

# Системы счисления

**Система счисления** — это способ записи (представления) чисел.

Системы счисления бывают:

- **непозиционными** (в этих системах значение цифры не зависит от ее позиции — положения в записи числа);
- **позиционными** (значение цифры зависит от позиции).

**Непозиционная** — самая древняя, в ней каждая цифра числа имеет величину, не зависящую от её позиции (разряда).

**Позиционная система** — значение каждой цифры зависит от её позиции (разряда) в числе. Например, привычная для нас 10-я система счисления — позиционная. Рассмотрим число 453. Цифра 4 обозначает количество сотен и соответствует числу 400, 5 — кол-во десятков и аналогично значению 50, а 3 — единиц и значению 3. Как видим — чем больше разряд — тем значение выше. Итоговое число можно представить, как сумму  $400+50+3=453$ .

**Десятичная система счисления**

Это одна из самых распространенных систем счисления. Именно её мы используем, когда называем цену товара и произносим номер автобуса. В каждом разряде (позиции) может использоваться только одна цифра из диапазона от 0 до 9. Основанием системы является число 10.

**Двоичная система счисления**

Эта система, в основном, используется в вычислительной технике. Почему не стали использовать привычную нам 10-ю? Первую вычислительную машину создал Блез Паскаль, использовавший в ней десятичную систему, которая оказалась неудобной в современных электронных машинах, поскольку требовалось производство устройств, способных работать в 10 состояниях, что увеличивало их цену и итоговые размеры машины. Этих недостатков лишены элементы, работающие в 2-ой системе. Двоичная позиционная система счисления имеет основание 2 и использует для записи числа 2 символа (цифры): 0 и 1. В каждом разряде допустима только одна цифра — либо 0, либо 1.

Примером может служить число 101. Оно аналогично числу 5 в десятичной системе счисления. Для того, чтобы перевести из 2-й в 10-ю необходимо умножить каждую цифру двоичного числа на основание системы счисления, в данном случае “2”, возведенное в степень, равную разряду.

Таким образом, число  $101_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 4 + 0 + 1 = 5_{10}$ .

Это же правило перевода работает для перевода чисел из ЛЮБОЙ системы счисления в десятичную. В этом убедимся на примере 8-ой и 16-ой СС.

## Восьмеричная система счисления

8-я система счисления, как и двоичная, часто применяется в цифровой технике. Имеет основание 8 и использует для записи числа цифры от 0 до 7.

Пример восьмеричного числа: 254. Для перевода в 10-ю систему необходимо каждый разряд исходного числа умножить на  $8^n$ , где  $n$  — это номер разряда. Получается, что  $254_8 = 2 \cdot 8^2 + 5 \cdot 8^1 + 4 \cdot 8^0 = 128 + 40 + 4 = 172_{10}$ .

## Шестнадцатеричная система счисления

Шестнадцатеричная система широко используется в современных компьютерах, например при помощи неё указывается цвет: #FFFFFF — белый цвет. Рассматриваемая система имеет основание 16 и использует для записи числа: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, где буквы равны 10, 11, 12, 13, 14, 15 соответственно.

## Преобразование в десятичную систему счисления

Имеется число  $a_1a_2a_3$  в системе счисления с основанием  $b$ . Для перевода в 10-ю систему необходимо каждый разряд числа умножить на  $b^n$ , где  $n$  — номер разряда. Таким образом,  $(a_1a_2a_3)_b = (a_1 \cdot b^2 + a_2 \cdot b^1 + a_3 \cdot b^0)_{10}$ .

Пример:  $101_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 4 + 0 + 1 = 5_{10}$

## Преобразование из десятичной системы счисления в другие

Целая часть:

1. Последовательно делим целую часть десятичного числа на основание системы, в которую переводим, пока десятичное число не станет равно нулю.
2. Полученные при делении остатки являются цифрами искомого числа. Число в новой системе записывают, начиная с последнего остатка.

Дробная часть:

1. Дробную часть десятичного числа умножаем на основание системы, в которую требуется перевести. Отделяем целую часть. Продолжаем умножать дробную часть на основание новой системы, пока она не станет равной 0.
2. Число в новой системе составляют целые части результатов умножения в порядке, соответствующем их получению.

Пример: переведем  $15_{10}$  в восьмеричную:

$$15 \setminus 8 = 1, \text{ остаток } 7$$

$$1 \setminus 8 = 0, \text{ остаток } 1$$

Записав все остатки снизу вверх, получаем итоговое число 17.

Следовательно,  $15_{10} = 17_8$ .

## Преобразование из двоичной в восьмеричную и шестнадцатеричную системы

Для перевода в восьмеричную — разбиваем двоичное число на группы по 3 цифры справа налево, а недостающие крайние разряды заполняем ведущими нулями. Далее преобразуем каждую группу, умножая последовательно разряды на  $2^n$ , где  $n$  — номер разряда.

В качестве примера возьмем число  $1001_2$ :  $1001_2 = 001\ 001 = (0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0) (0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0) = (0+0+1) (0+0+1) = 11_8$

Для перевода в шестнадцатеричную — разбиваем двоичное число на группы по 4 цифры справа налево, затем — аналогично преобразованию из 2-й в 8-ю.

## Преобразование из восьмеричной и шестнадцатеричной систем в двоичную

Перевод из восьмеричной в двоичную — преобразуем каждый разряд восьмеричного числа в двоичное 3-х разрядное число делением на 2 (более подробно о делении см. выше пункт “Преобразование из десятичной системы счисления в другие”), недостающие крайние разряды заполним ведущими нулями.

Для примера рассмотрим число  $45_8$ :  $45 = (100) (101) = 100101_2$

Перевод из 16-ой в 2-ю — преобразуем каждый разряд шестнадцатеричного числа в двоичное 4-х разрядное число делением на 2, недостающие крайние разряды заполняем ведущими нулями.

## Преобразование дробной части любой системы счисления в десятичную

Преобразование осуществляется также, как и для целых частей, за исключением того, что цифры числа умножаются на основание в степени “-n”, где  $n$  начинается от 1.

Пример:  $101,011_2 = (1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0), (0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3}) = (5), (0 + 0,25 + 0,125) = 5,375_{10}$

### Преобразование дробной части двоичной системы в 8- и 16-ую

Перевод дробной части осуществляется также, как и для целых частей числа, за тем лишь исключением, что разбивка на группы по 3 и 4 цифры идёт вправо от десятичной запятой, недостающие разряды дополняются нулями справа.

Пример:  $1001,01_2 = 001\ 001,010 = (0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0) (0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0), (0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) = (0+0+1) (0+0+1), (0+2+0) = 11,2_8$

### Преобразование дробной части десятичной системы в любую другую

Для перевода дробной части числа в другие системы счисления нужно обратить целую часть в ноль и начать умножение получившегося числа на основание системы, в которую нужно перевести. Если в результате умножения будут снова появляться целые части, их нужно повторно обращать в ноль, предварительно запомнив (записав) значение получившейся целой части. Операция заканчивается, когда дробная часть полностью обратится в ноль.

Для примера переведем  $10,625_{10}$  в двоичную систему:

$$0,625 \cdot 2 = 1,25$$

$$0,250 \cdot 2 = 0,5$$

$$0,5 \cdot 2 = 1,0$$

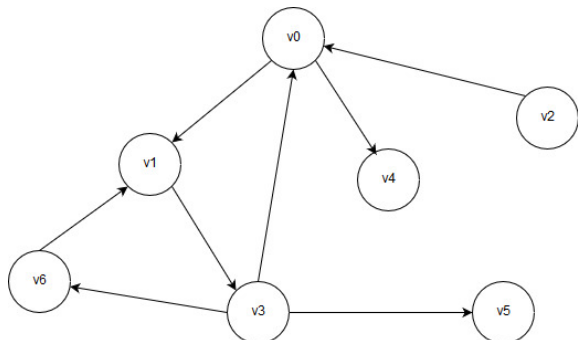
Записав все остатки сверху вниз, получаем

$$10,625_{10} = (1010), (101) = 1010,101_2$$

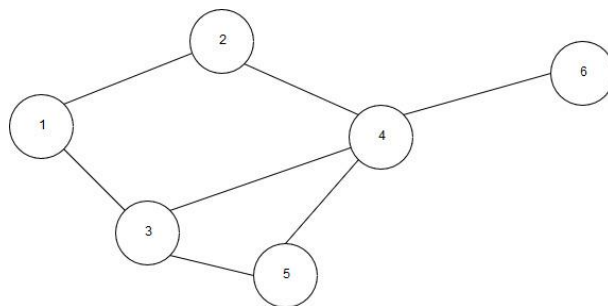
# Графы и матрицы

Графы делятся на **ориентированные и неориентированные**.

Ориентированный граф – такой граф, в котором можно двигаться от вершины к вершине только в одном направлении.

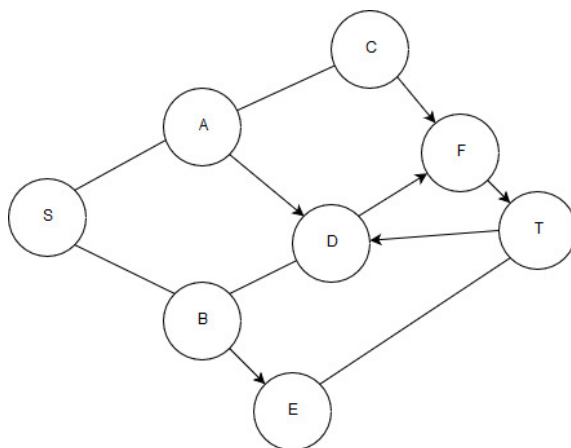


Ориентированный граф



Неориентированный граф

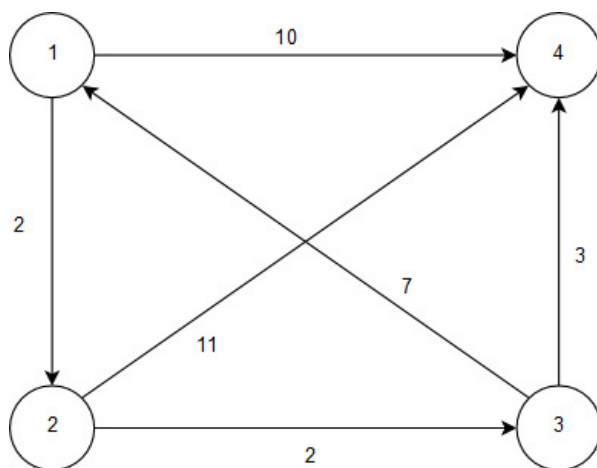
Граф, в котором присутствуют как ориентированные ребра, так и неориентированные, называется **смешанным**.



Смешанный граф

Кроме того, графы делятся на **взвешенные и невзвешенные**. Граф, в котором каждому ребру в соответствие поставлено некоторое числовое значение – вес, называется взвешенным графом. Если никакого числового значения ребрам не поставлено, то граф называется невзвешенным. Чаще всего, в названии графа указывают как его **ориентированность** или **неориентированность**, так и его **взвешенность** или **невзвешенность**.

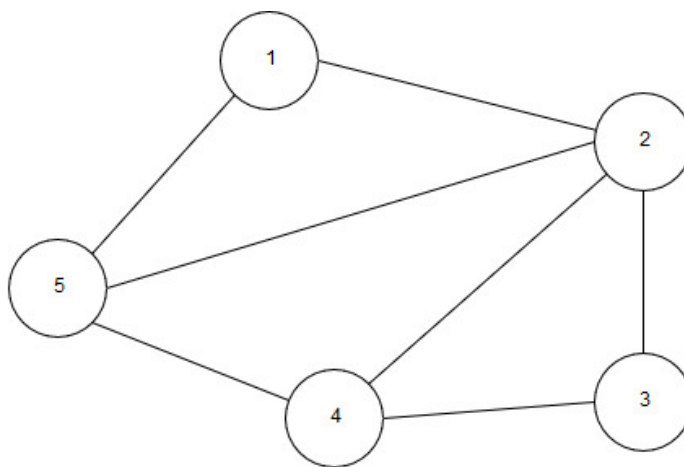
Граф, в котором между любой парой вершин существует, как минимум, один путь, называется **связным**. Если в графе существует хотя бы одна вершина, не связанная с другими, он называется **несвязным**.



Взвешенный связанный ориентированный граф

## Представление графов. Матрица смежности.

**Матрица смежности** графа – это способ представления графа в виде квадратной матрицы, в которой каждый элемент принимает одно из двух значений: 0 или 1 для невзвешенного графа. Значения 1 и 0 отображают существование ребра между вершинами.

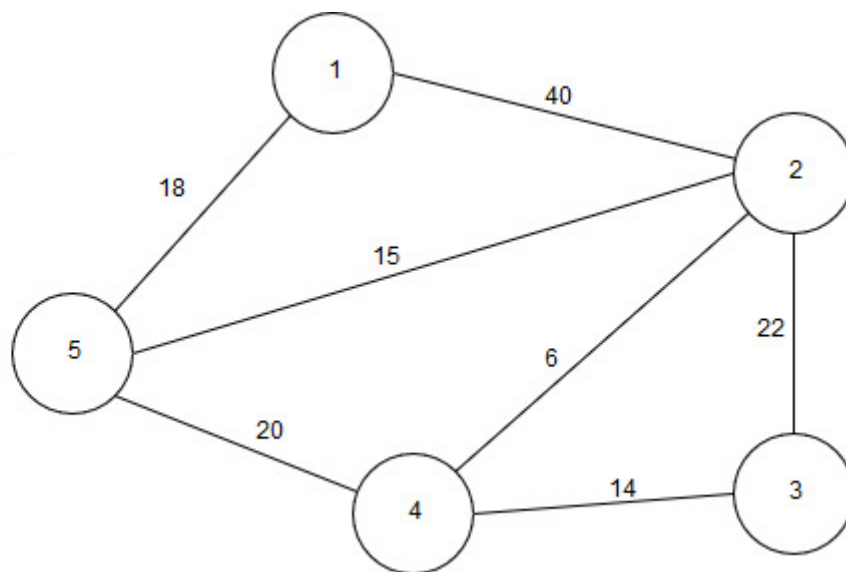


Матрица смежности для графа представлена следующим образом:

№ вершины	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

## Весовая матрица.

Для взвешенных графов **весовая матрица** не только отражает наличие ребра между двумя вершинами, но и вес данного ребра.



Весовая матрица будет выглядеть следующим образом:

№ вершины	1	2	3	4	5
1	0	40	0	0	18
2	40	0	22	6	15
3	0	22	0	14	0
4	0	6	14	0	22
5	18	15	0	20	0

# Программирование на Pascal.

Основная структура программы.

Правила языка Паскаль предусматривают единую для всех программ форму основной структуры:

```
Program <Имя программы>;  
<Раздел описаний>  
Begin  
<Тело программы>  
End.
```

Здесь слова Program, Begin и End являются служебными. Правильное и уместное употребление этих слов является обязательным.

Все объекты, не являющиеся зарезервированными в Паскале, наличие которых обусловлено инициативой программиста, перед первым использованием в программе должны быть описаны. Это производится для того, чтобы компьютер перед выполнением программы зарезервировал память под соответствующие объекты и поставил в соответствие этим участкам памяти идентификаторы.

Раздел описаний может состоять из пяти подразделов:

1. Описание меток (Label).
2. Описание типов (Type).
3. Описание констант (Const).
4. Описание переменных (Var).
5. Описание процедур и функций (Procedure, Function).

При отсутствии необходимости в каком-либо виде объектов, соответствующий подраздел может быть опущен.

Алфавит языка Паскаль составляют:

- 1) буквы латинского алфавита;
- 2) арабские цифры;
- 3) специальные знаки.

Использование символов первой группы чаще всего вопросов не вызывает, но свои тонкости здесь имеются. Во-первых, это употребление заглавных и строчных букв. Большинство существующих трансляторов не различают буквы разных регистров. Таким образом, записи "progRaM" и "PROGram" будем считать идентичными. Во-вторых, некоторые символы латиницы и кириллицы



совпадают по начертанию. Нельзя ли вместо буквы "К" латинской написать "К" русскую? Ответ: в тетради (если вы их сможете различить) - пожалуйста, в программе на ЭВМ - ни в коем случае. На вид они могут быть и похожи, но уж коды-то у них совершенно разные, а компьютер, как вам известно, оперирует внутри себя не буквами, а их числовыми кодами.

Наиболее часто употребляемым специальным символом является пробел (в значимых местах мы будем обозначать его в записях знаком "V"). Его использование связано с форматами основной структуры программы, разделов описаний, операторов. Не следует путать наличие пробела с отсутствием символа.

- конец программы, разделение целой и дробной частей вещественного числа (десятичная точка), разделение полей в переменной типа Record;
- , разделение элементов списков;
- .. указание диапазона;
- : используется в составе оператора присваивания, а также для указания формата вывода в операторе Writeln;
- ; отделяет один раздел программы от другого, разделяет операторы;
- ' используется для ограничения строковых констант;
- + \* / ( ) арифметические знаки (используются по своему назначению);
- < > знаки отношений;
- = используется в составе оператора присваивания, в разделах описаний констант и типов, используется как знак отношения (равно);
- @ имя специального оператора определения адреса переменной, подпрограммы;
- ^ используется для именования динамических переменных;
- { } ограничение комментариев в программе;
- [ ] заключают в себе индексы элементов массивов;
- \_ символ подчеркивания используется также как любая буква, например, в идентификаторах - вместо пробела;
- # обозначение символа по его коду;
- \$ обозначение директивы компилятора, обозначение шестнадцатеричного числа.

Существуют зарезервированные (базовые) типы в языке Паскаль, но, как далее вы убедитесь, есть также возможность создавать свои собственные, определяемые программистом типы переменных.

К базовым типам относятся:

тип целых чисел – Integer

тип "длинных" целых чисел – Longint

тип действительных (вещественных) чисел (то есть - с дробной частью) – Real

тип неотрицательных целых чисел от 0 до 255 – Byte

тип неотрицательных целых чисел от 0 до 65535 – Word

символьный тип – Char

строковый тип - String

логический тип - Boolean

Физически типы данных отличаются друг от друга количеством ячеек памяти (байтов), отводимых для хранения соответствующей переменной. Логическое же отличие проявляется в интерпретации хранящейся информации. Например, переменные типа Char и типа Byte занимают в памяти по одному байту. Однако в первом случае содержимое ячейки памяти интерпретируется как целое беззнаковое число, а во втором - как код (ASC) символа.

В отличие от констант, неименованных переменных не существует. Все используемые в программе переменные должны быть описаны в соответствующем разделе описания.

Оператор присваивания. Арифметические выражения.

Самым простым действием над переменной является занесение в нее величины соответствующего типа. Иногда говорят об этом, как о присвоении переменной конкретного значения. Такая команда (оператор) в общем виде выглядит на языке Паскаль следующим образом:

**<Имя переменной>:=<Выражение>;**

Выражение, указанное справа от знака ":", должно приводить к значению того же типа, какого и сама переменная, или типа, совместимого с переменной относительно команды присваивания. Например, переменной типа Real можно присвоить значение типа Integer или Word (впрочем, наоборот делать нельзя).

В состав арифметического выражения на языке Паскаль могут входить:

числовые константы; имена переменных; знаки математических операций; математические функции и функции, возвращающие число; открывающиеся и закрывающиеся круглые скобки.

