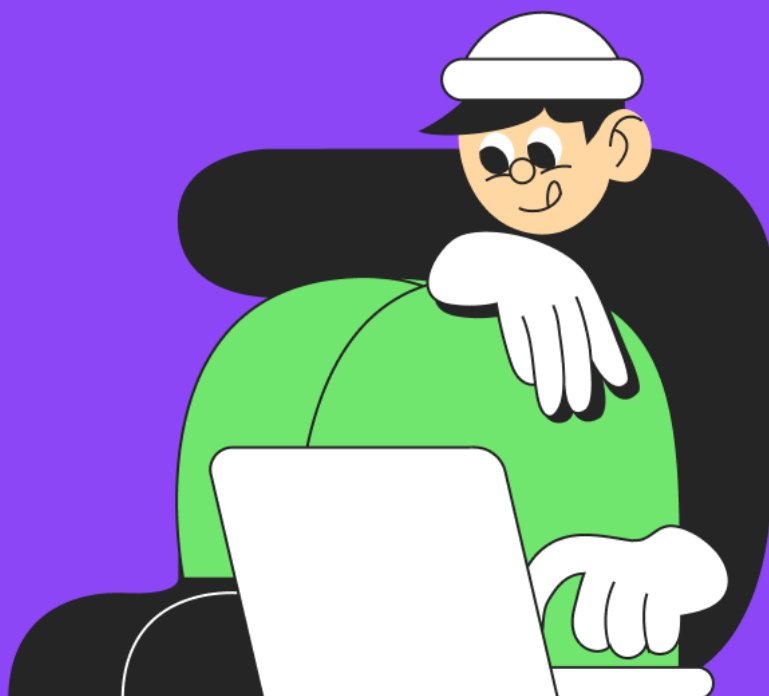
GeekBrains

Базовые алгоритмы и массивы, как структура данных



Оглавление

[00:01:35]

Приветствие

Я рад вас приветствовать на второй лекции курса «Введение в программирование» в программе Разработчик. С вами снова я Ильнар. Если вдруг кто забыл, то вот снова слайд с информацией обо мне. Помимо, работы в GeekBrains, я ещё профессиональный преподаватель. Я преподаю в казанском университете. Как я иногда люблю шутить на эту тему, если вам в детстве хотелось оказаться в казанском университете, почувствовать себя его студентом, то вы стали чуть ближе к этой своей мечте.

[00:02:07]

Ранее на курсе

Ранее мы рассматривали несколько основных вопросов.

- Аналогии с обычными языками.
- Различные способы поиска максимального элемента, например, у нас была морковка, с помощью которой мы с вами могли найти максимальный элемент из 5.
- Разные формы записи алгоритма, например, текстом или блок-схемой. Смотрели, как писать реальный код на Python и Java.

Далее у нас была задачка о том, как выбираться из лабиринта, если вы в него попали. У вас на дом оставался анализ блок-схемы, которой мы посмотрели. Там действительно была ошибка, и эта блок-схема циклится почти в самом начале нашего лабиринта. Если кто-то будет выходить из лабиринта по блок-схеме, то он никогда из него не выйдет. Думаю, на семинарах, вы всё разобрали, и, соответственно, дома нашли этот момент. Очень надеюсь, что у вас получилось составить блок-схему, с помощью которой можно выйти из лабиринта.

Также была довольно сложная задача с 2 друзьями и собакой, которая бегала между ними. Если вдруг вы после нашей лекции ещё раз не пересматривали и не перерешали её, настоятельно рекомендую это сделать. А лучше остановитесь на моменте, когда я рассказал вам условия этой задачи и попробуйте, не заглядывая дальше, решить задачу самостоятельно. Возьмите ручку, возьмите листочек и прорешайте, это будет максимально полезно.

[00:03:44]

Подготовка к уроку

Сегодня будет ряд заданий, которые необходимо будет выполнять. Поэтому позаботьтесь о том, чтобы у вас под рукой были листочек и ручка.

Также для тех, кто прошёл курс, умение учиться с Игорем. Есть предложение. Если вы в Notion продолжаете создавать своё пространство знаний и делаете в нём конспекты, будет здорово, если вы поделитесь этими конспектами, продемонстрируете, что у вас получается. Это будет полезно как для людей, учащихся вместе с вами, так и вам, скорее всего, будет интересно посмотреть на то, какие конспекты пишут другие люди. Помимо этого, мне будет очень интересно посмотреть, какие мысли из того, что я рассказываю, вы у себя фиксируете, и как это делаете. Поэтому в комментариях под этим уроком будет здорово, если вы оставите ссылки

на ваши конспекты. Если вдруг вы делаете это не Notion, а пишете на листе, можете сфотографировать и залить файлы, куда вам удобно. И в комментариях оставить на них ссылку. Я думаю, что мы все сможем посмотреть.

[00:04:48]

На этом уроке

Сегодня мы перейдём к структурам данных. Звучит всё очень масштабно и сильно. Но на самом деле ничего сложного нет, мы с ними познакомимся.

Помимо этого, мы посмотрим на простые алгоритмы с этими структурами, в первую очередь массив. И какие алгоритмы есть в этом массиве, попробуем перерешать заново задачи, которые были у нас на первом занятии и посмотрим, как они будут решаться, если использовать массивы.

Немного заглянем на кухню программиста. Посмотрим, что происходит, когда мы с вами запускаем программу. Мы пока ещё программы не запускали, и об этом чуть дальше поговорим, но будем уже немного смотреть на кухню. Что же там происходит, когда всё запускается.

[00:05:36]

Структура данных

Составление программы и написание кода — две разные профессии

Скорее всего, у многих из вас мог возникнуть вопрос. Для чего нам эти блок-схемы? Мы же пришли учиться программировать или тестировать программы написанные, или заниматься аналитикой.

Но поговорим немного про историю. Изначально составление алгоритмов и программ, и написание кода — были двумя разными профессиями.

Были люди, продумывающие алгоритм, что нужно будет сделать компьютеру. И другие люди, составляющие код, делали так, чтобы на конкретном компьютере он мог выполняться. Чтобы можно было занести код в программу, и там происходили вычисления. Раньше были огромные ЭВМ (электронные вычислительные машины), в них нужно было переносить код. Составление программы и занесение кода в машину, чтобы она выполнила его — две разные профессии и задачи.

Если сравнивать с тем, что у нас есть на бытовом уровне, можно вспомнить про стенографистов (людей, записывающих текст, раньше на печатных машинах, сейчас на компьютерах) и людей, произносящих этот текст. Так вот, произнесение текста, составление текста, когда мы эту речь начинаем готовить, продумывать — это составление программы, а вот запись текста — это стенограмма (запись программы на конкретном языке). Да, существуют определённые условия, есть свой синтаксис, надо правильно оформить текст, одни речевые обороты

использовать, другие исправлять — это отдельная профессия, ей нужно учиться, нужно уметь это делать. Но при этом это две разные профессии.

То же самое происходит у нас в программировании. У нас есть составление программ, когда вы продумываете алгоритм. Когда думаете, а как же нужно? Какие действия необходимо совершить? В каком порядке? Какие нам нужны переменные? Что мы с ними будем делать? И другая задача — это написание этой программы. То есть, когда у вас уже готова программа, алгоритм составлен, и необходимо его записать. И вот здесь вы выбираете тот язык, который лучше подходит для конкретной задачи. На основе синтаксиса и принципов языка записываете ваше решение на нужном языке. Потом оно выполняется программой. В большинстве случаев, когда учатся программировать не делят эти части. Моя же задача на этом курсе всё-таки разделить эти две профессии.

В первую очередь мы с вами будем заниматься составлением алгоритмов и программ, которые вы позже запишите, их анализом. Дальше, когда мы перейдём к следующему курсу, в нём начнётся программирование на конкретном языке. Мы будем записывать составленные алгоритмы согласно синтаксису выбранного языка. После вы будете запускать программы, смотреть, что с ними происходит. Но сейчас нам необходимо разделить две эти профессии.

[00:08:48]

Массивы

Теперь перейдём к структурам данных. Иногда удобнее работать с какими-то данными, если они подчиняются определённым взаимосвязям и у них есть структура. Массив — это самая первая структура, с которой чаще всего знакомятся. На самом деле, вы с ним уже практически знакомы.

