Конспекты лекций по дисциплине «Программирование и основы алгоритмизации» (основы программирования на C#)

Автор: Хабибулина Н.Ю.

Лекция 1 2
ЖИЗНЕННЫЙ ЦИКЛ ПО. ПРОЕКТИРОВАНИЕ ПО. АЛГОРИТМ 2
Лекция 2 16
ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C# 16
Лекция 3 22
ФУНКЦИИ (МЕТОДЫ) В C# 22
Лекция 4 32
СТРУКТУРИРОВАННЫЕ ТИПЫ ДАННЫХ 32
МАССИВЫ 32
СТРОКИ И РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ 39
Лекция 5 58
СТРУКТУРЫ 58
Лекция 6 66
РАБОТА С ФАЙЛАМИ 66

Лекция 1
ЖИЗНЕННЫЙ ЦИКЛ ПО. ПРОЕКТИРОВАНИЕ ПО. АЛГОРИТМ

Жизненный цикл программного продукта

Жизненный цикл программного продукта – это период времени, который начинается с момента принятия решения о необходимости создания программного продукта и заканчивается в момент его полного изъятия из эксплуатации.

Программы любого вида характеризуются **жизненным циклом**, состоящим из отдельных этапов:

a) маркетинг рынка программных средств, спецификация требований к программному продукту;

b) проектирование структуры программного продукта;

c) программирование (создание программного кода), тестирование, автономная и комплексная отладка программ;

d) документирование программного продукта, подготовка эксплуатационной и технологической документации;

e) выход на рынок программных средств, распространение программного продукта;

f) эксплуатация программного продукта пользователями;

g) сопровождение программного продукта;

h) снятие программного продукта с продажи, отказ от сопровождения.

На рисунке изображены этапы жизненного цикла и показаны их временное соответствие друг другу. Рассмотрим содержание отдельных этапов жизненного цикла.

Маркетинг и спецификация программного продукта предназначены для изучения требований к создаваемому программному продукту, а именно:

– определение состава и назначения функций обработки данных программного продукта;

– установление требований пользователя к характеру взаимодействия с программным продуктом, типу пользовательского интерфейса (система меню, использование манипулятора мышь, типы подсказок, виды экранных документов и т.п.);

– требования к комплексу технических и программных средств для эксплуатации программного продукта и т.д.

На данном этапе необходимо выполнить формализованную постановку задачи.

Если программный продукт создается не под заказ и предполагается выход на рынок программных средств, маркетинг выполняется в полном объеме: изучаются программные продукты-конкуренты и аналоги, обобщаются требования пользователей к программному продукту, устанавливается потенциальная емкость рынка сбыта, дается прогноз цены и объема продаж. Кроме того, важно оценить необходимые для разработки программного

продукта материальные, трудовые и финансовые ресурсы, ориентировочные длительности основных этапов жизненного цикла программного продукта.

Если программный продукт создается как заказное программное изделие для определенного заказчика, на данном этапе также важно правильно сформулировать и документировать задание на его разработку. Ошибочно понятое требование к программному продукту может привести к нежелательным результатам в процессе его эксплуатации.

Проектирование структуры программного продукта связано с алгоритмизацией процесса обработки данных, детализацией функций обработки, разработкой структуры программного продукта (архитектуры программных модулей), структуры информационной базы (базы данных) задачи, выбором методов и средств создания программ-технологии программирования.

Программирование, тестирование и отладка программ являются технической реализацией проектных решений и выполняются с помощью выбранного инструментария разработчика (алгоритмические языки и системы программирования, инструментальные среды разработчиков и т.п.).

Для больших и сложных программных комплексов, имеющих развитую модульную структуру построения, отдельные работы данного этапа могут выполняться параллельно, обеспечивая сокращение общего времени разработки программного продукта. Важная роль принадлежит используемым при этом инструментальным средствам программирования и отладки программ, поскольку они влияют на трудоемкость выполнения работ, их стоимость, качество создаваемых программ.

Документирование программного продукта является обязательным видом работ, выполняемых, как правило, не самим разработчиком, а лицом, связанным с распространением и внедрением программного продукта. Документация должна содержать необходимые сведения по установке и обеспечению надежной работы программного продукта, поддерживать пользователей при выполнении функций обработки, определять порядок комплексирования программного продукта с другими программами. Успех распространения и эксплуатации программного продукта в значительной степени зависит от качества его документации.

На машинном уровне программного продукта, как правило, создаются:

– автоматизированная контекстно-зависимая помощь (HELP);

– демонстрационные версии, работающие в активном режиме по типу обучающих систем (электронный учебник) или пассивном режиме (ролик, мультфильм) - для демонстрации функциональных возможностей программного продукта и информационной технологии его использования.

Выход программного продукта на рынок программных средств связан с организацией продаж массовому пользователю. Этот этап должен по возможности быть коротким, для продвижения программных продуктов применяются стандартные приемы маркетинга: реклама, увеличение числа каналов реализации, создание дилерской и дистрибьютерной сети, ценовая политика - продажа со скидками, сервисное обслуживание и др.

Требуется постоянная программа маркетинговых мероприятий и поддержки программных продуктов. Как правило, для каждого программного продукта существует своя форма кривой продаж, которая отражает спрос.

Вначале продажа программного продукта идет вверх - возрастающий участок кривой. Затем наступает стабилизация продаж программного продукта. Фирма-разработчик стремится к максимальной длительности периода стабильных продаж на высоком уровне. Далее происходит падение объема продаж, что является сигналом к изменению маркетинговой политики фирмы в отношении данного программного продукта, требуется модификация данного продукта, изменение цены или снятие с продажи.

Эксплуатация программного продукта идет параллельно с его сопровождением, при этом эксплуатация программ может начинаться и в случае отсутствия сопровождения или продолжаться в случае завершения сопровождения еще какое-то время. После снятия программного продукта с продажи определенное время также может выполняться его

сопровождение. В процессе эксплуатации программного продукта производится устранение обнаруженных ошибок.

Снятие программного продукта с продажи и отказ от сопровождения происходят, как правило, в случае изменения технической политики фирмы-разработчика, неэффективности работы программного продукта, наличия в нем неустранимых ошибок, отсутствия спроса.

Длительность жизненного цикла для различных программных продуктов неодинакова. Для большинства современных программных продуктов длительность жизненного цикла измеряется в годах (2-3 года). Хотя достаточно часто встречаются на компьютерах и давно снятые с производства программные продукты.

Особенность разработки программного продукта заключается в том, что на начальных этапах принимаются решения, реализуемые на последующих этапах. Допущенные ошибки, например при спецификации требований к программному продукту, приводят к огромным потерям на последующих этапах разработки или эксплуатации программного продукта и даже к неуспеху всего проекта. Так, при необходимости внесения изменений в спецификацию программного продукта следует повторить в полном объеме все последующие этапы проектирования и создания программного продукта.

1. Модели жизненного цикла

Методология проектирования программных систем описывает процесс создания и сопровождения программного обеспечения в виде жизненного цикла (ЖЦ) разработки, представляя его как некоторую последовательность стадий и выполняемых на них процессов.

Для каждого этапа определяются состав и последовательность выполняемых работ, критерии начала и завершения этапа, получаемые результаты, методы и средства, необходимые для выполнения работ, роли и ответственность участников и т.д. Такое формальное описание ЖЦ разработки позволяет спланировать и организовать процесс коллективной работы, а также обеспечить управление этим процессом.

Понятия жизненного цикла и модели жизненного цикла несколько отличаются, однако термин "жизненный цикл" часто используется в смысле "модели жизненного цикла". Модель жизненного цикла разработки ПО - обобщенная структура, содержащая процессы, действия и задачи, которые осуществляются в ходе разработки, функционирования и сопровождения типового программного продукта в течение всей жизни системы, т.е. от определения требований до завершения ее использования.

На данный момент можно выделить следующие часто упоминаемые и используемые модели жизненного цикла:

1. каскадная модель;
2. процессная или поэтапная модель с промежуточным контролем;
3. итерационная модель.

2.1 . Каскадная модель

Каскадная модель (или, как ее еще называют, водопадная) (рис.2) рассматривает последовательное выполнение всех этапов проекта в строго фиксированном порядке. Эта модель происходит от структуры диаграммы Ганта для поэтапного процесса. Переход на следующий этап означает полное завершение работ на предыдущем этапе. Модель подчеркивает, что результаты, полученные в ходе выполнения одного этапа, используются для выполнения следующего этапа.

Рис. 2 Каскадная (водопадная) модель ЖЦ разработки

Каскадная схема включает несколько важных операций, применимых ко всем проектам:

- составление плана действий по разработке системы;
- планирование работ, связанных с каждым действием;
- применение контрольных этапов отслеживания хода выполнения действий.

При разработке относительно простых программных систем каждое приложение представляло собой единый, функционально и информационно независимый блок. Для разработки такого типа приложений эффективным оказался каскадный способ. Каждый этап завершался после полного выполнения и документального оформления всех предусмотренных работ.

В результате можно выделить следующие положительные стороны применения каскадного подхода:

- результат каждого этапа - законченный документ, отвечающий критериям полноты и непротиворечивости;
- заранее заданная последовательность этапов упрощает задачу планирования и позволяет вести контроль сроков завершения каждого этапа.

Каскадный подход хорошо зарекомендовал себя при построении простых систем, когда в самом начале разработки можно достаточно точно и полно сформулировать все требования к системе. Основным недостатком этого подхода является то, что, как показывает практика, реальный процесс создания сложной системы никогда полностью не укладывается в такую жесткую схему из-за большой динамики корректировок и уточнений, которые могут поступать как в результате дальнейшего развития проекта, так и извне в качестве новых требований и уточнений.

Порой для улучшения характеристик каскадной модели в нее включают возможность возвратов на ранее выполненные этапы (рис.3). Межэтапные корректировки позволяют учитывать реально существующее взаимовлияние результатов разработки на различных стадиях разработки. Даже на уровне модели видно, что самыми трудоемкими являются возвраты на уровень уточнения исходных требований.

Одна из проблем такой модели ЖЦ разработки заключается в отсутствии возможности оценить конечный результат до завершения всего процесса разработки ПО. Как показывает практика, будущие пользователи охотнее вносят изменения и уточнения в ходе оценки какого-либо прототипа системы (которого в данной модели ЖЦ не строится).

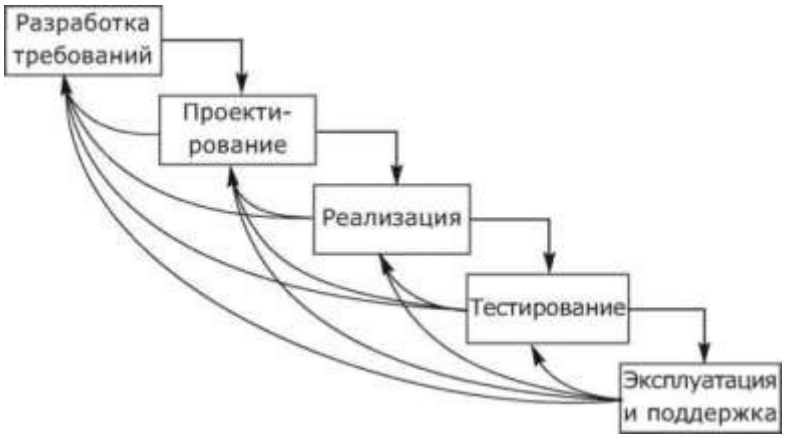


Рис. 3. Поэтапная модель с возвратами

Одним интересным вариантом развития поэтапной модели является V-образный жизненный цикл (рис.4). В этой модели акцент делается на работы, связанные с верификацией документов разработки. Нисходящая ветвь модели описывает собственно разработку программной документации: требования к ПО, функции программных элементов, архитектуру - связи программных функций, программный код. Восходящая ветвь - этапы верификации.

