GeekBrains

Структурная и процедурная парадигмы (лекция 2)

Парадигмы программирования и языки парадигм

Оглавление

Вступление

Добрый день, коллеги! Меня зовут Александр Левин. Сегодняшняя лекция будет состоять из нескольких разделов: сначала будет разбор структурного и процедурного программирования. Далее будем разбирать примеры кода в каждой из парадигм. После этого, рассмотрим их преимущества и недостатки в разных контекстах. И напоследок, разберем примеры кейсов где можно использовать ту или иную парадигму.

Давайте погрузимся в **структурную** парадигму более детально.

Термины, используемые в лекции

Процедурное программирование – это вид императивного программирования, которое заключается в процессе написания последовательности команд и вызовов процедур.

Структурное программирование (англ. Structured Programming) – это императивный стиль, который основан на последовательном исполнении хорошо структурированных “блоков” без goto.

Процедура (англ. procedure) – последовательность команд, выделенная в отдельный блок кода, который можно вызывать по его имени, то есть имени процедуры.

Утверждение / Последовательность (англ. Sequence) – это однократное выполнение операции.

Ветвление / Условный оператор (англ. Conditional) – однократное выполнение любого количества операций внутри ветвления после проверки условия.

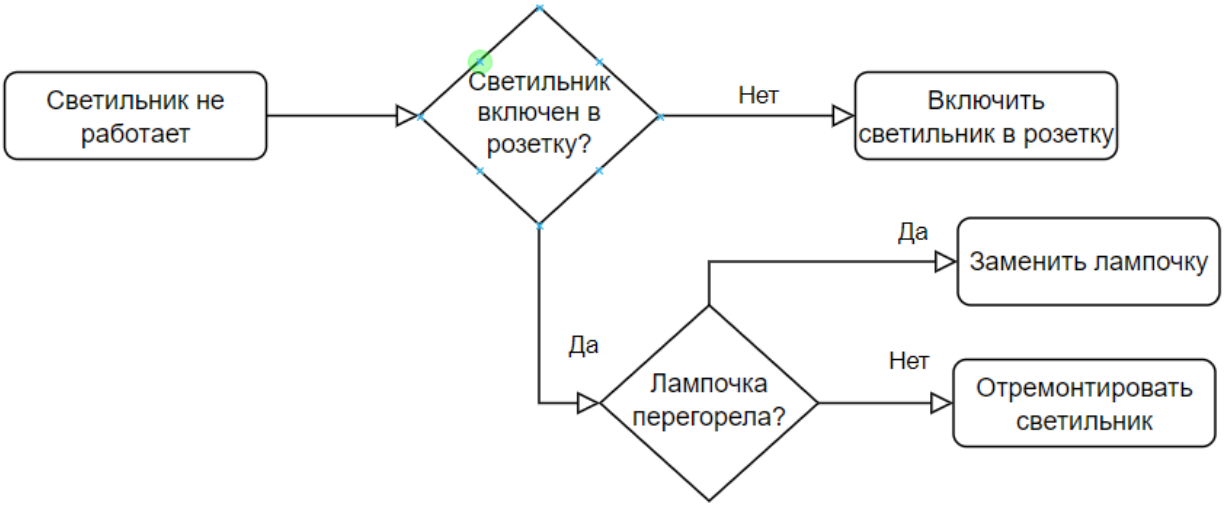
Цикл (англ. Loop) – многократное исполнение любого количества операций внутри цикла до тех пор, пока заранее определённое условие выполняется.

Оператор GOTO – оператор безусловного перехода управляющего потока к указанной точке программы.

Структурное программирование

Структурное программирование - это императивный стиль, который основан на последовательном исполнении хорошо структурированных “блоков” без goto. Исполняемые блоки бывают трёх типов: **утверждение**, **ветвление** и **цикл**. **Утверждение** - это однократное выполнение следующей написанной операции. **Ветвление** - это однократное выполнение любого количества операций внутри ветвления после проверки **условия** (условие, естественно, задаём мы сами). А **цикл** - это многократное исполнение любого количества операций внутри цикла до тех пор, пока заранее определённое **условие** выполняется.

Также, на вводной лекции мы разобрали несколько примеров: в блок-схемах и на псевдокоде, с циклами и без них. Вот пример одной из них.



