# Сети ЭВМ и телекоммуникации

## Модель взаимодействия открытых систем ISO.

Модель взаимодействия открытых систем OSI (Open Systems Interconnection) — базовая основопологащая модель, описывающая структуру передачи данных от одного приложения другому. Используется как абстрактная схема описания уровневого подхода описания работы Модель OSI состоит из семи концептуальных уровней. Каждый из этих уровней соответствует конкуретной задаче, соответствует выполнению определенной части некоего алгоритма. Благодаря модели OSI становится более понятна парадигма взаимодействия сетевого оборудования и программного обеспечения.

* Уровень 0. **Среда**. Данный уровень представляет посредников, соединяющих конечные компоненты сетевой структуры: кабели, радиолинии и т.д. Поскольку этот уровень де-факто не является элементом схемы, он указывает только на среду.
* Уровень 1. **Физический**. Включает физические аспекты передачи двоичной информации по линии связи. Детально описывает, например, напряжения, частоты, природу передающей среды. Этому уровню вменяется в обязанность поддержание связи и приём-передача битового потока. Безошибочность желательна, но не требуется.
* Уровень 2. **Канальный**. Обеспечивает безошибочную передачу блоков данных первый через уровень, который при передаче может искажать данные. Этот уровень должен определять начало и конец кадра в битовом потоке, формировать из данных, передаваемых физическим уровнем, кадры или последовательности кадров, включать процедуру проверки наличия ошибок и их исправления. Этот уровень (и только он) оперирует такими элементами, как битовые последовательности, методы кодирования, маркеры. Он несёт ответственность за правильную передачу данных (пакетов) на участках между непосредственно связанными элементами сети. Обеспечивает управление доступом к среде передачи.
* Уровень 3. **Сетевой**. Этот уровень пользуется возможностями, предоставляемыми вторым уровнем, для обеспечения связи любых двух точек в сети. Этот уровень осуществляет проводку сообщений по сети, которая может иметь много линий связи, или по множеству совместно работающих сетей, что требует маршрутизации, т.е. определения пути, по которому следует пересылать данные. Маршрутизация производится на этом же уровне. Выполняет обработку адресов, а также мультиплексирование и демультиплексирование. Основной функцией программного обеспечения на этом уровне является выборка информации из источника, преобразование её в пакеты и правильная передача в точку назначения.
* Уровень 4. **Транспортный**. Регламентирует пересылку данных между процессами, выполняемыми на компьютерах сети. Завершает организацию передачи данных: контролирует на сквозной основе поток данных, проходящий по маршруту, определённому третьим уровнем: правильность передачи блоков данных, правильность доставки в нужный пункт назначения, их комплектность, сохранность, порядок следования. Собирает информацию из блоков в её прежний вид. Или же оперирует с дейтаграммами, то есть ожидает отклика-подтверждения приёма из пункта назначения, проверяет правильность доставки и адресации, повторяет посылку дейтаграммы, если не пришёл отклик.
* Уровень 5. **Сеансовый**. Координирует взаимодействие связывающихся процессов: устанавливает связь, взаимодействует, восстанавливает аварийно оконченные сеансы. Он координирует не компьютеры и устройства, а процессы в сети, поддерживает их взаимодействие. То есть управляет сеансами связи между процессами прикладного уровня. Этот же уровень ответственен за картографию сети. Фактически он преобразовывает адреса, удобные для людей, в реальные сетевые адреса, например, в Internet это соответствует преобразованию региональных (доменных) компьютерных имён в числовые адреса глобальной, и наоборот.
* Уровень 6. **Представления данных**. Этот уровень имеет дело с синтаксисом и семантикой передаваемой информации. Здесь устанавливается взаимопонимание двух сообщающихся компьютеров относительно того, как они представляют и понимают по получении передаваемую информацию. На данном этапе решаются такие задачи, как перекодировка текстовой информации и изображений, сжатие и распаковка, поддержка сетевых файловых систем (NFS), абстрактных структур данных.
* Уровень 7. **Прикладной**. Обеспечивает интерфейс между пользователем и сетью, делает доступными для человека всевозможные услуги. На этом уровне реализуется, по крайней мере, пять прикладных служб: передача файлов, удалённый терминальный доступ, электронная передача сообщений, справочная служба и управление сетью. В конкретной реализации определяется пользователем согласно его необходимости и требованиям.

## Физические основы передачи данных. Цифровые и аналоговые сигналы.

## Система доменных имён DNS.

**DNS** (англ. Domain Name System — система доменных имён) — компьютерная распределённая система для получения информации о доменах. Основой DNS является представление об иерархической структуре доменного имени и зонах. Каждый сервер, отвечающий за имя, может делегировать ответственность за дальнейшую часть домена другому серверу, что позволяет возложить ответственность за актуальность информации на серверы различных организаций (людей), отвечающих только за «свою» часть доменного имени.

Особенности системы DNS:

* *Распределённость администрирования*Ответственность за разные части иерархической структуры несут разные люди или организации.
* *Распределённость хранения информации*Каждый узел сети в обязательном порядке должен хранить только те данные, которые входят в его зону ответственности и (возможно) адреса корневых DNS-серверов.
* *Кеширование информации*Узел может хранить некоторое количество данных не из своей зоны ответственности для уменьшения нагрузки на сеть.
* *Иерархическая структура*  
  Все узлы объединены в дерево, и каждый узел может или самостоятельно определять работу нижестоящих узлов, или делегировать (передавать) их другим узлам.
* *Резервирование*  
  За хранение и обслуживание своих узлов (зон) отвечают (обычно) несколько серверов, разделённые как физически, так и логически, что обеспечивает сохранность данных и продолжение работы даже в случае сбоя одного из узлов.

Первоначально в локальных сетях из небольшого числа компьютеров применялись плоские имена, состоящие из последовательности символов без разделения их на отдельные части, например MYCOMP. Для установления соответствия между символьными именами и числовыми адресами использовались широковещательные запросы. Однако для больших территориально распределенных сетей, работающих на основе протокола TCP/IP такой способ оказался неэффективным. Поэтому для установления соответствия между доменным именем и IP-адресом используется специальная система доменных имен (DNS, Domain Name System), которая основана на создаваемых администраторами сети таблиц соответствия.

Запись доменного имени начинается с самой младшей составляющей, затем после точки следует следующая по старшинству символьная часть имени и так далее. Последовательность заканчивается корневым именем, например: company.yandex.ru.

Построенная таким образом система имен позволяет разделять административную ответственность по поддержке уникальности имен в пределах своего уровня иерархии между различными людьми или организациями.

Корневой домен управляется центральными органами Интернета: IANA и Internic.

Домены верхнего уровня назначаются для каждой страны, а также для различных типов организаций. Имена этих доменов должны следовать международному стандарту ISO 3166. Для обозначения стран используются двухбуквенные аббревиатуры, например ru (Российская Федерация), us (США), it (Италия), fr (Франция).

Для различных типов организаций используются трехбуквенные аббревиатуры:

* net – сетевые организации;
* org – некоммерческие организации;
* com - коммерческие организации;
* edu – образовательные организации;
* gov – правительственные организации.

Администрирование каждого домена возлагается на отдельную организацию, которая делегирует администрирование поддоменов другим организациям.

Для получения доменного имени необходимо зарегистрироваться в соответствующей организации, которой организация InterNIC делегировала свои полномочия по распределению доменных имен.

Регистратором доменных имен в зоне ru до 2005 г. являлся Российский научно-исследовательский институт развития общественных сетей ( РосНИИРОС ). В настоящее время регистрация доменов осуществляется одним из действующих регистраторов.

DNS, использующая распределенную базу отображений "доменное имя – IP-адрес". Сервер домена хранит только имена, которые заканчиваются на следующем ниже по дереву уровне. Это позволяет распределять более равномерно нагрузку по разрешению имен между всеми DNS-серверами. Каждый DNS-сервер помимо таблицы отображения имен содержит ссылки на DNS-серверы своих поддоменов.

Существуют две схемы разрешения DNS-имен.

Нерекурсивная процедура:

1.DNS-клиент обращается к корневому DNS-серверу с указанием полного доменного имени;

2.DNS-сервер отвечает клиенту, указывая адрес следующего DNS-сервера, обслуживающего домен верхнего уровня, заданный в следующей старшей части имени;

3.DNS-клиент делает запрос следующего DNS-сервера, который отсылает его к DNS-серверу нужного поддомена и т.д., пока не будет найден DNS-сервер, в котором хранится соответствие запрошенного имени IP-адресу. Сервер дает окончательный ответ клиенту.

Рекурсивная процедура:

1.DNS-клиент запрашивает локальный DNS-сервер, обслуживающий поддомен, которому принадлежит клиент;

2.Далее

3.Если локальный DNS-сервер знает ответ, он возвращает его клиенту

4.Если локальный сервер не знает ответ, то он выполняет итеративные запросы к корневому серверу. После получения ответа сервер передает его клиенту.

Таким образом, при рекурсивной процедуре клиент фактически перепоручает работу своему серверу. Для ускорения поиска IP-адресов DNS-серверы широко применяют кэширование (на время от часов до нескольких дней) проходящих через них ответов.

Ключевыми понятиями DNS являются:

* **Домен** (англ. ***domain*** — область) — узел в дереве имён, вместе со всеми подчинёнными ему узлами (если таковые имеются), то есть именованная ветвь или поддерево в дереве имен. Структура доменного имени отражает порядок следования узлов в иерархии; доменное имя читается слева направо от младших доменов к доменам высшего уровня (в порядке повышения значимости), корневым доменом всей системы является точка (‘.’), ниже идут домены первого уровня (географические или тематические), затем — домены второго уровня, третьего и т. д. (например, для адреса ru.wikipedia.org домен первого уровня — org, второго wikipedia, третьего ru). На практике точку в конце имени часто опускают, но она бывает важна в случаях разделения между относительными доменами и FQDN (англ. Fully Qualifed Domain Name, полностью определённое имя домена).
* **Поддомен** (англ. ***subdomain***) — подчиненный домен. (например, wikipedia.org — поддомен домена org, а ru.wikipedia.org — домена wikipedia.org). Теоретически такое деление может достигать глубины 127 уровней, а каждая метка может содержать до 63 символов, пока общая длина вместе с точками не достигнет 254 символов. Но на практике регистраторы доменных имён используют более строгие ограничения. Например, если у вас есть домен вида mydomain.ru, вы можете создать для него различные поддомены вида mysite1.mydomain.ru, mysite2.mydomain.ru и т. д.
* **Ресурсная запись** — единица хранения и передачи информации в DNS. Каждая ресурсная запись имеет имя (то есть привязана к определенному Доменному имени, узлу в дереве имен), тип и поле данных, формат и содержание которого зависит от типа.
* **Зона** — часть дерева доменных имен (включая ресурсные записи), размещаемая как единое целое на некотором сервере доменных имен (DNS-сервере, см. ниже), а чаще — одновременно на нескольких серверах (см. ниже). Целью выделения части дерева в отдельную зону является передача ответственности (см. ниже) за соответствующий Домен другому лицу или организации, так называемое Делегирование (см. ниже). Как связная часть дерева, зона внутри тоже представляет собой дерево. Если рассматривать пространство имен DNS как структуру из зон, а не отдельных узлов/имен, тоже получается дерево; оправданно говорить о родительских и дочерних зонах, о старших и подчиненных. На практике, большинство зон 0-го и 1-го уровня (‘.’, ru, com, …) состоят из единственного узла, которому непосредственно подчиняются дочерние зоны. В больших корпоративных доменах (2-го и более уровней) иногда встречается образование дополнительных подчиненных уровней без выделения их в дочерние зоны.
* **Делегирование** — операция передачи ответственности за часть дерева доменных имен другому лицу или организации. За счет делегирования в DNS обеспечивается распределенность администрирования и хранения. Технически делегирование выражается в выделении этой части дерева в отдельную зону, и размещении этой зоны на DNS-сервере (см. ниже), управляемом этим лицом или организацией. При этом в родительскую зону включаются «склеивающие» ресурсные записи (NS и А), содержащие указатели на DNS-сервера дочерней зоны, а вся остальная информация, относящаяся к дочерней зоне, хранится уже на DNS-серверах дочерней зоны.
* **DNS-сервер** — специализированное ПО для обслуживания DNS, а также компьютер, на котором это ПО выполняется. DNS-сервер может быть ответственным за некоторые зоны и/или может перенаправлять запросы вышестоящим серверам.
* **DNS-клиент** — специализированная библиотека (или программа) для работы с DNS. В ряде случаев DNS-сервер выступает в роли DNS-клиента.
* **Авторитетность** (англ. ***authoritative***) — признак размещения зоны на DNS-сервере. Ответы DNS-сервера могут быть двух типов: авторитетные (когда сервер заявляет, что сам отвечает за зону) и неавторитетные (англ. Non-authoritative), когда сервер обрабатывает запрос, и возвращает ответ других серверов. В некоторых случаях вместо передачи запроса дальше DNS-сервер может вернуть уже известное ему (по запросам ранее) значение (режим кеширования).
* **DNS-запрос** (англ. ***DNS query***) — запрос от клиента (или сервера) серверу. Запрос может быть рекурсивным или нерекурсивным (см. Рекурсия).

Система DNS содержит иерархию DNS-серверов, соответствующую иерархии зон. Каждая зона поддерживается как минимум одним авторитетным сервером, на котором расположена информация о домене.  
Имя и IP-адрес не тождественны — один IP-адрес может иметь множество имён, что позволяет поддерживать на одном компьютере множество веб-сайтов (это называется виртуальный хостинг). Обратное тоже справедливо — одному имени может быть сопоставлено множество IP-адресов: это позволяет создавать балансировку нагрузки.  
Для повышения устойчивости системы используется множество серверов, содержащих идентичную информацию, а в протоколе есть средства, позволяющие поддерживать синхронность информации, расположенной на разных серверах. Существует 13 корневых серверов, их адреса практически не изменяются.

Протокол DNS использует для работы TCP- или UDP-порт 53 для ответов на запросы. Традиционно запросы и ответы отправляются в виде одной U.

**Записи DNS**, или **Ресурсные записи** (англ. ***Resource Records, RR***) — единицы хранения и передачи информации в DNS. Каждая ресурсная запись состоит из следующих полей:

* имя (NAME) — доменное имя, к которому привязана или которому «принадлежит» данная ресурсная запись,
* TTL (Time To Live) — допустимое время хранения данной ресурсной записи в кэше неответственного DNS-сервера,
* тип (TYPE) ресурсной записи — определяет формат и назначение данной ресурсной записи,
* класс (CLASS) ресурсной записи; теоретически считается, что DNS может использоваться не только с TCP/IP, но и с другими типами сетей, код в поле класс определяет тип сети,
* длина поля данных (RDLEN),
* поле данных (RDATA), формат и содержание которого зависит от типа записи.

Наиболее важные типы DNS-записей:

* **Запись A** (address record) или запись адреса связывает имя хоста с адресом IP. Например, запрос A-записи на имя referrals.icann.org вернет его IP адрес — 192.0.34.164
* **Запись AAAA** (IPv6 address record) связывает имя хоста с адресом протокола IPv6. Например, запрос AAAA-записи на имя K.ROOT-SERVERS.NET вернет его IPv6 адрес — 2001:7fd::1
* **Запись CNAME** (canonical name record) или каноническая запись имени (псевдоним) используется для перенаправления на другое имя
* **Запись MX** (mail exchange) или почтовый обменник указывает сервер(ы) обмена почтой для данного домена.
* **Запись NS** (name server) указывает на DNS-сервер для данного домена.
* **Запись PTR** (pointer) или запись указателя связывает IP хоста с его каноническим именем. Запрос в домене in-addr.arpa на IP хоста в reverse форме вернёт имя (FQDN) данного хоста (см. Обратный DNS-запрос). Например, (на момент написания), для IP адреса 192.0.34.164: запрос записи PTR 164.34.0.192.in-addr.arpa вернет его каноническое имя referrals.icann.org. В целях уменьшения объёма нежелательной корреспонденции (спама) многие серверы-получатели электронной почты могут проверять наличие PTR записи для хоста, с которого происходит отправка. В этом случае PTR запись для IP адреса должна соответствовать имени отправляющего почтового сервера, которым он представляется в процессе SMTP сессии.
* **Запись SOA** (Start of Authority) или начальная запись зоны указывает, на каком сервере хранится эталонная информация о данном домене, содержит контактную информацию лица, ответственного за данную зону, тайминги (параметры времени) кеширования зонной информации и взаимодействия DNS-серверов.
* **Запись SRV** (server selection) указывает на серверы для сервисов, используется, в частности, для Jabber и Active Directory.

**BIND** (***Berkeley Internet Name Domain***, до этого: Berkeley Internet Name Daemon) — открытая и наиболее распространённая реализация DNS-сервера, обеспечивающая выполнение преобразования DNS-имени в IP-адрес и наоборот.  
BIND поддерживается организацией Internet Systems Consortium. BIND был создан студентами и впервые был выпущен в BSD 4.3.  
В Unix этот сервер является стандартом де-факто.

## Типы DNS-серверов

По выполняемым функциям DNS-серверы делятся на несколько групп, в зависимости от конфигурации конкретный сервер может относиться к нескольким типам;

* **авторитативный DNS-сервер** — сервер, отвечающий за какую-либо зону. – **Мастер** или **первичный сервер** (в терминологии BIND) — сервер, имеющий право на внесение изменений в данные зоны. Обычно для зоны бывает только один мастер сервер. В случае Microsoft DNS-сервера и его интеграции с Active Directory мастер-серверов может быть несколько (так как репликация изменений осуществляется не средствами DNS-сервера, а средствами Active Directory, за счёт чего обеспечивается равноправность серверов и актуальность данных). – **Слейв** или **вторичный сервер**, не имеющий права на внесение изменений в данные зоны и получающий сообщения об изменениях от мастер-сервера. В отличие от мастер-сервера их может быть (практически) неограниченное количество. Слейв так же является авторитативным сервером (и пользователь не может различить мастер и слейв, разница появляется только на этапе конфигурирования/внесения изменений в настройки зоны).
* **Кэширующий DNS-сервер** — сервер, который обслуживает запросы клиентов, (получает рекурсивный запрос, выполняет его с помощью нерекурсивных запросов к авторитативным серверам или передаёт рекурсивный запрос вышестоящему DNS-серверу)
* **Локальный DNS-сервер**; используется для обслуживания DNS-клиентов, исполняющихся на локальной машине. Фактически, это разновидность кэширующего DNS-сервера, сконфигурированная для обслуживания локальных приложений.
* **Перенаправляющий DNS-сервер**; (англ. forwarder, внутренний DNS-сервер) сервер, перенаправляющий полученные рекурсивные запросы вышестоящему кэширующему серверу в виде рекурсивных запросов. Используется преимущественно для снижения нагрузки на кэширующий DNS-сервер.
* **Корневой DNS-сервер** — сервер, являющийся авторитативным за корневую зону. Общеупотребительных корневых серверов в мире всего 13 штук, их доменные имена находятся в зоне root-servers.net и называются a.root-servers.net, b.root-servers.net, … , m.root-servers.net. В определённых конфигурациях локальной сети возможна ситуация настройки локальных корневых серверов.
* **Регистрирующий DNS-сервер**. Сервер, принимающий динамические обновления от пользователей. Часто совмещается с DHCP-сервером. В Microsoft DNS-сервере при работе на контроллере домена сервер работает в режиме регистрирующего DNS-сервера, принимая от компьютеров домена информацию о соответствии имени и IP компьютера и обновляя в соответствии с ней данные зоны домена.

**Прямой запрос**

Прямой (forward) запрос — запрос на преобразование имени (символьного адреса) хоста в IP-адрес.

**Обратный запрос**

Обратный (reverse) запрос — запрос на преобразование IP-адреса в имя хоста.

## Понятие протокола, стека протоколов.

Протокол передачи данных — набор соглашений интерфейса логического уровня, которые определяют обмен данными между различными программами. Эти соглашения задают единообразный способ передачи сообщений и обработки ошибок при взаимодействии программного обеспечения разнесённой в пространстве аппаратуры, соединённой тем или иным интерфейсом.

Любая связь между устройствами возможна лишь благодаря протоколам. Они делятся на физические протоколы (регулируют то, как именно и какие сигналы будут идти от одного устройства к другому — например, импульсами по 5 вольт 100 раз в секунду или на определённой частоте радиосигналов) и логические протоколы, которые отвечают за смысл и передачу данных, когда связь уже установлена. Так, браузер на компьютере связывается с сервером по протоколу HTTP или HTTPS.

Сетево́й протоко́л — набор правил и действий (очерёдности действий), позволяющий осуществлять соединение и обмен данными между двумя и более включёнными в сеть устройствами.

Стек протоколов — это иерархически организованный набор сетевых протоколов, достаточный для организации взаимодействия узлов в сети. Протоколы работают в сети одновременно, значит работа протоколов должна быть организована так, чтобы не возникало конфликтов или незавершённых операций. Поэтому стек протоколов разбивается на иерархически построенные уровни, каждый из которых выполняет конкретную задачу — подготовку, приём, передачу данных и последующие действия с ними.

Количество уровней в стеке меняется в соответствии с конкретным стеком протоколов. Протоколы нижних уровней часто реализуются комбинацией программных и аппаратных средств, а протоколы верхних уровней, как правило, программными средствами.

Существует достаточное количество стеков протоколов, широко применяемых в сетях. Наиболее популярные стеки протоколов: OSI международной организации по стандартизации, TCP/IP, используемый в сети Internet и во многих сетях на основе операционной системы UNIX, IPX/SPX фирмы Novell, NetBIOS/SMB, разработанный фирмами Microsoft и IBM, DECnet корпорации Digital Equipment, SNA фирмы IBM и некоторые другие.

Сетевые протоколы предписывают правила работы компьютерам, которые подключены к сети. Они строятся по многоуровневому принципу. Протокол некоторого уровня определяет одно из технических правил связи. В настоящее время для сетевых протоколов используется модель [OSI](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%82%D0%B5%D0%B2%D0%B0%D1%8F_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_OSI) (Open System Interconnection — взаимодействие открытых систем, ВОС).

Модель OSI — 7-уровневая логическая модель работы сети. Реализуется группой протоколов и правил связи, организованных в несколько уровней:

* на физическом уровне определяются физические (механические, электрические, оптические) характеристики линий связи;
* на канальном уровне определяются правила использования физического уровня узлами сети;
* сетевой уровень отвечает за адресацию и доставку сообщений;
* транспортный уровень контролирует очерёдность прохождения компонентов сообщения;
* сеансовый уровень координирует связь между двумя прикладными программами, работающими на разных рабочих станциях;
* уровень представления служит для преобразования данных из внутреннего формата компьютера в формат передачи;
* прикладной уровень является пограничным между прикладной программой и другими уровнями, обеспечивая удобный интерфейс связи для сетевых программ пользователя.

Другая модель — [стек протоколов TCP/IP](https://ru.wikipedia.org/wiki/TCP/IP) — содержит 4 уровня:

* канальный уровень (link layer),
* сетевой уровень (Internet layer),
* транспортный уровень (transport layer),
* прикладной уровень (application layer).

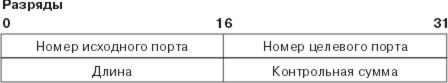
## Протокол UDP.

Протокол **UDP** (User Datagram Protocol, RFC-768) является одним из основных протоколов, расположенных непосредственно над IP. Он предоставляет прикладным процессам транспортные услуги, немногим отличающиеся от услуг протокола IP. Протокол UDP обеспечивает доставку дейтограмм, но не требует подтверждения их получения. Протокол UDP не требует соединения с удаленным модулем UDP ("бессвязный" протокол). К заголовку IP-пакета UDP добавляет поля *порт отправителя* и *порт получателя*, которые обеспечивают мультиплексирование информации между различными прикладными процессами, а также поля *длина* UDP-дейтограммы и *контрольная сумма*, позволяющие поддерживать целостность данных. Таким образом, если на уровне IP для определения места доставки пакета используется адрес, на уровне UDP - номер порта.

Хотя протокол UDP не гарантирует доставки, по умолчанию предполагается, что вероятность потери пакета достаточно мала.

Иногда возникает необходимость отправить сообщение от одного приложения другому приложению или процессу, выполняемому на другом компьютере, подключенном к сети. **UDP** обеспечивает передачу дейтаграмм между приложениями хостов Internet.

Для отправки дейтаграмм протокол **UDP** применяет протокол **IP**, поэтому **UDP** так же не устанавливает соединения, как и **IP**. Он не гарантирует доставку дейтаграммы и не обеспечивает защиту от дублирования данных. Однако **UDP** позволяет отправителю задать для сообщения исходный и целевой порты и обеспечивает проверку целостности данных и заголовка сообщения с помощью контрольной суммы. Это позволяет отправителю и получателю проверить правильность доставки сообщения.



*Заголовок пакета в протоколе UDP*

На рисунке показаны первые 32 бита стандартного заголовка пакета UDP. Первые 16 бит содержат номер исходного порта и длину. Вторые 16 бит содержат номер целевого порта и контрольную сумму.

Длина сообщения равна числу байт в UDP-дейтограмме, включая заголовок. Поле UDP контрольная сумма содержит код, полученный в результате контрольного суммирования UDP-заголовка и поля данные. Не трудно видеть, что этот протокол использует заголовок минимального размера (8 байт). Таблица номеров UDP-портов приведена ниже (4.4.2.1). Номера портов от 0 до 255 стандартизованы и использовать их в прикладных задачах не рекомендуется. Но и в интервале 255-1023 многие номера портов заняты, поэтому прежде чем использовать какой-то порт в своей программе, следует заглянуть в RFC-1700. Во второй колонке содержится стандартное имя, принятое в Internet, а в третей - записаны имена, принятые в UNIX.

Для надежной доставки дейтаграмм с помощью UDP в приложении должны быть предусмотрены процедуры проверки. Для надежной доставки потоков данных предназначен протокол TCP.

UDP решает проблему задержек в передаче данных, позволяя информации перемещаться молниеносно, даже если это означает возможность потери некоторых данных по пути. Это ключевой момент для приложений, где каждая секунда на счету, например, в онлайн-играх или видеоконференциях.

**Схема вычисления контрольных сумм**

Модуль IP передает поступающий IP-пакет модулю UDP, если в заголовке этого пакета указан код протокола UDP. Когда модуль UDP получает дейтограмму от модуля IP, он проверяет контрольную сумму, содержащуюся в ее заголовке. Если контрольная сумма равна нулю, это означает, что отправитель ее не подсчитал. ICMP, IGMP, UDP и TCP протоколы имеют один и тот же алгоритм вычисления контрольной суммы (RFC-1071). Но вычисление контрольной суммы для UDP имеет некоторые особенности. Во-первых, длина UDP-дейтограммы может содержать нечетное число байт, в этом случае к ней добавляется нулевой байт, который служит лишь для унификации алгоритма и никуда не пересылается. Во-вторых, при расчете контрольной суммы для UDP и TCP добавляются 12-байтные псевдо-заголовки, содержащие IP-адреса отправителя и получателя, код протокола и длину дейтограммы (см. рис. 4.4.2.2). Как и в случае IP-дейтограммы, если вычисленная контрольная сумма равна нулю, в соответствующее поле будет записан код 65535.

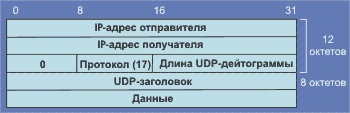


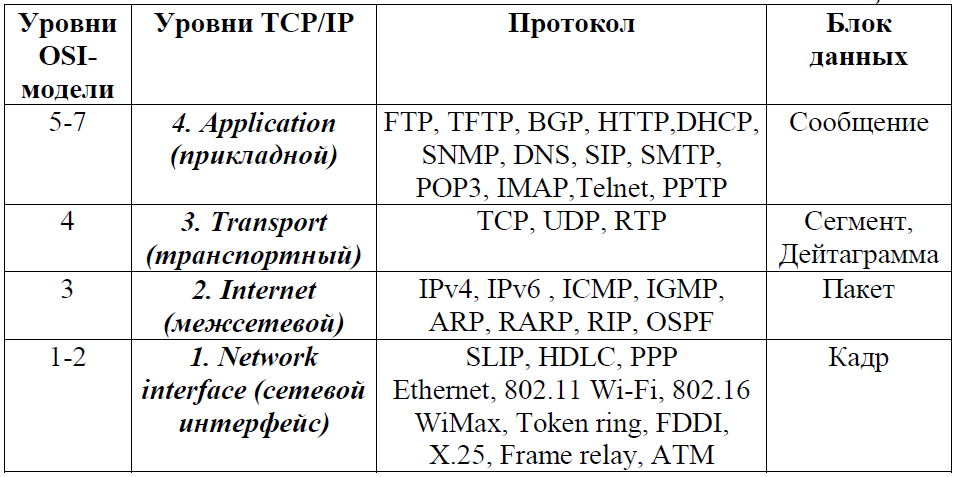
Рис. 4.4.2.2. Псевдозаголовок, используемый при расчете контрольной суммы

Если контрольная сумма правильная (или равна 0), то проверяется порт назначения, указанный в заголовке дейтограммы. Если прикладной процесс подключен к этому порту, то прикладное сообщение, содержащиеся в дейтограмме, становится в очередь к прикладному процессу для прочтения. В остальных случаях дейтограмма отбрасывается. Если дейтограммы поступают быстрее, чем их успевает обрабатывать прикладной процесс, то при переполнении очереди сообщений поступающие дейтограммы отбрасываются модулем UDP. Следует учитывать, что во многих посылках контрольное суммирование не охватывает адреса отправителя и места назначения. При некоторых схемах маршрутизации это приводит к зацикливанию пакетов в случае повреждения его адресной части (адресат не признает его "своим").

Может возникнуть вопрос, зачем вычислять и проверять контрольную сумму, если подтверждение доставки и повторная пересылка в рамках протокола не предусмотрены. Дело в том, что UDP используется не только для мультимедийных задач но и некоторыми другими протоколами (DNS, SNMP и др.), где повторные запросы и отклики могут выполняться на прикладном уровне.

Так как максимальная длина IP-дейтограммы равна 65535 байтам, максимальная протяженность информационного поля UDP-дейтограммы составляет 65507 байт. На практике большинство систем работает с UDP-дейтограммами с длиной 8192 байта или менее (Ethernet допускает 1508 байт). Детальное описание форматов, полей пакетов и пр. читатель может найти в RFC-768. Смотри также RFC-2147 (IPv6 Jumbo), RFC-2508 (компрессия заголовков) и RFC-3828 (Lightweight UDP).

## Стек сетевых протоколов TCP/IP.



Стек протоколов TCP/IP (Transmission Control Protocol/Internet Protocol, протокол управления передачей/протокол интернета) — сетевая модель, описывающая процесс передачи цифровых данных. Она названа по двум главным протоколам, по этой модели построена глобальная сеть интернет.

**Канальный уровень (link layer)**

Предназначение канального уровня — дать описание тому, как происходит обмен информацией на уровне сетевых устройств, определить, как информация будет передаваться от одного устройства к другому. Информация здесь кодируется, делится на пакеты и отправляется по нужному каналу, т.е. среде передачи.

Этот уровень также вычисляет максимальное расстояние, на которое пакеты возможно передать, частоту сигнала, задержку ответа и т.д. Все это — физические свойства среды передачи информации. На канальном уровне самым распространенным протоколом является Ethernet, который мы рассмотрим в конце статьи.

**Межсетевой уровень (internet layer)**

Глобальная сеть интернет состоит из множества локальных сетей, взаимодействующих между собой. Межсетевой уровень используется, чтобы описать обеспечение такого взаимодействия.

Межсетевое взаимодействие — это основной принцип построения интернета. Локальные сети по всему миру объединены в глобальную, а передачу данных между этими сетями осуществляют магистральные и пограничные маршрутизаторы.

Именно на межсетевом уровне функционирует протокол IP, позволивший объединить разные сети в глобальную. Как и протокол TCP, он дал название модели, рассматриваемой в статье.

**Транспортный уровень (transport layer)**

Постоянные резиденты транспортного уровня — протоколы TCP и UDP, они занимаются доставкой информации.

**TCP (протокол управления передачей)** — надежный, он обеспечивает передачу информации, проверяя дошла ли она, насколько полным является объем полученной информации и т.д. TCP дает возможность двум конечным устройствам производить обмен пакетами через предварительно установленное соединение. Он предоставляет услугу для приложений, повторно запрашивает потерянную информацию, устраняет дублирующие пакеты, регулируя загруженность сети. TCP гарантирует получение и сборку информации у адресата в правильном порядке.

**UDP (протокол пользовательских датаграмм)** — ненадежный, он занимается передачей автономных датаграмм. UDP не гарантирует, что всех датаграммы дойдут до получателя. Датаграммы уже содержат всю необходимую информацию, чтобы дойти до получателя, но они все равно могут быть потеряны или доставлены в порядке отличном от порядка при отправлении.

UDP обычно не используется, если требуется надежная передача информации. Использовать UDP имеет смысл там, где потеря части информации не будет критичной для приложения, например, в видеоиграх или потоковой передаче видео. UDP необходим, когда делать повторный запрос сложно или неоправданно по каким-то причинам.

Протоколы транспортного уровня не интерпретируют информацию, полученную с верхнего или нижних уровней, они служат только как канал передачи, но есть исключения. RSVP (Resource Reservation Protocol, протокол резервирования сетевых ресурсов) может использоваться, например, роутерами или сетевыми экранами в целях анализа трафика и принятия решений о его передаче или отклонении в зависимости от содержимого.

**Прикладной уровень (application layer)**

В модели TCP/IP отсутствуют дополнительные промежуточные уровни (представления и сеансовый) в отличие от OSI. Функции форматирования и представления данных делегированы библиотекам и программным интерфейсам приложений (API) — своего рода базам знаний, содержащим сведения о том, как приложения взаимодействуют между собой. Когда службы или приложения обращаются к библиотеке или API, те в ответ предоставляют набор действий, необходимых для выполнения задачи и полную инструкцию, каким образом эти действия нужно выполнять.

Протоколы прикладного уровня действуют для большинства приложений, они предоставляют услуги пользователю или обмениваются данными с «коллегами» с нижних уровней по уже установленным соединениям. Здесь для большинства приложений созданы свои протоколы. Например, браузеры используют HTTP для передачи гипертекста по сети, почтовые клиенты — SMTP для передачи почты, FTP-клиенты — протокол FTP для передачи файлов, службы DHCP — протокол назначения IP-адресов DHCP и так далее.

## Сетевые устройства: повторитель, мост, коммутатор, маршрутизатор, концентратор, шлюз.

**Коммутатор в компьютерных сетях** — применяется для соединения нескольких узлов компьютерной сети. Компьютерная сеть (вычислительная сеть, сеть передачи данных) — система связи компьютеров и/или компьютерного оборудования (серверы, маршрутизаторы и другое оборудование). Для передачи информации могут быть использованы различные физические явления, как правило — различные виды электрических сигналов, световых сигналов или электромагнитного излучения.

**PAN (**Personal Area Network) — персональная сеть, предназначенная для взаимодействия различных устройств, принадлежащих одному владельцу.

**LAN** (Local Area Network) — локальные сети, имеющие замкнутую инфраструктуру до выхода на поставщиков услуг. Термин «LAN» может описывать и маленькую офисную сеть, и сеть уровня большого завода, занимающего несколько сотен гектаров. Зарубежные источники дают даже близкую оценку — около шести миль (10 км) в радиусе. Локальные сети являются сетями закрытого типа, доступ к ним разрешен только ограниченному кругу пользователей, для которых работа в такой сети непосредственно связана с их профессиональной деятельностью.

**CAN** (Campus Area Network — кампусная сеть) — объединяет локальные сети близко расположенных зданий.

**MAN** (Metropolitan Area Network) — городские сети между учреждениями в пределах одного или нескольких городов, связывающие много локальных вычислительных сетей.

**WAN** (Wide Area Network) — глобальная сеть, покрывающая большие географические регионы, включающие в себя как локальные сети, так и прочие телекоммуникационные сети и устройства. Пример WAN — сети с коммутацией пакетов (Frame relay), через которую могут «разговаривать» между собой различные компьютерные сети. Глобальные сети являются открытыми и ориентированы на обслуживание любых пользователей.

**По типу функционального взаимодействия**Клиент-сервер, Смешанная сеть, Одноранговая сеть, Многоранговые сети

**По типу сетевой топологии**Шина, Кольцо, Двойное кольцо, Звезда, Решётка, Дерево

**По типу среды передачи**Проводные (телефонный провод, коаксиальный кабель, витая пара, волоконно-оптический кабель)

Беспроводные (передачей информации по радиоволнам в определенном частотном диапазоне)

**По функциональному назначению**Сети хранения данных, Серверные фермы, Сети управления процессом, Сети SOHO, домовые сети,

**По скорости передач**низкоскоростные (до 10 Мбит/с),среднескоростные (до 100 Мбит/с),высокоскоростные (свыше 100 Мбит/с);

**Сетевой концентратор или хаб** (жарг. от англ. hub — центр деятельности) — сетевое устройство, предназначенное для объединения нескольких устройств Ethernet в общий сегмент сети. Устройства подключаются при помощи витой пары, коаксиального кабеля или оптоволокна. Термин концентратор (хаб) применим также к другим технологиям передачи данных: USB, FireWire и пр.

**Принцип работы**Концентратор работает на физическом уровне сетевой модели OSI, повторяет приходящий на один порт сигнал на все активные порты. В случае поступления сигнала на два и более порта одновременно возникает коллизия, и передаваемые кадры данных теряются. Таким образом, все подключённые к концентратору устройства находятся в одном домене коллизий. Концентраторы всегда работают в режиме полудуплекса, все подключённые устройства Ethernet разделяют между собой предоставляемую полосу доступа.

**Упрощённое описание принципа работы**Хаб работает по следующему принципу: копирует все полученные пакеты во все порты. При этом может возникнуть проблема, при которой по двум и более портам приходят пакеты в одно и то же время. Другая проблема — безопасность — все пакеты доходят до всех компьютеров сети, поэтому существует возможность несанкционированного доступа к информации. И, наконец, ещё одной проблемой является то, что копирование пакетов повышает нагрузку на сеть, причём весьма существенно — весь трафик сегмента сети поступает к каждому из компьютеров и тем самым загружает сеть.

**Повторитель**(репи́тер, от англ. repeater) — сетевое оборудование, Предназначен для увеличения расстояния сетевого соединения путём повторения электрического сигнала «один в один». Бывают однопортовые повторители и многопортовые. В терминах модели OSI работает на физическом уровне.

**Сетевой шлюз** (англ. gateway) — аппаратный маршрутизатор или программное обеспечение для сопряжения компьютерных сетей, использующих разные протоколы (например, локальной и глобальной). Сетевой шлюз конвертирует протоколы одного типа физической среды в протоколы другой физической среды (сети). Например, при соединении локального компьютера с сетью Интернет вы используете сетевой шлюз.

**Роутеры (маршрутизаторы**) являются одним из примеров аппаратных сетевых шлюзов.Сетевые шлюзы работают на всех известных операционных системах. Основная задача сетевого шлюза — конвертировать протокол между сетями. Роутер сам по себе принимает, проводит и отправляет пакеты только среди сетей, использующих одинаковые протоколы. Сетевой шлюз может с одной стороны принять пакет, сформатированный под один протокол (например Apple Talk) и конвертировать в пакет другого протокола (например TCP/IP) перед отправкой в другой сегмент сети. Сетевые шлюзы могут быть аппаратным решением, программным обеспечением или тем и другим вместе, но обычно это программное обеспечение, установленное на роутер или компьютер. Сетевой шлюз должен понимать все протоколы, используемые роутером. Обычно сетевые шлюзы работают медленнее, чем сетевые мосты и коммутаторы**.**

**Сетевой шлюз** — это точка сети, которая служит выходом в другую сеть. В сети Интернет узлом или конечной точкой может быть или сетевой шлюз, или хост Сетевой шлюз может быть специальным аппаратным роутером или программным обеспечением, установленным на обычный сервер или персональный компьютер. Большинство компьютерных операционных систем использует термины, описанные выше.

**Маршрутиза́тор**или ро́утер — специализированный сетевой компьютер, имеющий минимум два сетевых интерфейса и пересылающий пакеты данных между различными сегментами сети, принимающий решения о пересылке на основании информации о топологии сети и определённых правил, заданных администратором. Маршрутизаторы делятся на программные и аппаратные. Маршрутизатор работает на более высоком «сетевом» уровне 3 сетевой модели OSI, нежели коммутатор и сетевой мост.Обычно маршрутизатор использует адрес получателя, указанный в пакетах данных, и определяет по таблице маршрутизации путь, по которому следует передать данные. Если в таблице маршрутизации для адреса нет описанного маршрута, пакет отбрасывается.Существуют и другие способы определения маршрута пересылки пакетов, когда, например, используется адрес отправителя, используемые протоколы верхних уровней и другая информация, содержащаяся в заголовках пакетов сетевого уровня. Нередко маршрутизаторы могут осуществлять трансляцию адресов отправителя и получателя, фильтрацию транзитного потока данных на основе определённых правил с целью ограничения доступа, шифрование/дешифрование передаваемых данных и т. д.

**Мост, сетевой мост, бридж**  — сетевое устройство 2 уровня модели OSI, предназначенное для объединения сегментов (подсети) компьютерной сети разных топологий и архитектур.

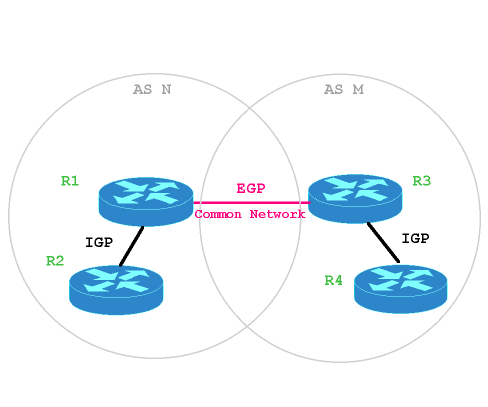
**Различия между коммутаторами и мостами**В общем случае коммутатор (свитч) и мост аналогичны по функциональности; разница заключается во внутреннем устройстве: мосты обрабатывают трафик, используя центральный процессор, коммутатор же использует коммутационную матрицу (аппаратную схему для коммутации пакетов). В настоящее время мосты практически не используются (так как для работы требуют производительный процессор), за исключением ситуаций, когда связываются сегменты сети с разной организацией первого уровня, например, между xDSL соединениями, оптикой, Ethernet’ом. В случае SOHO-оборудования, режим прозрачной коммутации часто называют «мостовым режимом» (bridging).

**Функциональные возможности**Мост обеспечивает: ограничение домена коллизий, задержку фреймов, адресованных узлу в сегменте отправителя, ограничение перехода из домена в домен ошибочных фреймов: ,карликов (фреймов меньшей длины, чем допускается по стандарту (64 байта)), фреймов с ошибками в CRC, фреймов с признаком «коллизия» ,затянувшихся фреймов (размером больше, чем разрешено стандартом)Мосты «изучают» характер расположения сегментов сети путем построения адресных таблиц вида «Интерфейс:MAC-адрес», в которых содержатся адреса всех сетевых устройств и сегментов, необходимых для получения доступа к данному устройству.

## Внешняя и внутренняя маршрутизация.

Протоколы внешней маршрутизации

Протоколы этого типа используются для определения маршрутов передачи данных между различными автономными системами. Такие протоколы обычно относят к классу Exterior Gateway Protocol. В настоящее время существуют два протокола данного типа:



Border Gateway Protocol

Exterior Gateway Protocol

Особенности внешней маршрутизации

Два маршрутизатора, которые обмениваются информацией о маршрутах, называются внутренними соседями в том случае, если они принадлежат к одной автономной системе и внешними – в том случае, если они принадлежат к различным автономным системам. На рисунке маршрутизаторы R2 R4 являются внутренними для автономных систем AS N и AS M соответственно. R1 и R3 совмещают функции внешнего и внутреннего маршрутизаторов. Маршрутизатор R1 представляет для автономной системы AS M маршруты к сетям, которые находятся в автономной системе AS N. Аналогичную функцию выполняет маршрутизатор R3 по отношению к маршрутам AS M.

Основная особенность протоколов внешней маршрутизации заключается в том, что они представляют метрики маршрутов, которые рассчитываются относительно некоторой общей сети, а не относительно своих интерфейсов.

Протоколы внутренней маршрутизации, используемые для

маршрутизации внутри автономной системы. Данный тип маршрутизации также

называют внутренней маршрутизацией автономной системы. Компании, организации

и даже операторы связи используют протоколы внутренней маршрутизации в своих

внутренних сетях. К протоколам внутренней маршрутизации относятся протоколы

RIP, EIGRP, OSPF и IS-IS.Существует четыре дистанционно-векторных протокола внутренней маршрутизации IPv4: • RIPv1 — устаревшая версия протокола первого поколения; • RIPv2 — простой дистанционно-векторный протокол; • IGRP — запатентованный протокол Cisco первого поколения (на сегодняшний день также устаревший, заменен протоколом EIGRP); • EIGRP — расширенная версия дистанционно-векторного протокола.

Протоколы внешнего шлюза (EGP), используемые для маршрутизации

между автономными системами. Маршрутизацию данного типа также называют

внешней маршрутизацией автономной системы. Взаимодействие между сетями

операторов связи и крупных компаний может осуществляться посредством протокола

внешней маршрутизации. Протокол пограничного шлюза (BGP) является

единственным в настоящее время официальным протоколом маршрутизации EGP,

используемым в Интернете

# Теория языков программирования и методы трансляции

## Формы Бэкуса-Наура.

Для определения языка требуется задать множество основных символов языка и описать его синтаксис (грамматику) и семантику. *Синтаксис* языка определяет правила составления корректных цепочек, состоящих из основных символов языка. *Семантика* задает смысловые значения конструкций языка, а также интерпретацию различных синтаксических конструкций [человеком].

Разработка ЯП начинается с определения его синтаксиса. Язык, предназначенный для описания другого языка, называется **метаязыком**. Метаязык задает систему обозначений, понятий языка и образованных из них конструкций, позволяющих представить описываемый язык с помощью определенных ранее понятий и отношений между ними. При этом каждое понятие языка подразумевает некоторую синтаксическую единицу (конструкцию) и определяемые ею свойства программных объектов или процесса обработки данных. Метаязыки используются для задания грамматики ЯП. При описании метаязыка используют два важных понятия.

**Терминал (терминальный символ)** – объект, непосредственно присутствующий в словах языка, соответствующего грамматике, и имеющий конкретное, неизменяемое значение (обобщение понятия «буквы»). В ЯП в качестве терминалов обычно берут все или часть стандартных символов ASCII – латинские буквы, цифры и специальные символы.

**Нетерминал (нетерминальный символ)** – объект, обозначающий какую-либо *сущность* языка (например: формула, арифметическое выражение, команда) и не имеющий конкретного символьного значения. Это можно трактовать как название структурной единицы.

Обычно терминалы обозначают маленькими буквами, а нетерминалы – большими.

Одним из первых появился *метаязык Хомского,* который вышел из математической логики и имеет такую систему обозначений:

* *символ* ***→*** отделяет левую часть правила от правой; обозначает *порождает* или *это есть*;
* ***нетерминалы*** обозначаются буквой *А* с индексом, указывающим на его номер;
* ***терминалы*** – это символы, используемые в описываемом языке;
* каждое правило определяет порождение одной новой цепочки;
* один и тот же нетерминал может встречаться в нескольких правилах слева.

В качестве примера предлагается рассмотреть описание идентификатора условного ЯП на этом метаязыке. Оно состоит из 65 правил:

*1. A1 → A* *2. A1 → B …* *26. A1 → Z*

*27. A1 → a* *28. A1 → b …* *52. A1 → z*

*53. A2 → 0* *54. A2 → 1 …* *62. A2 → 9*

*63. A3 → A1* *64. A3 → A3A1* *65. A3 → A3A2*

Данное описание идентификатора показывает громоздкость метаязыка, поэтому его можно эффективно использовать только для описания небольших абстрактных языков. Для более компактного описания позже был предложен

*метаязык Хомского-Шутценбергера*. Но наибольшее распространение для описания синтаксиса ЯП получил язык металингвистических формул Бэкуса и его модификации.

Форма Бэкуса-Наура (БНФ)впервые использовалась для описания синтаксиса ЯП Алгол 60. Наряду с новыми обозначениями метасимволов, в нем использовались содержательные обозначения нетерминалов, что сделало описание языка нагляднее и позволило применять данную нотацию для описания реальных ЯП.

В БНФ каждое определяемое понятие – это металингвистическая переменная (МЛП)*.* Ее значением может быть любая синтаксическая конструкция из некоторого фиксированного для этого понятия набора конструкций. Каждая металингвистическая форма (МЛФ) определяет одну МЛП и состоит из двух частей – левой и правой.

В левой части записывается определяемая МЛП, которая заключается в угловые скобки '<' и '>' (предполагается, что эти скобки являются метасимволами и не принадлежат алфавиту определяемого языка), например: <двоичное число>, <метка>, <арифметическое выражение>.

В правой части формы записываются все варианты определения конструкции (или *правила*), задаваемой этой формой. Каждый вариант представляет собой цепочку основных символов определяемого языка и МЛП. Варианты (правила) разделяются металингвистической связкой '|', имеющей смысл "или". Левая и правая части формы разделяются метасимволом '::=', означающим "по определению есть".

Ниже даётся описание идентификатора условного ЯП с использованием БНФ (каждое правило можно также записать и отдельно).

1. <буква> :: = А|В|С|D|E|…|W|X|Y|Z|a|b|c|d|e|…|w|x|y|z
2. <цифра> :: = 0|1|2|3|4|5|6|7|8|9
3. <идентификатор> ::= <буква>|<идентификатор><буква>|<идентификатор><цифра>

Характерной особенностью многих МЛФ является наличие в них *рекурсии.* Рекурсия имеет место в том случае, когда для определения конструкций ЯП используются МЛП, обозначающие саму определяемую конструкцию. Рекурсия может быть явной или неявной. *Явная рекурсия* используется, например, в определении МЛП «идентификатор». *Неявная рекурсия* имеет место в случае, когда МЛП, обозначающая какую-либо синтаксическую конструкцию, используется на некотором промежуточном шаге определения этой конструкции.

Наличие рекурсивных определений затрудняет чтение и понимание БНФ, хотя и является наиболее удобным способом описания бесконечных языков с помощью конечного числа правил. На практике для описания синтаксиса ЯП часто используют расширения БНФ, позволяющие более естественно представлять альтернативные, необязательные и повторяющиеся части МЛФ. Так, одно из расширений БНФ (РБНФ) разрешает использовать следующие упрощения:

* необязательные элементы СК заключаются в квадратные скобки;
* альтернативные варианты могут в случае необходимости заключаться в круглые скобки для образования многовариантного выбора;
* элементы СК, повторяющиеся нуль и более раз, заключаются в фигурные скобки '{' и '}'.

Любая синтаксическая конструкция, полученная с помощью РБНФ, может быть получена с помощью БНФ и наоборот.

## Классификация грамматик по Хомскому.

Грамматика – это описание способа построения предложений некоторого языка. Определив грамматику языка, мы указываем правила порождения цепочек символов, принадлежащих этому языку. Таким образом, грамматика – это генератор цепочек языка. Она относится ко второму способу определения языков – порождению цепочек символов.

По иерархии Хомского, грамматики делятся на **4 типа**, где каждый последующий тип является более ограниченным подмножеством предыдущего (но и легче поддающимся анализу):

* тип 0 – неограниченные грамматики – возможны любые правила;
* тип 1 – контекстно-зависимые грамматики – левая часть может содержать один нетерминал, окруженный «контекстом» (цепочки символов, в том же виде присутствующие в правой части); сам нетерминал заменяется непустой последовательностью символов в правой части;
* тип 2 – контекстно-свободные грамматики (КСГ) – левая часть состоит из одного нетерминала;
* тип 3 – регулярные грамматики – более простые, эквивалентные конечным автоматам.

Примером использования грамматики нулевого типа является естественный язык или **машина Тьюринга**. КСГ широко применяются для определения грамматической структуры в грамматическом анализе. Регулярные грамматики (в виде регулярных выражений) применяются как шаблоны для текстового поиска, разбивки и подстановки (в том числе – в лексическом анализе).

Грамматика ЯП содержит правила двух типов: первые определяют синтаксические конструкции ЯП (их довольно легко можно формально описать), вторые – семантические ограничения ЯП (обычно излагаются в неформальной форме). Поэтому любое описание (стандарт) ЯП обычно состоит из двух частей: вначале формально излагаются правила построения синтаксических конструкций, а потом на естественном языке дается описание семантических правил. Во многих компиляторах имеется специальная часть – семантический анализатор, проверяющий правильность программы.