Применение V-образной модели жизненного цикла позволяет сконцентрировать внимание на проверке результатов разработки, точно спланировать, какие свойства программного обеспечения будут исследоваться, на каких этапах и на основании какой документации. Обычно обращение к данному виду модели связано с повышенными требованиями к качеству результатов разработки. Именно такой подход к выбору модели жизненного цикла разработки характерен для бортовых авиационных систем, систем управления космическими аппаратами, разработки встроенного ПО.

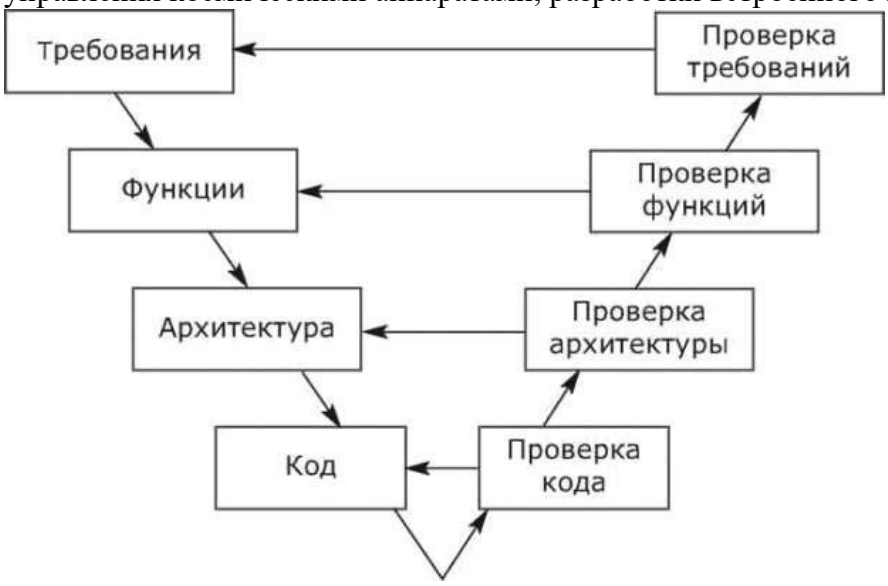


Рис. 4. V-образная модель жизненного цикла

2.2 Спиральная модель

Основная идея спиральной модели (рис.5) заключается в том, что на этапах анализа и проектирования реализуемость технических решений и степень удовлетворения потребностей заказчика проверяется путём создания прототипов. Каждый виток спирали соответствует созданию работоспособного фрагмента или версии системы, в процессе анализа которых проводится конкретизация деталей проекта. В результате выбирается обоснованный вариант, который действительно удовлетворяет требованиям заказчика.

Итеративная разработка отражает объективно существующий спиральный цикл создания сложных систем. Она позволяет переходить на следующий этап, не дожидаясь полного завершения работы за счёт частичной реализации функциональности программного продукта. Тем самым активизируется процесс уточнения и дополнения требований со стороны пользователей системы.

Рис. 5. Спиральная модель ЖЦ разработки

2.3. Процессная модель

Стоит различать понятие модели жизненного цикла и жизненный цикл конкретного программного продукта. Конкретный программный проект обычно использует различные модели ЖЦ для различных компонент программного продукта. Некоторые компоненты могут быть заимствованы из предшествующих проектов или приобретены на рынке ПО, другие проходят итерационный спиральный или каскадный ЖЦ. На практике мы имеем дело со множеством параллельно протекающих процессов разработки, каждый из которых может находиться в состоянии, отличном от других (рис.6).

Так, первая компонента разрабатывается по классическому жизненному циклу: требования, проектирование, реализация, интеграция. Вторая компонента - по сокращенному жизненному циклу. Третья относится к разряду приобретаемых или заимствованных компонент и потому проходит интеграционные испытания сразу после определения к ней требований.

Разработка четвертой компоненты явно определяется спиральным ЖЦ, в рамках которого последовательно реализуется два прототипа. На основании интеграционных исследований прототипов каждый раз производится уточнение требований. На третьем витке спирали, наконец, производится полномасштабное проектирование (создание документов описания архитектуры системы) и реализация окончательного варианта программной компоненты.

Применение процессной модели позволяет более точно отразить ход программной разработки, подчеркивая тот факт, что программный комплекс не однороден и содержит части, степень сложности которых сильно различается. Какие-то функции системы реализовались уже не один раз и могут использоваться практически без изменений в новых проектах. Другие требуют проведения исследовательских работ, многократного моделирования, оценки различных вариантов их реализации.

В общем случае следует отличать ЖЦ разработки ПО, который описывает именно процессы разработки, и ЖЦ самого ПО. В жизненном цикле проекта (разработки) присутствуют такие этапы, как формирование и обучение коллектива, закупка оборудования, создание стендов отладки и тестирования. ЖЦ программного продукта определяет, в свою очередь, процессы, связанные с функционированием самой программы, такие как:

- установка/настройка;
- эксплуатация;

- обновление;
- резервное копирование;
- временный останов;
- вывод из обращения.

Таким образом, можно и нужно говорить и о таком понятии, как ЖЦ проекта разработки ПО, который обычно на фазе инициализации включает в себя и подписание контракта с заказчиком, приобретение лицензий на инструментальное ПО, формирование среды информационной поддержки проекта, подбор и обучение персонала и т.п.

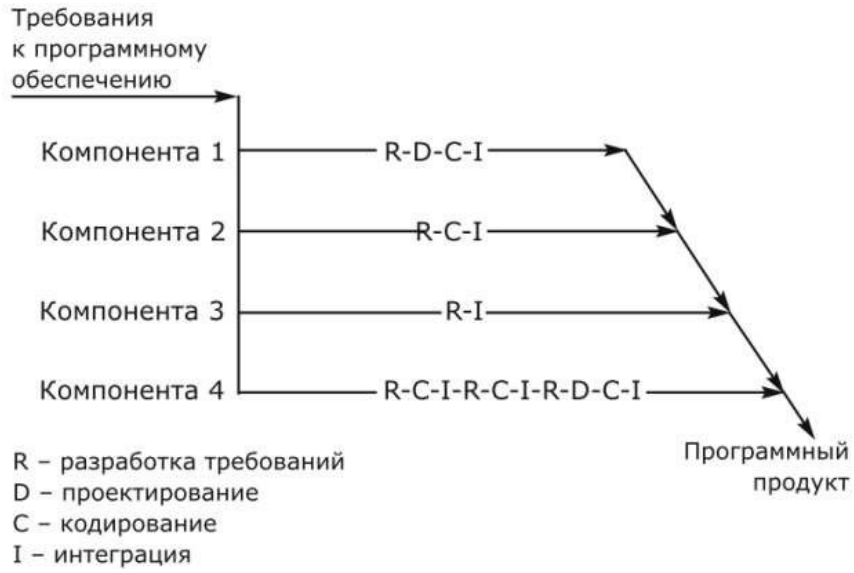


Рис. 6. Процессная модель разработки программного обеспечения

3. ПРОЕКТИРОВАНИЕ ПРОГРАММЫ

3.1. Введение

Разработка любой программы включает три этапа:

1) Проектирование (постановка задачи, построение математической модели, разработка алгоритма (алгоритмизация);
2) кодирование;
3) отладка.

Целью этапа ***проектирование*** является получение алгоритма решения задачи по переработке информации. Целью этапа ***кодирование*** является запись полученного алгоритма на выбранном языке (языках) программирования. ***Отладка*** программы включает ***тестирование*** (проверку правильности ее работы) и ***исправление ошибок***, найденных в результате тестирования.

Только для очень небольших программ этапы разработки программы выполняются строго последовательно во времени (рис. 6). На практике эти этапы выполняются ***параллельно*** (одновременно). Т.е. одновременно какая-то часть программы проектируется, другая ее часть кодируется, а третья – отлаживается. Естественно, что проектирование при этом несколько опережает этап кодирования, а кодирование – отладку (рис. 7).

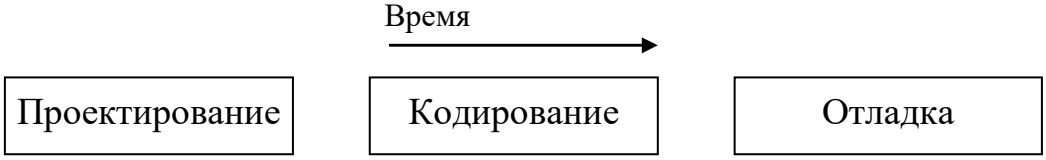


Рис. 6. Последовательное выполнение этапов разработки программы

Существующие технологии программирования различаются между собой реализацией указанных трех этапов. Главным из них является этап проектирования. При правильном его выполнении этапы кодирования и отладки выполняются достаточно просто. И наоборот, недостаточно хорошо выполненное проектирование программы всегда предшествует чрезмерно продолжительным этапам кодирования и отладки.

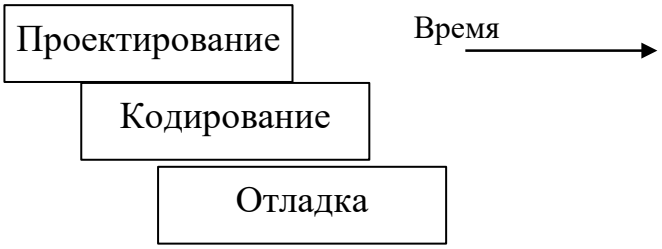


Рис. 7. Параллельное выполнение этапов разработки программы

После завершения разработки программы начинается ее ***сопровождение***. Во время сопровождения решаются две основные задачи:

1) исправление выявляющихся в ходе эксплуатации ошибок;
2) корректировка программы, вызванная изменениями в решаемой ею задаче по переработке информации.

3.2. Результаты проектирования

В процессе проектирования программы обычно получают три вида документов: 1) системную схему; 2) дерево подпрограмм; 3) алгоритмы подпрограмм.

Системная схема – небольшой документ, показывающий взаимодействие между программой – с одной стороны, и устройствами ввода-вывода и файлами во внешней памяти – с другой. Иными словами, системная схема описывает кратко интерфейс программы. На рис.8 приведен пример системной схемы для программы, выполняющей запросы своих пользователей по предоставлению сведений о сотрудниках организации.

Краткое описание интерфейсов программы, предоставляемое системной схемой, необходимо дополнить их подробным словесным описанием. Для этого необходимо описать структуру входных и выходных сообщений программы. ***Входные сообщения*** – сообщения, вводимые в программу пользователем с помощью клавиатуры терминала. ***Выходные сообщения*** – сообщения, выводимые программой пользователю (на экран его терминала). Кроме того, необходимо привести описание файлов, с которыми работает программа.

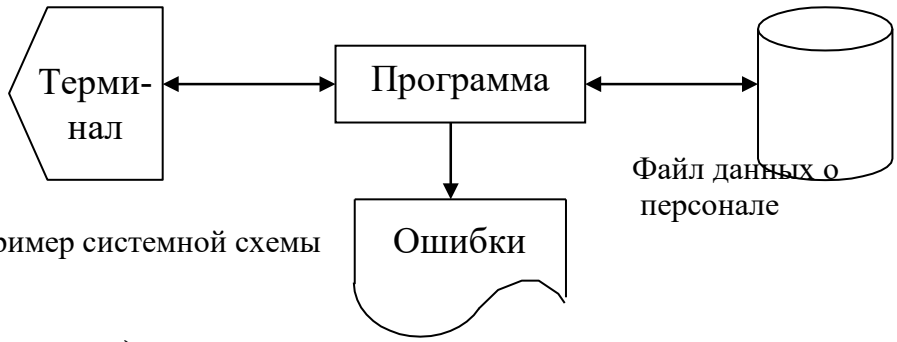


Рис. 8 Пример системной схемы

Дерево подпрограмм показывает – из каких подпрограмм состоит программа и как эти подпрограммы связаны по управлению. ***Связь по управлению*** – вызов (инициирование) одной подпрограммы другой. Причем системные подпрограммы в дерево подпрограмм обычно не включают.