Основное отличие структурного программирования от императивного - это запрет на использование оператора “**goto**”, который древние программисты использовали для того, чтобы продолжить исполнение с заданной строку из любой точки кода. Например реализация цикла **while** с использованием goto на языке C++:

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

int main() {
    int a = 5;
    LOOP:do {
        if (a < 20) {
            a = a + 1;
            goto LOOP;
        }
        cout << "value of a: " << a << endl;
    }
}
```

С goto или без

Основная дискуссия вокруг goto состоит в том, что с одной стороны - goto крайне сильно вредит читабельности и **структурированности** программы. Есть такой термин - “**спагетти-код**” - он как раз довольно точно описывает программный код без чёткой структуры (а метафора со спагетти - намекает на запутанность). Такой код труднее поддерживать, дорабатывать и оптимизировать, потому что вообще не очевиден **порядок исполнения программы** (control flow). То есть сложно сказать, что будет исполнено в начале, а что в конце, т.к. goto позволяет перескакивать из любой точки в любую другую точку кода. С другой стороны, goto упрощает процесс первоначальной разработки, когда ещё нечего поддерживать и оптимизировать. Используя goto, вам не нужно думать о конечной структуре заранее: если что-то пошло не по плану - просто пишите goto и программа работает (если повезло).

Несмотря на то, что существует ряд кейсов, когда код с **goto** и понятнее и эффективнее (например, с помощью goto бывает удобно управлять выходами из вложенных циклов, особенно, когда уровней вложенности > 2), в современном мире этот оператор считается абсолютным злом, к которому не стоит прибегать никогда или почти никогда. Именно эта точка зрения на сегодняшний день является доминирующей.

Как это обычно и бывает, есть лагерь специалистов, которые выступают “ЗА” и, аналогично, их антагонисты, которые выступают ”ПРОТИВ”. И споры идут на эту тему пламенные. Подискутировать по возможности с опытными людьми на данную тему, я всем крайне рекомендую, для расширения кругозора. Но в рамках данного курса важнее всего для нас будет следующий вывод: у данного инструмента, как и у любого другого, есть свои плюсы и минусы. И в некоторых ситуациях его

использовать может быть удобно и впоследствии даже хорошо, а в некоторых наоборот.

Например, при использовании goto,

- В некоторых случаях, может быть получена наиболее эффективная реализация алгоритма и наименьшая длина программы
- При грамотном использовании, оптимальное использование ресурсов

Но с другой стороны,

- Возможно получить спагетти код: то есть неочевидную последовательность исполнения программы
- Возникает вероятность возникновения ошибок

А если строго не использовать goto, то с одной стороны:

- Программа четко структурирована и состоит из блоков кода
- Спагетти кода нет: последовательность исполнения выглядит наглядно

Но с другой:

- Код может стать сложнее: больше строк, чем могло бы быть
- В некоторых случаях будет невозможно получить наиболее оптимальный алгоритм

Ещё подробнее об этом можно прочитать главу "Оператор goto" в книге Стива Макконелла "Совершенный код".

В любом случае, в современных языках программирования этот оператор вы почти никогда не встретите. И сейчас для большинства программистов кажется очевидным, что и без этого оператора можно написать что угодно. Но с точки зрения прогресса, важнее всего тут то, что в то время - эта мысль была прорывом компьютерной науки.

История структурного программирования

Изобретение структурного программирования как раз и было ответом на проблемы, связанные со спагетти-кодом, нечитабельностью и тяжелой поддерживаемостью программ. Но перед этим случилось очень важное событие: умные люди доказали, что структурный подход к созданию программ не просто удобный, но и универсальный.

В 1966 году вышла очень важная статья двух итальянских математиков: Джузеппе Якопини и Коррадо Бём-а. Она называлась “Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules”, что можно перевести как “Диаграммы

потоков, машины Тьюринга и языки со всего лишь двумя правилами формирования”. Говоря простым языком, в ней авторы доказали, что любой алгоритм может быть представлен в структурном виде с использованием исключительно трех конструкций: **последовательностей**, **ветвлений** и **циклов**. Именно этот сильный результат был положен в основу создания структурного программирования.

В 1968 году вышла статья, довольно известного сегодня, голландского ученого-компьютерщика **Эдсгера Дейкстры**, озаглавленная "**Go To Considered Harmful**" (то есть, “Go To - вреден”). В этой статье, утверждалось, что неконтролируемое использование инструкции **goto** в программировании часто приводит к созданию программ, которые трудно читать, понимать и, самое главное, - модифицировать. Дейкстра утверждал, что использование структурных управляющих конструкций (уже известных нам последовательностей, ветвлений и циклов) - приведет к созданию более читабельного и легко поддерживаемого кода.