Пример с гирями

Когда мы искали максимальный элемент из 5. Мы уже говорили, что вот 1 гиря, 2 гиря, 3 гиря, 4 гиря, 5 гиря. То есть у гирь уже были номера. Между ними уже была взаимосвязь, был какой-то порядок. Так вот, когда у нас есть взаимосвязь по индексу (номеру у этих объекта), тогда мы можем сказать, что у нас практически есть массив. Есть некоторая структура. У нас это гирьки, стоящие на столе. Они взаимосвязаны, имеют индекс, по которому их можно найти.

Массив — это структура данных, в которой можно по индексу найти определённый объект.

Теперь посмотрим, а для чего же массив нужен? У нас было 5 гирек и мы прекрасно обходились без массивов. Представим, что будет не 5 гирек, а 100 гирек. Если у нас 26 букв английского алфавита ещё есть, и первые 26 гирек, мы можем переименовать так же, как делали a, b, c, d, e, то с остальными непонятно, что делать? Нам нужно будет придумать, какие-то составные имена. В принципе, блок с инициализацией после заведения этих переменных станет очень большим. Это будет крайне неудобно менять. Далее, когда нам нужно будет перебирать эти гирьки, вспоминать, какую гирьку мы уже посмотрели, а с какой надо работать дальше? Именно поэтому гораздо удобнее, если объекты однотипные, заводить их в какую-то структуру.

[00:10:48]

Пример с гардеробом

Посмотрим, что такое массивы? Как они работают? Чтобы было понятнее, мы начнём с простых бытовых примеров.

Рассмотрим гардероб в театре. Каким принципам он подчиняется? У нас есть вешалки, и мы заранее знаем их количество. Допустим, у нас гардероб на 200 посадочных мест, соответственно, у нас есть 200 крючков, 200 номерков, на которые можно повесить элементы одежды. То есть заранее известно количество. Больше оно так просто не станет. Естественно, можно пригласить слесаря прибить новые крючочки, но в целом их количество заранее известно, и оно не меняется.

Помимо этого, на каждый крючок мы вешаем только элементы одежды, то есть на каждом крючке будут находиться однотипные вещи. Мы не сможем в гардероб припарковать машину, например, или положить туда штангу на какой-то крючок. В любом случае на каждом крючке у нас находится одинаковые, очень похожие друг на друга объекты.

На один крючок можно повесить только один элемент, то есть в одну ячейку массива можно положить только один элемент. И естественно, если в театре бывают исключения и вы можете попросить гардеробщицу на один номерок повесить два элемента одежды, то в программировании так сделать нельзя. У нас всё очень строго, в одну ячейку только один элемент. Вот здесь есть пару примеров, которые поясняют всё-таки, как у нас работает гардероб и почему этот похоже на массив.

[00:12:26]

Пример подземной парковки

Посмотрим ещё два похожих примера. У нас есть парковка подземная, она также очень похожа на массив. Почему? Потому что на парковке однотипные элементы, там только транспортные средства, причём примерно одинаковые. В большинстве случаев автомобили легковые с небольшими вариациями, мы вряд ли на подземный паркинг будем огромную фуру с прицепом загонять.

Помимо этого, все парковочные места у нас пронумерованы. Соответственно, вы можете легко найти свой автомобиль, если знаете, где его оставили. То есть некоторых порядок, есть некоторая структуры между этими элементами, взаимосвязь.

И последнее, у нас ограниченное количество машин может поместиться на эту парковку, то есть размер нашего массива, размер нашей парковки заранее известен, и не может поменяться. Если мы захотим, чтобы больше машин помещалось, то нам либо придётся сломать структуру и оставлять машины, где-то в проходах, где не положено их оставлять, либо делать пристрой и увеличивать. Получается, мы не сможем увеличить размер нашей парковки.

С массивами то же самое. В массивах заранее известно количество элементов, которые в них можно поместить. Заранее известен размер каждого элемента, тип каждого элемента, то есть мы знаем, что вот, например, здесь у нас вмещаются только машины. В гардеробе у нас помещается одежда. Тип элемента, хранящийся в массиве, у нас заранее известен. Помимо этого, мы можем по индексу получить нужный нам объект. В гардеробе можем получить по

номерку нашу одежду, на парковке по номеру места найти, где же находится наше транспортное средство.

[00:14:14]

Пример бабушкиного балкона

Рядом у нас есть пример бабушкиного балкона. Так вот балкон является примером, другой структуры данных, а может быть там и нет структуры. Это не может быть массивом. Почему?

На балконе мы можем хранить и левую сломанную лыжу, и старый неработающий холодильник, и телевизор, и одну гантелью, и многое другое. То есть объекты, хранящиеся на балконе, не одного типа. Там может храниться всё что угодно. По сути, это некоторая свалка объектов.

Далее у нас нет никакой индексации. Мы не можем по номеру или позиции понять, где что находится. Не можем с третьего места на балконе что-то найти. У нас всё равно какая-то общая куча.

Помимо этого, на балкон всегда может что-то ещё поместиться, даже если он кажется полным, сверху можно что-то положить. Так его размер будет увеличиваться. То есть у нас нет заранее фиксированного размера хранилища.

Посмотрите на эти объекты на парковку и балкон. Так вот, парковка относится к массиву, она гораздо ближе к массиву, по принципу своей работы, а балкон — это нечто максимально далёкое от массивов.

Будет здорово, если вы сможете в комментариях написать пару примеров, что из обычной жизни похоже на массив, а что точно нет, так как не подчиняется каким-то принципам. Недостаточно просто написать, что гардероб похож на массив. Лучше написать почему. Потому что есть один принцип, он соблюдается, и принцип другой, он тоже соблюдается. То же самое с контрпримером (примером не массива). Например, балкон. У него не соблюдается принцип индексации, мы не можем найти объекты по номеру, поэтому это не массив. Или в нём хранятся совершенно разные вещи, поэтому это не массив. Нужно привести пример и контрпример массива и показать, что принципы, которые мы сейчас обсудили, выполняются или не выполняются.

[00:16:27]

Можно выделить три параметра массива

Мы выведем на экран основные параметры, которые у нас, как следствие, будут получаться.

1. *Нам будет известно начало массива.*

Если у нас есть произвольный объект, где-то расположенный, и у него есть первый элемент, который где-то лежит. Мы можем этот первый элемент найти.

2. *Далее мы можем найти размер каждого элемента.*

Если нам известен тип. Если мы знаем, где объекты хранятся. То мы можем найти размер каждого элемента массива. В гардеробе у нас через каждые 30–40 см висят новые крючочки, и мы знаем, что от одного объекта для второго объекта вот такое расстояние. На парковочных

местах мы примерно знаем размер автомобиля, а чаще всего на парковках и очерченные размеры парковочных мест. Поэтому размер каждого элемента массива также нам известен.

3. *Мы знаем количество элементов в массиве это то, о чём мы с вами ранее говорили.*

Например, в гардеробе может быть 200 вешалок, а на парковочной площадке помещается 100 автомобилей.

Сейчас мы сделаем маленькую паузу, буквально на полторы минуты. В течение этого времени напишите, пару примеров объектов являющихся и не являющихся массивами. И главное — поясните почему?

[00:19:15]

Полезно искать аналогии в реальной жизни

Я надеюсь, что все смогли справиться. Если вдруг вам не хватило времени, то можете закончить попозже. Очень полезно искать аналогии между тем, что мы с вами делаем и реальной жизнью. На самом деле мы сейчас рассматриваем, не какие-то сугубо абстрактные вещи. Если в математике, которой я занимался во время написание диссертации, связь с реальностью найти было категорически невозможно. То здесь много прикладных примеров. Структуры данных и алгоритмы, рассматриваемые здесь, прочно связаны с реальным миром и оттуда в большинстве случаев пришли. Поэтому, если пока ещё не нашли аналогии, осмотритесь. Вокруг нас очень много массивов и не массивов. Нужно просто проверять критерии, о которых мы с вами говорили.

[00:20:07]

Устройство массивов внутри компьютера

Пример ленты с котиком

Теперь перейдём к устройству памяти компьютера. Если немного упростить то, что происходит в вашем компьютере, то память очень похожа на массив. Её можно представить как очень длинную ленту, в каждой ячейке которой что-то написано. Будем предполагать, что в каждой ячейке памяти хранятся какие-то символы. Сейчас на экране небольшой кусочек очень длинной ленты и лишь часть мы видим на слайде, потому что память в компьютере очень большая. Есть какие-то символы в разных ячейках.

В данный момент часть памяти можно представить как подмассив, что-то находящиеся внутри. В качестве некоторого массива можно посмотреть на эти три ячейки.

Есть три параметра, которые нам нужно знать.