Корнем дерева подпрограмм является процедура, называемая ***главной*** (***основной***) ***подпрограммой***. На следующем уровне дерева находятся подпрограммы, которые вызываются из основной подпрограммы и т.д. (рис. 9).

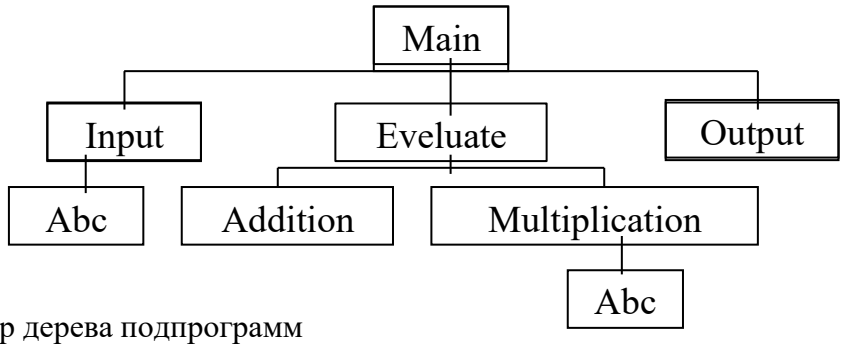


Рис. 9. Пример дерева подпрограмм

Как видно из рис.9, одна и та же подпрограмма (Abc) может быть многократно изображена на одном и том же дереве подпрограмм. Реально в памяти находится лишь одно тело подпрограммы, а ее повторения в дереве делаются для того, чтобы не затемнять рисунок пересечениями. (Дерево без пересечений воспринимается намного лучше.)

Проектирование любой подпрограммы включает два этапа: 1) определение интерфейсов; 2) получение алгоритма. На этапе ***определения интерфейсов*** подпрограмма рассматривается как “черный ящик” и для нее определяются границы (интерфейсы) с внешним миром. Основные подсистемы внешнего мира, с которыми взаимодействует подпрограмма:

1) подпрограмма (подпрограммы), вызывающая данную подпрограмму;
2) структуры данных (переменные, файлы), с которыми работает данная подпрограмма, но которые определены вне ее;
3) пользователь разрабатываемой программы.

Из перечисленных трех типов внешних подсистем только вызывающая подпрограмма является обязательной. Остальные подсистемы (пользователь и внешние структуры данных) при разработке конкретной подпрограммы могут отсутствовать. В этом случае интерфейс подпрограммы (процедуры) – перечень данных, которыми она обменивается с вызывающей ее программой. Сюда относятся входные и выходные параметры процедуры, а также области данных, на которые эти параметры указывают. Чем этот перечень меньше, тем лучше.

Результаты этапа выявления интерфейсов подпрограммы могут быть зафиксированы в системной схеме подобно тому, как это делалось для всей разрабатываемой программы. Но обычно это не делают, а ограничиваются описанием интерфейсов в виде вводных комментариев подпрограммы. В любом случае точное описание интерфейсов подпрограммы должно предшествовать разработке ее алгоритма. Иными словами, не выполнив постановку задачи, нельзя приступать к ее решению.

На этапе ***разработки алгоритма*** выявляется совокупность шагов, обеспечивающих перевод входных данных подпрограммы в ее выходные данные. Для представления алгоритма может быть использован один из специально предназначенных для этого языков, например, ***язык блок-схем***.

3.3. Методы проектирования программ

Применение любого метода проектирования начинается с получения системной схемы, т.к. не уточнив задачу, которую должна решать программа, дальнейшее ее проектирование не имеет смысла. Другие результаты проектирования (дерево подпрограмм и алгоритмы подпрограмм) могут быть получены различным образом во времени, в зависимости от используемого метода проектирования. Все эти методы можно разбить на три группы, различающиеся выбором объектов (подпрограмм), подлежащих проектированию:

1) проектирование “сверху-вниз”;

2) проектирование “снизу-вверх”;
3) проектирование “из центра”.

В методе проектирования **сверху вниз** (метод пошаговой детализации) построение дерева подпрограмм начинается с корня дерева, т.е. с главной подпрограммы. Для этой подпрограммы разрабатывается алгоритм, отвечающий требованиям структурного программирования. После этого принимается решение о способе реализации каждого этапа алгоритма. При этом относительно простые этапы алгоритма должны кодироваться в рамках данной подпрограммы, а более сложные этапы должны быть реализованы путем вызова соответствующих подпрограмм. Имена этих подпрограмм заносятся в дерево подпрограмм. Далее выбирается любая нерассмотренная подпрограмма и для нее выполняется разработка алгоритма. Подобный рост дерева подпрограмм продолжается до тех пор, пока не останется подпрограмм, требующих алгоритмизации.

Метод пошаговой детализации
(метод разработки "сверху-вниз", метод "нисходящего проектирования")

А. В решаемой задаче выделяется небольшое число (3-5) подзадач (более простые самостоятельные задачи). В проектируемой программе намечается соответствующее число блоков (подпрограмм) для решения подзадач.

Для блоков определяются:

- назначение;
- порядок выполнения;
- связь блоков по обрабатываемым данным.

Важно определение функционального назначения блока (что вход и что выход, что он делает), как делает - не уточняется.

Б. Общая схема программы составлена - проверяем правильность.

В. Если отдельные подзадачи достаточно сложные, повторяем процедуру детализации; выделяем новые подзадачи в данной подзадаче.

Г. Кодируем.

Типичные блоки

1. Задание исходных данных. Контроль правильности исходных данных.
2. Решение поставленной задачи.
3. Выдача результатов.

Особенности пошаговой детализации

- На каждом шаге детализации (уточнения) принимается небольшое число решений.
- Детализация блока проходит локально, независимо от остальной программы. Проверка правильности блока может проходить независимо от правильности всей программы.
- Проектирование программы не связано с тем языком, на котором ведется кодирование.
- Блоки должны быть достаточно независимы друг от друга.

Критерии проектирования (выбора) блоков:

1) сложность взаимодействия блока с другими блоками должна быть меньше сложности его внутренней структуры;
2) хороший блок снаружи проще, чем внутри;
3) хороший блок проще использовать, чем построить.

Сначала отлаживается главная программа, в которой вместо еще не написанных подпрограмм стоят "заглушки" ("муляж" подпрограммы - вместо работы печатает только сообщение о своей работе). По мере детализации "заглушки" заменяются на написанные подпрограммы, тем самым проверяется работа подпрограмм.

В методе проектирования **снизу вверх** первоначально проектируются подпрограммы, выполняющие сравнительно простые функции. (Разработка многих из этих подпрограмм сводится к выбору готовых подпрограмм, представленных в соответствующих библиотеках.) Далее разрабатываются подпрограммы, выполняющие более сложные функции и пользующиеся услугами ранее разработанных подпрограмм. Этот процесс продолжается до тех пор, пока очередная проектируемая подпрограмма не будет решать исходную задачу по переработке информации.

В методе проектирования **из центра** первоначально разрабатываются подпрограммы, предназначенные для решения задач по переработке информации, которые будут полезны (предположительно) для решения основной задачи. Но в отличие от метода “снизу-вверх” эти подпрограммы пользуются услугами таких подпрограмм, по крайней мере часть из которых на данный момент времени еще не разработана. Далее выполняется проектирование этих подпрограмм (движение “вниз”), а также проектирование вышестоящих подпрограмм (движение “вверх”). Движение “вверх” выполняется и заканчивается аналогично методу “снизу-вверх”, а движение “вниз” – аналогично методу “сверху-вниз”.

4. Алгоритмы

Под ***алгоритмом*** понимается способ преобразования представления информации. Algorithm - произошло от имени аль-Хорезми - автора известного арабского учебника по математике.

Алгоритм - *свод конечного числа правил, задающих последовательность выполнения операций при решении той или иной специфической задачи.*

Алгоритмы типичным образом решают не только частные задачи, но и классы задач. Подлежащие решению частные задачи, выделяемые по мере надобности из рассматриваемого класса, определяются с помощью параметров. Параметры играют роль исходных данных для алгоритма.

Пять важных особенностей алгоритма

Конечность (финитность). Алгоритм должен всегда заканчиваться после конечного числа шагов.

Определенность. Каждый шаг алгоритма должен быть точно определен.

Ввод. Алгоритм имеет некоторое (быть может, равное нулю) число входных данных, т. е. величин, заданных ему до начала работы.

Вывод. Алгоритм имеет одну или несколько выходных величин, т. е. величин, имеющих вполне определенное отношение ко входным данным.

Эффективность. Это означает, что все операции, которые необходимо произвести в алгоритме, должны быть достаточно простыми, чтобы их можно было в принципе выполнить точно и за конечный отрезок времени с помощью карандаша и бумаги.

Алгоритм должен быть практичным и хорошим с эстетической точки зрения.

Для алгоритмов важно различать следующие аспекты:

- постановку задачи, которая должна быть разрешена с помощью алгоритма;
- специфичный способ, каким решается задача, при этом для алгоритма различают:
  - (а) элементарные шаги обработки, которые имеются в распоряжении;
  - (б) описание выбора отдельных подлежащих выполнению шагов.

Для алгоритмически разрешимой постановки задачи всегда имеется много различных способов ее решения, т. е. различных алгоритмов.

Примеры "почти" алгоритмов: медицинский и кулинарный рецепты.

Примеры неформальных описаний алгоритмов

1. Арифметические операции над десятичными числами.

2. Алгоритм Евклида для вычисления наибольшего делителя (НОД).

*Постановка задачи***.** Пусть даны два положительных целых числа a и b, надо найти наибольший общий делитель НОД(a,b) чисел a и b.

2а. Первая версия алгоритма:

- если a=b, то справедливо НОД(a,b)=a;
- если a<b, то применяем алгоритм НОД к числам a и b-a;
- если b<a, то применяем алгоритм НОД к числам a-b и b.

Используется математическое свойство:

для любых положительных целых чисел x и y если x<y , то НОД(x,y)=НОД(y-x,x).

2б. Вторая версия алгоритма:

если a<b , то меняем местами значения (так, чтобы стало a>=b);
делим a на b, пусть r - остаток от деления (имеем a>=b>r>=0);
если r=0, то b- выход;
положить (заменить) a на b, b на r и вернуться к шагу 2.

3. Сортировка колоды карт.

*Постановка задачи***.** Имеется колода карт. Пусть на каждой карте зафиксировано одно натуральное число (ради простоты будем считать, что все числа попарно различны). Требуется отсортировать, т. е. упорядочить, колоду карт так, чтобы зафиксированные на картах числа образовывали монотонную (возрастающую или убывающую) последовательность.

4. Алгоритм вычисления дроби (a+b)/(a-b).

Сначала вычисляем (используя алгоритмы сложения и вычитания) значения выражений a+b и a-b (все равно, последовательно или одновременно), а потом образуем частное от деления полученных результатов (используя алгоритм деления).

Хотя описание алгоритма конечно и постоянно, количество фактически выполняемых тактов - величина переменная. Это оказывается возможным благодаря итерации (алгоритмы 2б,3б,3в,3г) или рекурсии (алгоритмы 2а, 3а,5). При словесном описании итерацию часто записывают в следующей форме "Пока выполнено определенное условие, повторяй". Рекурсия - это когда поставленная задача решается с помощью решения той же самой задачи, но с несколько измененными (более простыми) параметрами.

Классические элементы, которые встречаются в описаниях алгоритмов, это:

- выполнение элементарных шагов;
- разветвление по условию;
- повторение и рекурсия.