## Методика построения автоматов. 4 типа автоматов.

**Абстрактный автомат** – математическая абстракция, модель дискретного устройства, имеющего один вход, один выход и в каждый момент времени находящегося в одном состоянии из множества возможных. На вход этому устройству поступают символы одного алфавита, на выходе оно выдаёт символы другого (в общем случае) алфавита.

Формально абстрактный автомат определяется как A=(S,X,Y,δ,λ) , где

S – конечное множество состояний автомата,

X, Y – конечные входной и выходной алфавиты соответственно, из которых формируются строки, считываемые и выдаваемые автоматом,

δ : S ´ X →S – функция переходов,

l : S ´ X →Y – функция выходов.

**Конечный автомат** (КА) – абстрактный автомат, число возможных внутренних состояний которого конечно.

Теория автоматов лежит в основе цифровых технологий и программ. Компьютер является частным случаем практической реализации КА. Часть математического аппарата теории автоматов напрямую применяется при разработке ЛА и парсеров для ФЯ, в том числе ЯП, а также при построении компиляторов и разработке самих ЯП.

Существуют различные способы описания КА (задания алгоритма его функционирования). Например, КА может быть задан в виде упорядоченной пятерки элементов некоторых множеств: M = (V,Q,q0,F,δ) , где

V – входной алфавит (конечное множество входных символов), из которого формируются входные слова, воспринимаемые КА;

Q – множество внутренних состояний КА;

q0 – начальное состояние КА (q0⸦Q);

F – множество заключительных (конечных) состояний КА (F⸦Q);

δ – функция переходов, определенная как *δ(q,a)*; значение функции переходов на упорядоченной паре (*q* – состояние, *a* – входной символ или пустая цепочка) есть множество всех состояний, в которые из данного состояния возможен переход по данному входному символу или пустой цепочке.

Конечные автоматы подразделяются на детерминированные (ДКА) и недетерминированные (НКА).

**Детерминированный конечный автомат** – последовательность (кортеж) из пяти элементов (Q,Σ,δ,S0,F), где:

Q – множество состояний автомата

Σ – алфавит языка, который понимает автомат

δ – функция перехода, такая что δ: Q ´ Σ→Q

S0∈Q – начальное состояние

F∈ Q – множество конечных состояний.

**Недетерминированный конечный автомат** – последовательность (кортеж) из пяти элементов (Q,Σ,∆,S,F), где:

Q – множество состояний автомата

Σ – алфавит языка, который понимает автомат

∆ – отношение перехода, ∆={<*q,a,p*>:*q,p*∈ Q, *a*∈Σ ∪{ε}} , где {ε}- пустое слово. То есть, НКА может совершить скачок из состояния *q* в состояние *p*, в отличие от ДКА, через пустое слово, а также перейти из *q* по *a* в несколько состояний (что опять же в ДКА невозможно)

S Í Q – множество начальных состояний

F Í Q – множество конечных состояний.

НКА является обобщением ДКА.

**Автомат с магазинной памятью** (МПА) – это односторонний распознаватель, в потенциально бесконечной памяти которого элементы информации хранятся и используются так же, как и патроны в автомате – в каждый момент доступен только верхний элемент. МПА являются естественной моделью синтаксического анализатора контекстно-свободных языков*.*

Задать МПА – значит описать алгоритм его работы. Автомат считается заданным, если известны:

* алфавит входных сигналов;
* алфавит выходных сигналов;
* алфавит состояний;
* начальное состояние;
* функция переходов;
* функция выходов.

Используются следующие **способы задания автоматов**:

* словесное описание;
* графическое описание (направленные графы);
* табличное описание (таблица переходов и выходов);
* аналитическое описание (задание конкретного вида функций переходов и выходов).

**Преобразователи с магазинной памятью** получаются из МПА, если их снабдить выходной лентой и разрешить на каждом такте выдавать выходную цепочку конечной длины. Формально они задаются восьмеркой:

P = (Q,Σ,Г,∆,δ,q0,Z0,F),

где Q – конечное множество состояний преобразователя;

Σ – конечный входной алфавит;

Γ – конечный алфавит магазинных символов;

∆ – конечный выходной алфавит;

δ – множество переходов преобразователя (отображение множества вида Q´ (Σ ∪ {ε}) ×Τ\* в множество конечных подмножеств множества Q´Г\*´∆\*);

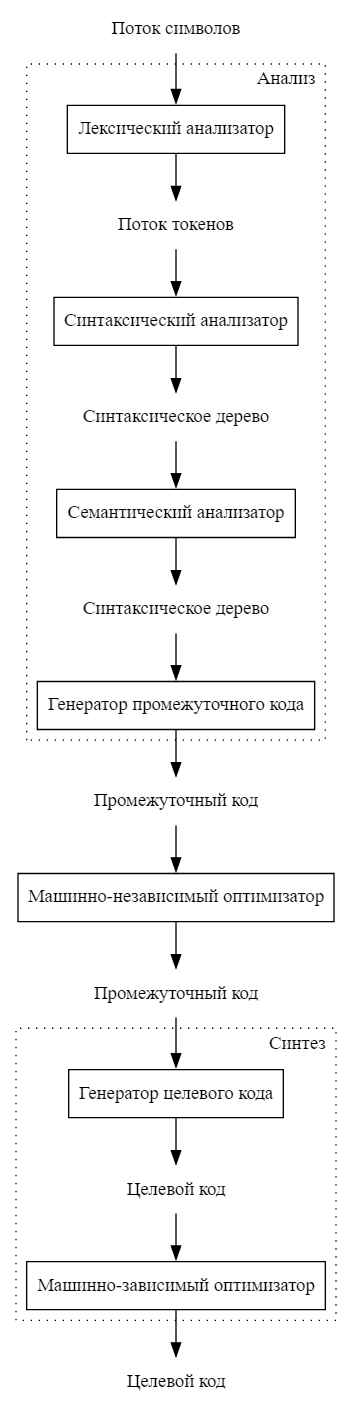
*q0* ∈ Q – начальное состояние автомата;

*Z0* ∈ Г – начальное состояние магазина;

*F* Í *Q* – множество заключительных состояний.

## Структура компилятора. Лексический анализатор.

Процесс компиляции обычно разделяется на две фазы: анализ и синтез.

В фазе анализа происходит чтение исходного текста программы, затем этот текст разбивается на элементарные блоки, на них накладывается грамматическая структура, и создаётся промежуточное представление исходного текста и собирается другая информация об исходном тексте. На этой фазе так же возможен статический анализ исходного текста.

В фазе синтеза, на основе промежуточного представления и прочей информации, строится представление исходной программы в целевом коде. На этой фазе так же возможны преобразования целевого кода, называемые оптимизациями.

Кроме того, между анализом и синтезом может находиться фаза преобразований промежуточного кода, называемая машинно-независимой оптимизацией.

Лексический анализ

Первая фаза компиляции называется лексическим анализом или сканированием.

Лексический анализатор соответственно так же называется лексером или сканером.

Лексический анализатор сканирует входной поток символов (исходного текста программы) и выделяет значащие последовательности символов, называемые лексемами.

Для каждой лексемы анализатор выводит токен, представляющий из себя комбинацию абстрактного символа (названия типа токена) и произвольного набора атрибутов. Часто в качестве “набора атрибутов” выступает ссылка в глобальную таблицу, называемую таблицей символов.

Синтаксический анализ

Вторая фаза – синтаксический анализ или разбор, парсинг (от англ. parsing).

Синтаксический анализатор соответственно называется так же парсером.

Парсер строит из токенов, полученных от лексера, древовидное промежуточное представление (часто неявно), отражающее грамматическую структуру исходного кода. Примером такого представления является синтаксическое дерево, где узлы представляют операцию, дочерние узлы – аргументы этой операции.

Например, синтаксическое дерево арифметического выражения 1+2∗31+2∗3 может иметь вид:

## Семантический анализ

Семантический анализатор использует синтаксическое дерево для проверки исходной программы на корректность.

На этом же этапе происходит проверка типов, и информация о типах переменных записывается в атрибуты соответствующих узлов синтаксического дерева.

Если спецификация языка разрешает неявное приведение типов, на этом этапе синтаксическое дерево может быть переписано с добавлением явных операций приведения типов.

## Генерация промежуточного кода

В процессе компиляции, могут создаваться несколько промежуточных представлений, в частности, синтаксическое дерево.

Как правило, после завершения синтаксического и семантического анализа, значительная часть высокоуровневой информации (типы, названия переменных, многие управляющие конструкции и т.п.) далее не требуется, в связи с чем многие компиляторы по достижении этой фазы генерируют более низкоуровневое представление, называемое обычно промежуточным кодом.

Основными требованиями к промежуточному коду являются, с одной стороны, простота его получения из синтаксического дерева, и с другой стороны, простота генерации на его основе машинного кода.

Как следствие, часто в качестве промежуточного кода используется последовательность инструкций для некой абстрактной вычислительной машины.

На этом этапе обычно принимаются решения о распределении памяти для хранения значений переменных.

Машинно-независимая оптимизация

На фазе машинно-независимой оптимизации, промежуточный код преобразуется с целью “улучшения” без изменений наблюдаемого поведения (в соответствии со спецификацией языка[1](https://wiki.livid.pp.ru/students/sp/lectures/1.html#fn1)). Под “улучшением” обычно понимается “ускорение”, но иногда возможны другие критерии, например “код меньшего размера” или “меньшее потребление памяти”.

Часто, алгоритм первичной генерации промежуточного кода достаточно простой, поэтому без фазы оптимизации, код оказывается достаточно неэффективным.

Объём работы, проделываемый различными компиляторами на этом этапе может сильно отличаться. Большинство распространённых на рынке компиляторов являются “оптимизирующими” и значительная часть времени компиляции уходит именно на оптимизацию (обычно есть способ отключить оптимизацию при необходимости).

Генерация целевого кода

Генератор целевого кода, получая на вход промежуточный код, отображает каждую команду промежуточного кода в одну или несколько команд целевого.

Кроме того, генератор целевого кода занимается задачей распределения регистров исполнительного устройства.

Машинно-зависимая оптимизация

Шаг машинно-зависимой оптимизации преобразует, как правило, уже целевой код. Основными способами оптимизации на данном этапе могут быть различные эквивалентные замены последовательностей машинных команд на более быстрые аналоги, не меняющие поведения перестановки команд или блоков команд, приводящие к ускорению и т.п.

Большинство решений машинно-зависимой оптимизации принимаются на основе модели исполнительного устройства, встроенной в компилятор. Например, в компилятор может быть включена информация об относительном времени выполнения различных инструкций определённого процессора (или семейства процессоров).

**Лексический анализ** («токенизация») – процесс аналитического разбора входной последовательности символов на распознанные группы – ***лексемы*** – с целью получения на выходе идентифицированных последовательностей, называемых **токенами** (подобно группировке букв в словах). В прикладном программировании понятия «лексема» и «токен» идентичны, но более сложные токенизаторы дополнительно классифицируют лексемы по различным типам («идентификатор, оператор», «часть речи» и т. п.).

Лексический анализ используется в компиляторах и интерпретаторах [исходного кода](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) ЯП, и в различных парсерах [слов](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B2%D0%BE) естественного языка. Как правило, он производится с точки зрения определённого ФЯ или набора языков. Язык, а точнее его грамматика, задаёт определённый набор лексем, которые могут встретиться на входе процесса.

Лексический анализатор (ЛА) обычно работает в две стадии –сканирование и оценка. На стадии сканирования ЛА обычно реализуется в виде конечного автомата, определяемого регулярными выражениями. В нём кодируется информация о возможных последовательностях символов, которые могут встречаться в токенах. Например, токен «целое число» может содержать любую последовательность десятичных цифр. Во многих случаях первый непробельный символ может использоваться для определения типа следующего токена, после чего входные символы обрабатываются один за другим пока не встретится символ, не входящий во множество допустимых символов для данного токена. В некоторых языках правила разбора лексем несколько более сложные и требуют возвратов назад по читаемой последовательности.

Чтобы получить токен со значением, соответствующим типу (например, целое или дробное число), выполняется **оценка** этой строки – проход по символам и вычисление значения. Токен с типом и подготовленным значением передаётся на вход синтаксического анализатора.

Результатом лексического анализа является список лексем-дескрипторов и таблицы. В таблицах хранятся значения выделенных в программе лексем.

**Дескриптор** − это пара вида: ( <тип лексемы> , < указатель>),

где <тип лексемы> − это, как правило, числовой код класса лексемы, который означает, что лексема принадлежит одному из конечного множества классов слов, выделенных в ЯП;

<указатель>− это может быть либо начальный адрес области основной памяти, в которой хранится адрес этой лексемы, либо число, адресующее элемент таблицы, в которой хранится значение этой лексемы.

## Синтаксические деревья и неоднозначность. Минимизация конечных автоматов.

Рассмотрим грамматику c правилами вывода

(**1.1.1**)

*B → B+B*⏐*B\*B*⏐*V*⏐*C*

*V → a*⏐*b*⏐*c*⏐*... x*⏐*y*⏐*z*

*C → 0*⏐*1*⏐*2*⏐*...8*⏐*9*

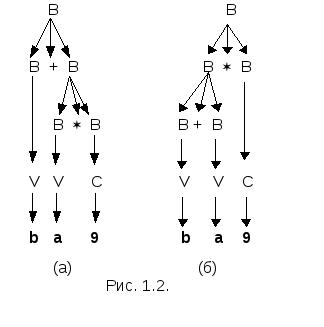
и вывод цепочки *b+a\*9*

(**1.1.2***) B⇒B+B⇒B+B*\**B⇒V+B*\**B⇒V+V*\**B⇒V+V*\**C⇒b+V*\**C⇒b+a*\**C⇒b+a*\**9*

Такая запись не очень удобна, так как по ней трудно определить в какой части сентенциальной формы проводилась замена и какой нетерминал породил тот или иной символ. Более наглядна запись в виде ***дерева вывода***или ***синтаксического дерева***, представленного на рис 1.2 (a).

Для того чтобы понять что выведено, применяем левый обход дерева. Идем от корня по крайней левой ветви, дойдя до терминала (конца ветви), выписываем его, возвращаемся до ближайшего разветвления и идем по самой левой из тех, которые еще не пройдены. Если все ветви данного узла уже исчерпаны, возвращаемся к предыдущему разветвлению, если оно есть. Продолжая таким образом, получим в результате *b+a\*9*. Кроме вывода (1.1.2) по данному дереву можно получить целую серию выводов, например,

(**1.1.3**) *B⇒B+B⇒B+B*\**B⇒B+B*\**C⇒V+V*\**9⇒B+V*\**9⇒B+a*\**C⇒V+a*\**C⇒b+a*\**9*

Заметим, что эти выводы отличаются лишь ***порядком*** применения правил и что синтаксическое дерево и грамматика не определяют точный порядок вывода. На каждом шаге вывода имеется некоторый произвол в выборе заменяемого нетерминала. На данном этапе эти различия порядка для нас несущественны и мы считаем выводы эквивалентными, если им соответствует одно и то же дерево. Более важным здесь является то, что цепочка *b+a*\**9* в данной грамматике имеет два дерева вывода (рисунки 1.2 (а) и (б)). Сентенциальная форма *B+B*\**B* имеет два синтаксических дерева и две основы: *B+B* и *B*\**B*. Грамматика неоднозначна и при разборе сентенциальной формы можно выбрать любую из основ. Нельзя сказать, что выполняется раньше: умножение или сложение. Из рис. 1.2 (б) следует, что *b+a*\**9* имеет два подвыражения*b+a* и *9*, хотя по смыслу необходимо иметь подвыражения *b* и *a*\**9*.

*Цепочка, порождаемая грамматикой,****неоднозначна****, если для ее вывода существует более одного синтаксического дерева.****Грамматика неоднозначна****, если она порождает неоднозначные цепочки, в противном случае она****однозначна****.*

Здесь речь идет о неоднозначной грамматике, а не языке. Изменяя неоднозначную грамматику можно получить однозначную грамматику для того же самого языка. Ниже приведена однозначная грамматика арифметических выражений

(**1.1.4**) *<врж> → <терм>*⎜*+ <терм>*⎜*− <терм>*⎜*<врж> + <терм>*⎜*<врж> − <терм>*

<*терм> → <множ>*⎜*<терм>*\* *<множ>*⎜*<терм> / <множ>*

<*множ> → (<врж>)* ⎜*i*⎜*k*

В этой грамматике *i* - любой идентификатор (имя переменной), а *k* - любая константа. Единственное дерево вывода для выражения *i+i*\**k* представлено на рис. 1.3. (a). В соответствии с предложенной грамматикой, эта, да и все остальные цепочки, порождаемые грамматикой (1.1.4) однозначны.

Определим теперь, что в выражении *i+i*\**k* должно выполняться раньше: сложение или умножение. Операндами для *+*, согласно дереву, является *<врж>*, из которого выводится *i*, и *<терм>*, порождающий *i*\**k*. Это означает, что умножение должно выполняться первым и образовать *<терм>* для сложения; следовательно, умножение предшествует сложению. Сделать наоборот можно используя только скобки, как показано на рис. 1.3 (б). Грамматику арифметических выражений (1.1.4) следует предпочесть грамматике (1.1.1) ввиду ее однозначности и учета приоритета операций.

Пусть дан автомат, распознающий определенный язык. Требуется найти эквивалентный автомат с наименьшим количеством состояний. Если в ДКА существуют два эквивалентных состояния, то при их объединении мы получим эквивалентный ДКА, так как распознаваемый язык не изменится.

Процесс минимизации КА начинается с поиска и удаления всех недостижимых состояний. Затем нужно найти такое разбиение множества состояний автомата, чтобы каждое подмножество содержало неразличимые состояния, т.е. если *s* и *t* принадлежат некоторому подмножеству, то для всех *a* из Σ *δ(s,a)* и *δ(t,a)* также принадлежат этому подмножеству.

Разобьём множество состояний на два подмножества: F и S–F. В дальнейшем нужно пытаться разбить каждое из подмножеств, соблюдая указанное выше условие. Если возникает ситуация, при которой невозможно разбить никакое множество состояний, то процесс разбиения заканчивается. В результате будет получен некоторый набор множеств состояний S1,…,Sk. Каждое из Si содержит только неразличимые состояния. Наконец, в множество состояний минимизированного автомата вносится по одному представителю каждого из множеств Si. На этом процесс завершается.

Пример. Дан исходный ДКА (рис.5).

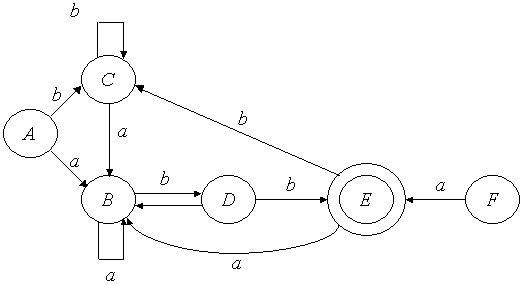


Рис.5. Исходный ДКА

Рассмотрим процесс минимизации данного автомата. Согласно алгоритму, вначале удаляются недостижимые состояния – в примере состояние F очевидно недостижимо и потому не попадёт в минимизированный автомат.

Затем множество состояний автомата разбивается на классы эквивалентности. Укажем такую последовательность разбиений:

1. E, ABCD
2. E, ABC, D, так как *δ(D,b)=E*.
3. E, AC, B, D, так как *δ(B,b)=D.*

Таким образом, состояния A и C неразличимы. Поэтому получаем следующий автомат (рис.6.):

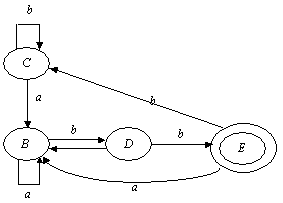


Рис.6. Минимизированный автомат

## Эквивалентные преобразования грамматик.

**Сентенциальная форма** – последовательность терминалов и нетерминалов, выводимых из начального символа.

Пусть дана КСГ c алфавитом терминалов {*a*, *b*} и начальным символом *S*:

*S* → *aS | A | a*

*B* → *b*

Для данной грамматики нетерминалы *A* и *B* не могут встречаться в сентенциальных формах выводов терминальных цепочек из *S*. Другими словами, они не принимают участия в порождении цепочек языка, то есть являются в этом смысле **бесполезными**. Любую КСГ можно привести к форме, не содержащей бесполезных символов.

Символ (терминал или нетерминал) **недостижимым** в КСГ, если он не появляется ни в одной сентенциальной форме этой грамматики.

Нетерминальный символ называется **бесплодным** в КСГ, если множество выводимых из этого символа терминальных цепочек пусто.

КСГ называется **приведенной**, если в ней нет недостижимых и бесплодных символов.

Символ X называется **полезным** в грамматике G=(N,T,P,S), если существует некоторое порождение (пошаговый процесс, в котором на каждом шаге из цепочки, уже полученной на предыдущем шаге, можно путем применения к ней правил замены получить новую цепочку) вида S→αXβ→w, где w∈N\*.

X может быть как переменной, так и терминалом, а выводимая цепочка αXβ – первой или последней в порождении. Если символ X не является полезным, то называется бесполезным.

Символ X называется порождающим*,* если X→w для некоторой терминальной цепочки w.

Символ X называется достижимым*,* если существует порождение S→αXβ для некоторых α и β.

Полезный символ является одновременно и порождающим и достижимым.

Если сначала удалить из грамматики непорождающие символы, а потом недостижимые (**но не наоборот!**), то останутся только полезные. По сути, это – **приведение** грамматики.

**Алгоритм удаления непорождающих символов**

**Шаг 0**. Создаём пустое множество порождающих нетерминалов.

**Шаг 1**. Находим правила, не содержащие нетерминалов в правых частях и добавляем нетерминалы, встречающихся в левых частях таких правил, в это множество.

**Шаг 2**. Если найдено такое правило, что все нетерминалы, стоящие в его правой части, уже входят в множество, то добавим в это множество нетерминалы, стоящие в его левой части.

**Шаг 3**. Повторяем предыдущий шаг, если множество порождающих нетерминалов изменилось.

В результате получаем множество всех порождающих нетерминалов грамматики, а все нетерминалы, не попавшие в него, являются непорождающими.

Алгоритм удаления недостижимых символов

**Шаг 0.** Создаём множество достижимых нетерминалов, состоящее из единственного элемента: {S}.

**Шаг 1.** Если найдено правило, в левой части которого стоит нетерминал, содержащийся в множестве, добавим в множество все нетерминалы из правой части.

**Шаг 2.** Повторяем предыдущий шаг, если множество достижимых нетерминалов изменилось.

Получаем множество всех достижимых нетерминалов, а нетерминалы, не попавшие в него, являются недостижимыми.

Некоторые применяемые на практике алгоритмы разбора по КСГ требуют, чтобы в грамматиках не было правил с пустой правой частью. Любую КСГ, не порождающую пустую цепочку, можно преобразовать в эквивалентную, не имеющую правил с пустыми правыми частями.

Алгоритм исключения правил с пустой правой частью

**Шаг 1.** Создаём множество Х нетерминалов грамматики G=(N,T,P,S), из которых выводима пустая цепочка.

**Шаг 2.** Удаляем из множества правил P все правила с пустой правой частью.

**Шаг 3.** Если S∈X, то вводим новый начальный символ S′, а в множество правил Pдобавляем правило S′ → S | ε.

**Шаг 4.** Для любого нетерминала A∈Xправило вида B→α1Aα2A...αnAαn+1,

где αi∈((N − {A}) ∪ T)\*, заменить 2*n* правилами, соответствующими всем возможным комбинациям вхождений А между αi:

B→α1α2...αnαn+1

B→α1α2...αnAαn+1

…

B→α1α2A...αnAαn+1

B→α1Aα2A...αnAαn+1

Замечание: если α*i*=ε для всех i=1,…,n+1, то получившееся на данном шаге правило B→ε не включаем в множество *P*

**Шаг 5.** Удаляем бесполезные символы и правила, их содержащие.

Пример. Рассмотрим грамматику с правилами S→*а*S*b*S | *b*S*a*S | ε.

Применяя алгоритм, получаем грамматику:

S'→S | ε

S→ *а*S*b*S | *b*S*a*S | *a*S*b* | *ab*S | *ab* | *b*S*a* | *ba*S | *ba*.

Цепное правило имеет вид A→B, где A и B являются нетерминалами. Для КСГ, содержащей цепные правила, можно построить эквивалентную ей грамматику, не содержащую цепных правил.

Алгоритм удаления цепных правил из грамматики:

**Шаг 1.** Находим все цепные пары в грамматике G.

**Шаг 2.** Для каждой цепной пары (A,B) добавляем в грамматику G′ все правила вида A→α, где B→α – нецепное правило из G.

**Шаг 3.** Удаляем все цепные правила.

Пример. Рассмотрим грамматику с правилами:

S→A|AB

A→SA|BB|B|bB|b

B→b|aA

Цепные правила: S→A и A→B

Цепные пары: (S,S), (A,A), (B,B)

Добавляем пары (S,A), (A,B), (S,B) для правила A→B и пары (S,A).

Меняем правила (жирным шрифтом выделены добавления):

S→**SA|BB|B|bB|b**|AB для (S,A),

A→SA|BB|**b|aA**|bB|b для (A,B),

S→SA|BB|**b|aA**|bB|b|AB для (S,B)

Удаляем повтор в A: A→SA|BB|aA|bB|b|

Удаляем повтор в S: S→SA|BB|aA|bB|b|AB

Правило вида A→αA , где A∈N , α∈(T∪N)\* , называется **праворекурсивным**, а правило вида A→ Aα – **леворекурсивным**. Для каждой КСГ G, содержащей леворекурсивные правила, можно построить эквивалентную грамматику G', не содержащую леворекурсивных правил.

**Алгоритм удаления леворекурсивных** **правил из грамматики:**

Пусть исходная грамматика G содержит правила:

A→Aα1 | Aα2 | ... |Aαn|β1 |β2 |...|βm ,

где ни одна цепочка β не начинается с A и αi,βj∈(T∪N)\*; i=1,…,n; j=1,…,m.

Введём новый нетерминал A' и преобразуем правила:

A → β1 |β2 |...|βm |β1A' |β2A'|...|βmA',

A'→ α1 |α2 |...|αn|α1A' |α2A'|...|αnA'.

Заменяя все правила с левой рекурсией в G описанным способом, получим грамматику G', такую что L(G)=L(G'), поскольку каждая цепочка, выведенная в грамматике G, может быть построена в грамматике G' и наоборот.

Рассмотрим построение выводов в G и G'. В грамматике G вывод цепочки имеет вид:

A→ Aα1 Aα1α1 Aα1α1α1 β1 α1α1α1.

В грамматике G' эта же цепочка выводится следующим образом:

A→ β1A'β1α1A'β1α1α1A'β1α1α1α1.

**Пример. Рассмотрим грамматику с правилами:**

A→Sα|Aα

S→Aβ

Здесь есть непосредственная левая рекурсия A→Aα.

Добавим нетерминал A′ и добавим правила:

A→SαA′

A′→αA′

Получаем новую грамматику с правилами:

A→SαA′|Sα

A′→αA′|α

S→Aβ

## Алгоритмы генерации объектного кода. Триады, тетрады.

Возможны различные формы внутреннего представления синтаксических конструкций исходной программы в компиляторе. На этапе синтаксического разбора часто используется форма, именуемая деревом вывода (методы его построения рассматривались в предыдущих лабораторных работах). Но формы представления, используемые на этапах синтаксического анализа, оказываются неудобными в работе при генерации и оптимизации объектного кода. Поэтому перед оптимизацией и непосредственно генерацией объектного кода внутреннее представление программы преобразуется в одну из соответствующих форм записи.

Примерами таких форм записи являются:

* обратная польская запись операций;
* тетрады операций;
* триады операций;
* собственно команды ассемблера.

Команды ассемблера удобны тем, что при их использовании внутреннее представление программы полностью соответствует объектному коду и сложные преобразования не требуются. Однако использование команд ассемблера требует дополнительных структур для отображения их взаимосвязи. Кроме того, внутреннее представление программы получается зависимым от результирующего кода, а это значит, что при ориентации компилятора на другой результирующий код потребуется перестраивать как само внутреннее представление программы, так и методы его обработки в алгоритмах оптимизации (при использовании триад или тетрад этого не требуется).

Для построения внутреннего представления объектного кода (в дальнейшем - просто кода) по дереву вывода может использоваться простейшая рекурсивная процедура. Эта процедура прежде всего должна определить тип узла дерева - он соответствует типу операции, символ которой находится в листе дерева для текущего узла. Этот лист является средним листом узла дерева для бинарных операций и крайним левым листом - для унарных операций. После определения типа процедура строит код для узла дерева в соответствии с типом операции. Если все узлы следующего уровня для текущего узла есть листья дерева, то в код включаются операнды, соответствующие этим листьям, и получившийся код становится результатом выполнения процедуры. Иначе процедура должна рекурсивно вызвать сама себя для генерации кода нижележащих узлов дерева и результат выполнения включить в свой порожденный код.

Поэтому для построения внутреннего представления объектного кода по дереву вывода в первую очередь необходимо разработать формы представления объектного кода для четырех случаев, соответствующих видам текущего узла дерева вывода:

* оба нижележащих узла дерева - листья (терминальные символы грамматики);
* только левый нижележащий узел является листом дерева;
* только правый нижележащий узел является листом дерева:
* оба нижележащих узла не являются листьями дерева.

Рассмотрим построение двух видов внутреннего представления по дереву вывода:

* построение ассемблерного кода по дереву вывода;
* построение списка триад по дереву вывода.

Существуют три формы записи выражений – префиксная, инфиксная и постфиксная. При префиксной записи операция записывается перед своими операндами, при инфиксной – между операндами, а при постфиксной – после операндов. Общепринятая запись арифметических выражений является примером инфиксной записи. Запись математических функций и функций в ЯП является префиксной (другие ее примеры – команды ассемблера, триады и тетрады).

**Тетрады** представляют собой запись операций в форме из четырех составляющих: операция, два операнда и результат операции. Например, тетрады могут выглядеть так:

<операция1>(<операнд1>,<операнд2>,<результат>).

Тетрады представляют собой линейную последовательность команд. При вычислении выражения, записанного в форме тетрад, они вычисляются одна за другой последовательно. Каждая тетрада в последовательности вычисляется так: операция, заданная тетрадой, выполняется над операндами и результат ее выполнения помещается в переменную, заданную результатом тетрады. Если какой-то из операндов (или оба операнда) в тетраде отсутствует (например, если тетрада представляет собой унарную операцию), то он может быть опущен или заменен пустым операндом (в зависимости от принятой формы записи и ее реализации).

**Триады** представляют собой запись операций в форме из трех составляющих – операция и два операнда:

<операция>(<операнд1>,<операнд2>).

Особенностью триад является то, что один или оба операнда могут быть ссылками на другую триаду в том случае, если в качестве операнда данной триады выступает результат выполнения другой триады. Поэтому триады при записи последовательно нумеруют для указания ссылок одних триад на другие.

Еще одно представление трехадресного кода состоит в использовании списка указателей на триады вместо списка самих триад. Такая реализация называется **косвенными триадами**.

## Трансляторы и компиляторы. Определения, примеры, методы работы.

Язык программирования задается описанием и реализуется в виде специальной программы – ***транслятора***. Существует два вида трансляторов – ***компиляторы*** и ***интерпретаторы***. Соответственно и языки программирования делятся на два класса – компилируемые и интерпретируемые.

Текст программы, записанный на каком-либо компилируемом языке, при помощи специальной программы компилятора преобразуется (компилируется) в набор инструкций для данного типа процессора (машинный код) и далее записывается в исполняемый файл, который может быть запущен на выполнение как отдельная программа. То есть компилятор переводит программу с языка высокого уровня на низкоуровневый язык, понятный процессору.

Если же программа написана на интерпретируемом языке, то интерпретатор непосредственно выполняет (интерпретирует) ее текст без предварительного перевода. То есть компилятор переводит программу на машинный язык сразу и целиком, создавая при этом отдельную программу, а интерпретатор переводит на машинный язык прямо во время исполнения программы. Можно сказать, что процессор компьютера — это аппаратный интерпретатор машинного кода.

Как правило, скомпилированные программы выполняются быстрее и не требуют для выполнения дополнительных средств, так как уже переведены на машинный язык. Вместе с тем при каждом изменении текста программы требуется ее перекомпиляция, что создает трудности для разработчиков. Кроме того, скомпилированная программа может выполняться только на том же типе компьютеров и, как правило, под той же операционной системой, на которые был рассчитан компилятор. Чтобы создать исполняемый файл для машины другого типа требуется новая компиляция.

Интерпретируемые языки обладают некоторыми специфическими дополнительными возможностями, кроме того, программы на них можно запускать сразу же после изменения, что облегчает разработку. Программа на интерпретируемом языке нередко может быть запущена на разных типах машин и операционных систем без дополнительных усилий. Однако интерпретируемые программы выполняются заметно медленнее, чем компилируемые, и они не могут выполняться без дополнительной программы-интерпретатора.

**Компилятор** — это вид транслятора, преобразующий исходный код с какого-либо [языка программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) на [машинный язык](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4)[[6]](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%82%D0%BE%D1%80#cite_note-%D0%A1%D0%B5%D0%B1%D0%B5%D1%81%D1%82%D0%B0-6).

Процесс компиляции, как правило, состоит из нескольких этапов:

* [лексический анализ](https://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7);
* [синтаксический анализ](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7);
* [семантический анализ](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D0%BD%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7);
* создание на основе результатов анализов промежуточного кода;
* [оптимизация](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D0%BC%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_%D0%BA%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80%D0%B0) промежуточного кода;
* создание [объектного кода](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C), в данном случае [машинного](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4).

Программа может использовать сервисы, предоставляемые [операционной системой](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), и сторонние [библиотеки](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) (например, библиотеки для работы с файлами и библиотеки для создания графического интерфейса). Для добавления в [объектный файл](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C) [машинного кода](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) из других [объектных файлов](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C) (кода [статических](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)#%D0%A1%D1%82%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B8) [библиотек](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))) и информации о [динамических](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)#%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B8) [библиотеках](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) выполняется *связывание* ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *link*) или *компоновка*. Связывание или компоновка выполняется [редактором связей или компоновщиком](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%BE%D0%B2%D1%89%D0%B8%D0%BA). Компоновщик может быть отдельной программой или частью [компилятора](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80). Компоновщик создаёт [исполняемый файл](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D0%BF%D0%BE%D0%BB%D0%BD%D0%B8%D0%BC%D1%8B%D0%B9_%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C). Исполняемый файл (программа) запускается следующим образом:

* по запросу пользователя в ядре [операционной системы](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) создаётся объект «[процесс](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0))»;
* [загрузчик программ](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D1%87%D0%B8%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC) [операционной системы](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) выполняет следующие действия:
* читает [исполняемый файл](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D0%BF%D0%BE%D0%BB%D0%BD%D1%8F%D0%B5%D0%BC%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB);
* загружает его в [память](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C);
* загружает в [память](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C) [динамические библиотеки](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)#%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B8);
* выполняет связывание машинного кода программы с динамическими библиотеками (динамическое связывание);
* передаёт управление программе.

Достоинства компиляции:

* компиляция программы выполняется один раз;
* наличие компилятора на устройстве, для которого компилируется программа, не требуется.

Недостатки компиляции:

* компиляция — медленный процесс;
* при внесении изменений в исходный код, требуется повторная компиляция.

[Ассемблер](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80) — компилятор, преобразующий текст с языка [ассемблера](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%B0%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80%D0%B0) на [машинный язык](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4). [Язык ассемблера](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%B0%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80%D0%B0) — [язык](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), близкий к [машинному языку](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4), язык [низкого уровня](https://ru.wikipedia.org/wiki/%D0%9D%D0%B8%D0%B7%D0%BA%D0%BE%D1%83%D1%80%D0%BE%D0%B2%D0%BD%D0%B5%D0%B2%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F).

Интерпретация — процесс чтения и выполнения [исходного кода](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4). Реализуется программой — [интерпретатором](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%82%D0%BE%D1%80).

Интерпретатор может работать двумя способами:

1. читать код и исполнять его сразу (*чистая интерпретация*[[6]](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%82%D0%BE%D1%80#cite_note-%D0%A1%D0%B5%D0%B1%D0%B5%D1%81%D1%82%D0%B0-6));
2. читать код, создавать в [памяти](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C) промежуточное представление кода ([байт-код](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4) или [p-код](https://ru.wikipedia.org/wiki/P-%D0%BA%D0%BE%D0%B4)), выполнять промежуточное представление кода (*смешанная реализация*[[6]](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%82%D0%BE%D1%80#cite_note-%D0%A1%D0%B5%D0%B1%D0%B5%D1%81%D1%82%D0%B0-6)).

В первом случае трансляция не используется, а во втором — используется трансляция исходного кода в промежуточный код.

Этапы работы интерпретатора:

1. [лексический анализ](https://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7);
2. [синтаксический анализ](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7);
3. [семантический анализ](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D0%BD%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7);
4. создание промежуточного представления кода (при чистой интерпретации не выполняется);
5. исполнение.

Интерпретатор моделирует машину ([виртуальную машину](https://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0)), реализует цикл выборки-исполнения команд машины. Команды машины записываются не на машинном языке, а на языке [высокого уровня](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%81%D0%BE%D0%BA%D0%BE%D1%83%D1%80%D0%BE%D0%B2%D0%BD%D0%B5%D0%B2%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F). Интерпретатор можно назвать исполнителем языка [виртуальной машины](https://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0).

Чистая интерпретация применяется, обычно, для языков с простой структурой, например, [языков сценариев](https://ru.wikipedia.org/wiki/%D0%A1%D1%86%D0%B5%D0%BD%D0%B0%D1%80%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA), языков [APL](https://ru.wikipedia.org/wiki/APL_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) и [Лисп](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D1%81%D0%BF).

Примеры интерпретаторов, создающих [байт-код](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4): [Perl](https://ru.wikipedia.org/wiki/Perl), [PHP](https://ru.wikipedia.org/wiki/PHP), [Python](https://ru.wikipedia.org/wiki/Python), [Erlang](https://ru.wikipedia.org/wiki/Erlang).

Достоинства интерпретаторов по сравнению с компиляторами:

* возможность работы в интерактивном режиме;
* отсутствие необходимости перекомпиляции [исходного кода](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) после внесения изменений и при переносе кода на другую платформу.

Недостатки интерпретаторов по сравнению с компиляторами:

* низкая производительность (машинный код исполняется процессором, а интерпретируемый код — интерпретатором; машинный код самого интерпретатора исполняется процессором);
* необходимость наличия интерпретатора на устройстве, на котором планируется интерпретация программы;
* обнаружение ошибок синтаксиса на этапе выполнения (актуально для чистых интерпретаторов).

Сравнение чистого интерпретатора и интерпретатора, создающего [байт-код](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4):

* чистый интерпретатор проще в реализации, так как для него не нужно писать код транслятора;
* интерпретатор, создающий [байт-код](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4), может выполнять его [оптимизацию](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D0%BC%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) и добиваться большей производительности, чем чистый интерпретатор;
* интерпретатор, создающий [байт-код](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4), потребляет больше ресурсов системы (трансляция в [байт-код](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4) занимает процессорное время; [байт-код](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4) занимает место в [памяти](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C)).

## Польская инверсная запись.

**Обратная польская запись** (ОПЗ) – форма записи математических и логических выражений, в которой операнды расположены перед знаками операций. Она также именуется как обратная бесскобочная запись, постфиксная нотация, польская инверсная запись (ПОЛИЗ).

Отличительная особенность ОПЗ – все аргументы (или операнды) расположены перед знаком операции. В общем виде при работе с ОПЗ руководствуются следующими правилами:

* Запись набора операций состоит из последовательности операндов и знаков операций. Операнды в выражении при письменной записи разделяются пробелами.
* Выражение читается слева направо. Когда в нём встречается знак операции, то выполняется соответствующая операция над двумя последними встретившимися перед ним операндами в порядке их записи. Результат операции заменяет в выражении последовательность её операндов и её знак, после чего выражение вычисляется дальше по тому же правилу.
* Результатом вычисления выражения становится результат последней вычисленной операции.

Вычисления на стеке. Алгоритм Дейкстры для преобразования выражений из инфиксной нотации в ОПЗ.

# Объектно-ориентированное программирование

## Сетевое программирование. Классы веб-программирования: WebClient, WebRequest, WebResponse, Uri, IPAddress, Dns. Сокеты..

## Классы. Конструкторы. Геттеры и сеттеры. Анонимные методы и делегаты.

## Наследование. Абстрактный класс. Производный класс. Переопределение методов. Интерфейсы. Многопоточное программирование. Классы потоков. Фоновые потоки. Операции потоков.

## ОО средства языка С++. Специальные методы классов — конструкторы и деструкторы. Привести пример на языке С++.

## Совместное использование функций. Перегрузка функций, виртуальные и статические методы. Привести пример на языке С++.

## Абстрактные классы и чистые виртуальные функции, перегрузка операторов. Привести пример на языке С++.

## Параллельное программирование. Планирование и приоритеты потоков, синхронизация потоков, блокировка потоков.

# Базы данных

## Основные функции СУБД.

Более точно, к числу функций СУБД принято относить следующие:

2.1.1. Непосредственное управление данными во внешней памяти

Эта функция включает обеспечение необходимых структур внешней памяти как для хранения данных, непосредственно входящих в БД, так и для служебных целей, например, для убыстрения доступа к данным в некоторых случаях (обычно для этого используются индексы). В некоторых реализациях СУБД активно используются возможности существующих файловых систем, в других работа производится вплоть до уровня устройств внешней памяти. Но подчеркнем, что в развитых СУБД пользователи в любом случае не обязаны знать, использует ли СУБД файловую систему, и если использует, то как организованы файлы. В частности, СУБД поддерживает собственную систему именования объектов БД.

2.1.2. Управление буферами оперативной памяти

СУБД обычно работают с БД значительного размера; по крайней мере этот размер обычно существенно больше доступного объема оперативной памяти. Понятно, что если при обращении к любому элементу данных будет производиться обмен с внешней памятью, то вся система будет работать со скоростью устройства внешней памяти. Практически единственным способом реального увеличения этой скорости является буферизация данных в оперативной памяти. При этом, даже если операционная система производит общесистемную буферизацию (как в случае ОС UNIX), этого недостаточно для целей СУБД, которая располагает гораздо большей информацией о полезности буферизации той или иной части БД. Поэтому в развитых СУБД поддерживается собственный набор буферов оперативной памяти с собственной дисциплиной замены буферов.

Заметим, что существует отдельное направление СУБД, которое ориентировано на постоянное присутствие в оперативной памяти всей БД. Это направление основывается на предположении, что в будущем объем оперативной памяти компьютеров будет настолько велик, что позволит не беспокоиться о буферизации. Пока эти работы находятся в стадии исследований.

2.1.3. Управление транзакциями

Транзакция - это последовательность операций над БД, рассматриваемых СУБД как единое целое. Либо транзакция успешно выполняется, и СУБД фиксирует (COMMIT) изменения БД, произведенные этой транзакцией, во внешней памяти, либо ни одно из этих изменений никак не отражается на состоянии БД. Понятие транзакции необходимо для поддержания логической целостности БД. Если вспомнить наш пример информационной системы с файлами СОТРУДНИКИ и ОТДЕЛЫ, то единственным способом не нарушить целостность БД при выполнении операции приема на работу нового сотрудника является объединение элементарных операций над файлами СОТРУДНИКИ и ОТДЕЛЫ в одну транзакцию. Таким образом, поддержание механизма транзакций является обязательным условием даже однопользовательских СУБД (если, конечно, такая система заслуживает названия СУБД). Но понятие транзакции гораздо более важно в многопользовательских СУБД.

То свойство, что каждая транзакция начинается при целостном состоянии БД и оставляет это состояние целостным после своего завершения, делает очень удобным использование понятия транзакции как единицы активности пользователя по отношению к БД. При соответствующем управлении параллельно выполняющимися транзакциями со стороны СУБД каждый из пользователей может в принципе ощущать себя единственным пользователем СУБД (на самом деле, это несколько идеализированное представление, поскольку в некоторых случаях пользователи многопользовательских СУБД могут ощутить присутствие своих коллег).

С управлением транзакциями в многопользовательской СУБД связаны важные понятия *сериализации транзакций* и *сериального плана выполнения смеси транзакций*. Под сериализаций параллельно выполняющихся транзакций понимается такой порядок планирования их работы, при котором суммарный эффект смеси транзакций эквивалентен эффекту их некоторого последовательного выполнения. Сериальный план выполнения смеси транзакций - это такой план, который приводит к сериализации транзакций. Понятно, что если удается добиться действительно сериального выполнения смеси транзакций, то для каждого пользователя, по инициативе которого образована транзакция, присутствие других транзакций будет незаметно (если не считать некоторого замедления работы по сравнению с однопользовательским режимом).

Существует несколько базовых алгоритмов сериализации транзакций. В централизованных СУБД наиболее распространены алгоритмы, основанные на синхронизационных захватах объектов БД. При использовании любого алгоритма сериализации возможны ситуации конфликтов между двумя или более транзакциями по доступу к объектам БД. В этом случае для поддержания сериализации необходимо выполнить откат (ликвидировать все изменения, произведенные в БД) одной или более транзакций. Это один из случаев, когда пользователь многопользовательской СУБД может реально (и достаточно неприятно) ощутить присутствие в системе транзакций других пользователей.

2.1.4. Журнализация

Одним из основных требований к СУБД является надежность хранения данных во внешней памяти. Под надежностью хранения понимается то, что СУБД должна быть в состоянии восстановить последнее согласованное состояние БД после любого аппаратного или программного сбоя. Обычно рассматриваются два возможных вида аппаратных сбоев: так называемые мягкие сбои, которые можно трактовать как внезапную остановку работы компьютера (например, аварийное выключение питания), и жесткие сбои, характеризуемые потерей информации на носителях внешней памяти. Примерами программных сбоев могут быть: аварийное завершение работы СУБД (по причине ошибки в программе или в результате некоторого аппаратного сбоя) или аварийное завершение пользовательской программы, в результате чего некоторая транзакция остается незавершенной. Первую ситуацию можно рассматривать как особый вид мягкого аппаратного сбоя; при возникновении последней требуется ликвидировать последствия только одной транзакции.

Понятно, что в любом случае для восстановления БД нужно располагать некоторой дополнительной информацией. Другими словами, поддержание надежности хранения данных в БД требует избыточности хранения данных, причем та часть данных, которая используется для восстановления, должна храниться особо надежно. Наиболее распространенным методом поддержания такой избыточной информации является ведение журнала изменений БД.

Журнал - это особая часть БД, недоступная пользователям СУБД и поддерживаемая с особой тщательностью (иногда поддерживаются две копии журнала, располагаемые на разных физических дисках), в которую поступают записи обо всех изменениях основной части БД. В разных СУБД изменения БД журнализуются на разных уровнях: иногда запись в журнале соответствует некоторой логической операции изменения БД (например, операции удаления строки из таблицы реляционной БД), иногда - минимальной внутренней операции модификации страницы внешней памяти; в некоторых системах одновременно используются оба подхода.

Во всех случаях придерживаются стратегии "упреждающей" записи в журнал (так называемого протокола Write Ahead Log - WAL). Грубо говоря, эта стратегия заключается в том, что запись об изменении любого объекта БД должна попасть во внешнюю память журнала раньше, чем измененный объект попадет во внешнюю память основной части БД. Известно, что если в СУБД корректно соблюдается протокол WAL, то с помощью журнала можно решить все проблемы восстановления БД после любого сбоя.

Самая простая ситуация восстановления - индивидуальный откат транзакции. Строго говоря, для этого не требуется общесистемный журнал изменений БД. Достаточно для каждой транзакции поддерживать локальный журнал операций модификации БД, выполненных в этой транзакции, и производить откат транзакции путем выполнения обратных операций, следуя от конца локального журнала. В некоторых СУБД так и делают, но в большинстве систем локальные журналы не поддерживают, а индивидуальный откат транзакции выполняют по общесистемному журналу, для чего все записи от одной транзакции связывают обратным списком (от конца к началу).

При мягком сбое во внешней памяти основной части БД могут находиться объекты, модифицированные транзакциями, не закончившимися к моменту сбоя, и могут отсутствовать объекты, модифицированные транзакциями, которые к моменту сбоя успешно завершились (по причине использования буферов оперативной памяти, содержимое которых при мягком сбое пропадает). При соблюдении протокола WAL во внешней памяти журнала должны гарантированно находиться записи, относящиеся к операциям модификации обоих видов объектов. Целью процесса восстановления после мягкого сбоя является состояние внешней памяти основной части БД, которое возникло бы при фиксации во внешней памяти изменений всех завершившихся транзакций и которое не содержало бы никаких следов незаконченных транзакций. Для того, чтобы этого добиться, сначала производят откат незавершенных транзакций (undo), а потом повторно воспроизводят (redo) те операции завершенных транзакций, результаты которых не отображены во внешней памяти. Этот процесс содержит много тонкостей, связанных с общей организацией управления буферами и журналом. Более подробно мы рассмотрим это в соответствующей лекции.

Для восстановления БД после жесткого сбоя используют журнал и архивную копию БД. Грубо говоря, архивная копия - это полная копия БД к моменту начала заполнения журнала (имеется много вариантов более гибкой трактовки смысла архивной копии). Конечно, для нормального восстановления БД после жесткого сбоя необходимо, чтобы журнал не пропал. Как уже отмечалось, к сохранности журнала во внешней памяти в СУБД предъявляются особо повышенные требования. Тогда восстановление БД состоит в том, что исходя из архивной копии по журналу воспроизводится работа всех транзакций, которые закончились к моменту сбоя. В принципе, можно даже воспроизвести работу незавершенных транзакций и продолжить их работу после завершения восстановления. Однако в реальных системах это обычно не делается, поскольку процесс восстановления после жесткого сбоя является достаточно длительным.

2.1.5. Поддержка языков БД

Для работы с базами данных используются специальные языки, в целом называемые *языками баз данных*. В ранних СУБД поддерживалось несколько специализированных по своим функциям языков. Чаще всего выделялись два языка - *язык определения схемы* БД *(SDL - Schema Definition Language)* и *язык манипулирования данными (DML - Data Manipulation Language).* SDL служил главным образом для определения логической структуры БД, т.е. той структуры БД, какой она представляется пользователям. DML содержал набор операторов манипулирования данными, т.е. операторов, позволяющих заносить данные в БД, удалять, модифицировать или выбирать существующие данные. Мы рассмотрим более подробно языки ранних СУБД в следующей лекции.

В современных СУБД обычно поддерживается единый интегрированный язык, содержащий все необходимые средства для работы с БД, начиная от ее создания, и обеспечивающий базовый пользовательский интерфейс с базами данных. Стандартным языком наиболее распространенных в настоящее время реляционных СУБД является язык SQL (Structured Query Language). В нескольких лекциях этого курса язык SQL будет рассматриваться достаточно подробно, а пока мы перечислим основные функции реляционной СУБД, поддерживаемые на "языковом" уровне (т.е. функции, поддерживаемые при реализации интерфейса SQL).

Прежде всего, язык SQL сочетает средства SDL и DML, т.е. позволяет определять схему реляционной БД и манипулировать данными. При этом именование объектов БД (для реляционной БД - именование таблиц и их столбцов) поддерживается на языковом уровне в том смысле, что компилятор языка SQL производит преобразование имен объектов в их внутренние идентификаторы на основании специально поддерживаемых служебных таблиц-каталогов. Внутренняя часть СУБД (ядро) вообще не работает с именами таблиц и их столбцов.

Язык SQL содержит специальные средства определения ограничений целостности БД. Опять же, ограничения целостности хранятся в специальных таблицах-каталогах, и обеспечение контроля целостности БД производится на языковом уровне, т.е. при компиляции операторов модификации БД компилятор SQL на основании имеющихся в БД ограничений целостности генерирует соответствующий программный код.

Специальные операторы языка SQL позволяют определять так называемые представления БД, фактически являющиеся хранимыми в БД запросами (результатом любого запроса к реляционной БД является таблица) с именованными столбцами. Для пользователя представление является такой же таблицей, как любая базовая таблица, хранимая в БД, но с помощью представлений можно ограничить или наоборот расширить видимость БД для конкретного пользователя. Поддержание представлений производится также на языковом уровне.

Наконец, авторизация доступа к объектам БД производится также на основе специального набора операторов SQL. Идея состоит в том, что для выполнения операторов SQL разного вида пользователь должен обладать различными полномочиями. Пользователь, создавший таблицу БД, обладает полным набором полномочий для работы с этой таблицей. В число этих полномочий входит полномочие на передачу всех или части полномочий другим пользователям, включая полномочие на передачу полномочий. Полномочия пользователей описываются в специальных таблицах-каталогах, контроль полномочий поддерживается на языковом уровне.