Правила построения выражений напоминают математические с некоторыми уточнениями. Выражение записывается в одну строку (никакой многоэтажности), между операндами обязательно должен стоять знак операции (Запись "2x" - не допускается), знаки некоторых операций и названия некоторых функций отличны от привычных вам.

Операции:

- + сложение;
- вычитание;
- / деление;
- \* умножение;

MOD остаток от деления (записывается так: A MOD B; читается: остаток от деления A на B); эта операция применима только к целым числам;

DIV целочисленное деление (записывается так A DIV B; читается: результат деления A на B без дробной части); эта операция тоже применяется только для целых операндов.

Аргументы функций всегда записываются в круглых скобках:

SIN(X) sin x;

COS(X) cos x;

ARCTAN(X) arctg x;

ABS(X) абсолютное значение x (в математике -  $|x|$ );

SQR(X) возведение x в квадрат;

SQRT(X) извлечение квадратного корня;

TRUNC(X) отбрасывание дробной части x;

ROUND(X) округление x до ближайшего целого числа;

После выполнения второго оператора присваивания в участке памяти, отведенном под переменную R, окажется результат указанного выражения, однако, к сожалению, узнать его мы не сможем, поскольку пока не имеем возможности "заглянуть" в память машины, вывести значение переменной хотя бы на экран.

Составной оператор

Этот оператор, строго говоря, оператором не является. Дело в том, что также как арифметические действия иногда бывает необходимо заключать в скобки, последовательности команд (операторов) тоже иногда требуют объединения. Это позволяют сделать так называемые операторные скобки. Формат (общий вид) составного оператора таков:

```
Begin  
<Оператор 1>;  
<Оператор 2>;  
.....  
<Оператор N>  
End;
```

Возможно, такая структура напоминает вам основную структуру программы. Действительно, отличие только в том, что после End в конце составного оператора ставится точка с запятой, а в конце программы - точка. По своей сути вся программа представляет собой большой составной оператор.

Обратите внимание на то, что точка с запятой перед End может не ставиться.

Составной оператор предоставляет возможность выполнить произвольное количество команд там, где подразумевается использование только одного оператора. Как вы узнаете потом, такая необходимость встречается довольно часто.

операторы ввода (форматы операторов):

```
Read(<Список ввода>;  
Readln(<Список ввода>;
```

В таком формате эти команды позволяют вводить данные в переменные во время выполнения программы с клавиатуры. Элементами списка ввода могут быть имена переменных, которые должны быть заполнены значениями, введенными с клавиатуры.

Выполнение операторов ввода происходит так: ход программы приостанавливается, на экран выводится курсор, компьютер ожидает от пользователя набора данных для переменных, имена которых указаны в списке ввода. Пользователь с клавиатуры вводит необходимые значения в том порядке, в котором они требуются списком ввода, нажимает Enter. После этого набранные данные попадают в соответствующие им переменные и выполнение программы продолжается.

Примечание: данные при вводе разделяются пробелами.

Разница между работой процедур Read и Readln (от Read line) состоит в следующем: после выполнения Read значение следующего

данного считывается с этой же строчки, а после выполнения Readln - с новой строки.

Для вывода информации в Паскале также есть две команды:  
Write(<Список вывода>);  
Writeln(<Список вывода>);

Такой формат использования Write и Writeln позволяет выводить на экран монитора данные из списка вывода. Элементами списка вывода могут являться имена переменных, выражения, константы. Прежде чем вывести на экран компьютер значения выражений сначала вычислит. Элементы списка, также как и в операторах ввода, разделяются запятыми.

Различие между двумя операторами вывода таково: после выполнения оператора Writeln (от Write line) происходит переход на новую строку, а после выполнения инструкции Write, переход на новую строку не происходит и печать по последующим командам вывода Write или Writeln будет происходить на той же строчке. При вызове оператора Writeln без параметров просто происходит переход на новую строку.

## Условный оператор

Одной из основных алгоритмических структур является ветвление (альтернатива).

Если условие выполняется, то будет выполнена инструкция "1", если нет, то - инструкция "2". Несмотря на то, что в схеме присутствуют два действия, выполнено будет только одно, так как условие либо ложно, либо истинно.

Как это записать на Паскале? Да точно так же, только по-английски.

Формат условного оператора на языке Паскаль:

```
If <условие>  
Then <оператор 1>  
Else <оператор 2>;
```

Обратите внимание на то, что в Then- и Else- части стоит только один оператор. Но что делать, чтобы решить задачу, в которой по выполнению или невыполнению условия нужно совершить не одно, а несколько действий? Здесь приходит на помощь уже известный вам

составной оператор. В операторные скобки можно заключить любое количество операторов.

Вариант условного оператора в этом случае:

If <условие>

Then Begin <группа операторов 1> end

Else Begin < группа операторов 2> end;

Цикл. Виды Циклов.

Циклом называется многократное повторение однотипных действий. Телом же цикла будем называть те самые действия, которые нужно многократно повторять.

Цикл "ПОКА"

Группа операторов, называемая "телом цикла", судя по этой схеме, будет выполняться пока истинно условие цикла. Выход из цикла произойдет, когда условие перестанет выполняться.

Если условие ложно изначально, то тело цикла не будет выполнено ни разу. Если условие изначально истинно и в теле цикла нет действий, влияющих на истинность этого условия, то тело цикла будет выполняться бесконечное количество раз. Такая ситуация называется "зацикливанием". Прервать зациклившуюся программу может либо оператор (нажав Ctrl+C), либо аварийный останов самой программы, в случае переполнения переменной, деления на ноль и т.п., поэтому использовать структуру цикла следует с осторожностью, хорошо понимая, что многократное выполнение должно когда-нибудь заканчиваться.

На языке Pascal структура цикла "Пока" записывается следующим образом:

While <условие> Do <оператор>;

Правда, лаконично? По-русски можно прочесть так: "Пока истинно условие, выполнять оператор". Здесь, так же как в формате условного оператора, подразумевается выполнение только одного оператора. Если необходимо выполнить несколько действий, то может быть использован составной оператор. Тогда формат оператора принимает такой вид:

While <условие> Do  
Begin

```
<оператор #1>;  
<оператор #2>;  
<оператор #3>;  
...
```

End;

## Цикл "ДО"

Этот вид цикла отличается от предыдущего в основном тем, что проверка условия повторения тела цикла находится не перед ним, а после. Поэтому цикл "До" называют циклом "с постусловием", а "Пока" - "с предусловием".

Обратите также внимание на то, что новая итерация (повторное выполнение тела цикла) происходит не тогда, когда условие справедливо, а как раз тогда, когда оно ложно. Поэтому цикл и получил свое название (выполнять тело цикла до выполнения соответствующего условия).

Интересно, что в случае, когда условие цикла изначально истинно, тело цикла все равно будет выполнено хотя бы один раз. Именно это отличие "до" от "пока" привело к тому, что в программировании они не подменяют друг друга, а используются для решения задач, к которым они более подходят.

Формат цикла на языке Pascal:

Repeat

```
<оператор #1>;  
<оператор #2>;  
<оператор #3>;  
...
```

Until <условие>;

Читается так: "Выполнять оператор #1, оператор #2. : до выполнения условия".

Здесь не требуется использование составного оператора, потому, что сами слова Repeat и Until являются операторными скобками.

Цикл "С параметром".

В данном случае параметром будет являться целочисленная переменная, которая будет изменяться на единицу при каждой итерации цикла. Таким образом, задав начальное и конечное значения для такой переменной, можно точно установить количество выполнений тела цикла. Нарисовать блок-схему такой структуры вы сможете сами после некоторых пояснений.

Форматов у этого вида цикла предусмотрено два:

For <И.П.>:=<Н.З.> To <К.З.> Do <оператор>;

For <И.П.>:=<Н.З.> Downto <К.З.> Do <оператор>;

Здесь И.П. - имя переменной-параметра, Н.З. - его начальное значение, К.З. - соответственно конечное значение параметра. В качестве начального и конечного значений

Читается данная структура так: "Для переменной (далее следует ее имя) от начального значения до конечного выполнять оператор (являющийся телом цикла)". Иногда цикл с параметром даже называют "Для" или "For". В первом случае параметр с каждой итерацией увеличивается на единицу, во втором - уменьшается.

Выполняется этот цикл по следующему алгоритму:

1. переменной-параметру присваивается начальное значение;
2. выполняется тело цикла;
3. переменная-параметр автоматически увеличивается на 1 (в первом случае формата);
4. если параметр превышает конечное значение, то происходит выход из цикла, иначе - переход к пункту 2.

Примечание: при использовании Downto параметр автоматически уменьшается на 1, а выход из цикла происходит тогда, когда параметр становится меньше конечного значения.

Таким образом, в отличие от первых двух видов цикла, этот цикл используется тогда, когда известно необходимое количество выполнений тела цикла.

Вообще говоря, цикл "Пока" является универсальным, то есть любая задача, требующая использования цикла, может быть решена с применением этой структуры. Циклы "До" и "С параметром" созданы для удобства программирования.



## Массивы

До сих пор мы рассматривали переменные, которые имели только одно значение, могли содержать в себе только одну величину определенного типа. Исключением являлись лишь строковые переменные, которые представляют собой совокупность данных символьного типа, но и при этом мы говорили о строке, как об отдельной величине.

Описание типа линейного массива выглядит так:

Type <Имя типа>=Array [<Диапазон индексов>] Of <Тип элементов>;

В качестве индексов могут выступать переменные любых порядковых типов. При указании диапазона начальный индекс не должен превышать конечный. Тип элементов массива может быть любым (стандартным или описанным ранее).

Описать переменную-массив можно и сразу (без предварительного описания типа) в разделе описания переменных:

Var <Переменная-массив> : Array [<Диапазон индексов>] Of <Тип элементов>;

Примеры описания массивов:

Var

S, BB : Array [1..40] Of Real;

N : Array ['A'..'Z'] Of Integer;

R : Array [-20..20] Of Word;

T : Array [1..40] Of Real;

Теперь переменные S, BB и T представляют собой массивы из сорока вещественных чисел; массив N имеет индексы символьного типа и целочисленные элементы; массив R может хранить в себе 41 число типа Word.

Единственным действием, которое возможно произвести с массивом целиком - присваивание. Для данного примера описания впоследствии допустима следующая запись:

S:=BB;

Однако, присваивать можно только массивы одинаковых типов. Даже массиву T присвоить массив S нельзя, хотя, казалось бы, их описания совпадают, произведены они в различных записях раздела описания.

Никаких других операций с массивами целиком произвести невозможно, но с элементами массивов можно работать точно также, как с простыми переменными соответствующего типа. Обращение к отдельному элементу массива производится при помощи указания имени всего массива и в квадратных скобках - индекса конкретного элемента. Например:

R[10] - элемент массива R с индексом 10.

Фундаментальное отличие компонента массива от простой переменной состоит в том, что для элемента массива в квадратных скобках может стоять не только непосредственное значение индекса, но и выражение, приводящее к значению индексного типа. Таким образом реализуется косвенная адресация:

BB[15] - прямая адресация;

BB[K] - косвенная адресация через переменную K, значение которой будет использовано в качестве индекса элемента массива BB.

Такая организация работы с такой структурой данных, как массив, позволяет использовать цикл для заполнения, обработки и распечатки его содержимого.

### Процедуры и функции

При решении сложных объемных задач часто целесообразно разбивать их на более простые. Метод последовательной детализации позволяет составить алгоритм из действий, которые, не являясь простыми, сами представляют собой достаточно самостоятельные алгоритмы. В этом случае говорят о вспомогательных алгоритмах или подпрограммах. Использование подпрограмм позволяет сделать основную программу более наглядной, понятной, а в случае, когда одна и та же последовательность команд встречается в программе несколько раз, даже более короткой и эффективной.

В языке Паскаль существует два вида подпрограмм: процедуры и функции, определяемые программистом. Процедурой в Паскале называется именованная последовательность инструкций, реализующая некоторое действие. Функция отличается от процедуры тем, что она должна обязательно выработать значение определенного типа.

Процедуры и функции, используемые в программе, должны быть соответствующим образом описаны до первого их упоминания. Вызов процедуры или функции производится по их имени.

Подпрограммы в языке Паскаль могут иметь параметры (значения, передаваемые в процедуру или функцию в качестве аргументов). При описании указываются так называемые формальные параметры (имена, под которыми будут фигурировать передаваемые данные внутри подпрограммы) и их типы. При вызове подпрограммы вместе с ее именем должны быть заданы все необходимые параметры в том порядке, в котором они находятся в описании. Значения, указываемые при вызове подпрограммы, называются фактическими параметрами.

Формат описания процедуры:

```
Procedure <Имя процедуры> (<Имя форм. параметра 1>:<Тип>;  
<Имя форм. параметра 2>:<Тип>?);  
<Раздел описаний>  
Begin  
<Тело процедуры>  
End;
```

Раздел описаний может иметь такие же подразделы, как и раздел описаний основной программы (описание процедур и функций - в том числе). Однако все описанные здесь объекты "видимы" лишь в этой процедуре. Они здесь локальны также, как и имена формальных параметров. Объекты, описанные ранее в разделе описаний основной программы и не переопределенные в процедуре, называются глобальными для этой подпрограммы и доступны для использования.

Легко заметить схожесть структуры программы целиком и любой из ее процедур. Действительно, ведь и процедура и основная программа реализуют некий алгоритм, просто процедура не дает решения всей задачи. Отличие в заголовке и в знаке после End.

Формат описания функции:

```
Function <Имя функции> (<Имя форм. параметра 1>:<Тип>;  
<Имя форм. параметра 2>:<Тип>?) : <Тип результата>;  
<Раздел описаний>  
Begin  
<Тело функции>  
End;
```

В теле функции обязательно должна быть хотя бы команда присвоения такого вида: <Имя функции>:=<Выражение>;

Указанное выражение должно приводить к значению того же типа, что и тип результата функции, описанный выше.

Вызов процедуры представляет в программе самостоятельную инструкцию:

<Имя процедуры>(<Фактический параметр 1>, < Фактический параметр 2>?);

Типы фактических параметров должны быть такими же, что и у соответствующих им формальных.

## Работа с файлами

Тип-файл представляет собой последовательность компонент одного типа, расположенных на внешнем устройстве (например, на диске). Элементы могут быть любого типа, за исключением самого типа-файла. Число элементов в файле при описании не объявляется. Работа с физическими файлами происходит через так называемые файловые переменные.

Для задания типа-файла следует использовать зарезервированные слова File и Of, после чего указать тип компонент файла.

Пример:

Type

N = File Of Integer; {Тип-файл целых чисел}

C = File Of Char; {Тип-файл символов}

Есть заранее определенный в Паскале тип файла с именем Text. Файлы этого типа называют текстовыми.

Введя файловый тип, можно определить и переменные файлового типа:

Var

F1 : N;

F2 : C;

F3 : Text;

Тип-файл можно описать и непосредственно при введении файловых переменных:

Var

Z : File Of Word;

Файловые переменные имеют специфическое применение. Над ними нельзя выполнять никаких операций (присваивать значение,

сравнивать и т.д.). Их можно использовать лишь для выполнения операций с файлами (чтение, запись и т.д.).

Перед тем, как осуществлять ввод-вывод, файловая переменная должна быть связана с конкретным внешним файлом при помощи процедуры Assign.

Формат:

Assign(<Имя файловой переменной>, <Имя файла>);

Имя файла задается либо строковой константой, либо через переменную типа Sting. Имя файла должно соответствовать правилам работающей в данный момент операционной системы. Если строка имени пустая, то связь файловой переменной осуществляется со стандартным устройством ввода-вывода (как правило - с консолью).

После этого файл должен быть открыт одной из процедур:

Reset(<Имя файловой переменной>);

Открывается существующий файл для чтения, указатель текущей компоненты файла настраивается на начало файла. Если физического файла, соответствующего файловой переменной не существует, то возникает ситуация ошибки ввода-вывода.

Rewrite(<Имя файловой переменной>);

Открывается новый пустой файл для записи, ему присваивается имя, заданное процедурой Assign. Если файл с таким именем уже существует, то он уничтожается.

После работы с файлом он, как правило, должен быть закрыт процедурой Close.

Close(<Имя файловой переменной>);

Теперь рассмотрим непосредственную организацию чтения и записи.

Для ввода информации из файла, открытого для чтения, используется уже знакомый вам оператор Read. Правда, в его формате и использовании вы заметите некоторые изменения:

Read(<Имя файловой переменной>, <Список ввода>);

Происходит считывание данных из файла в переменные, имена которых указаны в списке ввода. Переменные должны быть того же типа, что и компоненты файла.

Вывод информации производит, как можно догадаться оператор Write(<Имя файловой переменной>, <Список вывода>);

# Программирование на Python.

Встроенные типы данных следующие:

Название	Описание
None	неопределенное значение переменной
bool	Булевый тип: True/False
int	Целочисленный тип
float	Вещественный тип
complex	Комплексный тип (для комплексных чисел)
str	Строковый тип
Бинарные списки	
bytes	Список байт
bytearray	Байтовые массивы
memoryview	Специальные объекты для доступа к внутренним данным объекта через protocol buffer
Множества	
set	Множество
frozenset	Неизменяемое множество
Словари	
dict	Словарь

- `input()` – для ввода данных с клавиатуры;
- `print()` – для вывода данных в консоль.

Начнем с первой. Вызвать функцию `input` можно таким образом:

```
input()
```

и после ее запуска среда выполнения будет ожидать от нас ввода данных. Введем, допустим, число 5 и нажмем `enter`. Смотрите, эта функция возвратила нам это число, но в виде строки:

```
'5'
```

В действительности, данная функция всегда возвращает строку, чего бы мы не вводили с клавиатуры. Ну хорошо, а как нам сохранить в переменной введенное значение? Это можно сделать так:

```
a = input()
```

Теперь, на все введенные данные будет ссылаться переменная a.

Несмотря на то, что input всегда возвращает строки, нам в программах в ряде случаев будет требоваться ввод чисел. И здесь возникает вопрос: как число из строки преобразовать в обычное число, которое можно будет в последствие использовать в арифметических операциях? Это делается с помощью функции

`int(<аргумент>)`

Данная функция пытается переданный аргумент преобразовать в число и вернуть уже числовое значение. Например:

```
int('5')
```

вернет число 5, или

```
int(a)
```

преобразует значение a в числовое. Однако, здесь следует быть осторожным и иметь в виду, что если аргумент не удастся преобразовать в число, то возникнет ошибка:

```
int('12dfd')
```

Отлично, с этим разобрались. Теперь мы можем преобразовывать строки в числа, используя такую конструкцию:

```
a = int(input())
```

Здесь сначала сработает input, а затем, введенное строковое значение будет преобразовываться в число и переменная a уже будет ссылаться на числовое значение. Благодаря этому, данную переменную можно в дальнейшем использовать в арифметических операциях, например:

```
a+2
```

и так далее (об арифметических операциях речь пойдет на следующем занятии).

По аналогии с int работает функция

`float(<аргумент>)`

которая преобразовывает строку в вещественное число. С ее помощью можно выполнять, например, такие преобразования:

```
float('3')  
float('4.5')
```

Каждый раз мы будем получать вещественные значения. То есть, можно записывать и такую конструкцию:

```
a = float(input())
```

и вводить любые вещественные числа.