Способы фиксирования результатов проектирования

А. Блок-схема

Сначала описание алгоритма можно представлять, используя диаграмму управления потоком, в укрупненном блочном виде. Блок обычно не элементарен, не сводится к одному оператору, (его размеры неограниченны и выбираются произвольно). Другие блоки связаны с данным блоком только через точки входа и выхода. Поэтому если блок правильно решает задачу, т. е. всегда дает нужный результат, то его внутренняя структура несущественна для остальной части алгоритма. Отсутствие детального описания

внутренней структуры блока не мешает пониманию того, как работает алгоритм в целом; важно лишь, чтобы было четко определено, какие блоки запускают данный блок в работу, где лежит его исходная информация, где будет записан результат и куда переходить после окончания его работы.

Потом работа крупных блоков уточняется и заменяется на соответствующую блок-схему. После того, как получена достаточно подробная блок-схема программы, пишется код программы.

Недостатки блок-схем при проектировании:

- трудоемкость вычерчивания и перечерчивания при модификациях;
- потеря наглядности при детализации.

Б. Специальный язык - псевдокод

Правила обработки данных и условия не формализуются - запись на естественном языке. В качестве языка проектирования можно использовать любой язык программирования (например, Бейсик, Паскаль, Си), задавая содержание еще не детализированных блоков в виде. При детализации блоков комментарии следует оставлять.

При хороших комментариях структур данных, алгоритма и сложных мест в структурированной программе нет необходимости в никакой другой документации!

Пример описания на псевдокоде алгоритма

```
{ Алгоритм, распознающий, можно ли получить  последовательность знаков a из последовательности
знаков b посредством вычеркивания некоторых знаков.}
if {a - пустая последовательность знаков} then {ответ= "да"} else
if {b - пустая последовательность} then {ответ= "нет"}
else
        begin
                {сравнить первый знак последовательности a
                с первым знаком последовательности b}
                if {знаки совпадают}
                then { надо снова применить тот же алгоритм к остатку последовательности a и
                        остатку последовательности b}
                else { нужно снова применить тот же алгоритм к исходной последовательности a и
                        остатку последовательности b}
        end;
```

4.1. Алгоритмические языки

Использованный ранее неформальный способ описания алгоритмов отличается следующими недостатками:

- громоздок и излишне многословен;
- неоднозначность понимания.

Для представления, улучшения читаемости и для простоты представления алгоритмов, которые будут выполняться на компьютере, применяются ***алгоритмические языки*** (языки программирования).

Запись алгоритма на языке программирования называется **п*рограммой***.

При конструировании алгоритмического языка необходимо

- исходить из некоторого набора вычислительных структур (структур данных);
- вводить как составные операции, так и составные (структурированные) объекты;
- выбрать форму, удобную как для человека, который описывает алгоритм, так и для человека, который должен будет читать и понимать этот алгоритм, - форму, соответствующую кругу человеческих понятий и представлений.

4.2 Трансляция и выполнение

Компьютер - это автомат, выполняющий алгоритм. Для того, чтобы решить какую-либо задачу с помощью ЭВМ, необходимо алгоритм ввести в память машины, затем он должен интерпретироваться (т. е. восприниматься и выполняться) аппаратным путем.
Запись алгоритма на языке, понятном машине называется ***машинной программой***, а язык - ***машинный язык.*** Для разных типов компьютеров машинные языки разные.
Машинные операции кодируются с помощью цифрового кода (номера операции) и информации об операндах - адрес ячейки памяти, отведенной для хранения операнда. Следовательно, машинная программа ненаглядна, трудно понимаемая для человека.
Отличия алгоритмических языков от машинных языков:

- большая выразительностная возможность, алфавит алгоритмического языка шире алфавита машинного языка;
- набор операций не зависит от набора машинных операций и определяется удобством формулирования алгоритмов;
- одно предложение языка задает достаточно содержательный процесс обработки;
- удобное задание для человека операций, например, математическая запись;
- операнды имеют имена, выбираемые программистом (более удобно чем адреса);
- более широкий выбор типов данных и, более общо, широкий набор вычислительных структур.

Алгоритм на алгоритмическом языке не может быть выполнен непосредственно на компьютере. Он должен быть переведен на машинный язык.
Если возможен перевод алгоритмического языка на машинный по формальным правилам, то перевод осуществляется машиной, с помощью выполнения определенной машинной программы. Такая программа называется ***транслятором*** с данного алгоритмического языка (на данный машинный язык).
Обратите внимание! Транслятор - это алгоритм, для которого входом служит текст на алгоритмическом языке, а выходом - текст на машинном языке.
Транслятор является, грубо говоря, либо компилятором либо интерпретатором. ***Компилятор*** читает всю программу целиком и делает ее перевод на машинный язык и помещает команды в память компьютера. После того, как программа откомпилирована, исходная программа больше не нужна. ***Интерпретатор*** преобразует лишь небольшой фрагмент исходной программы в машинные команды, а затем дождавшись, когда компьютер их выполнит, переходит к обработке следующего фрагмента (пример - Бейсик).
Если мы всегда пишем правильные программы и имеем возможность работать с хорошим компилятором, то для практических целей можно условно считать компилятор и вычислительную машину единой машиной, которая может непосредственно выполнять программы. Тем самым, мы можем совершенно не задумываться о процессе компиляции. Комбинация компилятора и вычислительной машины иногда называется ***виртуальной машиной.***
Нам не нужно было бы вникать в структуру виртуальной машины, если бы не возможность возникновения ошибок.
Существуют три типа ошибок в программах:

- ***Синтаксические ошибки*** - ошибки, которые можно обнаружить при компиляции.
- ***Ошибки, обнаруживаемые при выполнении рабочей программы*** (при некоторых входных данных программа не работает, например, деление на нуль).
- Ошибки, которые не обнаруживаются компьютером ни при компиляции ни при выполнении программы (**неправильный алгоритм**).

Лекция 2
ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C#

1. ВВЕДЕНИЕ

Как правило, чтобы выполнить программу на C#, необходимо пройти через 6 этапов.

Первый этап представляет создание и редактирование файла с исходным текстом программы. Он может выполняться с помощью простейшего редактора текстов программ. Программист набирает в этом редакторе свою C# программу. При необходимости он снова обращается к ней и вносит с помощью этого редактора изменения в исходный текст программы. Далее программа запоминается на диске.

Второй этап – исходный программный код переводится в специальный промежуточный язык - байт-код - это промежуточное представление, в которое может быть переведена компьютерная программа. По сравнению с исходным кодом, удобным для создания и чтения человеком, байт-код — это компактное представление программы, уже прошедшей синтаксический и семантический анализ. В нём в явном виде закодированы типы, области видимости и т. п. С технической точки зрения, байт-код представляет собой машинно-независимый код низкого уровня, генерируемый транслятором из исходного кода. По форме байт-код похож на машинный код, но предназначен для исполнения не реальным процессором, а виртуальной машиной.

Третий этап — это компиляция. Как правило, программы на языке C# содержат ссылки на различные функции, которые определены вне самой программы. Например, в стандартных библиотеках или в личных библиотеках программистов. Объектный код, созданный компилятором, содержит "дыры" на месте этих отсутствующих частей.

Четвертый этап — компоновка. Компоновщик связывает объектный код с кодами отсутствующих функций и создает, таким образом, исполняемый загрузочный модуль (без пропущенных "дыр").

Пятый этап — загрузка. Перед выполнением программа должна быть размещена в памяти. Это делается с помощью загрузчика, который забирает загрузочный модуль программы с диска и перемещает его в память.

Наконец, шестой этап — это выполнение. Программа редко заработает с первой попытки. Каждый из названных этапов может заканчиваться ошибкой или неудачей из-за ошибки.

Тогда программист должен вернуться к редактированию исходного текста программы. Он должен внести необходимые изменения в текст программы, предварительно его хорошо проанализировав. Затем снова пройти через все этапы работы с исходным текстом программы до получения работающего без ошибок загрузочного модуля.

2. ПЕРЕМЕННЫЕ

http://professorweb.ru/my/csharp/charp_theory/level3/3_6.php

В C#, как и в C++, Паскале, каждая переменная имеет **имя, тип, размер и значение**.

Типы данных имеют особенное значение в C#, поскольку это строго типизированный язык. Это означает, что все операции подвергаются строгому контролю со стороны компилятора на соответствие типов, причем недопустимые операции не компилируются. Следовательно, строгий контроль типов позволяет исключить ошибки и повысить надежность программ. Для обеспечения контроля типов все переменные, выражения и значения должны принадлежать к определенному типу.

Основные типы приведены на рисунке 1.

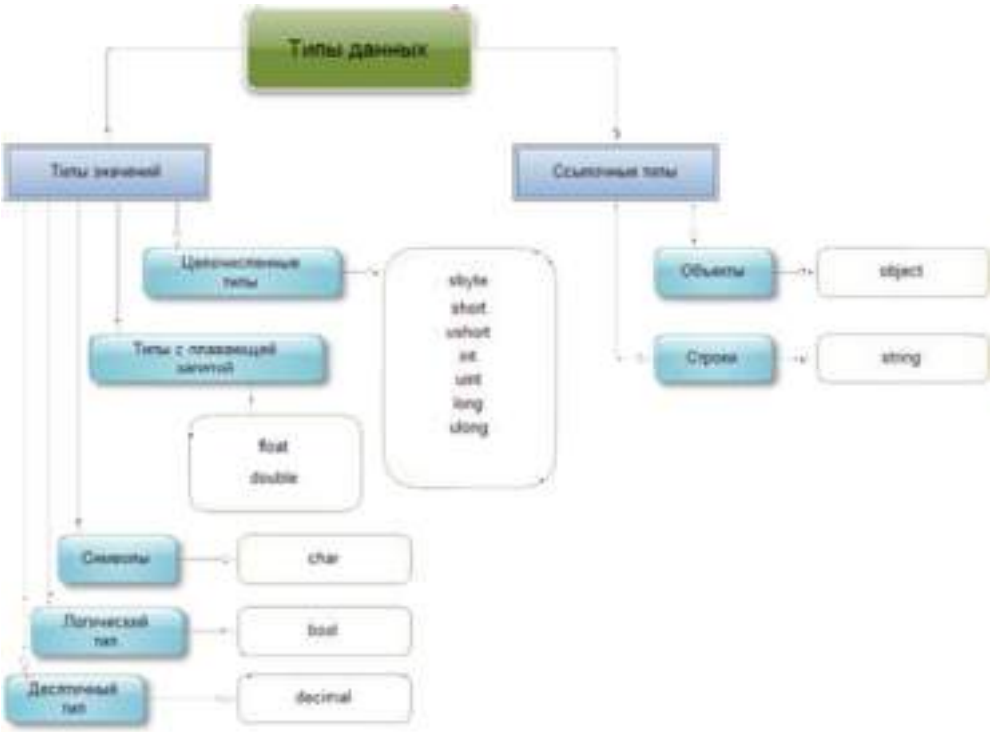


Рисунок 1 — Основные типы переменных в C#

Таблица 1 – Целочисленные типы C#

Тип	Тип CTS	Разрядность в битах	Диапазон
byte	System.Byte	8	0:255
sbyte	System.SByte	8	-128:127
short	System.Int16	16	-32768 : 32767
ushort	System.UInt16	16	0 : 65535
int	System.Int32	32	-2147483648 : 2147483647
uint	System.UInt32	32	0 : 4294967295
long	System.Int64	64	-9223372036854775808 : 9223372036854775807
ulong	System.UInt64	64	0 : 18446744073709551615

Десятичный тип данных

Для представления чисел с плавающей точкой высокой точности предусмотрен также десятичный тип ***decimal***, который предназначен для применения в финансовых расчетах. Этот тип имеет разрядность 128 бит для представления числовых значений в пределах от 1E-28 до 7,9E+28. Вам, вероятно, известно, что для обычных арифметических вычислений с плавающей точкой характерны ошибки округления десятичных значений. Эти ошибки исключаются при использовании типа ***decimal***, который позволяет представить числа с точностью до 28 (а иногда и 29) десятичных разрядов. Благодаря тому, что этот тип данных способен представлять десятичные значения без ошибок округления, он особенно удобен для расчетов, связанных с финансами

Объявление типов переменных делается соответствующим служебным словом с последующим перечислением имен переменных:

`int i,j,k,l,m` — перечисленные переменные будут целого типа и т.п. Причем, объявление типов можно совместить с присваиванием.