Другие ученые-компьютерщики, такие как **Тони Хоар** и **Никлаус Вирт**, также внесли свой вклад в развитие **структурного программирования**. Хоар в 1969 году предложил использовать конструкцию **if-then-else**, в то время как Вирт разработал язык программирования **Pascal**, который включал ряд структурных управляющих конструкций.

В общем и целом, примерно в 1970-х компьютерное комьюинити договорилось на том, что **goto** - это плохая практика и структурированность важнее, потому что с какого-то момента распутывать программы с **goto** становится неадекватно сложно. Поэтому **структурное** программирование быстро получило широкое признание и стала доминирующей.

Примеры языков

К примерам языков программирования, в которых goto реализован относятся

- C, C++, C#
- Pascal
- Perl
- PHP
- Basic
- Go

А например в

- Python
- Java
- Ruby

- Swift
- Kotlin

встроенного оператора goto нет.

Таким образом, структурированное программирование - это стиль программирования с использованием чёткой **структуры**, которая обеспечивается утверждениями, ветвлениями и циклами, плюс использование goto строго запрещено.

Теперь рассмотрим процедурное программирование.

Процедурное программирование

Отличие процедурного программирование от чисто императивного - возможность оборачивать свой код в процедуры. То есть, на самом деле, возможность подняться на более высокий уровень абстракции, вместо того, чтобы писать код сплошным полотном.

> 💡 Напомню, что **процедурное программирование** - это вид императивного программирования, которое заключается в процессе написания последовательности команд и вызовов процедур.
>
> 💡 Также напомню, что у процедур как правило есть **доступ к памяти** компьютера, на котором они исполняются, и следовательно там же существуют переменные, в которых можно что-то хранить, а потом изменять это что-то по мере исполнения программы.

Здесь тоже все исполняется последовательно (и, уже довольно давно, без GOTO). Но главное концептуальное отличие этой парадигмы от чисто императивного заключается в акценте на **процедурах**, на которые может быть разделён ваш код. **Процедура** - это последовательность команд, выделенная в отдельный блок кода, который можно вызывать по его имени, то есть **имени процедуры**. По сути - это программа внутри программы, которую можно вызывать по имени.

🔥 Здесь надо оговориться, что зачастую, 3 термина: **процедура**, **функция**, **метод** можно встретить в синонимичной форме. Но эти термины означают разное. **Функция** - это подпрограмма, результатом выполнения которой является возврат значения. Например, если мы определили что подпрограмма “сумма чисел” примет два числа на вход и вернёт их сумму, то она является функцией. **Процедура** - это подпрограмма, которая значения не возвращает, то есть смысл её выполнения - это те операции, которые были исполнены в процессе. С **методом** всё проще: **метод** - это просто подпрограмма, которая принадлежит классу. Причём **метод** по своей форме может являться как **функцией**, так и **процедурой**. О методах мы будем говорить подробнее в лекции об Объектно-ориентированном подходе.

🔥 Почему происходит эта путаница? В более ранних языках, например в Pascal-е, процедуры и функции существовали как отдельные операторы, с помощью которых можно было определить соответствующую сущность. Но со временем, в более новых языках, это различие в реализации было нейтрализовано. И сегодня, синтаксис процедуры и функции одинаковый в большей части языков. Реализуется этот синтаксис с помощью типов данных “пустоты”, таких как “void”, “None” или “NULL”. Так как, если есть такой тип данных, то мы про любую функцию можем сказать, что она возвращает такой пустой тип, а значит является процедурой по определению. Поэтому, используется термин “**процедура**” в основном в контексте обсуждения парадигм программирования, либо просто как синоним слова “**функция**”.

Ещё одна пара синонимов, которую стоит упомянуть: **процедурное** и **императивное программирование**. Строго говоря, это не совсем одно и то же. Несмотря на то, что все императивные языки, вплоть до низкоуровневого языка Ассемблера и даже машинного кода (хотя там это может быть затруднительно), поддерживают объявление процедур, у вас всё ещё есть возможность взять любой процедурный язык, отказаться от использования процедур, при этом использовать goto и писать все с нуля. В этом случае ваша программа будет исполнена чисто в императивной парадигме. Но так как сплошное полотно кода никто не пишет уже давно (примерно с 1950х годов), а процедурное программирование заполонило всё пространство внутри императивного, люди часто используют термин “императивный”, чтобы сказать “процедурный” и наборот.

Помимо этого, любая современная программа в процедурном стиле почти всегда является ещё и структурной (то есть включает в себя её принципы), но всякая структурная далеко НЕ всегда является процедурной. Несмотря на то, что процедурные программы называть структурными конечно же не принято.