В качестве примера рассмотрим простую программу вычисления периметра прямоугольника:

```
w = float(input())  
h = float(input())  
p = 2*(w+h)  
print(p)
```

Но здесь есть небольшое неудобство: при вводе значений пользователь не знает, что именно ему вводить. Нужно написать подсказки. Это реализуется так:

```
w = float(input("Введите ширину: "))  
h = float(input("Введите длину: "))
```

Теперь, он видит сообщение и понимает что нужно вводить с клавиатуры.

О функции print мы уже немного знаем, здесь рассмотрим подробнее различные возможности ее использования. Например, эту функцию можно записывать в таких вариациях:

```
print(1)  
print(1, 2)  
print(1, 2, 3)
```

И так далее, число аргументов может быть произвольным. Соответственно все эти значения в строку будут выведены в консоли. Причем, значения разделяются между собой пробелом. Это разделитель, который используется по умолчанию. Если нужно изменить значение этого разделителя, то для этого используется специальный именованный аргумент sep:

```
print(1, 2, 3, sep=",")  
print(1, 2, 3, sep="-")  
print(1, 2, 3, sep="***")
```



то есть, здесь можно прописывать самые разные строки-разделители.

Далее, вы уже заметили, что каждый вызов функции `print` делает перевод строки. Этот символ автоматически добавляет в конец выводимых данных. Но, мы также можем его изменить. Для этого используется именованный аргумент `end`:

```
print(1, 2, 3, sep=" ", end=':')
print(1, 2, 3, sep="-", end='--end--\n')
print(1, 2, 3, sep="***")
```

Смотрите, теперь у нас после первой строчки нет перевода строки, а поставлено двоеточие с пробелом, которые мы указали в аргументе `end`. После второго вывода в конце была добавлена строчка и указан символ `'\n'` перевода строки.

В качестве примера все это можно использовать для более гибкого вывода значений с помощью `print`:

```
name = "Федор"
print("Имя", name, sep=": ")
```

Но это не самый удобный вывод значений. Функция `print` позволяет делать довольно гибкий форматированный вывод данных с применением спецификаторов. Например:

```
name = "Федор"; age = 18
print("Имя %s, возраст %d"%(name, age))
```

В результате, вместо спецификатора `%s` будет подставлена первая переменная, указанная в скобках, в виде строки, а вместо `%d` – вторая переменная `age` в виде целого числа. То есть, для каждого типа данных существует свой спецификатор. Наиболее употребительные, следующие:

- `%d`, `%i`, `%u` – для вывода целочисленных значений;
- `%f` – для вывода вещественных значений;
- `%s` – для вывода строк;
- `%%` - для вывода символа `%`

Вот основные возможности функций `input` и `print` в Python.

И как уже говорили, в этом языке имеется три базовых типа для представления чисел:

- `int` – для целочисленных значений;

- float – для вещественных;
- complex – для комплексных.

С этими числами можно выполнять следующие арифметические операции:

Оператор	Описание	Приоритет
+	сложение	2
-	вычитание	2
*	умножение	3
/, //	деление	3
%	остаток деления	3
**	возведение в степень	4

И введем несколько определений:

**Операнд** – то, к чему применяется оператор. Например, в умножении  $5 * 2$  есть два операнда: левый операнд равен 5, а правый операнд равен 2. Иногда их называют «аргументами» вместо «операндов».

**Унарным** называется оператор, который применяется к одному операнду.

Например, оператор унарный минус "-" меняет знак числа на противоположный:

*#унарный минус*

`a=1; a=-a`

`print(a)`

Обратите внимание как записаны два оператора в одну строчку: они разделены точкой с запятой. Так тоже можно делать. Если каждый оператор начинается с новой строки, то точку с запятой ставить не обязательно, если пишем несколько операторов в одну строчку, то они разделяются точкой с запятой.

**Бинарным** называется оператор, который применяется к двум операндам.

Тот же минус существует и в бинарной форме:

`a = 1; b = 2`

`c = b-a #бинарный минус`

`print(c)`

Раз мы начали говорить об операциях + и -, то продолжим и отметим, что, в общем случае, можно использовать унарный плюс и минус, например:

```
a=2;  
print(+a, -a)
```

Конечно, +a это то же самое, что и a, поэтому, в основном, используется унарный минус. По приоритету унарные операции выше бинарных операций. Например, вот такая запись:

```
print(-a+b)
```

означает, что число -a возводится в степень 2, то есть, унарный минус имеет больший приоритет, чем бинарная операция \*\* возведения в степень.

Если же используются бинарные сложение и вычитание:

```
a=2; b=-10  
print(a+b)  
print(a-b)
```

то их приоритет становится наименьшим среди всех арифметических операций (они выполняются в последнюю очередь).

Следующая бинарная операция умножение работает так, как мы привыкли ее использовать в математике:

```
a=2; b=-5.8; c=2.3  
print( a+b*c )
```

Здесь сначала выполнится умножение, а затем – сложение. Если необходимо изменить приоритет выполнения операций, то используются круглые скобки:

```
print( (a+b)*c )
```

Далее, деление двух чисел (или переменных) можно выполнить двумя способами. Первый – традиционный, делает деление, привычное в математике, например:

```
3/2
```

получим ожидаемый результат 1,5. Однако те из вас, кто имеет опыт программирования на таких языках как C++ или Java, знают, что при делении двух целочисленных значений, результат также получался

целочисленным. Но в Python это не так! Его арифметические операции работают в соответствии с классическими правилами математики и деление здесь – это всегда полноценное деление двух значений, какими бы они ни были.

Однако, если все же требуется выполнить целочисленное деление (то есть, с отбрасыванием дробной части), то используется такой оператор:

```
3//2
```

И, как видите, теперь результат 1, а не 1,5. Причем, это целочисленное деление будет выполняться и с вещественными числами:

```
3.2//2
```

Вот такие два оператора деления существуют в Python.

Если же хотим вычислить остаток от целочисленного деления, то используется оператор:

```
x % y
```

С положительными целыми числами он работает также как и во многих других языках программирования. Например,

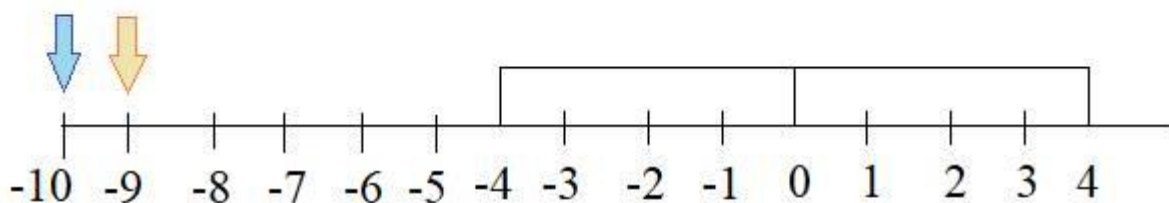
```
4 % 5 # 0  
7 % 5 # 2
```

и так далее, мы будем получать числа от 0 до 4. Но с отрицательными числами вычисления будут отличаться от того же языка C++. Например,

```
-9 % 5 # (в C++ это -4, а в Python – это 1)
```

Почему так? Дело в том, что когда то давно инженеры фирмы Intel неверно математически реализовали данную операцию. И язык C++ как наследник этой интеловской архитектуры реализует данную операцию путем вынесения знака «-» за скобки и вычисления обычного остатка от деления. Язык же Python делает это так, как принято в математике. Сначала находится ближайшее наименьшее число кратное 5. Это число -10 (для числа -9) и остаток берется как разность между этими числами:

$$-9 \% 5 = ?$$



$$-9 - (-10) = -9 + 10 = 1$$

то есть, остатки всегда будут положительными в диапазоне от 0 до 4, как это и должно быть по математике.

Все рассмотренные операторы (\*, /, //, %) имеют одинаковый приоритет и выполняются слева-направо. То есть, если записать

`a/b*c`

Следующая операция – возведение в степень. В самом простом варианте она записывается так:

```
x=2; y=5;  
x**y
```

здесь x, y могут быть и дробными числами. Например:

```
x=1.96; y=0.5;  
x**y
```

Это будет соответствовать извлечению квадратного корня из 1,96. Если запишем такую конструкцию:

```
27**(1/3)
```

то получим кубический корень из 27. Причем, обратите внимание, круглые скобки у степени здесь обязательны, т.к. приоритет операции \*\* выше, чем у деления. Если записать вот так:

```
27**1/3
```

Вот на это следует обращать внимание. И еще один нюанс. Операция возведения в степень выполняется справа-налево. То есть, если записать вот такую строчку:

```
2**3**2
```

Сначала (справа) вычисляется  $3^{**}2 = 9$ , а затем,  $2^{**}9 = 512$ . Все остальные арифметические операции работают слева-направо.

Используя оператор присваивания совместно с арифметическими операторами, можно выполнять некоторые полезные арифметические преобразования переменных. Например, очень часто требуется увеличить или уменьшить некую переменную на определенное число. Это можно сделать вот так:

```
i = 5; j = 3  
i = i+1  
j = j-2  
print(i, j)
```

Но, можно и короче, вот так:

```
i = 5; j = 3  
i += 1  
j -= 2  
print(i, j)
```

Они довольно часто используются в программировании. Также, помимо сложения и вычитания, можно записывать и такие выражения:

```
i *= 3  
j /= 4  
print(i, j)  
  
a = 5; b = 10  
a **= 2  
b //= 3  
print(a, b)
```

То есть, здесь до оператора присваивания можно записывать любую арифметическую операцию.

Все рассмотренные арифметические операции можно выполнять и с комплексными числами:

```
a = 1 + 2j
b = 2 - 3j
print(a+b)
print(a-b)
print(a*b)
print(a/b)
print(a**b)
```

Кроме операции целочисленного деления `//` и вычисления остатка от деления `%`. Дополнительно у объектов комплексных чисел есть свойства:

```
a = 1 + 2j
b = 2 - 3j
print(a.real)
print(b.imag)
```

для взятия действительной и мнимой части. И полезный метод:

```
sa = a.conjugate();
print(sa)
```

для получения комплексно-сопряженного числа.

В языке Python имеются встроенные функции для работы с числами. Наиболее полезные, следующие:

Название	Описание
<code>abs(x)</code>	вычисляет модуль числа <code>x</code>
<code>round(x)</code>	округляет <code>x</code> до ближайшего целого
<code>min(x1, x2,...,x_n)</code>	находит минимальное, среди указанных чисел
<code>max(x1, x2,...,x_n)</code>	находит максимальное, среди указанных чисел
<code>pow(x, y)</code>	возводит <code>x</code> в степень <code>y</code>

Также в языке Python имеется стандартная библиотека `math`, которая содержит большое количество стандартных математических функций. Чтобы ей воспользоваться, необходимо в начале программы подключить эту библиотеку. Делается это с помощью ключевого слова `import`, за которым указывается имя библиотеки:

```
import math
```

После этого становятся доступными следующие полезные функции:

Название	Описание
<code>math.ceil(x)</code>	возвращает ближайшее наибольшее целое для $x$
<code>math.floor(x)</code>	возвращает ближайшее наименьшее целое для $x$
<code>math.fabs(x)</code>	возвращает модуль числа $x$
<code>math.factorial(x)</code>	вычисляет факториал $x!$
<code>math.exp(x)</code>	вычисляет $e^x$
<code>math.log2(x)</code>	вычисляет логарифм по основанию 2
<code>math.log10(x)</code>	вычисляет логарифм по основанию 10
<code>math.log(x, [base])</code>	вычисляет логарифм по указанному основанию $base$ (по умолчанию $base = e$ – натуральный логарифм)
<code>math.pow(x, y)</code>	возводит число $x$ в степень $y$
<code>math.sqrt(x)</code>	вычисляет квадратный корень из $x$
Тригонометрические функции	
<code>math.cos(x)</code>	вычисляет косинус $x$
<code>math.sin(x)</code>	вычисляет синус $x$
<code>math.tan(x)</code>	вычисляет тангенс $x$
<code>math.acos(x)</code>	вычисляет арккосинус $x$
<code>math.asin(x)</code>	вычисляет арксинус $x$
<code>math.atan(x)</code>	вычисляет арктангенс $x$
Математические константы	
<code>math.pi</code>	число $\pi$
<code>math.e</code>	число $e$

Помимо этих есть и другие математические функции. При необходимости, о них можно почитать в официальной документации языка Python.

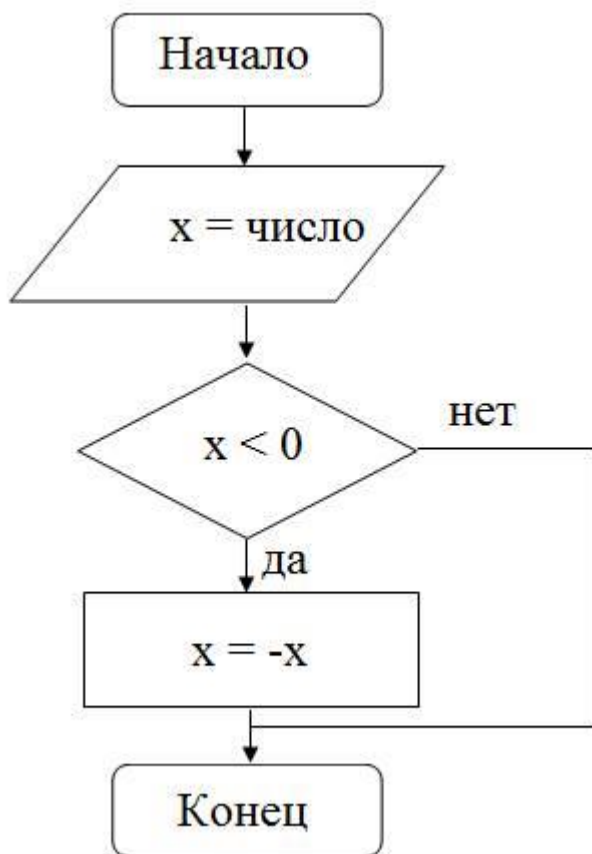
Применение этих функций вполне очевидно, например, их можно вызвать вот так:

```
import math
a = 2.4
b = math.cos(a)
print(b)
print( math.floor(1.7) )
print( math.ceil(1.7) )
print( math.sqrt(2.56) )
```



И так далее. Вот так работают арифметические операции в Python и вот такие математические функции имеются в стандартной библиотеке `math`.

Представьте, что вам нужно вычислить модуль числа, хранящегося в переменной `x`. Как это сделать? Очевидно, нужно реализовать такой алгоритм.



И в этом алгоритме есть вот такое ветвление программы: при  $x < 0$  меняется знак на противоположный, а при других  $x$  это не делается. В результате получаем модуль числа в переменной `x`.

Так вот, чтобы реализовать проверку таких условий в Python имеется один условный оператор `if`, который в самом простом случае имеет такой синтаксис:

`if(<условное выражение>) : оператор`

или так:

`if <условное выражение> : оператор`

Если условное выражение истинно, то выполняется оператор, записанный в `if`. Иначе этот оператор не выполняется. Используя

оператор ветвления, запишем программу для вычисления модуля числа:

```
x = -5
if(x < 0) : x = -x
print(x)
```

Здесь операция изменения знака переменной x будет выполняться только для отрицательных величин, а положительные просто выводятся в консоль, минуя эту операцию.

Какие операторы сравнения существуют в Python и как они работают? Многие из них нам известны из школьного курса математики, это:

a > b	Истинно, если a больше b
a < b	Истинно, если a меньше b
a >= b	Истинно, если a больше или равно b
a <= b	Истинно, если a меньше или равно b
a == b	Истинно, если a равно b (обратите внимание, для сравнения используется двойной знак равенства)
a != b	Истинно, если a не равно b

Все эти операторы при сравнении возвращают булево значение: True – истина или False – ложь. Например:

```
print(2 > 1)
print(2 == 1)
print(2 != 1)
```

Результат сравнения можно присвоить переменной, как и любое значение:

```
result = 7 > 5
print(result)
```

## Сравнение строк

Как вы видите, сравнение двух числовых значений выполняется вполне очевидным образом. Но можно ли, например, сравнивать строки между собой? Оказывается да, можно. Чтобы определить, что одна строка больше другой, Python использует «алфавитный» или «лексикографический» порядок. Другими словами, строки сравниваются посимвольно. Например:

```
print('Я' > 'А' )  
print( 'Кот' > 'Код' )  
print( 'Сонный' > 'Сон' )
```

Алгоритм сравнения двух строк довольно прост:

1. Сначала сравниваются первые символы строк.
2. Если первый символ первой строки больше (меньше), чем первый символ второй, то первая строка больше (меньше) второй.
3. Если первые символы равны, то таким же образом сравниваются уже вторые символы строк.

Сравнение продолжается, пока не закончится одна из строк. Если обе строки заканчиваются одновременно, и все их соответствующие символы равны между собой, то строки считаются равными. Иначе, большей считается более длинная строка.

В примерах выше сравнение 'Я' > 'А' завершится на первом шаге, тогда как строки "Кот" и "Код" будут сравниваться посимвольно:

1. К равна К.
2. о равна о.
3. т больше чем д.

На этом сравнение заканчивается. Первая строка больше.

### Конструкция if – elif – else

Теперь, когда мы знаем как сравниваются между собой величины, вернемся к нашему условному оператору if. И предположим, что хотим определить знак числа в переменной x. Конечно, проверку можно записать вот так:

```
x = -5  
if x < 0 : print("x отрицательное число")  
if x >= 0 : print("x неотрицательное число")
```

Но можно сделать лучше. Смотрите, мы здесь имеем дело со взаимоисключающими условиями, то есть, они не могут произойти одновременно: либо первое, либо второе. Для таких ситуаций можно использовать ключевое слово else – иначе, чтобы ускорить процесс проверки:

```
x = 5  
if x < 0:
```

```
print("x отрицательное число")
else:
    print("x неотрицательное число")
```

Теперь, у нас здесь всего одно условие. Если оно истинно, то выполнится первый print, а иначе – второй. Такая программа будет работать быстрее. И обратите внимание на синтаксис записи функции print: перед ней необходимо записать хотя бы один пробел (обычно ставится 4 пробела или символ табуляции). Эти пробелы в Python означают блок кода, который выполняется по некоторому условию. В данном случае блок кода состоит всего из одного оператора print. То же самое и для else.

В общем случае, синтаксис оператора if else следующий:

```
if(<выражение>): оператор 1
else: оператор 2
```

или

```
if(<выражение>):
    оператор 1
else:
    оператор 2
```

Если же мы хотим по такому принципу выполнить три проверки:  $x > 0$ ;  $x < 0$ ;  $x == 0$ , то предыдущую программу можно записать так:

```
if x < 0:
    print("x отрицательное число")
elif x > 0:
    print("x положительное число")
else:
    print("x равен 0")
```

И вообще таких конструкций

if – elif – elif – ... – elif – else

может быть много. Далее, обратим внимание на такой факт: во всех наших примерах по условию шел один оператор – print. Но что если нужно выполнить несколько операторов по некоторому условию? Для этого их нужно записывать по такому синтаксису:

```
if <условие>:
    оператор 1
```

оператор 1  
...  
оператор N

Например:

```
x = -10; sgn = 0
if x < 0:
    sgn = -1
    print("x отрицательное число", sgn)
elif x > 0:
    sgn = 1
    print("x положительное число", sgn)
else:
    print("x равен 0", sgn)
```

Здесь по первым двум условиям выполняется два оператора: присвоение значения переменной sgn и вывод результата в консоль.