1. Где массив начинается. Если нужно получить слово cat (нашего котика), то мы знаем, что первая буква находится в ячейке 127.
2. Размер каждой ячейки массива. Здесь ничего сложного, у нас одна ячейка — это один элемент. Получается, в 127 ячейке лежит буква «c», в 128 буква «a», в 129 буква «t».
3. Количество элементов, для этого массив должен быть закончен.

Из всей большой ленты я возьму эти три ячейки и буду рассматривать их как массив. Все три параметра, имеющиеся у нас, соблюдены. Мы знаем, откуда стартует массив, его размер и

количество ячеек. Таким образом, набор из трёх ячеек можно представить как массив из трёх элементов. Элементами, являются буквы. Каждая буква в нашем случае занимает одну ячейку памяти.

Итак, чтобы вытащить нашего котика из памяти, нужно пронумеровать необходимые ячейки. Нам нужно сказать, что из одной ячейки мы возьмём первую букву, из другой вторую и так далее. С котиком всё понятно, мы видим на экране 127, 128, 129 ячейку. Но мы рассмотрели частный случай. Что будет происходить, если мы не видим перед собой на экране весь массив, а только знаем эти параметры? Мы знаем, где он начинается. Знаем размер каждой ячейки и количество элементов. Как можно получить все эти ячейки?

[00:23:04]

Формула поиска n-ого элемента в массиве

Для этого есть формула:

a_n = start + (n-1) * cell_size

Разберёмся в написанном.

В качестве a_n будем рассматривать любую ячейку массива. Например, если нам нужно найти первый элемент массива, тогда вместо n поставим 1. Вот a_1 это первый элемент нашего массива.
Далее start — это адрес первой ячейки (места, откуда начинается массив). Если он нам известен, а он должен быть известен, потому что это один из обязательных параметров, то подставляем его.
Дальше почему-то n-1, с этим сейчас разберёмся.
Теперь умножаем, получившееся здесь, количество на cell_size (с английского cell — ячейка, size — размер).

Как формула работает на нашем котике

Есть cat, мы знаем, что у нас:

- массив начинается в ячейке 127, поэтому start = 127;
- каждый элемент занимает ровно одну ячейку, поэтому cell_size = 1;
- всего три таких элемента.

Теперь найдём, например, адрес первого элемента. Ищем a_1.

- start = 127, то есть ячейка под номером 127.
- N-1, мы ищем первый элемент, поэтому 1 - 1 = 0.
- Cell_size = 1.
- В результате — $a_1 = 127 + 0 \times 1$, то есть $a_1 = 127$.

Таким образом, мы понимаем, что первый элемент нашего массива буковка «c» находится в ячейке 127. Теперь вы, наверное, догадались, почему мы вычитаем 1, если бы не вычитали, то эта формула работала бы неправильно. При такой нумерации элементов нам приходится вычитать 1.

Попробуем по аналогии найти, например, третий элемент букву «t». Что у нас получается?

- A_3 = 127 + (3–1).
- Умножаем на cell_size. Cell_size = 1.
- Получаем 127+2×1=129. Третий элемент буква «t» находится в ячейке с номером 129.

Таким образом, если вам заранее известны все параметры массива, то вот по такой простой формуле вы можете найти любой элемент этого массива.

Пример с парковкой

Если известно, на каком месте, например, на 10 месте находится ваша машина. Вы знаете, где находится первая машина, то есть откуда необходимо стартовать. Вам необходимо пройти 9 машин, зная, сколько места занимает каждая из них, и 10 машина — уже ваша. Таким образом, в любом массиве это работает.

Пример с гардеробом

Зная, что ваша куртка висит в определённом месте, а вы стоите около первой куртки. Дальше зная свой номерок, вам необходимо просто пройти нужное количество вешалок, и там будет ваша куртка.

[00:26:52]

Разберём задачу с прошлого лекции

Теперь познакомившись с массивами, посмотрим, как можно было решить задачу с прошлого занятия, используя структуру данных.

У меня есть стаканчики, для удобства их расставлю. У нас было 1, 8, 3, 2, 6. Допустим, вот так они у нас стояли. Мы искали максимальный. Но нам приходилось именовать каждый из них. Был стаканчик a, стаканчик b, c, d, e. Но как я уже говорил, таких стаканчиков может быть очень много, например, 100 стаканчиков и имён всем не хватит.

Мы можем рассматривать то, что есть сейчас как массив. Имеется 5 элементов. Мы знаем, что каждая гирька имеет размер. Есть 1 гирька, рядом с ней стоят 2, 3, 4, 5. Помимо этого, мы по индексу легко можем их получить. Мне нужна третья гирька, я отсчитываю номер 3. Она ещё совпадает с весом гирьки.

Прежде чем перейдём дальше, мы должны обсудить один очень важный момент. Когда считаем объекты сами, мы считаем 1, 2, 3, 4, 5, но программисты любят всё оптимизировать. И эта часть в нашей формуле им не нравится. Нам приходится каждый раз, когда мы ищем какой-то элемент делать дополнительные вычисления, а это нагрузка на наш процессор. Что придумали программисты? Программисты решили, что нумеровать элементы массива мы будем не с 1, как привыкли считать в обычной жизни, а с 0. Эта гирька будет не первой, а нулевой. А эта гирька будет не второй, а с индексом 1. Чтобы не путаться, мы будем говорить, что эта гирька будет с индексом 0, эта с индексом 1 и т. д.

Что произойдёт? Если вместе с n, вместо 1 (первого элемента), поставить индекс 0, тогда можно не вычитать 1. Я начинаю от старта, для нулевого индекса здесь автоматически станет 0. То есть вместо n-1, я просто оставлю n, где n теперь индекс элемента.

[00:29:36]

Проверим на примере с котиком

Если не первый элемент, а элемент с индексом 0, тогда у нас что получится?

127 + 0 (индекс элемента) * 1 (размер ячейки) = 127

Мы получим 127, при этом нам не нужно вычитать 1.

При таком подходе нам нет необходимости выполнять дополнительные действия, и программа работает быстрее. Учитывая, что компьютеры стали быстро работать только последние 10–20 лет. До этого они были очень медленные, а каждая дополнительная операция стоила очень много, и по времени, и по затратам энергии. То такой способ стал очень выгодным решением.

Если мы ищем элемент с индексом 1. То к 127 прибавляем индекс этого элемента, равный 1 и на cell_size умножаем, получая 128.

127+1 =128

Если нам нужен 3 элемент. У 3 как мы обычно считаем, индекс равен 2. Мы к 127, прибавляем индекс этого элемента, равный 2.

127+2=129

Таким образом, мы точно так же можем находить все элементы, но при этом у нас на одно действие меньше совершается. Именно поэтому индексация в массивах идёт с 0. Поэтому у нас гирьки индексируются 0, 1, 2, 3 и 4.

[00:30:58]

Решаем задачу с помощью массива

Нужно понять, как к индексу можно обращаться. Представим блок-схему. Раньше были переменные a, b, c, d и т. д. Потом мы водили переменные, состоявшие из нескольких слов (firstFriendSpeed, secondFriendSpeed). Теперь у нас появляются новый объект, назовём его — numbers (от английского — числа). Считается хорошим тоном, когда вы именуете переменные заложенным в них смыслом. В нашей переменной numbers будет храниться массив, то есть набор некоторых элементов с определённым порядком. Элементы нам хорошо известно, это число 1, 8, 3, 2, 6. Масса гирек или цифры на стаканчиках, с которыми мы работали.

Вводим переменную size, говорящую о размере массива. Теперь нам нужен индекс. Мы будем стартовать с самого начала, поэтому индекс возьмём равным 0. Потому что у самого первого элемента нашего массива индекс равен 0. Также мы введём наш максимум это первый элемент. Если раньше у нас было написано, что максимум равен «a», то сейчас первую гирьку зовут не «a», её зовут numbers с индексом 0. Именно так можно получить элемент нашего массива.

Есть массив numbers, в квадратных скобках может быть указан индекс элемента, который мы хотим из него достать. Поэтому max = numbers [0], значит то же, что мы до этого писали, max=a или max=1, как первая гирька.

[00:32:57]

Конструкция цикла

Далее, уже знакомая нам конструкция — это цикл. То есть мы отсюда, в зависимости от ответа на вопрос уходим либо по веточке «yes», либо по веточке «no». Но если мы возвращаемся, то будем снова отвечать на этот вопрос. И мы будем делать это до тех пор, пока ответ на вопрос будет «да». Как только ответ станет «нет», мы выскочим.