Присваивание значения переменным делается с *помощью команды присваивания* (в C# используется *знак равенства*). Значения для символьных переменных заключаются в одинарные кавычки:

```
Int i=33; float a=.156;
Char f  =’5’, f2=’$’, ff=’1’
```

ДЗ – преобразование типов в C#

Основные арифметические и логические операции, допустимые в C#, приведены в таблице 2.

Таблица 2 — Основные арифметические и логические операции в C#

Арифметические операции	**символы**	**Логические операции**	**символы**
сложение	+	равно	==
вычитание	—	не равно	!=
умножение	*	больше	>
деление	/	меньше	<
вычисление остатка (деление по модулю)	%	больше или равно	>=
присваивание	=	меньше или равно	<=
отрицание	!	логическое умножение	&&
		логическое сложении	||

Присваивание значений переменных делается командой присваивания:j=5; i=j.

В C# имеются помимо привычных и сокращенные формы записи арифметических операций. Примеры:

Операция присваивания

a+=3 эквивалентно a=a+3
a– =3 эквивалентно a=a–3
a*=3 эквивалентно a=a*3
a/=3 эквивалентно a=a/3
a%=3 эквивалентно a=a%3

Операции инкремента

i++ постфиксная форма i=i+1;
++i префиксная форма i=i+1;

Операции декремента

i– – постфиксная форма i=i–1
– – i префиксная форма i=i–1

Операции инкремента и декремента применяются к целым числам. Различие между постфиксной и префиксной формами иллюстрируется приведенным ниже примером:

k=a+(i++) —d*(– – j); равнозначно – –j; k=a+i-d*j; i++;

3. ПРОГРАММА С ВЕТВЛЕНИЕМ

Опять-таки по аналогии со структурными языками будем называть программу, использующую структуру выбора, *программой с ветвлением*.

Структура выбора на C# имеет вид:

if (***условие*** **)** ***оператор 1;*** **else** ***оператор 2***

Обратим внимание, что в отличие, например, от Паскаля в таком операторе отсутствует служебное слово **then** и условие обязательно заключается в скобки. ***Оператор1*** исполняется в случае истинности ***условия***. ***Оператор2 —*** в случае ложности ***условия***.

В тех случаях, когда требуется выполнить несколько действий (в случае истинности или в случае ложности условия), то применяют фигурные скобки:
if ***условие*** **{*оператор 1_1;оператор 1_2}* else {*оператор 2_1; оператор 2_2}***

Существует сокращенная форма записи условного оператора:
if ***условие оператор.***
Условие может быть и весьма сложным. Здесь допускаются *конъюнкция* условий (одновременнное выполнение условий) — условия связываются при помощи &&, *дизъюнкция* условий (альтернативное выполнение условий) — обозначается | | и *инверсия* условий (обозначается знаком !).

4. ЦИКЛ С ПАРАМЕТРОМ

На языке C# циклы с параметром организуются с помощью структуры повторения
For (*парам*=нач.знач; *парам*<кон.знач; *парам*++) {*тело цикла}*

Отметим, что параметр может изменяться и не на единицу, а на другое (целое) число. Тогда вместо ***парам***++ следует писать, например, ***парам*+=2** (если параметр приращается на 2).

Пример 4.1 Рассчитать для заданного N величину N!

```
static void Main(string[] args)
        {
         int  i;
         float p = 1;
         var N = Single.Parse(System.Console.ReadLine());
         for (i = 1; i < N + 1; i++) p = p * i;
         Console.WriteLine("N!=" + p);
         System.Console.ReadKey();
        }
```

Разумеется, внутри цикла можно организовывать и вложенные циклы, причем число вложений, в принципе, не ограничено.

Пример 4.2: Получить таблицу умножения в виде таблицы Пифагора:

```
static void Main(string[] args)
        {
            int i, j;
            string l;
            l="";

                for (i = 1; i <= 9; i++)
                {
                  Console.WriteLine("\n");
                    l = "";
                    for (j = 1; j <= 9; j++)
                    {
                        var k = i * j;
                        if (k < 10) l = l + "  " + k;
                            else l = l + " " + k;

                    }
                     Console.WriteLine(l);
                }
```

```
            System.Console.ReadKey();
        }
```

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

5. ЦИКЛ—ПОКА

В C# имеется также структура повторения **While**, которая служит для организации цикла—ПОКА:

While (*условие*) {*тело цикла}*

Работает он так: ***тело цикла*** повторяется, пока заданное ***условие*** остается истинным.

Пример 6.1 Рассчитать сумму вида:

$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots$ *заданной точностью E.*

Иначе говоря, проводить суммирование, пока очередное слагаемое является большим E (по абсолютной величине).

```
static void Main(string[] args)
        {
            int i, k;
            double sum, e;

            Console.WriteLine("Введите E:");
            e = Convert.ToDouble(Console.ReadLine());
            sum = 0;
            i=1;
            k=1;
            while (1.0/i > e)
            {
                sum = sum + (double)k/i;
                i++;
                k = k * (-1);
            }
            Console.WriteLine("Результат:"+ sum);
            System.Console.ReadKey();
        }
    }
```

В данной программе использован известный нам алгоритм суммирования. Суть его в том, что вводится переменная (здесь это переменная ***s***), задается ей нулевое значение, а затем в нее накапливается сумма. Очередное слагаемое — обратное к очередному числу

натурального ряда. Однако четные (по номеру) слагаемые идут со знаком "минус", нечетные — со знаком "плюс". Для учета этого знака введена дополнительная переменная ***k***, которая последовательно принимает значение то +1, то –1. Переменная *i* последовательно дает очередное натуральное число (2, 3, 4 и т.д.). Для удобства она описана как целая (integer).

Обратите внимание, что нам здесь потребовалось произвести в программе преобразование типов (**целый** к типу **double** при выполнении суммы), иначе в результате деления целого на целое получался нуль! И в условии, задающем цикл, мы сделали явное преобразование целого к вещественному — записав вместо целой единицы вещественную единицу (1.0).

Для изменения порядка работы в цикле в языке C# применяются два оператора: **break** (для досрочного прерывания цикла) и **continue** – для пропуска следующих за ним операций к переходу в конец цикла, но из цикла выхода не производится.

Цикл do while

Цикл *do while* очень похож на цикл *while*. Единственное их различие в том, что при выполнении цикла *do while* один проход цикла будет выполнен независимо от условия. Решение задачи на поиск суммы чисел от 1 до 1000, с применением цикла *do while*.

```
#include <iostream>
intmain()
{
 inti=0;// инициализируем счетчик цикла.
 doublesum=0;// инициализируем счетчик суммы.
 do{// выполняем цикл.
  i++;
  sum+=i;
 }while(i<1000);// пока выполняется условие.
 cout<<"Сумма чисел от 1 до 1000 = "<<sum;
 return0;
}
```

```
#include <iostream.h>
//сумма бесконечного ряда
main()
{int i,k; double s,e;
cout<<"введи точность \n"; cin>>e;
s=0; i=1; k=1;
do {
s=s+(double)k/i;
i++;
k=k*(-1);}
while (1.0/i>e);
cout<<" сумма ="<<s<<" с точностью "<<e;
return 0;}
```

Лекция 3
ФУНКЦИИ (МЕТОДЫ) В C#

Первыми формами модульности, появившимися в языках программирования, были процедуры и функции. Они позволяли задавать определенную функциональность и многократно выполнять один и тот же параметризованный программный код при различных значениях параметров. Поскольку функции в математике использовались издавна, то появление их в языках программирования было совершенно естественным. Уже с первых шагов процедуры и функции позволяли решать одну из важнейших задач, стоящих перед программистами, - задачу повторного использования программного кода. Встроенные в язык функции давали возможность существенно расширить возможности языка программирования. Важным шагом в автоматизации программирования было появление библиотек процедур и функций, доступных из используемого языка.

Долгое время процедуры и функции играли не только функциональную, но и архитектурную роль. Весьма популярным при построении программных систем был метод функциональной декомпозиции «сверху вниз», и сегодня еще играющий важную роль. Но с появлением объектно-ориентированного программирования (ООП) архитектурная роль функциональных модулей отошла на второй план. Для объектно-ориентированных языков, к которым относится и язык C#, в роли архитектурного модуля выступает класс. Программная система строится из модулей, роль которых играют классы, но каждый из этих модулей имеют содержательную начинку, задавая некоторую абстракцию данных.

Прежнюю роль библиотек процедур и функций теперь играют библиотеки классов. Библиотека классов FCL, доступная в языке C#, существенно расширяет возможности языка. Знание классов этой библиотеки и методов этих классов совершенно необходимо для практического программирования на C# с использованием всей его мощи.

Процедуры и функции связываются с классом, они обеспечивают функциональность данных класса и называются методами класса. Главную роль в программной системе играют данные, а функции лишь служат данным. Напомним еще раз, что в C# процедуры и функции существуют только как методы некоторого класса, они не существуют вне класса. В данном контексте понятие класс распространяется и на все его частные случаи - структуры, интерфейсы, делегаты.

В языке C# нет специальных ключевых слов - *procedure* и *function*, но присутствуют сами эти понятия. Синтаксис объявления метода позволяет однозначно определить, чем является метод - процедурой или функцией.

Функция отличается от процедуры двумя особенностями:

1. Она всегда вычисляет некоторое значение, возвращаемое в качестве результата функции;
2. И вызывается в выражениях.

Процедура C# имеет свои особенности:

1. Она возвращает формальный результат ***void*****,** указывающий на отсутствие результата;
2. Вызов процедуры является оператором языка;
3. И она имеет входные и выходные аргументы, причем выходных аргументов - ее результатов - может быть достаточно много.

Обычно метод предпочитают реализовать в виде функции тогда, когда он имеет один выходной аргумент, рассматриваемый как результат вычисления значения функции. Возможность вызова функций в выражениях также влияет на выбор в пользу реализации метода в виде функции. В других случаях метод реализуют в виде процедуры.

Функции

Функция представляет собой небольшую подпрограмму. Если просто программа - это решение какой-то прикладной задачи, то функция – это тоже решение, только уже в рамках программы и, соответственно, она выполняет задачу «попроще». Функции позволяют уменьшить размер программы за счет того, что не нужно повторно писать какой-то фрагмент кода – необходимо просто вызвать сколько угодно и где нужно объявленную функцию.

Функции в C# также называют методами. Между этими двумя понятиями разница небольшая.

До настоящего момента весь код был написан в функции *main*. Функция *main* является главной функцией приложения и точкой входа программы. Любая функция в C# может быть объявлена только в рамках класса, так как C# - полностью объектно-ориентированный язык программирования (ООП). Объявление пользовательской функции внутри другой функции (например, *main*) недопустимо.

Описание функции (метода)

Синтаксически в описании метода различают две части - описание заголовка и описание тела метода:

заголовок_метода
тело_метода

Объявление функции имеет следующую структуру:

[модификатор доступа] [тип возвращаемого значения] [имя функции-идентификатор] ([список_формальных_параметров])
{
// тело функции
}

Модификатор доступа (области видимости) может быть *public, private, protected, internal*. Модификатор *public* показывает, что метод открыт и доступен для вызова клиентами и потомками класса. Модификатор *private* говорит, что метод предназначен для внутреннего использования в классе и доступен для вызова только в теле методов самого класса.
Отметим, что если модификатор доступа опущен, то по умолчанию предполагается, что он имеет значение *private* и метод является закрытым для клиентов и потомков класса.

Пока будем везде использовать *public*.

Между модификатором и типом может стоять ключевое слово *static*, что означает, что функция будет статичной. Из статичной функции можно вызывать другие функции, если они тоже статичные. Главная функция *main* – всегда *static*, поэтому все функции пока будем объявлять также статичными.