В ряде случаев конструкцию if-else удобнее записывать через **тернарный условный оператор**, который имеет такой синтаксис:

result = значение1 if <условие> else значение2

При истинности условия возвращается значение1, в противном случае – значение2. Например:

```
age = 18
accessAllowed = True if age >= 18 else False
print(accessAllowed)
```

Получим True, если возраст (age) больше или равен 18, иначе – False. Кстати, проверку из данного примера можно сделать короче, просто прописав

```
accessAllowed = age >= 18
```

здесь оператор >= вернет True при возрасте больше или равен 18 и False – в противном случае.

Теперь, когда мы разобрались с базовыми моментами проверки условий, сделаем следующий шаг и попробуем реализовать проверку попадания переменной x в диапазон [2; 7], то есть, условие должно быть истинным, когда x принимает значения в этом диапазоне чисел. Очевидно, что здесь должно быть две проверки: первая – мы

проверяем, что  $x \geq 2$  и вторая – проверяем, что  $x \leq 7$ . Если оба этих условия выполняются одновременно, то  $x$  попадает в наш диапазон. Реализовать такую проверку на Python можно так:

```
x = 4
if x >= 2 and x <= 7: print("x попадает в [2; 7]")
else: print("x не попадает в [2; 7]")
```

Смотрите, здесь записано два условия, объединенных по И (and – это И). В результате, общее составное условие будет считаться истинным, если истинно и первое и второе условие. Если хотя бы одно из этих условий ложно, то ложно и все составное условие. В результате мы корректно реализуем проверку на вхождение значения переменной в диапазон [2; 7].

А теперь давайте реализуем противоположное условие, что  $x$  не принадлежит диапазону [2; 7]. Условие будет таким:

```
x = 40
if(x < 2 or x > 7): print("x не попадает в [2; 7]")
else: print("x попадает в [2; 7]")
```

Здесь в составном условии используется связка по ИЛИ (or – это ИЛИ) и оно будет истинно, если истинно или первое, или второе условие. То есть, в нашем случае, если  $x < 2$  или  $x > 7$ , то делается вывод о невхождении переменной  $x$  в указанный диапазон.

Итак, запомните следующие правила:

- условие  $x \geq 2$  and  $x \leq 7$  истинно, если истинно каждое из подусловий ( $x \geq 2$  и  $x \leq 7$ ) и ложно, если ложно хотя бы одно из них;
- условие  $x < 2$  or  $x > 7$  истинно, если истинно хотя бы одно из подусловий ( $x < 2$  или  $x > 7$ ) и ложно, когда оба ложны.

Вот так можно записывать более сложные условия в условном операторе if. Причем они могут комбинироваться в любом сочетании, например:

```
x = 4; y = -2
if x >= 2 and x <= 7 and (y < 0 or y > 5):
    print("x попадает в [2; 7], y не попадает в [0; 5]")
```

Здесь реализована проверка, что  $x$  должно принадлежать [2; 7], а  $y$  не принадлежать [0; 5]. И обратите внимание вот на эти круглые скобки.

Дело в том, что приоритет у операции `and` выше, чем у `or`, поэтому без скобок у нас бы получилась вот такая проверка:

```
if (x >= 2 and x <= 7 and y < 0) or (y > 5)
```

то есть, мы проверяли бы, что `x` принадлежит `[2; 7]` и `y` меньше нуля ИЛИ `y` больше 5. Как вы понимаете – это уже совсем другая проверка. Поэтому учитывайте приоритет этих операций при формировании составного условия. Если нужно изменить приоритет – используйте круглые скобки.

## Одиночные проверки

Внутри условия можно прописывать и такие одиночные выражения:

```
x = 4; y = True; z = False
if(x): print("x = ", x, " дает true")
if(not 0): print("0 дает false")
if("0"): print("строка 0 дает true")
if(not ""): print("пустая строка дает false")
if(y): print("y = true дает true")
if(not z): print("z = false дает false")
```

Вот этот оператор `not` – это отрицание – НЕ, то есть, чтобы проверить, что `0` – это `false` мы преобразовываем его в противоположное состояние с помощью оператора отрицания НЕ в `true` и условие срабатывает. Аналогично и с переменной `z`, которая равна `false`.

Из этих примеров можно сделать такие выводы:

1. Любое число, отличное от нуля, дает `True`. Число `0` преобразуется в `False`.
2. Пустая строка – это `False`, любая другая строка с символами – это `True`.
3. С помощью оператора `not` можно менять условие на противоположное (в частности, `False` превращать в `True`).

Итак, в условиях мы можем использовать три оператора: `and`, `or` и `not`. Самый высокий приоритет у операции `not`, следующий приоритет имеет операция `and` и самый маленький приоритет у операции `or`. Вот так работает оператор `if` в Python.

Ни одна сколь-нибудь серьезная программа на Python не обходится без циклов. Что такое циклы?





Представьте, что спортсмен бежит по дорожкам стадиона и решил: пока не прошел один час, он бежит. То есть, пока выполняется условие (оно истинно – не прошел час), циклично выполняются некие действия – бегун бежит. Вот такую операцию на уровне языка Python выполняет оператор цикла `while`, имеющий следующий синтаксис:

```
while <условие> :           заголовок
    оператор 1
    оператор 2
    ...
    оператор N              тело цикла
```

Смотрите как в Python записывается группа операторов (тело цикла): вся группа должна иметь один и тот же сдвиг относительно оператора `while` (обычно ставят четыре пробела или символ табуляции). Причем такой отступ строго обязателен – это элемент синтаксиса python. И благодаря этому текст программы становится наглядным и хорошо читаемым. Это, безусловно, один из плюсов данного языка.

Однократное выполнение тела цикла называется **итерацией**. То есть, может быть первая итерация, вторая итерация, N-я итерация и так далее.

Давайте в качестве примера с помощью оператора цикла `while` вычислим вот такую вот сумму:

$$S = \sum_{i=1}^{1000} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{1000}$$

Расписывать все это через тысячу слагаемых не очень то удобно. И к тому же число слагаемых может зависеть от значения переменной и быть неопределенным. В таких задачах без циклов не обойтись. И программа будет выглядеть так:



```
S=0; i=1
while i <= 1000:
    S += 1/i
    i += 1
print(S)
```

В качестве выражения в цикле `while` можно писать все те же самые условия, что и в условном операторе `if`. Например, можно вычислять сумму `S` пока либо `i<=1000`, либо `S < 5`. Такое условие запишется так:

```
while i <= 1000 and S < 5:
```

здесь цикл будет работать пока `i<=1000` и `S<5` как только одно из подусловий станет ложным, все составное условие становится ложным и цикл завершит свою работу.

Вернемся к нашему спортсмену, бегущему по стадиону. И предположим, что прошел час, но бегун еще не завершил полный круг. Что произойдет? Цикл сразу завершится? Нет, проверка условия завершения происходит только на начальной отметке, то есть, спортсмен должен добежать круг целиком и только потом проверить: прошел час или нет.



Другими словами, пока целиком не выполнится текущая итерация тела цикла, оператор `while` продолжает свою работу. И как только условие цикла становится ложным, то бегун останавливается и цикл завершает свою работу.

А что будет, если условие в цикле `while` будет истинным всегда? В этом случае мы получим «вечный» цикл, программа фактически зависнет и наш спортсмен будет обречен на бесконечный бег по кругу.

```
S=0; i=1
while 1 : S += 1
print(S)
```

Далее, цикл `while` может иметь необязательный блок `else`, который идет после цикла:

```
while <условие> :  
    оператор 1  
    оператор 2  
    ...  
    оператор N  
  
else:  
    операторы  
    после завершения  
    цикла
```

заголовок

тело цикла

блок, выполняемый  
после завершения  
цикла `while`

последующие операторы

Это, вроде как естественный выход из оператора цикла. В нашей иллюстрации это может быть традиционное посещение спортсменом душа после пробежки.



else

while



И здесь часто возникает вопрос: а чем блок `else` отличается от блока операторов, просто идущих после блока `while`? Ведь когда цикл `while` завершится, мы так и так перейдем к последующим операторам! Однако, тут есть один нюанс. Любой оператор цикла в Python может быть досрочно прерван с помощью оператора

`break`

Как только он встречается в теле цикла, цикл (в данном случае while) завершает свою работу. Это как если вдруг возник пожар и спортсмен не дожидаясь окончания круга спешно бежит со стадиона. В этом случае спортсмену уже не до душа, он сразу хватается за свои вещи и убегает из спортивного комплекса. То есть, при досрочном прерывании работы цикла while, конструкция else не выполняется и управление переходит на последующие операторы. Вот в чем отличие блока else от операторов, стоящих непосредственно после while. Например:

```
S=0; i=-10
while i < 100:
    if i == 0: break
    S += 1/i
    i=i+1
else:
    print("Сумма вычислена корректно")
print(S)
```

Если здесь при вычислении суммы ожидается деление на 0, то срабатывает break и цикл досрочно прерывается. В этом случае блок else не срабатывает и мы не видим сообщения, что сумма вычислена корректно. Если же все проходит штатно (без вызова break), то в консоли появляется сообщение

Сумма вычислена корректно

означающее выполнение блока else.

Раз уж мы начали говорить об управляющем операторе break, сразу отметим второй подобный оператор

### continue

Этот оператор позволяет пропускать тело цикла и перейти к следующей итерации, не прерывая работу самого цикла. Например, мы хотим перебрать все целые значения от -4 до 4, исключая значение 0. Такую программу можно реализовать так:

```
S=0; i=-5
while i < 4:
    i=i+1
    if i == 0: continue
    print(i)
    S += 1/i
print(S)
```

При выполнении этой программы увидим, что в консоль выведены все значения кроме нуля. Так как при  $i=0$  срабатывает условие и выполняется оператор `continue`. Все что находится после этого оператора пропускается и цикл продолжается уже со значением  $i=1$ .

Вот так работают эти два управляющих оператора `break` и `continue`, которые можно использовать во всех операторах циклов.

## Оператор цикла `for`

Следующий и, наверное, самый часто используемый оператор цикла – это оператор `for`, который имеет такой синтаксис:

```
for <переменная> in <список> :  
    операторы 1...N
```

Например,

```
for x in 1,5,2,4:  
    print(x**2)
```

выведет в консоль квадраты соответствующих чисел. Но что, если мы хотим перебрать значения по порядку в соответствии с правилом:

начальное значение, шаг, конечное значение

Для этого используется генератор последовательностей

`range(start, stop, step)`

Например, если мы запишем его вот так:

```
for x in range(1,5,1):  
    print(x)
```

то в консоли увидим числа от 1 до 4 с шагом 1. То есть, `range` генерирует последовательность в интервале

`[1;5)`

Последнее значение не входит в интервал. Если в нашем примере поставить шаг отрицательный `-1`, то конечное значение 5 не может быть достигнуто и в этом случае Python возвратит пустую последовательность:

```
for x in range(1,5,-1):  
    print(x)
```

Если нам нужны числа от 5 до 1, то следует записывать range в таком виде:

```
for x in range(5,0,-1):  
    print(x)
```

Причем, в range можно записывать только целые числа, с вещественными он не работает.

Давайте перепишем нашу программу подсчета суммы

$$S = \sum_{i=1}^{1000} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{1000}$$

с помощью цикла for, получим:

```
S=0  
for i in range(1, 1001, 1):  
    S += 1/i  
print(S)
```

Здесь весь цикл записан буквально в одну строчку, а тело цикла состоит из одного оператора – подсчета суммы ряда.

Вторым примером рассмотрим задачу вычисления значений линейной функции

$$y = f(x) = kx + b, \quad x = 0; 0.1; 0.2; \dots; 0.5$$

Программа будет выглядеть так:

```
k = 0.5; b = 2  
lst = [0, 0.1, 0.2, 0.3, 0.4, 0.5]  
for x in lst:  
    print(x*k+b)
```

Этот пример показывает, что для перебора значений счетчика x можно использовать списки, сформированные ранее в программе. (О списках мы подробнее будем говорить на последующих занятиях). Здесь же приведем еще один пример:

```
msg = "Hello World!"  
for x in msg:  
    print(x)
```

Он показывает, что строку можно воспринимать как список и перебирать с помощью цикла for.

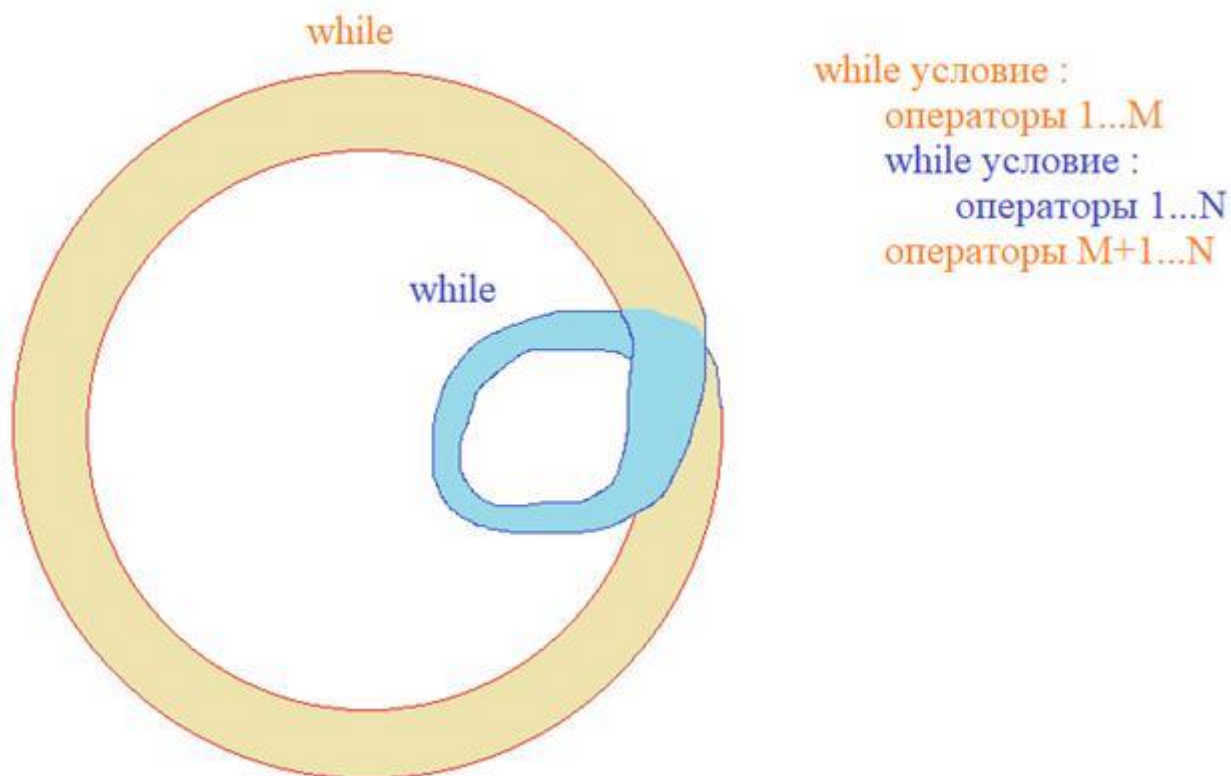
Также в цикле for можно использовать блок else, о котором мы говорили ранее:

```
for <переменная> in <список> :  
    операторы 1...N  
else:  
    операторы 1...N
```

## Вложенные циклы

Итак, мы с вами рассмотрели два оператора циклов: while и for. Все эти циклы можно комбинировать друг с другом. То есть, создавать вложенные циклы (цикл внутри цикла).

Как это работает? Представьте, что бегун начинает бежать по большому кругу, но затем, для продолжения бега, ему необходимо сделать еще несколько вложенных кругов, после чего он возвращается на большой круг и продолжает свой бег.



В частности, такие вложенные циклы очень полезны для перебора элементов матрицы

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{M1} & a_{M2} & \dots & a_{MN} \end{bmatrix}$$

Тогда мы делаем первый цикл от 1 до N и вложенный от 1 до M

```
A = [ [1,2,3], [4,5,6] ]
N=2; M=3
for i in range(N):
    for j in range(M):
        print(A[i][j])
    print()
```

Или для подсчета вот такой двойной суммы ряда

$$S = \sum_{i=1}^N \sum_{j=1}^M i \cdot j$$

Программа будет выглядеть так:

```
S=0; M=10; N=5
for i in range(1,N+1):
    for j in range(1,M+1):
        S += i*j
    print(S)
```

Мы здесь сначала пробегаем все значения j от 1 до M при фиксированном i=1, затем, значение i увеличивается на 1, становится 2 и при этом i снова пробегаются значения j от 1 до M. И так пока i не превысит значение N. То есть, второй цикл вложен вот в этот первый. И таких вложений можно делать сколько угодно.

Вот так работают операторы циклов в Python и теперь вы знаете как их можно применять на практике.

Часто в программах требуется хранить различные списки данных, например, список городов, число выигранных очков в серии игр, или значения некоторой функции:



Москва	1200	$y = f(x)$
Санкт-Петербург	200	0.5
Самара	500	0.55
Казань	2100	0.6
Тверь	100	0.4

Все это можно представить в виде упорядоченно списка, который в Python задается с помощью оператора квадратных скобок:

[элемент1, элемент2, ..., элементN]

Например, для хранения городов можно задать такой список:

```
lst = ["Москва", "Санкт-Петербург", "Тверь", "Казань"]
```

И он будет упорядоченный, то есть, каждому элементу здесь соответствует свой порядковый индекс, начиная с нулевого:

```
lst = ["Москва", "Санкт-Петербург", "Тверь", "Казань"]
```



Здесь синей рамкой отмечен сам список, внутри которого располагаются элементы. Если мы посмотрим тип объекта, на который ссылается переменная `lst`:

```
type(lst)
```

то увидим значение «list». Это как раз и есть тип списка. То есть, через переменную `lst` мы можем работать со списком в целом, и первое, что нас здесь интересует: как обратиться к определенному элементу этого списка? Для этого используется такой синтаксис:

список[индекс]

Например,

```
lst[0]
lst[2]
```

Но, если мы укажем не существующий индекс:



```
lst[5]
```

то возникнет ошибка. Чтобы этого избежать нам надо знать значение последнего индекса. Для этого можно воспользоваться функцией

```
len(список)
```

которая возвращает число элементов в списке:

```
len(lst)
```

вернет значение 4. Но, так как индексы начинаются с нуля, то последний индекс будет равен:

```
lastIndex = len(lst) - 1
```

То есть, можно записать вот так:

```
lst[len(lst)-1]
```

но можно и проще, вот так:

```
lst[-1]
```

Этот пример показывает, что при отрицательных индексах, мы начинаем движение с конца списка и значение -1 дает самый последний элемент.