В итоге, получается что эти три множества: императивная, структурная и процедурная парадигмы довольно плотно пересекаются, так как императивные языки как правило поддерживают сразу и структурный и процедурный подход. Но важно здесь именно то, какой функционал вы хотите использовать. Отказываетесь от **goto** в пользу структуры - значит используете структурную парадигму. Используете оформление кода в виде процедур - значит и процедурную используете. А если ни того, ни другого не делаете - значит просто пишете какую-то последовательность команд, то есть остаётесь в рамках императивной парадигмы.

История процедурного программирования

Процедурное программирование появилось гораздо раньше структурного, несмотря на то, что сегодня они практически слились в единое целое. По сути, с этой парадигмы началась история языков программирования в целом.

Сначала был машинный код - последовательность команд, воспринимаемых и исполняемых процессором напрямую. Машинный код - не является языком программирования, так как это инструкции, которые не нужно транслировать, и они сразу напрямую подаются на вход в процессор. На машинном коде было писать сложно и долго, несмотря на то что, такие программы исполнялись максимально эффективно.

Для того чтобы разработка стала более удобной, был придуман язык ассемблера - один из первых низкоуровневых языков програмирования. Assembler (в переводе - “сборщик”) - это транслятор языка ассемблера в машинный код. А язык ассемблера - это система записи машинных команд процессора в удобно человекочитаемом формате. На самом деле, язык ассемблера - это не конкретный язык, а группа языков, разработанных под определенную архитектуру процессора. То есть существовало множество архитектур процессора, каждая архитектура процессора “разговаривала” на своём машинном коде, соответственно для каждой из них нужен был свой язык ассемблера. Поэтому часто про эти языки говорят во множественном числе - “языки ассемблера” или “ассемблеры”. Но главное, что суть данного изобретения была в переходе от нулей и единиц к словам похожим на человеческие.

Время шло, компьютеры становились мощнее, программы объемнее, пришла эра языков высокого уровня. У этих языков появились компиляторы, а значит разрабатывать и исполнять программы теперь стало можно на любом процессоре. Одним из первых широко известных процедурных языков высокого уровня был язык Fortran. Его разработали в 1957 году для научных вычислений.

- 💡 Кстати говоря, GOTO в языке Fortran был, но в более поздних процедурных языках, таких как Pascal, этот оператор уже стал вытесняться структурной парадигмой.
- 💡 Между прочим, древний Fotran до сих пор используется в NASA и в некоторых узких научных кругах.

Примеры процедурных языков

Теперь обсудим языки программирования. Поскольку понятия императивной, структурной и процедурной парадигм появились примерно в одно и то же время и их принципы являются вложенными друг в друга множествами, то языков которые бы были **чистыми** в одной из этих парадигм - практически не существует. Некоторые языки всё ещё поддерживают goto, но как правило можно обойтись и без него. Большинство языков поддерживают и ветвления, и циклы, и процедуры (или функции). Поэтому принято говорить, что язык X поддерживает парадигму P1, но не поддерживает парагиму P2. А язык Y, например, наоборот - поддерживает P2, но не поддерживает P1. Давайте рассмотрим некоторые из известных языков в таком формате.

На приведенной таблице вы можете увидеть какие из парадигм поддерживаются приведенными языками программирования.

Язык программирования	**Структурная**	**Процедурная**
C, C++, C#	Да	Да
Python, Java	Да	Да
Pascal	Да	Да
Assembly	Нет	Да
HTML*	Да	Нет
BASIC	Нет	Да

* HTML - не совсем язык программирования (но язык разметки) и скорее декларативный, чем императивный, но структурная парадигма реализована за счёт последовательностей из тэгов и атрибутов.

Но всё-такие большинство императивных языков как правило поддерживают обе парадигмы, поэтому эти принципы в нашем современном восприятии уже постепенно сливаются в единое целое.

Давайте теперь разберём примеры структурного и процедурного кода.

Примеры кода

Countif

Рассмотрим следующую задачу. Предположим, нам необходимо реализовать программу, которая получает на вход список **li** и число **n**, а возвращает количество элементов в списке **li** равных этому заданному числу **n**. Проще говоря, в терминах Excel-а - реализовать функционал команды countif. Если у вас уже есть опыт программирования в данной парадигме - попробуйте поставить лекцию на паузу и выполнить данное упражнение самостоятельно. Если нет или задача кажется слишком сложной - ничего страшного, продолжайте смотреть. Сейчас будет 3 секунды паузы на принятие решения. –пауза 3 секунды –. Давайте разберём решение данной задачи выполненное в структурной парадигме на языке Python:

Структурная реализация

