**11-12 ОАП Лк5 Операции и выражения. Правила формирования и вычисления выражений. Структура программы.**

**Выражения**

*Программа* оперирует с данными. Числа можно складывать, вычитать, умножать, делить. Из разных величин можно составлять *выражения*, результат вычисления которых – новая величина. Приведем примеры *выражений*:

X \* 12 + Y // значение X умножить на 12 и к результату прибавить значение Y

val < 3 // сравнить значение val с 3

-9 // константное выражение -9

*Выражение*, после которого стоит точка с запятой – это *оператор-выражение*. Его смысл состоит в том, что *компьютер* должен выполнить все действия, записанные в данном *выражении*, иначе говоря, вычислить *выражение*.

x + y – 12; // сложить значения x и y и затем вычесть 12

a = b + 1; // прибавить единицу к значению b и запомнить результат в переменной a

***Выражения*** – это переменные, функции и константы, называемые операндами, объединенные знаками операций. **Операции** могут быть ***унарными*** – с одним операндом, например, минус; могут быть *бинарные* – с двумя операндами, например *сложение* или *деление*. В *Си*++ есть даже одна операция с тремя операндами – условное *выражение*. Чуть позже мы приведем *список* всех операций языка *Си*++ для встроенных типов данных. Подробно каждая операция будет разбираться при описании соответствующего типа данных. Кроме того, ряд операций будет рассмотрен в разделе, посвященном определению операторов для классов. Пока что мы ограничимся лишь общим описанием способов записи *выражений*.

В типизированном языке, которым является *Си*++, у переменных и констант есть определенный тип. Есть он и у результата *выражения*. Например, *операции* *сложения* ( + ), *умножения* ( \* ), *вычитания* ( - ) и *деления* ( / ), примененные к целым числам, выполняются по общепринятым математическим правилам и дают в результате целое *значение*. Те же *операции* можно применить к вещественным числам и получить вещественное *значение*.

Операции ***сравнения***: больше ( > ), меньше ( < ), равно ( == ), не равно ( != ) сравнивают значения чисел и выдают логическое значение: истина ( true ) или ложь ( false ).

**Операция присваивания**

***Присваивание*** – это тоже операция, она является частью *выражения*. Значение правого операнда присваивается левому операнду.

x = 2; // переменной x присвоить значение 2

cond = x < 2; // переменной cond присвоить значение true, если x меньше 2,

// в противном случае присвоить значение false

3 = 5; // ошибка, число 3 неспособно изменять свое значение

Последний пример иллюстрирует требование к левому операнду *операции присваивания*. Он должен быть способен хранить и изменять свое *значение*. Переменные, объявленные в программе, обладают подобным свойством. В следующем фрагменте программы

int x = 0;

x = 3;

x = 4; x = x + 1;

вначале объявляется *переменная* x с начальным значением 0. После этого *значение* x изменяется на 3, 4 и затем 5. Опять-таки, обратим внимание на последнюю строчку. При вычислении *операции присваивания* сначала вычисляется правый *операнд*, а затем левый. Когда вычисляется *выражение* x + 1, *значение переменной* x равно 4. Поэтому *значение* *выражения* x + 1 равно 5. После вычисления *операции присваивания* (или, проще говоря, после *присваивания* ) *значение переменной* x становится равным 5.

У *операции присваивания* тоже есть результат. Он равен значению левого операнда. Таким образом, *операция присваивания* может участвовать в более сложном *выражении*:

z = (x = y + 3);

В приведенном примере переменным x и z присваивается *значение* y + 3.

Очень часто в программе приходится *значение переменной* увеличивать или уменьшать на единицу. Для того чтобы сделать эти действия наиболее эффективными и удобными для использования, применяются предусмотренные в *Си*++ специальные знаки операций: ++ (увеличить на единицу) и -- (уменьшить на единицу). Существует две формы этих операций: префиксная и постфиксная. Рассмотрим их на примерах.

int x = 0; ++x;

*Значение* x увеличивается на единицу и становится равным 1.