Дальше, мы здесь с вами указываем размер нашего массива. Обратите внимание, мы могли не вводить переменную size. И поставить просто число 5. Что тогда бы случилось?

Если мы поменяли массив и в нём количество элементов стало другим вместо 5, например, взяли 10, то нам пришлось бы менять число не только в блоке инициализации, но и внутри нашей программы. А как вы помните, мы хотим сделать так, чтобы программа была более или менее универсальной. В самом начале мы для этого водили переменные. Мы именовали гирьки так, чтобы если взять другие гирьки, то менять числа нужно было только в блоках инициализации, которые этим гирькам соответствуют, хранящихся в этих переменных.

Сейчас мы сделали массив, у него есть параметр размера. Здесь мы его зафиксировали. Size=5. Если мы здесь его не зафиксировали в блоке инициализации, а вот сюда поставили число. То когда мы поменяем размер массива, нужно будет обязательно вспомнить и переписать это внутри всей программы. Поэтому крайне нежелательно внутри программы указывать конкретные числа, которые могут меняться. Вот +1 — это нормально, потому что у вас всегда при любом массиве и любых начальных данных, индекс будет меняться один за одним. Он будет просто расти. А если мы говорим, о каких-то параметрах, каких-то вводных данных в вашей программе, то их всегда необходимо определять в каком-то блоке инициализации, в самом начале работы вашей программы. Чтобы внутри не было никаких чисел.

[00:35:09]

Магические числа в программировании

В программировании есть для этих чисел — их называют магическими числами. Так как, когда другой человек смотрит на вашу программу, он не понимает, что это за числа. Это какая-то магия. Почему здесь стояла бы 5 вместо size. И человеку, смотрящему на программу, нужно догадаться, что 5 здесь, наверное, показывает какое количество элементов находится в массиве. Поэтому старайтесь в блок-схемах или программах, которые вы сейчас и в дальнейшем составите, не оставлять магических чисел. Всё должно быть понятно читателю.

Так вот, цикл будет работать до тех пор, пока index < size, мы будем выполнять какие-то действия. Но когда index станет равен или больше размера нашего массива, цикл закончится. Потому что, мы точно прошли весь массив и обработали все элементы.

Что же делать, если мы пока ещё смотрим на элементы массива? Если index совпадают. Мы точно так же отвечаем на следующий вопрос. Элемент, который мы сейчас с вами рассматриваем больше максимального или нет? В этом случае, что у нас будет? Index, пока ещё равен 0. Делаем переменную numbers 0 (это элемент массива с индексом 0, то есть 1) и сравниваем её с элементом хранящимся в переменной max (максимальное значение). В max хранится этот же самый элемент 1. На вопрос 1<1 ответ отрицательный, потому что стоит строго больше. Они равны между собой, одна 1 больше другой быть не может. Поэтому ответ отрицательный. Поэтому мы идём по веточке «No», и оказываемся в этой ячейке. С ней мы

познакомились, когда работали над задачкой с друзьями и собакой. Мы count увеличивали каждый раз, когда собака добегала до одного из друзей. Сейчас происходит то же самое. Только здесь мы будем увеличивать index. Каждый раз, когда мы заходим в эту ячейку, index увеличивается на 1. Сейчас index равен 0, так и запишем:

index = 0

Подставляем в правой стороне вместо index число и прибавляем 1. Получаем:

0 + 1 = 1

Когда мы вычислили правую сторону, подставляем её в переменную index. И после того как выходим из этой ячейки, у нас в переменной index будет храниться 1.

Возвращаемся в условия цикла, которые необходимо проверить. У нас получается 1<size (size=5). 1<5? Ответ положительный, поэтому мы вновь идём в цикл.

Теперь numbers [index] стал 1, поэтому numbers с index 1. Это следующее число. Именно поэтому меняем index, чтобы можно было переходить от одного элемента к другому. Мы последовательно сможем перебрать все элементы.

Сравниваем 8 и то, что хранится в переменной max. В переменной max до сих пор хранится первый элемент 1, потому что мы его не меняли. Получается numbers с index 1 это 8. Здесь у нас хранится 1. 8>1. Соответственно, идём по веточке «Yes». Что мы делаем? Мы в переменную max записываем следующий элемент numbers с индексом 1, 8 записалась в наш максимум.

Идём дальше по стрелочке, попадаем на увеличение индекса, индекс стал равен 2. Возвращаемся, отвечая на вопрос 2<5? Да. Продолжаем цикл.

Что случится в самом конце? Когда у нас index станет равным 5, то есть, после обработки число с index 6. Мы попадём в ячейку, и станет 4+1=5. В index запишется число 5. Мы вернёмся в цикл и на вопрос 5<5? Мы ответим отрицательно, потому что они опять совпадают, а здесь строгое неравенство. Далее пойдём в веточку «No», это будет завершением нашего алгоритма (блок-схемы). Узнаем, что максимальное число у нас найдено, и оно находится в переменной max.

Теперь мы можем то же самое проделать со стаканчиками. Всё полностью идентично выполняемому раньше. Просто сейчас стаканчики названы не a, b, c, d, e. A numbers 0, numbers 1, numbers 2, numbers 3, numbers 4.

Если мы аккуратно пройдём по этой блок-схеме, то сможем провести алгоритм и найти максимальный элемент. Опять окажется 8.

[00:40:36]

Повторим изученное сегодня

Итак. Надеюсь, вы запомнили главное отсюда. Как у нас задаются массивы. Как с ними работать. Как обратиться к элементу. Что когда мы фиксируем размер массива, в какой-то переменной, то не нужно добавлять магические числа. Числа, непонятные никому, кроме вас, про которые вы забудете уже через несколько недель или месяц. Ваша программа становится

гораздо более универсальной. Чтобы она работала для любых вводных данных, вам достаточно поменять всё происходящие в блоке инициализации. Остальное у вас будет работать без дополнительных изменений.

[00:41:20]

Упражнение для закрепления материала

Прошу вас сейчас сделать следующим маленькое упражнение. Попробуйте поменять числа, изменить вводные параметры. Например, если вы поменяете размер массива на 3, 4 или 7 элементов, то вам нужно будет поменять значение в переменной size. Потому что пока она не связана жёстко с массивом numbers, нам приходится менять её вручную. И попробуйте провести этот алгоритм аккуратненько для вашего случая.

В разных языках программирования есть разные способы связи этих параметров. На самом деле у массива есть параметр, который можно получить, но в Python это делается одним способом. Стоит отметить, что в Python для строгости нет массивов, есть списки. Массивы там есть, но они подключаются только к отдельным библиотекам. Но пока мы на это внимание не обращаем. Пока мы предполагаем, что массивы есть во всех, интересующих нас, языках или что-то очень похожее на массив. Соответственно, в разных языках таких как Python, Java, C++ или JavaScript, есть возможность найти размер массива (структуры данных). И не обязательно заводить его руками. Можно просто найти этот параметр или вызвать, позволяющие его найти, функции. И когда вы это сделаете для конкретного языка программирования, вам не нужно будет руками заводить размер вашего массива. И станет ещё лучше, потому что на одно магическое число в программе станет меньше. Пока мы работаем с блок-схемами. И чтобы ни один язык программирования не обижать, какие-то конкретные способы сейчас рассматривать не будем. Но позже к ним ещё перейдём.

Повторим задание. Введите новый массив, рассмотрите желательно с другим размером. Попробуйте, аккуратно пройти по этой блок-схеме. Я думаю, что вы с этим алгоритмом уже неплохо знакомы и вам понадобится буквально, пару минут просто чтобы пройтись по блокам, немного привыкнуть к обозначениям с массивами и индексами. Ещё раз запомнить, что индексация элементов начинается с 0. Именно поэтому у нас index = 0. Попробуйте сделать это упражнение, буквально 2 минутки вам на это.

[00:45:54]

Рассмотрим псевдокод с нашим алгоритмом

Я надеюсь, вы справились с заданием. Оно практически повторяет всё, что мы делали. Думаю, что решение далось вам очень легко.

Главное — ещё раз обратите внимание:

- как задавали массив;
- как происходит индексация;
- как обращаемся к элементам.

Теперь нет необходимости придумывать название каждого элемента, если они однотипные. Нам заранее известно их количество. Мы можем все их запаковать внутрь одной структуры данных (массива) и очень удобно с ними работать.

Но теперь посмотрим, как можно записать это в виде конкретной программы. Опять же, мы стартанём не с конкретного языка программирования, а с псевдокода.

Что такое псевдокод?