Более точное описание возможных реализаций этих функций на основе языка SQL будет приведено в лекциях, посвященных языку SQL и его реализации.

## Функциональные зависимости. Декомпозиция отношений.

Теперь мы можем ввести понятие функциональной зависимости (ФЗ):  
  
**Определение 1.** *Отношение R удовлетворяет ФЗ X → Y (где X, Y ⊆ R) тогда и только тогда, когда для любых кортежей ,  ∈ R выполняется: если [X] = [X], то [Y ] = [Y ]. В таком случае говорят, что X (детерминант, или определяющее множество атрибутов) функционально определяет Y (зависимое множество).*  
  
Иными словами, наличие ФЗ *X → Y* означает, что если мы имеем два кортежа в *R* и они совпадают по атрибутам *X*, то они будут совпадать и по атрибутам *Y*.  
А теперь по порядку. Рассмотрим атрибуты *Пациент* и *Пол* для которых хотим узнать, есть ли между ними зависимости или нет. Для такого множества атрибутов могут существовать следующие зависимости:

1. *Пациент → Пол*
2. *Пол → Пациент*

Согласно определению выше, для того чтобы удержалась первая зависимость, каждому уникальному значению столбца *Пациент* должно соответствовать только одно значение столбца *Пол*. И для таблицы-примера это действительно так. Однако в обратную сторону это не работает, то есть вторая зависимость не выполняется, а атрибут *Пол* не является детерминантом для *Пациента*. Аналогично, если взять зависимость *Доктор → Пациент*, можно заметить, что она нарушается, так как значение *Robin* по этому атрибуту имеет несколько разных значений — *Ellis и Graham*.

Последовательный переход от одной *нормальной формы* к другой при *нормализации* схем отношений реализуется через декомпозицию. Основной операцией, с помощью которой осуществляется *декомпозиция*, является *проекция*.

*Декомпозицией схемы отношения* R = {А1, А2, ...,Аn} *называется замена ее совокупностью подмножеств* R, *таких, что их объединение дает* R. *При этом допускается, чтобы подмножества были пересекающимися*.

*Алгоритм* декомпозиции основан на следующей теореме.

**Теорема о декомпозиции**. *Пусть* R(A, B, C) – *отношение*, A, B, C – *атрибуты*.

*Если* R *удовлетворяет зависимости* A->B, *то* R *равно соединению его проекций* A, B *и* A, C

R(A, B, C) = R(A, B), R(A, C)

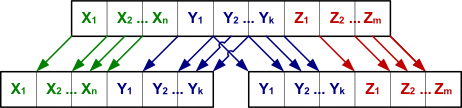
При *нормализации* необходимо выбирать такие декомпозиции, которые обладают свойством соединения без потерь. В этом случае, *декомпозиция* должна обеспечить то, что запросы (*выборка* данных по условию) к исходному отношению и отношениям, получаемым в результате декомпозиции, дадут одинаковый результат. Соответствующее условие будет выполняться, если каждый *кортеж отношения* R может быть представлен как *естественное соединение* его проекций на каждое из подмножеств. Для проверки, обладает ли *декомпозиция* данным свойством, используется специальные алгоритмы, описанные в литературе (в данной книге не рассматриваются).

Вторым важнейшим желательным свойством декомпозиции является свойство сохранения функциональных зависимостей. Стремление к тому, чтобы *декомпозиция* сохраняла зависимости, естественно. *Функциональные зависимости* являются некоторыми ограничениями на данные. Если *декомпозиция* не обладает этим свойством, то для того чтобы проверить, не нарушаются ли при вводе данных условия целостности (*функциональные зависимости*), нам приходится соединять все проекции.

Таким образом, для правильно построенного проекта *базы данных* необходимо, чтобы декомпозиции обладали свойством соединения без потерь, и желательно, чтобы они обладали свойством сохранения функциональных зависимостей.

Декомпозиция

Процедура нормализации предусматривает разбиение, или **декомпозицию**, данной переменной отношения на другие переменные отношения, причем декомпозиция должна быть обратимой, т.е. выполняться без потерь информации, то есть, соединение отношений, полученных при декомпозиции множества, должно давать исходное отношение Декомпозиция отношения R𝑅 на множества атрибутов A𝐴 и B𝐵: R(A,B)=πA(R)⋈πB(R)𝑅(𝐴,𝐵)=𝜋𝐴(𝑅)⋈𝜋𝐵(𝑅)

[](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:Decomposition.png)

Пример корректной декомпозиции

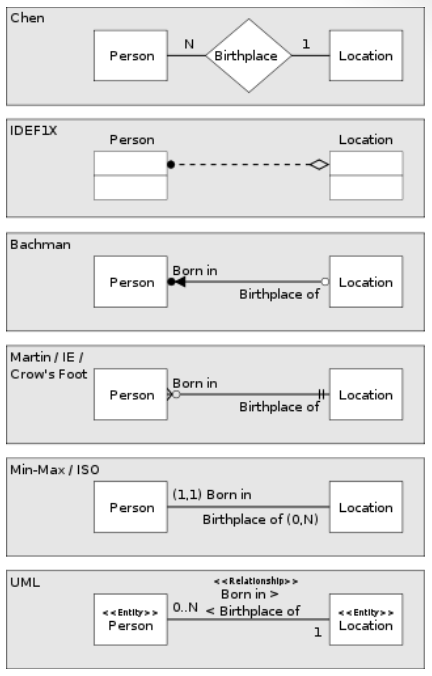
|  |  |
| --- | --- |
| Проекции на CId Phone и Lecturer Phone | Соединение CId Lecturer и Lecturer Phone |
| [Decomposition Example 1 1.png](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:Decomposition_Example_1_1.png) | [Decomposition Example 1 2.png](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:Decomposition_Example_1_2.png) |

Пример некорректной декомпозиции

При обратном соединении полученных отношений исходное отношений не было восстановлено — появились записи, которых не было ⇒ декомпозиция некорректна.

|  |  |
| --- | --- |
| Проекции на CId Phone и Lecturer Phone | Соединение CId Phone и Lecturer Phone |
| [Decomposition Example 2 1.png](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:Decomposition_Example_2_1.png) | [Decomposition Example 2 2.png](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:Decomposition_Example_2_2.png) |

## Модель сущность-связь. ERD.

Сущность (entity) – это предмет, который может быть идентифицирован некоторым способом, отличающим его от других предметов. Набор однородных сущностей – множество сущностей. • Примерами сущности являются конкретный человек или событие. • Атрибут – свойство сущности (как, правило, атомарное). • Связь (relationship) – это ассоциация, устанавливаемая между сущностями. Степень связи – количество связанных сущностей.

Нотации ER-диаграмм

•Классическая нотация П. Чена.

•Нотация IDEFIX (Integration

Definition for Information

Modeling).

•Нотация Ч. Бахмана.

•Нотация Дж. Мартина

("вороньи лапки").

•Нотация Ж.-Р. Абриаля (минмакс).

•Диаграммы классов UML.

Схема «сущность-связь» (также ERD или ER-диаграмма) — это разновидность блок-схемы, где показано, как разные «сущности» (люди, объекты, концепции и так далее) связаны между собой внутри системы. ER-диаграммы чаще всего применяются для проектирования и отладки реляционных баз данных в сфере образования, исследования и разработки программного обеспечения и информационных систем для бизнеса.

В ER-моделях и моделях данных обычно выделяют до трех уровней детализации:

Концептуальная модель данных

 — схема наивысшего уровня с минимальным количеством подробностей. Достоинство этого подхода заключается в возможности отобразить общую структуру модели и всю архитектуру системы. Менее масштабные системы могут обойтись и без этой модели. В этом случае можно сразу переходить к логической модели.

Логическая модель данных

 содержит более подробную информацию, нежели концептуальная модель. На этом уровне определяются более подробные операционные и транзакционные сущности. Логическая модель не зависит от технологии, в которой она будет применяться.

Физическая модель данных

: на основе каждой логической модели данных можно составить одну или две физических модели. В последних должно присутствовать достаточно технических подробностей для составления и внедрения самой базы данных.

Только реляционные данные

. Следует четко понимать, что цель ER-диаграмм — показать связи и отношения между элементами, поэтому они отображают только реляционную структуру.

Только для структурированных данных

. Данные должны быть четко разбиты на поля, столбцы и строки, иначе пользы от ER-диаграммы будет мало. Это касается и частично структурированных данных, так как только некоторые из них будут пригодны для работы.

Сложность интеграции с существующей базой данных

. Применение ER-моделей для интеграции с существующей базой данных — непростая задача по причине различия в архитектурах.

Области применения диаграмм «сущность-связь»

Проектирование баз данных

. ER-диаграммы применяются для моделирования и проектирования реляционных баз данных, причем как в плане логических и бизнес-правил (логические модели данных), так и в плане внедрения конкретных технологий (физические модели данных). В сфере разработки программного обеспечения ER-диаграмма, как правило, служит первым шагом в определении требований проекта по созданию информационных систем. На дальнейших этапах работы ER-диаграммы также применяются для моделирования конкретных баз данных. Реляционная база данных сопровождается соответствующей реляционной таблицей и при необходимости может быть представлена в этом формате.

Отладка баз данных

. ER-диаграммы применяются для анализа уже имеющихся баз данных с целью выявить и устранить ошибки в логике или развертывании. Диаграмма позволяет выявить, где именно закрались ошибки.

## SQL.Создание и модификация структуры таблиц БД.

Для создания таблиц используется команда **CREATE TABLE**. Общий формальный синтаксис команды **CREATE TABLE**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | CREATE TABLE название\_таблицы  (название\_столбца1 тип\_данных атрибуты\_столбца1,   название\_столбца2 тип\_данных атрибуты\_столбца2,   ................................................   название\_столбцаN тип\_данных атрибуты\_столбцаN,   атрибуты\_уровня\_таблицы  ) |

Для удаления таблицы применяется команда **DROP TABLE**, после которой указывается название удаляемой таблицы. Например, удалим таблицу users:

|  |  |
| --- | --- |
| 1 | DROP TABLE users; |

По аналогии с созданием таблицы, если мы попытаемся удалить таблицу, которая не существует, то мы столкнемся с ошибкой. В этом случае опять же с помощью операторов **IF EXISTS** проверять наличие таблицы перед удалением:

|  |  |
| --- | --- |
| 1 | DROP TABLE IF EXISTS users; |

Переименование таблицы

Для переименования таблицы применяется операторы **RENAME TO**, после которого указывается новое имя таблицы:

|  |  |
| --- | --- |
| 1  2 | ALTER TABLE users  RENAME TO people; |

Здесь таблица users переименовывается в "people".

Добавление нового столбца

Добавим в таблицу users новый столбец email:

|  |  |
| --- | --- |
| 1  2 | ALTER TABLE users  ADD COLUMN email TEXT NOT NULL; |

В данном случае столбец email имеет тип TEXT и для него определено ограничение NOT NULL.

Переименование столбца

Переименуем столбец email в login

|  |  |
| --- | --- |
| 1  2 | ALTER TABLE users  RENAME COLUMN email TO login; |

Удаление столбца

Удалим столбец login из таблицы users:

|  |  |
| --- | --- |
| 1  2 | ALTER TABLE users  DROP COLUMN login; |

## SQL. Выборка, проекция, соединение и сортировка.

Но при таком запросе производители повторяются. Теперь применим оператор **DISTINCT** для выборки уникальных значений:

|  |  |
| --- | --- |
| 1 | SELECT DISTINCT company FROM products; |

Оператор **LIMIT** позволяет задать ограничение на количество строк и имеет следующий синтаксис:

|  |  |
| --- | --- |
| 1 | LIMIT [смещение,] количество\_строк |

**Проекция** в SQL — это операция отбора определенных **столбцов** из таблицы:

SELECT column\_name FROM table\_name;

На контраст с этим, **выборка** предусматривает извлечение **строк**, которые соответствуют заданным условиям:

SELECT \* FROM table\_name WHERE condition;

Обе операции являются фундаментальными для формирования результатов запросов к SQL-базам данных.

Соединение

LEFT JOIN

Пока мы все еще не можем решить нашу исходную задачу. Для этого понадобится операция левого соединения LEFT JOIN:

LEFT JOIN берет все данные из одной таблицы и присоединяет к ним данные из другой, если они присутствуют. Если нет, то заполняет их NULL. Чисто технически этот запрос отличается только тем, что добавляется слово LEFT:

**SELECT** first\_name, title **FROM** users

**LEFT** **JOIN** topics **ON** users.id **=** topics.user\_id **LIMIT** 5;

LEFT JOIN полезен, когда нам нужно работать со всеми данными одной таблицы и связанными с ними записями, если они есть. Если их нет, то ничего страшного, мы все равно хотим получить данные из первой таблицы.



Сортировка

Оператор **ORDER BY** сортируют значения по одному или нескольких столбцам. Например, упорядочим выборку из таблицы products по столбцу price:

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM products  ORDER BY price; |

## SQL. Манипуляции данными.

Для добавления данных в SQLite применяется команда **INSERT**, которая имеет следующее формальное определение:

|  |  |
| --- | --- |
| 1  2 | INSERT INTO имя\_таблицы [(столбец1, стобец2, ... стобецN)]  VALUES (значение1, значение2, ... значениеN) |

Нередко при получении данных из БД выбираются только те данные, которые соответствуют некоторому определенному условию. Для фильтрации данных в команде SELECT применяется оператор **WHERE**, после которого указывается условие:

|  |  |
| --- | --- |
| 1 | WHERE условие |

Для обновления данных в SQLite применяется команда **UPDATE**. Она имеет следующий формальный синтаксис:

|  |  |
| --- | --- |
| 1  2  3 | UPDATE имя\_таблицы  SET столбец1 = значение1, столбец2 = значение2, ... столбецN = значениеN  [WHERE условие\_обновления] |

Команда **DELETE** удаляет данные из БД. Она имеет следующий формальный синтаксис:

|  |  |
| --- | --- |
| 1  2 | DELETE FROM имя\_таблицы  [WHERE условие\_удаления] |

Команда **REPLACE** по сути является сокращением от **INSERT OR REPLACE**. Ее идея состоит в следующем. Пи вставке данных может нарушаются ограничения **UNIQUE** или **PRIMARY KEY**, например, когда мы пытаемся добавить для столбца, который должен иметь уникальные значения, данные, которые уже есть в таблице. Этот конфликт ограничений призвана разрешить команда **REPLACE**.

Эта команда сначала удаляет строку, которая вызвала конфликт на уникальность данных, и затем вместо нее вставляет новую строку. То есть фактически все выглядит как замена строки.

Если происходит конфликт с ограничением **NOT NULL** (в столбец, для которого задано ограничение NOT NULL, вставляется значение NULL), команда REPLACE заменяется вставляемое значение NULL значением по умолчанию, которое принято для этого столбца. Если для столбца не установлено значение по умолчанию, то выполнение запроса отменяется.

Если конфликтов с ограничениями не происходит, то команда **REPLACE** по сути действует аналогично команде **INSERT**.

Команда REPLACE имеет следующий формальный синтаксис:

|  |  |
| --- | --- |
| 1  2 | INSERT OR REPLACE INTO table(столбец1, столбец2, ... столбецN)  VALUES(значение1, значение2, ... значениеN); |

## SQL. Триггеры. Процедуры. Функции. Агрегатные функции.

Триггеры представляют специальный тип хранимой процедуры, которая вызывается автоматически при выполнении определенного действия над таблицей или представлением, в частности, при добавлении, изменении или удалении данных, то есть при выполнении команд INSERT, UPDATE, DELETE.

Формальное определение триггера:

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE TRIGGER имя\_триггера  ON {имя\_таблицы | имя\_представления}  {AFTER | INSTEAD OF} [INSERT | UPDATE | DELETE]  AS выражения\_sql |

Для создания триггера применяется выражение **CREATE TRIGGER**, после которого идет имя триггера. Как правило, имя триггера отражает тип операций и имя таблицы, над которой производится операция.

Каждый триггер ассоциируется с определенной таблицей или представлением, имя которых указывается после слова **ON**.

Затем устанавливается тип триггера. Мы можем использовать один из двух типов:

* **AFTER**: выполняется после выполнения действия. Определяется только для таблиц.
* **INSTEAD OF**: выполняется вместо действия (то есть по сути действие - добавление, изменение или удаление - вообще не выполняется). Определяется для таблиц и представлений

После типа триггера идет указание операции, для которой определяется триггер: **INSERT**, **UPDATE** или **DELETE**.

Для триггера AFTER можно применять сразу для нескольких действий, например, UPDATE и INSERT. В этом случае операции указываются через запятую. Для триггера INSTEAD OF можно определить только одно действие.

И затем после слова AS идет набор выражений SQL, которые собственно и составляют тело триггера.

хранимые процедуры представляют набор инструкций, которые выполняются как единое целое. Тем самым хранимые процедуры позволяют упростить комплексные операции и вынести их в единый объект. Изменится процесс покупки товара, соответственно достаточно будет изменить код процедуры. То есть процедура также упрощает управление кодом.

Также хранимые процедуры позволяют ограничить доступ к данным в таблицах и тем самым уменьшить вероятность преднамеренных или неосознанных нежелательных действий в отношении этих данных.

И еще один важный аспект - производительность. Хранимые процедуры обычно выполняются быстрее, чем обычные SQL-инструкции. Все потому что код процедур компилируется один раз при первом ее запуске, а затем сохраняется в скомпилированной форме.

Для создания хранимой процедуры применяется команда **CREATE PROCEDURE** или **CREATE PROC**.

Таким образом, хранимая процедура имеет три ключевых особенности: упрощение кода, безопасность и производительность.

Выполнение процедуры

Для выполнения хранимой процедуры вызывается команда **EXEC** или **EXECUTE**

Для удаления процедуры применяется команда **DROP PROCEDURE**:

|  |  |
| --- | --- |
| 1 | DROP PROCEDURE ProductSummary |

Агрегатные функции выполняют вычисления над значениями в наборе строк. В T-SQL имеются следующие агрегатные функции:

* **AVG**: находит среднее значение
* **SUM**: находит сумму значений
* **MIN**: находит наименьшее значение
* **MAX**: находит наибольшее значение
* **COUNT**: находит количество строк в запросе

В качестве аргумента все агрегатные функции принимают выражение, которое представляет критерий дя определения значений. Зачастую, в качестве выражения выступает название столбца, над значениями которого надо проводить вычисления.

Выражения в функциях **AVG** и **SUM** должно представлять числовое значение. Выражение в функциях **MIN**, **MAX** и **COUNT** может представлять числовое или строковое значение или дату.

Все агрегатные функции за исключением COUNT(\*) игнорируют значения NULL.

Avg

Функция **Avg** возвращает среднее значение на диапазоне значений столбца таблицы.

SELECT AVG(Price) AS Average\_Price FROM Products

Count

Функция **Count** вычисляет количество строк в выборке. Есть две формы этой функции. Первая форма COUNT(\*) подсчитывает число строк в выборке:

|  |  |
| --- | --- |
| 1 | SELECT COUNT(\*) FROM Products |

Функции **Min** и **Max** возвращают соответственно минимальное и максимальное значение по столбцу. Например, найдем минимальную цену среди товаров:

|  |  |
| --- | --- |
| 1 | SELECT MIN(Price) FROM Products |

Поиск максимальной цены:

|  |  |
| --- | --- |
| 1 | SELECT MAX(Price) FROM Products |

Sum

Функция **Sum** вычисляет сумму значений столбца. Например, подсчитаем общее количество товаров:

|  |  |
| --- | --- |
| 1 | SELECT SUM(ProductCount) FROM Products |

Создание функций очень похоже на создание процедур и объектов просмотра. Недаром мы рассматриваем все эти темы в одной главе. Для создания функции используется оператор CREATE FUNCTION. В зависимости от типа, Объявление будет отличаться. Рассмотрим все три типа объявления.

Скалярная функция:

CREATE FUNCTION [ owner\_name. ] function\_name

( [ { @parameter\_name [AS] scalar\_parameter\_data\_type [ = default ] }

[ ,...n ] ] )

RETURNS scalar\_return\_data\_type

[ WITH < function\_option> [ [,] ...n] ]

[ AS ]

BEGIN

function\_body

RETURN scalar\_expression

END

Функция, возвращающая таблицу:

CREATE FUNCTION [ owner\_name. ] function\_name

( [ { @parameter\_name [AS] scalar\_parameter\_data\_type [ = default ] }

[ ,...n ] ] )

RETURNS TABLE

[ WITH < function\_option > [ [,] ...n ] ]

[ AS ]

RETURN [ ( ] select-stmt [ ) ]

Многооператорные функции:

CREATE FUNCTION [ owner\_name. ] function\_name

( [ { @parameter\_name [AS] scalar\_parameter\_data\_type [ = default ] }

[ ,...n ] ] )

RETURNS @return\_variable TABLE < table\_type\_definition >

[ WITH < function\_option > [ [,] ...n ] ]

[ AS ]

BEGIN

function\_body

RETURN

END

< function\_option > ::=

{ ENCRYPTION | SCHEMABINDING }

< table\_type\_definition > :: =

( { column\_definition | table\_constraint } [ ,...n ] )

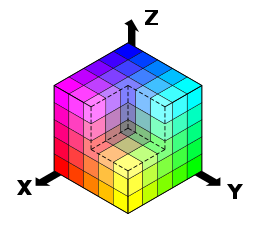
# Компьютерная графика и компьютерное моделирование

## Понятие системы координат. Многомерные пространства и проекции.

Систе́ма координа́т — комплекс определений, реализующий метод координат, то есть способ определять положение и перемещение точки или тела с помощью чисел или других символов. Совокупность чисел, определяющих положение конкретной точки, называется координатами этой точки.

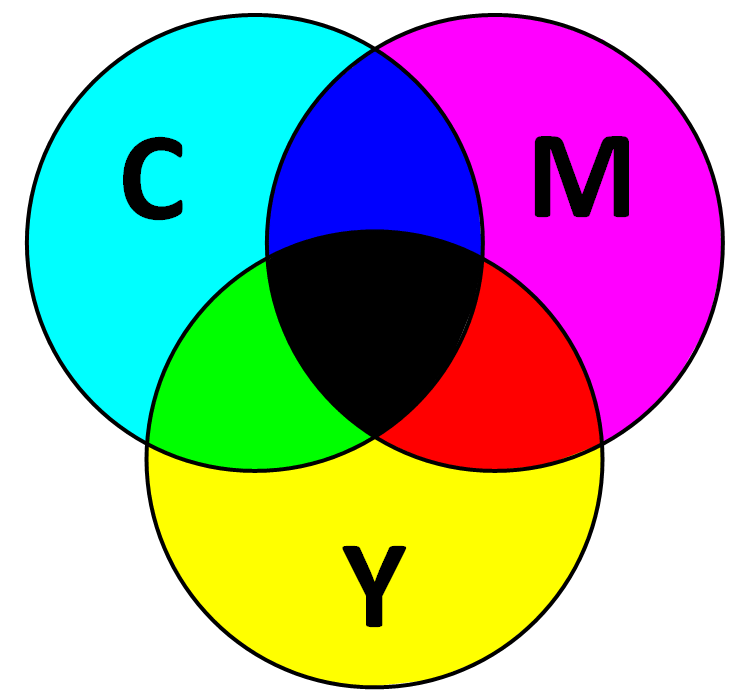
## Системы математического описания цвета. Модели CMYK, RGB, CIE LAB.

**RGB** ([аббревиатура](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D0%B1%D1%80%D0%B5%D0%B2%D0%B8%D0%B0%D1%82%D1%83%D1%80%D0%B0) английских слов **R**ed, **G**reen, **B**lue — [красный](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B0%D1%81%D0%BD%D1%8B%D0%B9_%D1%86%D0%B2%D0%B5%D1%82), [зелёный](https://ru.wikipedia.org/wiki/%D0%97%D0%B5%D0%BB%D1%91%D0%BD%D1%8B%D0%B9_%D1%86%D0%B2%D0%B5%D1%82), [синий](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D0%B8%D0%B9_%D1%86%D0%B2%D0%B5%D1%82)) или **КЗС** — [аддитивная](https://ru.wikipedia.org/wiki/%D0%90%D0%B4%D0%B4%D0%B8%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D1%81%D0%BC%D0%B5%D1%88%D0%B5%D0%BD%D0%B8%D0%B5_%D1%86%D0%B2%D0%B5%D1%82%D0%BE%D0%B2) [цветовая модель](https://ru.wikipedia.org/wiki/%D0%A6%D0%B2%D0%B5%D1%82%D0%BE%D0%B2%D0%B0%D1%8F_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C), описывающая способ кодирования [цвета](https://ru.wikipedia.org/wiki/%D0%A6%D0%B2%D0%B5%D1%82) для [цветовоспроизведения](https://ru.wikipedia.org/wiki/%D0%A6%D0%B2%D0%B5%D1%82%D0%BE%D0%B2%D0%BE%D1%81%D0%BF%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5) с помощью трёх цветов, которые принято называть [*основными*](https://ru.wikipedia.org/wiki/%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D0%BD%D1%8B%D0%B5_%D1%86%D0%B2%D0%B5%D1%82%D0%B0). Выбор основных цветов обусловлен особенностями физиологии восприятия цвета [сетчаткой](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BB%D0%B1%D0%BE%D1%87%D0%BA%D0%B8_(%D1%81%D0%B5%D1%82%D1%87%D0%B0%D1%82%D0%BA%D0%B0)) человеческого глаза.

RGB-модель является [аддитивной](https://ru.wikipedia.org/wiki/%D0%90%D0%B4%D0%B4%D0%B8%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D1%81%D0%BC%D0%B5%D1%88%D0%B5%D0%BD%D0%B8%D0%B5_%D1%86%D0%B2%D0%B5%D1%82%D0%BE%D0%B2), где производные цвета получаются путём добавления к чёрному цвету различных пропорций *основных цветов* модели. При отсутствии всякого излучения (добавления) — нет никакого цвета — [чёрный](https://ru.wikipedia.org/wiki/%D0%A7%D1%91%D1%80%D0%BD%D1%8B%D0%B9_%D1%86%D0%B2%D0%B5%D1%82), смешение всех трёх в определённой[[*какой?*](https://ru.wikipedia.org/wiki/%D0%92%D0%B8%D0%BA%D0%B8%D0%BF%D0%B5%D0%B4%D0%B8%D1%8F:%D0%98%D0%B7%D0%B1%D0%B5%D0%B3%D0%B0%D0%B9%D1%82%D0%B5_%D0%BD%D0%B5%D0%BE%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D1%91%D0%BD%D0%BD%D1%8B%D1%85_%D0%B2%D1%8B%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B9)] пропорции — даёт [белый](https://ru.wikipedia.org/wiki/%D0%91%D0%B5%D0%BB%D1%8B%D0%B9_%D1%86%D0%B2%D0%B5%D1%82). Если цвет экрана, освещённого цветным прожектором, обозначается в RGB как (r1, g1, b1), а цвет того же экрана, освещённого другим прожектором, — (r2, g2, b2), то при освещении двумя прожекторами цвет экрана будет обозначаться как (r1+r2, g1+g2, b1+b2).

Изображение в данной цветовой модели состоит из трёх каналов. При смешении основных излучений, например, синего (B) и красного (R), получается [пурпурный](https://ru.wikipedia.org/wiki/%D0%9F%D1%83%D1%80%D0%BF%D1%83%D1%80%D0%BD%D1%8B%D0%B9) (M, magenta), зелёного (G) и красного (R) — [жёлтый](https://ru.wikipedia.org/wiki/%D0%96%D1%91%D0%BB%D1%82%D1%8B%D0%B9) (Y, yellow), зелёного (G) и синего (B) — [циановый](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D0%B5-%D0%B7%D0%B5%D0%BB%D1%91%D0%BD%D1%8B%D0%B9_%D1%86%D0%B2%D0%B5%D1%82) (С, cyan). При смешении всех трёх основных излучений получается белый цвет (W, white).

В [телевизорах](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BB%D0%B5%D0%B2%D0%B8%D0%B7%D0%BE%D1%80) и [мониторах](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%BD%D0%B8%D1%82%D0%BE%D1%80_(%D1%83%D1%81%D1%82%D1%80%D0%BE%D0%B9%D1%81%D1%82%D0%B2%D0%BE)) [ЭЛТ](https://ru.wikipedia.org/wiki/%D0%AD%D0%9B%D0%A2) применяются три электронных пушки для красного, зелёного и синего каналов. В ЖК- и других матричных мониторах и телевизорах носителями трёх цветов являются светоточки ([светодиоды](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D0%B5%D1%82%D0%BE%D0%B4%D0%B8%D0%BE%D0%B4), [светофильтры](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D0%B5%D1%82%D0%BE%D1%84%D0%B8%D0%BB%D1%8C%D1%82%D1%80)).

 **CMYK** ([**C**yan](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D0%B5-%D0%B7%D0%B5%D0%BB%D1%91%D0%BD%D1%8B%D0%B9_%D1%86%D0%B2%D0%B5%D1%82), [**M**agenta](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D0%B4%D0%B6%D0%B5%D0%BD%D1%82%D0%B0_(%D1%86%D0%B2%D0%B5%D1%82)), [**Y**ellow](https://ru.wikipedia.org/wiki/%D0%96%D1%91%D0%BB%D1%82%D1%8B%D0%B9_%D1%86%D0%B2%D0%B5%D1%82), **K**ey или [Blac**k**](https://ru.wikipedia.org/wiki/%D0%A7%D1%91%D1%80%D0%BD%D1%8B%D0%B9_%D1%86%D0%B2%D0%B5%D1%82)) или **CMY+K** — [субтрактивная](https://ru.wikipedia.org/wiki/%D0%A1%D1%83%D0%B1%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D1%81%D0%B8%D0%BD%D1%82%D0%B5%D0%B7) [схема формирования цвета](https://ru.wikipedia.org/wiki/%D0%A6%D0%B2%D0%B5%D1%82%D0%BE%D0%B2%D0%B0%D1%8F_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C), а также **четырёхцветная**[**автотипия**](https://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D1%82%D0%B8%D0%BF%D0%B8%D1%8F), используемая прежде всего в [полиграфии](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%8F) для стандартной *триадной печати* или печати *триадными красками*. Она использует [голубой](https://ru.wikipedia.org/wiki/%D0%93%D0%BE%D0%BB%D1%83%D0%B1%D0%BE%D0%B9_%D1%86%D0%B2%D0%B5%D1%82), [пурпурный](https://ru.wikipedia.org/wiki/%D0%9F%D1%83%D1%80%D0%BF%D1%83%D1%80%D0%BD%D1%8B%D0%B9_%D1%86%D0%B2%D0%B5%D1%82) и [жёлтый](https://ru.wikipedia.org/wiki/%D0%96%D1%91%D0%BB%D1%82%D1%8B%D0%B9_%D1%86%D0%B2%D0%B5%D1%82) цвета в роли основных, а также [чёрный](https://ru.wikipedia.org/wiki/%D0%A7%D1%91%D1%80%D0%BD%D1%8B%D0%B9_%D1%86%D0%B2%D0%B5%D1%82) цвет[[1]](https://ru.wikipedia.org/wiki/CMYK#cite_note-1).

Цвет в CMYK зависит не только от спектральных характеристик пигментов и от способа их нанесения, но и их количества, характеристик подложки (бумаги и др.) и других факторов. Фактически, цифры CMYK являются лишь набором аппаратных данных для [фотонаборного автомата](https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%82%D0%BE%D0%BD%D0%B0%D0%B1%D0%BE%D1%80%D0%BD%D1%8B%D0%B9_%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82) или [CTP](https://ru.wikipedia.org/wiki/CTP) и не определяют [цвет](https://ru.wikipedia.org/wiki/%D0%A6%D0%B2%D0%B5%D1%82) однозначно.

Так как модель CMYK применяют в основном в [полиграфии](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%8F) при цветной [печати](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%87%D0%B0%D1%82%D0%B0%D0%BD%D0%B8%D0%B5), а бумага и прочие печатные материалы являются поверхностями, отражающими свет, удобнее считать, какое количество света отразилось от той или иной поверхности, нежели сколько поглотилось. Таким образом, если вычесть из белого три первичных цвета, [RGB](https://ru.wikipedia.org/wiki/RGB), мы получим тройку [дополнительных цветов](https://ru.wikipedia.org/wiki/%D0%94%D0%BE%D0%BF%D0%BE%D0%BB%D0%BD%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%86%D0%B2%D0%B5%D1%82%D0%B0) CMY. «Субтрактивный» означает «вычитаемый» — из белого вычитаются первичные цвета.

В CMYK используются четыре цвета, первые три в аббревиатуре названы по первой букве цвета, а в качестве четвёртого используется [чёрный](https://ru.wikipedia.org/wiki/%D0%A7%D1%91%D1%80%D0%BD%D1%8B%D0%B9_%D1%86%D0%B2%D0%B5%D1%82). Одна из версий утверждает, что K — сокращение от [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *blacK*. Согласно этой версии, при [выводе полиграфических плёнок](https://ru.wikipedia.org/w/index.php?title=%D0%92%D1%8B%D0%B2%D0%BE%D0%B4_%D0%BF%D0%BE%D0%BB%D0%B8%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D1%85_%D0%BF%D0%BB%D1%91%D0%BD%D0%BE%D0%BA&action=edit&redlink=1) на них одной буквой указывался цвет, которому они принадлежат. Чёрный не стали обозначать B, чтобы не путать с B ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *blue*) из модели [RGB](https://ru.wikipedia.org/wiki/RGB), а стали обозначать K (по последней букве). Профессиональные цветокорректоры работают с десятью каналами [RGBCMYKLab](https://ru.wikipedia.org/w/index.php?title=RGBCMYKLab&action=edit&redlink=1), используя доступные цветовые пространства. Поэтому при обозначении CMYK как CMYB фраза «манипуляция с каналом B» требовала бы уточнения «манипуляция с каналом B из CMYB», что было бы неудобно.

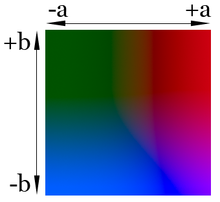
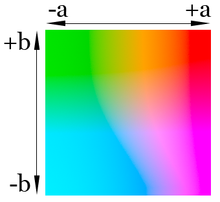
Согласно другому варианту, K является сокращением от слова *ключевой:* в англоязычных странах термином *key plate* обозначается печатная форма для чёрной краски, печатаемая последней поверх заранее напечатанных трёх предыдущих красок.

Третий вариант говорит о немецком происхождении К — [нем.](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%BC%D0%B5%D1%86%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Kontur*. Эта версия подтверждается ещё и тем, что многие старые монтажники так и называют соответствующую плёнку — контур, контурная. Тем более, что в технологии печати чёрный и вправду как бы окантовывает изображение.

**LAB** — аббревиатура названия двух разных (хотя и похожих) [цветовых пространств](https://ru.wikipedia.org/wiki/%D0%A6%D0%B2%D0%B5%D1%82%D0%BE%D0%B2%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%BE). Более известным и распространенным является **CIELAB** (точнее, CIE 1976 **L\*a\*b\***), другим — **Hunter Lab** (точнее, Hunter L, a, b). Таким образом, Lab — это неформальная аббревиатура, не определяющая цветовое пространство однозначно. Чаще всего, говоря о пространстве Lab, подразумевают CIELAB.

При разработке Lab преследовалась цель создания цветового пространства, изменение цвета в котором будет более линейным с точки зрения человеческого восприятия (по сравнению с [XYZ](https://ru.wikipedia.org/wiki/%D0%A6%D0%B2%D0%B5%D1%82%D0%BE%D0%B2%D0%B0%D1%8F_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C#%D0%A6%D0%B2%D0%B5%D1%82%D0%BE%D0%B2%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%BE_CIE_XYZ)), то есть с тем, чтобы одинаковое изменение значений координат цвета в разных областях цветового пространства производило одинаковое ощущение изменения цвета. Таким образом математически корректировалась бы нелинейность восприятия цвета человеком. Оба цветовых пространства рассчитываются относительно определенного значения [точки белого](https://ru.wikipedia.org/wiki/%D0%A2%D0%BE%D1%87%D0%BA%D0%B0_%D0%B1%D0%B5%D0%BB%D0%BE%D0%B3%D0%BE). Если значение точки белого дополнительно не указывается, подразумевается, что значения Lab рассчитаны для стандартного осветителя D50.

В цветовом пространстве Lab значение [светлоты](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D0%B5%D1%82%D0%BB%D0%BE%D1%82%D0%B0_(%D1%86%D0%B2%D0%B5%D1%82)) отделено от значения хроматической составляющей цвета ([тон](https://ru.wikipedia.org/wiki/%D0%A2%D0%BE%D0%BD_(%D1%86%D0%B2%D0%B5%D1%82)), [насыщенность](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D1%81%D1%8B%D1%89%D0%B5%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D1%8C_(%D1%86%D0%B2%D0%B5%D1%82))). Светлота задана координатой L (изменяется от 0 до 100, то есть от самого темного до самого светлого), хроматическая составляющая — двумя декартовыми координатами a и b. Первая обозначает положение цвета в диапазоне от зелено-голубого до красно-малинового, вторая — от голубого до желтого.

Ввиду того что в преобразовании из XYZ в LAB используются формулы, содержащие кубические корни, LAB представляет собой сильно нелинейную систему. Это затрудняет применение привычных операций над 3-мерными векторами в этом цветовом пространстве. Две наиболее широко используемые [формулы цветового различия](https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D1%83%D0%BB%D0%B0_%D1%86%D0%B2%D0%B5%D1%82%D0%BE%D0%B2%D0%BE%D0%B3%D0%BE_%D0%BE%D1%82%D0%BB%D0%B8%D1%87%D0%B8%D1%8F), используемые в программах обработки изображений — CIEDE1976, вычисляемая как расстояние между точками в [евклидовом пространстве](https://ru.wikipedia.org/wiki/%D0%95%D0%B2%D0%BA%D0%BB%D0%B8%D0%B4%D0%BE%D0%B2%D0%BE_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%BE) (квадратный корень из суммы квадратов разностей координат), и [CIEDE2000](https://ru.wikipedia.org/wiki/CIEDE2000), более поздний стандарт, дающий гораздо лучший результат, но в то же время чрезвычайно сложный для вычислений.

## Возможности и ограничения аппаратных средств в представлении цвета. Разрешающая способность устройств ввода и вывода.

Основной характеристикой растровых устройств, предназначенных для вывода (или ввода) изображений, является *разрешение* или *разрешающая способность*.

Для устройств ввода и вывода изображений точки характеризуются строго определёнными размерами, зависящими от возможностей или режимов работы оборудования. На отпечатке растровую сетку формируют пятна краски или тонера, называемые точками. При сканировании изображения или съёмке цифровой фотокамерой растровая сетка соответствует линейке (матрице) светочувствительных элементов. В работе монитора растр зависит от технологических особенностей изготовления электронно-лучевой трубки или жидкокристаллической панели и определяется максимальным количеством отдельных точек, которые можно генерировать на экране.

При всей кажущейся схожести понятий не следует путать разрешение изображения с разрешающей способностью устройств графического ввода и вывода. *Аппаратной разрешающей способностью* графического устройства называется его техническая характеристика, определяющая, сколько точек растровой сетки помещается на единицу длины выводимого изображения (для устройств вывода) или в скольких точках на единицу длины выполняется фиксация характеристик изображения (для устройств ввода).

В связи с различием в понятиях разрешений изображения и устройств ввода-вывода, различается и связанная с ними терминология. В зависимости от устройства, с помощью которого получается изображение, применяются следующие единицы измерения:

**spi**(samples per inc) – отсчётов (элементов) на дюйм;

**dpi**(dots per inch) – точек на дюйм;

**ppi**(pixel per inch) – пикселей на дюйм;

**lpi**(line per inch) – линий на дюйм.

Несмотря на то, что пиксели, отсчёты, точки и линии характеризуют разные аспекты изображения, между ними существует определённая связь. Пиксель остаётся основным элементом (кирпичиком) растровых изображений.

**Экран монитора**

Аппаратная разрешающая способность монитора определяется технологией изготовления его электронно-лучевой трубки или жидкокристаллической панели, содержащей ячейки для воспроизведения пикселей растрового изображения. Большинство офисных мониторов IBM-компьютеров могут работать с различными, но типовыми размерами растровой сетки экрана: 640×480, 800×600, 1024×768, 1152×864 и т. д. Чтобы вычислить размер, который будет иметь изображение на экране, надо знать, сколько пикселей монитора укладывается в единице длины, то есть *разрешение монитора*. Разрешение монитора равно числу пикселей по горизонтали или вертикали, делённому на соответствующий размер видимой части экрана.

Например, стандарт SuрегVGA предполагает, что на экране монитора воспроизводится 800 пикселей по горизонтали и 600 пикселей по вертикали. На мониторе с размером диагонали 14 дюймов это соответствует аппаратной разрешающей способности примерно 72 пикселя на дюйм. Стандартные мониторы платформы Macintosh предоставляют аппаратную разрешающую способность 72 пикселя на дюйм (640 пикселей по горизонтали и 480 по вертикали).

В современных мониторах пользователь имеет возможность выбрать один из нескольких возможных вариантов аппаратной разрешающей способности средствами операционной системы. Аппаратная разрешающая способность монитора никак не связана с разрешением пиксельного изображения, которое на нем воспроизводится. Если численные значения этих величин совпадают, изображение будет воспроизведено на экране в натуральную величину. Однако, если разрешение изображения выше, чем разрешающая способность монитора, размеры изображения на экране будут больше его истинных размеров. По этой причине изображения, предназначенные только для экранного просмотра, рассчитываются на наиболее типичные величины экранных разрешений – 72 и 96 ppi. Большие разрешения делают элементы интерфейса слишком мелкими и трудно различимыми, а меньшие – напротив, слишком крупными и громоздкими.

Предположим, что изображение представляет собой квадрат со стороной 1 дюйм и разрешением 300 ррi. При выводе такого изобра­жения на экран монитора с аппаратным разрешением 72 ррi размеры квадрата будут в четыре с лишним раза больше. Это происходит из-за того, что при выво­де пиксельного изображения на экран каждому пикселю растра изображения ста­вится в соответствие пиксель растра экрана монитора. Поэтому при подготовке изображений для вывода на экран монитора следует руководствоваться следующим правилом: разрешение изображения должно со­ответствовать разрешающей способности экрана.

**Разрешение принтеров и фотонаборных автоматов**

При выводе растрового изображения на лазерный принтер или фотонаборный автомат аппаратная разрешающая способность устройств вывода измеряется в точках на дюйм(dots per inch,dpi).

Точки, которые упоминаются в названии единиц измерения аппаратной разрешающей способности, представляют собой точки принтера. *Точкой принтера* называется область носителя с минимально возможными размерами, которая может быть заполнена частицами тонера.

Надо заметить, что точка принтера может соответствовать пикселю изображения только в одном случае – когда на печать выводится монохромное изображение, содержащее только чёрный и белый цвета. В случае же с полутоновым изображением, перед принтером стоит задача – осуществить печать пикселя серого оттенка. Но принтер не содержит серого тонера. Поэтому, чтобы имитировать печать серым цветом, пиксели полутоновых изображений воспроизводятся на носителе не точками принтера, а более крупными структурными единицами – элементами растра.

*Элементом полиграфического*, или *типографского*,*растра*называют квадратный участок носителя изображения, имеющий фиксированные размеры, часть площади которого (от 0 до 100 %) может быть заполнена частицами тонера. Оттенки серого цвета в различных пикселях изображения имитируются разной степенью заполнения элементов полиграфического растра тонером принтера.

Преобразование исходного растра изображения в полиграфический растр перед выводом на печать называется *растрированием*и выполняется программными средствами. После выполнения процедуры растрирования каждому пикселю изображения ставится в соответствие определённый элемент (ячейка) полиграфического растра.

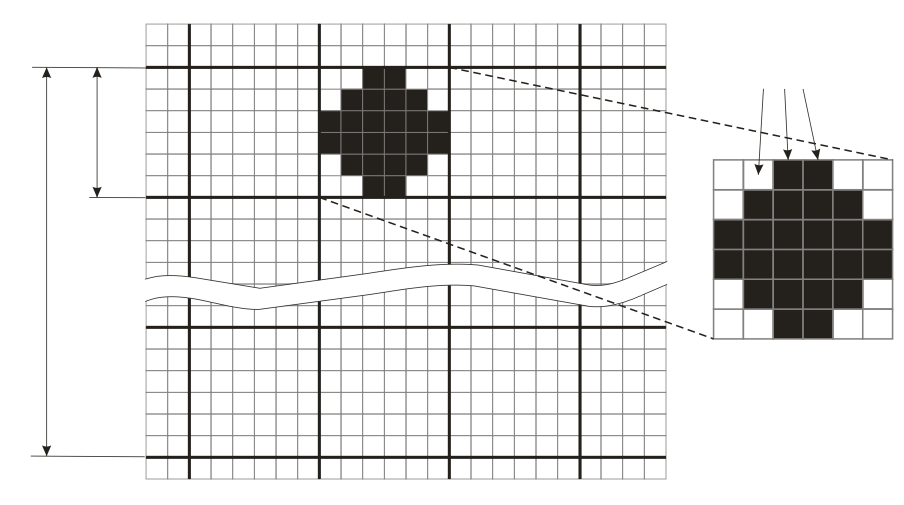
Основной характеристикой типографского растра является *линиатура*(или частота растра), то есть количество составленных из элементов полиграфического растра линий, приходящихся на единицу длины носителя (рис. 17). Линиатура измеряется в линиях на дюйм (line per inch, lpi).

Линиатура растра для печати полутоновых изображений может быть задана произвольно. Ограничено при этом только верхнее её значение, которым является разрешение принтера. Значение линиатуры полиграфического растра обычно определяется с учётом свойств бумаги, качества типографских красок и полиграфического оборудования, на котором предполагается печатать тираж.

Газеты и монохромные издания на бумаге невысокого качества обычно печатаются с линиатурой 75-90 lpi. Для печати журналов выбираются значения линиатуры в диапазоне от 133 до 150 lpi. При печати высококачественной полиграфической продукции выбирают значение линиатуры, превышающее 150 lpi.

Отношение разрешающей способности устройства вывода к линиатуре растра даёт размер стороны ячейки растра, измеряемый в точках принтера. Таким образом, легко можно определить количество точек принтера, составляющих элемент полиграфического растра. Так, например, если линиатура растра установлена равной 100 линий на дюйм, а разрешение принтера 600 точек на дюйм, то сторона ячейки растра равна 600/100 = 6 точкам. При таких условиях элемент растра формируется из 6×6 = 36 точек принтера. Эта величина имеет очень большое значение для адекватной передачи фотографического изображения, потому что именно она определяет количество возможных для передачи оттенков базового цвета.

Количество оттенков базового цвета цветовой модели, которое может быть воспроизведено за счёт разной степени заполнения элементов полиграфического растра, называется *тональной разрешающей способностью*.

Количество оттенков, передаваемое растром, можно определить по формуле

N = (R/L)2+1, (5)

где N – тональная разрешающая способность;

R – разрешение устройства вывода, (dpi);

L – линиатура растра, (lpi).

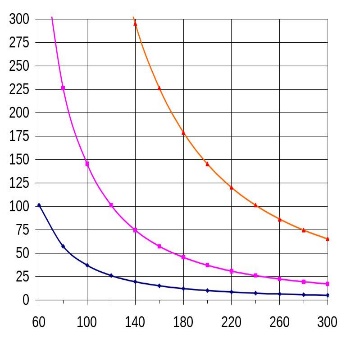
На рисунке 18 в графической форме наглядно представлена зависимость тонального разрешения печатающего устройства от установленного для него значения линиатуры полиграфического растра. Графики зависимостей построены для устройств вывода с наиболее распространёнными значениями пространственного разрешения.

Рис. 18. Графики зависимостей тонального разрешения от линиатуры полиграфического растра и аппаратного разрешения устройства вывода

Как видно из графиков, желание пользователя увеличить значение линиатуры приведёт к снижению тонального разрешения, то есть к уменьшению числа передаваемых принтером оттенков. Следует отметить, что это означает также снижение возможности отображать плавные переходы тона.

Создание изображения в полутоновой модели предполагает передачу 256 градаций серого. Человеческий глаз способен воспринять около 120 оттенков. Если же рассчитать по приведённой формуле (5) количество оттенков, которые способен передать при печати обычный лазерный принтер при линиатуре растра 150lpi, то получится всего лишь (600/150)2+1=17. Даже при уменьшении значения линиатуры до 120lpi, количество передаваемых оттенков будет равно только 26. Этим объясняется посредственное качество фотографий, распечатываемых на офисном лазерном принтере.

Безусловно, при уменьшении линиатуры полиграфического растра будет возрастать количество передаваемых оттенков, и при некотором значении линиатуры тональная разрешающая способность даже при печати офисным принтером может достигнуть вполне приемлемых значений – 120 или даже 256. Однако надо заметить, что с уменьшением линиатуры увеличивается размер растровой ячейки, что делает пиксельную структуру изображения более заметной. Чем выше разрешение устройства ввода (меньше размер точки), тем более подробно оно способно перевести оригинал в цифровую форму. Высокое разрешение устройства вывода позволяет создать более детализированное изображение и сделать растр менее заметным. Из графиков (рис. 18) видно, что практически идеальные параметры вывода обеспечивает фотонаборный автомат с пространственным разрешением 2400dpi. Он способен воспроизвести все 256 тонов базового цвета при линиатуре 150 lpi. В этом случае качество изображения практически полностью определяется линиатурой – чем она выше, тем лучше качество напечатанного изображения.

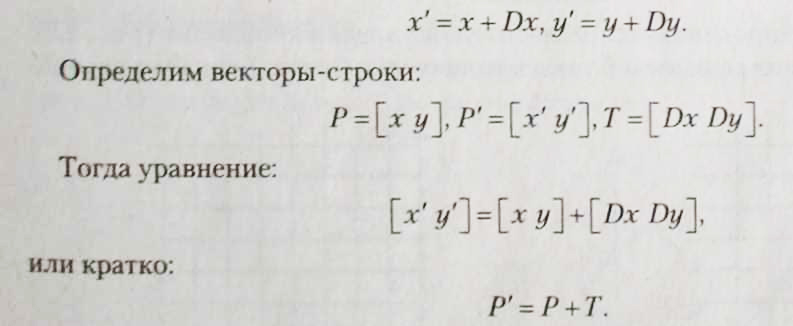
Таким образом, при фиксированном разрешении принтера, компромисс между широким тоновым диапазоном и чёткостью изображения достигается правильным выбором линиатуры печатного растра.

## Двумерные преобразования. Композиция двумерных преобразований.

В основе изменения графической информации лежат три основных преобразо­вания: *перенос*, *масштабирование* и *поворот.* На их основе строятся все известные изменения объектов в графических системах.

**Перенос**

Точки на плоскости можно перенести в новые позиции путем добавления к ко­ординатам этих точек констант переноса. Для каждой точки *Р(х, у*), которая пе­ремещается в новую точку *Р'(х',у')* сдвигаясь на *Dx* единиц по оси *X* и на *Dy* по оси *Y.* можно написать



Но объект представляет собой множество точек. Его можно переносить, при­меняя уравнение переноса к каждой точке. Однако каждый отрезок состоит из бесконечного числа точек, и этот процесс длился бы бесконечно долго. Удобнее перенести все точки, принадлежащие отрезку, путем перемещения одних лишь крайних его точек, а затем вычертить новый отрезок между ними (рис. 2.1).