Далее, для перебора элементов списка в Python очень удобно использовать цикл for:

```
lst = ["Москва", "Санкт-Петербург", "Тверь", "Казань"]
for city in lst:
    print(city)
```

Смотрите, как это легко и просто делается! Конечно, мы можем распечатать весь контейнер целиком, просто записав:

```
print(lst)
```

Но здесь нет перебора всех элементов, а просто печать содержимого. Вернемся к циклу. Здесь переменная city будет ссылаться на элементы внутри списка lst. Давайте выведем в консоль дополнительно еще тип выводимого значения:

```
print(city, type(city))
```

Увидим везде строковый тип str. Но раз city ссылается на элементы списка, можно ли их изменить, присвоив этой переменной другое значение?

```
city = "новое значение"
```

Если мы теперь выведем список lst в консоль:

```
print(lst)
```

то окажется, что список не изменился. Почему? Дело в том, что когда мы присваиваем переменной новое значение, то она просто начинает ссылаться на новый объект и на состоянии списка это никак не сказывается.



Вот этот момент работы переменных как ссылок на объекты всегда следует учитывать при программировании на Python. Также следует помнить, что в этом языке типы данных делятся на два класса: **изменяемые** и **неизменяемые**. Все предыдущие типы, что мы проходили на прошлых занятиях относились к неизменяемым. Это были:

числа, булевы значения, строки

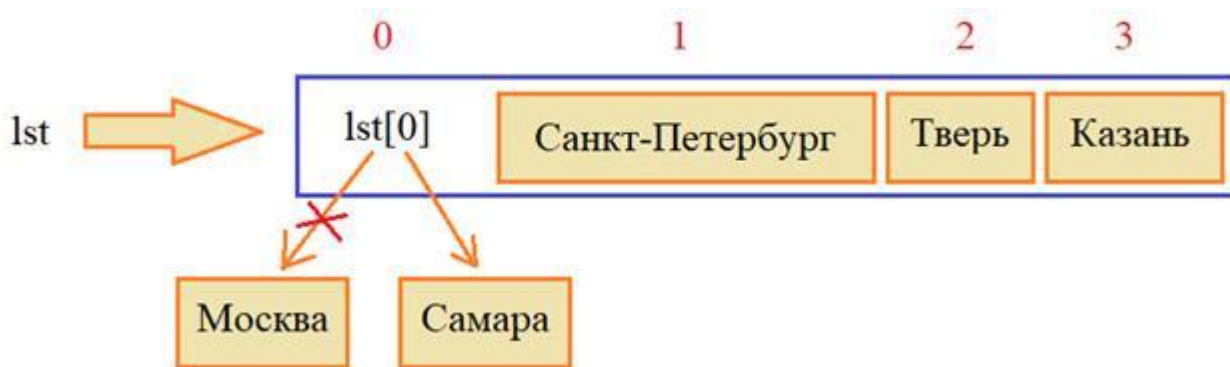
Но вот список list относится к изменяемым типам, то есть, мы можем изменить его состояние, не создавая нового объекта. В самом простом случае мы можем воспользоваться таким синтаксисом:

список[индекс] = значение

Например, так:

```
lst[0] = "Самара"
```

Теперь первый элемент не «Москва», а «Самара». В действительности, здесь произошло следующее:



Мы поменяли значение ссылки `lst[0]` первого элемента списка. Изначально она ссылалась на объект «Москва», а затем, стала ссылаться на объект «Самара». Прежний объект автоматически удаляется сборщиком мусора. Вот так происходит изменение элементов списка, то есть, меняются значения ссылок на новые объекты.

Теперь вернемся к вопросу изменения элементов списка внутри цикла `for`. Предположим, у нас имеется список чисел:

```
digs = [-1, 0, 5, 3, 2]
```

и в цикле мы хотим его изменить на их квадраты. Для этого запишем цикл в таком виде:

```
digs = [-1, 0, 5, 3, 2]
for x in range(5):
    digs[x] **= 2 #digs[x] = digs[x]**2
print(digs)
```

Или, чтобы не указывать конкретное число в функции `range`, ее можно переписать так:

```
for x in range(len(digs)):
```

И программа будет работать со списком произвольной длины.

Во всех наших примерах мы создавали списки небольшой длины и потому могли их запросто записать в программе. Но что если требуется создать список размерностью в 100 или 1000 элементов? Для этого можно воспользоваться такой конструкцией, например:

```
A = [0]*1000
```

создает список из 1000 элементов со значением 0. Фактически, мы здесь сначала создали список из одного нулевого элемента, а затем, размножили его до тысячи. Или, можно сделать так:

```
A = ["none"]*100
```

Получим 100 элементов со строкой «none». И так далее. Кстати, если требуется создать пустой список, то это будет так:

```
A = []
```

Ну хорошо, есть у нас список из 100 или 1000 или другого числа элементов. Но как нам теперь с ним работать, например, занести туда какие-либо данные. Для наглядности, предположим, пользователь вводит N чисел с клавиатуры ( $N < 100$ ) и пока он вводит положительные значения, мы их добавляем в список. Как только он ввел какое-либо отрицательное число, считывание прекращается и мы вычисляем среднее арифметическое введенных значений. Это можно реализовать так:

```
digs = [0]*100
N = 0; x = 0
while x >= 0:
    x = int(input("Введите целое число: "))
    digs[N] = x
    N += 1

S = 0
for x in range(N):
    S += digs[x]
S = S/N;

print("S = %f, N = %d"%(S, N))
```

Теперь, когда мы в целом познакомились со списками, отметим следующие моменты. Список может состоять из произвольных данных, например:

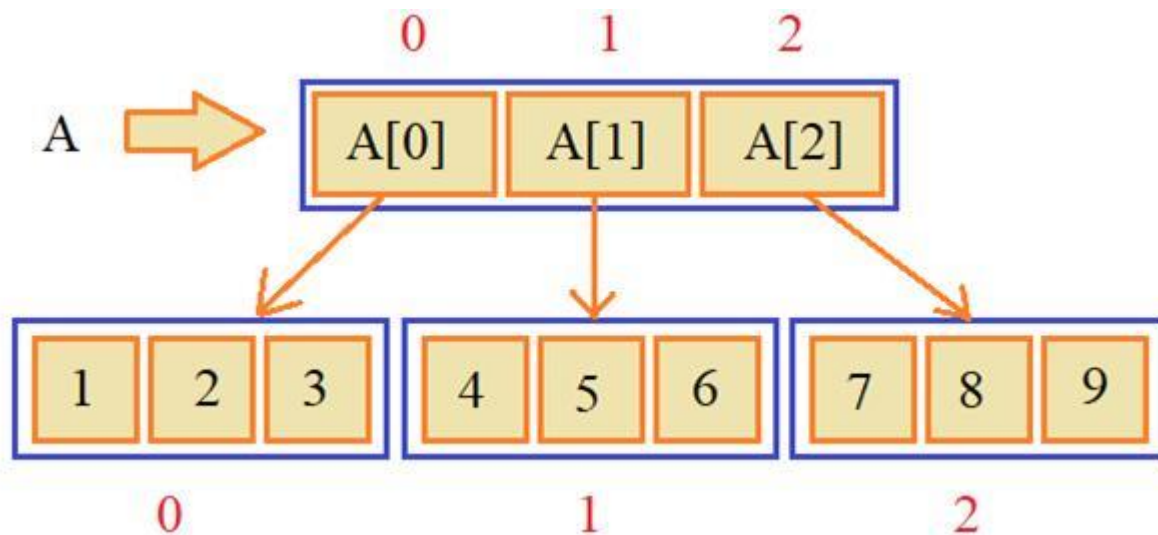
```
t = ["строка", 5, 5.7, True, [1,2,3]]
```

Причем, его длина

```
len(t)
```

будет равна 5, т.к. последний элемент – вложенный список здесь воспринимается как один отдельный элемент. И этот пример показывает как можно создавать двумерные списки:

```
A = [[1,2,3], [4,5,6], [7,8,9]]
```



Для доступа к конкретному числу следует сначала обратиться к первому списку:

```
A[1]
```

а, затем, ко второму:

```
A[1][0]
```

Получим значение 4. С этими списками можно выполнять все те же самые операции, о которых мы говорили ранее, например, изменить значение:

```
A[1][2] = -1
```

Далее, списки можно объединять друг с другом, используя оператор `+`:

```
[1,2,3] + ["Москва", "Тверь"]
```

Используя этот оператор, можно добавлять новые элементы к списку:

```
digs = [1,2,3,4]
digs = digs + [5]
digs += [6]
```

или в начало:

```
digs = ["числа"]+digs
```

И здесь обратите внимание, что мы объединяем именно списки, то есть, вот такая запись:

```
digs = digs+3
```

приведет к ошибке, т.к. 3 – это число, а не список.

Следующий оператор:

```
3 in digs
```

возвращает True, если элемент, записанный слева, присутствует в списке, указанный справа. Иначе, значение False:

```
-1 in digs
```

Или, можно делать так:

```
[1,2,3] in A
```

То есть, в качестве элемента может быть любой тип данных.

Следующие две полезные функции:

```
digs = [1,2,3,4]  
max(digs)  
min(digs)
```

находят минимальное или максимальное числовое значение. И если в списке имеется не числовой элемент:

```
digs += "a"
```

то эти функции приводят к ошибкам.

Также можно вычислять сумму элементов числового списка:

```
d = [1,2,3,4]  
sum(d)
```

выполнять сортировку чисел по возрастанию:

```
d = [-1, 0, 5, 3, 2, 5]  
sorted(d)
```

или, по убыванию:

```
sorted(d, reverse=True)
```

Эта функция возвращает новый объект-список, прежний d остается без изменений.

Наконец, можно сравнивать списки между собой:

```
[1,2,3] == [1,2,3]
[1,2,3] != [1,2,3]
[1,2,3] > [1,2,3]
```

В последнем сравнении получим False, т.к. списки равны, но если записать так:

```
[10,2,3] > [1,2,3]
```

то первый список будет больше второго. Здесь сравнение больше, меньше выполняется по тому же принципу, что и у строк: перебираются последовательно элементы, и если текущий элемент первого списка больше соответствующего элемента второго списка, то первый список больше второго. И аналогично, при сравнении меньше:

```
[10,2,3] < [1,2,3]
```

Все эти сравнения работают с однотипными данными:

```
[1,2, "abc"] > [1,2, "abc"]
```

сработает корректно, а вот так:

```
[1,2,3] > [1,2, "abc"]
```

Произойдет ошибка, т.к. число 3 не может быть сравнено со строкой «abc».

Для списков применим механизм срезов, о котором мы уже говорили, рассматривая строки. Например, пусть у нас имеется список из городов:

```
lst = ["Москва", "Санкт-Петербург", "Тверь", "Казань"]
```

Используя синтаксис:

список[start:end]

можно выделять элементы, начиная с индекса start и заканчивая, но не включая индекс end. В частности, вот такая конструкция:

```
lst[1:3]
```

возвратит список из двух городов:

```
<p align=center>['Санкт-Петербург', 'Тверь']
```

То есть, здесь создается новый объект list, содержащий эти элементы:

```
lst2 = lst[2:4]
```

Прежний список lst не меняется. Если индексы принимают отрицательные значения, то отсчет идет с конца списка:

```
lst[-2:-1]
```

получим

```
['Тверь']
```

так как индекс -1 – последний элемент не включается, остается только «Тверь». Или, можно записать так:

```
lst[0:-1]
```

тогда возьмем с первого элемента и до предпоследнего:

```
['Москва', 'Санкт-Петербург', 'Тверь']
```

У срезов можно записывать любые числовые индексы к ошибкам это не приведет. Например:

```
lst[1:9999]
```

вернет список со 2-го элемента и по последний:

```
['Санкт-Петербург', 'Тверь', 'Казань']
```

Этот же результат можно получить и так:

```
lst[1:]
```

то есть, не указывая последний индекс, берутся все оставшиеся элементы. Если же записать срез так:



```
lst[:3]
```

то элементы выбираются с самого начала и до третьего индекса, то есть, получим:

```
['Москва', 'Санкт-Петербург', 'Тверь']
```

Если не указывать ни начало, ни конец, то будет возвращен список:

```
lst[:]
```

Спрашивается: создает ли данная операция копию списка? Да, создается и в этом легко убедиться, записав такие строки:

```
c = lst[:]
print(id(c), id(lst))
```

И мы увидим разные значения id, которые говорят, что обе переменные ссылаются на разные списки. Также копию списка, можно сделать с помощью функции-конструктора list:

```
c=list(lst)
```

Далее, в срезах можно указывать шаг следования (по умолчанию он равен 1). Для этого пишется еще одно двоеточие и указывается шаг:

```
lst[::2]
```

получим:

```
['Москва', 'Тверь']
```

Или, такие варианты:

```
lst[1:4:2]
lst[:4:3]
lst[1::2]
```

Если указать отрицательный шаг, то перебор будет происходить в обратном порядке:

```
lst[::-1]
```

Так как список – это изменяемый объект, то мы можем срезам присваивать новые значения. Делается это таким образом:

```
lst[1:3] = "Владимир", "Астрахань"
```

```
lst = ["Москва", "Санкт-Петербург", "Тверь", "Казань"]
```



В результате, получаем список:

```
['Москва', 'Владимир', 'Астрахань', 'Казань']
```

Или даже так. Большему срезу присвоить меньшее число элементов:

```
lst[0:3] = 'Пермь', 'Пенза'
```

В итоге получаем список:

```
['Пермь', 'Пенза', 'Казань']
```

Однако, если нам нужно просто удалить какой-либо элемент, то это делается с помощью оператора del:

```
del lst[1]
```

В результате будет удален элемент с индексом 1 из списка lst:

```
['Пермь', 'Казань']
```

## Методы списков

Давайте теперь предположим, что у нас имеется список из чисел:

```
a = [1, -54, 3, 23, 43, -45, 0]
```

и мы хотим в конец этого списка добавить значение. Это можно сделать с помощью метода:

```
a.append(100)
```

И обратите внимание: метод `append` ничего не возвращает, то есть, он меняет сам список благодаря тому, что он относится к изменяемому типу данных. Поэтому писать здесь конструкцию типа

```
a = a.append(100)
```

категорически не следует, так мы только потеряем весь наш список! И этим методы списков отличаются от методов строк, когда мы записывали:

```
string="Hello"  
string = string.upper()
```

Здесь метод `upper` возвращает измененную строку, поэтому все работает как и ожидается. А метод `append` ничего не возвращает, и присваивать значение `None` переменной `a` не имеет смысла, тем более, что все работает и так:

```
a = [1, -54, 3, 23, 43, -45, 0]  
a.append(100)
```

Причем, мы в методе `append` можем записать не только число, но и другой тип данных, например, строку:

```
a.append("hello")
```

тогда в конец списка будет добавлен этот элемент. Или, булево значение:

```
a.append(True)
```

Или еще один список:

```
a.append([1,2,3])
```

И так далее. Главное, чтобы было указано одно конкретное значение. Вот так работать не будет:

```
a.append(1,2)
```

Если нам нужно вставить элемент в произвольную позицию, то используется метод

```
a.insert(3, -1000)
```

Здесь мы указываем индекс вставляемого элемента и далее значение самого элемента.

Следующий метод `remove` удаляет элемент по значению:

```
a.remove(True)
a.remove('hello')
```

Он находит первый подходящий элемент и удаляет его, остальные не трогает. Если же указывается несуществующий элемент:

```
a.remove('hello2')
```

то возникает ошибка. Еще один метод для удаления

```
a.pop()
```

выполняет удаление последнего элемента и при этом, возвращает его значение. В самом списке последний элемент пропадает. То есть, с помощью этого метода можно сохранять удаленный элемент в какой-либо переменной:

```
end = a.pop()
```

Также в этом методе можно указывать индекс удаляемого элемента, например:

```
a.pop(3)
```

Если нам нужно очистить весь список – удалить все элементы, то можно воспользоваться методом:

```
a.clear()
```

Получим пустой список. Следующий метод

```
a = [1, -54, 3, 23, 43, -45, 0]
c = a.copy()
```

возвращает копию списка. Это эквивалентно конструкции:

```
c = list(a)
```

В этом можно убедиться так:

```
c[1] = 1
```

и список `c` будет отличаться от списка `a`.

Следующий метод count позволяет найти число элементов с указанным значением:

```
c.count(1)  
c.count(-45)
```

Если же нам нужен индекс определенного значения, то для этого используется метод index:

```
c.index(-45)  
c.index(1)
```

возвратит 0, т.к. берется индекс только первого найденного элемента. Но, мы здесь можем указать стартовое значение для поиска:

```
c.index(1, 1)
```

Здесь поиск будет начинаться с индекса 1, то есть, со второго элемента. Или, так:

```
c.index(23, 1, 5)
```

Ищем число 23 с 1-го индекса и по 5-й не включая его. Если элемент не находится

```
c.index(23, 1, 3)
```

то метод приводит к ошибке. Чтобы этого избежать в своих программах, можно вначале проверить: существует ли такой элемент в нашем срезе:

```
23 in c[1:3]
```

и при значении True далее уже определять индекс этого элемента.

Следующий метод

```
c.reverse()
```

меняет порядок следования элементов на обратный.

Последний метод, который мы рассмотрим, это

```
c.sort()
```

выполняет сортировку элементов списка по возрастанию. Для сортировки по убыванию, следует этот метод записать так:

```
c.sort(reverse=True)
```

Причем, этот метод работает и со строками:

```
lst = ["Москва", "Санкт-Петербург", "Тверь", "Казань"]  
lst.sort()
```

Здесь используется лексикографическое сравнение, о котором мы говорили, когда рассматривали строки.

Это все основные методы списков и чтобы вам было проще ориентироваться, приведу следующую таблицу:

Метод	Описание
append()	Добавляет элемент в конец списка
insert()	Вставляет элемент в указанное место списка
remove()	Удаляет элемент по значению
pop()	Удаляет последний элемент, либо элемент с указанным индексом
clear()	Очищает список (удаляет все элементы)
copy()	Возвращает копию списка
count()	Возвращает число элементов с указанным значением
index()	Возвращает индекс первого найденного элемента
reverse()	Меняет порядок следования элементов на обратный
sort()	Сортирует элементы списка

Вот мы с вами и подошли к одному из фундаментальных моментов в изучении языка Python – функциям. Что это такое? Смотрите. Например, уже знакомая вам функция

```
print()
```

выводит сообщения в консоль. Фактически же при ее вызове выполняется определенный фрагмент программы, результатом которого и является вывод информации в заданном виде. И это очень удобно. Благодаря наличию таких функций нам не нужно каждый раз писать дублирующие инструкции для выполнения типовых операций. Собственно, это главное предназначение функций – многократное выполнение определенного фрагмента программы.

Язык Python позволяет программисту создавать свои собственные функции. Для этого используется следующий синтаксис:

```
def <имя функции>([список аргументов]):  
    оператор 1  
    оператор 2  
    ...  
    оператор N
```

Здесь имя функции придумывается программистом подобно именам переменных и, так как функция – это определенное действие, то ее имя следует выбирать как глагол, например:

go, show, get, set и т.п.

Далее, идет набор операторов, которые образуют **тело функции**. Именно они начинают выполняться при ее вызове.

Давайте зададим простейшую функцию, которая будет выводить «hello» в консоль:

```
def sayHello():  
    print("hello")
```

Смотрите, мы здесь придумали имя функции «sayHello», записали пустые круглые скобки без аргументов и через двоеточие определили тело функции в виде конструкции print("hello"). Но это лишь определение функции. самого вызова здесь еще нет и если запустить программу, то ничего не произойдет.