Псевдокод — это текст, максимально похожий на языки программирования, но без жёстких синтаксических требований. Если помните, что у нас, например, в Java нужно было ставить точки запятой. В Python не нужно было брать скобки. И вот для каждого языка, свои требования, в псевдокоде таких жёстких требований нет. Он похож на эти языки и переписать программу с псевдокода на конкретный язык, необходимый нам, не составляет никакого труда. Но псевдокод пишется для человека. Программа в большинстве случаев его не поймёт, только если посмотреть на язык, у которого синтаксис совпадает с тем, что мы здесь написали. Но вот для человека всё становится понятно, потому что синтаксические требования не обязательные для нас, мы просто опускаем.

Разберём запись в псевдокоде

Первые две строчки — это блок инициализации, которые был на блок-схеме. Есть массив numbers, заданный нами. В разных языках он задаётся по-разному, но пока будет так. Далее, ввели переменную size (размер нашего массива). Начинаем алгоритм с того, что пойдём к элементу с индексом ноль (самого первого для нас элемента) и возьмём максимальный элемент самым первым элементом массива, то есть элемента с индексом 0. Теперь начинаем цикл.

Смотрите, когда я читал блок-схему, то несколько раз говорил: «До тех пор, пока выполняется условие». И стоит уточнить, что для англоговорящих людей программирование на старте даётся проще, чем для людей, не очень хорошо знающих английский язык или незнающих его вообще. Потому что, если посмотреть на написанное, здесь практически связанный английский текст. У нас написано:

- while (index < size) do

Что получается если это перевести?

- До тех пор, пока индекс меньше размера, делай.

То, о чём мы говорили. Пока индекс меньше, чем размер массива, нам нужно что-то делать. И именно это здесь записано. В очень похожих терминах этот цикл создаётся почти в любых языках программирования. У нас есть:

1. while do циклы;
2. до тех пор;
3. после идёт какое-то условие, которое нам необходимо записать;
4. какой-то призыв к действию, а где-то он может опускаться просто, чтобы язык стал более лаконичны.

Но изначально у нас циклы называются, чаще всего while do.

Что мы с вами делаем дальше? По нашей блок схеме нам нужно было ответить на вопрос, рассматриваемый сейчас элемент массива больше максимального или нет? Опять же, некоторые знания английского языка позволяет нам прочитать написанное.

№ строки	Псевдокод	Перевод
5	while (index < size) do	До тех пор, пока индекс меньше размера, делай.
6	If (numbers[index] > max) then	Если это условие выполняется тогда,
7	max = numbers[index]	делаем эту строку.
8	index = index + 1	Если условие не выполняется, то переходим на эту строку.

Обратите внимание, что часть текста написана друг под другом, а часть с отступами. С одной стороны, это делается для простоты восприятия, с другой стороны, в некоторых языках программирования, например, в Python, используется для отделения блоков кода друг от друга. Есть ряд действий, которые необходимо совершать, некоторые количество раз внутри цикла. Нам надо понять.

- Необходимо проверять и выполнять условие, пока мы в цикле или где-то снаружи тоже нужно делать?
- Вот max = numbers[index] надо делать, только если условия верное или всегда?

На эти вопрос отвечают отступы. Если вы посмотрите, то после while вот эти три строчки находятся немного в глубине, с отдалением от левого края. Эти строчки находятся внутри цикла. То есть весь блок кода мы будем повторять, пока цикл будет выполняться. То же самое относится и к строчке 7. Она у нас находится в отдалении от if. Значит, если условие будет выполняться, тогда мы будем выполнять именно эту строчку. Но как только мы снова попадаем под нашу букву «i», в if уходим на один уровень. Это будет означать, что пошёл следующий блок кода. Если на вопрос if numbers [index] > max мы ответим утвердительно, то будем выполнять эту строчку, а потом перейдём к следующей. А если ответ на вопрос будет отрицательным, тогда мы, соответственно, просто пропустим тот блок кода, который нужно было выполнять и сразу перейдём к следующему.

Это очень удобно. В некоторых языках необходимо брать текст скобочки, в том же Паскале необходимо было выделять, например, блоками begin end (begin — начало, end — конец). Какой-то блок кода могли обернуть этими скобками, но вот в языке Python было придумано такое решение, когда мы просто делаем отступы, и с помощью этих отступов визуально становится понятно, к чему относится та или иная строчка. К циклу while относятся эти три строчки, и мы не перейдём на 9 строку до тех пор, пока этот цикл будет выполняться. Пока он будет крутиться, как любят говорить программисты. Как только условие перестанет выполняться, то есть index станет равен или больше, чем size. То эти три строчки мы пропустим и сразу перейдём на следующую.

Переведём псевдокод на язык программирования Java

Посмотрите немного на код. Мы сейчас попробовали его прочитать. Зная перевод слов, можно понять, что происходит. Есть некоторый цикл, который начинается со слова while (до тех пор), есть некоторые условия, if (если).

Посмотрим на этот код уже не на псевдокоде, то есть без каких-то у нас синтаксических особенностей, а на языке программирования Java. Напомню, что в некоторых языках необходимо указывать тип, создаваемой нами, переменной. Чтобы понимать, что будет храниться внутри переменной. Обратите внимание, в Java две квадратные скобки означают массив. Массив некоторых чисел. В Python можно обходиться без этого, но вот код на Java будет выглядеть так. Если вы попробуйте сравнить с тем, что было на псевдокоде, то всё очень похоже. Нам единственное необходимо оформить этот код в соответствии с синтаксисом языка программирования Java. Сами числа здесь задаются не в квадратных скобках, а в фигурных скобках. В конце каждой строчки нам необходимо поставить точку с запятой. Помимо этого, необходимо указывать тип значения, который будет храниться в этой переменной. Всё остальное максимально похоже на то, что у нас было до этого.

Обращу ваше внимание в Java не обязательно делать эти отступы, вместо отступов блоки кода можно выделять скобками. Но визуально гораздо удобнее воспринимать программу, если соблюдать отступы. Даже если конкретный язык программирования синтаксически этого не требуют, например, в Python это прямо жёсткое требование, и если вы это условия не соблюдаете, программа будет работать неправильно или не будет работать вообще. То в Java это условие не обязательное, но крайне желательное, если вы напишете всё с выравниванием по левому краю, то вашу программу читать станет очень неудобно. Поэтому даже в тех языках, где это не требуется, отступы используются. Итак, надеюсь, что с этой программой вам также всё понятно. Попробуйте сравнить то, что было на псевдокоде. То, что написано в языке Java. У нас есть просто ряд маленьких синтаксических особенностей, которые нам необходимо выполнять, а далее соответственно, всё у нас работает, как и должно.

[00:55:43]

Задача по нахождению скалярного произведения

Теперь рассмотрим следующую задачку. Это будет небольшое упражнение для вас. Необходимо записать алгоритм нахождения скалярного произведения двух векторов. Здесь ребята, которые достаточно давно учились в школе и уже забыли физику и математику, могут немного поёжиться, опять скалярное произведение, опять что-то. На самом деле эта штука, имеет достаточно большое применение не только в математике, но и в программировании. Для некоторых алгоритмов вам это будет важно. Особенно если мы говорим про машинное обучение и какие-то аналитические выкладки, то скалярное произведение будет очень часто применяться. Но чтобы не вспоминать, что это такое. Я просто приведу некоторые формулы.

(a, b) x (c, d) = a x c + b x d

Вот у нас есть некоторые вектор с двумя координатами a и b, и ещё один вектор с координатами c и d. Чтобы найти скалярное произведение, нужно соответствующие элементы перемножить между собой, то есть, a×c, b×d и потом их сложить. Таким образом, вы найдёте скалярное произведение.

Для программистов не сильно важно, что это вектора, имеющие какой-то физический смысл. Для нас это просто некоторые наборы чисел. У нас есть набор из двух чисел и ещё один набор тоже из двух чисел. Что нужно сделать? Нужно соответствующее числа перемножить, а потом сложить всё, что получится. Если снова обратиться к нашим стаканчикам, можно взять, например, две пары чисел 1 и 8, 3 и 2.

Важно, чтобы количество элементов в каждом из наших массивов (наборов) было одинаковым. Если будет разным, то для кого-то не будет соответствующего элемента. Например, если здесь было три элемента, а здесь два элемента, то по похожей формуле мы не смогли бы всё посчитать. Мы бы первый с первым перемножили, второй со вторым перемножили, а третьему просто нет пары. Поэтому скалярное произведение работает только для одинаковых наборов элементов.

