AND (2+16-2 > 6 OR False)
5. 5+9*3^2-4 > 10 AND (18-2 > 6 OR False)
6. 5+9*3^2-4 > 10 AND (16 > 6 OR False)
7. 5+9*3^2-4 > 10 AND (True OR False)
8. 5+9*3^2-4 > 10 AND True



Смена приоритета

9. 5+9*9-4>10 AND True

Выражение слева выполняется по обычным правилам

10. 5+81-4>10 AND True
11. 86-4>10 AND True
12. 82>10 AND True
13. True AND True
14. **True**