```
1 ## Structured Programming
2 input_li = input('Enter list li: ') # [1,2,3,4,5,6,6,5,5,1,2,3,5]
3 input_n = input('Enter number n: ') # 6
4 counter = 0
5 for el in input_li:
6     if el == input_n:
7         counter += 1
8 print(counter)                     # 2
```

В первой строке написан комментарий. Во второй и третьей - мы объявляем ввод данных с клавиатуры. Далее объявляется переменная-счётчик “counter” и перебираются элементы из входного массива. Для каждого элемента выполняется проверка: если он равен числу n - увеличиваем counter на единицу. Теперь перейдём к процедурной парадигме:

Процедурная реализация

```
 1 ## Procedural Programming
 2 input_li = input('Enter list li: ') # [1,2,3,4,5,6,6,5,5,1,2,3,5]
 3 input_n = input('Enter number n: ') # 6
 4 def countif(li, n):
 5     counter = 0
 6     for el in li:
 7         if el == n:
 8             counter += 1
 9     return counter
10
11 countif(input_li, input_n)          # 2
12
```

Здесь все то же самое, кроме того, что почти вся содержательная часть программы обернута в функцию countif и вызывается в 11 строке. В итоге, и там и там считается количество элементов в списке соответствующих значению. Но первый пример использует чисто структурный подход, а во втором часть программы исполняется через вызов процедуры.

Сортировка пузырьком

Рассмотрим ещё один пример. На этот раз вам необходимо реализовать сортировку пызурьком в Python. Также, как и с предыдущим примером, если вы понимаете о чем идет речь или где об этом почитать, попробуйте написать программу самостоятельно, а если нет, продолжайте смотреть. И сейчас также будет пауза 3 секунды на принятия решения (–пауза 3 секунды–).

И так, задача сортировки массива - это каноническая задача, в которой на вход вашей программе подаётся массив содержащий элементы, которые должны быть упорядочены по какому-то признаку. Простейший пример упорядоченности - упорядоченность чисел, то есть расположение чисел в порядке возрастания или убывания. Конечно, бывают и другие типы порядков, например, слова могут быть упорядочены в соответствии с лексикографическим порядком, но это уже немного другая тема. А в наиболее распространенном случае, массив, который нужно упорядочить содержит числа.

Для того, чтобы это сделать существует множество алгоритмов. Один из них - сортировка пузырьком. Сортировка пузырьком - далеко не самый эффективный из них, но очень подходящий в качестве тренировочного примера. А если вам интересно ещё поизучать про алгоритмы сортировки, рекомендую книгу “Грокаем алгоритмы” - Адитья Бхаргава.

Предположим, мы хотим отсортировать массив в порядке возрастания чисел. Основная суть сортировки пузырьком заключается в том, чтобы перебирать элементы из массива парами, сравнивая элементы пары между собой на каждом шаге. Поскольку мы хотим отсортировать массив в порядке возрастания, то если очередной левый элемент из пары больше другого, поменяем элементы местами, а если наоборот - не будем ничего делать. Далее будем проходиться так по списку до тех пор, пока на каждом шаге прохода не встретиться ни одного случая, когда левый элемент больше правого - то есть это и будет означать, что массив успешно отсортирован.

Лучше всего создать интуицию об алгоритме поможет визуализация сортировки

Псевдокод этого алгоритма выглядит следующим образом:

```
for i in range(n):
    for j in range(n - i - 1):
        if arr[j] > arr[j + 1]:
            swap(arr[j], arr[j + 1])
```

Первый цикл нужен для того, чтобы проходов по массиву было столько, сколько элементов в массиве. Это так, поскольку на каждом проходе каждый элемент сдвигается не более чем на один индекс влево, хотя в худшем случае на вход мы можем получить массив, в котором наименьший элемент расположен максимально справа (то есть в конце массива). Второй цикл нужно, чтобы обеспечить собственно каждый проход по элементам массива и, как вы уже скорее всего заметили, этот цикл меньше чем предыдущий на i - 1 шаг. Почему это так: поскольку к i - 1 шагу ровно столько элементов справа отсортированы гарантировано, то есть они уже стоят на своих наиболее правых позициях и правее не поедут. Вначале в самый правый конец гарантированно уедет самый большой элемент (так как при сравнении со всеми он будет больше), затем на следующем шаге к нему приедет второй по величине и дальше чем он не поедет, и так далее. Поэтому из n во втором цикле вычитается i. А единица вычитается, поскольку счёт в программировании как правило идёт с нуля, несмотря на то, что после нулевого шага один элемент (самый большом) уже стоит в правом конец, поэтому ещё минус 1. И в итоге n - i - 1. Далее идёт условный оператор (ветвление), который проверяет является ли элемент слева большем чем элемент справа. И наконец, если да, меняем их местами. Давайте рассмотрим реализацию сортировки пузырьком на языке Python.

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                # Меняем элементы местами
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

# Example usage
my_array = [64, 25, 12, 22, 11]
sorted_array = bubble_sort(my_array)
print(sorted_array)
```

Здесь всё очень похоже на псевдокод, но кое-что добавилось: появилось определение n - как длины массива (строка 2), появился сам входной массив (строка 11), вывод результата (строка 13), а ещё появилась функция сортировки (строка 1) и её вызов (строка 12). Рассмотрим данный пример в контексте парадигм. Попробуем ответить себе на следующие вопросы: является ли данная

программа императивной? Является ли она структурной? И является ли процедурной? И почему это так?