Чтобы вызвать эту функцию, нужно указать ее имя и в конце обязательно поставить круглые скобки даже если мы не передаем ей никаких аргументов:

```
sayHello()
```

Эти круглые скобки являются оператором вызова функции с указанным именем. Теперь, при запуске программы в консоли появится сообщение «hello».

Имя функции без круглых скобок – это фактически ссылка на функцию:

```
print( type(sayHello) )
```

то есть, ссылка на специальный объект, представляющий ту или иную функцию. А раз это ссылка, то мы можем выполнить такую операцию:

```
f = sayHello
```

тем самым определить ее синоним и вызвать ее уже через это второе имя:

```
f()
```

Как мы говорили в самом начале, функции, как правило, создаются для их многократного вызова. И действительно, мы теперь, можем ее вызывать в любом месте нашей программы необходимое число раз, например, так:

```
sayHello()  
print("-----")  
sayHello()
```

Здесь будет уже два вызова этой функции. И так далее. Причем, обратите внимание, мы вызываем функцию только после ее определения. То есть, если записать ее вызвать в самом начале программы, то возникнет ошибка, т.к. данная функция не была определена. Это вроде как:

"сначала нужно испечь пирог и только потом можно его есть."

Также и с функциями: мы их сначала определяем и только потом можем вызывать. Поэтому определение функций обычно идет в самом начале, а потом уже их вызовы в основной программе.

Если нужно определить еще одну функцию, то мы ее можем записать после первой:

```
def myAbs(x):  
    x = -x if(x<0) else x
```

Имена функций должны быть уникальными (также как и имена переменных), поэтому я назвал ее myAbs, т.к. функция abs уже существует. И предполагаю, что она будет вычислять модуль переданного ей числа. Соответственно, в круглых скобках обозначаю этот аргумент. Если теперь мы ее вызовем:

```
print( myAbs(-5) )
```

то увидим значение None. Это произошло потому, что функция myAbs явно не возвращает никакого значения. По идее, мы ожидаем возврата переменной x. Для этого нужно записать оператор return, после которого через пробел указываем возвращаемую величину:



```
def myAbs(x):  
    x = -x if(x<0) else x  
    return x
```

Теперь, при вызове функции, получим ожидаемое значение 5. Как это в деталях работает? Вызывая функцию с аргументом -5, переменная x начинает ссылаться на этот числовой объект. Далее, выполняется тело функции и идет проверка: если  $x < 0$ , то  $x = -x$  (меняем знак числа), иначе x не меняется. Затем, выполняется оператор return и функция myAbs возвращает вычисленное значение x.

Такой подход позволяет передавать функции самые разные значения, например, так:

```
print( myAbs(15) )  
a = 100  
print( myAbs(a) )
```

И это делает ее работу универсальной – с любыми числовыми данными. Причем, как только встречается оператор return функция завершает свою работу. То есть, если после данного оператора будут идти еще какие-либо конструкции:

```
def myAbs(x):  
    x = -x if(x<0) else x  
    return x  
    print(x)
```

То при вызове этой функции:

```
val = myAbs(-5.8)
```

Ничего в консоль выведено не будет, т.к. вызов был завершен на операторе return. А вот если поставить print до этого оператора:

```
def myAbs(x):  
    x = -x if(x<0) else x  
    print(x)  
    return x
```

то мы увидим значение 5.8. Используя эту особенность, можно определять такие функции:

```
def isPositive(x):  
    if x >= 0:  
        return True
```

```
else:  
    return False
```

В данном случае мы будем получать значения True для неотрицательных чисел и False – для отрицательных. И далее, ее можно использовать так:

```
p = []  
for a in range(-5, 11):  
    if isPositive(a):  
        p.append(a)  
  
print(p)
```

В результате, список будет содержать только положительные числа. Правда, в данном случае, функцию можно записать гораздо короче:

```
def isPositive(x):  
    return x >= 0
```

Здесь сам оператор `>=` будет возвращать значение True или False.

Если нужно создать функцию, принимающую два аргумента, например, для вычисления площади прямоугольника, то это делается так:

```
def getSquare(w, h):  
    return 2*(w+h)
```

То есть, аргументы перечисляются через запятую, а тело функции состоит всего из одного оператора `return`, в котором сразу выполняются необходимые вычисления.

Вызовем эту функцию:

```
p = getSquare(10, 5.5)  
print(p)
```

И увидим результат ее работы – значение 31,0. При этом, на первое значение 10 ссылается первый аргумент `w`, а на второе 5.5 – второй аргумент `h`. Вот так можно определять различное число аргументов у функций.

Далее, при вызове функций мы должны им передавать ровно столько параметров, сколько указано в их определении. Например, вот такие вызовы работать не будут:

```
myAbs()  
myAbs(1, 2)  
sayHello("abc")
```

Здесь указано или слишком много, или слишком мало **фактических параметров**.

Однако у любой функции можно добавить **формальные параметры** со значениями по умолчанию:

```
def sayHello(msg, end="!"):
    print(msg+end)
```

И теперь, можно вызвать эту функцию так:

```
sayHello("Hello")
```

или так:

```
sayHello("Hello", "?")
```

Смотрите, если формальный параметр не указан, то берется его значение по умолчанию. Если же мы его явно задаем, то берется переданное значение. Здесь нужно помнить только одно правило: формальные аргументы должны быть записаны последними в списке аргументов функции. То есть, вот такая запись:

```
def sayHello(end="!", msg):
```

приведет к синтаксической ошибке.

Теперь, давайте добавим этой функции еще один вот такой формальный параметр:

```
def sayHello(msg, end="!", sep = ": "):
    print("Message"+sep+msg+end)
```

И функция будет выводить сообщение в формате: «Message»+sep+msg+end. Вызвать эту функцию мы можем таким образом:

```
sayHello("Hello", "?", " ")
```

и каждому параметру здесь будет соответствовать свое значение в соответствии с указанным порядком. А можно ли вызвать эту функцию, указав только первый и последний аргумент? Оказывается да, Python позволяет это делать. Вот таким образом:

```
sayHello("Hello", sep=" ")
```

Мы здесь вторым аргументом явно указываем имя формального параметра и присваиваем ему желаемое значение. В результате аргументы `msg` и `sep` будут принимать переданные значения, а аргумент `end` – значение по умолчанию. Это называется **именованные параметры**, когда мы указываем не просто значение, но еще и имя параметра.

Если нам требуется сразу вернуть несколько значений, то это можно сделать так. Предположим наша функция будет сразу определять и периметр и площадь прямоугольника:

```
def perAndSq(w, h):  
    return 2*(w+h), w*h
```

И, далее, вызываем ее:

```
res = perAndSq(2.3, 5)  
print(res)
```

получаем результат в виде кортежа из двух чисел. Или, так:

```
per, sq = perAndSq(2.3, 5)  
print(per, sq)
```

Аналогичным образом можно возвращать и списки и словари и вообще любые типы данных.

Далее, в теле функции можно записывать самые разные конструкции языка Python. Например, для возведения числа в целую степень, можно определить такую функцию:

```
def myPow(x, n):  
    sx = 1  
    while n > 0:  
        sx *= x  
        n -= 1  
    return sx
```

И, затем, вызвать ее:

```
p = myPow(3, 5)  
print(p)
```

Интересной особенностью Python в определении функций является возможность переопределять уже существующие функции. Например, у нас задана вот такая функция:

```
def sayHello():  
    print("hello")
```

Тогда ниже мы можем ее переопределить, если укажем то же самое имя:

```
def sayHello():  
    print("----- hello -----")
```

Теперь, при ее вызове:

```
sayHello()
```

увидим выполнение последнего, переопределенного варианта. Если дальше ее переопределить вот так:

```
def sayHello(msg):  
    print(msg)
```

то все равно будет доступна только одна такая функция, но теперь уже с одним обязательным аргументом:

```
sayHello("привет мир")
```

Когда это может пригодиться на практике? Например, если мы хотим определить некоторую функцию в зависимости от условия:

```
TYPE_FUNC = True
```

```
if TYPE_FUNC:  
    def sayHello():  
        print("hello")  
else:  
    def sayHello(msg):  
        print(msg)
```

```
sayHello()
```

Здесь при значении переменной TYPE\_FUNC равной True будет определен первый вариант функции, а иначе – второй вариант. Иногда это бывает полезно.

## Элементы функционального подхода к программированию

При написании программ приветствуется такой подход, который называется **функциональным программированием**.

Продемонстрирую его на следующем примере. Предположим, нам нужна функция, которая находит максимальное значение из двух чисел:

```
def max2(a, b):  
    if a > b:  
        return a  
    return b
```

И вызвать мы ее можем так:

```
print( max2(2, -3) )
```

Затем, нам потребовалась функция, которая бы находила максимальное из трех чисел. Как ее можно реализовать? Используя идею функционального программирования, это можно сделать следующим образом:

```
def max3(a, b, c):  
    return max2(a, max2(b, c))
```

И вызвать так:

```
print( max3(2, -3, 5) )
```

Смотрите, здесь оператор return возвращает значение, которое возвращает функция max2. Но, прежде чем она будет выполнена, вызовется другая функция max2, которая определит максимальное среди чисел b и c. То есть, прежде чем вызвать первую функцию max2 необходимо вычислить ее параметры: первый просто берется их x, а второй вычисляется вложенной функцией max2. Вот так это работает и вот что из себя представляет элемент функционального подхода к программированию.

Причем, благодаря гибкости языка Python, мы можем вызвать эту функцию и для нахождения максимальной строки:

```
print( max3("ab", "cd", "abc") )
```

так как строки могут спокойно сравниваться между собой. И вообще, любые величины, которые можно сравнивать на больше и меньше, можно подставлять в качестве аргументов функции max3 и max2.

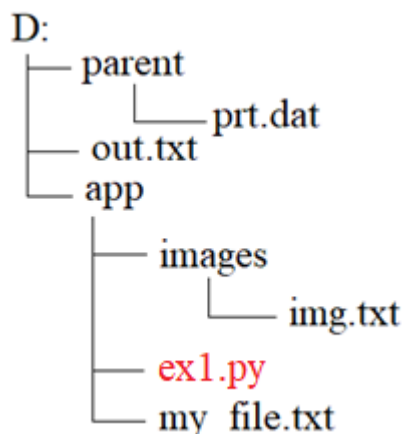
как в Python можно считывать информацию из файлов и записывать ее в файлы. Что такое файлы и зачем они нужны, думаю объяснять не надо, т.к. если вы дошли до этого занятия, значит, проблем с пониманием таких базовых вещей у вас нет. Поэтому сразу перейдем к функции

```
open(file [, mode='r', encoding=None, ...])
```

через которую и осуществляется работа с файлами. Здесь

- file – это путь к файлу вместе с его именем;
- mode – режим доступа к файлу;
- encoding – кодировка файла.

Для начала определимся с понятием «путь к файлу». Представим, что наш файл ex1.py находится в каталоге app:



Тогда, чтобы обратиться к файлу my\_file.txt путь можно записать так:

"my\_file.txt"

или

"d:\\app\\my\_file.txt"

или так:

"d:/app/my\_file.txt"

Последние два варианта представляют собой абсолютный путь к файлу, то есть, полный путь, начиная с указания диска. Причем, обычно используют обратный слеш в качестве разделителя: так короче писать и такой путь будет корректно восприниматься как под ОС Windows, так и Linux. Первый же вариант – это относительный путь, относительно рабочего каталога.

Теперь, предположим, мы хотим обратиться к файлу `img.txt`. Это можно сделать так:

```
"images/img.txt"
```

или так:

```
"d:/app/images/img.txt"
```

Для доступа к `out.txt` пути будут записаны так:

```
"../out.txt"
```

```
"d:/out.txt"
```

Обратите внимание, здесь две точки означают переход к родительскому каталогу, то есть, выход из каталога `app` на один уровень вверх.

И, наконец, для доступа к файлу `prt.dat` пути запишутся так:

```
"../parent/prt.dat"
```

```
"d:/ parent/prt.dat"
```

Вот так следует прописывать пути к файлам. В нашем случае мы имеем текстовый файл «`myfile.txt`», который находится в том же каталоге, что и программа `ex1.py`, поэтому путь можно записать просто указав имя файла:

```
file = open("myfile.txt")
```

В результате переменная `file` будет ссылаться на файловый объект, через который и происходит работа с файлами. Если указать неверный путь, например, так:

```
file = open("myfile2.txt")
```

то возникнет ошибка `FileNotFoundError`. Это стандартное исключение и как их обрабатывать мы с вами говорили на предыдущем занятии. Поэтому, запишем этот критический код в блоке `try`:

```
try:  
    file = open("myfile2.txt")  
except FileNotFoundError:  
    print("Невозможно открыть файл")
```



Изменим имя файла на верное и посмотрим, как далее можно с ним работать. По умолчанию функция `open` открывает файл в текстовом режиме на чтение. Это режим

```
mode = "r"
```

Если нам нужно поменять режим доступа к файлу, например, открыть его на запись, то это явно указывается вторым параметром функции `open`:

```
file = open("out.txt", "w")
```

В Python имеются следующие режимы доступа:

Название	Описание
'r'	открытие на чтение (значение по умолчанию)
'w'	открытие на запись (содержимое файла удаляется, а если его нет, то создается новый)
'x'	открытие файла на запись, если его нет генерирует исключение
'a'	открытие на дозапись (информация добавляется в конец файла)
Дополнения	
'b'	открытие в бинарном режиме доступа к информации файла
't'	открытие в текстовом режиме доступа (если явно не указывается, то используется по умолчанию)
'+'	открытие на чтение и запись одновременно

Здесь мы имеем три основных режима доступа: на чтение, запись и добавление. И еще три возможных расширения этих режимов, например,

- 'rt' – чтение в текстовом режиме;
- 'wb' – запись в бинарном режиме;
- 'a+' – дозапись или чтение данных из файла.

### Чтение информации из файла

В чем отличие текстового режима от бинарного мы поговорим позже, а сейчас откроем файл на чтение в текстовом режиме:

```
file = open("myfile.txt")
```

и прочитаем его содержимое с помощью метода read:

```
print( file.read() )
```

В результате, получим строку, в которой будет находиться прочитанное содержимое. Действительно, в этом файле находятся эти строки из поэмы Пушкина А.С. «Медный всадник». И здесь есть один тонкий момент. Наш текстовый файл имеет кодировку Windows-1251 и эта кодировка используется по умолчанию в функции read. Но, если изменить кодировку файла, например, на популярную UTF-8, то после запуска программы увидим в консоли вот такую белиберду. Как это можно исправить, не меняя кодировки самого файла? Для этого следует воспользоваться именованным параметром encoding и записать метод open вот так:

```
file = open("myfile.txt", encoding="utf-8" )
```

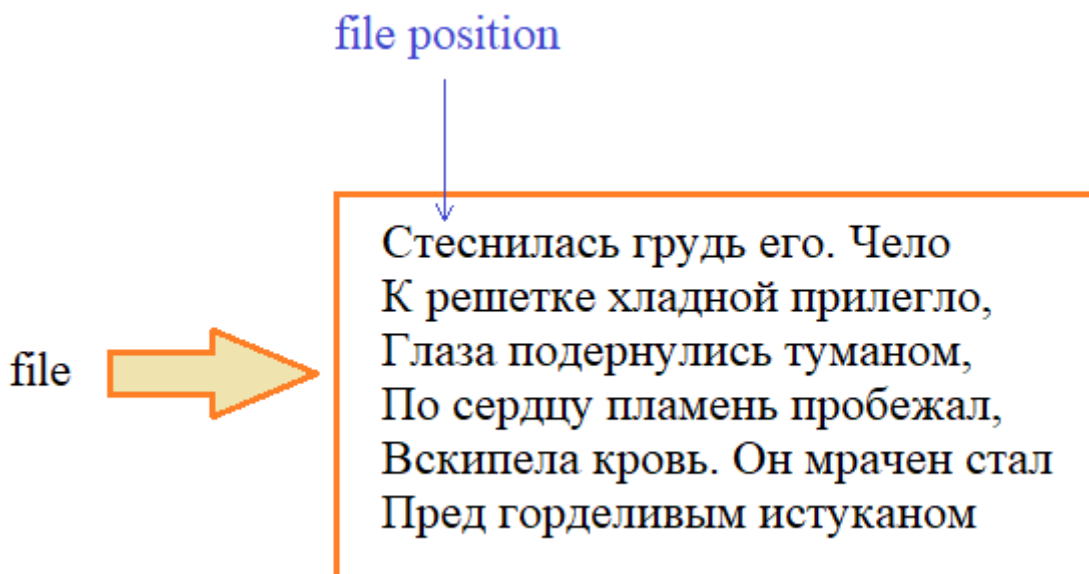
Теперь все будет работать корректно. Далее, в методе read мы можем указать некий числовой аргумент, например,

```
print( file.read(2) )
```

Тогда из файла будут считаны первые два символа. И смотрите, если мы запишем два таких вызова подряд:

```
print( file.read(2) )  
print( file.read(2) )
```

то увидим, что при следующем вызове метод read продолжил читать следующие два символа. Почему так произошло? Дело в том, что у файлового объекта, на который ссылается переменная file, имеется внутренний указатель позиции (file position), который показывает с какого места производить считывание информации.



Когда мы вызываем метод `read(2)` эта позиция автоматически сдвигается от начала файла на два символа, т.к. мы именно столько считываем. И при повторном вызове `read(2)` считывание продолжается, т.е. берутся следующие два символа. Соответственно, позиция файла сдвигается дальше. И так, пока не дойдем до конца.

Но мы в Python можем управлять этой файловой позицией с помощью метода

`seek(offset[, from_what])`

Например, вот такая запись:

```
file.seek(0)
```

будет означать, что мы устанавливаем позицию в начало и тогда такие строчки:

```
print( file.read(2) )  
file.seek(0)  
print( file.read(2) )
```

будут считывать одни и те же первые символы. Если же мы хотим узнать текущую позицию в файле, то следует вызвать метод `tell`:

```
pos = file.tell()  
print( pos )
```

Следующий полезный метод – это `readline` позволяет построчно считывать информацию из текстового файла:

```
s = file.readline()
print( s )
```

Здесь концом строки считается символ переноса 'n', либо конец файла. Причем, этот символ переноса строки будет также присутствовать в строке. Мы в этом можем убедиться, вызвав дважды эту функцию:

```
print( file.readline() )
print( file.readline() )
```

Здесь в консоли строчки будут разделены пустой строкой. Это как раз из-за того, что один перенос идет из прочитанной строки, а второй добавляется самой функцией print. Поэтому, если их записать вот так:

```
print( file.readline(), end="" )
print( file.readline(), end="" )
```

то вывод будет построчным с одним переносом.

Если нам нужно последовательно прочитать все строчки из файла, то для этого обычно используют цикл for следующим образом:

```
for line in file:
    print( line, end="" )
```

Этот пример показывает, что объект файл является итерируемым и на каждой итерации возвращает очередную строку.

Или же, все строчки можно прочитать методом

```
s = file.readlines()
```

и тогда переменная s будет ссылаться на упорядоченный список с этими строками:

```
print( s )
```

Однако этот метод следует использовать с осторожностью, т.к. для больших файлов может возникнуть ошибка нехватки памяти для хранения полученного списка.