Функция может возвращать значение или не возвращать. Если функция, например, возвращает целое число, то нужно указать тип *int*. Если функция не возвращает никакого значения, то для этого используется ключевое слово ***void***. Повторим, что функции, которые не возвращают значение, еще называют процедурами.

Имя функции – идентификатор - необходимо формировать так, чтобы имя отображало суть функции. Используйте глаголы или словосочетания с глаголами. Примеры: *GetAge(), Sort(), SetVisibility()*.

Вот несколько простейших примеров описания методов:

```
void A() {...};
int B(){...};
public void C(){...};
```

Методы A и B являются закрытыми, а метод C - открыт. Методы A и C реализованы процедурами, а метод B - функцией, возвращающей целое значение.

Параметры – это те данные, которые необходимы для выполнения функции. Если параметров несколько, они отделяются запятой. Параметры могут отсутствовать – тогда обязательно необходимо ставить ().

Параметры, задаваемые при описании функции (метода), называются ***формальными параметрами***.

Параметры, задаваемые при вызове функции (метода), называются ***фактическими параметрами***.

Рассмотрим синтаксис объявления формального параметра:

[ref | out | params] тип_параметра имя_параметра

Обязательным является указание типа и имени параметра. Отметим, никаких ограничений на тип параметра не накладывается. Он может быть любым скалярным типом.

Несмотря на фиксированное число формальных параметров, есть возможность при вызове метода передавать ему произвольное число фактических параметров. Для реализации этой возможности в списке формальных параметров необходимо задать ключевое слово ***params***. Оно задается один раз и указывается только для последнего параметра списка, объявляемого как массив произвольного типа. При вызове метода этому формальному параметру соответствует произвольное число фактических параметров.

Содержательно, все параметры метода разделяются на три группы:

1. входные,
2. выходные,
3. обновляемые.

Параметры первой группы передают информацию методу, их значения в теле метода только читаются.

Параметры второй группы представляют собой результаты метода, они получают значения в ходе работы метода.

Параметры третьей группы выполняют обе функции. Их значения используются в ходе вычислений и обновляются в результате работы метода.

Выходные параметры всегда должны сопровождаться ключевым словом ***out***, обновляемые - ***ref***. Что же касается входных параметров, то, как правило, они задаются без ключевого слова, хотя иногда их полезно объявлять с параметром *ref*. Отметим, если параметр объявлен как выходной с ключевым словом *out*, то в теле метода обязательно должен присутствовать оператор присваивания, задающий значение этому параметру. В противном случае возникает ошибка еще на этапе компиляции.

Повторим, что первая строка функции, где указываются тип, имя, параметры и т.д. называется **заголовком функции**.

Соответствие списков формальных и фактических параметров

Между списком формальных и списком фактических параметров должно выполняться определенное соответствие по числу, порядку следования, типу и статусу параметров. Если в первом списке *n* формальных параметров, то фактических параметров должно быть не

меньше *n* (соответствие по числу). Каждому *i*-му формальному параметру (для всех *i* от 1 до *n-1*) ставится в соответствие *i*-й фактический параметр. Последнему формальному параметру, при условии, что он объявлен с ключевым словом *params*, ставятся в соответствие все оставшиеся фактические параметры (**соответствие по порядку**).

Если формальный параметр объявлен с типом *T*, то выражение, задающее фактический параметр, должно быть согласовано по типу с типом *T*: допускает преобразование к типу *T*, совпадает с типом *T* или является его потомком (**соответствие по типу**).

Пример функции, которая не возвращает значение

Напишем простую функцию, которая будет заменять в массиве строк указанное имя на другое. Данная функция будет принимать три аргумента: массив строк, имя, которое необходимо заменить и новое имя. Так как функция не будет возвращать значение, указываем тип *void* перед именем функции.

```
public static void ReplaceName(string[] names, string name, string newName)
{
  for (int i=0; i < names.Length; i++)
  {
    if (names[i] == name)
    names[i] = newName;
  }
}
```

Сама функция очень простая. Проходим в цикле по элементам массива строк и смотрим, равен ли элемент указанному имени. Если да, то заменяем его на новое имя.

Функция написана, и теперь используем ее:

```
class Program
{

  public static void ReplaceName(string[] names, string name, string newName)
  {
    for (int i=0; i < names.Length; i++)
    {
      if (names[i] == name)
        names[i] = newName;
      }
    }

  static void main(string[] args)
  {
    string[] names = { "Sergey", "Maxim", "Andrey", "Oleg", "Andrey", "Ivan", "Sergey" };
    ReplaceName(names, "Andrey", "Nikolay"); // вызов функции. Все строки "Andrey" в массиве будут
заменены на "Nikolay"
    ReplaceName(names, "Ivan", "Vladimir");
  }
}
```

После вызова функции два раза в этой программе, массив будет выглядеть так:

"Sergey", "Maxim", " Nikolay ", "Oleg", " Nikolay ", " Vladimir ", "Sergey".

Пример функции, которая возвращает значения

Напишем функцию, которая будет находить максимальное число в массиве. Аргумент у этой функции будет один – массив целых чисел. Тип возвращаемого значения – целое число *int*.

```
public static int GetMax(int[] array)
{
  int max = array[0];
  for (int i = 1; i < array.Length; i++)
  {
    if (array[i] > max)
      max = array[i];
  }
  return max;
}
```

Логика функции проста. Создаем переменную *max*, в которую записываем первый элемент массива. Дальше в цикле сравниваем каждый элемент со значением в *max*, если элемент больше, чем *max*, то записываем в *max* этот элемент. В конце функции используем оператор ***return***, чтобы вернуть результат.

Оператор **return** должен быть обязательно в функции, которая возвращает значение.

Используем нашу функцию:

```
class Program
{
  public static int GetMax(int[] array)
  {
    int max = array[0];
    for (int i = 1; i < array.Length; i++)
    {
      if (array[i] > max)
        max = array[i];
    }
    return max;
  }

 static void main(string[] args)
  {
    int[] numbers = { 3, 32, 16, 27, 55, 43, 2, 34 };
    int max;
    max = GetMax(numbers); //вызов такой функции можно использовать при присваивании значения
    Console.WriteLine(GetMax(numbers)); // вызов функции также можно использовать как аргумент
при вызове другой функции. WriteLine() – тоже функция.
    Console.ReadKey();
  }
}
```

Оператор *return*

Когда встречается этот оператор, происходит выход из функции и код ниже (если он есть) выполняться не будет (например, в функцию передан такой параметр, при котором нет смысла выполнять функцию). Он похож на оператор break, который используется для выхода из циклов. Этот оператор также можно использовать и в функциях, которые не возвращают значение. Оператор return допустимо использовать несколько раз в функции, но этого делать не рекомендуется.

Рекурсия

Рекурсия является одним из наиболее мощных средств в арсенале программиста. Рекурсивные структуры данных и рекурсивные методы широко используются при построении программных систем. Рекурсивные методы, как правило, наиболее всего удобны при работе с рекурсивными структурами данных - списками, деревьями. Рекурсивные методы обхода деревьев служат классическим примером. Метод P называется рекурсивным, если при выполнении тела метода происходит вызов метода P.

Рекурсия может быть прямой, если вызов P происходит непосредственно в теле метода P. Рекурсия может быть косвенной, если в теле P вызывается метод Q (эта цепочка может быть продолжена), в теле которого вызывается метод P.

Для того чтобы рекурсия не приводила к зацикливанию, в тело нормального рекурсивного метода всегда встраивается оператор выбора, одна из ветвей которого не содержит рекурсивных вызовов. Если в теле рекурсивного метода рекурсивный вызов встречается только один раз, значит, что рекурсию можно заменить обычным циклом, что приводит к более эффективной программе, поскольку реализация рекурсии требует временных затрат и работы со стековой памятью.

Приведем вначале простейший пример рекурсивного определения функции, вычисляющей факториал целого числа:

```
 //factorial
public long factorial(int n) {
        if (n <= 1) return 1;
        else return n * factorial(n - 1);
}
```

Функция factorial является примером прямого рекурсивного определения - в ее теле она сама себя вызывает. Здесь, как и положено, есть нерекурсивная ветвь, завершающая вычисления, когда n становится равным единице. Это пример так называемой «хвостовой» рекурсии, когда в теле встречается ровно один рекурсивный вызов, стоящий в конце соответствующего выражения. Хвостовую рекурсию намного проще записать в виде обычного цикла. Вот циклическое определение той же функции:

```
//fact
public long fact(int n) {
        long res = 1;
        for (int i = 2; i <= n; i++) res *= i;
        return (res);
}
```

Конечно, циклическое определение проще, понятнее и эффективнее, и применять рекурсию в подобных ситуациях не следует. Интересно сравнить время вычислений, дающее некоторое представление о том, насколько эффективно реализуется рекурсия. Вот соответствующий тест, решающий эту задачу:

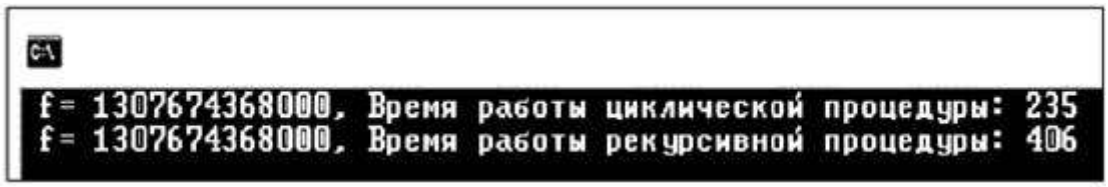
```
public void TestTailRec() {
        long time1, time2;
        long f = 0;
        time1 = getTimeInMilliseconds();
        for (int i = 1; i < 1000000; i++)
          f = fact(15);
        time2 = getTimeInMilliseconds();
        Console.WriteLine(" f= {0}, " + "Время работы циклической процедуры :{1}", f,time2 -time1);
        time1 = getTimeInMilliseconds();
        for (int i = 1; i < 1000000; i++)
          f = factorial(15);
        time2 = getTimeInMilliseconds();
        Console.WriteLine(" f= {0}, " + "Время работы рекурсивной процедуры:{1}", f,time2 -time1);
}
```

Каждая из функций вызывается в цикле, работающем 1000000 раз. До начала цикла и после его окончания вычисляется текущее время. Разность этих времен и дает оценку времени работы функций. Обе функции вычисляют факториал числа 15.

Проводить сравнение эффективности работы различных вариантов - это частый прием, используемый при разработке программ. Встроенный тип **DateTime** обеспечивает необходимую поддержку для получения текущего времени. Он совершенно необходим, когда приходится работать с датами. Статический метод **Now** класса **DateTime** возвращает объект этого класса, соответствующий дате и времени в момент создания объекта. Многочисленные свойства этого объекта позволяют извлечь требуемые характеристики. Приведем текст функции **getTimelnMilliseconds:**

```
private long getTimeInMilliseconds() {
        DateTime time = DateTime.Now;
        return (((time.Hour*60 + time.Minute)*60 + time.Second)*1000 + time.Millisecond);
}
```

Результаты измерений времени работы рекурсивного и циклического вариантов функций слегка отличаются от запуска к запуску, но порядок остается одним и тем же. Эти результаты показаны на рис. 22.



Вовсе не обязательно, что рекурсивные методы будут работать медленнее нерекурсивных. Классическим примером являются методы сортировки. Известно, что время работы нерекурсивной пузырьковой сортировки имеет порядок *c*n2*, где c - некоторая константа. Для рекурсивной процедуры сортировки слиянием время работы - *q*n*log(n),* где q - константа. Понятно, что для больших n сортировка слиянием работает быстрее, независимо от соотношения значений констант. Сортировка слиянием - хороший пример применения рекурсивных методов. Она демонстрирует известный прием, суть которого в том, что исходная задача разбивается на подзадачи меньшей размерности, допускающие решение тем же алгоритмом. Решения отдельных подзадач затем объединяются, давая решение исходной задачи. В задаче сортировки исходный массив размерности *n* можно разбить на два массива размерности *n/2*, для каждого из которых рекурсивно вызывается метод сортировки слиянием. Полученные отсортированные массивы сливаются в единый массив с сохранением упорядоченности.