Сейчас самое время поставить лекцию на паузу и записать ответы на данные вопросы. Переходим к ответам.

Данная программа является императивной, так как реализация сортировки написана пошагово, на уровне манипуляций с конкретными переменными: их значениями и памятью. Она также является структурной, так как весь алгоритм реализован с помощью трех управляющих конструкций: последовательного исполнения строк, циклов (for) и ветвлений (if) и без использования оператора goto. И конечно же она также является процедурной, поскольку мы реализовали сортировку и оформили её в виде процедуры, а точнее функции (хотя в общем-то могли и просто написать два цикла и ветвление без def и return). Можно ли было сделать по-другому? Конечно! Но в данном случае, функция - это удобно, потому что входные массивы могут быть разными, и конкретно этот кусок кода с сортировкой с большой вероятностью будет переиспользован несколько раз, возможно в разных контекстах. И вам удобнее вызывать его по имени вместо того, чтобы копировать в разных частях финальной программы - такой стиль называется “дублирование кода” и считается плохой практикой.

Таким образом, на все три вопроса - ответ утвердительный. Конечно, мы разобрали в качестве примеров очень небольшие программы. В реальности программа может сочетать в себе разные парадигмы. Но самое важное - понимать какой выбор имеется у вас как у разработчика и какой лучше применять в каждом контексте.

Поэтому, давайте перейдём к преимуществам и недостаткам каждой из этих двух парадигм, чтобы попробовать в этом разобраться.

Преимущества и недостатки

Начнём со структурной парадигмы.

Преимущества структурного программирования:

- Легче искать: так как код структурирован, последовательность его работы легко понятна и можно быстро найти нужную часть
- Легче поддерживать: код разделен на функциональные блоки, поэтому изменение кода в одном блоке не повлияет на другие
- Легче тестировать: код разделен на функциональные блоки, каждый из которых может быть протестирован отдельно (например с помощью unit-тестирования)
- Уменьшение вероятности ошибок: структурное программирование позволяет лучше контролировать процесс разработки и предотвращать ошибки на ранних стадиях (опять же из-за удобного разделения на блоки)

Недостатки структурного программирования:

- Возможная неэффективность: структуры в структурном программировании позволяют реализовать любой алгоритм (то есть, обладают “полнотой”), но нет гарантий что эта реализация оптимальнее всех
- Нехватка абстракций: может приводить к дублированию кода и, соответственно, ошибкам
- Трудности при распараллеливании: если ваш проект необходимо запускать в режиме многопоточности, структурный код будет распараллелить сложнее, чем процедурный
- Риск потери данных: так как все переменные скорее всего находятся в одной области видимости, то есть шанс потерять “оригиналы”

Преимущества процедурного программирования:

- Легче понимать что происходит в коде: поскольку некоторые части кода обернуты в процедуры, понимать его становится проще с помощью чтения “сверху - вниз”, начиная с высокого уровня абстракций и заканчивая пошаговыми манипуляциями с конкретными переменными
- Многопоточность: выполнение процедур легко переписать на режим многопоточности. Чтобы распараллелить программы, как правило достаточно сказать компьютеру “выполняй эту процедуру параллельно n раз“
- Возможность повторного переиспользования кода: так как он упакован в красивые процедуры, которые можно объявить один раз, а потом вызвать откуда угодно
- Сообщество: широкая распространённость парадигмы обеспечивает быструю взаимопомощь внутри сообщества программистов
- Универсальность: поскольку заранее почти никаких процедур верхнего уровня не определено, а следовательно и конвенции по их использованию не связывают руки программисту

Недостатки процедурного программирования:

- Разработка может длиться дольше, т.к. программа устроена сложнее, чем просто структурные последовательности, а значит и поиск конкретного куска кода длится дольше, и в итоге дебажить может быть сложнее
- Потеря данных внутри процедур и невоспроизводимость: поскольку во главе стола процедурного программирования стоят именно процедуры, которые не знают ничего про данные, кроме того, “что с ними необходимо сделать”, то и промежуточные значения внутри процедур хранится не будут если только явно это не попросить

А в каких ситуациях нам важны эти свойства? Вкратце: если важна структура и надёжность - выгоднее **структурное** программирование. Хотите и можете разбивать код на переиспользуемые части - **процедурное**. И очень важно, что два этих утверждения не противоречат друг другу: можно применять и то и другое

одновременно. Давайте разберёмся подробнее какие могут быть признаки того что ту или иную парадигму использовать скорее стоит или точно нет.