По сути это все методы для считывания информации из файла. И, смотрите, как только мы завершили работу с файлом, его следует закрыть. Для этого используется метод close:

```
file.close()
```

Конечно, прописывая эту строчку, мы не увидим никакой разницы в работе программы. Но, во-первых, закрывая файл, мы освобождаем память, связанную с этим файлом и, во-вторых, у нас не будет проблем в потере данных при их записи в файл. А, вообще, лучше просто запомнить: после завершения работы с файлом, его нужно закрыть. Причем, организовать программу лучше так:

```
try:
    file = open("myfile.txt")

    try:
        s = file.readlines()
        print( s )
    finally:
        file.close()

except FileNotFoundError:
    print("Невозможно открыть файл")
```

Мы здесь создаем вложенный блок try, в который помещаем критический текст программы при работе с файлом и далее блок finally, который будет выполнен при любом стечении обстоятельств, а значит, файл гарантированно будет закрыт.

Или же, забегаая немного вперед, отмечу, что часто для открытия файла пользуются так называемым **менеджером контекста**, когда файл открывают при помощи оператора with:

```
try:
    with open("myfile.txt", "r") as file:    # file = open("myfile.txt")
        s = file.readlines()
        print( s )

except FileNotFoundError:
    print("Невозможно открыть файл")
```

При таком подходе файл закрывается автоматически после выполнения всех инструкций внутри этого менеджера. В этом можно убедиться, выведем в консоль флаг, сигнализирующий закрытие файла:

```
finally:
    print(file.closed)
```

Запустим программу, видите, все работает также и при этом файл автоматически закрывается. Даже если произойдет критическая ошибка, например, пропишем такую конструкцию:

```
print( int(s) )
```

то, как видим, файл все равно закрывается. Вот в этом удобство такого подхода при работе с файлами.

## Запись информации в файл

Теперь давайте посмотрим, как происходит запись информации в файл. Во-первых, нам нужно открыть файл на запись, например, так:

```
file = open("out.txt", "w")
```

и далее вызвать метод write:

```
file.write("Hello World!")
```

В результате у нас будет создан файл out.txt со строкой «Hello World!». Причем, этот файл будет располагаться в том же каталоге, что и файл с текстом программы на Python.

Далее сделаем такую операцию: запишем метод write следующим образом:

```
file.write("Hello")
```

И снова выполним эту программу. Смотрите, в нашем файле out.txt прежнее содержимое исчезло и появилось новое – строка «Hello». То есть, когда мы открываем файл на запись в режимах

w, wt, wb,

то прежнее содержимое файла удаляется. Вот этот момент следует всегда помнить.

Теперь посмотрим, что будет, если вызвать метод write несколько раз подряд:

```
file.write("Hello1")  
file.write("Hello2")  
file.write("Hello3")
```

Смотрите, у нас в файле появились эти строчки друг за другом. То есть, здесь как и со считыванием: объект file записывает информацию,

начиная с текущей файловой позиции, и автоматически перемещает ее при выполнении метода `write`.

Если мы хотим записать эти строки в файл каждую с новой строки, то в конце каждой пропишем символ переноса строки:

```
file.write("Hello1\n")
file.write("Hello2\n")
file.write("Hello3\n")
```

Далее, для дозаписи информации в файл, то есть, записи с сохранением предыдущего содержимого, файл следует открыть в режиме `'a'`:

```
file = open("out.txt", "a")
```

Тогда, выполняя эту программу, мы в файле увидим уже шесть строчек. И смотрите, в зависимости от режима доступа к файлу, мы должны использовать или методы для записи, или методы для чтения. Например, если вот здесь попытаться прочитать информацию с помощью метода `read`:

```
file.read()
```

то возникнет ошибка доступа. Если же мы хотим и записывать и считывать информацию, то можно воспользоваться режимом `a+`:

```
file = open("out.txt", "a+")
```

Так как здесь файловый указатель стоит на последней позиции, то для считывания информации, поставим его в самое начало:

```
file.seek(0)
print( file.read() )
```

А вот запись данных всегда осуществляется в конец файла.

Следующий полезный метод для записи информации – это `writelines`:

```
file.writelines(["Hello1\n", "Hello2\n"])
```

Он записывает несколько строк, указанных в коллекции. Иногда это бывает удобно, если в процессе обработки текста мы имеем список и его требуется целиком поместить в файл.

**Чтение и запись в бинарном режиме доступа**

Что такое бинарный режим доступа? Это когда данные из файла считываются один в один без какой-либо обработки. Обычно это используется для сохранения и считывания объектов. Давайте предположим, что нужно сохранить в файл вот такой список:

```
books = [  
    ("Евгений Онегин", "Пушкин А.С.", 200),  
    ("Муму", "Тургенев И.С.", 250),  
    ("Мастер и Маргарита", "Булгаков М.А.", 500),  
    ("Мертвые души", "Гоголь Н.В.", 190)  
]
```

Откроем файл на запись в бинарном режиме:

```
file = open("out.bin", "wb")
```

Далее, для работы с бинарными данными подключим специальный встроенный модуль pickle:

```
import pickle
```

И вызовем него метод dump:

```
pickle.dump(books, file)
```

Все, мы сохранили этот объект в файл. Теперь прочитаем эти данные. Откроем файл на чтение в бинарном режиме:

```
file = open("out.bin", "rb")
```

и далее вызовем метод load модуля pickle:

```
bs = pickle.load(file)
```

Все, теперь переменная bs ссылается на эквивалентный список:

```
print( bs )
```

Аналогичным образом можно записывать и считывать сразу несколько объектов. Например, так:

```
import pickle
```

```
book1 = ["Евгений Онегин", "Пушкин А.С.", 200]  
book2 = ["Муму", "Тургенев И.С.", 250]  
book3 = ["Мастер и Маргарита", "Булгаков М.А.", 500]  
book4 = ["Мертвые души", "Гоголь Н.В.", 190]
```



```

try:
    file = open("out.bin", "wb")

    try:
        pickle.dump(book1, file)
        pickle.dump(book2, file)
        pickle.dump(book3, file)
        pickle.dump(book4, file)

    finally:
        file.close()

except FileNotFoundError:
    print("Невозможно открыть файл")

```

А, затем, считывание в том же порядке:

```

file = open("out.bin", "rb")
b1 = pickle.load(file)
b2 = pickle.load(file)
b3 = pickle.load(file)
b4 = pickle.load(file)

print( b1, b2, b3, b4, sep="\n" )

```

Вот так в Python выполняется запись и считывание данных из файла.

# Программирование на C++.

## Код первой программы

Наберите следующий код:

```
#include <iostream>
#include <cstdlib> // для system
using namespace std;

int main()
{
    cout << "Hello, world!" << endl;
    system("pause"); // Только для тех, у кого MS Visual Studio
    return 0;
}
```

Описание синтаксиса

Директива `#include` используется для подключения других файлов в код. Строка `#include <iostream>`, будет заменена содержимым файла «`iostream.h`», который находится в стандартной библиотеке языка и отвечает за ввод и вывод данных на экран.

`#include <cstdlib>` подключает стандартную библиотеку языка C. Это подключение необходимо для работы функции `system`.

Содержимое третьей строки — `using namespace std;` указывает на то, что мы используем по умолчанию пространство имен с названием «`std`». Все то, что находится внутри фигурных скобок функции `int main() {}` будет автоматически выполняться после запуска программы.

Строка `cout << "Hello, world!" << endl;` говорит программе выводить сообщение с текстом «**Hello, world**» на экран.

Оператор `cout` предназначен для вывода текста на экран командной строки. После него ставятся две угловые кавычки (`<<`). Далее идет текст, который должен выводиться. Он помещается в двойные кавычки. Оператор `endl` переводит строку на уровень ниже.

Если в процессе выполнения произойдет какой-либо сбой, то будет сгенерирован код ошибки, отличный от нуля. Если же работа программы завершилась без сбоев, то код ошибки будет равен нулю. Команда `return 0` необходима для того, чтобы передать операционной системе сообщение об удачном завершении программы.

— В конце каждой команды ставится **точка с запятой**.

## Компиляция и запуск

Теперь скомпилируйте и запустите программу. Тем, кто пользуется MS Visual Studio, нужно нажать сочетание клавиш «Ctrl+F5».

Пользователям GCC нужно выполнить следующие команды:

```
с++ имя_файла.cpp -o имя_выходного_бинарника # компиляция кода  
./имя_выходного_бинарника # запуск программы
```

Если программа собралась с первого раза, то хорошо. Если компилятор говорит о наличии ошибок, значит вы что-то сделали неправильно.

Прочитайте текст ошибки и попробуйте ее исправить своими силами. Если не получится, напишите о вашей проблеме в комментариях.

В качестве домашнего задания, переделайте эту программу так, чтобы вместо сообщения «Hello, World» выводилось сообщение «Hello, User».

## Типы данных

В языке C++ *все переменные* имеют определенный тип данных. Например, переменная, имеющая целочисленный тип не может содержать ничего кроме целых чисел, а переменная с плавающей точкой — только дробные числа.

**Тип данных** присваивается переменной при ее объявлении или инициализации. Ниже приведены основные типы данных языка C++, которые нам понадобятся.

Основные типы данных в C++

- **int** — целочисленный тип данных.
- **float** — тип данных с плавающей запятой.
- **double** — тип данных с плавающей запятой двойной точности.
- **char** — символьный тип данных.
- **bool** — логический тип данных.

## Объявление переменной

Объявление переменной в C++ происходит таким образом: сначала указывается тип данных для этой переменной а затем название этой переменной.

Пример объявления переменных

```
int a; // объявление переменной a целого типа.
```

`float b;` // объявление переменной `b` типа данных с плавающей запятой.

`double c = 14.2;` // инициализация переменной типа `double`.

`char d = 's';` // инициализация переменной типа `char`.

`bool k = true;` // инициализация логической переменной `k`.

- Заметьте, что в C++ **оператор присваивания (=)** — не является знаком равенства и не может использоваться для сравнения значений. Оператор равенства записывается как «двойное равно» — **==**.
- Присваивание используется для сохранения определенного значения в переменной. Например, запись вида `a = 10` задает переменной `a` значение числа 10.

## Простой калькулятор на C++

Сейчас мы напишем простую программу-калькулятор, которая будет принимать от пользователя два целых числа, а затем определять их сумму:

```
#include <iostream>
using namespace std;
```

```
int main()
{
    setlocale(0, "");
    /*7*/ int a, b; // объявление двух переменных a и b целого типа
    данных.
    cout << "Введите первое число: ";
    cin >> a; // пользователь присваивает переменной a какое-либо
    значение.
    cout << "Введите второе число: ";
    cin >> b;
    /*12*/ int c = a + b; // новой переменной c присваиваем значение
    суммы введенных пользователем данных.
    cout << "Сумма чисел = " << c << endl; // вывод ответа.
    return 0;
}
```

Разбор кода

В 7-й строке кода программы мы объявляем переменные «`a`» и «`b`» целого типа `int`. В следующей строке кода выводится сообщение пользователю, чтобы он ввел с клавиатуры первое число.

В 9-й строке стоит еще незнакомая вам конструкция — `cin >>`. С помощью нее у пользователя запрашивается ввод значения переменной «a» с клавиатуры. Аналогичным образом задается значение переменной «b».

В 12-й строке мы производим инициализацию переменной «c» суммой переменных «a» и «b». Далее находится уже знакомый вам оператор `cout`, который выводит на экран строку и значение переменной «c».

- При выводе переменных, они *не заключаются в кавычки*, в отличие от строк.

```
#include <iostream>
using namespace std;
```

```
int main()
{
    setlocale(0, "");
    double num;

    cout << "Введите произвольное число: ";
    cin >> num;

    if (num < 10) { // Если введенное число меньше 10.
        cout << "Это число меньше 10." << endl;
    } else { // иначе
        cout << "Это число больше либо равно 10." << endl;
    }
    return 0;
}
```

Если вы запустите эту программу, то при вводе числа, меньшего десяти, будет выводиться соответствующее сообщение.

Если введенное число окажется большим, либо равным десяти — отобразится другое сообщение.

## Оператор if

Оператор `if` служит для того, чтобы выполнить какую-либо операцию в том случае, когда условие является верным. *Условная конструкция* в C++ всегда записывается в круглых скобках после оператора `if`.

Внутри фигурных скобок указывается тело условия. Если условие выполнится, то начнется выполнение всех команд, которые находятся между фигурными скобками.

### Пример конструкции ветвления

```
if (num < 10) { // Если введенное число меньше 10.  
    cout << "Это число меньше 10." << endl;  
} else { // иначе  
    cout << "Это число больше либо равно 10." << endl;  
}
```

Здесь говорится: «**Если** переменная **num** меньше 10 — вывести соответствующее сообщение. **Иначе**, вывести другое сообщение».

Усовершенствуем программу так, чтобы она выводила сообщение, о том, что переменная **num** равна десяти:

```
if (num < 10) { // Если введенное число меньше 10.  
    cout << "Это число меньше 10." << endl;  
} else if (num == 10) {  
    cout << "Это число равно 10." << endl;  
} else { // иначе  
    cout << "Это число больше 10." << endl;  
}
```

Здесь мы проверяем три условия:

- Первое — когда введенное число меньше 10-ти
- Второе — когда число равно 10-ти
- И третье — когда число больше десяти

Заметьте, что во втором условии, при проверке равенства, мы используем оператор равенства — **==**, а не оператор присваивания, потому что мы не изменяем значение переменной при проверке, а сравниваем ее текущее значение с числом 10.

- Если поставить оператор присваивания в условии, то при проверке условия, значение переменной изменится, после чего это условие выполнится.

Каждому оператору **if** соответствует только один оператор **else**. Совокупность этих операторов — **else if** означает, что если не выполнилось предыдущее условие, то проверить данное. Если ни одно из условий не верно, то выполняется тело оператора **else**.

Если после оператора **if**, **else** или их связки **else if** должна выполняться только одна команда, то фигурные скобки можно не

ставить. Предыдущую программу можно записать следующим образом:

```
#include <iostream>
using namespace std;

int main()
{
    setlocale(0, "");
    double num;

    cout << "Введите произвольное число: ";
    cin >> num;

    if (num < 10) // Если введенное число меньше 10.
        cout << "Это число меньше 10." << endl;
    else if (num == 10)
        cout << "Это число равно 10." << endl;
    else // иначе
        cout << "Это число больше 10." << endl;

    return 0;
}
```

Такой метод записи выглядит более компактно. Если при выполнении условия нам требуется выполнить более одной команды, то фигурные скобки необходимы. Например:

```
#include <iostream>
using namespace std;

int main()
{
    setlocale(0, "");
    double num;
    int k;

    cout << "Введите произвольное число: ";
    cin >> num;

    if (num < 10) { // Если введенное число меньше 10.
        cout << "Это число меньше 10." << endl;
        k = 1;
    } else if (num == 10) {
        cout << "Это число равно 10." << endl;
        k = 2;
    }
}
```

```
} else { // иначе
    cout << "Это число больше 10." << endl;
    k = 3;
}
```

```
cout << "k = " << k << endl;
return 0;
}
```

Данная программа проверяет значение переменной **num**. Если она меньше 10, то присваивает переменной **k** значение единицы. Если переменная **num** равна десяти, то присваивает переменной **k** значение двойки. В противном случае — значение тройки. После выполнения ветвления, значение переменной **k** выводится на экран.

Хорошенько потренируйтесь, попробуйте придумать свой пример с ветвлением.

## Цикл for

Если мы знаем точное количество действий (итераций) цикла, то можем использовать **цикл for**. Синтаксис его выглядит примерно так:

```
for (действие до начала цикла;
    условие продолжения цикла;
    действия в конце каждой итерации цикла) {
    инструкция цикла;
    инструкция цикла 2;
    инструкция цикла N;
}
```

Итерацией цикла называется один проход этого цикла

Существует частный случай этой записи, который мы сегодня и разберем:

```
for (счетчик = значение; счетчик < значение; шаг цикла) {
    тело цикла;
}
```

**Счетчик цикла** — это переменная, в которой хранится количество проходов данного цикла.

Описание синтаксиса

1. Сначала присваивается первоначальное значение счетчику, после чего ставится точка с запятой.



2. Затем задается конечное значение счетчика цикла. После того, как значение счетчика достигнет указанного предела, цикл завершится. Снова ставим точку с запятой.
3. Задаем шаг цикла. **Шаг цикла** — это значение, на которое будет увеличиваться или уменьшаться счетчик цикла при каждом проходе.

Пример кода

Напишем программу, которая будет считать сумму всех чисел от 1 до 1000.

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int i; // счетчик цикла
    int sum = 0; // сумма чисел от 1 до 1000.
    setlocale(0, "");
    for (i = 1; i <= 1000; i++) // задаем начальное значение 1, конечное
    1000 и задаем шаг цикла - 1.
    {
        sum = sum + i;
    }
    cout << "Сумма чисел от 1 до 1000 = " << sum << endl;
    return 0;
}
```

Если мы скомпилируем этот код и запустим программу, то она покажет нам ответ: 500500. Это и есть сумма всех целых чисел от 1 до 1000. Если считать это вручную, понадобится очень много времени и сил. Цикл выполнил всю рутинную работу за нас.

Заметьте, что конечное значение счетчика я задал нестрогим неравенством (  $\leq$  — меньше либо равно), поскольку, если бы я поставил знак меньше, то цикл произвел бы 999 итераций, т.е. на одну меньше, чем требуется. Это довольно важный момент, т.к. здесь новички часто допускают ошибки, особенно при работе с [массивами](#) (о них будет рассказано в следующем уроке). Значение шага цикла я задал равное единице.  $i++$  — это тоже самое, что и  $i = i + 1$ .

В теле цикла, при каждом проходе программа увеличивает значение переменной **sum** на **i**. Еще один очень важный момент — в начале программы я присвоил переменной **sum** значение нуля. Если бы я этого не сделал, программа вылетела бы в [сегфолт](#). При объявлении

переменной без ее инициализации что эта переменная будет хранить «мусор».

Естественно к мусору мы ничего прибавить не можем. Некоторые компиляторы, такие как **gcc**, инициализирует переменную нулем при ее объявлении.

## Цикл while

Когда мы не знаем, сколько итераций должен произвести цикл, нам понадобится цикл **while** или **do...while**. Синтаксис цикла **while** в C++ выглядит следующим образом.

```
while (Условие) {  
    Тело цикла;  
}
```

Данный цикл будет выполняться, пока условие, указанное в круглых скобках является истиной. Решим ту же задачу с помощью цикла **while**. Хотя здесь мы точно знаем, сколько итераций должен выполнить цикл, очень часто бывают ситуации, когда это значение неизвестно.

Ниже приведен исходный код программы, считающей сумму всех целых чисел от 1 до 1000.

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    setlocale(0, "");  
    int i = 0; // инициализируем счетчик цикла.  
    int sum = 0; // инициализируем счетчик суммы.  
    while (i < 1000)  
    {  
        i++;  
        sum += i;  
    }  
    cout << "Сумма чисел от 1 до 1000 = " << sum << endl;  
    return 0;  
}
```

После компиляции программа выдаст результат, аналогичный результату работы предыдущей программы. Но поясним несколько важных моментов. Я задал строгое неравенство в условии цикла и инициализировал счетчик *i* нулем, так как в цикле **while** происходит на

одну итерацию больше, потому он будет выполняться, до тех пор, пока значение счетчика перестает удовлетворять условию, но данная итерация все равно выполнится. Если бы мы поставили нестрогое неравенство, то цикл бы закончился, когда переменная *i* стала бы равна 1001 и выполнилось бы на одну итерацию больше.

