2. ВЫРАЖЕНИЯ

Выражение – это сочетание различных операндов и операций, например: a + b a/b c << d и т. д.

2.1. Операция и выражение присваивания

Операция присваивания обозначается символом '='

Простейший вид операции присвоения: **v = e**

Здесь **v** – любое выражение, которое может принимать значение, **e** – произвольное выражение.

Операция присвоения выполняется справа налево, т. е. сначала вычисляется значение выражения **e**, а затем это значение присваивается левому операнду **v**. Левый операнд в операции присваивания должен быть так называемым **адресным выражением**, которое иначе называют ℓ**-value**. Примером адресного, или именующего, выражения является имя переменной.

Не является ℓ-value, например, выражение a + b.

Адресным выражением никогда не являются константы.

В языке C++ операция присваивания образует выражение присваивания, то есть a = b

означает не только засылку в **a** значения **b**, но и то, что a = b является выражением, значением которого является левый операнд после присвоения. Отсюда следует, что возможна, например, такая запись: a = b = c = d = e + 2;

Итак, результатом выражения присваивания является его левый операнд. Если тип правого операнда не совпадает с типом левого, то значение справа преобразуется к типу левого операнда (если это возможно). При этом может произойти потеря значения, например:

```
int i; char ch; i =
3.14; ch = 777;
```

Здесь **i** получает значение 3, а значение 777 слишком велико, чтобы быть представленным как **char**, поэтому значение **ch** будет зависеть от способа, которым конкретная реализация производит преобразование из большего в меньший целый тип.

Существует так называемая комбинированная операция присваивания вида **a оп= b**

Здесь **оп** – знак одной из бинарных операций:

+ - * / % >> << & | ^ && ||.

Присваивание

a оп= b эквивалентно **a = a оп b**, за исключением того, что адресное выражение вычисляется только один раз.

Примеры:

a += 2 *означает* a = a + 2

bottom_count[2*i + 3*j + k] = bottom_count[2*i + 3*j + k]*=2 *означает* bottom_count[2*i + 3*j + k]*2

s/= a *означает* s = s/a

Результатом операции присваивания является ее левый операнд; следовательно, ее результат – адресное выражение и поэтому возможна запись

(a = b) += c;

Это эквивалентно следующим двум операторам:

a = b; a = a + c;

2.2. Арифметические операции

Бинарными арифметическими операциями являются + - * / %.

Существуют также унарные операции + и -.

При делении **целых** дробная часть **отбрасывается**.

Так, 10 / 3 дает 3, в то время как 10 / 3.0 дает 3.33333...

Операция a % b применяется только к целым операндам и ее результае – остаток от деления a на b:

10 % 3 дает 1, 2
% 3 дает 2, 1
2% 2 дает 0.

2.3. Операции отношения

В языке C++ определены следующие операции отношения (или сравнения):

Операция	Символ	Пример	Результат операции
Больше	>	x > y	true, если x больше y, в противном случае – false
Меньше	<	x < y	true, если x меньше y, в противном случае – false
Больше или равно	>=	x >= y	true, если x больше или равно y, в противном случае – false
Меньше или равно	<=	x <= y	true, если x меньше или равно y, в противном случае – false
Равно	==	x == y	true, если x равно y, в противном случае – false
Не равно	!=	x != y	true, если x не равно y, в противном случае – false

Операции отношения => > <= < имеют одинаковый приоритет. Непосредственно за ними по уровню старшинства следуют операции == (равно), != (не равно) с одинаковым приоритетом.

Операции отношения младше арифметических операций, так что выражения типа i < lim + 3 понимаются как i < (lim + 3).

2.4. Логические операции

К логическим операциям относятся:

- унарная операция логическое НЕ, **!** (отрицание);
- бинарная операция логическое И, **&&** (конъюнкция);
  бинарная операция логическое ИЛИ, **||** (дизъюнкция).

Операнды логических операций могут быть целых, плавающих и некоторых других типов, при этом в каждой операции могут участвовать операнды различных типов.

Операнды логических выражений вычисляются слева направо.

Результатом логической операции является **false** или **true** типа **bool**.

Операция **!операнд** дает 0, если операнд ненулевой, и 1 – если операнд равен нулю.

Операция && (И-логическое, логическое умножение) дает значение 1, если оба операнда имеют ненулевое значение. Если один из операндов

равен 0, то результат также равен 0. Если значение первого операнда равно 0, то второй операнд не вычисляется.

Операция || (ИЛИ-логическое, логическое сложение) вырабатывает значение 0, если оба операнда равны 0. Если какой-нибудь из операндов имеет ненулевое значение, то результат операции равен 1. Если первый операнд имеет ненулевое значение, то второй операнд не вычисляется.