Процедурное программирование обычно используется в проектах, где организация кода с помощью функций и процедур является хорошей идеей. Например:

- Когда необходимо решить задачу, которая может быть “удачно” разбита на более мелкие подзадачи или шаги, выполняемые последовательно. Причём эти подзадачи удобно реализовать с помощью функций или процедур
- Когда в вашем проекте есть части кода, используемые несколько раз
- Когда проект имеет линейную структуру и не требует сложных (то есть, составных) манипуляций с данными

С другой стороны, скорее всего не стоит использовать процедурную парадигму (по крайней мере, в чистом виде), если у вас:

- Большие объемы данных, с которыми нужно производить сложные манипуляции и/или хранить промежуточные состояния
- Сложная система или приложение не укладывающаяся в форму последовательности шагов и их результатов

Структурное программирование может быть хорошим выбором в следующих случаях:

- Когда вы решаете задачу, которая может быть разбита на неповторояющиеся структурированные куски кода
- Когда вам важна стабильность работы и предсказуемость (если сравнивать с неструктурным подходом)
- Когда важно, чтобы весь код был “на ладони”, например при разработке прототипов или обучении
- Когда необходима быстрая разработка небольшой программы, которую не планируется расширять в ближайшее время

С другой стороны, структурная парадигма станет не самым лучшим выбором, если:

- Некоторые части вашего кода планируется использовать множество раз. В этом случае его будет выгоднее обернуть в процедуру, то есть обратиться к процедурной парадигме
- Вас в большей степени интересует эффективность алгоритма, чем читаемость кода и его структура. В этом случае, вам может быть полезен табуированный goto

Предлагаю рассмотреть некоторые кейсы из жизни на конкретных примерах.

Реальные кейсы

Кейс - Проект построения дашбордов

Предположим, вы программист и получили заказ от небольшой компании, в которой часть решений принимается на основе визуализации созданной в Excel-е. Руководство приняло решение сэкономить время своих сотрудников на ручных операциях и автоматизировать их с помощью кастомной программы, которую вам и заказали. Собственно процесс as-is выглядят следующим образом: человек открывает таблицу, вручную перебивает в неё данные, потом строит какие-то графики, основываясь на введенных данных. Вы поговорили с аналитиками, которые сейчас создают эти графики, и они рассказали вам что и как именно они делают. Вам остается только это реализовать. Поскольку мы на данный момент мы разобрали только две парадигмы, давайте ограничим вопросы следующими:

- Насколько важно применять структурную парадигму?
- Необходимо ли использовать процедурное программирование?

Оговоримся, что отвечать на эти вопросы не обязательно перед началом написания кода. Вы можете ответить и поменять свои ответы и на первый и на второй вопросы в процессе работы над проектом. Но скорее всего вы придете к чему-то похожему на следующие размышления.

Про **структурную парадигму** ответ довольно простой: скорее всего у вас особо не будет выбора. Вы просто будете её использовать. Почему так: во-первых, структурная парадигма, как мы помним, запрещает использование goto в целях обеспечения стабильности программы и ясности её структуры для читателей программы, а в данной кейсе вы не только создаете программу которую очень вероятно будут читать и дорабатывать аналитики и разработчики компании заказчика, но и использоваться эта программа будет для обработки данных и создания графиков, что ещё больше повышает важность читабельности этого кода в случае возникновения ошибок. Точнее, читабельность будет важнее алгоритмической эффективности, которую может привнести goto. Во-вторых, по состоянию прикладной области на сегодняшний день, языки программирования, которыми вас ограничит данный кейс вероятнее всего не будут поддерживать goto (речь идёт, прежде всего, о языках Python, R и Java, в которых goto не поддерживается нативно).

Про **процедурную парадигму** главный вопрос, который стоит задать себе - “собираюсь ли я переиспользовать части программы, которые сейчас пишу?”, и если в любой момент написания вы ответили на этот вопрос утвердительно - вам стоит использовать процедуры для оформления этой части в подпрограмму.