--x;

*Значение* x уменьшается на единицу и становится равным 0.

int y = ++x;

*Значение* x опять увеличивается на единицу. Результат *операции* ++ – новое *значение* x, т.е. переменной y присваивается *значение* 1.

int z = x++;

Здесь используется *постфиксная запись* *операции* увеличения на единицу. *Значение переменной* x до выполнения *операции* равно 1. Сама операция та же – *значение* x увеличивается на единицу и становится равным 2. Однако результат постфиксной *операции* – это *значение* аргумента до увеличения. Таким образом, переменной z присваивается *значение* 1. Аналогично, результатом постфиксной *операции* уменьшения на единицу является начальное *значение* операнда, а префиксной – его конечное *значение*.

Подобными мотивами оптимизации и сокращения записи руководствовались создатели языка *Си* (а затем и *Си*++), когда вводили новые знаки операций типа "выполнить операцию и присвоить". Довольно часто одна и та же *переменная* используется в левой и правой части *операции присваивания*, например:

x = x + 5;

y = y \* 3;

z = z – (x + y);

В *Си*++ эти *выражения* можно записать короче:

x += 5;

y \*= 3;

z -= x + y;

Т.е. *запись* oper= означает, что левый *операнд* вначале используется как левый *операнд* *операции* oper, а затем как правый *операнд* *операции присваивания* результата *операции* oper. Кроме краткости *выражения*, такая *запись* облегчает оптимизацию программы компилятором.

### Все операции языка Си++

Наряду с общепринятыми арифметическими и *логическими операциями*, в языке *Си*++ имеется набор операций для работы с битами – поразрядные И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ и НЕ, а также *сдвиги*.

Особняком стоит операция sizeof. Эта операция позволяет определить, сколько памяти занимает то или иное *значение*. Например:

sizeof(long); // сколько байтов занимает тип long

sizeof (b); // сколько байтов занимает переменная b

Операция sizeof в качестве аргумента берет имя типа или *выражение*. *Аргумент* заключается в скобки (если *аргумент* – *выражение*, скобки не обязательны). Результат *операции* – *целое число*, равное количеству байтов, которое необходимо для хранения в памяти заданной величины.

Ниже приводятся все *операции* языка *Си*++.

#### Арифметические операции

+ сложение - вычитание \* умножение / деление

Операции *сложения*, *вычитания*, *умножения* и *деления* целых и вещественных чисел. Результат операции – число, по типу соответствующее большему по разрядности операнду. Например, *сложение* чисел типа short и long в результате дает число типа long.

% остаток

Операция нахождения *остатка от деления* одного целого числа на другое. Тип результата – целое число.

- минус + плюс

Операция "минус" – это *унарная операция*, при которой знак числа изменяется на противоположный. Она применима к любым числам со знаком. Операция "плюс" существует для симметрии. Она ничего не делает, т.е. примененная к целому числу, его же и выдает.

++ увеличить на единицу, префиксная и постфиксная формы

-- уменьшить на единицу, префиксная и постфиксная формы

Эти операции иногда называют ***"автоувеличением"*** (инкремент) и ***"автоуменьшением"*** (декремент). Они увеличивают (или, соответственно, уменьшают) операнд на единицу. Разница между постфиксной (знак операции записывается после операнда, например x++ ) и префиксной (знак операции записывается перед операндом, например --y ) операциями заключается в том, что в первом случае результатом является значение операнда до изменения на единицу, а во втором случае – после изменения на единицу.

#### Операции сравнения

== равно != не равно < меньше > больше

<= меньше или равно >= больше или равно

Операции *сравнения*. Сравнивать можно операнды любого типа, но либо они должны быть оба одного и того же встроенного типа ( *сравнение* на равенство и неравенство работает для двух величин любого типа), либо между ними должна быть определена соответствующая операция *сравнения*. Результат – логическое значение true или false.