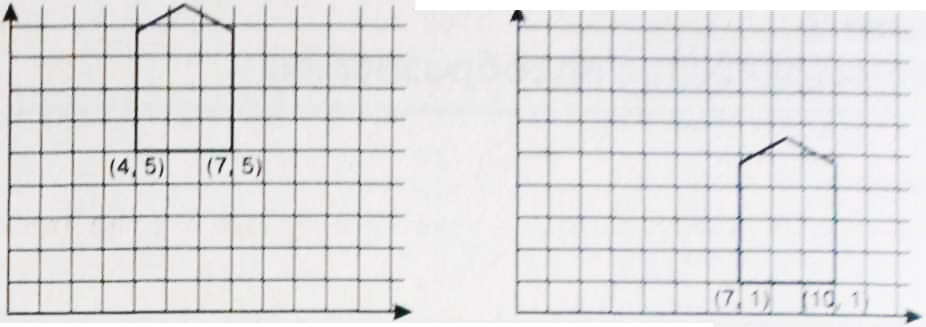


Рис. 2.1. Перенос объекта

# Масштабирование

Точки можно масштабировать (растянуть) в Sx раз по оси X и в Sy раз по оси Y. Получим новые точки с помощью умножения:

x’ = x\*Sx,y’ = y\*Sy

Определить S в виде матрицы:

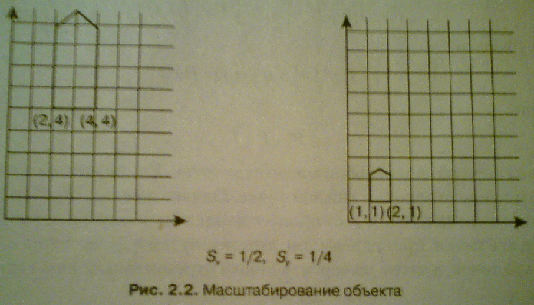
https://studfile.net/html/2706/289/html_hsLdpE7zJx.XXY6/htmlconvd-Z19Nmj_html_7ac91159ee9ff814.gif ,

мhttps://studfile.net/html/2706/289/html_hsLdpE7zJx.XXY6/htmlconvd-Z19Nmj_html_ff991afe4814faf3.png https://studfile.net/html/2706/289/html_hsLdpE7zJx.XXY6/htmlconvd-Z19Nmj_html_9c0a210e3ec3805a.gifожно записать в матричной форме:

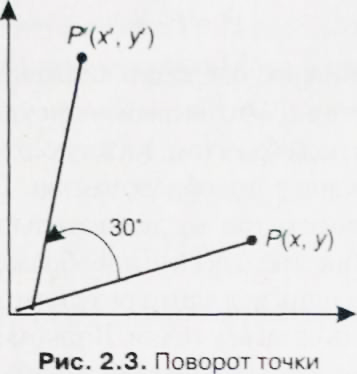
[x’y’] = [xy] \*

или

P’ = P\*S.

М асштабирование производится относительно начала координат (рис. *2.2).*В результате домик стал меньше и ближе к началу координат. Если бы масштабные коэффициенты были больше единицы, домик увеличился бы и отдалился от начала координат. Пропорции домика тоже изменились. Было применено неоднородное масштабирование, при котором Sx ≠ Sy. Однородное масштабирование (Sx = Sy ) не влияет на пропорции.

**Поворот**

Т очки могут быть повернуты на угол α относительно начала координат (рис. 2.3)*.*

Тогда координаты точки Р':

*x’ = xcosΘ – ysinΘ,*

*y’ = xsinΘ + ycosΘ*

В матричной форме:

или P’ = P\*R , где *R* — матрица поворота.

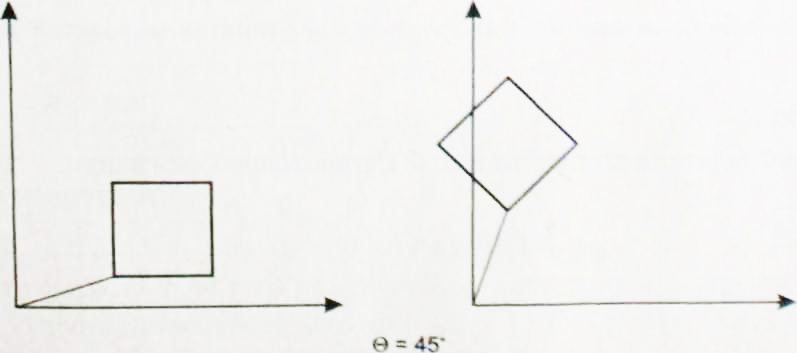
П оложительное значение соответствует повороту против часовой стрелки (рис. 2.4), отрицательное — по часовой стрелке.

Рис. 2.4. Поворот объекта

**Однородные координаты и двумерные преобразования**

Преобразования переноса, масштабирования и поворота в матричной форме имеют вид:

P’ = P + T

P’ = P\*S

P’ = P\*R

Перенос реализуется с помощью операции сложения, а масштабирование и по­ворот — с помощью умножения. Это вызывает неудобство при осуществлении нескольких преобразований над объектом. Каждую точку объекта придется после­довательно подвергать каждому преобразованию. Если объект имеет N точек и необходимо провести, например, три последовательных преобразования, то пона­добится 3 \* N действий. Удобно было бы все преобразования представить в единой форме. Тогда можно было бы один раз найти результирующую матрицу преобразо­вания, а затем лишь умножить ее на все точки. В таком случае количество действий стало бы *N + 2,* где 2 — это две операции по умножению трех матриц. Если выразить точки в однородных координатах, то все три преобразования можно реализовать с помощью операции умножения. И хотя умножение выполняется аппаратно медленнее сложения, и размерность матриц увеличивается на единицу, при большом количестве точек объекта наблюдается ощутимое преимущество.

В однородных координатах точка *Р(х, у*) записывается как *Р( W\*х, W\*у,W*), где*W—* масштабный множитель, не равный нулю.

При этом если точка задана в однородных координатах *Р(Х, Y,*W), то можно найти ее декартовы координаты:

X = https://studfile.net/html/2706/289/html_hsLdpE7zJx.XXY6/htmlconvd-Z19Nmj_html_ffedd9ee867eace3.gif , Y = https://studfile.net/html/2706/289/html_hsLdpE7zJx.XXY6/htmlconvd-Z19Nmj_html_4fccaaeb6da1097a.gif .

Если же W = 1, то операция деления не нужна:

P(x,y,1) , P’(x’,y’,1)

Основные преобразования в однородных координатах выразятся следующим образом.

Перенос

Уhttps://studfile.net/html/2706/289/html_hsLdpE7zJx.XXY6/htmlconvd-Z19Nmj_html_a23e3bca6f2c102e.gif равнение переноса запишется в виде матрицы преобразования:

1 0 0

[x’ y’ 1] = [x y 1] \* 0 1 0

Dx Dy 1

Или P’ = P \* T (Dx , Dy), где

П еремножив, получим:

[x’ y’ 1] = [x + Dx , y + Dy , 1].

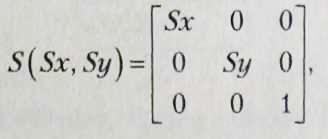
То есть перенос – функция аддитивная.

## М асштабирование

/

[x’ y’ 1] = [x y 1] \*

Уравнение масштабирования в матричной форме имеет вид:

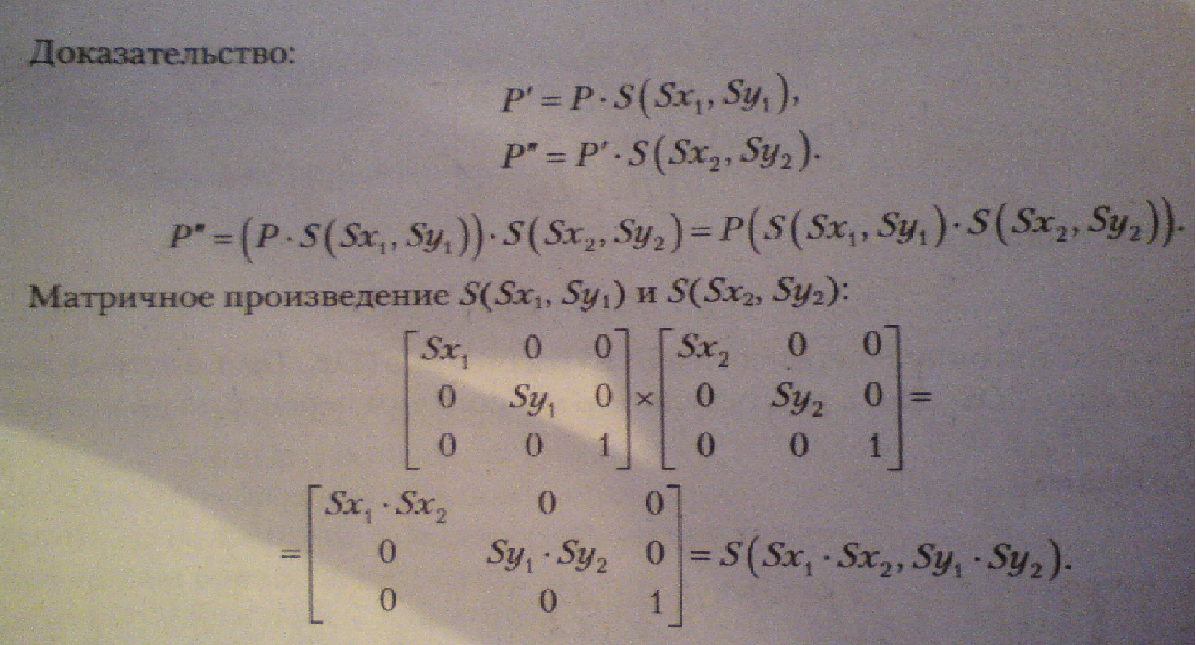
О пределяя

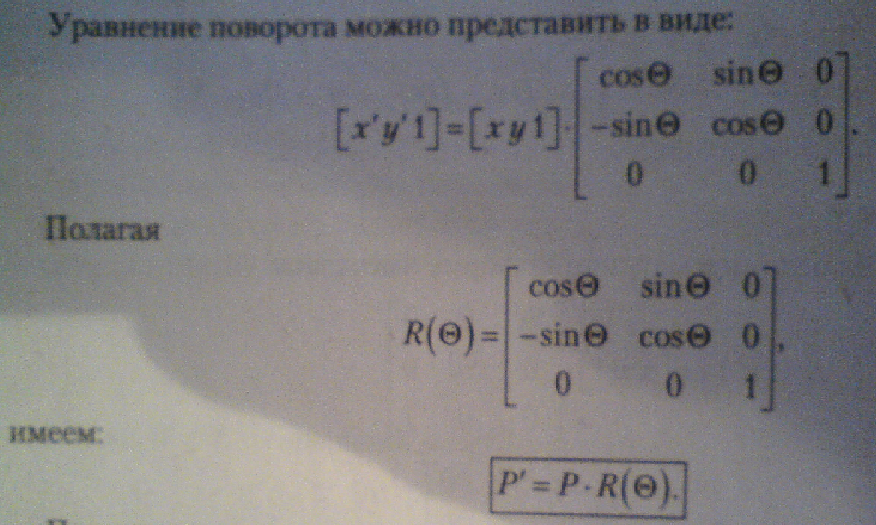
имеем:

*P’ =P\*S(Sx,Sy)*

Перемножив, получим:

[x’ y’ 1] = [x\*Sxy\*Sy1].

Докажем что масштабирование – функция мультипликативная, то есть если точку P(x,y) промасштабировать в точку P’(x’,y’) c S= (Sx1,Sy1), а потом – в точку P’’(x’’,y’’) c S=(Sx2,Sy2), то результат будет иметь вид: S =(Sx1\*Sx2\*Sy1\*Sy2).

С помощью масштабирования легко реализуется осевая симметрия. Для этого используется отрицательные значения коэффициентов масштабирования в матрице.

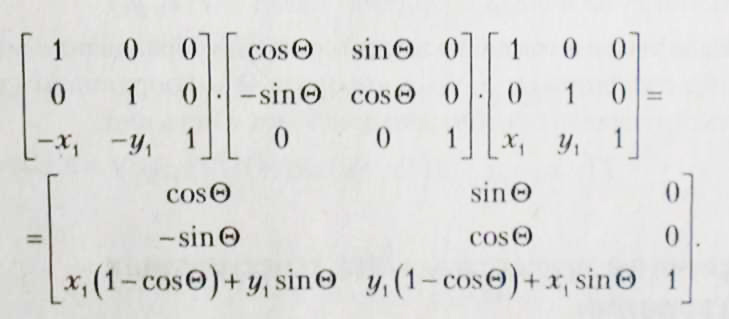
Поворот

Но обычно при работе с графической системой объект подвергается сразу не­скольким преобразованиям. Для получения желаемого результата *используют*композицию преобразований, объединяя матрицы T,S,R*.* К точке более эффек­тивно применять одно результирующее преобразование, чем ряд преобразований последовательно.

Рассмотрим, например, поворот объекта относительно некоторой точки *Р1(х1у1).*

Ранее был рассмотрен поворот относительно начала координат. Для решения этой задачи разобьем ее на три части (три элементарных преобразования) (рис. 2.5):

1. перенос точки *Р1* в начало координат **-** *T(-x1,-y1)*
2. поворот – **R(***Θ***)**
3. перенести точку ***Р1***из начала координат в исходную позицию-*T(x1,y1)*

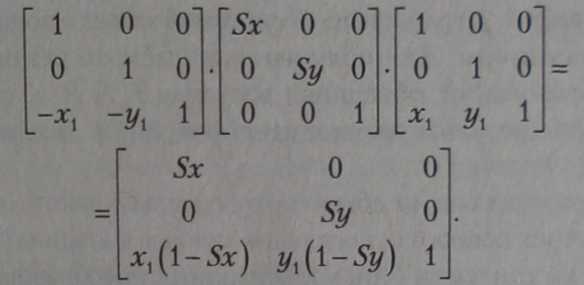
Р https://studfile.net/html/2706/289/html_hsLdpE7zJx.XXY6/htmlconvd-Z19Nmj_html_4a7371da5dcefedb.jpgезультирующее преобразование

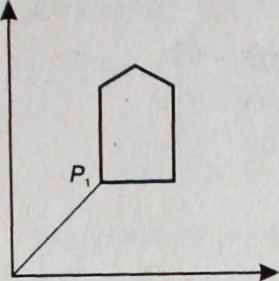
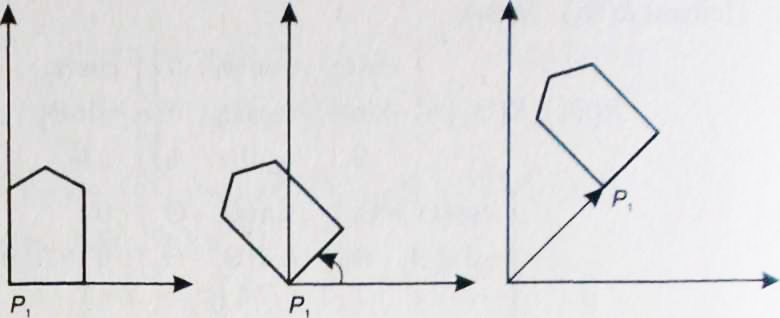
Этот пример хорошо иллюстрирует, как применение однородных координат упрощает задачу.

Аналогично, если нужно отмасштабировать объект относительно точки ***Р1(х1у1****),*а не начала координат, следует:

1. перенести точку *Р1* в начало координат **-** *T(-x1,-y1)*
2. масштабировать *S(Sx,Sy)*
3. перенести точку ***Р1***назад-*T(-x1,-y1)*

Рис. 2.5. Поворот объекта относительно точки P1

 Результат имеет вид

Если нужно масштабировать, повернуть и расположить в нужном месте домик (центром поворота и масштабирования является точка P1), то необходимо выпол­нить:

1. перенос точки P1 в начало координат — ***Т(-x1,-y1)***
2. масштабирование — ***S(Sx, Sy*)**
3. поворот – **R(***Θ***)**
4. перенос точки P1 из начала координат назад **Т(х1,,y1)**

В структуре данных, в которой содержится это преобразование, могут находиться и масштабный коэффициент S, угол поворота *Θ* и координаты (x1,y1). И тогда матрица результирующего преобразования будет иметь вид.

T(-x1,-y1)\*S(Sx,Sy)\*R(Θ)\*T(x1,y1)

## Матричное представление трехмерных преобразований

Аналогично тому, как двумерные преобразования описываются матрицами раз­мером 3x3, трехмерные могут быть представлены в виде матриц 4 х 4. И тогда трех­мерная точка Р(х, у, z) записывается в однородных координатах как P(W \* х, W \* y, W \* z, W), где W ≠ 0. Если же W- 1, то точка представляется в виде Р(х, у, z, 1).

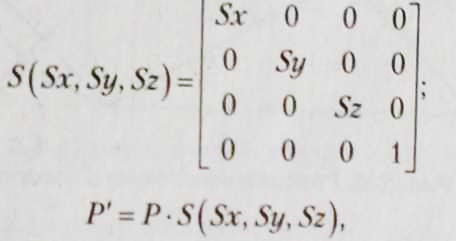
**Перенос**

Трехмерный перенос является простым расширением двумерного:

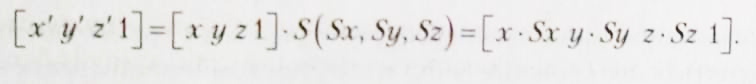


**Масштабирование**

Расширяется аналогичным образом:

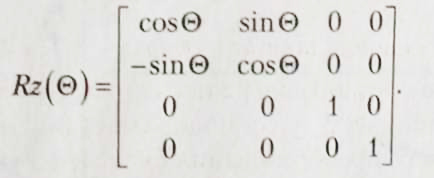


или

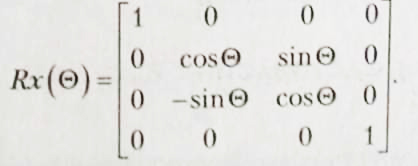


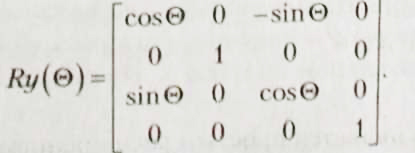
**Поворот**

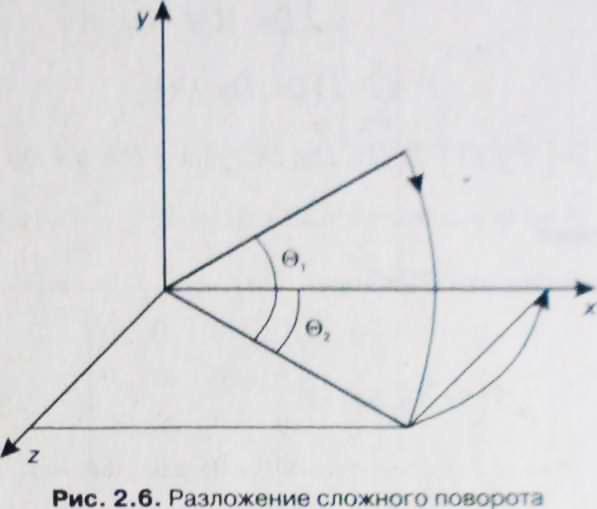
Двумерный поворот, рассмотренный ранее, является в то же время трехмерным поворотом вокруг оси Z:



Матрица поворота вокруг оси X:

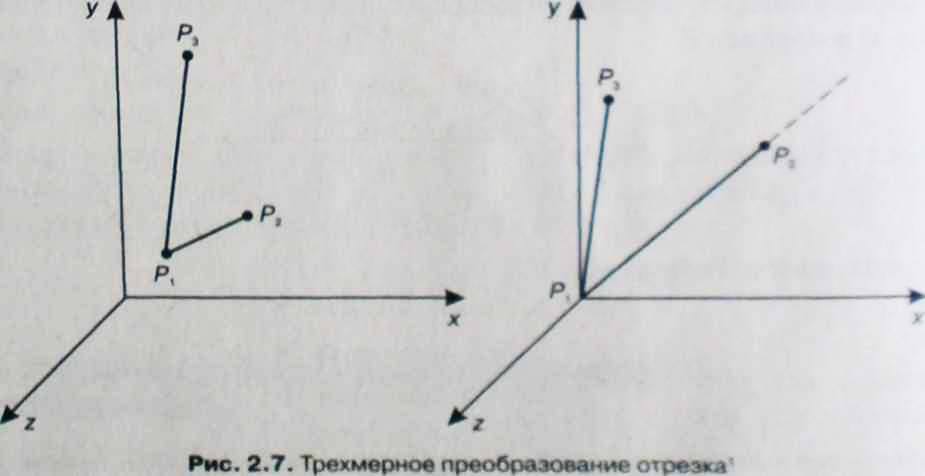


М атрица поворота вокруг оси Y:

1. *Θ* 1- поворот вокруг оси Z до совмещения с плоскостью XZ.
2. *Θ2 -* п оворот вокруг оси Y до совмещения с полуосью X.

## Композиции трехмерных преобразований

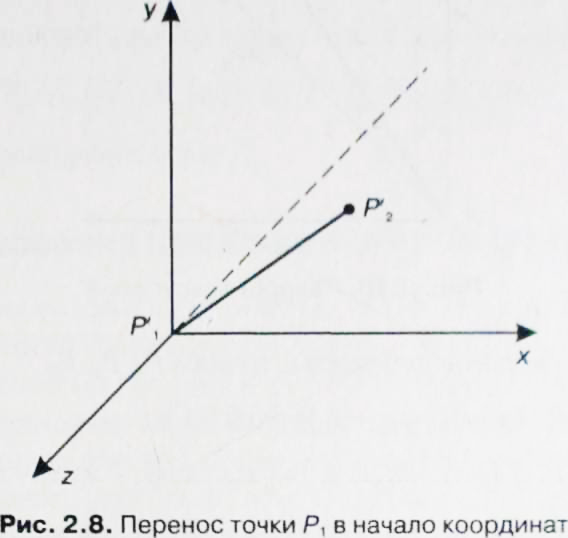
Как и в случае двумерных преобразований, работая с трехмерными можно выполнять более сложные действия путем комбинации элементарных раций. Ниже рассмотрен пример преобразования трехмерного отрезка (рис. 2.7)



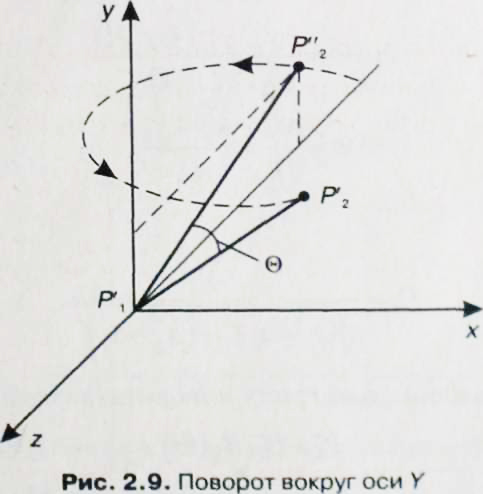
Необходимо преобразовать отрезок P1P2 из начальной позиции в конечную таким образом, чтобы точка Р1 совпала с началом координат, а отрезок P1P2 рас­полагался вдоль отрицательной полуоси Z. На длины отрезков преобразование не воздействует.

Для решения этой задачи следует выполнить три шага:

1. перенос точки Р1 в начало координат (рис. 2.8):

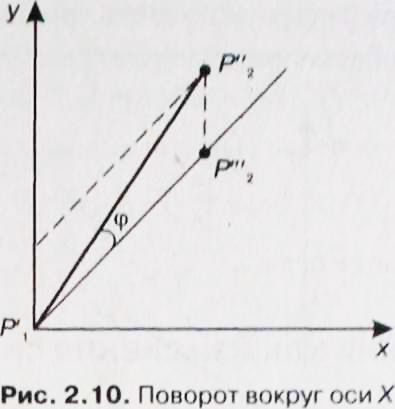
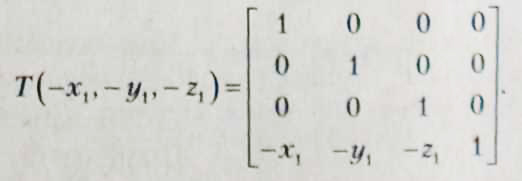


2) поворот вокруг оси У до совмещения отрезка P1P2 с плоскостью YZ (рис. 2.9)



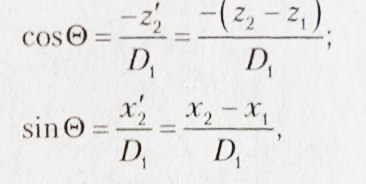
3) поворот вокруг оси X до совмещения отрезка P1P2 с отрицательной полу­осью Z (рис. 2.10).

Матрица преобразования при переносе точки P1 в начало координат имеет вид:

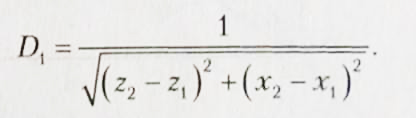
Применим преобразование переноса к точкам P1,P2, P3.



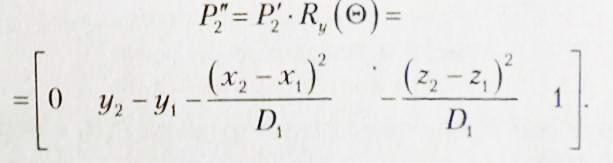
При повороте вокруг оси У на угол Θ (угол положительный) (см. рис. 2.9) опре­деляется:



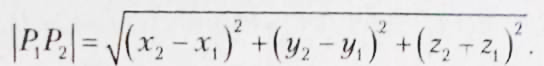
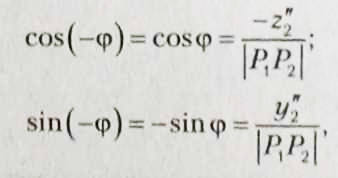
Где



Подставляя эти выражения в матрицу поворота, находим



Поворот вокруг оси X на угол φ (угол отрицательный) (см. рис. 2.10) выража­ется:



где

Общий результат поворота после выполнения трех действий:

## Графические примитивы, математическое описание и возможность отрисовки в программной среде.

## Анимированные изображения. Покадровая анимация.

# Операционные системы

## Понятие операционной системы в узком и широком смысле.

## История развития ОС.

## Мультипрограммные пакетные ОС.

## Принцип модульности. Соглашение о связях в ОС.

## Командный язык ОС. Исполнение команд. Формат команд.

## Управление виртуальной памятью. Страничная, сегментная и сегментно-страничная память.

## Понятие процесса и ресурса. Планирование процессов.

## Дисциплины диспетчеризации процессов.

## Современные ОС для ПК. Отличия и преимущества.

## История развития отечественных ОС

# Защита информации

## Принципы и методы шифрования с открытым ключом (ассиметричного).

**Асимметричное шифрование**— это метод шифрования данных, предполагающий использование двух ключей — открытого и закрытого. Открытый (публичный) ключ применяется для шифрования информации и может передаваться по незащищенным каналам. Закрытый (приватный) ключ применяется для расшифровки данных, зашифрованных открытым ключом. Открытый и закрытый ключи — это очень большие числа, связанные друг с другом определенной функцией, но так, что, зная одно, крайне сложно вычислить второе.

Асимметричное шифрование используется для защиты информации при ее передаче, также на его принципах построена работа электронных подписей.

**Принцип действия асимметричного шифрования**

Схема передачи данных между двумя субъектами (А и Б) с использованием открытого ключа выглядит следующим образом:

* Субъект А генерирует пару ключей, открытый и закрытый (публичный и приватный).
* Субъект А передает открытый ключ субъекту Б. Передача может осуществляться по незащищенным каналам.
* Субъект Б шифрует пакет данных при помощи полученного открытого ключа и передает его А. Передача может осуществляться по незащищенным каналам.
* Субъект А расшифровывает полученную от Б информацию при помощи секретного, закрытого ключа.

В такой схеме перехват любых данных, передаваемых по незащищенным каналам, не имеет смысла, поскольку восстановить исходную информацию возможно только при помощи закрытого ключа, известного лишь получателю и не требующего передачи.

Асимметричные алгоритмы

Наиболее распространенные алгоритмы асимметричного шифрования:

* [RSA](https://encyclopedia.kaspersky.ru/glossary/rsa/) (аббревиатура от Rivest, Shamir и Adelman, фамилий создателей алгоритма) — алгоритм, в основе которого лежит вычислительная сложность факторизации (разложения на множители) больших чисел. Применяется в защищенных протоколах SSL и TLS, стандартах шифрования, например в PGP и S/MIME, и так далее. Используется и для шифрования данных, и для создания цифровых подписей.
* DSA (Digital Signature Algorithm, «алгоритм цифровой подписи») — алгоритм, основанный на сложности вычисления дискретных логарифмов. Используется для генерации цифровых подписей. Является частью стандарта DSS (Digital Signature Standard, «стандарт цифровой подписи»).
* Схема Эль-Гамаля — алгоритм, основанный на сложности вычисления дискретных логарифмов. Лежит в основе DSA и устаревшего российского стандарта ГОСТ 34.10–94. Применяется как для шифрования, так и для создания цифровых подписей.
* ECDSA (Elliptic Curve Digital Signature Algorithm) — алгоритм, основанный на сложности вычисления дискретного логарифма в группе точек эллиптической кривой. Применяется для генерации цифровых подписей, в частности для подтверждения транзакций в криптовалюте Ripple.

Надежность асимметричного шифрования

Теоретически приватный ключ от асимметричного шифра можно вычислить, зная публичный ключ и механизм, лежащий в основе алгоритма шифрования (последнее — открытая информация). Надежными считаются шифры, для которых это нецелесообразно с практической точки зрения. Так, на взлом шифра, выполненного с помощью алгоритма RSA с ключом длиной 768 бит на компьютере с одноядерным процессором AMD Opteron с частотой 2,2 ГГц, бывшем в ходу в середине 2000-х, ушло бы 2000 лет.

При этом фактическая надежность шифрования зависит в основном от длины ключа и сложности решения задачи, лежащей в основе алгоритма шифрования, для существующих технологий. Поскольку производительность вычислительных машин постоянно растет, длину ключей необходимо время от времени увеличивать. Так, в 1977-м (год публикации алгоритма RSA) невозможной с практической точки зрения считалась расшифровка сообщения, закодированного с помощью ключа длиной 426 бит, а сейчас для шифрования этим методом используются ключи от 1024 до 4096 бит, причем первые уже переходят в категорию ненадежных.

Что касается эффективности поиска ключа, то она незначительно меняется с течением времени, но может скачкообразно увеличиться с появлением кардинально новых технологий (например, квантовых компьютеров). В этом случае может потребоваться поиск альтернативных подходов к шифрованию.

## Цифровая подпись.

**Электро́нная по́дпись** (ЭП), **электро́нная цифровая по́дпись** (ЭЦП), **цифровая по́дпись** (ЦП) позволяет подтвердить [авторство](https://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D1%80%D1%81%D1%82%D0%B2%D0%BE) [электронного документа](https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%BD%D1%8B%D0%B9_%D0%B4%D0%BE%D0%BA%D1%83%D0%BC%D0%B5%D0%BD%D1%82) (будь то реальное лицо или, например, аккаунт в [криптовалютной](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D0%B2%D0%B0%D0%BB%D1%8E%D1%82%D0%B0) системе). Подпись связана и с автором, и с самим [документом](https://ru.wikipedia.org/wiki/%D0%94%D0%BE%D0%BA%D1%83%D0%BC%D0%B5%D0%BD%D1%82) с помощью криптографических методов и не может быть подделана с помощью обычного копирования.

ЭЦП — это реквизит электронного документа, полученный в результате криптографического преобразования [информации](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D1%8F) с использованием [закрытого ключа](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BA%D1%80%D1%8B%D1%82%D1%8B%D0%B9_%D0%BA%D0%BB%D1%8E%D1%87) подписи и позволяющий проверить отсутствие искажения информации в электронном документе с момента формирования подписи (целостность), принадлежность подписи владельцу [сертификата ключа подписи](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%82_%D0%BE%D1%82%D0%BA%D1%80%D1%8B%D1%82%D0%BE%D0%B3%D0%BE_%D0%BA%D0%BB%D1%8E%D1%87%D0%B0) (авторство), а в случае успешной проверки подтвердить факт подписания электронного документа (неотказуемость).

**Основные принципы**

Широко применяемая в настоящее время технология электронной подписи основана на [асимметричном шифровании с открытым ключом](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%81_%D0%BE%D1%82%D0%BA%D1%80%D1%8B%D1%82%D1%8B%D0%BC_%D0%BA%D0%BB%D1%8E%D1%87%D0%BE%D0%BC) и опирается на следующие принципы:

* Можно сгенерировать пару очень больших чисел (открытый ключ и закрытый ключ) так, чтобы, зная открытый ключ, нельзя было вычислить закрытый ключ за разумный срок. Механизм генерации ключей строго определён и является общеизвестным. При этом каждому открытому ключу соответствует определённый закрытый ключ. Если, например, Иван Иванов публикует свой открытый ключ, то можно быть уверенным, что соответствующий закрытый ключ есть только у него.
* Имеются надёжные методы шифрования, позволяющие зашифровать сообщение открытым ключом так, чтобы расшифровать его можно было только закрытым ключом[[Прим. 1]](https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BF%D0%BE%D0%B4%D0%BF%D0%B8%D1%81%D1%8C#cite_note-1). Механизм шифрования является общеизвестным.
* Если электронный документ поддается расшифровке с помощью открытого ключа[[Прим. 2]](https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BF%D0%BE%D0%B4%D0%BF%D0%B8%D1%81%D1%8C#cite_note-2), то можно быть уверенным, что он был зашифрован с помощью уникального закрытого ключа. Если документ расшифрован с помощью открытого ключа Ивана Иванова, то это подтверждает его авторство: зашифровать данный документ мог только Иванов, т.к. он является единственным обладателем закрытого ключа.

Однако шифровать весь документ было бы неудобно, поэтому шифруется только его [хеш](https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) — небольшой объём данных, жёстко привязанный к документу с помощью математических преобразований и идентифицирующий его[[Прим. 3]](https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BF%D0%BE%D0%B4%D0%BF%D0%B8%D1%81%D1%8C#cite_note-3). Механизм хеширования строго определён и является общеизвестным. Шифрованный хеш и является электронной подписью.

**Алгоритмы**

Существует несколько схем построения цифровой подписи:

* На основе алгоритмов [симметричного шифрования](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BC%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%87%D0%BD%D1%8B%D0%B5_%D0%BA%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B). Данная схема предусматривает наличие в системе третьего лица — арбитра, пользующегося доверием обеих сторон. Авторизацией документа является сам факт зашифрования его секретным ключом и передача его арбитру.[[4]](https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BF%D0%BE%D0%B4%D0%BF%D0%B8%D1%81%D1%8C#cite_note-%D0%A1%D0%B8%D0%BC%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D0%AD%D0%9F-7)
* На основе алгоритмов [асимметричного шифрования](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%81_%D0%BE%D1%82%D0%BA%D1%80%D1%8B%D1%82%D1%8B%D0%BC_%D0%BA%D0%BB%D1%8E%D1%87%D0%BE%D0%BC). На данный момент такие схемы ЭП наиболее распространены и находят широкое применение.

Кроме того, существуют другие разновидности цифровых подписей (групповая подпись, неоспоримая подпись, доверенная подпись), которые являются модификациями описанных выше схем.[[4]](https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BF%D0%BE%D0%B4%D0%BF%D0%B8%D1%81%D1%8C#cite_note-%D0%A1%D0%B8%D0%BC%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D0%AD%D0%9F-7) Их появление обусловлено разнообразием задач, решаемых с помощью ЭП.

## Дискреционное, мандатное и ролевое разграничение доступа.

**Модели разграничения доступа**

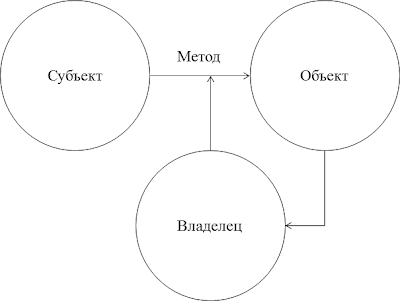


*Разграничение доступа* — совокупность правил, регламентирующих порядок и условия доступа субъекта к [объектам информационной системы](http://www.dataved.ru/2014/02/controlled-access-concepts.html). Также данные правила называют *правами доступа* или *политиками безопасности*. Существуют две основные модели разграничения доступа:

* мандатное разграничение доступа;
* дискреционное (избирательное) разграничение доступа.

**Мандатное разграничение доступа**

В мандатной модели обычные пользователи лишены возможности управлять настройками политик безопасности. Например, возможность доступа к тому или иному объекту определяется уровнем секретности объекта и уровнем доступа пользователя, которые жестко заданы для каждого пользователя и объекта. Данная модель обладает невысокой гибкостью и высокой трудоемкостью настройки политик безопасности, но при этом позволяет достичь высокого уровня управляемости безопасностью.



**Дискреционное разграничение доступа**

*Дискреционное (избирательное, контролируемое) разграничение доступа* — управление доступом субъектов к объектам базируется на том, что пользователи в том или ином объеме могут управлять настройками политик безопасности. Наиболее популярной реализацией дискреционной модели является модель, которая реализует ограничение доступа к файлам и объектам межпроцессной коммуникации в обычных пользовательских представителях семейств операционных систем Unix и Windows. В этих реализациях пользователь может произвольно изменить права доступа к файлу, который он создал, например, сделать его общедоступным.

**Владелец**

Одним из стандартных понятий, появляющихся в стандартной реализации дискреционной модели доступа в рамках файловой системы, является *владелец* объекта — субъект, который несет ответственность за конфиденциальность, целостность и доступность объекта. На владельца возлагается ответственность за корректное ограничение доступа к данному объекту других субъектов, другими словами, имеет возможность предоставить те или иные права доступа к объекту любому другому субъекту. Обычно владельцем объекта автоматически назначается субъект, создавший данный объект. В дальнейшем владелец с помощью соответствующего метода доступа к объекту может быть изменен.

Избирательное управление доступом является базовой реализацией разграничительной политики доступа к ресурсам при обработке конфиденциальных сведений, согласно требованиям к системе защиты информации.

Улучшенные с точки зрения безопасности версии операционных систем ограничивают доступ к файлам и другим объектам с помощью мандатного ограничения доступа. В результате пользователь, даже являясь создателем объекта, не может произвольно менять права доступа к этому файлу. Не гарантируется и то, что после создания пользователь сохранит доступ к созданному объекту, если специально это не оговорено в соответствующей политике безопасности.

**Привилегии**

Другим понятием, возникающим в реализациях дискреционной модели доступа, является понятие *привилегии* на определенный метод доступа. Говорят, что субъект имеет некоторую *привилегию*, если он имеет право на доступ по некоторому методу ко всем объектам, поддерживающим данный метод доступа.

Как правило, небольшая группа привилегированных пользователей — контролирует все процессы и настройки системного уровня. Подобных пользователей называют *суперпользователями*.

**Ролевая модель**

Удобным средством описания политик безопасности является ролевая модель. Ролевая модель представляет собой набор групп, объединяющих пользователей с одинаковым уровнем доступа. Уровень доступа к тому или иному объекту определяется по совокупности групп, которым принадлежит субъект. Например, если пользователь принадлежит группе *Стерилизационное отделение* и группе *Заведующие отделениями*, он обладает полномочиями, предписанными должности заведующего стерилизационным отделением.

Ролевая модель может использоваться для описания политик безопасности как в дискреционной, так и в мандатной модели разграничения доступа.

# Структуры и алгоритмы обработки данных

## Включённые в стандартные библиотеки языков программирования структуры данных. Описание и принцип работы.

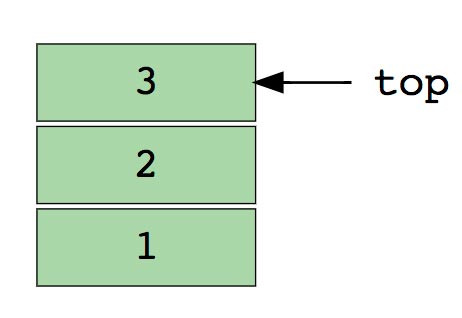
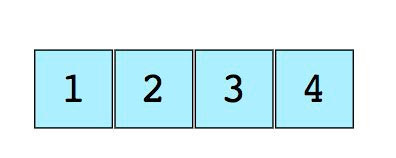
## Какие бывают?

**Линейные**, элементы образуют последовательность или линейный список, обход узлов линеен. Примеры: Массивы. Связанный список, стеки и очереди.  
  
**Нелинейные**, если обход узлов нелинейный, а данные не последовательны. Пример: граф и деревья.

## Основные структуры данных.

1. Массивы
2. Стеки
3. Очереди
4. Связанные списки
5. Графы
6. Деревья
7. Префиксные деревья
8. Хэш таблицы

Массивы

Массив — это самая простая и широко используемая структура данных. Другие структуры данных, такие как стеки и очереди, являются производными от массивов.  
  
Изображение простого массива размера 4, содержащего элементы (1, 2, 3 и 4).  
  
  
  
Каждому элементу данных присваивается положительное числовое значение (индекс), который соответствует позиции элемента в массиве. Большинство языков определяют начальный индекс массива как 0.

Бывают

**Одномерные**, как показано выше.

**Многомерные**, массивы внутри массивов.

Основные операции

* Insert-вставляет элемент по заданному индексу
* Get-возвращает элемент по заданному индексу
* Delete-удаление элемента по заданному индексу
* Size-получить общее количество элементов в массиве

Вопросы

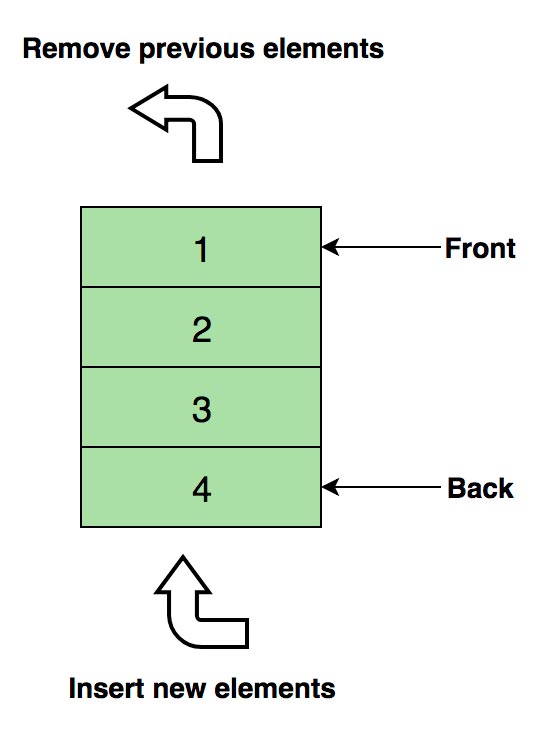
* Найти второй минимальный элемент массива
* Первые неповторяющиеся целые числа в массиве
* Объединить два отсортированных массива
* Изменение порядка положительных и отрицательных значений в массиве

Стеки

Стек — абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»).  
Это не массивы. Это очередь. [Придумал Алан Тюринг.](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA)  
Примером стека может быть куча книг, расположенных в вертикальном порядке. Для того, чтобы получить книгу, которая где-то посередине, вам нужно будет удалить все книги, размещенные на ней. Так работает метод LIFO (Last In First Out). Функция «Отменить» в приложениях работает по LIFO.  
Изображение стека, в три элемента (1, 2 и 3), где 3 находится наверху и будет удален первым.

Основные операции

* Push-вставляет элемент сверху
* Pop-возвращает верхний элемент после удаления из стека
* isEmpty-возвращает true, если стек пуст
* Top-возвращает верхний элемент без удаления из стека



Вопросы

* Реализовать очередь с помощью стека
* Сортировка значений в стеке
* Реализация двух стеков в массиве
* Реверс строки с помощью стека

Очереди

Подобно стекам, очередь — хранит элемент последовательным образом. Существенное отличие от стека – использование FIFO (First in First Out) вместо LIFO.  
Пример очереди – очередь людей. Последний занял последним и будешь, а первый первым ее и покинет.  
Изображение очереди, в четыре элемента (1, 2, 3 и 4), где 1 находится наверху и будет удален первым

Основные операции

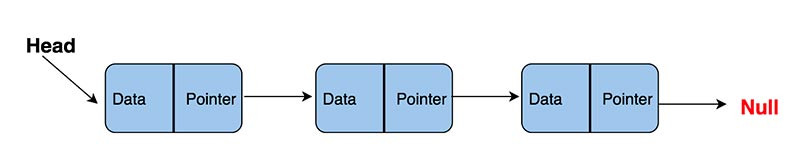
* Enqueue—) — вставляет элемент в конец очереди
* Dequeue () — удаляет элемент из начала очереди
* isEmpty () — возвращает значение true, если очередь пуста
* Top () — возвращает первый элемент очереди

Вопросы

* Реализовать cтек с помощью очереди
* Реверс первых N элементов очереди
* Генерация двоичных чисел от 1 до N с помощью очереди

Связанный список  
Связанный список – массив где каждый элемент является отдельным объектом и состоит из двух элементов – данных и ссылки на следующий узел.  
Принципиальным преимуществом перед массивом является структурная гибкость: порядок элементов связного списка может не совпадать с порядком расположения элементов данных в памяти компьютера, а порядок обхода списка всегда явно задаётся его внутренними связями.

Бывают

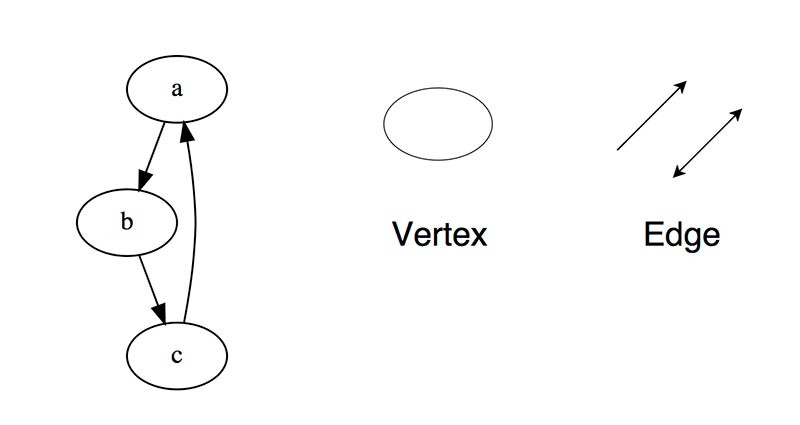
**Однонаправленный**, каждый узел хранит адрес или ссылку на следующий узел в списке и последний узел имеет следующий адрес или ссылку как NULL.  
1->2->3->4->NULL  
**Двунаправленный**, две ссылки, связанные с каждым узлом, одним из опорных пунктов на следующий узел и один к предыдущему узлу.  
NULL<-1<->2<->3->NULL  
**Круговой**, все узлы соединяются, образуя круг. В конце нет NULL. Циклический связанный список может быть одно-или двукратным циклическим связанным списком.  
1->2->3->1  
Самое частое, линейный однонаправленный список. Пример – файловая система.  


Основные операции

* InsertAtEnd — Вставка заданного элемента в конец списка
* InsertAtHead — Вставка элемента в начало списка
* Delete — удаляет заданный элемент из списка
* DeleteAtHead — удаляет первый элемент списка
* Search — возвращает заданный элемент из списка
* isEmpty — возвращает True, если связанный список пуст

Вопросы

* Реверс связанного списка
* Определение цикла в связанном списке
* Возврат N элемента из конца в связанном списке
* Удаление дубликатов из связанного списка

**Графы**  
Граф-это набор узлов (вершин), которые соединены друг с другом в виде сети ребрами (дугами).  


Бывают

**Ориентированный**, ребра являются направленными, т.е. существует только одно доступное направление между двумя связными вершинами.  
**Неориентированные**, к каждому из ребер можно осуществлять переход в обоих направлениях.  
Смешанные

Встречаются в таких формах как

* Матрица смежности
* Список смежности

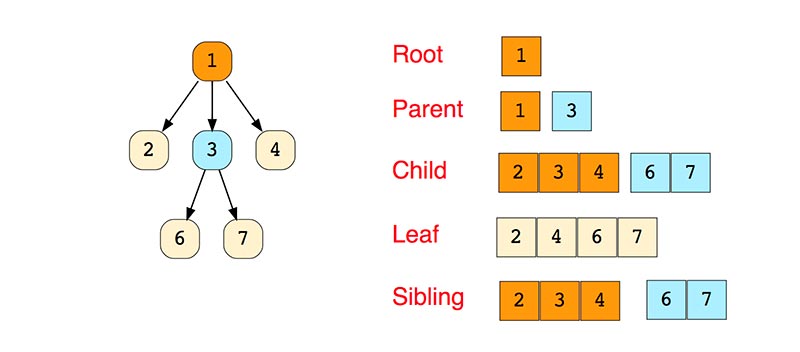
Общие алгоритмы обхода графа

* Поиск в ширину – обход по уровням
* Поиск в глубину – обход по вершинам

Вопросы

* Реализовать поиск по ширине и глубине
* Проверить является ли граф деревом или нет
* Посчитать количество ребер в графе
* Найти кратчайший путь между двумя вершинами

Деревья

Дерево-это иерархическая структура данных, состоящая из узлов (вершин) и ребер (дуг). Деревья по сути связанные графы без циклов.  
Древовидные структуры везде и всюду. Дерево скилов в играх знают все.  
Простое дерево  
Типы деревьев

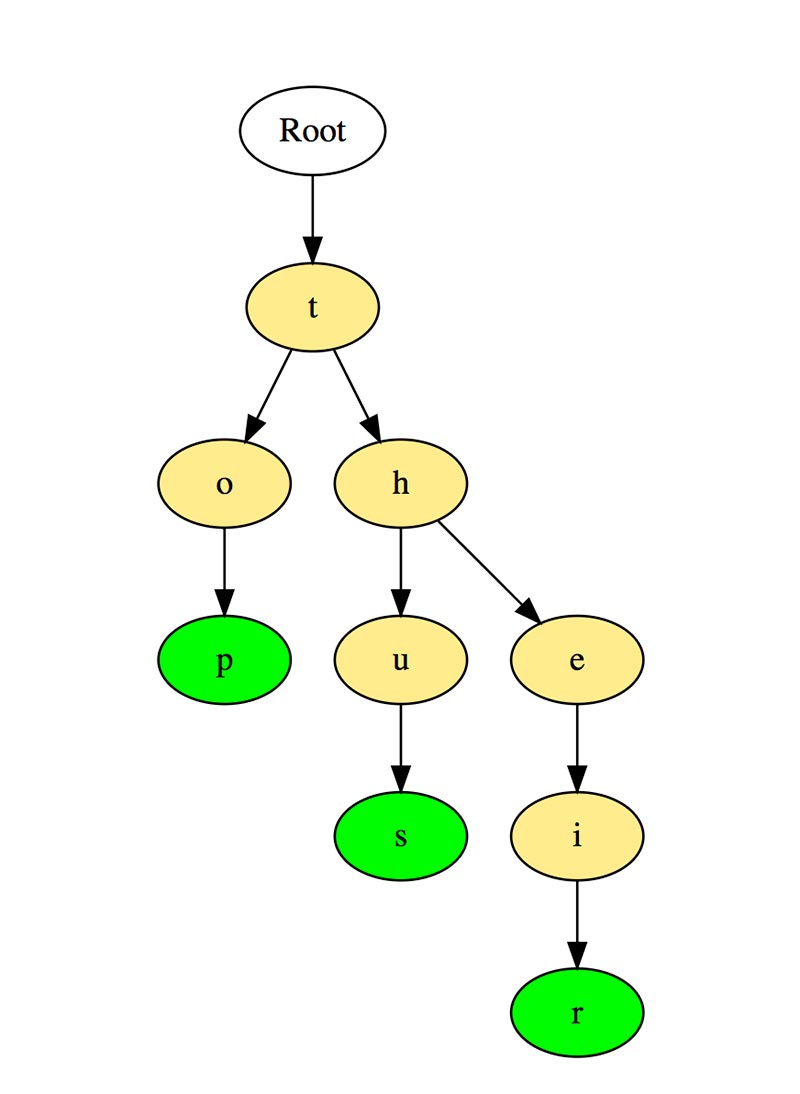
* N дерево
* Сбалансированное дерево
* [Бинарное дерево](https://habr.com/post/267855/)
* Дерево Бинарного Поиска
* [AVL дерево](https://habr.com/post/150732/)
* [2-3-4 деревья](https://habr.com/post/273687/)

Бинарное дерево самое распространенное.  
*«Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение (оно же является в данном случае и ключом) и ссылки на левого и правого потомка. » —*[*Procs*](https://habr.com/users/procs/)

Три способа обхода дерева

* В прямом порядке (сверху вниз) — префиксная форма.
* В симметричном порядке (слева направо) — инфиксная форма.
* В обратном порядке (снизу вверх) — постфиксная форма.

Вопросы

* Найти высоту бинарного дерева
* Найти N наименьший элемент в двоичном дереве поиска
* Найти узлы на расстоянии N от корня
* Найти предков N узла в двоичном дереве

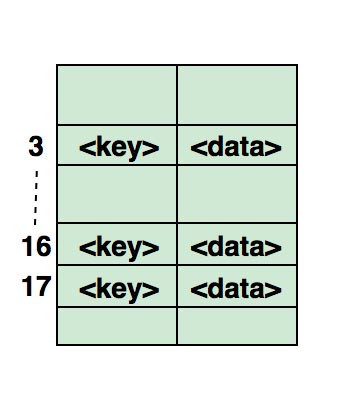
Trie ( префиксное деревое )

Разновидность дерева для строк, быстрый поиск. Словари. Т9.  
Вот как такое дерево хранит слова «top», «thus» и «their». Слова хранятся сверху вниз, зеленые цветные узлы «p», «s» и «r» указывают на конец «top», «thus « и «their» соответственно.