Например, есть вектор 1, 8 и вектор 3, 2. Чтобы найти скалярное произведение, необходимо перемножить первый элемент из одного набора с первым элементом из второго набора. То есть мы 1 × 3 = 3 и перемножив вторые элементы (8 и 2), соответственно 8 × 2 = 16. Получается 3 и 16. Складываем их и получаем число 19. Значит, скалярным произведением векторов 1, 8 и 3, 2 будет число 19.

Что вам сейчас необходимо сделать? Попробуйте на листочках или в программе для рисования блок-схем, составить блок-схему для подсчёта скалярного произведения. Для простоты сначала можете начать из наборов с двумя числами. Есть один вектор или один массив из двух элементов и второй элемент также из двух элементов. Потом попробуйте сделать более общий случай. Потому что в математике скалярные произведения считаются для совершенно любых векторов. Здесь может быть по 100 элементов, а может быть по тысяче элементов в каждом, никакого значения это не имеет. Формула работает точно так же. Мы просто соответствующие элементы между собой должны перемножить и сложить всё это в одну переменную, в одно число. На эту задачу выделим 5 минут, думаю, этого времени будет достаточно. Попробуйте решить сначала для двух векторов по два числа, а затем расписать более общий случай.

[01:04:36]

Пример решения задачи

Итак, я думаю, что все справились с задачей хотя бы для двух элементов. Посмотрим на возможное решение, как эту блок-схему можно было бы записать. Мы с вами что имеем? У нас есть опять блока инициализации, и мы с вами вводим два вектора. По аналогии с задачей про друзей и собаку, у нас есть firstVector (1 вектор) и secondVector(2 вектор). Допустим, задали числа 1, 8 и 6, 3. Главное, размеры векторов одинаковые и у каждого элемента есть свой напарник из второго вектора.

Дальше вводим некоторые аккумуляторы. Скорее всего, вы на семинарах уже смотрели, как это работает и для чего нужно. Сюда я буду складывать результат скалярного произведения, соответственно, называю переменную scalarProduct (скалярное произведение). Так как туда я буду складывать сумму, мне нужно будет просто прибавлять произведения пар каждых элементов, изначально возьму 0, потому что относительно там сложения 0 это вот нейтральный элемент. Можем прибавлять 0, можем вычитать 0, ничего не поменяется. Поэтому, когда берём аккумулятор для того, чтобы сложение работало, мы берём 0. Ну и опять же мы начинаем с элемента с index 0, поэтому index изначально взяли равным 0. Начинаем наш цикл. Сравниваем

index со значением размера нашего массива и считаем скалярное произведение. Мы в аккумулятор к уже имеющемуся значению, собственно, он аккумулирует эти значения, будем добавлять произведение двух каких-то элементов. Берём firstVector с индексом 0 и secondVector с индексом 0, здесь будет из первого вектора 1, из второго вектора 6. ScalarProduct сейчас, аккумулятор равен 0. Справа получается 0 + 1 × 6. И в аккумулятор в scalarProduct записывается число 6.

Переходим дальше, увеличиваем наш индекс. Index теперь стал равен 1, возвращаемся в условия нашего цикла, 1<2, ответ положительный. Соответственно, мы возвращаемся к scalarProduct, сейчас у нас сохранится число 6, потому что на предыдущей итерации мы с вами это записали. Итерация — это один проход нашего цикла. 6 плюс firstVector с индексом 1, то есть 8, а здесь второй вектор с индексом 1, 3. Получаем 8 × 3 = 24. Здесь стоит 6, здесь 24, сложили, получилось 30. 30 записали в наш аккумулятор scalarProduct. Индекс теперь стал равен 2. Возвращаемся снова к нашему условию. Отвечая на вопрос 2<2 или, нет? Ответ отрицательный, потому что 2 равно 2. 2 не может быть меньше. Поэтому идём по веточке «no», и печатаем значение нашего скалярного произведения. Действительно, скалярное произведение двух этих векторов первого и второго равно 30, как мы с вами и нашли.

[01:08:09]

Разберём ошибку в блок-схеме

Внимательные студенты уже, наверное, в комментариях написали, какая ошибка есть в этом алгоритме. Причём ошибка не смысловая, то есть, видите, он работает правильно. А ошибка с точки зрения программирования, неточность, которую стоит избегать. Уверен, в комментариях уже есть несколько записей на эту тему, что здесь есть магическое число. О них мы говорили чуть ранее. Если вектора будут размером не 2, а например, по 3 или 5 элементов в каждом, то нужно не только поменять входные значения, но и внутри программы найти, где были жёстко зафиксированы наши магические числа. Которые, когда мы только писали программу, вроде были понятны. Но если мы вернёмся к программе через неделю или месяц, то, скорее всего, забудем про это. Можно поменять здесь массивы, а потом удивляться, почему значение будет неправильным.

Почему только первые два элемента из каждого набора берутся? Да, ваша программа будет работать, она будет выдавать какой-то ответ, но ответ будет неправильным. Например, если здесь было пять чисел и здесь было пять чисел, то программа посчитает скалярное произведение только для первых двух элементов из каждого набора, ответ будет, но неправильный.

Исправим эту ошибку. Думаю, что все уже знают, как это можно исправить. Просто добавим ещё одну переменную, в которой фиксируем размеры массивов. Пока предполагается, что нам точно дадут массивы одинаковых размеров, нигде дополнительные проверки не делаем, чтобы не усложнять программу. Но в целом, если захотим сделать максимально точную программу или алгоритм, нужно где-то ещё проверять, что размеры наших массивов совпадают, чтобы мы могли посчитать скалярное произведение. Но пока оставим это за скобками. Будем считать, что нам точно дадут два одинаковых массива. Здесь укажем размер массивов, если бы их было по пять элементов. Мы поменяли входные данные по массивам и размер массива в блоке инициализации. Дальше, если это где-то используется в нашей программе, есть переменная

size. Поменяв её, мы поменяем и в основном блоке, который производит все вычисления. Такая же схема, только уже без магических чисел.

[01:10:44]

Кухня программиста

Теперь от работы с массивами и от алгоритмов на массивах мы перейдём к следующему. Мы составляем какие-то блок-схемы. Пишем программы на псевдокоде. Видели программы, написанные на Java или Python. Но при этом что происходит с нашей системой и компьютером, когда мы запускаем эти программы? Почему какие-то буквы, команды, числа становятся понятны нашему компьютеру? Как он понимает, какие действия необходимо выполнять? Как он их выполняет?

На самом деле компьютер — просто симпатичная коробочка с проводами. Начнём с бытовых аналогий. Если представить, будто мы повара, и готовим на кухне то, что сначала приходит в голову? Например, нам нужны продукты, кастрюля и плита. И вроде как всё. Можем, что-нибудь приготовить. Но если посмотреть на картину более широко, то нужна целая инфраструктура. Нужна мойка и вода. Для газовой плиты необходим подвод газа, а если плита электрическая необходимо электричество. Если готовить на костре, то потребуются дрова или другое топливо. Из посуды помимо кастрюль нужны ещё ножи, ложки, шумовки, мешалки и многое другое. То есть чтобы мы смогли приготовить нужна огромная инфраструктура. Оказываясь в чистом поле с набором продуктов, но без всей этой инфраструктуры, мы ничего приготовить не сможем. То же самое и с программированием, если взять просто текст, просто программный код, как-то написанный, то без инфраструктуры все условия, циклы и прочее просто не будут работать.

[01:12:54]

Что есть программа?

Теперь познакомимся поближе с этой инфраструктурой. Для этого нам необходимо посмотреть, что такое программа. Программа на самом деле это простой текстовый файл с некоторым расширением. То, что мы писали ранее на псевдокоде или на Java, это просто текст, отформатированный по каким-то правилам. Помимо этого, если мы хотим, чтобы наша программа по двойному щелчку запускалась или, какие-то действия происходили, когда мы по ней кликаем, у неё должно быть расширение. И вот знакомые вам расширение указаны в левом блоке. Все вы знаете про jpg, png, txt, mp3 и так далее. То есть, есть много разных расширений, с которыми вы знакомы.

Что же такое расширение?

Какая магия хранится за этим? Маленький секрет в том, что расширения нужны только для нашего удобства. Например, когда мы два раза кликнем по картинке, компьютер понимал, что с картинкой надо делать. Если компьютер знает, что у неё расширение .png, он запустит программу, способную открыть этот файл. Если расширение файла не указано, компьютер сам

спросит у нас, в какой программе необходимо запустить файл, по которому сейчас два раза кликнули? Так вот, в компьютере есть некоторые переменные (ячейки памяти), хранящими соответствия между расширениями и программами, умеющими работать с этими расширениями. Например, после запуска текстового файла, ваш компьютер (операционная система) смотрят в таблицу соответствия между расширением и программами. Выбирают нужную программу и запускают её. В запущенную программу передаётся файл, который вы хотели открыть.