#### Логические операции

&& логическое И || логическое ИЛИ ! логическое НЕ

***Логические операции*** конъюнкции, дизъюнкции и отрицания. В качестве операндов выступают логические значения, результат – тоже логическое значение **true** или **false** .

#### Битовые операции

& битовое И | битовое ИЛИ ^ битовое ИСКЛЮЧАЮЩЕЕ ИЛИ ~ битовое НЕ

***Побитовые операции*** над целыми числами. Соответствующая операция выполняется над каждым битом операндов. Результатом является целое число.

<< сдвиг влево

>> сдвиг вправо

Побитовый ***сдвиг*** левого операнда на количество разрядов, соответствующее значению правого операнда. Результатом является целое число.

#### Условная операция

операнд1?операнд2:операнд3

***Тернарная операция***; если значение первого операнда – истина, то результат – второй операнд; если ложь – результат – третий операнд. Первый операнд должен быть логическим значением, второй и третий операнды могут быть любого, но одного и того же типа, а результат будет того же типа, что и третий операнд.

#### Последовательность

, последовательность

Выполнить *выражение* до запятой, затем *выражение* после запятой. Два произвольных *выражения* можно поставить рядом, разделив их запятой. Они будут выполняться **последовательно**, и результатом всего *выражения* будет результат последнего *выражения*.

#### Операции присваивания

= присваивание

Присвоить значение правого операнда левому. Результат *операции присваивания* – это значение правого операнда.

+=, -=, \*=, /=, %=, |=, &=, ^=, <<=, >>=

выполнить операцию и присвоить

Выполнить соответствующую операцию с левым операндом и правым операндом и присвоить результат левому операнду. Типы операндов должны быть такими, что, во-первых, для них должна быть определена соответствующая арифметическая операция, а во-вторых, результат может быть присвоен левому операнду.

### Домашнее задание Порядок вычисления выражений

### СТРУКТУРА ПРОГРАММЫ C++

Программа на языке С/С++ состоит из:

1. Директив процессора
2. Описаний
3. Функций

Главная функция носит имя main и должна быть обязательно.  
Общую структуру программы на языке С/С++ можно представить следующим образом:

|  |  |
| --- | --- |
|  | директивы препроцессора //1 |
|  | описание глобальных переменных //2 |
|  | тип\_результата main (параметры) //3 |
|  | { |
|  | тело главной функции |
|  | } |
|  | тип\_результата F1 (параметры 1) { |
|  | тело функции F1; |
|  | } |
|  | тип\_результата F2 (параметры 2) { |
|  | тело функции F2; |
|  | } |
|  | тип\_результата FN (параметры N) { |
|  | тело функции FN; |
|  | } |

*Директивы препроцессора* – определяют действия по преобразованию программы перед компиляцией, а также включают инструкции, которым компилятор следует во время компиляции;  
*Объявления* – описания переменных, функций, структур, классов и типов данных;  
*Описание функций*, выполняемых в проекте. В языке С++ существует только один вид подпрограмм – подпрограмма-функция.  
Функция состоит из *заголовка* и *тела функции*.  
Заголовок состоит из:

1. Типа результата, возвращаемого функцией;
2. Имени функции;
3. Списка параметров, необходимых для выполнения функции.

**Тело функции** заключается в операторные скобки {}, и содержит *описание локальных данных*и *операторы функции*.

где, *тип\_результата* – это тип того значения, которое функция должна вычислить (если функция не должна возвращать значение, указывается тип void);  
*имя\_функции*– имя, с которым можно обращаться к этой функции;  
*параметры* – список её аргументов (может отсутствовать).  
Точка с запятой (;) является разделителем между описаниями и операторами.

### Директивы процессора

Понятие препроцессора является одним из ключевых в языке С++.  
*Препроцессор* – это программа, действующая как фильтр на этапе компиляции.  
*Исходная программа => Препроцессор => Расширенная исходная программа => Компилятор*

Перед тем, как попасть на вход компилятора, исходная программа проходит через препроцессор.