Вопросы

* Подсчитать общее количество слов
* Вывести все слова
* Сортировка элементов массива с префиксного дерева
* Создание словаря T9

Хэш таблицы

Хэширование — это процесс, используемый для уникальной идентификации объектов и хранения каждого объекта в заранее рассчитанном уникальном индексе (ключе).  
Объект хранится в виде пары «ключ-значение», а коллекция таких элементов называется «словарем». Каждый объект можно найти с помощью этого ключа.  
По сути это массив, в котором ключ представлен в виде хеш-функции.  
Эффективность хеширования зависит от  
Функции хеширования

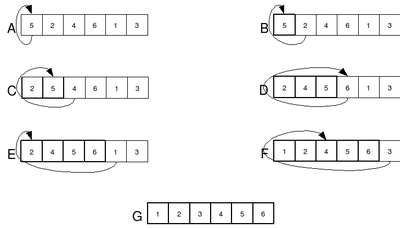
* Размера хэш-таблицы
* Метода борьбы с коллизиями

Пример сопоставления хеша в массиве. Индекс этого массива вычисляется через хэш-функцию.

Вопросы

* Найти симметричные пары в массиве
* Найти, если массив является подмножеством другого массива
* Описать открытое хеширование

## Массивы. Сортировка вставками. Метод «пузырька». Сортировка выбором. Быстрая сортировка.

Сортировка вставками (*Insertion Sort*) — это простой алгоритм сортировки. Суть его заключается в том что, на каждом шаге алгоритма мы берем один из элементов массива, находим позицию для вставки и вставляем. Стоит отметить что массив из 1-го элемента считается отсортированным.

Задача заключается в следующем: есть часть массива, которая уже отсортирована, и требуется вставить остальные элементы массива в отсортированную часть, сохранив при этом упорядоченность. Для этого на каждом шаге алгоритма мы выбираем один из элементов входных данных и вставляем его на нужную позицию в уже отсортированной части массива, до тех пор пока весь набор входных данных не будет отсортирован. Метод выбора очередного элемента из исходного массива произволен, однако обычно (и с целью получения устойчивого алгоритма сортировки), элементы вставляются по порядку их появления во входном массиве.

Так как в процессе работы алгоритма могут меняться местами только соседние элементы, каждый обмен уменьшает число [инверсий](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A2%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0_%D0%B8%D0%BD%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D0%B9) на единицу. Следовательно, количество обменов равно количеству инверсий в исходном массиве вне зависимости от реализации сортировки. Максимальное количество инверсий содержится в массиве, элементы которого отсортированы по невозрастанию. Число инверсий в таком массиве n(n−1)2𝑛(𝑛−1)2.

Алгоритм работает за O(n+k)𝑂(𝑛+𝑘), где k𝑘 — число обменов элементов входного массива, равное числу инверсий. В среднем и в худшем случае — за O(n2)𝑂(𝑛2). Минимальные оценки встречаются в случае уже упорядоченной исходной последовательности элементов, наихудшие — когда они расположены в обратном порядке.

**Псевдокод**

**function** insertionSort(a):

**for** i = 1 **to** n - 1

j = i - 1

**while** j ⩾⩾ 0 **and** a[j] > a[j + 1]

swap(a[j], a[j + 1])

j--

**Пример работы**

Пример работы алгоритма для массива [5,2,4,3,1][5,2,4,3,1]

|  |  |  |
| --- | --- | --- |
| **До** | **После** | **Описание шага** |
| *Первый проход (проталкиваем второй элемент —***2***)* | | |
| **5 2** 4 3 1 | **2 5** 4 3 1 | Алгоритм сравнивает второй элемент с первым и меняет их местами. |
| *Второй проход (проталкиваем третий элемент —***4***)* | | |
| 2 **5 4** 3 1 | 2 **4 5** 3 1 | Сравнивает третий со вторым и меняет местами |
| **2 4** 5 3 1 | **2 4** 5 3 1 | Второй и первый отсортированы, swap не требуется |
| *Третий проход (проталкиваем четвертый —***3***)* | | |
| 2 4 **5 3** 1 | 2 4 **3 5** 1 | Меняет четвертый и третий местами |
| 2 **4 3** 5 1 | 2 **3 4** 5 1 | Меняет третий и второй местами |
| **2 3** 4 5 1 | **2 3** 4 5 1 | Второй и первый отсортированы, swap не требуется |
| *Четвертый проход (проталкиваем пятый элемент —***1***)* | | |
| 2 3 4 **5 1** | 2 3 4 **1 5** | Меняет пятый и четвертый местами |
| 2 3 **4 1** 5 | 2 3 **1 4** 5 | Меняет четвертый и третий местами |
| 2 **3 1** 4 5 | 2 **1 3** 4 5 | Меняет третий и второй местами |
| **2 1** 3 4 5 | **1 2** 3 4 5 | Меняет второй и первый местами. Массив отсортирован. |

**Сортировка простыми обменами**, **сортировка пузырьком** (англ. *bubble sort*) — один из квадратичных алгоритмов сортировки.

**Алгоритм**

Алгоритм состоит в повторяющихся проходах по сортируемому массиву. На каждой итерации последовательно сравниваются соседние элементы, и, если порядок в паре неверный, то элементы меняют местами. За каждый проход по массиву как минимум один элемент встает на свое место, поэтому необходимо совершить не более n−1𝑛−1 проходов, где n𝑛 размер массива, чтобы отсортировать массив.

Ниже приведен псевдокод сортировки пузырьком, на вход которой подается массив a[0..n−1]𝑎[0..𝑛−1].

**function** bubbleSort(a):

**for** i = 0 **to** n - 2

**for** j = 0 **to** n - 2

**if** a[j] > a[j + 1]

swap(a[j], a[j + 1])

**Оптимизация**

* Можно заметить, что после i𝑖-ой итерации внешнего цикла i𝑖 последних элементов уже находятся на своих местах в отсортированном порядке, поэтому нет необходимости производить их сравнения друг с другом. Следовательно, внутренний цикл можно выполнять не до n−2𝑛−2, а до n−i−2𝑛−𝑖−2.
* Также заметим, что если после выполнения внутреннего цикла не произошло ни одного обмена, то массив уже отсортирован, и продолжать что-то делать бессмысленно. Поэтому внутренний цикл можно выполнять не n−1𝑛−1 раз, а до тех пор, пока во внутреннем цикле происходят обмены.

При использовании первой оптимизации сортировка принимает следующий вид:

**function** bubbleSort(a):

**for** i = 0 **to** n - 2

**for** j = 0 **to** n - i - 2

**if** a[j] > a[j + 1]

swap(a[j], a[j + 1])

При использовании же обеих оптимизаций сортировка пузырьком выглядит так:

**function** bubbleSort(a):

i = 0

t = *true*

**while** t

t = *false*

**for** j = 0 **to** n - i - 2

**if** a[j] > a[j + 1]

swap(a[j], a[j + 1])

t = *true*

i = i + 1

**Сложность**

В данной сортировке выполняются всего два различных вида операции: сравнение элементов и их обмен. Поэтому время всего алгоритма T=T1+T2𝑇=𝑇1+𝑇2, где T1𝑇1 — время, затрачиваемое на сравнение элементов, а T2𝑇2 — время, за которое мы производим все необходимые обмены элементов.

Так как в алгоритме меняться местами могут только соседние элементы, то каждый обмен уменьшает количество [инверсий](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A2%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0_%D0%B8%D0%BD%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D0%B9) на единицу. Следовательно, количество обменов равно количеству инверсий в исходном массиве вне зависимости от реализации сортировки. Максимальное количество инверсий содержится в массиве, элементы которого отсортированы по убыванию. Несложно посчитать, что количество инверсий в таком массиве n(n−1)2𝑛(𝑛−1)2. Получаем, что T2=O(n2)𝑇2=𝑂(𝑛2).

В неоптимизированной реализации на каждой итерации внутреннего цикла производятся n−1𝑛−1 сравнений, а так как внутренний цикл запускается также n−1𝑛−1 раз, то за весь алгоритм сортировки производятся (n−1)2(𝑛−1)2 сравнений.

В оптимизированной версии точное количество сравнений зависит от исходного массива. Известно, что худший случай равен n(n−1)2𝑛(𝑛−1)2, а лучший — n−1𝑛−1. Следовательно, T1=O(n2)𝑇1=𝑂(𝑛2).

В итоге получаем T=T1+T2=O(n2)+O(n2)=O(n2)𝑇=𝑇1+𝑇2=𝑂(𝑛2)+𝑂(𝑛2)=𝑂(𝑛2).

**Пример работы алгоритма**

Возьмём массив [5,1,4,2,8][5,1,4,2,8] и отсортируем значения по возрастанию, используя сортировку пузырьком. Выделены те элементы, которые сравниваются на данном этапе.

**Первый проход:**

|  |  |  |
| --- | --- | --- |
| **До** | **После** | **Описание шага** |
| **5 1** 4 2 8 | **1 5** 4 2 8 | Здесь алгоритм сравнивает два первых элемента и меняет их местами. |
| 1 **5 4** 2 8 | 1 **4 5** 2 8 | Меняет местами, так как 5 > 4 |
| 1 4 **5 2** 8 | 1 4 **2 5** 8 | Меняет местами, так как 5 > 2 |
| 1 4 2 **5 8** | 1 4 2 **5 8** | Теперь, ввиду того, что элементы стоят на своих местах (8 > 5), алгоритм не меняет их местами. |

**Второй проход:**

|  |  |  |
| --- | --- | --- |
| **До** | **После** | **Описание шага** |
| **1 4** 2 5 8 | **1 4** 2 5 8 |  |
| 1 **4 2** 5 8 | 1 **2 4** 5 8 | Меняет местами, так как 4 > 2 |
| 1 2 **4 5** 8 | 1 2 **4 5** 8 |  |
| 1 2 4 **5 8** | 1 2 4 **5 8** |  |

Теперь массив полностью отсортирован, но неоптимизированный алгоритм проведет еще два прохода, на которых ничего не изменится, в отличие от алгоритма, использующего вторую оптимизацию, который сделает один проход и прекратит свою работу, так как не сделает за этот проход ни одного обмена.

**Cортировка выбором** (англ. *selection sort*) — простой алгоритм сортировки со сложностью O(n2)𝑂(𝑛2), где n𝑛 — количество элементов для сортировки.

**Алгоритм**

На каждом i𝑖-ом шаге алгоритма находим i𝑖-ый минимальный элемент и меняем его местами с i𝑖-ым элементом в массиве. Таким образом будет получен массив, отсортированный по неубыванию.

**Псевдокод**

**Вариант 1.** Будем каждый раз проходить по всем еще не отсортированным элементам, и, как только найдем элемент меньше, чем первый из неотсортированных, поменяем их местами. Таким образом будет нужно O(n2)𝑂(𝑛2) обменов (для каждого i𝑖 требуется O(n−i)𝑂(𝑛−𝑖) обменов).

**function** selectionSort(**T[n]** a):

**for** i = 0 **to** n - 2

**for** j = i + 1 **to** n - 1

**if** a[i] > a[j]

swap(a[i], a[j])

**Вариант 2.** Второй вариант немного более экономный. Здесь мы будем менять местами элементы только 11 раз для каждого i𝑖, всего будет нужно O(n)𝑂(𝑛) обменов. Для этого сначала мы будем проходить по всем еще не отсортированным элементам, искать минимальный, и только потом менять местами минимальный и первый из неотсортированных.

**function** selectionSort(**T[n]** a):

**for** i = 0 **to** n - 2

min = i

**for** j = i + 1 **to** n - 1

**if** a[j] < a[min]

min = j

swap(a[i], a[min])

**Пример**

Пусть дана последовательность из 55 элементов 5,4,1,2,35,4,1,2,3. Будем выделять текущий элемент на каждом шаге фиолетовым цветом, а минимальный черным жирным.

|  |  |
| --- | --- |
| **Массив** | **Описание шага** |
| *Первый проход (текущий массив начинается с первого элемента)* | | |
| **5** 4 **1** 2 3 | Находим первый минимальный элемент — **1** |  |
| **1** 4 **5** 2 3 | Меняем минимальный и первый элементы местами |  |
| *Второй проход (текущий массив начинается со следующего элемента)* | | |
| 1 **4** 5 **2** 3 | Находим следующий минимальный элемент — **2** |  |
| 1 **2** 5 **4** 3 | Меняем минимальный и второй элементы местами |  |
| *Третий проход (текущий массив начинается со следующего элемента)* | | |
| 1 2 **5** 4 **3** | Находим следующий минимальный элемент — **3** |  |
| 1 2 **3** 4 **5** | Меняем минимальный и третий элементы местами |  |
| *Четвертый проход (текущий массив начинается со следующего элемента)* | | |
| 1 2 3 **4** 5 | Находим следующий минимальный элемент — **4**. Меняем его местами с самим собой. |  |
| 1 2 3 4 5 | Массив отсортирован |  |

**Быстрая сортировка** (англ. *quick sort*, сортировка Хоара) — один из самых известных и широко используемых алгоритмов сортировки. Среднее время работы O(nlogn)𝑂(𝑛log⁡𝑛), что является асимптотически оптимальным временем работы для алгоритма, основанного на сравнении. Хотя время работы алгоритма для массива из n𝑛 элементов в худшем случае может составить Θ(n2)Θ(𝑛2), на практике этот алгоритм является одним из самых быстрых.

**Алгоритм**

Быстрый метод сортировки функционирует по принципу "разделяй и властвуй".

* Массив a[l…r]𝑎[𝑙…𝑟] типа T𝑇 разбивается на два (возможно пустых) подмассива a[l…q]𝑎[𝑙…𝑞] и a[q+1…r]𝑎[𝑞+1…𝑟], таких, что каждый элемент a[l…q]𝑎[𝑙…𝑞] меньше или равен a[q]𝑎[𝑞], который в свою очередь, не превышает любой элемент подмассива a[q+1…r]𝑎[𝑞+1…𝑟]. Индекс вычисляется в ходе процедуры разбиения.
* Подмассивы a[l…q]𝑎[𝑙…𝑞] и a[q+1…r]𝑎[𝑞+1…𝑟] сортируются с помощью рекурсивного вызова процедуры быстрой сортировки.
* Поскольку подмассивы сортируются на месте, для их объединения не требуются никакие действия: весь массив a[l…r]𝑎[𝑙…𝑟] оказывается отсортированным.

**Псевдокод**

**void** quicksort(a: **T**[n], **int** l, **int** r)

**if** l < r

**int** q = partition(a, l, r)

quicksort(a, l, q)

quicksort(a, q + 1, r)

Для сортировки всего массива необходимо выполнить процедуру quicksort(a,0,length[a]−1)quicksort(a,0,length[a]−1).

Разбиение массива

Основной шаг алгоритма сортировки — процедура partitionpartition, которая переставляет элементы массива a[l…r]𝑎[𝑙…𝑟] типа T𝑇 нужным образом. Разбиение осуществляется с использованием следующей стратегии. Прежде всего, в качестве разделяющего элемента произвольно выбирается элемент a[(l+r)/2]𝑎[(𝑙+𝑟)/2]. Далее начинается просмотр с левого конца массива, который продолжается до тех пор, пока не будет найден элемент, превосходящий по значению разделяющий элемент, затем выполняется просмотр, начиная с правого конца массива, который продолжается до тех пор, пока не отыскивается элемент, который по значению меньше разделяющего. Оба элемента, на которых просмотр был прерван, очевидно, находятся не на своих местах в разделенном массиве, и потому они меняются местами. Так продолжаем дальше, пока не убедимся в том, что слева от левого указателя не осталось ни одного элемента, который был бы больше по значению разделяющего, и ни одного элемента справа от правого указателя, которые были бы меньше по значению разделяющего элемента.

Переменная v𝑣 сохраняет значение разделяющего элемента a[(l+r)/2]𝑎[(𝑙+𝑟)/2], a i𝑖 и j𝑗 представляет собой, соответственно, указатели левого и правого просмотра. Цикл разделения увеличивает значение i𝑖 и уменьшает значение j𝑗 на 11, причем условие, что ни один элемент слева от i𝑖 не больше v𝑣 и ни один элемент справа от j𝑗 не меньше v𝑣, не нарушается. Как только значения указателей пересекаются, процедура разбиения завершается.

**int** partition(a: **T**[n], **int** l, **int** r)

**T** v = a[(l + r) / 2]

**int** i = l

**int** j = r

**while** (i ⩽⩽ j)

**while** (a[i] < v)

i++

**while** (a[j] > v)

j--

**if** (i ⩾⩾ j)

**break**

swap(a[i++], a[j--])

**return** j

**Асимптотика**

Худшее время работы

Предположим, что мы разбиваем массив так, что одна часть содержит n−1𝑛−1 элементов, а вторая — 11. Поскольку процедура разбиения занимает время Θ(n)Θ(𝑛), для времени работы T(n)𝑇(𝑛) получаем соотношение:

T(n)=T(n−1)+Θ(n)=∑k=1nΘ(k)=Θ(∑k=1nk)=Θ(n2)𝑇(𝑛)=𝑇(𝑛−1)+Θ(𝑛)=∑𝑘=1𝑛Θ(𝑘)=Θ(∑𝑘=1𝑛𝑘)=Θ(𝑛2).

Мы видим, что при максимально несбалансированном разбиении время работы составляет Θ(n2)Θ(𝑛2). В частности, это происходит, если массив изначально отсортирован.

## Бинарные деревья. Рекурсивный и итеративный методы обхода дерева.

**Двои́чное де́рево** (Бинарное дерево) — иерархическая [структура данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), в которой каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками. Двоичное дерево является упорядоченным [ориентированным деревом](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_(%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2)).

Для практических целей обычно используют два подвида двоичных деревьев — [двоичное дерево поиска](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0) и [двоичная куча](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D0%BA%D1%83%D1%87%D0%B0).

Существует следующее рекурсивное определение двоичного дерева (см. [БНФ](https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D0%B0_%D0%91%D1%8D%D0%BA%D1%83%D1%81%D0%B0_%E2%80%94_%D0%9D%D0%B0%D1%83%D1%80%D0%B0)):

<двоичное дерево> ::= ( <данные> <двоичное дерево> <двоичное дерево> ) | null .

То есть двоичное дерево либо является пустым, либо состоит из данных и двух поддеревьев (каждое из которых может быть пустым). Очевидным, но важным для понимания фактом является то, что каждое поддерево в свою очередь тоже является деревом. Если у некоторого узла оба поддерева пустые, то он называется листовым узлом (листовой вершиной) или конечным (терминальным) узлом[[1]](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE#cite_note-1).

Существует достаточно много алгоритмов работы с древовидными структурами, в которых часто встречается понятие обхода (traversing) дерева или "прохода" по дереву. При таком методе исследования дерева каждый узел посещается только один раз, а полный обход задает линейное упорядочивание узлов, что позволяет упростить алгоритм, так как при этом можно использовать понятие "следующий" узел. т.е. узел стоящий после данного при выбранном порядке обхода.

Существует несколько принципиально разных способов обхода дерева:

Обход в прямом порядке

Каждый узел посещается *до* того, как посещены его потомки.

Для корня дерева рекурсивно вызывается следующая процедура:

Посетить узел

Обойти левое поддерево

Обойти правое поддерево

Примеры использования обхода:

* решение задачи методом деления на части
* разузлование сверху
* стратегия "разделяй и властвуй" (Сортировка Фон Hеймана, быстрая сортировка, одновременное нахождение максимума и минимума последовательности чисел, умножение длинных чисел).

Симметричный обход

Посещаем сначало левое поддерево, затем узел, затем - правое поддерево.

Для корня дерева рекурсивно вызывается следующая процедура:

Обойти левое поддерево

Посетить узел

Обойти правое поддерево

Обход в обратном порядке

Узлы посещаются 'снизу вверх'.

Для корня дерева рекурсивно вызывается следующая процедура:

Обойти левое поддерево

Обойти правое поддерево

Посетить узел

Примеры использования обхода:

* анализ игр с полной информацией
* разузлование снизу
* динамическое программирование

Обход в ширину

При обходе в ширину узлы посещаются уровень за уровнем(N-й *уровень* дерева - множество узлов с высотой N). Каждый уровень обходится слева направо.

Для реализации используется структура queue - очередь с методами

* enqueue - поставить в очередь
* dequeue - взять из очереди
* empty - возвращает TRUE, если очередь пуста, иначе - FALSE

q.enqueue(root); // корень в очередь

while (! q.empty) {

x = q.dequeue();

visit x; // посетить x

if (! x.left.empty) // x.left - левое поддерево

q.enqueue(x.left);

if (! x.right.empty) // x.right - правое поддерево

q.enqueue(x.right);

}

Рекурсивные обходы можно, очевидно, организовать и с помощью стека, 'развернув' рекурсию.

## Хеш-таблицы. Хеширование делением. Хеширование умножением. Универсальное хеширование. Хеширование с разрешением коллизий методом цепочек.

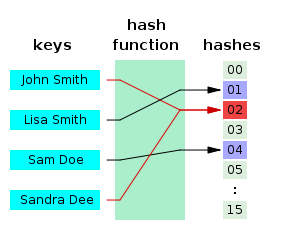
**Хеш-табли́ца** — [структура данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), реализующая интерфейс [ассоциативного массива](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%BE%D1%86%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2), а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию удаления и операцию поиска пары по ключу.

Важное свойство хеш-таблиц состоит в том, что, при некоторых разумных допущениях, все три операции (добавление, удаление, поиск элемента) в среднем выполняются за время 𝑂(1). Но при этом не гарантируется, что время выполнения отдельной операции мало́. Это связано с тем, что при достижении некоторого значения коэффициента заполнения необходимо осуществлять перестройку индекса хеш-таблицы: создать новый массив 𝐻′ увеличенного размера и заново добавить в него все пары из старого массива пар (он же старая хеш-таблица).

Хеш — это какая-то функция, сопоставляющая объектам какого-то множества числовые значения из ограниченного промежутка.

«Хорошая» хеш-функция:

* Быстро считается — за линейное от размера объекта время;
* Имеет не очень большие значения — влезающие в 64 бита;
* «Детерминировано-случайная» — если хеш может принимать 𝑛*n* различных значений, то вероятность того, что хеши от двух случайных объектов совпадут, равна примерно 1𝑛*n*1​.

Обычно хеш-функция не является взаимно однозначной: одному хешу может соответствовать много объектов. Такие функции называют *сюръективными*.

Для некоторых задач удобнее работать с хешами, чем с самими объектами. Пусть даны 𝑛*n* строк длины 𝑚*m*, и нас просят 𝑞*q* раз проверять произвольные две на равенство. Вместо наивной проверки за 𝑂(𝑞⋅𝑛⋅𝑚)*O*(*q*⋅*n*⋅*m*), мы можем посчитать хеши всех строк, сохранить, и во время ответа на запрос сравнивать два числа, а не две строки.

«Хеш-функции», основанные на делении**[**[**править**](https://ru.wikipedia.org/w/index.php?title=%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F&veaction=edit&section=3)**|**[**править код**](https://ru.wikipedia.org/w/index.php?title=%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F&action=edit&section=3)**]**

1. «Хеш-код» как остаток от деления на число всех возможных «хешей»**[**[**править**](https://ru.wikipedia.org/w/index.php?title=%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F&veaction=edit&section=4)**|**[**править код**](https://ru.wikipedia.org/w/index.php?title=%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F&action=edit&section=4)**]**

Хеш-функция может вычислять «хеш» как остаток от деления входных данных на 𝑀:

ℎ(𝑘)=𝑘mod𝑀,

где 𝑀 — количество возможных «хешей» (выходных данных).

При чётном 𝑀 и при чётном 𝑘 значение [функции](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) будет чётным. При чётном 𝑀 и при нечётном 𝑘 значение [функции](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) будет нечётным. Не следует использовать в качестве 𝑀 степень основания [системы счисления](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F) [компьютера](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80), так как «хеш-код» (выходные данные) будет зависеть только от *нескольких* цифр числа 𝑘 (входных данных), расположенных справа, что приведёт к большому количеству [коллизий](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BB%D0%BB%D0%B8%D0%B7%D0%B8%D1%8F_%D1%85%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8). На практике в качестве 𝑀 обычно выбирают [простое число](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%81%D1%82%D0%BE%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE); в большинстве случаев этот выбор вполне удовлетворителен.

2. «Хеш-код» как набор коэффициентов получаемого полинома**[**[**править**](https://ru.wikipedia.org/w/index.php?title=%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F&veaction=edit&section=5)**|**[**править код**](https://ru.wikipedia.org/w/index.php?title=%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F&action=edit&section=5)**]**

Хеш-функция может выполнять деление входных данных на [полином](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BD%D0%BE%D0%BC) по модулю два. Тогда 𝑀 должна являться степенью двойки. Бинарные ключи (𝐾=𝑘𝑛−1𝑘𝑛−2...𝑘0) представляются в виде *полиномов*. В качестве «хеш-кода» «берутся» значения коэффициентов *полинома*, полученного как остаток от деления входных данных 𝐾 на заранее выбранный полином 𝑃 степени 𝑚:

𝐾(𝑥)mod𝑃(𝑥)=ℎ𝑚−1𝑥𝑚−1+⋯+ℎ1𝑥+ℎ0

ℎ(𝑥)=ℎ𝑚−1...ℎ1ℎ0

При правильном выборе 𝑃(𝑥) гарантируется отсутствие коллизий между почти одинаковыми ключами[[4]](https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F#cite_note-_63deeb926b89e866-4).

«Хеш-функции», основанные на умножении**[**[**править**](https://ru.wikipedia.org/w/index.php?title=%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F&veaction=edit&section=6)**|**[**править код**](https://ru.wikipedia.org/w/index.php?title=%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F&action=edit&section=6)**]**

Обозначим символом 𝑤 количество чисел, представимых [машинным словом](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D0%BE%D0%B5_%D1%81%D0%BB%D0%BE%D0%B2%D0%BE). Например, для 32-разрядных [компьютеров, совместимых с IBM PC](https://ru.wikipedia.org/wiki/IBM_PC-%D1%81%D0%BE%D0%B2%D0%BC%D0%B5%D1%81%D1%82%D0%B8%D0%BC%D1%8B%D0%B9_%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80), 𝑤=232.

Выберем некую константу 𝐴 так, чтобы 𝐴 была взаимно простой с 𝑤. Тогда хеш-функция, использующая умножение, может иметь следующий вид:

ℎ(𝐾)=[𝑀⌊𝐴𝑤∗𝐾⌋].

В этом случае на компьютере с двоичной системой счисления 𝑀 является степенью двойки, и ℎ(𝐾) будет состоять из старших битов правой половины произведения 𝐴∗𝐾.

Одним из преимуществ хеш-функций, основанных на делении и умножении, является выгодное использование неслучайности реальных ключей. Например, если ключи представляют собой [арифметическую прогрессию](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D0%B8%D1%84%D0%BC%D0%B5%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B5%D1%81%D1%81%D0%B8%D1%8F) (например, последовательность имён «Имя 1», «Имя 2», «Имя 3»), хеш-функция, использующая умножение, отобразит арифметическую прогрессию в приближённо арифметическую прогрессию различных хеш-значений, что уменьшит количество коллизий по сравнению со случайной ситуацией[[4]](https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F#cite_note-_63deeb926b89e866-4).

Одной из хеш-функций, использующих умножение, является хеш-функция, использующая хеширование [Фибоначчи](https://ru.wikipedia.org/wiki/%D0%A4%D0%B8%D0%B1%D0%BE%D0%BD%D0%B0%D1%87%D1%87%D0%B8). Хеширование [Фибоначчи](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%B0_%D0%A4%D0%B8%D0%B1%D0%BE%D0%BD%D0%B0%D1%87%D1%87%D0%B8) основано на свойствах числа 𝜑 — [золотого сечения](https://ru.wikipedia.org/wiki/%D0%97%D0%BE%D0%BB%D0%BE%D1%82%D0%BE%D0%B5_%D1%81%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5). В качестве константы 𝐴 здесь выбирается целое число, ближайшее к 𝜑−1∗𝑤 и взаимно простое с 𝑤[[4]](https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F#cite_note-_63deeb926b89e866-4).

[**Универсальное** хеширование](https://ru.wikipedia.org/wiki/%D0%A3%D0%BD%D0%B8%D0%B2%D0%B5%D1%80%D1%81%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D1%85%D0%B5%D1%88%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) — хеширование, при котором используется не одна конкретная хеш-функция, а некоторая хеш-функция, выбираемая из заданного семейства хеш-функций по случайному [алгоритму](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC). Обычно отличается малым числом коллизий. Применяется, например, при реализации хеш-таблиц и в криптографии.

**Описание**

Предположим, что требуется отобразить ключи из [пространства](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%BE_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) 𝑈 в числа [𝑚]. На входе алгоритм получает данные из некоторого набора 𝑆∈𝑈 размерностью 𝑛. Набор заранее неизвестен. Как правило, алгоритм должен обеспечить наименьшее число [коллизий](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BB%D0%BB%D0%B8%D0%B7%D0%B8%D1%8F_%D1%85%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8), чего трудно добиться, используя какую-то определённую хеш-функцию. Число коллизий можно уменьшить, если каждый раз при хешировании выбирать хеш-функцию случайным образом. Хеш-функция выбирается из определённого набора хеш-функций, называемого универсальным семейством 𝐻={ℎ:𝑈→[𝑚]}[[7]](https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F#cite_note-Miltersen-7).

## Структуры динамических множеств: стеки, очереди, связанные списки, деревья. Основные операции над элементами множеств.

Рассмотрено ранее

# Технология разработки программного обеспечения

## Классификация моделей процесса создания ПО

* Code and fix — модель кодирования и устранения ошибок;
* Waterfall Model — каскадная модель, или «водопад»;
* V-model — V-образная модель, разработка через тестирование;
* Incremental Model — инкрементная модель;
* Iterative Model — итеративная (или итерационная) модель;
* Spiral Model — спиральная модель;
* Chaos model — модель хаоса;
* Prototype Model — прототипная модель.

Из этих моделей наиболее популярны пять основных: каскадная, V-образная, инкрементная, итерационная и спиральная. Разберём их подробнее.

**Waterfall (каскадная модель, или «водопад»)**

В этой модели разработка осуществляется поэтапно: каждая следующая стадия начинается только после того, как заканчивается предыдущая. Если всё делать правильно, «водопад» будет наиболее быстрой и простой моделью. Применяется уже почти полвека, с 1970-х.



**Преимущества «водопада»**

* *Разработку просто контролировать.* Заказчик всегда знает, чем сейчас заняты программисты, может управлять сроками и стоимостью.
* *Стоимость проекта определяется на начальном этапе.* Все шаги запланированы уже на этапе согласования договора, ПО пишется непрерывно «от и до».
* *Не нужно нанимать тестировщиков с серьёзной технической подготовкой.* Тестировщики смогут опираться на подробную техническую документацию.

Недостатки каскадной модели

Тестирование начинается на последних этапах разработки. Если в требованиях к продукту была допущена ошибка, то исправить её будет стоить дорого. Тестировщики обнаружат её, когда разработчик уже написал код, а технические писатели — документацию.

Заказчик видит готовый продукт в конце разработки и только тогда может дать обратную связь.Велика вероятность, что результат его не устроит.

Разработчики пишут много технической документации, что задерживает работы. Чем обширнее документация у проекта, тем больше изменений нужно вносить и дольше их согласовывать.

«Водопад» подходит для разработки проектов в медицинской и космической отрасли, где уже сформирована обширная база документов (СНиПов и спецификаций), на основе которых можно написать требования к новому ПО.

При работе с каскадной моделью основная задача — написать подробные требования к разработке. На этапе тестирования не должно выясниться, что в них есть ошибка, которая влияет на весь продукт.

**V-образная модель (разработка через тестирование)**

Это усовершенствованная каскадная модель, в которой заказчик с командой программистов одновременно составляют требования к системе и описывают, как будут тестировать её на каждом этапе. История этой модели начинается в 1980-х.

**Преимущества V-образной модели**

* Количество ошибок в архитектуре ПО сводится к минимуму.

**Недостатки V-образной модели**

* Если при разработке архитектуры была допущена ошибка, то вернуться и исправить её будет стоить дорого, как и в «водопаде».

V-модель подходит *для проектов, в которых важна надёжность и цена ошибки очень высока. Например, при разработке подушек безопасности для автомобилей или систем наблюдения за пациентами в клиниках.*

**Incremental Model (инкрементная модель)**

Это модель разработки по частям (increment в переводе с англ. — приращение) уходит корнями в 1930-е. Рассмотрим её на примере создания социальной сети.

1. Заказчик решил, что хочет запустить соцсеть, и написал подробное техническое задание. Программисты предложили реализовать основные функции — страницу с личной информацией и чат. А затем протестировать на пользователях, «взлетит или нет».
2. Команда разработки показывает продукт заказчику и выпускает его на рынок. Если и заказчику, и пользователям социальная сеть нравится, работа над ней продолжается, но уже по частям.
3. Программисты параллельно создают функциональность для загрузки фотографий, обмена документами, прослушивания музыки и других действий, согласованных с заказчиком. Инкремент за инкрементом они совершенствуют продукт, приближаясь к описанному в техническом задании.

Преимущества инкрементной модели

Не нужно вкладывать много денег на начальном этапе. Заказчик оплачивает создание основных функций, получает продукт, «выкатывает» его на рынок — и по итогам обратной связи решает, продолжать ли разработку.

Можно быстро получить фидбэк от пользователей и оперативно обновить техническое задание. Так снижается риск создать продукт, который никому не нужен.

Ошибка обходится дешевле.Если при разработке архитектуры была допущена ошибка, то исправить её будет стоить не так дорого, как в «водопаде» или V-образной модели.

**Недостатки инкрементной модели**

* *Каждая команда программистов разрабатывает свою функциональность и может реализовать интерфейс продукта по-своему.* Чтобы этого не произошло, важно на этапе обсуждения техзадания объяснить, каким он будет, чтобы у всех участников проекта сложилось единое понимание.
* *Разработчики будут оттягивать доработку основной функциональности и «пилить мелочёвку».* Чтобы этого не случилось, менеджер проекта должен контролировать, чем занимается каждая команда.

Инкрементная модель подходит для *проектов, в которых точное техзадание прописано уже на старте, а продукт должен быстро выйти на рынок.*

**Iterative Model (итеративная модель)**

Это модель, при которой заказчик не обязан понимать, какой продукт хочет получить в итоге, и может не прописывать сразу подробное техзадание.

Рассмотрим на примере создания мессенджера, как эта модель работает.

1. Заказчик решил, что хочет создать мессенджер. Разработчики сделали приложение, в котором можно добавить друга и запустить чат на двоих.
2. Мессенджер «выкатили» в магазин приложений, пользователи начали его скачивать и активно использовать. Заказчик понял, что продукт пользуется популярностью, и решил его доработать.
3. Программисты добавили в мессенджер возможность просмотра видео, загрузки фотографий, записи аудиосообщений. Они постепенно улучшают функциональность приложения, адаптируют его к требованиям рынка.

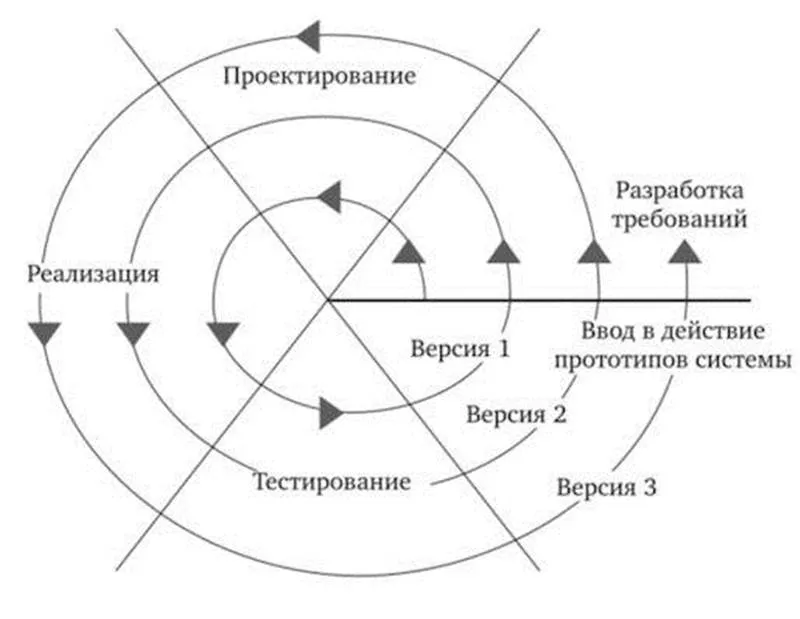
**Преимущества итеративной модели**

* *Быстрый выпуск минимального продукта*даёт возможность оперативно получать обратную связь от заказчика и пользователей. А значит, фокусироваться на наиболее важных функциях ПО и улучшать их в соответствии с требованиями рынка и пожеланиями клиента.
* *Постоянное тестирование пользователями* позволяет быстро обнаруживать и устранять ошибки.

**Недостатки итеративной модели**

* *Использование на начальном этапе баз данных или серверов*— первые сложно масштабировать, а вторые не выдерживают нагрузку. Возможно, придётся переписывать большую часть приложения.
* *Отсутствие фиксированного бюджета и сроков.* Заказчик не знает, как выглядит конечная цель и когда закончится разработка.

Итеративная модель подходит для работы над *большими проектами с неопределёнными требованиями*, либо для задач с *инновационным подходом,*когда заказчик не уверен в результате.

**Spiral Model (спиральная модель)**

Используя эту модель, заказчик и команда разработчиков серьёзно анализируют риски проекта и выполняют его итерациями. Последующая стадия основывается на предыдущей, а в конце каждого витка — цикла итераций — принимается решение, продолжать ли проект. Эту модель начали использовать в 1988 году.

Спиральная модель похожа на инкрементную, но здесь гораздо больше времени уделяется оценке рисков. С каждым новым витком спирали процесс усложняется. Эта модель часто используется в исследовательских проектах и там, где высоки риски.

**Преимущества спиральной модели**

* *Большое внимание уделяется проработке рисков.*

**Недостатки спиральной модели**

* *Есть риск застрять на начальном этапе*— бесконечно совершенствовать первую версию продукта и не продвинуться к следующим.
* *Разработка длится долго и стоит дорого.*

На основе итеративной модели была создана Agile — не модель и не методология, а скорее подход к разработке.

**Что такое Agile?**

Agile («эджайл») переводится с английского как «гибкий». Включает в себя практики, подходы и методологии, которые помогают создавать продукт более эффективно:

* экстремальное программирование (Extreme Programming, XP);
* бережливую разработку программного обеспечения (Lean);
* фреймворк для управления проектами Scrum;
* разработку, управляемую функциональностью (Feature-driven development, FDD);
* разработку через тестирование (Test-driven development, TDD);
* методологию «чистой комнаты» (Cleanroom Software Engineering);
* итеративно-инкрементальный метод разработки (OpenUP);
* методологию разработки Microsoft Solutions Framework (MSF);
* метод разработки динамических систем (Dynamic Systems Development Method, DSDM);
* метод управления разработкой Kanban.

Различия между Agile и традиционным подходом к разработке мы свели в таблице:

## Обработка исключений. Сборки dll.

**Обрабо́тка исключи́тельных ситуа́ций** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *exception handling*) — механизм [языков программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), предназначенный для описания реакции программы на ошибки времени выполнения и другие возможные проблемы (*исключения*), которые могут возникнуть при выполнении программы и приводят к невозможности (бессмысленности) дальнейшей отработки программой её базового алгоритма. В русском языке также применяется более короткая форма термина: «**обработка исключений**».

Во время выполнения программы могут возникать ситуации, когда состояние внешних данных, устройств ввода-вывода или компьютерной системы в целом делает дальнейшие вычисления в соответствии с базовым алгоритмом невозможными или бессмысленными. Классические примеры подобных ситуаций приведены ниже.

* Целочисленное [деление на ноль](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BD%D0%B0_%D0%BD%D0%BE%D0%BB%D1%8C). Конечного результата у данной операции быть не может, поэтому ни дальнейшие вычисления, ни попытка использования результата деления не приведут к решению задачи.
* Ошибка при попытке считать данные с внешнего устройства. Если данные не удаётся получить, любые дальнейшие запланированные операции с ними бессмысленны.
* Исчерпание доступной памяти. Если в какой-то момент система оказывается не в состоянии выделить достаточный для прикладной программы объём оперативной памяти, программа не сможет работать нормально.
* Появление сигнала аварийного отключения электропитания системы. Прикладную задачу, по всей видимости, решить не удастся, в лучшем случае (при наличии какого-то резерва питания) прикладная программа может позаботиться о сохранении данных.

**Виды исключительных ситуаций**

Исключительные ситуации, возникающие при работе программы, можно разделить на два основных типа: синхронные и асинхронные, принципы реакции на которые существенно различаются.

* **Синхронные исключения** могут возникнуть только в определённых, заранее известных точках программы. Так, ошибка чтения файла или коммуникационного канала, нехватка памяти — типичные синхронные исключения, так как возникают они только в операции чтения или в операции выделения памяти соответственно.
* **Асинхронные исключения** могут возникать в любой момент времени и не зависят от того, какую конкретно инструкцию программы выполняет система. Типичные примеры таких исключений: аварийный отказ питания или поступление новых данных.

Некоторые типы исключений могут быть отнесены как к синхронным, так и к асинхронным. Например, инструкция деления на ноль формально должна приводить к синхронному исключению, так как логически оно возникает именно при выполнении данной команды, но на некоторых платформах за счёт глубокой [конвейеризации](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%B2%D0%B5%D0%B9%D0%B5%D1%80_(%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80)) исключение может фактически оказаться асинхронным.

В отсутствие собственного механизма обработки исключений для прикладных программ наиболее общей реакцией на любую исключительную ситуацию является немедленное прекращение выполнения с выдачей пользователю сообщения о характере исключения. Можно сказать, что в подобных случаях единственным и универсальным обработчиком исключений становится операционная система. Например, в операционную систему [Windows](https://ru.wikipedia.org/wiki/Windows) встроена утилита [Dr. Watson](https://ru.wikipedia.org/wiki/Dr._Watson_(%D0%BE%D1%82%D0%BB%D0%B0%D0%B4%D1%87%D0%B8%D0%BA)), которая занимается сбором информации о необработанном исключении и её отправкой на специальный сервер компании [Microsoft](https://ru.wikipedia.org/wiki/Microsoft).

Возможно игнорирование исключительной ситуации и продолжение работы, но такая тактика опасна, так как приводит к ошибочным результатам работы программ или возникновению ошибок впоследствии. Например, проигнорировав ошибку чтения из файла блока данных, программа получит в своё распоряжение не те данные, которые она должна была считать, а какие-то другие. Результаты их использования предугадать невозможно.

Обработка исключительных ситуаций самой программой заключается в том, что при возникновении исключительной ситуации управление передаётся некоторому заранее определённому *обработчику* — блоку кода, процедуре, функции, которые выполняют необходимые действия.

Существует два принципиально разных механизма функционирования обработчиков исключений.

* **Обработка с возвратом** подразумевает, что обработчик исключения ликвидирует возникшую проблему и приводит программу в состояние, когда она может работать дальше по основному алгоритму. В этом случае после того, как выполнится код обработчика, управление передаётся обратно в ту точку программы, где возникла исключительная ситуация (либо на команду, вызвавшую исключение, либо на следующую за ней, как в некоторых старых диалектах языка BASIC) и выполнение программы продолжается. Обработка с возвратом типична для обработчиков асинхронных исключений (которые обычно возникают по причинам, не связанным прямо с выполняемым кодом), для обработки синхронных исключений она малопригодна.
* **Обработка без возврата** заключается в том, что после выполнения кода обработчика исключения управление передаётся в некоторое, заранее заданное место программы, и с него продолжается исполнение. То есть, фактически, при возникновении исключения команда, во время работы которой оно возникло, заменяется на [безусловный переход](https://ru.wikipedia.org/wiki/%D0%91%D0%B5%D0%B7%D1%83%D1%81%D0%BB%D0%BE%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BF%D0%B5%D1%80%D0%B5%D1%85%D0%BE%D0%B4) к заданному оператору.

Существует два варианта подключения обработчика исключительных ситуаций к программе: структурная и неструктурная обработка исключений.

Неструктурная обработка исключений реализуется в виде механизма регистрации [функций](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) или команд-обработчиков для каждого возможного [типа](https://ru.wikipedia.org/wiki/%D0%A2%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) исключения. [Язык программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) или его системные [библиотеки](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) предоставляют программисту как минимум две стандартные процедуры: регистрации обработчика и разрегистрации обработчика. Вызов первой из них «привязывает» обработчик к определённому исключению, вызов второй — отменяет эту «привязку». Если исключение происходит, выполнение основного кода программы немедленно прерывается и начинается выполнение обработчика. По завершении обработчика управление передаётся либо в некоторую наперёд заданную точку программы, либо обратно в точку возникновения исключения (в зависимости от заданного способа обработки — с возвратом или без). Независимо от того, какая часть программы в данный момент выполняется, на определённое исключение всегда реагирует последний зарегистрированный для него обработчик. В некоторых языках зарегистрированный обработчик сохраняет силу только в пределах текущего блока кода (процедуры, функции), тогда процедура разрегистрации не требуется.

Структурная обработка исключений требует обязательной поддержки со стороны [языка программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) — наличия специальных [синтаксических](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) конструкций. Такая конструкция содержит [блок](https://ru.wikipedia.org/wiki/%D0%91%D0%BB%D0%BE%D0%BA_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) контролируемого кода и обработчик (обработчики) исключений. Наиболее общий вид такой конструкции (условный):

НачалоБлока

... // Контролируемый код

...

если (условие) то СоздатьИсключение Исключение2

...

Обработчик Исключение1

... // Код обработчика для Исключения1

Обработчик Исключение2

... // Код обработчика для Исключения2

ОбработчикНеобработанных

... // Код обработки ранее не обработанных исключений

КонецБлока

Здесь «НачалоБлока» и «КонецБлока» — [ключевые слова](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D1%80%D0%B5%D0%B7%D0%B5%D1%80%D0%B2%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D1%81%D0%BB%D0%BE%D0%B2%D0%BE), которые ограничивают блок контролируемого кода, а «Обработчик» — начало блока обработки соответствующего исключения. Если внутри блока, от начала до первого обработчика, произойдёт исключение, то произойдёт переход на обработчик, написанный для него, после чего весь блок завершится и исполнение будет продолжено со следующей за ним команды. В некоторых языках нет специальных ключевых слов для ограничения блока контролируемого кода, вместо этого обработчик (обработчики) исключений могут быть встроены в некоторые или во все синтаксические конструкции, объединяющие несколько операторов. Так, например, в языке Ада любой составной оператор (begin — end) может содержать обработчик исключений.

«ОбработчикНеобработанных» — это обработчик исключений, которые не соответствуют ни одному из описанных выше в данном блоке. Обработчики исключений в реальности могут описываться по-разному (один обработчик на все исключения, по одному обработчику на каждый тип исключения), но принципиально они работают одинаково: при возникновении исключения находится первый соответствующий ему обработчик в данном блоке, его код выполняется, после чего выполнение блока завершается. Исключения могут возникать как в результате программных ошибок, так и путём явной их генерации с помощью соответствующей команды (в примере — команда «СоздатьИсключение»). С точки зрения обработчиков такие искусственно созданные исключения ничем не отличаются от любых других.

Блоки обработки исключений могут многократно входить друг в друга, как явно (текстуально), так и неявно (например, в блоке вызывается процедура, которая сама имеет блок обработки исключений). Если ни один из обработчиков в текущем блоке не может обработать исключение, то выполнение данного блока немедленно завершается, и управление передаётся на ближайший подходящий обработчик более высокого уровня иерархии. Это продолжается до тех пор, пока обработчик не найдётся и не обработает исключение или пока исключение не выйдет из обработчиков заданных программистом и не будет передано системному обработчику по умолчанию, аварийно закроющему программу.

Иногда бывает неудобно завершать обработку исключения в текущем блоке, то есть желательно, чтобы при возникновении исключения в текущем блоке обработчик выполнил какие-то действия, но исключение продолжило бы обрабатываться на более высоком уровне (обычно так бывает, когда обработчик данного блока не полностью обрабатывает исключение, а лишь частично). В таких случаях в обработчике исключений генерируется новое исключение или возобновляется, с помощью специальной команды, ранее появившееся. Код обработчиков не является защищённым в данном блоке, поэтому созданное в нём исключение будет обрабатываться в блоках более высокого уровня.

Помимо блоков контролируемого кода для обработки исключений, языки программирования могут поддерживать блоки с гарантированным завершением. Их использование оказывается удобным тогда, когда в некотором блоке кода, независимо от того, произошли ли какие-то ошибки, необходимо перед его завершением выполнить определённые действия. Простейший пример: если в процедуре динамически создаётся какой-то локальный объект в памяти, то перед выходом из этой процедуры объект должен быть уничтожен (чтобы избежать утечки памяти), независимо от того, произошли после его создания ошибки или нет. Такая возможность реализуется блоками кода вида:

НачалоБлока

... // Основной код

Завершение

... // Код завершения

КонецБлока

Область действия обработчиков начинается специальным ключевым словом try или просто языковым маркером начала блока (например, begin) и заканчивается перед описанием обработчиков (catch, except, resque). Обработчиков может быть несколько, один за одним, и каждый может указывать тип исключения, который он обрабатывает. Как правило, никакого подбора наиболее подходящего обработчика не производится, и выполняется первый же обработчик, совместимый по типу с исключением. Поэтому порядок следования обработчиков имеет важное значение: если обработчик, совместимый с многими или всеми типами исключений, окажется в тексте прежде специфических обработчиков для конкретных типов, то специфические обработчики не будут использоваться вовсе.

Некоторые языки также допускают специальный блок (else), который выполняется, если ни одного исключения не было сгенерировано в соответствующей области действия. Чаще встречается возможность гарантированного завершения блока кода (finally, ensure). Заметным исключением является Си++, где такой конструкции нет. Вместо неё используется автоматический вызов [деструкторов](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80) объектов. Вместе с тем существуют нестандартные расширения Си++, поддерживающие и функциональность finally (например в [MFC](https://ru.wikipedia.org/wiki/MFC)).

**Сборка**

Сборки создаются в виде исполняемого файла (EXE) или файла библиотеки динамической компоновки (DLL) и являются стандартными блоками приложений .NET. Они предоставляют сведения для среды CLR, которые нужны для распознавания реализаций типов.

Когда мы создаем приложение в результате компиляции в Visual Studio или в консоли, результатом этой работы является файл exe или dll (в зависимости от выбранных настроек), который называется сборкой приложения. **Сборка** является базовой структурной единицей в .NET, на уровне которой проходит контроль версий, развертывание и конфигурация приложения.

Сборки кристаллизуют всю библиотеку классов .NET - при написании кода и создании сборки своего приложения мы используем пространства имен, которые размещены в других сборках .NET.

Сборки имеют следующие составляющие:

* Манифест, который содержит метаданные сборки
* Метаданные типов. Используя эти метаданные, сборка определяет местоположение типов в файле приложения, а также места размещения их в памяти
* Собственно код приложения на языке MSIL, в который компилируется код C#
* Ресурсы

Все эти компоненты могут находиться в одном файле, и тогда сборка представляет один единственный файл в формате exe или dll.

**Манифест сборки**

Ключевым компонентом сборки является ее **манифест**. Если у сборки отсутствует манифест, то заключенный в ней код MSIL выполняться не будет. Манифест может находиться в одном файле с исполняемым кодом сборки, а может размещаться и в отдельном файле.

Манифест хранит следующие данные:

* **Имя сборки**
* **Номер версии**: основной и дополнительный номера. Используется для управления версиями
* **Язык и региональные параметры**: информация о языке и региональных параметрах, которые поддерживает сборка
* **Информация о строгом имени**: открытый ключ издателя
* **Список всех файлов сборки**: хэш и имя каждого из входящих в сборку файлов
* **Список ссылок на другие сборки**, которые использует текущая сборка
* **Список ссылок на типы**, используемые сборкой

Таким образом, манифест позволяет системе определить все файлы, входящие в сборку, сопоставить ссылки на типы, ресурсы, сборки с их файлами, управлять контролем версий.

## Проектирование информационных систем и ПО средствами UML.

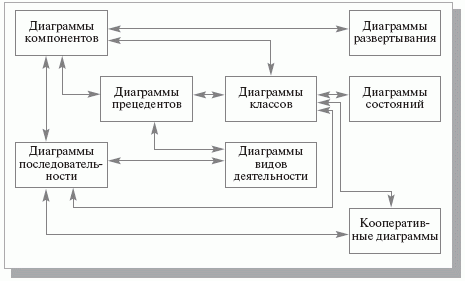
*UML* обеспечивает поддержку всех *этапов жизненного цикла* ИС и предоставляет для этих целей ряд графических средств – диаграмм.