С картинками всё так же. Если вы два раза кликнули по картинке, то система запустит нужную программу и передаст в неё изображение. Если соответствие не настроено, то мы можем сами выбрать программу для запуска.

Можете попробовать запустить, например, обработчик текстов. Тот же самый Microsoft Word и попробовать открыть в нём изображение. Ошибки произойдут не потому, что Word, не умеет работать с этим файлом, а потому, что он настроен для другой информации. Можно и картинку отобразить в текстовом редакторе и увидеть её содержимое, но фактически это не текст. Поэтому в операционных системах делают просто связки, что одно файлы открываются в этих программах, а другие файлы в других программах.

Если вы работаете не на Windows, а на Linux или Ubuntu, то, скорее всего, знаете, что расширение файлов не обязательно. Когда мы работаем в операционной системе Unix, то очень часто можем самостоятельно сказать, что в какой программе нужно запускать файл. Расширение файлов — это просто некоторые удобства для нас, чтобы системе было проще помочь нам.

Итак, возвращаемся. Есть программа, написанная нами, это какой-то текстовый файл. Мы добавляем к нему расширение. Например:

для Java — .java;

для Python — .py;

для Java Script — .js.

Для каждого языка зафиксировано своё расширение. Если вы создали программу с таким расширением, то дальше системе будет проще понять, что же с файлом делать. Итак, с текстовым файлом с расширением наша программа исходный код имеет.

[01:16:54]
Как компьютер считывает код?

Теперь посмотрим наш компьютер это некоторая система, некоторая коробочка с проводами и с элементами. Подробнее об этом всем вы поговорите на уроках информатики. Но чтобы у нас лекция была связана, об этом тоже поговорим.

Если всё упростить, то в программировании при создании электронной вычислительных машин было принято, что когда тока на каком-то элементе нет, это будет 0, а когда ток на элементе есть, то это 1. Вот можем представить себе два состояния какого-то транзистора или провода, если ток по нему течёт, значит, будет 1, если ток по нему не течёт, значит, будет 0. То есть у нас

есть два состояния, но при этом, даже если мы говорим про числа в нашей привычной системе счисления чисел гораздо больше. Это не только 0 и 1. Тут пришла на помощь математика. Мы можем перевести число из десятичной системы счисления в двоичную систему счисления. Что это значит? Слева запишем числа, привычные нам, то есть 3, 7, 16, 83, а справа эти же числа как набор 0 и 1. Про то, как всё это происходит, мы поговорим на уроках информатики в следующем курсе. Сейчас важно запомнить, что мы каким-то образом сможем это сделать, такая возможность есть. Далее, если у нас с одним элементом всё понятно, вот ток идёт 1, ток не идёт 0. Когда у нас есть целый набор элементов, мы можем просто взять некоторые сочетания элементов и получить необходимое. Соответственно, в нашем компьютере миллиарды таких элементов, миллиарды этих транзисторов, с помощью которых можно представлять информацию в виде 0 и 1.

Хорошо, мы разобрались, что делать, когда есть числа, которые нам нужны и есть 0 и 1. Что же делать, например, с буквами? С буквами ситуация такая же. Компьютер, в общем-то, работает только с 0 и 1. Мы научились, эти нули и единицы переводить в десятичные числа, но что же делать с буквами? Для букв принцип, следующий: придумали таблицу, некоторый массив, где у каждого символа (глифа), который мы видим, есть свой номер (число). В этом случае мы просто будем говорить компьютеру нужный номер, а он из таблицы нужный символ, выведет нам на экран. Таким образом, мы что получим? Что пользуюсь только 0 и 1, мы можем работать не только с числами, но и с любыми символами, которые мы сами сможем закодировать. Подробнее об этом на информатике вы ещё поговорите. Собственно, перейдём немного к истории.

[01:19:51]
Немного истории

Изначально программисты и компьютеры говорили на одном языке

Посмотрим, как выглядели первые компьютеры, не факт, что это прямо самый первый, но это очень древняя модель. И если мы говорим про то, что компьютер работает на основе электричества, и он определяет есть ток или нет тока, и на основе этого какие-то вычисления происходят, то в самых первых компьютерах были просто переключатели или провода, которые необходимо было перетыкать. Если мы каким-то образом соединили провода в компьютере, то на элементах электричество появилось. А если по-другому перемкнули, то на каких-то элементах электричество пропало. Таким образом, если нам необходимо ввести нашу программу, вести какие-то исходные данные, то нам необходимо физически поменять переключатель и переткнуть провода так, чтобы то, что мы хотим в наш компьютер занести, соответствовало напряжению на разных элементах внутри нашего компьютера.

И теперь возвращаемся к самому началу лекции. Помните, я говорил, что есть люди, составляющие алгоритмы, и есть люди, набирающие эти алгоритмы в виде какого-то программного кода. Так вот на компьютере, это можно продемонстрировать. Есть человек, продумывающий алгоритм, входные данные в этот алгоритм, какие данные необходимо подать, чтобы что-то у нас получилось. И есть другие специально обученные люди, инженеры, наборщики кода, которые в этом компьютере переключают переключатели, соединяют разные элементы проводами, делают так, чтобы наши входные данные, соответствовали тому, что придёт физически в компьютер. Далее, когда всё правильно соединено, компьютер запускали

и через какое-то время, а чаще это очень долго могло работать, несколько дней, недели, месяцы получали результат этих вычислений. И если вдруг какие-то элементы были соединены неверно, об ошибочном результате мы узнали бы только в самом конце. Поэтому я и говорю, что профессия составителя алгоритмов и наборщика кода это две разные профессии.

Стали появляться «посредники» между пользователем и компьютером

Соответственно, что случилось дальше? Перетыкать провода и переключать переключатели — это очень неудобной и трудоёмкий процесс. Появились перфокарты. Перфокарта, это кусок картона, на котором в разных рядах есть отверстия. Эту карту вставляют в картоприёмник и начинают замыкать контакты. Смотрите, где есть отверстие, там контакты замыкаются, то есть ток пойдёт. А где отверстия нет, эти два элемента будут изолированы и электричество не пойдёт. По сути, то, что делалось ранее, перетыканием проводов или переключением переключателей, у нас сейчас весь этот набор переключателей можно закодировать в виде отверстий в перфокарте. И уже один человек, если он правильным образом эти отверстия сделал, может настроить много переключателей за один раз. То есть программирование уже на листочке в виде картонных карт появилось в таком виде.

[01:23:06]
Разница между языками «высокого» и «низкого» уровня

Сейчас мы видим, что между пользователем компьютера и компьютером начинают появляться посредники. Изначально они говорили на одном языке. Сначала у нас человек прямо перетыкал провода. Потом появились перфокарты, которые с помощью набора отверстий позволили заносить нашу программу в какой-то вид, пока ещё человеку непонятный, но и машине тоже не сразу понятный, надо было в картоприемник вставлять. Появляются посредники. И чем больше таких посредников между изначальной задумкой (программой), придуманной человеком и машиной, выполняющей программу, тем больше отличаются языки по уровню.

Языки высокого уровня — это когда между человеком и машиной множество посредников. Они максимально похожи на обычную человеческую речь. Когда мы с вами смотрели на Python или Java код, в целом зная английский язык, вы можете этот код прочитать. В языках высокого уровня появляются специальные программы читающие, написанный в текстовом редакторе, файл и преобразующие его в команды для машины. Появляется посредник, который из понятного нам языка, делает программу для машины.

Языки низкого уровня — это когда посредников между человеком и машиной становится меньше. Например, ассемблер, команды, отдаваемые вами в этом языке, не являются обычными человеческими командами. Они гораздо ближе к, выполняемым в вашем оборудовании (процессором, памятью).

Если вы слышали, что есть языки программирования высокого уровня, языки программирования низкого уровня, то это — не ругательство, не обидные эпитеты, а просто указания на то, какое количество посредников у нас там есть.

Два основных типа трансляторов

- Интерпретаторы.
- Компиляторы.