Теперь давайте рассмотрим по порядку исходный код нашей программы. Сначала мы инициализируем счетчик цикла и переменную, хранящую сумму чисел.

В данном случае мы обязательно должны присвоить счетчику цикла какое-либо значение, т.к. в предыдущей программе мы это значение присваивали внутри цикла **for**, здесь же, если мы не инициализируем счетчик цикла, то в него попадет «мусор» и компилятор в лучшем случае выдаст нам ошибку, а в худшем, если программа соберется — сегфолт практически неизбежен.

Затем мы описываем условие цикла — **«пока переменная *i* меньше 1000 — выполняй цикл»**. При каждой итерации цикла значение переменной-счетчика *i* увеличивается на единицу внутри цикла.

Когда выполнится 1000 итераций цикла, счетчик станет равным 999 и следующая итерация уже не выполнится, поскольку 1000 не меньше 1000. Выражение ***sum* += *i*** является укороченной записью ***sum* = *sum* + *i***.

После окончания выполнения цикла, выводим сообщение с ответом.

## Цикл **do while**

Цикл **do while** очень похож на цикл **while**. Единственное их различие в том, что при выполнении цикла **do while** один проход цикла будет выполнен независимо от условия. Решение задачи на поиск суммы чисел от 1 до 1000, с применением цикла **do while**.

```
#include <iostream>
using namespace std;
```

```
int main ()
{
    setlocale(0, "");
    int i = 0; // инициализируем счетчик цикла.
    int sum = 0; // инициализируем счетчик суммы.
    do { // выполняем цикл.
        i++;
        sum += i;
    } while (i < 1000); // пока выполняется условие.
```

```
cout << "Сумма чисел от 1 до 1000 = " << sum << endl;
return 0;
}
```

Принципиального отличия нет, но если присвоить переменной `i` значение, большее, чем 1000, то цикл все равно выполнит хотя бы один проход.

Попрактикуйтесь, поэкспериментируйте над собственными примерами задач. Циклы — очень важная вещь, поэтому им стоит уделить побольше внимания.

### Пример инициализации массива

```
string students[10] = {
    "Иванов", "Петров", "Сидоров",
    "Ахмедов", "Ерошкин", "Выхин",
    "Андреев", "Вин Дизель", "Картошкин", "Чубайс"
};
```

### Описание синтаксиса

Массив создается почти так же, как и обычная переменная. Для хранения десяти фамилий нам нужен массив, состоящий из 10 элементов. Количество элементов массива задается при его объявлении и заключается в квадратные скобки.

Чтобы описать элементы массива сразу при его создании, можно использовать фигурные скобки. В фигурных скобках значения элементов массива перечисляются через запятую. В конце закрывающей фигурной скобки ставится точка с запятой.

Попробуем вывести наш массив на экран с помощью оператора **cout**.

```
#include <iostream>
#include <string>
```

```
int main()
{
    std::string students[10] = {
        "Иванов", "Петров", "Сидоров",
        "Ахмедов", "Ерошкин", "Выхин",
        "Андреев", "Вин Дизель", "Картошкин", "Чубайс"
    };
    std::cout << students << std::endl; // Пытаемся вывести весь массив
    // непосредственно
    return 0;
}
```

}

Скомпилируйте этот код и посмотрите, на результат работы программы. Готово? А теперь запустите программу еще раз и сравните с предыдущим результатом. В моей операционной системе вывод был следующим:

- Первый вывод: `0x7fff8b85820`
- Второй вывод: `0x7fff7a335f90`
- Третий вывод: `0x7fff847eb40`

Мы видим, что выводится адрес этого массива в оперативной памяти, а никакие не «Иванов» и «Петров».

Дело в том, что при создании переменной, ей выделяется определенное место в памяти. Если мы объявляем переменную типа `int`, то на машинном уровне она описывается двумя параметрами — ее адресом и размером хранимых данных.

Массивы в памяти хранятся таким же образом. Массив типа `int` из 10 элементов описывается с помощью адреса его первого элемента и количества байт, которое может вместить этот массив. Если для хранения одного целого числа выделяется 4 байта, то для массива из десяти целых чисел будет выделено 40 байт.

Так почему же, при повторном запуске программы, адреса различаются? Это сделано для защиты от атак [переполнения буфера](#). Такая технология называется [рандомизацией адресного пространства](#) и реализована в большинстве популярных ОС.

Попробуем вывести первый элемент массива — фамилию студента Иванова.

```
#include <iostream>
#include <string>
```

```
int main()
{
    std::string students[10] = {
        "Иванов", "Петров", "Сидоров",
        "Ахмедов", "Ерошкин", "Выхин",
        "Андреев", "Вин Дизель", "Картошкин", "Чубайс"
    };
    std::cout << students[0] << std::endl;
    return 0;
}
```

Смотрим, компилируем, запускаем. Убедились, что вывелся именно «Иванов». Заметьте, что нумерация элементов массива в C++ начинается с нуля. Следовательно, фамилия первого студента находится в `students[0]`, а фамилия последнего — в `students[9]`.

В большинстве языков программирования нумерация элементов массива также начинается с нуля.

Попробуем вывести список всех студентов. Но сначала подумаем, а что если бы вместо группы из десяти студентов, была бы кафедра их ста, факультет из тысячи, или даже весь университет? Ну не будем же мы писать десятки тысяч строк с `cout`?

Конечно же нет! Мы возьмем на вооружение циклы, о которых был написан [предыдущий урок](#).

### Вывод элементов массива через цикл

```
#include <iostream>
#include <string>

int main()
{
    std::string students[10] = {
        "Иванов", "Петров", "Сидоров",
        "Ахмедов", "Ерошкин", "Выхин",
        "Андреев", "Вин Дизель", "Картошкин", "Чубайс"
    };
    for (int i = 0; i < 10; i++) {
        std::cout << students[i] << std::endl;
    }

    return 0;
}
```

Если бы нам пришлось выводить массив из нескольких тысяч фамилий, то мы бы просто увеличили конечное значение счетчика цикла — строку `for (...; i < 10; ...)` заменили на `for (...; i < 10000; ...)`.

Заметьте что счетчик нашего цикла начинается с нуля, а заканчивается девяткой. Если вместо оператора строгого неравенства — `i < 10` использовать оператор «меньше, либо равно» — `i <= 10`, то на последней итерации программа обратится к несуществующему элементу массива — `students[10]`. Это может привести к [ошибкам сегментации](#) и аварийному завершению программы. Будьте внимательны — подобные ошибки бывает сложно отловить.

Массив, как и любую переменную можно не заполнять значениями при объявлении.

### Объявление массива без инициализации

```
string students[10];  
// или  
string teachers[5];
```

Элементы такого массива обычно содержат в себе «мусор» из выделенной, но еще не инициализированной, памяти. Некоторые компиляторы, такие как GCC, заполняют все элементы массива нулями при его создании.

При создании статического массива, для указания его размера может использоваться только константа. Размер выделяемой памяти определяется на этапе компиляции и не может изменяться в процессе выполнения.

```
int n;  
cin >> n;  
string students[n]; /* Неверно */
```

Выделение памяти в процессе выполнения возможно при работе с динамическими массивами. Но о них немного позже.

Заполним с клавиатуры пустой массив из 10 элементов.

### Заполнение массива с клавиатуры

```
#include <iostream>  
#include <string>
```

```
using std::cout;  
using std::cin;  
using std::endl;
```

```
int main()  
{  
    int arr[10];  
  
    // Заполняем массив с клавиатуры  
    for (int i = 0; i < 10; i++) {  
        cout << "[" << i + 1 << "]" << ": ";  
        cin >> arr[i];  
    }  
}
```



```
// И выводим заполненный массив.
```

```
cout << "\nВаш массив: ";
```

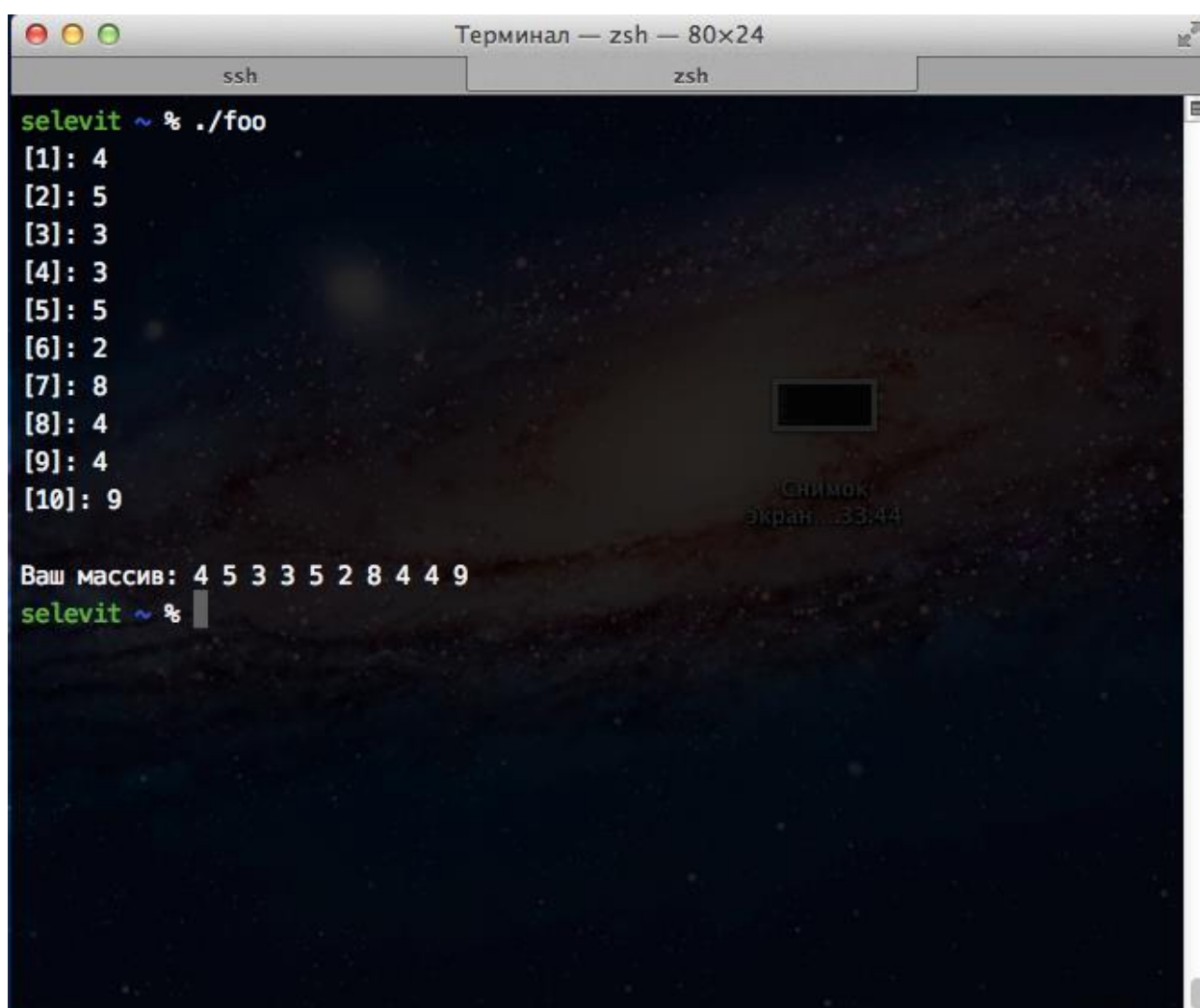
```
for (int i = 0; i < 10; ++i) {  
    cout << arr[i] << " ";  
}
```

```
cout << endl;
```

```
return 0;
```

```
}
```

Скомпилируем эту программу и проверим ее работу.



```
Терминал — zsh — 80x24  
ssh zsh  
selevit ~ % ./foo  
[1]: 4  
[2]: 5  
[3]: 3  
[4]: 3  
[5]: 5  
[6]: 2  
[7]: 8  
[8]: 4  
[9]: 4  
[10]: 9  
  
Ваш массив: 4 5 3 3 5 2 8 4 4 9  
selevit ~ %
```

Если у вас возникают проблемы при компиляции исходников из уроков — внимательно прочитайте ошибку компилятора, попробуйте проанализировать и исправить ее.

Очень часто в программировании необходимо выполнять одни и те же действия. Например, мы хотим выводить пользователю сообщения об



ошибке в разных местах программы, если он ввел неверное значение. без функций это выглядело бы так:

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string valid_pass = "qwerty123";
    string user_pass;
    cout << "Введите пароль: ";
    getline(cin, user_pass);
    if (user_pass == valid_pass) {
        cout << "Доступ разрешен." << endl;
    } else {
        cout << "Неверный пароль!" << endl;
    }
    return 0;
}
```

А вот аналогичный пример с функцией:

```
#include <iostream>
#include <string>

using namespace std;

void check_pass (string password)
{
    string valid_pass = "qwerty123";
    if (password == valid_pass) {
        cout << "Доступ разрешен." << endl;
    } else {
        cout << "Неверный пароль!" << endl;
    }
}

int main()
{
    string user_pass;
    cout << "Введите пароль: ";
    getline (cin, user_pass);
    check_pass (user_pass);
    return 0;
}
```

}

По сути, после компиляции не будет никакой разницы для процессора, как для первого кода, так и для второго. Но ведь такую проверку пароля мы можем делать в нашей программе довольно много раз. И тогда получается копиаста и код становится нечитаемым. Функции — один из самых важных компонентов языка C++.

- Любая функция имеет тип, также, как и любая переменная.
- Функция может возвращать значение, тип которого в большинстве случаев аналогично типу самой функции.
- Если функция не возвращает никакого значения, то она должна иметь тип **void** (такие функции иногда называют процедурами)
- При объявлении функции, после ее типа должно находиться имя функции и две круглые скобки — открывающая и закрывающая, внутри которых могут находиться один или несколько аргументов функции, которых также может не быть вообще.
- после списка аргументов функции ставится открывающая фигурная скобка, после которой находится само тело функции.
- В конце тела функции обязательно ставится закрывающая фигурная скобка.

### Пример построения функции

```
#include <iostream>
using namespace std;

void function_name ()
{
    cout << "Hello, world" << endl;
}

int main()
{
    function_name(); // Вызов функции
    return 0;
}
```

Перед вами тривиальная программа, **Hello, world**, только реализованная с использованием функций.

Если мы хотим вывести «Hello, world» где-то еще, нам просто нужно вызвать соответствующую функцию. В данном случае это делается так: **function\_name()**; . Вызов функции имеет вид имени функции с

последующими круглыми скобками. Эти скобки могут быть пустыми, если функция не имеет аргументов. Если же аргументы в самой функции есть, их необходимо указать в круглых скобках.

Также существует такое понятие, как параметры функции по умолчанию. Такие параметры можно не указывать при вызове функции, т.к. они примут значение по умолчанию, указанно после знака присваивания после данного параметра и списке всех параметров функции.

В предыдущих примерах мы использовали функции типа **void**, которые не возвращают никакого значения. Как многие уже догадались, оператор **return** используется для возвращения вычисляемого функцией значения.

Рассмотрим пример функции, возвращающей значение на примере проверки пароля.

```
#include <iostream>
#include <string>

using namespace std;

string check_pass (string password)
{
    string valid_pass = "qwerty123";
    string error_message;
    if (password == valid_pass) {
        error_message = "Доступ разрешен.";
    } else {
        error_message = "Неверный пароль!";
    }
    return error_message;
}

int main()
{
    string user_pass;
    cout << "Введите пароль: ";
    getline (cin, user_pass);
    string error_msg = check_pass (user_pass);
    cout << error_msg << endl;
    return 0;
}
```

В данном случае функция **check\_pass** имеет тип **string**, следовательно она будет возвращать только значение типа **string**,

иными словами говоря строку. Давайте рассмотрим алгоритм работы этой программы.

Самой первой выполняется функция **main()**, которая должна присутствовать в каждой программе. Теперь мы объявляем переменную **user\_pass** типа `string`, затем выводим пользователю сообщение «Введите пароль», который после ввода попадает в строку `user_pass`. А вот дальше начинает работать наша собственная функция **check\_pass()**.

В качестве аргумента этой функции передается строка, введенная пользователем.

Аргумент функции — это, если сказать простым языком переменные или константы вызывающей функции, которые будет использовать вызываемая функция.

При объявлении функций создается **формальный параметр**, имя которого может отличаться от параметра, передаваемого при вызове этой функции. Но типы формальных параметров и передаваемых функции аргументов в большинстве случаев должны быть аналогичны.

После того, как произошел вызов функции **check\_pass()**, начинает работать данная функция. Если функцию нигде не вызвать, то этот код будет проигнорирован программой. Итак, мы передали в качестве аргумента строку, которую ввел пользователь.

Теперь эта строка в полном распоряжении функции (хочу обратить Ваше внимание на то, что переменные и константы, объявленные в разных функциях независимы друг от друга, они даже могут иметь одинаковые имена. В следующих уроках я расскажу о том, что такое область видимости, локальные и глобальные переменные).

Теперь мы проверяем, правильный ли пароль ввел пользователь или нет. если пользователь ввел правильный пароль, присваиваем переменной **error\_message** соответствующее значение. если нет, то сообщение об ошибке.

После этой проверки мы **возвращаем** переменную **error\_message**. На этом работа нашей функции закончена. А теперь, в функции **main()**, то значение, которое возвратила наша функция мы присваиваем переменной **error\_msg** и выводим это значение (строку) на экран терминала.

Также, можно организовать повторный ввод пароля с помощью **рекурсии** (о ней мы еще поговорим). Если объяснять вкратце, рекурсия — это когда функция вызывает сама себя. Смотрите еще один пример:

```
#include <iostream>
#include <string>
```

```
using namespace std;
```

```
bool password_is_valid (string password)
{
    string valid_pass = "qwerty123";
    if (valid_pass == password)
        return true;
    else
        return false;
}
```

```
void get_pass ()
{
    string user_pass;
    cout << "Введите пароль: ";
    getline(cin, user_pass);
    if (!password_is_valid(user_pass)) {
        cout << "Неверный пароль!" << endl;
        get_pass (); // Здесь делаем рекурсию
    } else {
        cout << "Доступ разрешен." << endl;
    }
}
```

```
int main()
{
    get_pass ();
    return 0;
}
```

Функции очень сильно облегчают работу программисту и намного повышают читаемость и понятность кода, в том числе и для самого разработчика (не удивляйтесь этому, т. к. если вы откроете код, написанный вами полгода назад, не сразу поймете соль, поверьте на слово).

Не расстраивайтесь, если не сразу поймете все аспекты функций в C++, т. к. это довольно сложная тема и мы еще будем разбирать примеры с функциями в следующих уроках.

Совет: не бойтесь экспериментировать, это очень хорошая практика, а после прочтения данной статьи порешайте элементарные задачи, но с использованием функций. Это будет очень полезно для вас.