Использование параметров. Передача параметров

Параметры объявляются в скобках после имени метода. Синтаксис объявления параметров такой же, как и у переменных. А областью действия параметров является тело метода.

В общем случае параметры могут передаваться методу либо **по значению**, либо **по ссылке**. Когда переменная передается по ссылке, вызываемый метод получает саму переменную, поэтому любые изменения, которым она подвергнется внутри метода, останутся в силе после его завершения. Но если переменная передается по значению, вызываемый метод получает копию этой переменной, а это значит, что все изменения в ней по завершении метода будут утеряны.

Если необходимо, чтобы вызываемый метод изменял значение параметра, необходимо его передавать по ссылке, используя ключевые слова **ref**.

Передача параметра по значению

В следующем примере демонстрируется передача параметров по значению. Переменная *n* передается с помощью значения в метод *SquareIt*. Любые изменения, выполняемые внутри метода, не влияют на значение переменной.

```
class PassingValByVal
{
   static void SquareIt(int x)
   // The parameter x is passed by value.
   // Changes to x will not affect the original value of x.
   {
      x *= x;
      System.Console.WriteLine("The value inside the method: {0}", x);
   }
   static void Main()
   {
      int n = 5;
      System.Console.WriteLine("The value before calling the method: {0}", n);

      SquareIt(n); // Passing the variable by value.
      System.Console.WriteLine("The value after calling the method: {0}", n);

      System.Console.WriteLine("Press any key to exit.");
      System.Console.ReadKey();
   }
}
/* Output:
   The value before calling the method: 5
   The value inside the method: 25
   The value after calling the method: 5
*/
```

Передача параметра по ссылке

Следующий пример такой же, как и предыдущий, за исключением того, что аргумент передается в качестве **ref** параметр.

```
class PassingValByRef
{
   static void SquareIt(ref int x)
   // The parameter x is passed by reference.
   // Changes to x will affect the original value of x.
   {
      x *= x;
      System.Console.WriteLine("The value inside the method: {0}", x);
   }
   static void Main()
   {
      int n = 5;
      System.Console.WriteLine("The value before calling the method: {0}", n);

      SquareIt(ref n); // Passing the variable by reference.
      System.Console.WriteLine("The value after calling the method: {0}", n);

      System.Console.WriteLine("Press any key to exit.");
      System.Console.ReadKey();
   }
}
/* Output:
   The value before calling the method: 5
   The value inside the method: 25
   The value after calling the method: 25
*/
```

В этом примере передается не значение переменной n, а ссылка на переменную n. Параметр x не является типом int; он является ссылкой на тип **int**, в данном случае ссылкой на переменную n. Следовательно, если x возводится в квадрат внутри метода, это означает, что в квадрат возводится то, что находится по ссылке.

Методы и их перегрузка

Существование в классе методов с одним и тем же именем называется ***перегрузкой***, а сами одноименные методы называются ***перегруженными***. Перегрузка методов полезна, когда требуется решать подобные задачи с разным набором аргументов.

Типичный пример - это нахождение площади треугольника. Площадь можно вычислить по трем сторонам, по двум углам и стороне, по двум сторонам и углу между ними и при многих других наборах аргументов. Считается удобным во всех случаях иметь для метода одно имя, например, ***Square*****,** и всегда, когда нужно вычислить площадь, не задумываясь, вызывать метод ***Square*****,** передавая ему известные в данный момент.

Перегрузка характерна и для знаков операций. В зависимости от типов аргументов, один и тот же знак может выполнять фактически разные операции. Классическим примером является знак операции сложения «+», который играет роль операции сложения не только для арифметических данных разных типов, но и выполняет конкатенацию строк.

Перегрузка требует уточнения **семантики вызова метода**. Когда встречается вызов неперегруженного метода, то имя метода в вызове однозначно определяет, тело какого метода должно выполняться в точке вызова. Когда же метод перегружен, то знания имени недостаточно - оно не уникально. **Уникальной характеристикой перегруженных методов является их сигнатура**. Перегруженные методы, имея одинаковое имя, должны отличаться либо числом аргументов, либо их типами, либо ключевыми словами (отметим: с точки зрения сигнатуры, ключевые слова ***ref*** и ***out*** не отличаются). Уникальность сигнатуры позволяет вызвать требуемый перегруженный метод.

Насколько полезна перегрузка методов? Здесь нет экономии кода, поскольку каждую реализацию нужно задавать явно; нет выигрыша по времени - напротив, требуются определенные затраты на поиск подходящей реализации, который может приводить к конфликтам, обнаруживаемым на этапе компиляции. В нашем примере (про площадь треугольника) вполне разумно иметь три метода с разными именами и осознанно вызывать метод, применимый к данным аргументам.

Существуют ситуации, где перегрузка полезна. Например, класс ***Convert*****,** у которого **16 методов** с разными именами, зависящими от целевого типа преобразования. Каждый из этих 16 методов перегружен, и в свою очередь, имеет 16 реализаций в зависимости от типа источника.

Приведём исходный код метода, тестирующего работу перегруженных функций:

```
/// Группа перегруженных методов A()
/// первый аргумент представляет сумму кубов
/// произвольного числа оставшихся аргументов
/// Аргументы могут быть разного типа.


private void A(out long p2, int p1)
{
        p2 = (long) Math.Pow(p1, 3);
        Console.WriteLine("Метод A-1");
}

private void A(out long p2, params int[ ] p)
{
p2 = 0;
for (int i = 0; i < p.Length; i++)
            p2 += (long) Math.Pow(p[i], 3);
Console.WriteLine("Метод A-2");
}

private void A(out double p2, double p1)
{
        p2 = Math.Pow(p1, 3);
```

```
        Console.WriteLine( “Метод  A-3");
}

private void A(out double p2, params double[ ] p)
{
p2 = 0;
for (int i = 0; i < p.Length; i++)
  p2 += Math.Pow(p[i], 3);
Console.WriteLine("Метод A-4");
}


public void TestLoadMethods() {
        long u = 0;
        double v = 0;
        A(out u, 7);
        A(out v, 7.5);
        Console.WriteLine("u= {0}, v= {1}", u, v);

        A(out v, 7);
        Console.WriteLine("v= {0}", v);

        A(out u, 7, 11, 13);
        A(out v, 7.5, Math.Sin(11.5) + Math.Cos(13.5), 15.5);
        Console.WriteLine("u= {0}, v= {1}", u, v);

} //TestLoadMethods
```

На рис. 21 показаны результаты этого тестирования.

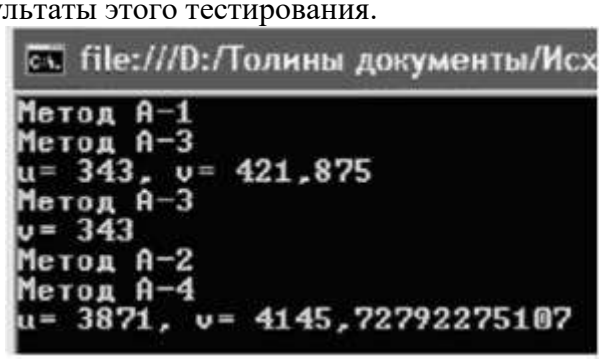


Рисунок 21. Тестирование перегрузки методов

Лекция 4
СТРУКТУРИРОВАННЫЕ ТИПЫ ДАННЫХ
МАССИВЫ

Массивы представляют собой упорядоченные коллекции элементов с одним и тем же типом данных. Доступ к ним осуществляется при помощи имени массива в совокупности со смещением от начала массива нужного элемента.

Общие сведения о массивах

Массив имеет следующие свойства.

Массив может быть одномерным, многомерным или массивом массивов.

Количество элементов в массиве задается, когда создается экземпляр массива. Это значения невозможно изменить во время существования экземпляра.

Значения по умолчанию числовых элементов массива задано равным нулю, а элементы ссылок имеют значение NULL.

Индексация массивов начинается с нуля: массив с n элементами индексируется от 0 до n-1.

Элементы массива могут быть любых типов, включая тип массива.

Типы массива являются *ссылочными типами*, производными от абстрактного базового типа Array.

Можно сохранить несколько переменных одного типа в структуре данных "массив". Массив объявляется указанием типа элементов.

```
type[]  arrayName;
```

В следующем примере показано создание одномерных, многомерных массивов.

```
class TestArraysClass
{
    static void Main()
    {
        // Declare a single-dimensional array
        int[] array1 = new int[5];

        // Declare and set array element values
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };
    }
}
```

```
// Синтаксис инициализации массива с использованием
// ключевого слова new
int[] myArr = new int[] {10,20,30,40,50};

// Синтаксис инициализации массива без использования
// ключевого слова new
string[] info = { "Фамилия", "Имя", "Отчество" };

// Используем ключевое слово new и желаемый размер
char[] symbol = new char[4] { 'X','Y','Z','M' };
```

Одномерный массив

Одномерный массив хранит фиксированное число элементов в линейном порядке, и для определения каждого элемента требуется лишь одно значение индекса. В C# квадратные скобки в объявлении массива должны следовать за типом данных. Поэтому массив типа integers объявляется с использованием следующего синтаксиса:

```
int[] arr1;
```

Следующее объявление является недопустимым в C#:

```
//int arr2[];  //compile error
```

После объявления массива необходимо установить его размер с помощью ключевого слова new. Ниже показано, как объявляется ссылка на массив.

```
int[] arr;
arr = new int[5];  // create a 5 element integer array
```

Доступ к элементам одномерного массива осуществляется при помощи индекса. Индексы массива C# также отсчитываются с нуля. Следующий код осуществляет доступ к последнему элементу предыдущего массива:

```
System.Console.WriteLine(arr[4]);  // access the 5th element
```

Инициализация

Элементы массива C# могут быть инициализированы при создании при помощи следующего синтаксиса:

```
int[] arr2Lines;
arr2Lines = new int[5] {1, 2, 3, 4, 5};
```

В C# число инициализаторов должно точно соответствовать размеру массива. Этот факт можно использовать для объявления и инициализации массива C# в одной строке:

```
int[] arr1Line = {1, 2, 3, 4, 5};
```

Этот синтаксис создает массив, размер которого равен числу инициализаторов.

Инициализация в цикле программы

Другим способом инициализации массива в C# является использование цикла for. Следующий цикл обнуляет все элементы массива:

```
int[] TaxRates = new int[5];

for (int i=0; i<TaxRates.Length; i++)
{
    TaxRates[i] = 0;
}
```

Многомерные массивы

В C# можно создавать регулярные многомерные массивы, которые представляют собой матрицу значений одного типа.

Объявить прямоугольный многомерный массив можно с помощью следующего синтаксиса:

```
int[,] arr2D;      // declare the array reference
float[,,,] arr4D;  // declare the array reference
```

После объявления данному массиву следует выделить память следующим образом:

```
arr2D = new int[5,4];  // allocate space for 5 x 4 integers
```

Затем для доступа к элементам массива используется приведенный ниже синтаксис:

```
arr2D[4,3] = 906;
```

Поскольку индексация массивов начинается с нуля, элементу в пятом столбце четвертой строки присваивается значение 906.