На этапе создания *концептуальной модели* для описания бизнес-деятельности используются *модели бизнес-прецедентов* и диаграммы видов деятельности, для описания *бизнес-объектов* – *модели бизнес-объектов* и *диаграммы последовательностей*.

На этапе создания логической модели ИС описание требований к системе задается в виде модели и описания системных *прецедентов*, а предварительное проектирование осуществляется с использованием *диаграмм классов*, диаграмм последовательностей и диаграмм состояний.

На этапе создания *физической модели* детальное проектирование выполняется с использованием *диаграмм классов*, *диаграмм компонентов*, *диаграмм развертывания*.

Ниже приводятся определения и описывается назначение перечисленных диаграмм и моделей применительно к задачам *проектирования ИС* (в скобках приведены *альтернативные* названия диаграмм, использующиеся в современной литературе).

*Диаграммы прецедентов* (*диаграммы вариантов использования*, *use case* diagrams) – это обобщенная модель функционирования системы в окружающей среде.

Диаграммы видов деятельности (диаграммы деятельностей, *activity diagrams*) – модель бизнес-процесса или поведения системы в рамках *прецедента*.

*Диаграммы взаимодействия* (*interaction* diagrams) – модель процесса обмена сообщениями между объектами, представляется в виде диаграмм последовательностей (*sequence diagrams*) или кооперативных диаграмм (*collaboration* diagrams).

*Диаграммы состояний* (statechart diagrams) – модель динамического поведения системы и ее компонентов при переходе из одного состояния в другое.

*Диаграммы классов* (*class diagrams*) – *логическая модель* базовой структуры системы, отражает статическую структуру системы и связи между ее элементами.

Диаграммы *базы данных* (*database* diagrams) — модель *структуры базы данных*, отображает таблицы, столбцы, ограничения и т.п.

Диаграммы компонентов (*component* diagrams) – модель иерархии подсистем, отражает физическое *размещение* баз данных, приложений и интерфейсов ИС.

Диаграммы развертывания (диаграммы размещения, *deployment* diagrams) – модель физической архитектуры системы, отображает аппаратную конфигурацию ИС.

На [рис. 12.1](https://intuit.ru/studies/courses/2195/55/lecture/1640?page=1#image.12.1) показаны отношения между различными видами диаграмм *UML*. Указатели стрелок можно интерпретировать как *отношение* "является источником входных данных для..." (например, *диаграмма* *прецедентов* является источником данных для диаграмм видов деятельности и последовательности). Приведенная схема является наглядной иллюстрацией итеративного характера разработки моделей с использованием *UML*.

## Системы управления версиями ПО.

Системы управления версиями — это программное обеспечение, помогающее отслеживать изменения в коде с течением времени. Когда разработчик редактирует код, система управления версиями создает моментальный снимок файлов. Затем этот моментальный снимок сохраняется, чтобы при необходимости им было можно воспользоваться позже.

Без управления версиями разработчики заманчивы хранить несколько копий кода на своем компьютере. Это опасно, так как легко изменить или удалить файл в неправильной копии кода, потенциально потеряв работу. Системы управления версиями решают эту проблему, управляя всеми версиями кода, но предоставляя команде одну версию одновременно.

**Почему управление версиями имеет значение**

Есть много вещей, которые могут занять время в качестве разработчика. Воспроизведение ошибок, обучение новых инструментов и добавление новых функций или содержимого всего лишь в нескольких примерах. По мере увеличения масштаба требований пользователей управление версиями помогает командам работать вместе и отправлять их вовремя.

**Преимущества управления версиями**

Управление версиями обеспечивает множество аспектов рабочей среды.

**Создание бизнес-правил**

Рабочие процессы управления версиями препятствуют хаосу всех пользователей, использующих собственный процесс разработки с различными и несовместимыми инструментами. Системы управления версиями предоставляют принудительное применение и разрешения, чтобы все оставались на одной странице.

**Использование версий**

Каждая версия содержит описание изменений в версии, например исправление ошибки или добавление функции. Эти описания помогают команде следовать изменениям кода по версии вместо отдельных изменений в файле. Код, хранящийся в версиях, можно в любой момент при необходимости просмотреть и восстановить из системы управления версиями. Версии упрощают создание новой версии кода.

**Код вместе**

Управление версиями синхронизирует версии и гарантирует, что изменения не конфликтуют с изменениями от других пользователей. Команда полагается на управление версиями, чтобы помочь устранить и предотвратить конфликты, даже если люди вносят изменения одновременно.

**Сохранение журнала**

Управление версиями сохраняет журнал изменений, так как команда сохраняет новые версии кода. Участники группы могут просмотреть историю, чтобы узнать, кто, почему и когда были внесены изменения. История дает командам уверенность в эксперименте, так как легко откатиться к предыдущей хорошей версии в любое время. Журнал позволяет любому пользователю работать с базовой версией кода, например исправить ошибку в предыдущем выпуске.

**Автоматизация задач**

Функции автоматизации управления версиями экономят время и создают согласованные результаты. Автоматизация тестирования, анализа кода и развертывания при сохранении новых версий в элементе управления версиями является тремя примерами.

**Ежедневный цикл работы**

При некоторых вариациях, определяемых особенностями системы и деталями принятого технологического процесса, обычный цикл работы разработчика в течение рабочего дня выглядит следующим образом.

**Обновление рабочей копии**

По мере внесения изменений в основную версию проекта рабочая копия на компьютере разработчика стареет: расхождение её с основной версией проекта увеличивается. Это повышает риск возникновения конфликтных изменений (см. [ниже](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F_%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%8F%D0%BC%D0%B8#%D0%9A%D0%BE%D0%BD%D1%84%D0%BB%D0%B8%D0%BA%D1%82_%D0%B8%D0%B7%D0%BC%D0%B5%D0%BD%D0%B5%D0%BD%D0%B8%D0%B9)). Поэтому удобно поддерживать рабочую копию в состоянии, максимально близком к текущей основной версии, для чего разработчик выполняет операцию обновления рабочей копии (**update**) насколько возможно часто (реальная частота обновлений определяется частотой внесения изменений, зависящей от активности разработки и числа разработчиков, а также временем, затрачиваемым на каждое обновление — если оно велико, разработчик вынужден ограничивать частоту обновлений, чтобы не терять время).

**Модификация проекта**

Разработчик модифицирует проект, изменяя входящие в него файлы в рабочей копии в соответствии с проектным заданием. Эта работа производится локально и не требует обращений к серверу VCS.

**Фиксация изменений**

Завершив очередной этап работы над заданием, разработчик фиксирует (**commit**) свои изменения, передавая их на сервер (либо в основную ветвь, если работа над заданием полностью завершена, либо в отдельную ветвь разработки данного задания). VCS может требовать от разработчика перед фиксацией обязательно выполнить обновление рабочей копии. При наличии в системе поддержки отложенных изменений (**shelving**) изменения могут быть переданы на сервер без фиксации. Если утверждённая политика работы в VCS это позволяет, то фиксация изменений может проводиться не ежедневно, а только по завершении работы над заданием; в этом случае до завершения работы все связанные с заданием изменения сохраняются только в локальной рабочей копии разработчика.

Ветвление, слияние версий, конфликты и их решения, теги, ci/cd

## Отказоустойчивые и критические системы.

Отказоустойчивость – свойство системы сохранять свою работоспособность после отказа одного или нескольких составных компонентов. Отказоустойчивость определяется количеством любых последовательных единичных отказов компонентов, после чего сохраняется работоспособность системы в целом. Базовый уровень отказоустойчивости подразумевает защиту от отказа одного любого элемента – исключение единой точки отказа. Основной способ повышения отказоустойчивости – избыточность. Наиболее эффективный метод при этом – аппаратная избыточность (резервирование).

Отказоустойчивость следует отличать от отказобезопасности – способности системы при отказе некоторых частей переходить в режим работы, не представляющий опасности для людей, окружающей среды или материальных ценностей. В реальных системах эти два требования могут выступать совместно.

Отказоустойчивость связана со такими характеристиками системы как коэффициент готовности (показывает, какую долю времени от общего времени службы система находится в рабочем состоянии) и надёжность системы (вероятность отказа в единицу времени).

Отказоустойчивая архитектура с точки зрения инженерии – это метод проектирования отказоустойчивых систем, которые способны продолжать выполнение запланированных операций (возможно, с понижением эффективности) при отказе их компонентов. Термин часто используется для описания компьютерных систем, спроектированных продолжать работу в той или иной степени, с возможным уменьшением пропускной способности или увеличением времени отклика, в случае отказа части системы. Это означает, что система в целом не прекратит свою работу при возникновении проблем с аппаратной или программной частью.

Избыточностью называют функциональность, в которой нет необходимости при безотказной работе системы. Пример – запчасти, автоматически включающиеся в работу, если какая-то из основных частей ломается. Например, большие грузовики могут потерять шину без серьёзных последствий, так как на них установлено много шин, и потеря одной не является критичной. Впервые идея включения избыточных частей для увеличения надёжности системы была высказана Джоном фон Нейманом в 1950-х годах.

Различают два типа избыточности – пространственную (путём введения дополнительных компонентов, функций или данных) и временну́ю (путём повторных вычислений, после чего полученный результат сравнивается с сохранённой копией предыдущего результата).

Надо иметь в виду, что отказоустойчивые решения имеют недостатки:

Помехи в обнаружении однокомпонентных неполадок. Например, водитель автомобиля может не заметить, что шина проколота, если используется любая отказоустойчивая система. Проблема может быть решена путём добавления специальной системы слежения за давлением в камерах, предупреждая водителя, если оно падает. Альтернатива – ручная система обнаружения неполадок (осмотр водителем шин на каждой остановке).

Помехи в обнаружении многокомпонентных неполадок. Отказостойкость одного компонента может мешать обнаружению неполадок в другом. Например, если часть *B* выполняет некую операцию на основе данных из части *A*, то отказостойкость части *В* может скрыть проблему, возникшую в *А*. Если часть *В* будет заменена на менее отказоустойчивую, то система может внезапно отказать, при этом будет казаться, что проблема заключается в новой части *В*. И только после проверки системы станет ясно, что проблема была в части *А*.

Уменьшение приоритета исправления неполадки. Даже если оператор знает о наличии проблемы, существование отказоустойчивой системы уменьшит важность исправления проблемы. Если проблемы не исправлять, то это, скорей всего, приведет к полному отказу системы, когда отказостойкая часть сломается полностью, или же когда сломаются все избыточные части.

Сложность проверки. Для некоторых крайне важных отказоустойчивых систем (например, атомный реактор) нет простого пути, чтобы удостовериться, что запасные части находятся в рабочем состоянии.

Цена. И отказостойкие, и избыточные компоненты увеличивают стоимость системы. Это может быть цена или вес (космос и беспилотники).

Некачественные компоненты. Отказоустойчивая архитектура может позволить использование некачественных частей, которые иначе сделали бы систему неработающей. Эта практика при снижении расходов уменьшает надёжность системы.

Есть разница между отказоустойчивыми системами и системами, в которых редко возникают проблемы. Если некие устройства выходят из строя раз в сто лет, они крайне безотказны, но если поломка случается, они прекращают свою работу полностью, т.е. не являются отказоустойчивыми.

Но есть системы, отказы которых могут приводить к значительным экономическим потерям, физическим повреждениям или создавать угрозу человеческой жизни. Такие системы называют критическими (КС)*.* Главное требование к ним – функциональная надежность во всех составляющих (работоспособность, безотказность, безопасность и защищенность). Есть три основных типа КС:

1. Системы, критические пo обеспечению безопасности *–* их отказ приводит к разрушениям, создает угрозу жизни человека или наносит вред окружающей среде (системы ПВО).

2. Системы, критические для целевого назначения *–* их отказможет привести к ошибкам в действиях для обеспечения цели (навигационная система космического корабля).

3. Системы, критические для бизнеса *–* их отказ может нанести вред делу, в котором они используется (система, обслуживающая счета клиентов в банке).

Цена ошибки КС часто очень велика. Она включает как прямые расходы, связанные с внесением изменений в систему или ее заменой, так и косвенные расходы (например, судебные). Из высокой возможной цены отказа системы следует, что качество методов разработки и сам процесс создания ПО обычно более важны, чем стоимость применения этих методов. Поэтому при создании КС обычно используются испытанные методы разработки, а не новые, еще не имевшие большого практического применения.

Из-за необходимости управлять большим числом сенсоров и сложных исполнительных механизмов требуется управляющее ПО. Для КС такое ПО подразделяется на два класса, критических по критерию безопасности.

1. Первичное ПО, которое включается в систему в виде отдельного блока управления. Неправильная работа такого ПО может быть причиной отказа оборудования, вследствие которого может возникнуть угроза жизни человека и т.п.

2. Вторичное ПО, отказ которого косвенным образом может привести к непредвиденным последствиям. Например, медицинская база данных, содержащая описание лекарств, предназначенных пациентам.

Для характеристики КС используется ряд терминов. Например:

Безотказность – способность системы безотказно работать определенное время с указанной целью в определенном окружении.

Работоспособность – способность системы правильно функционировать и предоставлять вовремя требуемые сервисы.

Защищенность – способность системы защищать себя от внешних случайных или преднамеренных воздействий.

Работоспособность и безотказность систем носят вероятностный характер и могут быть выражены количественно. Безопасность и защищенность редко выражаются в виде числовых показателей, но их можно сравнивать по относительной шкале уровней (INES в атомной отрасли).

Рассматривая надежность КС, можно выделить три типа компонентов, склонных к отказу:

1. Аппаратные средства системы, отказывающие либо из-за ошибок конструирования, либо из-за ошибок изготовления, либо из-за полного износа.

2. ПО системы, которое может отказывать из-за ошибок либо в технических требованиях к системе, либо в архитектуре системы, либо в программном коде.

3. Человеческий фактор, под влиянием которого может быть нарушена правильная работа системы.

Условия надежности КС порождают разные виды требований к ним:

1. Функциональные требования подразумевают возможность обнаружения ошибок, восстановления системы и обеспечения ее защиты от сбоев.

2. Нефункциональные требования можно трактовать как требования безотказности и работоспособности системы.

3. Требования «не делать». В противоположность обычным функциональным требованиям, которые определяют систему посредством требований «делать», условия «не делать» описывают систему «от противного», определяя ее поведение, которое недопустимо, например:

* Система не должна позволять пользователю изменять файлы, которые он не создавал (защищенность).
* Система не должна разрешать режим обратного хода, если самолет находится в полете (безопасность).
* Система не должна одновременно выполнять действия по нескольким сигналам тревоги.

## Подходы к созданию защиты систем с конфиденциальной информацией.

На вопрос о том, будет ли в разрабатываемой АС закрытая информация, ответ должен содержаться в документе «Аналитическое обоснование необходимости разработки системы защиты информации в АС». В нём рекомендуется иметь такие разделы:

* информационная характеристика создаваемой АС и описание организационной структуры, для которой эта АС создаётся;
* характеристика комплекса технических средств, ПО, режимов работы, технологии процесса обработки информации;
* угрозы ИБ и перечень мероприятий по их предупреждению и предотвращению;
* обоснование необходимости системы защиты информации;
* перечень сертифицированных средств защиты или обоснование их разработки;
* обоснование необходимости использования технических и организационных решений;
* оценка различных затрат на разработку системы защиты информации, а также ориентировочные сроки её разработки;
* перечень мер по обеспечению режима секретности (если таковая есть).

При создании АС с закрытой информацией главная угроза – отсутствие общей стратегии обеспечения ИБ. А это означает непонимание рисков с точки зрения утечки информации, отсутствие модели угроз, отсутствие классификации информации по степени важности и по степени доступа, отсутствие регламентов поведения сотрудников и ряд прочих важных аспектов. Недостаточно знать, КАК построить решение, – надо уметь обосновывать ЗАЧЕМ это решение следует строить и ПОЧЕМУ строить его нужно именно так. И на вопросы «зачем» и «почему» следует отвечать с точки зрения задач организации. Для этого в организации должна существовать служба информационной безопасности (СИБ). Она может находиться в составе отдела информационных технологий, внутри службы общей безопасности (отдела защиты информации), либо может быть организована в виде полноценной независимой структуры, подчиняющейся топ-менеджменту.

Целью ИБ является обеспечение конфиденциальности, доступности и целостности информационных ресурсов организации. К сожалению, зачастую эти функции разделяются, и тогда обеспечение целостности и доступности считается прерогативой ИТ-службы, а СИБ занимается только вопросами конфиденциальности. Часто это вызвано ошибочным мнением, что ИБ – это прежде всего обеспечение секретности. Разделение должно быть не в функциях, а в масштабах выполнения этих функций. Для ИТ-службы это прежде всего АИС, а для СИБ – вся организация вместе с её людьми, бизнес-процессами, контрагентами, информационными потоками, которые могут воплощаться не только в электронно-цифровом виде.

Создание системы ИБ рекомендуется проводить поэтапно.

Первый этап – проведение аудита всех имеющихся в наличии ресурсов АИС, в результате которого и должно возникнуть ясное представление об объектах защиты, возможных угрозах и степени рисков при их реализации.

Второй этап – определение источников угроз. Как правило, их может быть несколько. Выделить источник угроз – значит, оценить его цели (если источник преднамеренный) или возможное воздействие (непреднамеренный), вероятность (или интенсивность) его появления. Если речь идёт о злоумышленных действиях лица (или группы лиц), то требуется оценить его организационные и технические возможности для доступа к информации. Мировой и отечественный опыт показывает, что более 90% реализованных угроз приходится на инсайдеров и действия слепой стихии.

Третий этап – проведение анализа уязвимости, в результате которого создаются детальные пошаговые сценарии тревожных событий. Каждый сценарий должен сопровождаться таблицей его характеристик (правдоподобности, последствий события и степенью приемлемого для организации риска). Как правило, принято различать следующие виды угроз:

* утечка информации при несанкционированном доступе (НСД);
* нарушение целостности данных (изменение, подлог, удаление);
* отказ в санкционированном доступе к информации в тот момент, когда она нужна легальным пользователям.

Четвертый этап – создание пакетов мер защиты (электронные системы безопасности, физические барьеры, охрана, политики, процедуры).

Пятый этап – внедрение системы с преобразованием рекомендаций в спецификации и инструкции, а также – формирование политик.

В число разработчиков системы ИБ обязаны входить разные лица, поскольку одно из главных правил – **никто не должен сам себе выставлять требования по защите информации**.

Политику ИБ целесообразно рассматривать на трёх уровнях детализации. На первом уровне формулируются цели обеспечения безопасности информации, которые в дальнейшем определяют правила и требования по всем вопросам безопасности информации и становятся обязательными для всех структурных подразделений организации. При этом выпускается документ «Концепция обеспечения безопасности информации», определяющий стратегические, долгосрочные решения по организации ИБ, интегрирующий все другие документы по поставленным целям и задачам. Концепция позволяет правильно организовать взаимодействие подразделений предприятия в вопросах обеспечения ИБ и распределить ответственность должностных лиц в решении этих вопросов.

На втором уровне обычно разрабатывают два документа – Регламент обеспечения безопасности информации и Общие технические требования по обеспечению безопасности информации, которые являются организационно-распорядительными документами, то есть регламентируют все вопросы организации и проведения работ по защите информации, составления положений, регламентов, должностных инструкций и т.п.

К третьему уровню принадлежит исполнительная документация, включающая в себя различные должностные положения и инструкции. Кроме того, данный уровень содержит эксплуатационные документы средств защиты информации, обеспечивающих разграничение доступа к защищаемым ресурсам, систему мониторинга и контроля. Этот уровень опирается на эксплуатационную документацию используемых программно-технических средств защиты, общесистемного и прикладного ПО, а также на стратегию и тактику защиты, обеспечиваемую техническими и программными средствами системы защиты информации. Примеры документов этого уровня:

* положение о категорировании информационных ресурсов;
* положение о подразделении СИБ;
* положение об администраторе безопасности информации;
* инструкция по организации парольной защиты;
* перечень защищаемых объектов информатизации с указанием их характеристик и условий эксплуатации;
* акты о категорировании защищаемых объектов информатизации;
* перечень защищаемых ресурсов в локальных сетях;
* инструкция о порядке предоставления доступа к ресурсам АИС и таблицы разграничения доступа;
* приказ о вводе защищаемого помещения в эксплуатацию и назначении ответственного за помещение лица;
* приказ о вводе в эксплуатацию средств вычислительной техники, обрабатывающих информацию ограниченного доступа, и назначении ответственных лиц;
* инструкция по организации антивирусной защиты в АИС;
* регламент установки, модификации и технического обслуживания оборудования и ПО АИС;
* инструкция по ведению делопроизводства конфиденциальной документации;
* инструкция о порядке учёта, хранения и обращения со съёмными носителями информации;
* акты классификации АИС или отдельных компьютеров, обрабатывающих информацию ограниченного доступа.

Система защиты информации от НСД должна включать три подсистемы.

Подсистема управления доступом должна обеспечивать:

* контроль и разграничение доступа в систему;
* защиту ресурсов локальных вычислительных сетей (ЛВС) от НСД со стороны субъектов доступа корпоративной сети и из внешних телекоммуникационных сетей;
* сегментирование ЛВС и защиту сегментов ЛВС от НСД при организации обработки информации различных категорий доступа в соответствии с требованиями документов регулирующих органов;
* ведение профилей пользователей и полномочий, необходимых для выполнения той или иной ролевой функции, а также ограничение доступа к ведению профилей и полномочий пользователей.

Подсистема регистрации и учёта должна обеспечивать:

* регистрацию системных событий и попыток НСД к защищаемым ресурсам в электронных журналах и оперативное оповещение об этих событиях администраторов безопасности;
* протоколирование изменения профилей и полномочий пользователей;
* протоколирование результатов проверки целостности данных, программных модулей, файлов данных.

Подсистема обеспечения целостности должна предоставлять:

* набор средств для автоматизированного восстановления данных, программных модулей, файлов данных;
* возможность восстановления системы защиты информации от НСД;
* возможность проверки целостности программных модулей и файлов данных (в том числе, как часть процедуры запуска системы).

При создании АИС для объектов, относящихся к критической информационной инфраструктуре, следует руководствоваться соответствующими нормативными документами и прежде всего – Федеральным законом «О безопасности критической информационной инфраструктуры Российской Федерации» от 26.07.2017 N 187-ФЗ.

ФСТЭК России является основным государственным ведомством, ведающим вопросами защиты информации, разработки методик и стратегий такой работы и закреплением данных механизмов и критериев на законодательном уровне. Именно постановления, приказы и руководящие документы ФСТЭК берутся за основу при проектировании и исполнении АИС, защищённых от НСД на различных уровнях. Эти документы определяют классификацию АИС по различным критериям, предписывают использование тех или иных средств и принципов защиты информации, исходя из степени конфиденциальности и важности информации и её обработки на предприятии и т.д. Часть руководящих документов используется в ходе оценки защищённости АИС от НСД, в том числе – в ходе сетевой работы и при взаимодействии в сети Интернет.

ФСТЭК и ряд регулирующих органов требуют, чтобы АИС, в которых обрабатываются сведения, относящиеся к информации с ограниченным доступом, были классифицированы по уровню защищенности от НСД. Классификацию осуществляет комиссия, назначаемая руководителем предприятия. По результатам своей работы комиссия составляет акт классификации, в котором определяется класс, к которому относится АИС, обозначаемый сочетанием цифры и буквы. Цифра означает группу с точки зрения количества пользователей АИС и распределения их прав. Внутри каждой группы может быть несколько уровней (обозначаемых буквой) с точки зрения конфиденциальности защищаемой информации.

Третья группа – это однопользовательская АИС. Для неё определены классы 3А (секретные сведения) и 3Б (несекретные).

Вторая группа – многопользовательская АИС с равными полномочиями всех пользователей. Для неё определены классы 2А (секретные сведения) и 2Б (несекретные).

Первая группа – многопользовательская АИС с разными полномочиями пользователей. Для неё определены классы 1А, 1Б, 1В (секретные сведения), 1Г, 1Д (несекретные).

Класс определяет набор специфических требований по защите информации от НСД. В полном варианте набор насчитывает более двух десятков требований (от идентификации доступа в систему до использования шифрования). Важным элементом классификации является определение необходимости наличия сертифицированных средств защиты. Сертификация в области ИБ преследует две цели: предоставление гарантий отсутствия в средствах обработки информации недекларированных возможностей (закладок) и подтверждение качества (эффективности) продуктов.

Для получения лицензии ФСТЭК на работу с секретными и конфиденциальными сведениями компания обязана привести систему защиты информации в соответствие с приказами и рекомендациями ФСТЭК.

## Спецификация ПО. Типы, виды, назначение, примеры. Языки спецификаций.

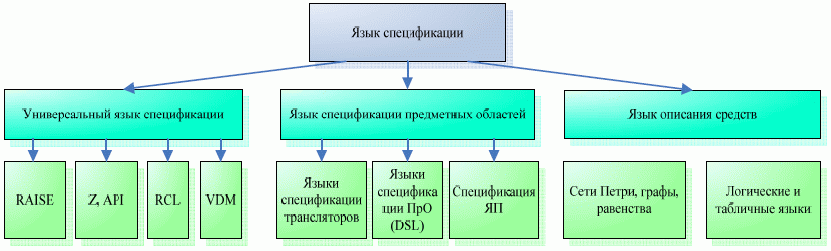
**Спецификация требований программного обеспечения** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *software requirements specification*, SRS) — [структурированный](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) набор требований/запросов (функциональность, производительность, конструктивные ограничения и атрибуты) к [программному обеспечению](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) и его внешним интерфейсам. (Определение на основе IEEE Std 1012:2004) Предназначен для того, чтобы установить базу для соглашения между [заказчиком](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BA%D0%B0%D0%B7%D1%87%D0%B8%D0%BA) и [разработчиком](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%81%D1%82) (или подрядчиками) о том, как должен функционировать [программный продукт](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5).

Может включать ряд [пользовательских сценариев](https://ru.wikipedia.org/wiki/%D0%A1%D1%86%D0%B5%D0%BD%D0%B0%D1%80%D0%B8%D0%B9_%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *use cases*), которые описывают варианты взаимодействия между [пользователями](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C) и [программным обеспечением](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5).

Пользовательские сценарии являются средством представления [функциональных требований](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%82%D1%80%D0%B5%D0%B1%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F). В дополнение к пользовательским сценариям, [спецификация](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B5%D1%86%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F) также содержит [нефункциональные требования](https://ru.wikipedia.org/w/index.php?title=%D0%9D%D0%B5%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%82%D1%80%D0%B5%D0%B1%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F&action=edit&redlink=1), которые налагают ограничения на дизайн или реализацию (такие как требования производительности, [стандарты качества](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D1%8B_%D0%BA%D0%B0%D1%87%D0%B5%D1%81%D1%82%D0%B2%D0%B0&action=edit&redlink=1), или [проектные ограничения](https://ru.wikipedia.org/w/index.php?title=%D0%9F%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%BD%D1%8B%D0%B5_%D0%BE%D0%B3%D1%80%D0%B0%D0%BD%D0%B8%D1%87%D0%B5%D0%BD%D0%B8%D1%8F&action=edit&redlink=1))

Спецификация ПМ подразделяется на синтаксическую (описание входного интерфейса) и функциональную (описание семантики функций, выполняемых этим модулем по каждому из его входов).

В процессе разработки программы ее модульная структура может по-разному формироваться и использоваться для определения порядка программирования и отладки модулей, указанных в этой структуре. Классическими считаются методы восходящей и нисходящей разработки структуры программы.



**Универсальные языки спецификации** (*VDM*, Z, RAISE и др.) имеют общематематическую основу и следующие виды средств:

* логики первого порядка, включая кванторы;
* арифметические операции;
* средства образования множеств с помощью логических формул и операций над множествами;
* средства описания конечных последовательностей (кортежей, списков) и операции над ними;
* средства описания конечных функций и операции над ними;
* средства описания древовидных структур;
* средства построения областей или множества объектов, включая произведения, объединения и рекурсивные определения;
* определение функций с помощью выражений и равенств, включая рекурсивные определения;
* процедурные средства ЯП (операторы присваивания, цикла, выбора, выхода);
* *операции композиции*, аргументами и результатами которых могут быть функции, выражения, операторы.

В *VDM* и RAISE нет средств описания графовых структур, управления и параллелизма, однако имеется механизм конструирования новых структур данных.

**Языки спецификации областей**включают в себя следующие языки:

* спецификации доменов;
* описания взаимодействий;
* спецификации ЯП и трансляторов;
* спецификации БД и знаний;
* *спецификации пакетов* прикладных программ и др.

Каждый из этих языков имеет специализированные средства, отображающие специфические особенности соответствующей области.

Язык спецификации доменов *DSL* (*Domain* Specific *Language*) представляет некоторое *подмножество* языка программирования и специально средства для описания специальных проблем домена [[6.14](https://intuit.ru/studies/courses/2190/237/literature#literature.6.14)]. Он подразделяется на внешние и внутренние языки. *Внешние языки*(типа Unix, *XML* и др.) *по* уровню выше языка описания приложения. Описание в нем сводится к языку *DSL* специальными генераторами или текстовыми редакторами, трансформирующими абстрактные понятия домена к понятиям языка *DSL*. *Внутренние языки*(С, С++), а также языки *Java*, Smalltalk ограничены синтаксисом и семантикой основного базового языка программирования приложений.

Языки описания взаимодействий и параллельного выполнения в отличие от ЯП позволяют специфицировать процессы управления вычислениями, передачей сообщений и взаимодействием объектов в распределенных системах.

*Метаязыки* позволяют специфицировать контекстные зависимости синтаксиса ЯП, лексический и синтаксический *анализ* трансляторов с помощью регулярных выражений КС-*грамматик в форме Бэкуса-Наура*. Для спецификации семантики языков используется *формализм* равенств. Техника описания ЯП основывается на атрибутных грамматиках и абстрактных типах данных. Задача описания ЯП для перевода решаются путем использования денотационных, алгебраических и атрибутных подходов, а также логических терминов, ориентированных на верификацию [[6.11](https://intuit.ru/studies/courses/2190/237/literature#literature.6.11)-[6.16](https://intuit.ru/studies/courses/2190/237/literature#literature.6.16)].

## Техническое задание. Определения, правила написания.

Техническое задание (ТЗ, техзадание) — документ или несколько документов, определяющих цель, структуру, свойства и методы какого-либо проекта, и исключающие двусмысленное толкование различными исполнителями[1][2]. Иными словами, техническое задание — это инструмент коммуникации между заказчиком и исполнителем, который помогает выстроить линию общения с помощью создания внутри него некоего абстрактного элемента, наделенного видением, чувствами и знаниями заказчика

«Техническое задание на создание (развитие или модернизацию) системы». Важно иметь в виду, что создание любой АС без ТЗ обречено на столкновение с серьёзными проблемами, и разработчикам не рекомендуется соглашаться на такой вариант. Нередко возникает вопрос: «Кто должен разрабатывать ТЗ – заказчик или исполнитель?». Наиболее оптимальным вариантом считается их совместное участие в решении этого вопроса, поскольку только при этом можно учесть все аспекты создаваемой АС. Если же в процессе создания АС предусмотрено изготовление и поставка программно-технических средств, то разумно привлечь к составлению документа и представителя организации-поставщика.

Достаточно давно был выпущен ГОСТ 34.602–89, который распространялся на АС для автоматизации различных видов деятельности и устанавливал состав, содержание, правила оформления документа вида ТЗ. Заложенные в этом ГОСТе рекомендации практически не устарели, так что ниже излагаются основные положения, которым рекомендуется следовать при написании ТЗ.

ТЗ на АС должно содержать следующие разделы (с возможным делением на подразделы):

1. Общие сведения.

2. Назначение и цели создания (развития) системы.

3. Характеристика объектов автоматизации.

4. Требования к системе.

5. Состав и содержание работ по созданию системы.

6. Порядок контроля и приемки системы.

7. Требования к составу и содержанию работ по подготовке ОА к вводу АС в действие.

8. Требования к документированию.

9. Источники разработки.

10. Приложения.

В раздел «Общие сведения» включают: полное наименование системы и ее условное обозначение; шифр темы или номер договора; наименование предприятий разработчика и заказчика системы и их реквизиты; перечень документов, на основании которых создается система, кем и когда утверждены эти документы; плановые сроки начала и окончания работы по созданию системы; сведения об источниках и порядке финансирования работ; порядок оформления и предъявления заказчику результатов работ по созданию системы (ее частей), по изготовлению и наладке отдельных средств и программно-технических (программно-методических) комплексов системы.

В разделе «Назначение и цели создания системы» указывают вид автоматизируемой деятельности (управление, проектирование и т. п.) и перечень ОА, на которых предполагается ее использовать. В целях создания приводят наименования и требуемые значения технических, технологических, производственно-экономических или других показателей ОА, которые должны быть достигнуты, и указывают критерии оценки достижения целей создания системы.

В характеристиках указываются краткие сведения об ОА или ссылки на документы, содержащие такую информацию; сведения об условиях эксплуатации ОА и характеристиках окружающей среды.

Раздел «Требования к системе» содержит ряд подразделов, где приводят ссылки на действующие нормативно-технические документы, определяющие требования к системам соответствующего вида.

Подраздел «Требования к системе в целом» включает:

* Требования к структуре и функционированию системы, где приводится перечень подсистем, их назначение и основные характеристики; требования к способам и средствам связи для информационного обмена между компонентами системы и смежными системами; требования к режимам функционирования системы; требования по диагностированию системы; перспективы развития, модернизации системы.
* Требования к численности и квалификации персонала на АС, включая порядок его подготовки и контроля знаний; требуемый режим работы.
* Требования к показателям назначения АС (значения параметров, характеризующих степень соответствия системы ее назначению).
* Требования к надежности включают состав и количественные значения показателей надежности для системы в целом или ее подсистем; перечень аварийных ситуаций, по которым должны быть регламентированы требования к надежности, и значения соответствующих показателей.
* Требования по безопасности при монтаже, наладке, эксплуатации, обслуживании и ремонте технических средств системы; указываются допустимые уровни освещенности, вибрационных и шумовых нагрузок.
* Требования по эргономике и технической эстетике.
* Требования к эксплуатации, техническому обслуживанию, ремонту и хранению.
* Требования к защите информации от несанкционированного доступа.

В подразделе «Требования по сохранности информации» приводят перечень событий (аварий, отказов технических средств), при которых должна быть обеспечена сохранность информации в системе.

Подраздел «Требования к средствам защиты от внешних воздействий» касается радиоэлектронной защиты средств АС; требования по стойкости, устойчивости и прочности к внешним воздействиям (среде применения).

Подраздел «Требования к стандартизации и унификации» устанавливает степень использования стандартных методов реализации функций, типовых математических методов и моделей, унифицированных форм управленческих документов, общесоюзных классификаторов информации.

В подразделе «Требования к функциям (задачам)» по каждой подсистеме приводят перечень функций, задач или их комплексов; временной регламент реализации каждой функции; требования к качеству реализации каждой функции, к форме представления выходной информации, характеристики необходимой точности и времени выполнения.

В подразделе «Требования к видам обеспечения» в зависимости от вида системы приводят требования к математическому, информационному, лингвистическому, программному, техническому, метрологическому, организационному, методическому и другим видам обеспечения системы.

Раздел «Состав и содержание работ по созданию (развитию) системы» должен содержать перечень стадий и этапов работ по созданию, сроки их выполнения, перечень организаций – исполнителей работ, ссылки на документы, подтверждающие согласие этих организаций на участие в создании системы, или запись, определяющую ответственного (заказчик или разработчик) за проведение этих работ. В данном разделе также приводят перечень документов, предъявляемых по окончании соответствующих стадий и этапов работ, и указывают вид и порядок проведения экспертизы и приёмки работ.

Раздел «Порядок контроля и приемки системы» определяет виды, состав, объём и методы испытаний системы и ее составных частей; общие требования к приёмке работ по стадиям (перечень участвующих предприятий, место и сроки проведения), порядок согласования и утверждения приёмочной документации; статус приёмочной комиссии (государственная, межведомственная, ведомственная).

Раздел «Требования к составу и содержанию работ по подготовке ОА к вводу системы в действие» содержит перечень основных мероприятий и их исполнителей, которые следует выполнить при подготовке ОА к вводу АС в действие. Это, например, приведение поступающей в систему информации к виду, пригодному для компьютерной обработки; изменения, которые необходимо осуществить в ОА; создание условий функционирования ОА, при которых гарантируется соответствие создаваемой системы требованиям, содержащимся в ТЗ; создание необходимых для функционирования системы подразделений и служб; сроки и порядок комплектования штатов и обучения персонала.

Раздел «Требования к документированию» содержит согласованный разработчиком и заказчиком системы перечень подлежащих разработке комплектов и видов документов (в том числе, перечень документов, выпускаемых на машинных носителях).

В разделе «Источники разработки» должны быть перечислены документы и информационные материалы, на основании которых разрабатывалось ТЗ и которые должны быть использованы при создании системы.

## Управление качеством. Тестирование и отладка программного средства.

## Критерии качества программного продукта. Обеспечение примитивов качества ПО.

Все критерии качества делятся на «внешние», которые может обнаружить пользователь программного обеспечения, и «внутренние», которые видят только разработчики, создающие это программное обеспечение.

Внешние факторы, которые в том числе описаны в стандарте ИСО 9126:

1. корректность
2. устойчивость
3. расширяемость
4. повторное использование
5. совместимость, эффективность
6. переносимость
7. простота использования
8. функциональность
9. своевременность

Примеры внутренних факторов: читаемость, модульность, легкость обнаружения ошибок.

Для конкретизации качества ПС по каждому из критериев используется стандартизованный набор достаточно простых свойств ПС [4.3-4.6], однозначно интерпретируемых разработчиками. Такие свойства мы будем называть примитивами качества ПС. Некоторые из примитивов могут использоваться по нескольким критериям. Ниже приводится зависимость критериев качества от примитивов качества ПС.

*Функциональность:* завершенность.

*Надежность:*завершенность, точность, автономность, устойчивость, защищенность.

*Легкость применения:* П-документированность, информативность (только применительно к документации по применению), коммуникабельность, устойчивость, защищенность.

*Эффективность:*временнaя эффективность, эффективность по памяти, эффективность по устройствам.

*Сопровождаемость.*С данным критерием связано много различных примитивов качества. Однако их можно распределить по двум группам, выделив два подкритерия качества: изучаемость и модифицируемость. Изучаемость - это характеристики ПС, которые позволяют минимизировать усилия по изучению и пониманию программ и документации ПС. Модифицируемость - это характеристики ПС, которые упрощают внесение в него необходимых изменений и доработок.

*Изучаемость:* С-документированность, информативность (здесь применительно и к документации по сопровождению), понятность, структурированность, удобочитаемость.

*Модифицируемость:* расширяемость, структурированность, модульность.

*Мобильность:* независимость от устройств, автономность, структурированность, модульность.

Ниже даются определения используемых примитивов качества ПС [4.3-4.5].

*Завершенность* - свойство, характеризующее степень обладания ПС всеми необходимыми частями и чертами, требующимися для выполнения своих явных и неявных функций.

*Точность*- мера, характеризующая приемлемость величины погрешности в выдаваемых программами ПС результатах с точки зрения предполагаемого их использования.

*Автономность* - свойство, характеризующее способность ПС выполнять предписанные функции без помощи или поддержки других компонент программного обеспечения.

*Устойчивость*- свойство, характеризующее способность ПС продолжать корректное функционирование, несмотря на задание неправильных (ошибочных) входных данных.

*Защищенность*- свойство, характеризующее способность ПС противостоять преднамеренным или нечаянным деструктивным (разрушающим) действиям пользователя.

*П-документированность*- свойство, характеризующее наличие, полноту, понятность, доступность и наглядность учебной, инструктивной и справочной документации, необходимой для применения ПС.

*Информативность*- свойство, характеризующее наличие в составе ПС информации, необходимой и достаточной для понимания назначения ПС, принятых предположений , существующих ограничений, входных данных и результатов работы отдельных компонент, а также текущего состояния программ в процессе их функционирования.

*Коммуникабельность*- свойство, характеризующее степень, в которой ПС облегчает задание или описание входных данных, а также обеспечивает выдачу полезных сведений в форме и с содержанием, простыми для понимания.

*Временнaя эффективность*- мера, характеризующая способность ПС выполнять возложенные на него функции за определенный отрезок времени.

*Эффективность по памяти*- мера, характеризующая способность ПС выполнять возложенные на него функции при определенных ограничениях на используемую память.

*Эффективность по устройствам* - мера, характеризующая экономичность использования устройств машины для решения поставленной задачи.

*С-документировапнность*- свойство, характеризующее с точки зрения наличия документации, отражающей требования к ПС и результаты различных этапов разработки данной ПС, включающие возможности, ограничения и другие черты ПС, а также их обоснование.

*Понятность*- свойство, характеризующее степень в которой ПС позволяет изучающему его лицу понять его назначение, сделанные допущения и ограничения, входные данные и результаты работы его программ, тексты этих программ и состояние их реализации. Этот примитив качества синтезирован нами из таких примитивов ИСО [4.4], как согласованность, самодокументированность, четкость и, собственно, понятность.

*Структурированность*- свойство, характеризующее программы ПС с точки зрения организации взаимосвязанных их частей в единое целое определенным образом (например, в соответствии с принципами структурного программирования).

*Удобочитаемость*- свойство, характеризующее легкость восприятия текста программ ПС (отступы, фрагментация, форматив-

ность).

*Расширяемость*- свойство, характеризующее способность ПС к использованию большего объема памяти для хранения данных или расширению функциональных возможностей отдельных компонент.

*Модульность*- свойство, характеризующее ПС с точки зрения организации его программ из таких дискретных компонент, что изменение одной из них оказывает минимальное воздействие на другие компоненты.

*Независимость от устройств*- свойство, характеризующее способность ПС работать на разнообразном аппаратном обеспечении (различных типах, марках, моделях ЭВМ).

## Жизненный цикл программного продукта. Определения, описание, примеры, характерные особенности.

Жизненный цикл (ЖЦ) программного продукта – период времени, который начинается с момента принятия решения о необходимости создания продукта и заканчивается в момент его полного изъятия из эксплуатации.

В общем случае, ЖЦ определяется моделью и описывается в форме методологии (метода). Модель определяет концептуальный взгляд на организацию ЖЦ, основные фазы ЖЦ и принципы перехода между ними. Методологиязадает комплекс работ, их детальное содержание и ролевую ответственность специалистов на всех этапах выбранной модели ЖЦ, обычно определяет и саму модель, а также рекомендует практики, позволяющие максимально эффективно воспользоваться соответствующей методологией и ее моделью.

Большинство стандартов определяют ЖЦ как структуру декомпозиции работ, процессов и их взаимосвязей. Декомпозиция процессов строится на основе двух важнейших принципов – модульность (задачи в процессе являются функционально связанными; связь между процессами минимальна; если функция используется более, чем одним процессом, она сама является процессом) и ответственность (каждый процесс управляется и/или контролируется конкретным лицом, определенным для заданного ЖЦ, например, в виде роли в проектной команде; функция, чьи части находятся в компетенции различных лиц, не может рассматриваться как самостоятельный процесс).

Основные процессы ЖЦ:

* Приобретение
* Поставка
* Разработка
* Эксплуатация
* Сопровождение.

Вспомогательные процессы ЖЦ:

* Документирование
* Управление конфигурацией
* Обеспечение качества
* Верификация
* Аттестация
* Совместный анализ
* Аудит
* Решение проблем.

Организационные процессы ЖЦ:

* Управление
* Создание инфраструктуры
* Усовершенствование
* Обучение.

Далее пример из моделей создания по

## Документирование программных средств. Типы и виды документации. Описание.

Разрабатываемые документы можно разделить на три категории:

1) Документация разработки. Здесьопределяют требования, которым должен удовлетворять продукт, а также описывают процедуры мониторинга проекта и обеспечения его качества. В неё должно включаться подробное описание ПО (логика, взаимосвязи, форматы и хранение данных и т. д.).

Типовыми документами разработки являются:

* анализы осуществимости и исходные заявки;
* спецификации требований;
* спецификации функций;
* проектные спецификации, включая спецификации программ и данных;
* планы разработки;
* планы сборки и тестирования ПО;
* планы обеспечения качества, стандарты и графики;
* защитная и тестовая информация.

2) Документация продукции. Она представляет информацию, необходимую для эксплуатации, сопровождения, модернизации, преобразования и передачи продукта.

Типовые документы продукции включают:

* учебные руководства;
* справочные руководства и руководства пользователей (обычных и администраторов) или П-документация;
* руководства по сопровождению ПО (С-документация);
* брошюры и информационные листовки, посвященные продукции.

3) Документация управления проектом:

* планы и графики для каждой стадии процесса разработки;
* отчеты об использовании ресурсов и об изменениях графиков;
* отчеты о согласованных изменениях ПО;
* отчеты о решениях, связанных с разработкой;
* распределение обязанностей персонала;
* рабочие документы, обеспечивающие связь между разработчиками (в том числе заметки, переписка, временные версии документов).

Состав пользовательской документации зависит от аудиторий пользователей, на которых ориентирована разрабатываемая ПС, и от режима использования документов. Под аудиторией понимается контингент пользователей ПС, у которого есть необходимость в определенной пользовательской документации. Пользовательская документация должна содержать информацию, необходимую для каждой аудитории. Под режимом использования документа понимается способ, определяющий, каким образом используется этот документ. Обычно пользователю достаточно больших программных систем требуются либо документы для изучения ПС (использование в виде инструкции), либо для уточнения некоторой информации (использование в виде справочника).

Разработка пользовательской документации начинается сразу после создания внешнего описания. Качество этой документации может существенно определять успех разработки. Она должна быть достаточно проста и удобна для пользователя. Поэтому, хотя черновые варианты пользовательских документов создаются основными разработчиками ПС, к созданию их окончательных вариантов часто привлекаются профессиональные технические писатели. Кроме того, для обеспечения качества документации разработан ряд стандартов, в которых предписывается порядок разработки этой документации, формулируются требования к каждому виду пользовательских документов и определяются их структура и содержание.

Документацию по сопровождению ПС можно разбить на две группы. Первая группа содержит итоговые документы каждого технологического этапа разработки ПС:

* Внешнее описание ПС.
* Описание архитектуры ПС, включая внешнюю спецификацию каждой ее программы (подсистемы).
* Описание модульной структуры каждой программы ПС, включая внешнюю спецификацию каждого включенного в нее модуля.
* Спецификация каждого модуля и описание его строения.
* Тексты модулей на выбранном языке программирования.
* Документы установления достоверности ПС (схема тестирования и описание комплекта тестов).

Документацией второй группы обычно является руководство по сопровождению ПС, которое описывает особенности реализации ПС (в частности, трудности, которые пришлось преодолевать). В нем также фиксируются части ПС, которые являются аппаратно- и программно-зависимыми.

Требования к содержанию документов, разрабатываемых при создании АС, установлены указаниями РД 50-34.698-90, а также соответствующими государственными стандартами Единой системы программной документации (ЕСПД), Единой системы конструкторской документации (ЕСКД) и ГОСТ 34.602. Виды и комплектность документов регламентированы ГОСТ 34.201. Содержание документов является общим для всех видов АС и может дополняться разработчиком документов с учётом особенностей создаваемой АС.

Все документы можно разбить на несколько видов. Первый из них касается общесистемных решений. Во многом эти вопросы освещаются в техническом задании (ТЗ) на АС, о чём подробнее будет сказано в следующей теме.

* Пояснительные записки к эскизному и техническому проектам.
* Схема функциональной структуры.
* Описание автоматизируемых функций.
* Описание постановки задачи (комплекса задач).
* Локальная смета и локальный сметный расчет.
* Паспорт.
* Формуляр.
* Общее описание системы.
* Программа и методика испытаний.

К отчетным документам относят акт и отчет о результатах испытаний, акт технического состояния системы после испытаний. В приложения включают перечень методик испытаний, математических и комплексных моделей, применяемых для оценки характеристик системы.

Вторая вид документов касается организационного обеспечения АС:

* Описание организационной структуры (в том числе и её изменения).
* Руководство пользователя.
* Описание технологического процесса обработки данных.

Третий вид относится к описанию ПО – структура, функции частей ПО, руководство программиста, методы и средства разработки, операционная система, платформа и т.п.

Если в АС предусматривается обработка конфиденциальной информации, то необходимо создание дополнительных документов, о чём подробнее будет сказано в другой теме.

## проектом. Классические методы анализа. Измерения, меры и метрики.

## Правила «хорошего тона» при написании ПО.

**Соблюдайте единый Code style.** Если программист приходит работать в организацию, особенно крупную, то чаще всего его знакомят с правилами оформления кода в конкретном проекте (соглашение по code style). Это не случайный каприз работодателя, а свидетельство серьёзного подхода.  
Вот несколько общих правил, с которыми вы можете столкнуться:  
соблюдайте переносы фигурных скобок и отступы — это значительно улучшает восприятие отдельных блоков кода  
соблюдайте правило вертикали — части одного запроса или условия должны находиться на одном отступе.

**Не используйте «магические числа».**Магические числа не случайно относят к анти-паттернам программирования, проще говоря, правилам того, как не надо писать программный код. Чаще всего магическое число как анти-паттерн представляет собой используемую в коде константу, смысл которой неясен без комментария. Такие числа не только усложняют понимание кода и ухудшают его читабельность, но и приносят проблемы во время рефакторинга.  
Например, в коде есть строка:  
DrawWindow( 50, 70, 1000, 500 );

Очевидно, она не вызовет ошибок в работе кода, но и её смысл не всем понятен. Гораздо лучше не полениться и сразу написать таким образом:  
**int** left = 50;  
**int** top = 70;  
**int** width = 1000;  
**int** height = 500;  
DrawWindow( left, top, width, height );

Иногда магические числа возникают при использовании общепринятых констант, например, при записи числа π. Допустим, в проекте было внесено:  
SquareCircle = 3.14\*rad\*rad

Что тут плохого? А плохое есть. Например, если в ходе работы понадобится сделать расчёт с высокой точностью, придётся искать все вхождения константы в коде, а это трата трудового ресурса.

**Используйте осмысленные имена для переменных, функций, классов.**Всем программистам известен термин “обфускация кода” — сознательное запутывание программного года с помощью приложения-обфускатора. Она делается с целью скрыть реализацию и превращает код в невнятный набор символов, переименовывает переменные, меняет имена методов, функций и проч… К сожалению, случается так, что код и без обфускации выглядит запутанно — именно за счёт бессмысленных имён переменных и функций: var\_3698, myBestClass, NewMethodFinal и т.д… Это не только мешает разработчикам, которые участвуют в проекте, но и приводит к бесконечному количеству комментариев. Между тем, переименовав функцию, можно избавиться от комментариев — её имя будет само говорить о том, что она делает.

**Создавайте методы как новый уровень абстракции с осмысленными именами** и **делайте методы компактными.** Вообще, сегодня модульность кода доступна каждому программисту, а это значит, что нужно стремиться создавать абстракции там, где это возможно. Абстракция — это один из способов сокрытия деталей реализации функциональности. Создавая отдельные небольшие методы, программист получает хороший код, разделённый на блоки, в которых содержится реализация каждой из функций. При таком подходе нередко увеличивается количество строк кода. Есть даже определённые рекомендации, которые указывают длину метода не более 10 строк. Конечно, размер каждого метода остаётся целиком на усмотрении разработчика и зависит от многих факторов. Наш совет: всё просто, делайте метод компактным так, чтобы один метод выполнял одну задачу. Отдельные вынесенные сущности проще улучшить, например, вставить проверку входных данных прямо в начале метода.

**В начале методов проверяйте входные данные.**На уровне кода нужно обязательно делать проверки входных данных во всех или практически во всех методах. Это связано с пользовательским поведением: будущие пользователи могут вводить любые данные, которые могут вызвать сбои в работе программы. В любом методе, даже в том, который использовался всего один раз, обязательно нужно организовывать проверку данных и создавать обработку ошибок. Это стоит сделать, поскольку метод не только выступает как уровень абстракции, но и необходим для переиспользования. В принципе, возможно разделить методы на те, в которых нужно делать проверку, и те, в которых её делать необязательно, но для полной уверенности и защиты от «хитрого пользователя» лучше проверять все входные данные.