Это два полюса в программировании. Так вот язык Python относится к интерпретируемым языкам, написанное на нём, будет проходить через интерпретатор. Язык Java относится к компилируемым языкам, написанное на нём пройдёт через компилятор. Если на пальцах объяснять разницу между ними, то интерпретатор выполняет вашу программу построчно. Например, ваша программа состоит из 10 строк. Сначала выполнится первая строчка, она переведётся в команду для процессора, он там всё это выполнит. Затем выполнится вторая строчка, она также переводится в специальные команды, и будет выполнена. Потом третья строчка и так далее. То есть по каждой строчке у нас будет происходить какие-то преобразования, выполняться всё на вашем компьютере и, соответственно, результат вы будете видеть поэлементно.

Если мы говорим про компилируемые языки программирования. В этом случае компилятор весь код, сначала переведёт в нужный ему вид, и после выполнит. Если вот так на пальцах это рассуждать, то выглядит именно так. Но на самом деле в современных языках программирования уже такого чёткого деления на чистый интерпретируемые и чисто компилируемые языки практически не существует. Просто есть те, кто ближе находятся к интерпретируемым. Есть те, кто ближе находится к компилируемому. Везде есть уже некоторые смесь этих двух составляющих, но в целом вот выделяют два типа трансляторов.

Трансляторы — это программы пееводящие исходный код,написанный нами текстовый файл, в более понятный для машины код. Хотя на самом деле это всё равно непрямые указания вашему компьютеру. Надо понимать, если заглянуть немного вглубь, то есть операционная система, распоряжающаяся всеми ресурсами компьютера и интерпретаторы (или компилятор) — это всего лишь одна программа из целой сотни запущенных. Когда вы даёте программе задание, она переводит программу в удобный для себя вид, далее обращается к операционной системе. Операционная система выделяет ей определённые мощности и ресурсы. Затем через драйверы обращается к конкретным устройствам, например, к процессору, памяти, сетевой карте или монитору. После этого выводит всё на экран. Обратно всё идёт в том же самом направлении. То есть на самом деле количество посредников между вами и компьютером сейчас очень много. Если раньше мы перетыкали провода, то сейчас не только интерпретаторы и компиляторы этим занимаются. Есть множество посредников между нами и конкретными действиями оборудования. Но вот основных типов, которые в программировании используются у нас два.

[01:28:19]

Ошибки бывают трёх видов

Теперь рассмотрим, какие бывают ошибки в программировании:

- Синтаксические.

- Выполнения.
- Логические.

Синтаксические ошибки

Когда вы пишете программу, на каком-то конкретном языке. То есть вот алгоритм составили, программа должна это решать. Учли всё, что вам необходимо и начали писать решение на конкретном языке программирования. У этого языка программирования есть набор требований, например, надо поставить точки с запятой, условия взять в скобки или, наоборот, скобки не ставятся, ставится двоеточие, то есть, есть некоторые набор требований для конкретного языка программирования. И если вы какое-то требование упустили, например, не поставили точку с запятой, то это будет синтаксической ошибкой. Транслятор вашего языка программирования, когда попытается преобразовать вашу программу в более удобный для себя вид, в котором сможет передать его следующему посреднику, он не справится с задачей. Не сможет, например, отделить одну команду от другой просто потому, что между ними почему-то не стоит точка с запятой, и просто перестанет работать. Это ошибки, которые проще всего найти, ведь транслятор вам просто подскажет, что вот в нужной строчке в какой-нибудь пятой строке у вас, например, не хватает точки с запятой, скорее всего. Попросит вас её исправить. Вы её исправите и запустите программу ещё раз. Это вот синтаксические ошибки, они относятся к конкретному языку программирования.

Ошибки выполнения

Когда синтаксически ваша программа написана, верно, но, например, вы начали делить на 0. В школьной математике вы сами прекрасно знаете, что делить на ноль нельзя, но компьютер заранее это проверить не может. Например, вы делите число «a» на число «b». В одном случае, когда одни входные данные, дали число «b» у вас, например, равно 2 и всё прекрасно работает. А когда вы поменяли входные данные, взяли другие наборы чисел или ещё что-то поменяли, у вас число «b» в одном случае стало, например, 0. И вы взяли какое-нибудь число и поделили его на 0. Программа не знает, что с этим делать, но на этапе синтаксического анализа, когда она только проверяла, насколько красиво вы написали программу, она поверить это не смогла. Что вы будете делать, зависит от заданных входных параметров. Когда-то это работает, когда-то нет. Это называется ошибками выполнения, когда по ходу выполнения программы получилось так, что конкретно в этом случае программа сломалась, что-то работает неправильно. Такие ошибки найти уже сложнее, потому что здесь вам помогает транслятор, а там эти ошибки выявляются не всегда.

Логические ошибки.

Когда у вас сам алгоритм составлен неверно. Например, вы хотели найти максимальное число, но где-то поменяли знак не в ту сторону и нашли минимальное число. При этом синтаксические у вас всё работает, верно. Ошибок выполнения никаких не возникает, но результат выполнения программы не соответствует тому, что нужно было получить. Эти ошибки возникают уже на этапе составления алгоритма, на этапе проектирования блок-схем, когда вы определяете, что же нужно было сделать.

Логические ошибки почти так же сложно найти, как ошибки выполнения. Но я бы сказал, что их даже сложнее найти. Потому что вам компилятор про них или интерпретатор вообще ничего не

скажет. Если ошибки выполнения всё-таки можно детектировать с помощью вашего транслятора. Когда вы в каких-то ситуациях поделили на 0, то всё это появилась. В логических ошибках только вы сами сможете найти проблему, когда поймёте, что программа почему-то сделала не то, что вы от неё хотели.

Поэтому вот когда вы пишете ваши программы, обращайте внимание на то, что могут быть такие типы ошибок. С синтаксическими вам поможет транслятор. Ошибки выполнения нужно просто смотреть и понимать, где ваша программа может сломаться, и это всё может зависеть от входных данных. А логические ошибки — просто необходимо очень аккуратно готовить ваши алгоритмы, блок-схемы, и потом аккуратно же переводить их в синтаксис языка, выбранного для написания программы.

[01:32:42]

Подведём итоги этой и прошлой лекции

Мы поняли разницу между программированием и языками программирования. Это две отдельные профессии, раньше разделённые, а сейчас объединяющиеся ввиду того, что компьютер начинает работать быстрее, языки программирования стали более высокого уровня, и человек, составляющий программу, может написать её сразу на компьютере.

Но хочу предостеречь от того, чтобы вы сразу начали писать программу, как только мы начнём изучать конкретные языки программирования. Можно привести такую аналогию, это как пытаться рассказать сказку до конца, не зная, чем же ваша сказка должна закончиться, пытаясь разобраться с нею походу. Если вы начинаете импровизировать и рассказывать ребёнку какую-то сказку, вы можете закопаться и может оказаться так, что придётся рассказывать её заново. Или вы начинаете писать какую-то книгу и при этом до конца не знаете, чем должно всё закончиться. И мысли могут завести вас в совершенно непонятные ситуации. Можно сказать, что, например, музыканты могут импровизировать, но если вы начинаете играть без нот, это уже достаточно высокий уровень, на котором так делать можно. Но стоит отметить, например, в творчестве это может работать, когда вы на ходу сочиняете, импровизируете, но программирование — это больше инженерная практика. Это инженерные науки, и здесь всё должно быть точно. Вам заранее необходимо знать, что вы должны получить, продумать схему, составить алгоритм и только после этого записать вашу программу в виде какого-то исполняемого кода. Может оказаться так, что решение, придуманное вами, будет лучше работать не на том языке, который вы изначально хотели, а на других технологиях. Поэтому крайне важно сначала всё-таки продумывает решение, а потом только записывать его на каком-то языке, а не пытаться соединять эти две профессии сразу воедино.

Итоги

- Поняли разницу между программированием и языками программирования.
- Решили задачу по нахождению максимального числа. На первой лекции мы решали её без структур данных, сейчас использовали для этого массивы.
- Узнали, как выглядят массивы в реальной жизни и в программировании. В комментариях вы написали, какие есть примеры массивов, и что, не является массивом.

- Решили ещё одну задачу, используя массивы и циклы.
- Заглянули на кухню программиста.
- Немного окунулись в то, как работали компьютеры изначально. Что с ними происходило в дальнейшем. Какие бывают языки программирования. Бывают языки высокого уровня и низкого уровня.
- Узнали, что для запуска программы, нам нужны ещё программы. Трансляторы, которые будут переводить программу из текстового файла, написанного нами, на каком-то языке программирования, в выполняемый для компьютера код.

Ждём вас на следующем семинаре, и следующая лекция будет заключительная по нашему курсу «Введение в программирование». После этого мы с вами будем приближаться к написанию программ на конкретных языках.

Всем спасибо, пока!