По приоритету эти операции распределены так: !, &&, ||.

2.5. Побитовые операции

К побитовым, или поразрядным, операциям относятся:

- операция поразрядного И &;
- операция поразрядного ИЛИ |;
- операция поразрядного исключающего ИЛИ ^;
- унарная операция поразрядного отрицания (дополнение) ~.

Операнды поразрядных операций могут быть любого целого типа.

Операция & сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если оба соответствующих бита единицы, то соответствующий бит результата устанавливается в 1, в противном случае в 0.

Операция | сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если любой из них или оба равны 1, то соответствующий бит результата устанавливается в 1, в противном случае в 0.

Операция ^. Если один из сравниваемых битов равен 0, а другой равен 1, то соответствующий бит результата устанавливается в 1, в противном случае, т. е. когда оба бита равны 1 или оба равны 0, бит результата устанавливается в 0.

Операция ~ меняет в битовом представлении операнда 0 на 1, а 1 – на 0.

Побитовая операция & часто используется для маскирования некоторого множества битов. Например,

```
C = N & 0177
```

передает в **C** семь младших битов **N**, полагая остальные равными 0. (С первого нуля в C++ начинаются восьмеричные константы; с **0X** – шестнадцатеричные константы.)

Пусть N равно 642. Приведем побитовое представление N, восьмеричной константы 0177 и результата C:

N	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0
0177 C	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Операция | используется для включения битов:

C = N | MASK устанавливает в 1 те биты в **N**, которые

равны 1 в **MASK**.

Еще примеры:

short a = 0x45ff, b = 0x00ff;
short c; c = a ^ b; // c:
0x4500 c = a | b; // c:
0x45ff c = a & b; // c:
0x00ff c = ~ a; // c: -
0x3a00 c = ~ b; // c:
-0x7f00

Этот фрагмент программы можно проиллюстрировать так:

a b c = a ^	0	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1
b c = a | b	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
c = a & b	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0
c = ~b	0	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1
C	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
помощью	1	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0

операции & можно определить остаток от деления операнда типа unsigned int на 2, 4, 8, 16 и т. д. Для этого достаточно применить операцию & к делимому с масками 0x01, 0x03, 0x07, 0x0f,
0x1f и т. д.

Например, 7&0x03 дает 3.

Другими словами, выделяются младшие биты числа, а остальные устанавливаются в 0.

2.6. Сдвиги

Операции сдвига << >> осуществляют, соответственно, сдвиг влево и вправо своего левого операнда на число битовых позиций, заданных правым операндом. Таким образом, X << 2 сдвигает X влево на 2 позиции, заполняя освобождающиеся биты нулями, что эквивалентно

умножению на 4. Сдвиг вправо величины без знака сопровождается дополнением старших битов нулями.

Сдвиг вправо такой величины на **n** битов эквивалентен целочисленному делению левого операнда на 2 в степени **n**.

Так,
5 << 3 дает 40; 7

>> 2 дает 1.

Отметим, что правый операнд должен быть константным выражением, т. е. выражением, включающим в себя только константы. Если правый операнд отрицателен или он больше, или равен числу битов левого операнда, то результат сдвига не определен. Типом результата операции сдвига является тип левого операнда.

2.7. Операции автоувеличения и автоуменьшения ++ и --

Эти операции являются унарными операциями присваивания. Они, соответственно, увеличивают или уменьшают значение операнда на 1. Операнд должен быть целого или плавающего типа (или типа указатель) и быть не константным адресным выражением (т. е. без слова **const** в описании). Тип результата соответствует типу операнда.

Префиксная форма операций:

++операнд --операнд

Постфиксная форма:

операнд++ операнд--.

Если знак операции стоит перед **операндом**, результатом операции является **увеличенное или уменьшенное** значение **операнда**. При этом результат является адресным выражением (*l*-value).

Если знак операции стоит после **операнда**, значением выражения будет значение **операнда**. **После использования** этого результата значение операнда увеличивается или уменьшается. Результат постфиксной формы этих операций не является *l*-value. Примеры:

```
int i = 0, j = 0, k, l;
k = ++i;              // Здесь k = 1 и i стало равно 1;
l  = j++;             // l = 0 , а j стало равно 1;
--k;                    // k = 0;
++j;                    // j стало равно 2.
```

Иначе говоря, результат выполнения оператора

k = ++i;

тот же, что и в последовательности операторов i

= i + 1; k = i;

А результат оператора k

= i++;

такой же, как и в последовательности

k = i; i = i + 1;

2.8. Тернарная, или условная, операция

Тернарная операция, т. е. операция с тремя операндами, имеет форму

операнд1 ? операнд2 : операнд3

Первый операнд может быть целого или плавающего типа (а также указателем, ссылкой или элементом перечисления). Для этой операции важно, является значение первого операнда нулем или нет. Если **операнд1** не равен 0, то вычисляется **операнд2** и его значение является результатом операции. Если **операнд1** равен 0, то вычисляется **операнд3** и его значение является результатом операции. Заметим, что вычисляется либо **операнд2**, либо **операнд3**, но не оба. Пример: max = a <= b ? b : a;

Здесь переменной max присваивается максимальное значение из переменных a и b.