**Реализуйте при помощи наследования только отношение «является». В остальных случаях – композиция.**Композиция является одним из ключевых паттернов, нацеленных на облегчение восприятия кода и, в отличие от наследования, не нарушает принцип инкапсуляции. Допустим, у вас есть класс Руль и класс Колесо. Класс Автомобиль можно реализовать как наследник класса-предка Руль, но ведь Автомобилю нужны и свойства класса Колесо.  
Соответственно, программист начинает плодить наследование. А ведь даже с точки зрения обывательской логики класс Автомобиль — это композиция элементов. Допустим, есть такой код, когда новый класс создаётся с использованием наследования (класс ScreenElement наследует поля и методы класса Coordinate и расширяет этот класс):  
сlass Coordinate {  
 **public** **int** x;  
 **public** **int** y;  
}  
**class** ScreenElement : Coordinate {  
**public** **char** symbol;  
}

Используем композицию:

сlass Coordinate {  
 **public** **int** x;  
 **public** **int** y;  
}  
**class** ScreenElement {  
 **public** Coordinate coordinate;  
 **public** **char** symbol;  
}

Композиция — неплохая замена наследованию, этот паттерн более простой для дальнейшего понимания написанного кода. Можно придерживаться такого правила: выбирать наследование, только если нужный класс схож с классом-предком и не будет использовать методы других классов. К тому же, композиция избавляет программиста ещё от одной проблемы — исключает конфликт имён, который случается при наследовании. Есть у композиции и недостаток: размножение количества объектов может оказывать влияние на производительность. Но опять же, это зависит от масштаба проекта и должно оцениваться разработчиком в каждом случае отдельно.  
**Отделяйте интерфейс от реализации.**Любой используемый в программе класс состоит из интерфейса (того, что доступно при использовании класса извне) и реализации (методы). В коде интерфейс должен быть отделён от реализации как для соблюдения одного из принципов ООП, инкапсуляции, так и для улучшения читабельности кода.

## Авторские права и лицензирование ПО.

[Программы для ЭВМ](https://www.tadviser.ru/index.php/%D0%A1%D1%82%D0%B0%D1%82%D1%8C%D1%8F:%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) являются объектом авторских прав и регулируются отдельной, 4 частью Гражданского Кодекса РФ. Автору произведения или иному правообладателю принадлежит исключительное право использовать произведение в любой форме и любым не противоречащим закону способом.

Исключительное право на результат интеллектуальной деятельности первоначально возникает у его автора. Это право может быть передано автором другому лицу по договору. Правообладатель может по своему усмотрению разрешать или запрещать другим лицам использование результата интеллектуальной деятельности или средства индивидуализации (неисключительное право на произведение).

Неисключительное право может предоставляться на временной или постоянной основе. Условия использования описаны в лицензионном договоре. Отсутствие запрета не считается согласием (разрешением).

Программное обеспечение – объект авторского права, для использования которого требуется лицензия. Лицензия предоставляет право на установку, использование, доступ, отображение, запуск или любое другое взаимодействие с ПО. Объём прав и условия использования описаны в соответствующих лицензионных соглашениях.

Лицензии по типам можно разделить на следующие группы:

Коммерческая лицензия (Commercial) - наиболее часто используемые коммерческими гигантами Microsoft, SAP, Oracle, Corel Corporation, Autodesk и т.д.

Бесплатная лицензия (Freeware) - наиболее любимая лицензия в РФ. Позволяет свободно и без ограничений использовать ПО. Без денежных обязательств.

Свободное программное обеспечение (open source). Разница между freeware и open source, даже если это одинаковые по функционалу программы, в дальнейшей судьбе продуктов. Freeware умирают через несклько лет - семья, дети, новые проекты, за которые платят. Свободные программы (Open-source) - умирают реже.

Пробные программы (Shareware) - условно бесплатное программное обеспечение. Платные программы, которые на пробу разрешается использовать даром.

Экзотические программные лицензии - сложноформализуемые требования правообладателей. Например - Бесплатно для жителей бывшего СССР.

## Методологии разработки ПО: Waterfall, V-model, Agile, XP, Scrum, Kanban, DevOps.

Рассмотрено ранее

Kanban — это популярный подход к реализации принципов agile и DevOps при разработке ПО. Методика предполагает обсуждение производительности в режиме реального времени и полную прозрачность рабочих процессов. Рабочие задачи визуально представлены на доске Kanban, что позволяет участникам команды видеть состояние каждой задачи в любой момент времени.

DevOps — это не просто совместная работа команд по разработке и эксплуатации. Это не только инструменты и практики. DevOps — это образ мышления, а также изменения в культуре, при которых команда меняет стиль работы.



## Задачи и проблемы сопровождения.

Сопровождение ПО определяется стандартом IEEE 1219 как модификация программного продукта после передачи в эксплуатацию для устранения сбоев, улучшения показателей производительности и/или других характеристик продукта, или адаптации продукта для использования в модифицированном окружении.

Сопровождение – это обычный процесс изменения системы после ее поставки заказчику. Эти изменения могут быть как элементарно простыми (исправление ошибок программирования), так и более серьёзными, связанными с корректировкой отдельных недоработок либо приведением в соответствие с новыми требованиями. Сопровождение ПО заключается в повторном применении предшествующих этапов ЖЦ к существующей программе, но не в разработке новой программы и не в значительном изменении архитектуры системы.

Различают три вида сопровождения в зависимости от цели:

* Исправление ошибок.
* Адаптация ПО к специфическим условиям эксплуатации.
* Изменение функциональных возможностей системы.

На практике чёткое разграничение между различными видами сопровождения провести сложно.

Причиной высоких затрат на сопровождение является сложность модернизации системы после ее внедрения, поскольку расширить функциональные возможности намного легче в процессе создания системы. Ниже приведены ключевые факторы, которые определяют стоимость разработки и сопровождения и могут привести к подорожанию сопровождения.

* Стабильность команды разработчиков.
* Ответственность согласно контракту.
* Квалификация специалистов.
* Возраст и структура программы.

Первые три проблемы объясняются тем, что многие организации всё ещё считают сопровождение второстепенным делом и не желают инвестировать средства для снижения затрат на будущее сопровождение. Руководству организаций необходимо помнить, что у систем редко бывает чётко определенный срок функционирования, и они могут находиться в эксплуатации в той либо иной форме неограниченное время.

В конце XX века М. Леман сформулировал восемь принципов эволюции ПО, которые связывают с деятельностью по сопровождению. Эти принципы часто называют «законами Лемана».

Первый принцип гласит, что используемая программа должна быть непрерывно адаптируемой, иначе она будет становиться всё менее удовлетворительной. Из него следует необходимость постоянного сопровождения системы. При изменении окружения, в котором работает система, появляются новые требования, и система должна неизбежно изменяться с тем, чтобы им соответствовать. Изменения системы носят циклический характер, когда новые требования порождают появление новой версии системы, что, в свою очередь, вызывает изменения системного окружения; это находит отражение в формировании новых требований к системе и т.д. Эти идеи впоследствии нашли своё отражение и дополнение в шестом и седьмом принципах.

Второй принцип исходит из того, что по мере эволюционирования программы её сложность растёт, если не производится работ по стабилизации и уменьшению сложности. В неявном виде это подразумевает необходимость профилактического обслуживания, которое требует средств и времени.

В третьем принципе утверждается, что процесс эволюции программы является саморегулируемым, с близким к нормальному распределению масштабом атрибутов продукта и процесса. Как только система превышает определенный размер, она начинает действовать подобно некой инерционной массе. Размер становится препятствием для новых изменений, поскольку эти изменения с большой вероятностью станут причиной ошибок в системе, которые снизят эффективность нововведений. В восьмом принципе эти идеи получили развитие с точки зрения влияния обратной связи в сложных системах.

Четвёртый принцип касается сохранения организационной стабильности. Из него следует, что изменения ресурсов или персонала оказывают незначительное влияние на долгосрочное развитие системы, а крупные команды программистов неэффективны, так как время, потраченное на общение и внутрикомандные связи, превышает время непосредственной работы над системой.

Пятый принцип говорит о сохранении осведомлённости, имея в виду, что в течение активной жизни эволюционирующей программы основное содержание последующих релизов статистически неизменно. В частности, из него следует, что за весь ЖЦ системы количество изменений в каждой версии остается приблизительно одинаковым.

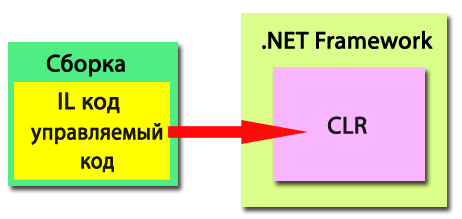
Из техник сопровождения следует отметить:

* Реинжиниринг – детальная оценка и перестройка ПО для формирования понимания, воссоздания и дальнейшей реализации его функций в новой форме (например, с использованием новых технологий и платформ, при сохранении существующей и добавлении новой функциональности). Реинжиниринг часто проводится для замены устаревшего ПО.
* Рефакторинг –трансформация ПО, в процессе которой программная система реорганизуется с целью улучшения структуры, не переписываясь и без изменения поведения.

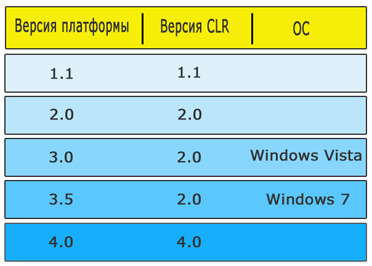
# Программирование под платформу .NET

## Основные компоненты платформы .NET. CLR, IL,JIT-компилятор, FCL.

osnovnye-komponenty-net-framework-clr-i-framework_1.pngВ первой статье Вы узнали, что платформа .NET Framework состоит из 2 основных компонентов, это общеязыковая исполняющая среда (CLR) и библиотека классов платформы (FCL).  
Сегодня мы более подробно рассмотрим, каждый из этих компонентов.  
**CLR**  
Ядром платформы .NET Framework является общеязыковая исполняющая среда (Common Language Runtime) или сокращенно CLR.  
На компьютере храниться в виде файла динамической библиотеки **mscoree.dll**, в проекте данная сборка подключается автоматически.  
В предыдущей статье мы научились создавать файл сборку. Данный файл содержит внутри себя управляемый код на промежуточном языке IL, который может выполняться только в среде CLR, которую предоставляет нам платформа .NET Framework.

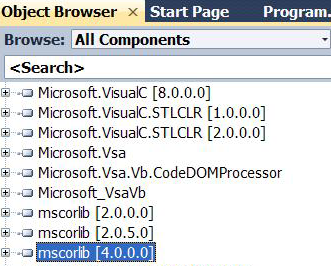
  
Какие задачи решает CLR?

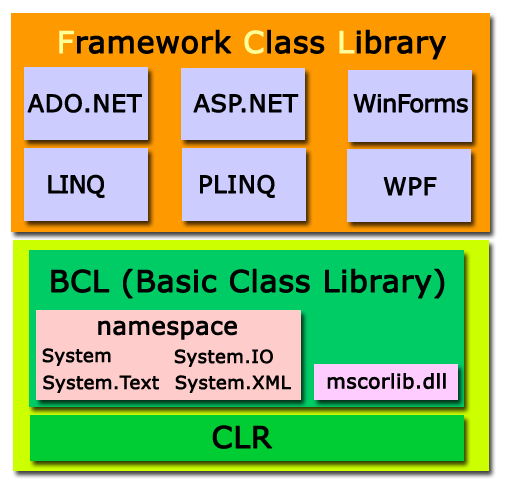
1. Загрузка пользовательских сборок
2. Управление памятью
3. Работа с потоками
4. Безопасность выполнения кода
5. Очистка памяти (сборщик мусора)
6. и многое другое  
   Стоит понимать, что версия среды и версия платформы могут отличаться, как показано на рисунке ниже.

  
На рисунке видно, что версии среды 3.0, 3.5 &#8212; не существует. Поэтому важно понимать, что версия платформы и версия среды &#8212; это разные вещи.  
Кроме двух компонентов, которыми часто описывают всю платформу, существует ещё два не мало важных компонента, которые составляют основу платформы .NET

**Библиотеки**  
На самом деле в .NET существует два типа библиотек:

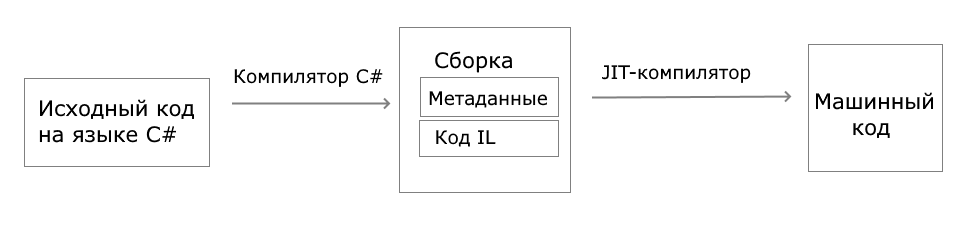
1. BCL (Base Class Library) библиотека базовых классов
2. FCL (Framework Сlass Library) библиотека классов платформы

**Base Class Library (BCL)**  
Base Class Library (BCL) &#8212; это стандартная библиотека для всех языков платформы .NET, которая содержит набор пространств имен. Внутри которых содержится набор базовых типов, которые обеспечивают: файловый и консольный ввод-вывод, безопасность, XML, сетевой доступ, многопоточную обработку и многое другое.  
Основной или главной сборкой данной библиотеки является файл динамической библиотеки &#8212; mscorlib.dll

  
Данная библиотека подключается автоматически во время создания проекта, так же стоит понимать, что существуют и другие файлы сборки базовых библиотек.  
**Framework Class Library (FCL)**  
Вторым типом библиотек платформы .NET Framework является библиотека классов платформы (FCL). Она является вторым основным компонентом платформы .NET, которая упрощают работу программиста, предоставляя ему, более расширенные библиотеки, такие как: ASP.NET, ADO.NET, Windows Forms, WPF.  
В принципе можно сказать, что **Framework Class Library** включает в себя библиотеку базовых классов BCL, которая, по сути, является её ядром.  
Тогда на вопрос, из каких основных компонентов состоит платформа .NET? Можно ответить так:  
.NET Framework = CLR + FCL  
Такую официальную формулировку так же приводят: MSDN, Wiki и многие другие авторы книг.  
&#171;The key components of the .NET Framework are the common language runtime (CLR) and the .NET Framework class library&#187;. MSDN  
Но мы знаем, что FCL включает в себя библиотеки базовых классов. И правильнее формулировка должна звучать так:  
.NET Framework = CLR + Libraries (FCL, BCL)  
Но тут конечно можно долго спорить.

Для создания приложений на платформе .NET можно использовать различные языки программирования - C#, VB.NET, F#, другие языки, которые имеют неофициальную поддержку .NET (типа Deflfi .NET и т.д.). При компиляции кода на любом из этих языков создается **сборка** - это может быть выполняемое приложение, либо библиотека классов, которая используется в других приложениях. Однако вне зависимости от того, какой язык используется, скомпилированная сборка содержит код в некоторой промежуточной форме, независимой от исходного языка программирования, целевой машины и ее операционной системы. Благодаря этому проекты на одном .NET-языке могут использовать сборки, написанные на другом .NET-совместимом языке программирования. При запуске приложения и соотвественно запуске файла сборки общеязыковая среда выполнения CLR загружает ее и преобразует ее код в машинный код для последующего выполнения.

Промежуточное представление приложений .NET, предназначенное для общеязыковой среды выполнения CLR. среды, включает в себя два основных компонента: метаданные и управляемый код (managed code). **Метаданные** — это система дескрипторов всех компонентов приложения (классов, структур и их элементов и т.д.) и их отношения. **Управляемый код** представляет функциональность методов приложения, закодированных в специальной бинарной форме, которая называется **MSIL** или **Microsoft intermediate language** (другие названия - Common Intermediate Language (CIL) или просто Intermediate Language (IL). Затем уже при выполнении сборки JIT-компилятор компилирует методы, закодированные в MSIL, в бинарный/машинный код текущей платформы, который затем собственно и выполняется.



Для работы с кодом IL Microsoft предоставляет ряд утилит, в частности, ildasm и ilasm.

Когда среда CLR выполняет IL код, то происходит следующее: при вызове блока IL-кода JIT-компилятор проверяет и преобразует его в машинные команды которые он сохраняет в динамическом блоке памяти. Затем, он возвращается к структуре данных типа и заменяет адрес вызываемого метода адресом этого блока памяти и при следующем запросе повторная компиляция не будет выполнена, а будет использован уже скомпилированный код.

А за превращение IL в последовательность машинных команд чаще всего отвечает Just-In-Time-компилятор (JIT). Да, на сегодняшний день есть NGen, Mono AOT, .NET Native, но JIT-компиляция всё ещё лидирует в мире .NET-приложений.

**JIT-компиляция** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *just-in-time*, компиляция «точно в нужное время»), **динамическая компиляция** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *dynamic translation*) — технология увеличения производительности программных систем, использующих [байт-код](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4), путём компиляции байт-кода в [машинный код](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) или в другой формат непосредственно во время работы программы. Таким образом достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом[[1]](https://ru.wikipedia.org/wiki/JIT-%D0%BA%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%86%D0%B8%D1%8F#cite_note-1) (сравнимая с компилируемыми языками) за счёт увеличения потребления памяти (для хранения результатов компиляции) и затрат времени на компиляцию. Технология JIT базируется на двух более ранних идеях, касающихся среды выполнения: [*компиляции байт-кода*](https://ru.wikipedia.org/wiki/AOT-%D0%BA%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%86%D0%B8%D1%8F) и *динамической компиляции*.

Так как JIT-компиляция является, по сути, одной из форм динамической компиляции, она позволяет применять такие технологии, как [адаптивная оптимизация](https://ru.wikipedia.org/wiki/%D0%90%D0%B4%D0%B0%D0%BF%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D1%82%D0%B8%D0%BC%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) и [динамическая рекомпиляция](https://ru.wikipedia.org/w/index.php?title=%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%80%D0%B5%D0%BA%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%86%D0%B8%D1%8F&action=edit&redlink=1). Благодаря этому JIT-компиляция может показывать лучшие результаты в плане производительности, чем статическая компиляция. [Интерпретация](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%82%D0%BE%D1%80) и JIT-компиляция особенно хорошо подходят для [динамических языков программирования](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), при этом среда исполнения справляется с [поздним связыванием типов](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BB%D0%B0%D1%81%D1%82%D1%8C_%D0%B2%D0%B8%D0%B4%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B8#%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5_vs._%D0%B4%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5_%D0%BE%D0%B1%D0%BB%D0%B0%D1%81%D1%82%D0%B8_%D0%B2%D0%B8%D0%B4%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B8) и гарантирует безопасность исполнения.

Основная цель использования JIT — достичь и превзойти производительность статической компиляции, сохраняя при этом преимущества динамической компиляции:

* Большинство тяжеловесных операций, таких, как [парсинг](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D1%81%D0%B8%D0%BD%D0%B3) исходного кода и выполнение базовых оптимизаций, происходит во время компиляции (до развёртывания), в то время как компиляция в машинный код из байт-кода происходит быстрее, чем из исходного кода.
* Байт-код более переносим (в отличие от машинного кода).
* Среда может контролировать выполнение байт-кода после компиляции, поэтому приложение может быть запущено в [песочнице](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%81%D0%BE%D1%87%D0%BD%D0%B8%D1%86%D0%B0_(%D0%B1%D0%B5%D0%B7%D0%BE%D0%BF%D0%B0%D1%81%D0%BD%D0%BE%D1%81%D1%82%D1%8C)) (для нативных программ такая возможность тоже существует, но реализация данной технологии сложнее).
* Компиляторы из байт-кода в машинный код легче в реализации, так как большинство работы по оптимизации уже было проделано компилятором.

JIT, как правило, эффективней, чем интерпретация кода. К тому же в некоторых случаях JIT может показывать большую производительность по сравнению со статической компиляцией за счёт оптимизаций, возможных только во время исполнения:

1. Компиляция может осуществляться непосредственно для целевого процессора и операционной системы, на которой запущено приложение. Например, JIT может использовать векторные [SSE2](https://ru.wikipedia.org/wiki/SSE2) расширения процессора, если он обнаружит их поддержку.
2. Среда может собирать статистику о работающей программе и производить оптимизации с учётом этой информации. Некоторые статические компиляторы также могут принимать на вход информацию о предыдущих запусках приложения.
3. Среда может делать глобальные оптимизации кода (например, встраивание библиотечных функций в код) без потери преимуществ динамической компиляции и без [накладных расходов](https://ru.wikipedia.org/w/index.php?title=%D0%9D%D0%B0%D0%BA%D0%BB%D0%B0%D0%B4%D0%BD%D1%8B%D0%B5_%D1%80%D0%B0%D1%81%D1%85%D0%BE%D0%B4%D1%8B_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)&action=edit&redlink=1), присущих статическим компиляторам и [компоновщикам](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%BE%D0%B2%D1%89%D0%B8%D0%BA).
4. Более простое перестраивание кода для лучшего использования [кэша](https://ru.wikipedia.org/wiki/%D0%9A%D1%8D%D1%88_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80%D0%B0).

## Объектно-ориентированное программирование на языке C#.

Объектно-ориентированное программирование основано на четырех основных принципах:

* **Абстракция**: абстрактное поведение объектов обобщается в классах
* **Инкапсуляция данных**: свойства и методы инкапсулируются в виде классов и скрыты от внешнего доступа.
* **Наследование**: свойства и методы могут быть унаследованы одним классом от другого класса
* **Полиморфизм**: множество форм - объекты могут принимать различные формы в зависимости от их использования

«В объектно-ориентированном программировании класс представляет своего рода образец объектов, шаблон, на основе которого могут быть созданы отдельные экземпляры (объекты) во время выполнения программы. Внутри класса разработчик определяет свойства и методы, которыми должны обладать отдельные экземпляры объекта. Свойства представляют состояние экземпляров объекта, методы и их поведение.

Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. Совокупность подобных характеристик можно назвать шаблоном человека или классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек (фактически экземпляр данного класса) будет представлять объект этого класса.

Существуют классические объектно-ориентированные языки как Java или C#. Также есть языки, который в той или иной мере применяют ООП, но чисто объектно-ориентированными языками не являются, как например, JavaScript.

Рассмотрим вкратце ООП на примере Java и JavaScript.

**Абстракция**

В объектно-ориентированном программировании основу или асбстракцию для объектов определяют классы. Классы содержат общее состояние и поведение объектов. Например, нам надо представить в программе человека. В большинстве языков программирования для определения классов применяется ключевое слово class. Так, в Java мы могли бы определить следующий класс Person, который представляет человека:

|  |  |
| --- | --- |
| 1 | class Person{ } |

Аналогичное определение класса человека в JavaScript:

|  |  |
| --- | --- |
| 1 | class Person{ } |

В процессе определения абстрации объектов - класса мы абстрагируемся от конкретных признаков объектов и выделяем общие для них характеристики и поведение. Набор общих признаков объектов образует состояние класса. Так, мы можем выделить у человека такие признаки как имя и возраст. Эти характеристики будут представлять состояние. Для определения состояния в классах обычно используются поля или переменные класса (в некоторых языках их называют свойствами, в других языках свойства и поля класса разделяются). Например, в Java мы могли бы определить состояние следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5 | // класс человека  class Person{      String name;    // имя человека      int age;       // возраст человека  } |

Здесь переменная name представляет тип String (строку) и хранит имя человека. Переменная age представляет тип int или число и хранит возраст.

Аналогичное определение в JavaScript:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class Person{        name;   // имя      age;    // возраст  } |

Человек может производить некоторые действия. Например, человек может идти, спать, питаться и т.д. Это то, что называется поведение объекта. В контексте нашей программы пусть поведение человека ограничено тем, что он говорит, как его зовут и сколько ему лет. Для определения поведения/действий класса определяются методы. Например, добавим в класс Person метод say, с помощью которого человек будет сообщать информацию о себе. Пример на Java:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // класс человека  class Person{      String name;    // имя человека      int age;       // возраст человека      // человек сообщает информацию о себе      public void say(){          System.out.printf("Меня зовут %s \n", name);          System.out.printf("Мне %d лет \n", age);      }  } |

В методе say просто выводим на консоль с помощью метода System.out.printf() данные полей класса.

Аналогичный пример для JavaScript:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Person{        name;   // имя      age;    // возраст      say(){          console.log("Меня зовут", this.name);          console.log("Мне", this.age, "лет");      }  } |

Таким образом, мы определили в классе Person состояние (поля name и age) и поведение (метод say). Теперь мы можем создавать объекты класса Person - конкретных людей, которые будут обладать подобным состоянием и поведением. Обычно для создания объектов применяется конструктор - специальный метод, который выполняет инициализацию объекта. Во многих языках можно использовать конструктор по умолчанию, а можно определить свой. Например, мы хотим, чтобы при создании у объектов (конкретных людей) уже были установлены имя и возраст. Для этого добавим в класс конструктор. В Java метод конструктора называется по имени класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Person{      String name;    // имя человека      int age;       // возраст человека        public Person(String name, int age){          this.name=name;          this.age = age;      }        public void say(){          System.out.printf("Меня зовут %s \n", name);          System.out.printf("Мне %d лет \n", age);      }  } |

Здесь конструктор получает извне через два параметра name и age значения для одноименных переменных.

В JavaScript для определения конструктора применяется метод со специальным именем **constructor**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class Person{        name;   // имя      age;    // возраст      constructor(name, age){          this.name = name;          this.age = age;      }      say(){          console.log("Меня зовут", this.name);          console.log("Мне", this.age, "лет");      }  } |

Таким образом, у нас готова асбтракция человека в виде класса Person, и мы ее можем использовать для создания объекта данного класса. Например, создание объектов в Java:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | public class Program{      public static void main(String[] args) {            // создаем объект класса Person          Person tom = new Person("Tom", 39);          // обращаемся к методу say объекта          tom.say();            // создаем второй объект класса Person          Person sam = new Person("Sam", 25);          // обращаемся к методу say объекта          sam.say();      }  }  // класс человека  class Person{      String name;    // имя человека      int age;       // возраст человека        public Person(String name, int age){            this.name=name;          this.age = age;      }        public void say(){          System.out.printf("Меня зовут %s \n", name);          System.out.printf("Мне %d лет \n", age);      }  } |

Аналогичная программа на JavaScript:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | class Person{        name;   // имя      age;    // возраст      constructor(name, age){          this.name = name;          this.age = age;      }      say(){          console.log("Меня зовут", this.name);          console.log("Мне", this.age, "лет");      }  }  // создаем объект класса Person  const tom = new Person("Tom", 39);  // обращаемся к методу say объекта  tom.say();    // создаем второй объект класса Person  const sam = new Person("Sam", 25);  // обращаемся к методу say объекта  sam.say(); |

В обоих версиях создаются два объекта - tom и sam, у которых вызывается метод say. В обоих языках создание объекта в принципе аналогично: для этого применяется ключевое слово new после которого идет вызов конструктора, в который передаются значения для параметров. В других языках создание объектов может отличаться, но в приниципе оно будет аналогичным.

В обоих случаях мы получим один и тот же консольный вывод.

Меня зовут Tom

Мне 39 лет

Меня зовут Sam

Мне 25 лет

**Инкапсуляция**

Инкапсуляция данных (или сокрытие информации) представляет группировку свойств и методов в классы, при которой детали реализации остаются скрытыми и защищены от нежелательного изменения. Часто под инкапсуляцию попадает состояние объекта, а также методы, которые должны применяться только внутри класса. Например, в примере выше класс Person имеет поле age, которое хранит возраст. При прямом доступе к полю age ему можно присвоить любое значение, в том числе и некорректно значение возраста, например:

|  |  |
| --- | --- |
| 1  2 | const tom = new Person("Tom", 39);  tom.age = 12345; |

Вместо прямого доступа класс предоставляет специальные методы установки и получения значения полей. Такие методы доступа могут защитить от присвоения полям недопустимых значений.

В таких языках программирования, как Java, можно использовать специальные ключевые слова - модификаторы доступа, чтобы управлять доступом к полям класса извне. Например, ключевое слово **private** при применении к полям/методам класса делает из нелоступными извне:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | class Person{      private String name;    // приватное поле доступно только внутри класса      private int age;       // приватное поле доступно только внутри класса        public Person(String name, int age){          this.name=name;          this.age = age;      }        public void say(){          System.out.printf("Меня зовут %s \n", name);          System.out.printf("Мне %d лет \n", age);      }  }  public class Program{      public static void main(String[] args) {            Person tom = new Person("Tom", 39);          tom.age = 12345;    // ! Ошибка  - поле age не доступно вне класса Person      }  } |

В других языках для скрытия состояния и поведения класса извне могут применяться другие инструменты. Так, JavaScript можно закрыть поля и методы внутри класса можно, поместив в начало названия полей/методов символ решетки **#**:

class Person{

#name; // имя

#age; // возраст

constructor(name, age){

this.#name = name;

this.#age = age;

}

say(){

console.log("Меня зовут", this.#name);

console.log("Мне", this.#age, "лет");

}

}

const tom = new Person("Tom", 39);

tom.#age = 12345; // ! Ошибка - свойство #age не доступно вне класса Person

Тем не менее даже к инкапсулированному состоянию может потребоваться доступ. Например, мы хотим устанавливать для свойства age новые значения, если они представляют корректный возраст. Или мы хотим получать значение свойства name.

Для этого в классе обычно определяются специальные методы доступа, которые опосредуют доступ к приватным полям. Например, в Java:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | public class Program{      public static void main(String[] args) {            Person tom = new Person("Tom", 39);          System.out.println(tom.getName());  // Tom          System.out.println(tom.getAge());  // 39          tom.setAge(22);          System.out.println(tom.getAge());  // 22          tom.setAge(1222);          System.out.println(tom.getAge());  // 22      }  }  // класс человека  class Person{      private String name;    // имя человека      private int age;       // возраст человека        public Person(String name, int age){            this.name=name;          this.age = age;      }      //  метод для получения имени      public String getName(){ return name; }      //  метод для получения возраста      public int getAge(){return age;}      // метод для установки возраста      public void setAge(int value){          // если переданное значение представляет валидный возраст, изменяем age          if(value > 0 && value < 110) age = value;      }        public void say(){          System.out.printf("Меня зовут %s \n", name);          System.out.printf("Мне %d лет \n", age);      }  } |

Здесь для получения имени определен метод getName, для получения возраста - метод getAge, а для установки возраста - метод setAge. Причем метод setAge изменяет возраст, если он представляет валидное значение (от 1 до 109). Для поля name также можно было бы определить метод для установки значения, но в данном случае предположим, что свойство name будет доступно только для чтения (в реальной жизни не так уж часто меняют имя).

В данном случае методы getName/getAge/setAge еще называются **методами доступа**. Методы getName/getAge называются "геттерами" (getters), так как они получают значение, а метод setAge - сеттером (setter), так как он устанавливает значение.

В разных языках для определения геттеров и сеттеров могут быть специальные конструкции. Например, в JavaScript есть специальные конструкции **get** и **set**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | class Person{        #name;   // имя      #age;    // возраст      constructor(name, age){          this.#name = name;          this.#age = age;      }      get name(){     // геттер для имени          return this.#name;      }      set age(value){ // сеттер для возраста          if(value > 0 && value < 110) this.#age = value;      }      get age(){      // геттер для возраста          return this.#age;      }      say(){          console.log("Меня зовут", this.#name);          console.log("Мне", this.#age, "лет");      }  }    const tom = new Person("Tom", 39);  console.log(tom.name);  // Tom  console.log(tom.age);  // 39  tom.age = 22;  console.log(tom.age);  // 22  tom.age =1222;  console.log(tom.age);  // 22 |

**Наследование**

Третий принцип ООП представляет наследование, который предполагает, что один класс может наследовать от другого поля/свойства и методы. Например, выше мы определили класс человека. Теперь определим класс работника некоторой компании. Работник по сути имеет те же признаки, что и Person, - имя и возраст и то же поведение, добавляя ним свои дополнительные - например, название компании, где работает работник. В этом случае мы можем создать класс работника и унаследовать его от класса Person. В различных языках для установки наследования между классами могут применяться различные операторы. В Java для этого применяется оператор **extends**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | public class Program{      public static void main(String[] args) {            Employee sam = new Employee("Sam", 25, "Google");          sam.say();      }  }  // класс человека  class Person{      private String name;    // имя человека      private int age;       // возраст человека        public Person(String name, int age){            this.name=name;          this.age = age;      }        public void say(){          System.out.printf("Меня зовут %s \n", name);          System.out.printf("Мне %d лет \n", age);      }  }  // Employee унаследован от класса Person  class Employee extends Person{        private String company;     //  компания работника      public Employee(String name, int age, String company){          super(name, age);   //  вызываем конструктор класса Person          this.company = company;      }      // переопределяем метод say      @Override      public void say(){          super.say();    // вызываем метод say из класса Person          System.out.printf("Я работаю в %s \n", company);      }  } |

Здесь определен класс Employee, который наследуется от класса Person и который добавляет поле company для хранения названия компании. В этом отношении класс Person еще называется базовым классом, родительским классом, суперклассом. А класс Employee - классом-наследником, производным классом, подклассом.

Чтобы при создании работника получить это свойство извне, в классе Employee определен свой конструктор, который принимает три параметра - имя, возраст и компанию. В ряде распространенных объектно-ориентированных языков конструктор класса-наследника должен обязательно вызывать конструктор базового класса. В Java это делается с помощью ключевого слова **super**

|  |  |
| --- | --- |
| 1 | super(name, age); |

Кроме того, класс-наследник обычно может не просто унаследовать функционал, но и переопределить его при необходимости. В данном случае переопределяем метод say, для чего к нему в Java применяется аннотация @Override. Например, мы хотим кроме имени и возраста выводить также и компанию работника:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Override  public void say(){      super.say();    // вызываем метод say из класса Person      System.out.printf("Я работаю в %s \n", company);  } |

Здесь сначала обращаемся к методу say базового класса Person - super.say(). Это, во-первых, позволяет избежать ненужного дублирования кода (не надо писать строки для вывода на консоль имени и возраста). Во-вторых, поля name и age в Person - приватные и поэтому в классах-наследниках недоступны. Для обращения к функционалу базового класса в Java также используется слово **super**

Аналогичная программа на JavaScript:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | class Person{        #name;   // имя      #age;    // возраст      constructor(name, age){          this.#name = name;          this.#age = age;      }      say(){          console.log("Меня зовут", this.#name);          console.log("Мне", this.#age, "лет");      }  }  // класс Employee унаследован от Person  class Employee extends Person{        #company; // компания работника      constructor(name, age, company){          super(name, age);   // вызываем конструктор базового класса Person          this.#company = company;      }      say(){          super.say();    // вызов реализации метода say из класса Person          console.log("Я работаю в", this.#company);      }  }    const sam = new Employee("Sam", 25, "Google");  sam.say(); |

Консольный вывод программ на обоих языках будет идентичен:

Меня зовут Sam

Мне 25 лет

Я работаю в Google

**Полиморфизм**

Полиморфизм представляет способность объектов принимать другой тип (форму) или представлять себя как другой тип в зависимости от контекста или их использования. Применение к классам наследования позволяет установить связь "является". То есть работник является человеком или объект Employee является Person. Соответственно, если, к примеру, функция ожидает объект типа Person, то в качестве аргументов могут быть переданы как объекты Person, так и объекты Employee:

Так, определим следующую программу на Java:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | public class Program{      public static void main(String[] args) {            Person tom = new Person("Tom", 39);          Employee sam = new Employee("Sam", 25, "Google");          printPerson(tom);          printPerson(sam);        }      public static void printPerson(Person person){          person.say();          System.out.println();      }  }  // класс человека  class Person{      private String name;    // имя человека      private int age;       // возраст человека        public Person(String name, int age){            this.name=name;          this.age = age;      }        public void say(){          System.out.printf("Меня зовут %s \n", name);          System.out.printf("Мне %d лет \n", age);      }  }  // Employee унаследован от класса Person  class Employee extends Person{        private String company;     //  компания работника      public Employee(String name, int age, String company){          super(name, age);   //  вызываем конструктор класса Person          this.company = company;      }      // переопределяем метод say      @Override      public void say(){          super.say();    // вызываем метод say из класса Person          System.out.printf("Я работаю в %s \n", company);      }  } |

Здесь метод printPerson принимает объект Person:

|  |  |
| --- | --- |
| 1  2  3  4 | public static void printPerson(Person person){      person.say();      System.out.println();  } |

Но так как объект Employee - это тоже объект Person, то в эту функцию также можно передать объект Employee

|  |  |
| --- | --- |
| 1  2  3  4 | Person tom = new Person("Tom", 39);  Employee sam = new Employee("Sam", 25, "Google");  printPerson(tom);  printPerson(sam); |

Для JavaScript в силу слабой типизации принцип полиморфизма меньше характерен, так как параметру функции мы можем передать любой объект:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | class Person{        #name;   // имя      #age;    // возраст      constructor(name, age){          this.#name = name;          this.#age = age;      }      say(){          console.log("Меня зовут", this.#name);          console.log("Мне", this.#age, "лет");      }  }  class Employee extends Person{        #company; // компания работника      constructor(name, age, company){          super(name, age);   // вызываем конструктор базового класса Person          this.#company = company;      }      say(){          super.say();    // вызов реализации метода say из класса Person          console.log("Я работаю в", this.#company);      }  }    function printPerson(person){      person.say();      console.log("\n");  }  const tom = new Person("Tom", 39);  const sam = new Employee("Sam", 25, "Google");  printPerson(tom);  printPerson(sam); |

Консольный вывод программ на обоих языках:

Меня зовут Tom

Мне 39 лет

Меня зовут Sam

Мне 25 лет

Я работаю в Google

## C#. Типы данных. Переменные, операции, выражения.

В языке C# есть следующие базовые типы данных:

* **bool**: хранит значение true или false (логические литералы). Представлен системным типом System.Boolean

|  |  |
| --- | --- |
| 1  2 | bool alive = true;  bool isDead = false; |

* **byte**: хранит целое число от 0 до 255 и занимает 1 байт. Представлен системным типом System.Byte

|  |  |
| --- | --- |
| 1  2 | byte bit1 = 1;  byte bit2 = 102; |

* **sbyte**: хранит целое число от -128 до 127 и занимает 1 байт. Представлен системным типом System.SByte

|  |  |
| --- | --- |
| 1  2 | sbyte bit1 = -101;  sbyte bit2 = 102; |

* **short**: хранит целое число от -32768 до 32767 и занимает 2 байта. Представлен системным типом System.Int16

|  |  |
| --- | --- |
| 1  2 | short n1 = 1;  short n2 = 102; |

* **ushort**: хранит целое число от 0 до 65535 и занимает 2 байта. Представлен системным типом System.UInt16

|  |  |
| --- | --- |
| 1  2 | ushort n1 = 1;  ushort n2 = 102; |

* **int**: хранит целое число от -2147483648 до 2147483647 и занимает 4 байта. Представлен системным типом System.Int32. Все целочисленные литералы по умолчанию представляют значения типа int:

|  |  |
| --- | --- |
| 1  2  3 | int a = 10;  int b = 0b101;  // бинарная форма b =5  int c = 0xFF;   // шестнадцатеричная форма c = 255 |

* **uint**: хранит целое число от 0 до 4294967295 и занимает 4 байта. Представлен системным типом System.UInt32

|  |  |
| --- | --- |
| 1  2  3 | uint a = 10;  uint b = 0b101;  uint c = 0xFF; |

* **long**: хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт. Представлен системным типом System.Int64

|  |  |
| --- | --- |
| 1  2  3 | long a = -10;  long b = 0b101;  long c = 0xFF; |

* **ulong**: хранит целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт. Представлен системным типом System.UInt64

|  |  |
| --- | --- |
| 1  2  3 | ulong a = 10;  ulong b = 0b101;  ulong c = 0xFF; |

* **float**: хранит число с плавающей точкой от -3.4\*1038 до 3.4\*1038 и занимает 4 байта. Представлен системным типом System.Single
* **double**: хранит число с плавающей точкой от ±5.0\*10-324 до ±1.7\*10308 и занимает 8 байта. Представлен системным типом System.Double
* **decimal**: хранит десятичное дробное число. Если употребляется без десятичной запятой, имеет значение от ±1.0\*10-28 до ±7.9228\*1028, может хранить 28 знаков после запятой и занимает 16 байт. Представлен системным типом System.Decimal
* **char**: хранит одиночный символ в кодировке Unicode и занимает 2 байта. Представлен системным типом System.Char. Этому типу соответствуют символьные литералы:

|  |  |
| --- | --- |
| 1  2  3 | char a = 'A';  char b = '\x5A';  char c = '\u0420'; |

* **string**: хранит набор символов Unicode. Представлен системным типом System.String. Этому типу соответствуют строковые литералы.

|  |  |
| --- | --- |
| 1  2 | string hello = "Hello";  string word = "world"; |

* **object**: может хранить значение любого типа данных и занимает 4 байта на 32-разрядной платформе и 8 байт на 64-разрядной платформе. Представлен системным типом System.Object, который является базовым для всех других типов и классов .NET.

|  |  |
| --- | --- |
| 1  2  3 | object a = 22;  object b = 3.14;  object c = "hello code"; |

Однако мы можем использовать и модель неявной типизации:

|  |  |
| --- | --- |
| 1  2 | var hello = "Hell to World";  var c = 20; |

Для неявной типизации вместо названия типа данных используется ключевое слово var. Затем уже при компиляции компилятор сам выводит тип данных исходя из присвоенного значения. Так как по умолчанию все целочисленные значения рассматриваются как значения типа int, то поэтому в итоге переменная c будет иметь тип int. Аналогично переменной hello присваивается строка, поэтому эта переменная будет иметь тип string

Эти переменные подобны обычным, однако они имеют некоторые ограничения.

Во-первых, мы не можем сначала объявить неявно типизируемую переменную, а затем инициализировать:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | // этот код работает  int a;  a = 20;    // этот код не работает  var c;  c= 20; |

Во-вторых, мы не можем указать в качестве значения неявно типизируемой переменной null:

|  |  |
| --- | --- |
| 1  2 | // этот код не работает  var c=null; |

Так как значение null, то компилятор не сможет вывести тип данных.

Для хранения данных в программе применяются **переменные**. Переменная представляет именнованную область памяти, в которой хранится значение определенного типа. Переменная имеет тип, имя и значение. Тип определяет, какого рода информацию может хранить переменная.

Перед использованием любую переменную надо определить. Синтаксис определения переменной выглядит следующим образом:

|  |  |
| --- | --- |
| 1 | тип имя\_переменной; |

Вначале идет тип переменной, потом ее имя. В качестве имени переменной может выступать любое произвольное название, которое удовлетворяет следующим требованиям:

* имя может содержать любые цифры, буквы и символ подчеркивания, при этом первый символ в имени должен быть буквой или символом подчеркивания
* в имени не должно быть знаков пунктуации и пробелов
* имя не может быть ключевым словом языка C#. Таких слов не так много, и при работе в Visual Studio среда разработки подсвечивает ключевые слова синим цветом.

Хотя имя переменой может быть любым, но следует давать переменным описательные имена, которые будут говорить об их предназначении.

Например, определим простейшую переменную:

|  |  |
| --- | --- |
| 1 | string name; |

В данном случае определена переменная name, которая имеет тип **string**. то есть переменная представляет строку. Поскольку определение переменной представляет собой инструкцию, то после него ставится точка с запятой.

При этом следует учитывать, что C# является регистрозависимым языком, поэтому следующие два определения переменных будут представлять две разные переменные:

|  |  |
| --- | --- |
| 1  2 | string name;  string Name; |

После определения переменной можно присвоить некоторое значение:

|  |  |
| --- | --- |
| 1  2 | string name;  name = "Tom"; |

Так как переменная name представляет тип string, то есть строку, то мы можем присвоить ей строку в двойных кавычках. Причем переменной можно присвоить только то значение, которое соответствует ее типу.

Следующие операторы выполняют арифметические операции с операндами числовых типов:

унарные — ++ (приращение), -- (уменьшение), + (плюс) и - (минус);

бинарные — \* (умножение), / (деление), % (остаток от деления), + (сложение) и - (вычитание).

Эти операторы поддерживаются всеми целочисленными типами и типами с плавающей запятой.

В следующем списке перечислены арифметические операторы в порядке убывания приоритета:

* Постфиксный инкремент x++ и декремент x--
* Префиксный инкремент ++x и декремент --x, унарные операторы + и -
* Мультипликативные операторы \*, /, и %
* Аддитивные операторы + и -

Бинарные арифметические операторы имеют левую ассоциативность. То есть операторы с одинаковым приоритетом вычисляются в направлении слева направо.

Порядок вычисления, определяемый приоритетом и ассоциативностью операторов, можно изменить с помощью скобок (()).

**Логические операции**

* **&**(логическое умножение)

Умножение производится поразрядно, и если у обоих операндов значения разрядов равно 1, то операция возвращает 1, иначе возвращается число 0.

**|** (логическое сложение)

Похоже на логическое умножение, операция также производится по двоичным разрядам, но теперь возвращается единица, если хотя бы у одного числа в данном разряде имеется единица.

**^** (логическое исключающее ИЛИ)

Также эту операцию называют XOR, нередко ее применяют для простого шифрования

**~** (логическое отрицание или инверсия)

Еще одна поразрядная операция, которая инвертирует все разряды: если значение разряда равно 1, то оно становится равным нулю, и наоборот.

Операции сдвига также производятся над разрядами чисел. Сдвиг может происходить вправо и влево.

* x<<y - сдвигает число x влево на y разрядов. Например, 4<<1 сдвигает число 4 (которое в двоичном представлении 100) на один разряд влево, то есть в итоге получается 1000 или число 8 в десятичном представлении.
* x>>y - сдвигает число x вправо на y разрядов. Например, 16>>1 сдвигает число 16 (которое в двоичном представлении 10000) на один разряд вправо, то есть в итоге получается 1000 или число 8 в десятичном представлении.

Кроме базовой операции присвоения в C# есть еще ряд операций:

* **+=**: присваивание после сложения. Присваивает левому операнду сумму левого и правого операндов: выражение **A += B** равнозначно выражению **A = A + B**
* **-=**: присваивание после вычитания. Присваивает левому операнду разность левого и правого операндов: **A -= B** эквивалентно **A = A - B**
* **\*=**: присваивание после умножения. Присваивает левому операнду произведение левого и правого операндов: **A \*= B** эквивалентно **A = A \* B**
* **/=**: присваивание после деления. Присваивает левому операнду частное левого и правого операндов: **A /= B** эквивалентно **A = A / B**
* **%=**: присваивание после деления по модулю. Присваивает левому операнду остаток от целочисленного деления левого операнда на правый: **A %= B** эквивалентно **A = A % B**
* **&=**: присваивание после поразрядной конъюнкции. Присваивает левому операнду результат поразрядной конъюнкции его битового представления с битовым представлением правого операнда: **A &= B** эквивалентно **A = A & B**
* **|=**: присваивание после поразрядной дизъюнкции. Присваивает левому операнду результат поразрядной дизъюнкции его битового представления с битовым представлением правого операнда: **A |= B** эквивалентно **A = A | B**
* **^=**: присваивание после операции исключающего ИЛИ. Присваивает левому операнду результат операции исключающего ИЛИ его битового представления с битовым представлением правого операнда: **A ^= B** эквивалентно **A = A ^ B**
* **<<=**: присваивание после сдвига разрядов влево. Присваивает левому операнду результат сдвига его битового представления влево на определенное количество разрядов, равное значению правого операнда: **A <<= B** эквивалентно **A = A << B**
* **>>=**: присваивание после сдвига разрядов вправо. Присваивает левому операнду результат сдвига его битового представления вправо на определенное количество разрядов, равное значению правого операнда: **A >>= B** эквивалентно **A = A >> B**

## C#. Операторы цикла, условный, выбора, перехода.

Операторы итерации многократно выполняют инструкцию или блок инструкций. [Оператор](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/iteration-statements#the-for-statement)for выполняет текст, пока указанное логическое выражение вычисляетсяtrue. [Инструкция](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/iteration-statements#the-foreach-statement)foreach перечисляет элементы коллекции и выполняет его текст для каждого элемента коллекции. [Оператор](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/iteration-statements#the-do-statement)do условно выполняет свой текст один или несколько раз. [Оператор](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/iteration-statements#the-while-statement)while условно выполняет его тело ноль или более раз.

В любой момент в тексте инструкции итерации можно выйти из цикла с помощью инструкции[break](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/jump-statements#the-break-statement). Вы можете перейти к следующей итерации в цикле с помощью инструкции[continue](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/jump-statements#the-continue-statement).

for (int i = 0; i < 3; i++)

{

Console.Write(i);

}

Раздел итератора может содержать ноль или более следующих выражений оператора, разделенных запятыми:

* префиксное или постфиксное выражение [приращения](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/arithmetic-operators#increment-operator-), такое как ++i или i++
* префиксное или постфиксное выражение [декремента](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/arithmetic-operators#decrement-operator---), такое как --i или i--
* [присваивание](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/assignment-operator)
* вызов метода
* [Выражение await](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/await)
* создание объекта с помощью [new оператора](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/new-operator)

Оператор foreach выполняет оператор или блок операторов для каждого элемента в экземпляре типа, который реализует интерфейс [System.Collections.IEnumerable](https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.ienumerable) или [System.Collections.Generic.IEnumerable<T>](https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.generic.ienumerable-1)

foreach (int element in fibNumbers)

{

Console.Write($"{element} ");

}

 Его можно использовать с экземпляром любого типа, который удовлетворяет следующим условиям:

* Тип имеет открытый метод без параметров GetEnumerator. Этот GetEnumerator метод может быть методом[расширения типа](https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/extension-methods).
* тип возвращаемого значения метода GetEnumerator должен содержать открытое свойство Current и открытый метод MoveNext без параметров с типом возвращаемого значения bool.

Инструкцию await foreach можно использовать для использования асинхронного потока данных, то есть типа коллекции, реализующего [IAsyncEnumerable<T>](https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.generic.iasyncenumerable-1) интерфейс. Каждую итерацию цикла можно приостановить, пока будет осуществляться асинхронное извлечение следующего элемента. В следующем примере показано использование оператора await foreach.

C#Копировать

await foreach (var item in GenerateSequenceAsync())

{

Console.WriteLine(item);

}

Оператор await foreach можно также использовать с экземпляром любого типа, который удовлетворяет следующим условиям:

* Тип имеет открытый метод без параметров GetAsyncEnumerator. Этот метод может быть [методом расширения](https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/extension-methods) типа.
* Тип возвращаемого значения метода GetAsyncEnumerator имеет открытое свойство Current и открытый метод без параметров MoveNextAsync, тип возвращаемого значения которого — [Task<bool>](https://learn.microsoft.com/ru-ru/dotnet/api/system.threading.tasks.task-1), [ValueTask<bool>](https://learn.microsoft.com/ru-ru/dotnet/api/system.threading.tasks.valuetask-1) или любой другой подтверждающий ожидание тип, метод ожидания которого GetResult возвращает значение bool.

Оператор do выполняет оператор или блок операторов, пока определенное логическое выражение равно значению true. Так как это выражение оценивается после каждого выполнения цикла, цикл do выполняется один или несколько раз. Цикл do отличается от [while цикла](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/iteration-statements#the-while-statement), который выполняется нулевым или более раз.

В следующем примере показано применение оператора do.

C#

int n = 0;

do

{

Console.Write(n);

n++;

} while (n < 5);

Оператор while выполняет оператор или блок операторов, пока определенное логическое выражение равно значению true. Так как это выражение оценивается перед каждым выполнением цикла, цикл while выполняется ноль или несколько раз. Цикл while отличается от [do цикла](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/iteration-statements#the-do-statement), который выполняется один или несколько раз.

В следующем примере показано применение оператора while.

C#

int n = 0;

while (n < 5)

{

Console.Write(n);

n++;

}

if-elseswitch Операторы ifвыбора и инструкции для выполнения из множества возможных путей на основе значения выражения. Оператор [if выполняет инструкцию](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/selection-statements#the-if-statement) только в том случае, если заданное логическое выражение вычисляется true. [Оператор](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/selection-statements#the-if-statement)if-else позволяет выбрать, какие из двух путей кода следует выполнять на основе логического выражения. [Оператор](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/selection-statements#the-switch-statement)switch выбирает список инструкций для выполнения на основе сопоставления шаблона с выражением.

if (char.IsUpper(ch))

{

Console.WriteLine($"An uppercase letter: {ch}");

}

switch (measurement)

{

case < 0.0:

Console.WriteLine($"Measured value is {measurement}; too low.");

break;

Операторы перехода безоговорочно передают управление. [Оператор](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/jump-statements#the-break-statement)break завершает ближайшее заключающее [оператор](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/iteration-statements) итерации или [switch инструкцию](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/selection-statements#the-switch-statement). [Оператор](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/jump-statements#the-continue-statement)continue запускает новую итерацию ближайшего заключенного [оператора](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/iteration-statements) итерации. Оператор [return](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/jump-statements#the-return-statement) завершает выполнение функции, в которой она отображается, и возвращает элемент управления вызывающему объекту. [Оператор](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/jump-statements#the-goto-statement)goto передает элемент управления инструкции, помеченной меткой.