Кейс - Парсер курса валют

Рассмотрим ещё один кейс. Предположим, вас попросили написать программу, которая раз в 1 час находит и скачивает курс конвертации нескольких валют с сайта ЦБ. Попробуем ответить на те же самые вопросы:

- Насколько важно применять структурную парадигму?
- Необходимо ли использовать процедурное программирование?

Скорее всего псевдокод вашей программы будет работать примерно так:

```
Делать каждый час:
    Получить сайт ЦБ
    Для каждой из валют:
        Найти курс конвертации
        Сохранить этот курс
    Разорвать соединение
```

Похоже, что использовать неструктурное программирование в данном случае нет никакого смысла. Поэтому, с легкой душой принимаем лучшую практику **структурного** программирования. Теперь рассмотрим **процедурное**. Уже по псевдокоду видно, что в данной программе будет несколько весьма нетривиальных шагов, которые довольно удобно завернуть в процедуры и использовать каждый раз независимо. Например, “получить сайт ЦБ” - скорее всего либо скачает сам сайт, либо получит подключение к базе данных ЦБ по API, если таковой имеется. В любом случае - это сто процентов какое-то непростое действие, которое уже реализовано в сторонних библиотеках, но является процедурой или функцией (в зависимости от функционала). Далее следуют слова “для каждой из валют - найти курс конвертации”. Сразу представляется, что “найти курс конвертации” - это функция, которая получает на вход собственно название валюты, курс которой необходимо найти и возвращает значение этого курса. “Сохранить” - тоже делается какой-то процедурой: самописной, встроенной или из библотеки - неважно. В итоге мы видим, что процедурного программирования скорее всего не избежать в данном кейсе.

Кейс - Прототип

С последним кейсом можно встретиться в реальной жизни довольно часто. Вас попросили реализовать алгоритм, который пока что неизвестно будет ли работать, и для того чтобы убедиться в этом, необходимо прогнать несколько тестов и получить прототип. Или просто посчитать какое-то значение на лету. Или провести какую-то простую одноразовую аналитику без автоматизации. В этом случае, ваша программа - это все еще последовательность детализированных команд, без описания результата, то есть, в любом случае, все еще **императивная программа**. Но уже далеко не факт что **процедурная**.

Поскольку писать вы будете быстро, не думая о дальнейшем использовании этого кода, возможно вам просто не понадобится заворачивать код в процедуры, думать над тем какие у них входы и выходы, равно как и ставить исполнение этого скрипта в режим многопоточности. Так что вероятно, вы не будете использовать **процедурный подход** в данном кейсе.

Про структурную парадигму - вопрос более интересный. Все еще есть случаи, когда **неструктурное** программирование показывает лучшие результаты по эффективности за счет использования нашего любимого (или нет) goto. Нужно его использовать или нет - вопрос к производительности и (самое важное!) вашему опыту. Если вы умеете оптимизировать алгоритм за счет goto и сходу знаете как это сделать в данном конкретном случае - прекрасно! По default-у, конечно же, использовать его все же не стоит.

Итоги лекции

Сегодня мы погрузились в структурную и процедурную парадигмы. Изучили их основные принципы и особенности на конкретных примерах. Разобрали их историю. Посмотрели примеры программ реализованных в этих парадигмах и поговорили про преимущества и недостатки каждой из них. И наконец, разобрали какую роль эти преимущества и недостатки могут играть на примере реальных практических кейсов.

Если у вас остались вопросы, вы можете написать их мне в Telegram:
@alexlevinML

Спасибо что прослушали это лекцию и увидимся в следующей!

—

Домашнее задание

Текст домашнего задания и условия его сдачи (ссылка, файл и пр.) Обычно после лекций мы не даем домашнее задание - оно чаще всего будет дано студенту после семинара. Посоветуйся с методистом, должно ли тут быть домашнее задание.

Что можно почитать еще?

1. Книга “Грокаем алгоритмы” - Адитья Бхаргава
2. Статья Джузеппе Якопини и Коррадо Бём-а “Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules”
3. Статья Эдсгера Дейкстры "Go To Considered Harmful”
4. Книга “Совершенный код” Стива Макконелла
5. Статьи на Хабре:
    a. https://habr.com/ru/post/114470/
    b. https://habr.com/ru/post/114211/
    c. https://habr.com/ru/post/271131/

Используемая литература

1. https://www.l3harrisgeospatial.com/Learn/Blogs/Blog-Details/ArtMID/10198/ArticleID/15289/Why-should-GOTO-be-avoided
2. https://cs.lmu.edu/~ray/notes/paradigms/#:~:text=Imperative%20%3A%20Programming%20with%20an%20explicit,%2Dfree%2C%20nested%20control%20structures.