Если в условной операции **операнд2** и **операнд3** являются адресными выражениями, то тернарная операция может стоять слева от знака присваивания:

a < b ? a: b = c * x + d;

Здесь значение выражения c * x + d присваивается меньшей из двух переменных a и b.

2.9. Операция следования

Символом операции следования является **,** (запятая). Выражения, разделенные этим символом, выполняются слева направо строго в том порядке, в котором они перечислены.

Результатом этой операции является результат последнего выражения. Если оно является адресным выражением, то и результат операции также является адресным выражением.

Примеры:

int a = 3, b = 8, c; // здесь запятая – разделитель, а не операция; c = (a++, a+b); // значения **a** и **c** станут равны 4 и 12 соответственно; (b--, c)*= 3; // значения **b** и **c** станут равны 7 и 36 соответственно. Операция следования часто используется в операторе **for** (будет рассмотрен далее). В различные части этого оператора можно включить несколько выражений, например для параллельного изменения двух индексов. Это иллюстрируется функцией REVERSE(S), которая располагает строку S в обратном порядке на том же месте.

```
void reverse ( char s[ ] ){int c, i, j; for ( i
= 0, j = strlen(s) - 1; i < j; i++, j--){ c =
s[i]; s[i] = s[j]; s[j] = c;} }
```

В этом примере strlen(S) – функция, вычисляющая число символов в строке S (без символа '\0').

Запятые, которые разделяют аргументы функций, переменные в описаниях и т. д. не имеют отношения к операции **запятая** и не обеспечивают вычислений слева направо.

2.10. Приоритеты операций и порядок вычисления

Сведения об операциях C++ приведены в табл. 4.

Таблица 4

Приоритет операций и порядок их выполнения

Приоритет	Операция	Примечание	Порядок выполнения
1	::	Разрешение контекста	слева направо
2	-> . [ ] ( ) a++ a--	Извлечение Индексирование массива Вызов функции Постинкремент и постдекремент	слева направо

3	++a --a ! ~ - + & * new, delete sizeof (type)	Прединкремент и преддекремент Логическое НЕ и инверсия Унарный -, унарный + Получение адреса Разрешение указателя Работа с динамической памятью Определение размера Приведение к типу type	справа налево
4	->* .*	Извлечение	слева направо

Окончание табл. 4

Приоритет	Операция	Примечание	Порядок выполнения
5	* / %	Умножение, деление, целый остаток от целого деления	слева направо
6	+ -	Бинарные сложение и вычитание	слева направо
7	<< >>	Сдвиги	слева направо
8	< <= > =>	Сравнение	слева направо
9	== !=	Булевские операции равно и не равно	слева направо
10	&	Побитовое И	слева направо
11	^	Побитовое исключающее ИЛИ	слева направо
12	|	Побитовое ИЛИ	слева направо
13	&&	И-логическое	слева направо
14	||	ИЛИ-логическое	слева направо
15	?: = *= /= %= += и т. д.	Тернарная операция Присваивания	справа налево
16	throw	Операция генерации исключения	справа налево
17	,	Запятая (следование)	слева направо

Первый приоритет является наивысшим. Большинство операций выполняется слева направо. Например, выражение a + b + c интерпретируется как (a + b) + c.

Если нужно изменить порядок действий, то применяются круглые скобки.

Выражение 7.□a + b/-c интерпретируется как (7. □a) + (b/(-c)).

Можно изменить этот порядок:

7. (a + b)/(-c).

Отметим, что выражения, в которые входит одна из бинарных операций + & ^ |, могут перегруппировываться компилятором, даже если они заключены в круглые скобки. Для обеспечения нужного порядка вычисления можно использовать явные промежуточные вычисления. В C++ не фиксируется порядок вычисления операндов в выражении. Например, в c = sin(a x + b) + fabs(x);

может сначала быть вычислен первый операнд, а затем второй, а может быть и наоборот. В простых случаях это не имеет значения. Но если необходим определенный порядок, нужно вводить промежуточные переменные.