Оператор break завершает выполнение ближайшего внешнего [оператора итерации](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/iteration-statements) (то есть цикла for, foreach, while или do) или [оператора switch](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/selection-statements#the-switch-statement). Оператор break передает управление оператору, который расположен после завершенного оператора (если таковой есть).

Оператор continue начинает новую итерацию ближайшего внешнего [оператора итерации](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/iteration-statements) (то есть цикла for, foreach, while или do),

Возвращаемое ref значение — это псевдоним другой переменной в область вызываемого метода. Любое применение возвращаемого ссылочного значения можно рассматривать как применение псевдонима соответствующей переменной.

* При назначении его значения вы назначаете значение переменной, которую он псевдонимирует.
* При чтении его значения вы считываете значение переменной, которую он псевдонимирует.
* Если вы возвращаете его *по ссылке*, вы возвращаете псевдоним в ту же переменную.
* Если передать его другому методу *по ссылке*, вы передаете ссылку на переменную, которую он псевдонимирует.
* Создавая для псевдонима [локальную ссылочную переменную](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/declarations#reference-variables), вы создаете новый псевдоним для той же переменной.

Возвращаемое значение ref должно быть [*ref-safe-context*](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/method-parameters#safe-context-of-references-and-values) для вызывающего метода. Это означает:

* Время существования возвращаемого значения должно превышать период выполнения метода. Другими словами, это не может быть локальная переменная в методе, который возвращает его. Это может быть экземпляр статического поля или класса, а также переданный в метод аргумент. Попытка вернуть локальную переменную создает ошибку компилятора CS8168, "Не удается вернуть локальный obj по ссылке, так как он не является локальным.
* Возвращаемое значение не может быть литеральным null. Метод с возвратом ссылок может возвращать псевдоним переменной, значение которой в настоящее время null является значением (неинстанизированным) или [типом значения](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/nullable-value-types) NULL для типа значения.
* Возвращаемое значение не может быть константой, элементом перечисления, возвращаемым значением из свойства или методом или classstruct.

Кроме того, значения возвращаемых ссылок не допускаются для асинхронных методов. Асинхронный метод может вернуть управление до того, как будет завершено его выполнение и станет известно его возвращаемое значение.

Метод, возвращающий *возвращаемое значение* ссылки, должен:

* Добавьте ссылку ключевое слово перед типом возвращаемого значения.
* Каждый оператор [return](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/jump-statements#the-return-statement) в теле метода включает ключевое слово [ref](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/ref) перед именем возвращаемого экземпляра.

for (int row = 0; row < matrix.Length; row++)

{

for (int col = 0; col < matrix[row].Length; col++)

{

if (matrix[row][col] == target)

{

goto Found;

}

}

}

Console.WriteLine($"Not found {target} in matrix {key}.");

continue;

Found:

Console.WriteLine($"Found {target} in matrix {key}.");

## C#. Классы. Структуры. Перечисления. Конструкторы.

# является полноценным объектно-ориентированным языком. Это значит, что программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов.

Описанием объекта является **класс**, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. То есть некоторый шаблон - этот шаблон можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек (фактически экземпляр данного класса) будет представлять объект этого класса.

По сути класс представляет новый тип, который определяется пользователем. Класс определяется с помощью ключевого слова **сlass**:

|  |  |
| --- | --- |
| 1  2  3  4 | class название\_класса  {      // содержимое класса  } |

Класс может хранить некоторые данные. Для хранения данных в классе применяются **поля**. По сути **поля класса** - это переменные, определенные на уровне класса.

Кроме того, класс может определять некоторое поведение или выполняемые действия. Для определения поведения в классе применяются методы.

После определения класса мы можем создавать его объекты. Для создания объекта применяются **конструкторы**. По сути конструкторы представляют специальные методы, которые называются так же как и класс, и которые вызываются при создании нового объекта класса и выполняют инициализацию объекта. Общий синтаксис вызова конструктора:

|  |  |
| --- | --- |
| 1 | new конструктор\_класса(параметры\_конструктора); |

Сначала идет оператор **new**, который выделяет память для объекта, а после него идет вызов **конструктора**.

**Конструктор по умолчанию**

Если в классе не определено ни одного конструктора (как в случае с нашим классом Person), то для этого класса автоматически создается пустой конструктор по умолчанию, который не принимает никаких параметров.

Для обращения к функциональности класса - полям, методам (а также другим элементам класса) применяется точечная нотация точки - после объекта класса ставится точка, а затем элемент класса:

|  |  |
| --- | --- |
| 1  2 | объект.поле\_класса  объект.метод\_класса(параметры\_метода) |

Обычно классы помещаются в отдельные файлы. Нередко для одного класса предназначен один файл.

Как правило, конструктор выполняет инициализацию объекта. При этом если в классе определяются свои конструкторы, то он лишается конструктора по умолчанию.

На уровне кода конструктор представляет метод, который называется по имени класса, который может иметь параметры, но для него не надо определять возвращаемый тип.

онструкторы могут иметь модификаторы, которые указываются перед именем конструктора. Так, в данном случае, чтобы конструктор был доступен вне класса Person, он определен с модификатором public. Ключевое слово this представляет ссылку на текущий экземпляр/объект класса.

Начиная с версии C# 12 в язык C# была добавлена поддержка первичных конструкторов (Primary constructors). Первичные конструкторы позволяют добавлять параметры к определению класса и использовать эти параметры внутри класса

public class Person(string name, int age)

{

    public Person(string name) : this(name, 18) { }

    public void Print() => Console.WriteLine($"name: {name}, age: {age}");

}

аряду с классами структуры представляют еще один способ создания собственных типов данных в C#. Более того многие примитивные типы, например, int, double и т.д., по сути являются структурами.

**Определение структуры**

Для определения структуры применяется ключевое слово **struct**:

|  |  |
| --- | --- |
| 1  2  3  4 | struct имя\_структуры  {      // элементы структуры  } |

После слова **struct** идет название структуры и далее в фигурных скобках размещаются элементы структуры - поля, методы и т.д.

Начиная с версии C# 12, если структура имеет пустое определение (не содержат полей, свойств, методов), то фигурные скобки после названия типа можно не использовать:

|  |  |
| --- | --- |
| 1 | struct Person; |

Как и классы, структуры могут хранить состояние в виде полей (переменных) и определять поведение в виде методов.

И как и в случае с классами, для обращения к функциональности структуры - полям, методам и другим компонентам структуры применяется точечная нотация - после объекта структуры ставится точка, а затем указывается компонент структуры:

|  |  |
| --- | --- |
| 1  2 | объект.поле\_структуры  объект.метод\_структуры(параметры\_метода) |

сли нам необходимо скопировать в один объект структуры значения из другого с небольшими изменениями, то мы можем использовать оператор **with**:

|  |  |
| --- | --- |
| 1  2  3 | Person tom = new Person { name = "Tom", age = 22 };  Person bob = tom with { name = "Bob" };  bob.Print();    // Имя: Bob  Возраст: 22 |

В данном случае объект bob получает все значения объекта tom, а затем после оператора **with** в фигурных скобках указывается поля со значениями, которые мы хотим изменить.

Кроме примитивных типов данных в языке программирования C# есть такой тип как **enum** или **перечисление**. Перечисления представляют набор логически связанных констант.

Объявление перечисления происходит с помощью оператора **enum**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | enum название\_перечисления  {      // значения перечисления      значение1,      значение2,      .......      значениеN  } |

После оператора enum идет название перечисления. И затем в фигурных скобках через запятую перечисляются константы перечисления.

Зачастую переменная перечисления выступает в качестве хранилища состояния, в зависимости от которого производятся некоторые действия

Константы перечисления могут иметь тип. Тип указывается после названия перечисления через двоеточие:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | enum Time : byte  {      Morning,      Afternoon,      Evening,      Night  } |

Тип перечисления обязательно должен представлять целочисленный тип (byte, sbyte, short, ushort, int, uint, long, ulong). Если тип явным образом не указан, то по умолчанию используется тип **int**.

Тип влияет на значения, которые могут иметь константы. По умолчанию каждому элементу перечисления присваивается целочисленное значение, причем первый элемент будет иметь значение 0, второй - 1 и так далее.

## C#. Индексаторы. Семейства. Клонирование. Итераторы.

**Индексаторы** позволяют индексировать объекты и обращаться к данным по индексу. Фактически с помощью индексаторов мы можем работать с объектами как с массивами. По форме они напоминают свойства со стандартными блоками get и set, которые возвращают и присваивают значение.

Формальное определение индексатора:

|  |  |
| --- | --- |
| 1  2  3  4  5 | возвращаемый\_тип this [Тип параметр1, ...]  {      get { ... }      set { ... }  } |

В отличие от свойств индексатор не имеет названия. Вместо него указывается ключевое слово **this**, после которого в квадратных скобках идут параметры. Индексатор должен иметь как минимум один параметр.

Посмотрим на примере. Допустим, у нас есть класс **Person**, который представляет человека, и класс **Company**, который представляет некоторую компанию, где работают люди. Используем индексаторы для определения класса Company:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class Person  {      public string Name { get;}      public Person(string name) => Name=name;  }  class Company  {      Person[] personal;      public Company(Person[] people) => personal = people;      // индексатор      public Person this[int index]      {          get => personal[index];          set => personal[index] = value;      }  } |

Для возвращения объекта в индексаторе определен блок **get**:

|  |  |
| --- | --- |
| 1 | get => personal[index]; |

Поскольку индексатор имеет тип Person, то в блоке get нам надо возвратить объект этого типа с помощью оператора return. Здесь мы можем определить разнообразную логику. В данном случае просто возвращаем объект из массива personal.

В блоке **set**, как и в обычном свойстве, получаем через параметр **value** переданный объект Person и сохраняем его в массив по индексу.

|  |  |
| --- | --- |
| 1 | set => personal[index] = value; |

Подобно методам индексаторы можно перегружать. В этом случае также индексаторы должны отличаться по количеству, типу или порядку используемых параметров.

// индексатор

    public Person this[int index]

    {

        get => personal[index];

        set => personal[index] = value;

    }

    public Person this[string name]

    {

        get

        {

            foreach (var person in personal)

            {

                if (person.Name == name) return person;

            }

            throw new Exception("Unknown name");

        }

    }

 Поскольку классы представляют ссылочные типы, то это накладывает некоторые ограничения на их использование. В частности, допустим, у нас есть следующий класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class Person  {      public string Name { get; set; }      public int Age { get; set; }      public Person(string name, int age)      {          Name = name;          Age = age;      }  } |

Создадим один объект Person и попробуем скопировать его данные в другой объект Person:

|  |  |
| --- | --- |
| 1  2  3  4 | var tom = new Person("Tom", 23);  var bob = tom;  bob.Name = "Bob";  Console.WriteLine(tom.Name); // Bob |

В данном случае объекты tom и bob будут указывать на один и тот же объект в памяти, поэтому изменения свойств для переменной bob затронут также и переменную tom.

Чтобы переменная bob указывала на новый объект, но при этом имела значения из переменной tom, мы можем применить клонирование с помощью реализации интерфейса **ICloneable**:

|  |  |
| --- | --- |
| 1  2  3  4 | public interface ICloneable  {      object Clone();  } |

**Поверхностное копирование**

Реализация интерфейса в классе Person могла бы выглядеть следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Person : ICloneable  {      public string Name { get; set; }      public int Age { get; set; }      public Person(string name, int age)      {          Name = name;          Age = age;      }      public object Clone()      {          return new Person(Name, Age);      }  } |

Использование:

|  |  |
| --- | --- |
| 1  2  3  4 | var tom = new Person("Tom", 23);  var bob = (Person)tom.Clone();  bob.Name = "Bob";  Console.WriteLine(tom.Name); // Tom |

Теперь все нормально копируется, изменения в свойствах переменной bob не сказываются на свойствах из переменной tom.

Для сокращения кода копирования мы можем использовать специальный метод **MemberwiseClone()**, который возвращает копию объекта:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Person : ICloneable  {      public string Name { get; set; }      public int Age { get; set; }      public Person(string name, int age)      {          Name = name;          Age = age;      }      public object Clone()      {          return MemberwiseClone();      }  } |

Этот метод реализует **поверхностное (неглубокое) копирование**. Однако данного копирования может быть недостаточно. Например, пусть класс Person содержит ссылку на объект класса Company:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class Person : ICloneable  {      public string Name { get; set; }      public int Age { get; set; }      public Company Work { get; set; }      public Person(string name, int age, Company company)      {          Name = name;          Age = age;          Work = company;      }      public object Clone() => MemberwiseClone();  }  class Company  {      public string Name { get; set; }      public Company(string name) => Name = name;  } |

В этом случае при копировании новая копия будет указывать на тот же объект Company:

|  |  |
| --- | --- |
| 1  2  3  4 | var tom = new Person("Tom", 23, new Company("Microsoft"));  var bob = (Person)tom.Clone();  bob.Work.Name = "Google";  Console.WriteLine(tom.Work.Name); // Google - а должно быть Microsoft |

**Глубокое копирование**

Поверхностное копирование работает только для свойств, представляющих примитивные типы, но не для сложных объектов. И в этом случае надо применять **глубокое копирование**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class Person : ICloneable  {      public string Name { get; set; }      public int Age { get; set; }      public Company Work { get; set; }      public Person(string name, int age, Company company)      {          Name = name;          Age = age;          Work = company;      }      public object Clone() => new Person(Name, Age, new Company(Work.Name));  }  class Company  {      public string Name { get; set; }      public Company(string name) => Name = name;  } |

**Итератор** по сути представляет блок кода, который использует оператор **yield** для перебора набора значений. Данный блок кода может представлять тело метода, оператора или блок get в свойствах.

Итератор использует две специальных инструкции:

* **yield return**: определяет возвращаемый элемент
* **yield break**: указывает, что последовательность больше не имеет элементов

Рассмотрим небольшой пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | Numbers numbers = new Numbers();  foreach (int n in numbers)  {      Console.WriteLine(n);  }    class Numbers  {      public IEnumerator<int> GetEnumerator()      {          for (int i = 0; i < 6; i++)          {              yield return i \* i;          }      }  } |

## C#. Интерфейсы. Делегаты. События.

Интерфейс представляет ссылочный тип, который может определять некоторый функционал - набор методов и свойств без реализации. Затем этот функционал реализуют классы и структуры, которые применяют данные интерфейсы.

**Определение интерфейса**

Для определения интерфейса используется ключевое слово **interface**. Как правило, названия интерфейсов в C# начинаются с заглавной буквы **I**, например, IComparable, IEnumerable (так называемая венгерская нотация), однако это не обязательное требование, а больше стиль программирования.

Что может определять интерфейс? В целом интерфейсы могут определять следующие сущности:

* Методы
* Свойства
* Индексаторы
* События
* Статические поля и константы (начиная с версии C# 8.0)

|  |
| --- |
| interface IMovable  {      // константа      const int minSpeed = 0;     // минимальная скорость      // статическая переменная      static int maxSpeed = 60;   // максимальная скорость      // метод      void Move();                // движение      // свойство      string Name { get; set; }   // название        delegate void MoveHandler(string message);  // определение делегата для события      // событие      event MoveHandler MoveEvent;    // событие движения  } |

В данном случае определен интерфейс IMovable, который представляет некоторый движущийся объект. Данный интерфейс содержит различные компоненты, которые описывают возможности движущегося объекта. То есть интерфейс описывает некоторый функционал, который должен быть у движущегося объекта.

Методы и свойства интерфейса могут не иметь реализации, в этом они сближаются с абстрактными методами и свойствами абстрактных классов. В данном случае интерфейс определяет метод Move, который будет представлять некоторое передвижение. Он не имеет реализации, не принимает никаких параметров и ничего не возвращает.

То же самое в данном случае касается свойства Name. На первый взгляд оно похоже на автоматическое свойство. Но в реальности это определение свойства в интерфейсе, которое не имеет реализации, а не автосвойство.

Модификаторы доступа

Еще один момент в объявлении интерфейса: если его члены - методы и свойства не имеют модификаторов доступа, то фактически по умолчанию доступ **public**, так как цель интерфейса - определение функционала для реализации его классом. Это касается также и констант и статических переменных, которые в классах и структурах по умолчанию имееют модификатор private. В интерфейсах же они имеют по умолчанию модификатор public. И например, мы могли бы обратиться к константе minSpeed и переменной maxSpeed интерфейса IMovable:

|  |  |
| --- | --- |
| 1  2 | Console.WriteLine(IMovable.maxSpeed);   // 60  Console.WriteLine(IMovable.minSpeed);   // 0 |

Но также, начиная с версии C# 8.0, мы можем явно указывать модификаторы доступа у компонентов интерфейса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | interface IMovable  {      public const int minSpeed = 0;     // минимальная скорость      private static int maxSpeed = 60;   // максимальная скорость      public void Move();      protected internal string Name { get; set; }    // название      public delegate void MoveHandler(string message);  // определение делегата для события      public event MoveHandler MoveEvent;    // событие движения  } |

Как и классы, интерфейсы по умолчанию имеют уровень доступа **internal**, то есть такой интерфейс доступен только в рамках текущего проекта. Но с помощью модификатора public мы можем сделать интерфейс общедоступным:

|  |  |
| --- | --- |
| 1  2  3  4 | public interface IMovable  {      void Move();  } |

Реализация по умолчанию

Также начиная с версии C# 8.0 интерфейсы поддерживают реализацию методов и свойств по умолчанию. Это значит, что мы можем определить в интерфейсах полноценные методы и свойства, которые имеют реализацию как в обычных классах и структурах. Например, определим реализацию метода Move по умолчанию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | interface IMovable  {      // реализация метода по умолчанию      void Move()      {          Console.WriteLine("Walking");      }  } |

С реализацией свойств по умолчанию в интерфейсах дело обстоит несколько сложнее, поскольку мы не можем определять в интерфейсах нестатические переменные, соответственно в свойствах интерфейса мы не можем манипулировать состоянием объекта. Тем не менее реализацию по умолчанию для свойств мы тоже можем определять:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | interface IMovable  {      // реализация метода по умолчанию      void Move() => Console.WriteLine("Walking");      // реализация свойства по умолчанию      // свойство только для чтения      int MaxSpeed { get { return 0; } }  } |

Стоит отметить, что если интерфейс имеет приватные методы и свойства (то есть с модификатором private), то они должны иметь реализацию по умолчанию. То же самое относится к статическим методам (не обязательно приватным):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | Console.WriteLine(IMovable.MaxSpeed);   // 60  IMovable.MaxSpeed = 65;  Console.WriteLine(IMovable.MaxSpeed);   // 65  double time = IMovable.GetTime(500, 10);  Console.WriteLine(time);    // 50    interface IMovable  {      public const int minSpeed = 0;     // минимальная скорость      private static int maxSpeed = 60;   // максимальная скорость      // находим время, за которое надо пройти расстояние distance со скоростью speed      static double GetTime(double distance, double speed) => distance / speed;      static int MaxSpeed      {          get => maxSpeed;          set          {              if (value > 0) maxSpeed = value;          }      }  } |

Делегаты представляют такие объекты, которые указывают на методы. То есть делегаты - это указатели на методы и с помощью делегатов мы можем вызвать данные методы.

**Определение делегатов**

Для объявления делегата используется ключевое слово **delegate**, после которого идет возвращаемый тип, название и параметры. Например:

|  |  |
| --- | --- |
| 1 | delegate void Message(); |

Делегат Message в качестве возвращаемого типа имеет тип void (то есть ничего не возвращает) и не принимает никаких параметров. Это значит, что этот делегат может указывать на любой метод, который не принимает никаких параметров и ничего не возвращает.

Рассмотрим применение этого делегата:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Message mes;            // 2. Создаем переменную делегата  mes = Hello;            // 3. Присваиваем этой переменной адрес метода  mes();                  // 4. Вызываем метод    void Hello() => Console.WriteLine("Hello METANIT.COM");    delegate void Message(); // 1. Объявляем делегат |

Прежде всего сначала необходимо определить сам делегат:

|  |  |
| --- | --- |
| 1 | delegate void Message(); // 1. Объявляем делегат |

Для использования делегата объявляется переменная этого делегата:

|  |  |
| --- | --- |
| 1 | Message mes; // 2. Создаем переменную делегата |

Далее в делегат передается адрес определенного метода (в нашем случае метода Hello). Обратите внимание, что данный метод имеет тот же возвращаемый тип и тот же набор параметров (в данном случае отсутствие параметров), что и делегат.

|  |  |
| --- | --- |
| 1 | mes = Hello; // 3. Присваиваем этой переменной адрес метода |

Затем через делегат вызываем метод, на который ссылается данный делегат:

|  |  |
| --- | --- |
| 1 | mes(); // 4. Вызываем метод |

Вызов делегата производится подобно вызову метода.

При этом делегаты необязательно могут указывать только на методы, которые определены в том же классе, где определена переменная делегата. Это могут быть также методы из других классов и структур.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | Message message1 = Welcome.Print;  Message message2 = new Hello().Display;    message1(); // Welcome  message2(); // Привет    delegate void Message();    class Welcome  {      public static void Print() => Console.WriteLine("Welcome");  }  class Hello  {      public void Display() => Console.WriteLine("Привет");  } |

**Место определения делегата**

Если мы определяем делегат в прогаммах верхнего уровня (top-level program), которую по умолчанию представляет файл Program.cs начиная с версии C# 10, как в примере выше, то, как и другие типы, делегат определяется в конце кода. Но в принцие делегат можно определять внутри класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class Program  {      delegate void Message(); // 1. Объявляем делегат      static void Main()      {          Message mes;            // 2. Создаем переменную делегата          mes = Hello;            // 3. Присваиваем этой переменной адрес метода          mes();                  // 4. Вызываем метод            void Hello() => Console.WriteLine("Hello METANIT.COM");      }  } |

Либо вне класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | delegate void Message(); // 1. Объявляем делегат  class Program  {      static void Main()      {          Message mes;            // 2. Создаем переменную делегата          mes = Hello;            // 3. Присваиваем этой переменной адрес метода          mes();                  // 4. Вызываем метод            void Hello() => Console.WriteLine("Hello METANIT.COM");      }  } |

**Параметры и результат делегата**

Рассмотрим определение и применение делегата, который принимает параметры и возвращает результат:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | Operation operation = Add;      // делегат указывает на метод Add  int result = operation(4, 5);   // фактически Add(4, 5)  Console.WriteLine(result);      // 9    operation = Multiply;           // теперь делегат указывает на метод Multiply  result = operation(4, 5);       // фактически Multiply(4, 5)  Console.WriteLine(result);      // 20    int Add(int x, int y) => x + y;    int Multiply(int x, int y) => x \* y;    delegate int Operation(int x, int y); |

В данном случае делегат Operation возвращает значение типа int и имеет два параметра типа int. Поэтому этому делегату соответствует любой метод, который возвращает значение типа int и принимает два параметра типа int. В данном случае это методы Add и Multiply. То есть мы можем присвоить переменной делегата любой из этих методов и вызывать.

Поскольку делегат принимает два параметра типа int, то при его вызове необходимо передать значения для этих параметров: operation(4,5).

**Присвоение ссылки на метод**

Выше переменной делегата напрямую присваивался метод. Есть еще один способ - создание объекта делегата с помощью конструктора, в который передается нужный метод:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Operation operation1 = Add;  Operation operation2 = new Operation(Add);    int Add(int x, int y) => x + y;    delegate int Operation(int x, int y); |

Оба способа равноценны.

**Соответствие методов делегату**

Как было написано выше, методы соответствуют делегату, если они имеют один и тот же возвращаемый тип и один и тот же набор параметров. Но надо учитывать, что во внимание также принимаются модификаторы **ref**, **in** и **out**. Например, пусть у нас есть делегат:

|  |  |
| --- | --- |
| 1 | delegate void SomeDel(int a, double b); |

Этому делегату соответствует, например, следующий метод:

|  |  |
| --- | --- |
| 1 | void SomeMethod1(int g, double n) { } |

А следующие методы НЕ соответствуют:

|  |  |
| --- | --- |
| 1  2  3  4 | double SomeMethod2(int g, double n) { return g + n; }  void SomeMethod3(double n, int g) { }  void SomeMethod4(ref int g, double n) { }  void SomeMethod5(out int g, double n) { g = 6; } |

Здесь метод SomeMethod2 имеет другой возвращаемый тип, отличный от типа делегата. SomeMethod3 имеет другой набор параметров. Параметры SomeMethod4 и SomeMethod5 также отличаются от параметров делегата, поскольку имеют модификаторы ref и out.

**Добавление методов в делегат**

В примерах выше переменная делегата указывала на один метод. В реальности же делегат может указывать на множество методов, которые имеют ту же сигнатуру и возвращаемые тип. Все методы в делегате попадают в специальный список - список вызова или invocation list. И при вызове делегата все методы из этого списка последовательно вызываются. И мы можем добавлять в этот список не один, а несколько методов. Для добавления методов в делегат применяется операция **+=**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Message message = Hello;  message += HowAreYou;  // теперь message указывает на два метода  message();              // вызываются оба метода - Hello и HowAreYou    void Hello() => Console.WriteLine("Hello");  void HowAreYou() => Console.WriteLine("How are you?");    delegate void Message(); |

В данном случае в список вызова делегата message добавляются два метода - Hello и HowAreYou. И при вызове message вызываются сразу оба этих метода.

Однако стоит отметить, что в реальности будет происходить создание нового объекта делегата, который получит методы старой копии делегата и новый метод, и новый созданный объект делегата будет присвоен переменной message.

При добавлении делегатов следует учитывать, что мы можем добавить ссылку на один и тот же метод несколько раз, и в списке вызова делегата тогда будет несколько ссылок на один и то же метод. Соответственно при вызове делегата добавленный метод будет вызываться столько раз, сколько он был добавлен:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Message message = Hello;  message += HowAreYou;  message += Hello;  message += Hello;    message(); |

Консольный вывод:

Hello

How are you?

Hello

Hello

Подобным образом мы можем удалять методы из делегата с помощью операций **-=**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Message? message = Hello;  message += HowAreYou;  message();  // вызываются все методы из message  message -= HowAreYou;   // удаляем метод HowAreYou  if (message != null) message(); // вызывается метод Hello |

При удалении методов из делегата фактически будет создаваться новый делегат, который в списке вызова методов будет содержать на один метод меньше.

Стоит отметить, что при удалении метода может сложиться ситуация, что в делегате не будет методов, и тогда переменная будет иметь значение null. Поэтому в данном случае переменная определена не просто как переменная типа Message, а именно **Message?**, то есть типа, который может представлять как делегат Message, так и значение null.

Кроме того, перед вторым вызовом мы проверяем переменную на значение null.

При удалении следует учитывать, что если делегат содержит несколько ссылок на один и тот же метод, то операция -= начинает поиск с конца списка вызова делегата и удаляет только первое найденное вхождение. Если подобного метода в списке вызова делегата нет, то операция -= не имеет никакого эффекта.

**Объединение делегатов**

Делегаты можно объединять в другие делегаты. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Message mes1 = Hello;  Message mes2 = HowAreYou;  Message mes3 = mes1 + mes2; // объединяем делегаты  mes3(); // вызываются все методы из mes1 и mes2    void Hello() => Console.WriteLine("Hello");  void HowAreYou() => Console.WriteLine("How are you?");    delegate void Message(); |

В данном случае объект mes3 представляет объединение делегатов mes1 и mes2. Объединение делегатов значит, что в список вызова делегата mes3 попадут все методы из делегатов mes1 и mes2. И при вызове делегата mes3 все эти методы одновременно будут вызваны.

**Вызов делегата**

В примерах выше делегат вызывался как обычный метод. Если делегат принимал параметры, то при его вызове для параметров передавались необходимые значения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Message mes = Hello;  mes();  Operation op = Add;  int n = op(3, 4);  Console.WriteLine(n);    void Hello() => Console.WriteLine("Hello");  int Add(int x, int y) => x + y;    delegate int Operation(int x, int y);  delegate void Message(); |

Другой способ вызова делегата представляет метод **Invoke()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Message mes = Hello;  mes.Invoke(); // Hello  Operation op = Add;  int n = op.Invoke(3, 4);  Console.WriteLine(n);   // 7    void Hello() => Console.WriteLine("Hello");  int Add(int x, int y) => x + y;    delegate int Operation(int x, int y);  delegate void Message(); |

Если делегат принимает параметры, то в метод Invoke передаются значения для этих параметров.

Следует учитывать, что если делегат пуст, то есть в его списке вызова нет ссылок ни на один из методов (то есть делегат равен Null), то при вызове такого делегата мы получим исключение, как, например, в следующем случае:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Message? mes;  //mes();        // ! Ошибка: делегат равен null    Operation? op = Add;  op -= Add;      // делегат op пуст  int n = op(3, 4);       // !Ошибка: делегат равен null |

Поэтому при вызове делегата всегда лучше проверять, не равен ли он null. Либо можно использовать метод Invoke и оператор условного null:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Message? mes = null;  mes?.Invoke();        // ошибки нет, делегат просто не вызывается    Operation? op = Add;  op -= Add;          // делегат op пуст  int? n = op?.Invoke(3, 4);   // ошибки нет, делегат просто не вызывается, а n = null |

Если делегат возвращает некоторое значение, то возвращается значение последнего метода из списка вызова (если в списке вызова несколько методов). Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Operation op = Subtract;  op += Multiply;  op += Add;  Console.WriteLine(op(7, 2));    // Add(7,2) = 9    int Add(int x, int y) => x + y;  int Subtract(int x, int y) => x - y;  int Multiply(int x, int y) => x \* y;    delegate int Operation(int x, int y); |

**Обобщенные делегаты**

Делегаты, как и другие типы, могут быть обобщенными, например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | Operation<decimal, int> squareOperation = Square;  decimal result1 = squareOperation(5);  Console.WriteLine(result1);  // 25    Operation<int, int> doubleOperation = Double;  int result2 = doubleOperation(5);  Console.WriteLine(result2);  // 10    decimal Square(int n) => n \* n;  int Double(int n) => n + n;    delegate T Operation<T, K>(K val); |

Здесь делегат Operation типизируется двумя параметрами типов. Параметр T представляет тип возвращаемого значения. А параметр K представляет тип передаваемого в делегат параметра. Таким образом, этому делегату соответствует метод, который принимает параметр любого типа и возвращает значение любого типа.

В прогамме мы можем определить переменные делегата под определенный метод. Например, делегату Operation<decimal, int> соответствует метод, который принимает число int и возвращает число типа decimal. А делегату Operation<int, int> соответствует метод, который принимает и возвращает число типа int.

**Делегаты как параметры методов**

Также делегаты могут быть параметрами методов. Благодаря этому один метод в качестве параметров может получать действия - другие методы. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | DoOperation(5, 4, Add);         // 9  DoOperation(5, 4, Subtract);    // 1  DoOperation(5, 4, Multiply);    // 20    void DoOperation(int a, int b, Operation op)  {      Console.WriteLine(op(a,b));  }  int Add(int x, int y) => x + y;  int Subtract(int x, int y) => x - y;  int Multiply(int x, int y) => x \* y;    delegate int Operation(int x, int y); |

Здесь метод **DoOperation** в качестве параметров принимает два числа и некоторое действие в виде делегата Operation. В внутри метода вызываем делегат Operation, передавая ему числа из первых двух параметров.

При вызове метода DoOperation мы можем передать в него в качестве третьего параметра метод, который соответствует делегату Operation.

**Возвращение делегатов из метода**

Также делегаты можно возвращать из методов. То есть мы можем возвращать из метода какое-то действие в виде другого метода. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | Operation operation = SelectOperation(OperationType.Add);  Console.WriteLine(operation(10, 4));    // 14    operation = SelectOperation(OperationType.Subtract);  Console.WriteLine(operation(10, 4));    // 6    operation = SelectOperation(OperationType.Multiply);  Console.WriteLine(operation(10, 4));    // 40    Operation SelectOperation(OperationType opType)  {      switch (opType)      {          case OperationType.Add: return Add;          case OperationType.Subtract: return Subtract;          default: return Multiply;      }  }    int Add(int x, int y) => x + y;  int Subtract(int x, int y) => x - y;  int Multiply(int x, int y) => x \* y;    enum OperationType  {      Add, Subtract, Multiply  }  delegate int Operation(int x, int y); |

В данном случае метод SelectOperation() в качестве параметра принимает перечисление типа OperationType. Это перечисление хранит три константы, каждая из которых соответствует определенной арифметической операции. И в самом методе в зависимости от значения параметра возвращаем определенный метод. Причем поскольку возвращаемый тип метода - делегат Operation, то метод должен возвратить метод, который соответствует этому делегату - в нашем случае это методы Add, Subtract, Multiply. То есть если параметр метода SelectOperation равен OperationType.Add, то возвращается метод Add, который выполняет сложение двух чисел:

|  |  |
| --- | --- |
| 1 | case OperationType.Add: return Add; |

При вызове метода SelectOperation мы можем получить из него нужное действие в переменную operation:

|  |  |
| --- | --- |
| 1 | Operation operation = SelectOperation(OperationType.Add); |

И при вызове переменной operation фактически будет вызываться полученный из SelectOperation метод:

|  |  |
| --- | --- |
| 1  2 | Operation operation = SelectOperation(OperationType.Add);   // Здесь operation = Add  Console.WriteLine(operation(10, 4));    // 14 |

**События** сигнализируют системе о том, что произошло определенное действие. И если нам надо отследить эти действия, то как раз мы можем применять события.

События объявляются в классе с помощью ключевого слова **event**, после которого указывается тип делегата, который представляет событие:

|  |  |
| --- | --- |
| 1  2 | delegate void AccountHandler(string message);  event AccountHandler Notify; |

В данном случае вначале определяется делегат AccountHandler, который принимает один параметр типа string. Затем с помощью ключевого слова **event** определяется событие с именем Notify, которое представляет делегат AccountHandler. Название для события может быть произвольным, но в любом случае оно должно представлять некоторый делегат.

Определив событие, мы можем его вызвать в программе как метод, используя имя события:

|  |  |
| --- | --- |
| 1 | Notify("Произошло действие"); |

Поскольку событие Notify представляет делегат AccountHandler, который принимает один параметр типа string - строку, то при вызове события нам надо передать в него строку.

Однако при вызове событий мы можем столкнуться с тем, что событие равно null в случае, если для его не определен обработчик. Поэтому при вызове события лучше его всегда проверять на null. Например, так:

|  |  |
| --- | --- |
| 1 | if(Notify !=null) Notify("Произошло действие"); |

Или так:

|  |  |
| --- | --- |
| 1 | Notify?.Invoke("Произошло действие"); |

В этом случае поскольку событие представляет делегат, то мы можем его вызвать с помощью метода **Invoke()**, передав в него необходимые значения для параметров.

С событием может быть связан один или несколько обработчиков. Обработчики событий - это именно то, что выполняется при вызове событий. Нередко в качестве обработчиков событий применяются методы. Каждый обработчик событий по списку параметров и возвращаемому типу должен соответствовать делегату, который представляет событие. Для добавления обработчика события применяется операция **+=**:

|  |  |
| --- | --- |
| 1 | Notify += обработчик события; |

Определим обработчики для события Notify, чтобы получить в программе нужные уведомления:

Для одного события можно установить несколько обработчиков и потом в любой момент времени их удалить. Для удаления обработчиков применяется операция **-=**.

С помощью специальных акссесоров **add/remove** мы можем управлять добавлением и удалением обработчиков. Как правило, подобная функциональность редко требуется, но тем не менее мы ее можем использовать.

## C#. Классы - прототипы, параметризованные коллекции.

Многие алгоритмы *не зависят от типов данных*, с которыми они работают. Например, такие алгоритмы как сортировка и поиск. Возможность отделить алгоритмы от типов данных предоставляют классы-прототипы, которые в литературе встречаются под названиями “*шаблоны классов”, ” параметризованные классы”, ” обобщённые классы”.*

Классы прототипы имеют параметр – тип данных и поэтому еще называются параметризованными классами. Чаще всего эти классы применяются для хранения данных, то есть в качестве *контейнерных классов*или*коллекций*.

Существуют “стандартные” классы прототипы, которые называются **параметризованными коллекциями**. Такие коллекции – это уже готовые шаблоны для различных структур данных – стеков, очередей, списков, бинарных деревьев и т.д. Эти коллекции хранятся в пространстве имён*System.Collections.Generic*.

Во всех параметризованных коллекциях имеется так называемый параметр в качестве которого обычно выступает тип данных с которым работает эта коллекция.

В качестве примера рассмотрим параметризованную коллекцию **List <T>,**которая является двойником обычной коллекции (семейства) ArrayList.

**Пример применения параметризованной коллекции List <T>**

using System.Collections.Generic;

using MonsterLib; //описание классов Monster и Daemon смотри в предыдущих //лекциях – данные классы включены в библиотеку MonsterLib

…

static void Main(string[] args)

{

//объявление экземпляра коллекции List с параметром – типом данных – классом Monster

List<Monster> mas = new List<Monster>();

//добавление в коллекцию объектов класса Moster и производного от него Daemon

mas.Add(new Monster ("Вася"));

mas.Add(new Daemon("Демон", 3));

//вывод всех элементов в коллекции

foreach (Monster x in mas) x.passport();

//объявление экземпляра коллекции List с параметром – типом данных – int

List<int> masl = new List<int>();

//добавление в коллекцию целых чисел

masl.Add(5);

masl.Add(1);

//применение метода Sort() коллекции List для целых чисел

masl.Sort();

//к элементам коллекции можно обратиться **по индексу!**

int a = masl[0];

Console.WriteLine(a);

//вывод всех элементов в коллекции

foreach(int x in masl) Console.WriteLine( x+ ” ”);

}

В предложенном примере реализовано две коллекции на основе параметризованной коллекции List<Т>. Первая коллекцияmasсодержит список (аналог одномерного массива) экземпляров (объектов) классов типаMonster. В данной коллекцииmasможно хранить элементы классаMonster, а также любого класса, производного отMonster. Другие типы данных в коллекцииmasнедопустимы.

Существуют параметризованные коллекции с несколькими параметрами, как например, коллекция Dictionary<T,K> - так называемый словарь. У данной коллекции два параметра – тип ключей и тип значений, хранимых в словаре.

В следующем примере демонстрируется применение параметризованной коллекции Dictionary<T,K>.

**Пример применения параметризованной коллекции Dictionary <T,K>**

В примере программы считывается содержимое текстового файла, считанный текст разбивается на слова и подсчитывается количество повторений каждого слова в тексте.

В качестве ключей в примере используются слова, считанные из файла, а значения представляют собой целочисленные счетчики, которые увеличиваются на единицу, когда слово встречается очередной раз.

using System.Collection.Generic;

using System.IO;

…

static void Main(string[] args)

{

//файл должен существовать!!!

StreamReader f = new StreamReader(@"d:\text.txt");

//считывание содержимого файла в одну строку

string s = f.ReadToEnd();

//массив всех символов-разделителей слов

char[] razdeliteli = { '.', ' ', ',', '!' };

//s.Split(raxdeliteli) – разделение строки на слова, используя разделители. Результат - массив строк(слов) slova

List<string> slova = new List<string>(s.Split(razdeliteli));

//mas – список с параметрами – слово и количество его вхождений, причем слова-это ключи, значения-счетчики вхождений

Dictionary<string, int> mas = new Dictionary<string, int>();

foreach (string w in slova)

{

//если в массиве по ключам(словам) слово встретилось впервые в //значение, соответствующее этому слову заносится 1, если слово //уже встречалось значение увеличивается на единицу

//mas[w]-количество повторений слова w

if (mas.ContainsKey(w)) mas[w]++; else mas[w] = 1;

}

//вывод всех слов в массиве mas по ключам

foreach (string w in mas.Keys)

Console.WriteLine("{0}\t{1}", w, mas[w]);

Console.ReadKey();

}

Язык С# позволяет создавать собственные классы-прототипы и их разновидности – интерфейсы, структуры, делегаты, события.

Рассмотрим создание класса-прототипа на примере стека. Параметр типа данных, которые охраняться в стеке указывается угловых скобках после имени класса, а затем используется таким же образом, как и обычные типы.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.IO;

namespace ConsoleApplication3

{

class Program

{

public class Stack<T> //шаблонный класс, собственный класс-прототип

{

int count; //количество элементов в стеке

T[] items; //сами элементы стека

int top; //вершина стека

public int Count //свойство “количество”

{

get { return count; }

set { if ((top > 0)&&(top <=count)) count = value; else Console.WriteLine("Error!!!"); }

}

public void InitStack() //первоначальное создание стека

{

Console.Write("Введите количество элементов стека: ");

count=Convert.ToInt32(Console.ReadLine());

items = new T[count]; //выделяем память для массива элементов стека

top = count; //первоначально вершина стека – в конце

}

public bool EndOfStack()//конец стека?

{

if (top == count) return true;

else return false;

}

public void Push(T item)//помещение элемента в стек

{

if (top == 0) { Console.WriteLine("Error"); return; }

else

{

items[top-1] = item;

top = top - 1;

}

}

public T Pop() //извлечение элемента из стека

{

if (top == count) { Console.WriteLine("Error"); } else top = top + 1; return items[top-1];}

}

static void Main(string[] args)

{

//При использовании класса-прототипа вместо параметра т подставляется //конкретный тип данных

//стек целых чисел

Stack<int> ints = new Stack<int>();

ints.InitStack();

for (int i = 0; i < ints.Count;i++ )

ints.Push(i);

while(!ints.EndOfStack())

{

int x = ints.Pop();

Console.WriteLine(x + " ");

}

//стек дробных чисел

Stack<double> double\_s = new Stack<double>();

double\_s.InitStack();

for (double i = 0; i < double\_s.Count; i++)

double\_s.Push(i);

while (!double\_s.EndOfStack())

{

double y = double\_s.Pop();

Console.WriteLine("{0:f}",y);

}

Console.ReadKey(); } }}

**Обобщенные(параметризованные) методы, ограничения на использование параметризованных типов**

Ограничения

Класс-прототип может содержать произвольное число параметров типа. Для каждого параметра можно задать ограничения, указывающие каким требованиям должен удовлетворять аргумент, соответствующий этому параметру. Например, можно указать, что это юудет тип, использующий некоторый интерфейс (см.пример программы, приведенной ниже).

Ограничения задаются после ключевого слова where, например:

**public class Stack<T>**

**where T:struct {…}**

Здесь задано ограничение, что класс стек может использовать элементы только значимого типа (типа структуры). Для ссылочного типанеобходимо использовать ключевое слово class.

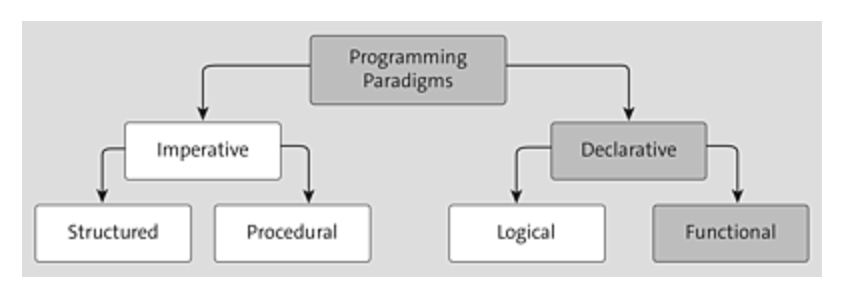
Указание в качетстве ограничений имени класса означает, что соответствующий параметр (аргумент) должен быть инициализирован или именем этого класса, либо его потомка.

# Функциональное и логическое программирование

## Функциональное программирование.

Функциональное программирование представляет одну из парадигм программирования, которая базируется на следующих принципах:

* Функции представляют полноценные объекты ("first-class objects", "first-class citizens"). Это означает, что переменным можно присваивать также, как и другие объекты и примитивные значения; функции можно использовать в качестве аргументов других функций или в качестве их возвращаемого значения.
* Функции работают с неизменяемыми структурами данных. Структуры данных в функциональном программировании обычно неизменяемы или не изменяются. Операции же, которые выполняются над структурами данных, при необходимости создают новые структуры данных и возвращают их в качестве результатов. Например, в чисто функциональных языках программирования после создания списков или других структур данных их нельзя изменить
* Функции не имеют побочных эффектов. В функциональном программировании функции обычно вообще не имеют побочных эффектов и ведут себя скорее как математические функции. Это означает, что функции в функциональном программировании всегда возвращают один и тот же результат для одних и тех же входных данных и не вызывают никаких побочных эффектов. В чисто функциональных языках побочные эффекты уже предотвращаются самим языком.
* Функциональные программы применяют декларативный стиль программирования. Императивное программирование — это парадигма программирования, в которой компьютеру даются очень точные индивидуальные инструкции о том, **как** решить проблему. В императивных программах используются явные операторы цикла (циклы while, for и т. д.), условные операторы (if-else) и их последовательности. Функциональные же программы исповедуют декларативный стиль, что означает, что программа больше говорит, **что** следует делать, а не **как**. В результате функциональные программы обычно более читабельны, более содержательны и более компактны, чем эквивалентный императивный код.



Различные языки могут не быть чисто функциональными, но тем не менее могут поддерживать функциональную парадигму. К примеру, возьмем JavaScript, который не является чистым функциональным языком, однако позволяет писать в функциональном стиле. Рассмотрим следующую программу перебора массива на JavaScript:

## Логическое программирование.

Логическое программирование — это технология, позволяющая довольно близко приблизиться к воплощению декларативного идеала, описанного в главе 7, согласно которому системы должны конструироваться путем представления знаний на некотором формальном языке, а задачи решаться путем применения процессов логического вывода к этим знаниям. Такой идеал выражен в следующем уравнении Роберта Ковальского:

Алгоритм = Логика + Управление

Одним из языков логического программирования, намного превосходящим все прочие по своей распространенности, является Prolog. Количество его пользователей насчитывает сотни тысяч. Он используется в основном в качестве языка быстрой разработки прототипов, а также служит для решения задач символических манипуляций, таких как написание компиляторов [1536] и синтаксический анализ текстов на естественном языке [1208]. На языке Prolog было написано много экспертных систем для юридических, медицинских, финансовых и других проблемных областей.

Программы Prolog представляют собой множества определенных выражений, записанных в системе обозначений, немного отличающейся от используемой стандартной логики первого порядка. В языке Prolog прописные буквы применяются для обозначения переменных, а строчные — для обозначения констант. Выражения записываются с головой, предшествующей телу; символ : - служит для обозначения импликации, направленной влево, запятые разделяют литералы в теле, а точка обозначает конец высказывания, как показано ниже.

http://rriai.org.ru/illustr/ai2-543.jpg

Язык Prolog включает "синтаксические упрощения" (syntactic sugar) для обозначения списков и арифметических выражений. Например, ниже приведена программа Prolog для предиката append (X, Υ, Ζ), которая выполняется успешно, если список Ζ представляет собой результат дополнения списка Υ списком х.

http://rriai.org.ru/illustr/ai2-544.jpg

На естественном языке эти выражения можно прочитать так: во-первых, дополнение списка Υ пустым списком приводит к получению того же списка Υ, и, во-вторых, [ А | Z ] — это результат дополнения списка Υ списком [ А | X], при условии, что Z— это результат дополнения списка Υ списком X. Такое определение предиката append на первый взгляд кажется весьма подобным соответствующему определению на языке Lisp, но фактически является гораздо более мощным. Например, в систему можно ввести запрос append (А, В, [1,2] ) — какие два списка можно дополнить один другим, чтобы получить [1,2]? Система возвратит следующие решения:

http://rriai.org.ru/illustr/ai2-545.jpg

Выполнение программ Prolog осуществляется по принципу обратного логического вывода с поиском в глубину, при котором попытка применения выражений выполняется в том порядке, в каком они записаны в базу знаний. Но некоторые описанные ниже особенности языка Prolog выходят за рамки стандартного логического вывода.

•    В нем предусмотрено множество встроенных функций для выполнения арифметических операций. Литералы, в которых используются соответствующие функциональные символы, "доказываются" путем выполнения кода, а не осуществления дальнейшего логического вывода. Например, цель "X is 4+3" достигается успешно после связывания переменной X со значением 7. С другой стороны, попытка достижения цели "5 is X+Y" оканчивается неудачей, поскольку эти встроенные функции не обеспечивают решения произвольных уравнений.

•    В языке предусмотрены встроенные предикаты, вызывающие при их выполнении побочные эффекты. К ним относятся предикаты ввода-вывода и предикаты assert/retract для модификации базы знаний. Такие предикаты не имеют аналогов в логике и могут порождать некоторые эффекты, вызывающие путаницу, например, если факты подтверждаются (и вводятся в базу знаний) некоторой ветвью дерева доказательства, которая в конечном итоге оканчивается неудачей.

 В языке Prolog допускается определенная форма отрицания —отрицание как недостижение цели. Отрицаемая цель not P считается доказанной, если системе не удается доказать Р. Таким образом, следующее высказывание:

http://rriai.org.ru/illustr/ai2-546.jpg можно прочитать так: "Любого следует считать живым, если нельзя доказать, что он мертв".

•    В языке Prolog предусмотрен оператор равенства, "=", но он не обладает всей мощью логического равенства. Цель с оператором равенства достигается успешно, если в ней два терма являются унифицируемыми, а в противном случае попытка ее достижения оканчивается неудачей. Таким образом, цель x+Y=2+3 достигается успешно, после связывания переменной X со значением 2, a Y — со значением 3, а попытка достижения цели morningstar=evenings tar оканчивается неудачей. (В классической логике последнее равенство может быть или не быть истинным.) Не могут быть подтверждены (введены в базу знаний) какие-либо факты или правила, касающиеся равенства.

•    Из алгоритма унификации Prolog исключена проверка вхождения. Это означает, что могут быть сделаны некоторые противоречивые логические выводы; такая проблема возникает редко, за исключением тех ситуаций, когда язык Prolog используется для доказательства математических теорем.

Решения, принятые при проектировании языка Prolog, представляют собой компромисс между стремлениями обеспечить декларативность и вычислительную эффективность (по крайней мере, эффективность в той ее трактовке, которая существовала в период разработки языка Prolog). Мы вернемся к этой теме после рассмотрения того, как реализована система Prolog.

# Системы искусственного интеллекта

## Классификация и представление знаний. Модели представления знаний.

## Понятия фазификации, дефазификации и лингвистической переменной. Операции с нечёткими множествами. Обобщённая процедура нечёткого логического вывода.

## Алгоритмы нечёткого логического вывода. Методы дефазификации.

## Основные понятия и определения интеллектуального анализа данных. Средства построения интеллектуальных систем поддержки принятия решений.

## Аналитические и обучаемые модели. Алгоритмы машинного обучения.

## Теория игр в искусственном интеллекте.

## Классификатор «2А» систем искусственного интеллекта. Тест Тьюринга.

## Методы интеллектуального анализа данных. Нейронные сети. Эволюционные алгоритмы.

## Архитектура системы поддержки принятия решения. OLTP и OLAP системы.

### **Литература**

1. Симонович С.В. Информатика. Базовый курс.
2. Альфред В. Ахо и др. Структуры данных и алгоритмы.
3. Кормен Т. и др. Алгоритмы: построение и анализ.
4. Финогенов К.Г. Основы языка Ассемблера.
5. Юров В. и др. ASSEMBLER.Учебный курс.
6. Гордеев А. В. и др. Системное программное обеспечение.
7. Олифер В.Г., Олифер Н.А. Сетевые операционные системы.
8. Олифер В.Г., Олифер Н.А. Компьютерные сети.
9. Сергиенко А.Б. Цифровая обработка сигналов.
10. Пол А. ООП на С++.
11. Мамаев М. и др. Технология защиты информации в Интернете. Специальный справочник.
12. Гмурман . Теория вероятности и математическая статистика.
13. Турчак Л.И. Основы численных методов.
14. Красс М.С. и др. Основы математики и её приложения в экономическом образовании.
15. Круглов В.В., Дли М.И., Голубов Р.Ю. Нечёткая логика
16. и искусственные нейронные сети. 2001.
17. Гаврилова Т.А., Хорошевский В.Ф. Базы знаний интеллектуальных систем. 2001.
18. Круглов В.В., Борисов В.В. Искусственные нейронные сети: теория и практика. 2002.
19. Чулюков В.А., Астахова И.Ф., Потапов А.С. Системы искусственного интеллекта. Практический курс. 2008.
20. Цилькер Б.Я., Орлов С.А. Организация ЭВМ и систем: Учебник для вузов. – СПб.: Питер, 2011. – 688 с.
21. Мюллер С. Модернизация и ремонт ПК, 19-е издание.: Пер. с англ. – М.: Издательский дом «Вильямс», 2011. – 1072 с.