Основному тексту программы предшествуют *директивы препроцессора* (preprocessor directive), которые начинаются с символа решётки #, не являются выражением языка С++ (и потому не заканчивается точкой с запятой).

*Команды (директивы) препроцессора* представляют собой инструкции, записанные в исходном тексте программы, используемые для того, чтобы облегчить модификацию программ и сделать их независимыми от особенностей различных реализаций компилятора.  
Например:

|  |  |
| --- | --- |
|  | #include <имя\_файла> |

Директивы могут быть записаны в любом месте исходного файла, но их действие распространяется только от точки программы, в которой они записаны, до конца исходного файла.  
Файлы, которые появляются в директивах, обычно заканчиваются на .h, что означает файл заголовков (header file). Файлы заголовков содержат объявления констант, переменных и функций, необходимы для работы программы.

|  |  |
| --- | --- |
|  | #include <stdio.h> /\*функции стандартного ввода-вывода\*/ |
|  | #include <conio.h> /\*функции консольного ввода-вывода\*/ |
|  | #include <math.h> /\*стандартные математические функции\*/ |

### Директива #include

Директива препроцессора #include позволяет включать в текст программы, написанной программистом, тексты других программ.  
Эта директив имеет две формы записи:

|  |  |
| --- | --- |
|  | #include <имя\_включаемого\_файла> |
|  | #include “имя\_включаемого\_файла” |

В случае, когда имя\_включаемого\_файла (файла включений) записывается в угловых кавычках, то поиск этого файла осуществляется в стандартных системных каталогах (каталоге подключаемых файлов (include directory), то есть в месте, где компьютерная система хранит все доступные для использования файлы заголовков).  
Например:

|  |  |
| --- | --- |
|  | #include <stdio.h> //фрагмент на С |
|  | или |
|  | #include <iostream> //фрагмент на С++ |

Если же имя\_включаемого\_файла указано в кавычках, то сначала поиск осуществляется в текущем каталоге и лишь затем в системных.

|  |  |
| --- | --- |
|  | #include “file2.h” //Файл, содержащий объявления переменных и типов, |
|  | //созданных программистом и используемых в программе |
|  | #include “file2.cpp” //Файл, содержащий, например, объявления или описания функций, //используемых в текущем модуле(файле) |

Имя\_включаемого\_файла может также содержать полный путь для его поиска, что наблюдается редко, так как это предполагает жёсткую связь программы с конфигурацией файловой системы конкретной ЭВМ.  
Пример:

|  |  |
| --- | --- |
|  | #include <stdio.h> /\* функции стандартного ввода-вывода \*/ |
|  | #include <conio.h /\* функции консольного ввода-вывода \*/ |
|  | #include <math.h>/\* стандартные математические функции \*/ |
|  | #include <iostream> // потоковый ввод/вывод |

### Директивы #define, #undef

Директива препроцессора **#define** позволяет заменять все вхождения указанного идентификатора какой-то определенной последовательностью символов.  
Удобно использовать для объявления и определения констант, когда одно и то же значение (например, длина массива) используется в разных местах программы.

|  |  |
| --- | --- |
|  | #define идентификатор последовательность символов |

Например:

|  |  |
| --- | --- |
|  | #define lin\_arr 40 |
|  | void main(void) { |
|  | int arr[lin\_arr], i; |
|  | … |
|  | for (i=0; i<lin\_arr; i++) |
|  | arr[i]=i+1; |
|  | … |

В результате препроцессорной обработки получится сле­дующий текст программы:

|  |  |
| --- | --- |
|  | void main(void) |
|  | { |
|  | int arr[40], i; |
|  | for (i=0; i<40; i++) |
|  | arr[i]=i+1; |
|  | … |

Изменив значение **lin\_arr** на любое другое только в директиве ***#define***, автоматически изменятся и длина массива, и условие окончания цикла.  
Также удобно использовать директиву #define для опре­деления и описания функций (func - функция, выбирающая ми­нимальное из двух чисел):