Инициализация

Многомерные массивы можно создавать, настраивать и инициализировать в одном операторе любым из следующих методов.

```
int[,] arr4 = new int [2,3] { {1,2,3}, {4,5,6} };
int[,] arr5 = new int [,]   { {1,2,3}, {4,5,6}, {4,5,6},{4,5,0} }

int[,] arr6 =               { {1,2,3}, {4,5,6} };
```

Инициализация в цикле программы

Все элементы массива могут быть инициализированы посредством вложенного цикла, как показано ниже.

```
int[,] arr7 = new int[5,4];

for(int i=0; i<5; i++)
{
    for(int j=0; j<4; j++)
    {
        arr7[i,j] = i*j;  // initialize each element to zero
    }
}
int S=0;
for(int i=0; i<5; i++)
{
    for(int j=0; j<4; j++)
    {
        S = s+arr7[i,j];  // initialize each element to zero
    }
}
```

Инициализация в цикле программы случайными числами

```
/ Объявляем двумерный массив
int[,] myArr = new int[4, 5];

Random ran = new Random();

// Инициализируем данный массив
for (int i = 0; i < 4; i++)
{
    for (int j = 0; j < 5; j++)
    {
       myArr[i, j] = ran.Next(1, 15);
       Console.Write("{0}\t", myArr[i, j]);
    }
    Console.WriteLine();
}
```

Массивы трех и более измерений

В C# допускаются массивы трех и более измерений. Ниже приведена общая форма объявления многомерного массива:

```
тип[,...,] имя_массива = new тип[размер1, размер2, ... размеры];
```

Ниже приведен пример программы, использующей трехмерный массив:

```
int[,,] myArr = new int[5,5,5];

for (int i = 0; i < 5; i++)
   for (int j = 0; j < 5; j++)
      for (int k = 0; k < 5; k++)
         myArr[i, j, k] = i + j + k;
```

Использование оператора foreach с массивами (

В C# также предусмотрен оператор foreach. Этот оператор обеспечивает простой и понятный способ выполнения итерации элементов в массиве. Например, следующий код создает массив numbers и осуществляет его итерацию с помощью оператора foreach.

```
int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
foreach (int i in numbers)
{
    System.Console.Write("{0} ", i);
}
//Output: 4 5 6 1 2 3 -2 -1 0
```

Этот же метод можно использовать для итерации элементов в многомерных массивах, например:

```
int[,] numbers2D = new int[3, 2] { { 9, 99 }, { 3, 33 }, { 5, 55 } };
// Or use the short form:
// int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write("{0} ", i);
}
// Output: 9 99 3 33 5 55
```

Массивы массивов

C# поддерживают создание массивов массивов, или непрямоугольных массивов, в которых каждая строка содержит различное число столбцов. Например, в следующем массиве массивов содержится четыре записи в первой строке и три — во второй.

```
int[][] jaggedArray = new int[2][];
jaggedArray[0] = new int[4];
jaggedArray[1] = new int[3];
```

Класс System.Array

В платформе .NET Framework массивы реализуются как экземпляры класса Array. Этот класс обеспечивает несколько ценных методов, например Sort и Reverse.

Следующий пример демонстрирует, насколько просто работать с этими методами. Сначала меняется порядок элементов в массиве с помощью метода Reverse, затем элементы сортируются методом Sort.

```
class ArrayMethods
{
    static void Main()
    {
        // Create a string array of size 5:
        string[] employeeNames = new string[5];
```

```
            // Read 5 employee names from user:
            System.Console.WriteLine("Enter five employee names:");
            for(int i=0; i<employeeNames.Length; i++)
            {
                employeeNames[i]= System.Console.ReadLine();
            }

            // Print the array in original order:
            System.Console.WriteLine("\nArray in Original Order:");
            foreach(string employeeName in employeeNames)
            {
                System.Console.Write("{0}  ", employeeName);
            }

            // Reverse the array:
            System.Array.Reverse(employeeNames);

            // Print the array in reverse order:
            System.Console.WriteLine("\n\nArray in Reverse Order:");
            foreach(string employeeName in employeeNames)
            {
                System.Console.Write("{0}  ", employeeName);
            }

            // Sort the array:
            System.Array.Sort(employeeNames);

            // Print the array in sorted order:
            System.Console.WriteLine("\n\nArray in Sorted Order:");
            foreach(string employeeName in employeeNames)
            {
                System.Console.Write("{0}  ", employeeName);
            }
        }
    }
```

Свойство Length

Реализация в C# массивов в виде объектов дает целый ряд преимуществ. Одно из них заключается в том, что с каждым массивом связано свойство Length, содержащее число элементов, из которых может состоять массив. Следовательно, у каждого массива имеется специальное свойство, позволяющее определить его длину.

Когда запрашивается длина многомерного массива, то возвращается общее число элементов, из которых может состоять массив. Благодаря наличию у массивов свойства Length операции с массивами во многих алгоритмах становятся более простыми, а значит, и более надежными. Давайте рассмотрим пример использования свойства Length:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] myArr = { 1, 2, 3, 4 };

            for (int i = 0; i < myArr.Length; i++)
                Console.WriteLine(myArr[i]);

            Console.ReadLine();
        }
    }
}
```

СОРТИРОВКА МАССИВА

Сортировкой массива называется перераспределение элементов массива в порядке возрастания или убывания значений элементов.

Рассмотрим три простейших метода сортировки массива, на которых основаны более сложные способы, использующиеся в специализированных программах обработки больших объёмов данных.

Метод простого выбора

Таблица 1.1

Номер прогона	Элементы массива *m*				
1	5	3	2	1	4
2	1	3	2	5	4
3	1	2	3	5	4
4	1	2	3	5	4
Результат	1	2	3	4	5

Метод простого выбора заключается в выборе наименьшего (в случае сортировки по возрастанию) или наибольшего (в случае сортировки по убыванию) значения элементов массива; затем этот элемент меняется местами с первым элементом. Далее выбирается наименьший (наибольший) элемент из оставшейся неотсортированной части массива и меняется местами со вторым элементом и т.д.

Заметим, что количество прогонов будет на единицу меньше количества элементов массива. Поэтому если количество элементов массива *m* будет *n*, то количество прогонов составит $n-1$.

Метод «пузырька»

Метод «пузырька» заключается в последовательном сравнении соседних элементов массива и переприсвоении значений этих элементов в зависимости от вида сортировки (по возрастанию или по убыванию). Таким образом, за один прогон на своё место встаёт одно значение, «всплывая», как пузырёк газа в жидкости, откуда и название метода (см. таблицу 1.2 — пример сортировки по возрастанию).

В таблице 1.2 фигурной границей показаны элементы, которые отсортированы в предыдущем прогоне и не участвуют в текущем прогоне. Прогоном будем называть цикл, в котором происходит перемещение значения на своё место в неотсортированной части массива. Полужирным шрифтом выделены сравниваемые значения. Стрелками показаны элементы, значения которых меняются местами.

Таблица 1.2.

Номер прогона	Элементы массива *m*				
1	5	2	3	1	4
	2	5	3	1	4
	2	3	5	1	4
	2	3	1	5	4
2	2	3	1	4	5
	2	3	1	4	5
	2	1	3	4	5
3	2	1	3	4	5
	1	2	3	4	5
4	1	2	3	4	5
Результат	1	2	3	4	5

Метод простых вставок (включения)

Хотя в методе «пузырька» нет лишних переприсвоений, однако на каждом прогоне сравниваются все пары элементов, что также увеличивает время сортировки. Это устраняется в *методе простых вставок*, в котором значение элемента из неотсортированной части массива, меняясь местами со значениями из отсортированной части, встаёт на своё место.

В таблице 1.3 представлена сортировка по возрастанию массива $m = (5 \quad 2 \quad 3 \quad 1 \quad 4)$ методом простых вставок.

Полужирным шрифтом выделены сравниваемые значения. Стрелками показаны элементы, значения которых меняются местами. Фигурной границей показаны элементы, значения которых не сравниваются в текущем прогоне, что сокращает время сортировки. Прогоном будем называть цикл, в котором значение из неотсортированной части занимает своё место в отсортированной части массива.

Таблица 1.3

Номер прогона	Элементы массива *m*				
1	5	2	3	1	4
2	2	5	3	1	4
	2	3	5	1	4
3	2	3	5	1	4
	2	3	1	5	4
	2	1	3	5	4
4	1	2	3	5	4
	1	2	3	4	5
Результат	1	2	3	4	5

СТРОКИ И РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

С точки зрения регулярного программирования строковый **тип данных *string*** относится к числу самых важных в C#. Этот тип определяет и поддерживает символьные строки. В целом ряде других языков программирования строка представляет собой массив символов. А в C# строки являются **объектами.**

Построение строк

Самый простой способ построить символьную строку — воспользоваться строковым литералом. Например, в следующей строке кода переменной ссылки на строку *str* присваивается ссылка на строковый литерал:

```
string str = "Пример строки";
```

В данном случае переменная *str* инициализируется последовательностью символов *"Пример строки"*. Объект типа *string* можно также создать из массива типа *char*.

Например:

```
char[] chararray = {'e', 'x', 'a', 'm', 'p', 'l', 'e'};

string str = new string(chararray);
```

Как только объект типа *string* будет создан, его можно использовать везде, где только требуется строка текста, заключенного в кавычки.

Работа со строками

В классе *System.String* предоставляется набор методов для определения длины символьных данных, поиска подстроки в текущей строке, преобразования символов из верхнего регистра в нижний и наоборот, и т.д. Далее мы рассмотрим этот класс более подробно.

Поле, индексатор и свойство класса String

В классе *String* определено единственное поле:

```
public static readonly string Empty;
```

Поле *Empty* обозначает пустую строку, т.е. такую строку, которая не содержит символы. Этим оно отличается от пустой ссылки типа *String*, которая просто делается на несуществующий объект.

Помимо этого, в классе *String* определен единственный индексатор, доступный только для чтения:

```
public char this[int index] { get; }
```

Этот индексатор позволяет получить символ по указанному индексу. Индексация строк, как и массивов, начинается с нуля. Объекты типа *String* отличаются постоянством и не изменяются, поэтому вполне логично, что в классе *String* поддерживается индексатор, доступный только для чтения.

И наконец, в классе *String* определено единственное свойство, доступное только для чтения:

```
public int Length { get; }
```

Свойство *Length* возвращает количество символов в строке.

В примере ниже показано использование индексатора и свойства *Length*:

```
using System;

class Example
{
    static void Main()
    {
        String str = "Простая строка";
        // Получить длину строки и 6й символ в строке используя индексатор
        Console.WriteLine("Длина строки - {0}, 6й символ - '{1}'", str.Length,
str[5]);
    }
}
```

Операторы класса String

В классе String перегружаются два следующих оператора: == и !=. Оператор == служит для проверки двух символьных строк на равенство. Когда оператор == применяется к ссылкам на объекты, он обычно проверяет, делаются ли обе ссылки на один и тот же объект. А когда оператор == применяется к ссылкам на объекты типа *String*, то на предмет равенства сравнивается содержимое самих строк.

Это же относится и к оператору !=. Когда он применяется к ссылкам на объекты типа *String*, то на предмет неравенства сравнивается содержимое самих строк.

В то же время другие операторы отношения, в том числе < и >=, сравнивают ссылки на объекты типа *String* таким же образом, как и на объекты других типов.

А для того чтобы проверить, является ли одна строка больше другой, следует вызвать метод ***Compare(),*** определенный в классе *String*.

Методы класса String

В следующей таблице перечислены некоторые наиболее интересные методы этого класса, сгруппированные по назначению:

Методы работы со строками

Метод	**Структура и перегруженные версии**	**Назначение**
Сравнение строк		
Compare()	*public static int Compare(string strA, string strB)* *public static int Compare(string strA, string strB, bool ignoreCase)*	Статический метод, сравнивает строку *strA* со строкой *strB*. Возвращает положительное значение, если строка *strA* больше строки *strB*; отрицательное значение, если строка *strA* меньше строки *strB*; и нуль, если строки *strA* и *strB* равны. Сравнение выполняется с учетом регистра. Если параметр *ignoreCase* принимает логическое значение *true*, то при сравнении не учитываются различия между прописным и строчным вариантами букв. В противном случае эти различия учитываются.

	public static int Compare(string strA, int indexA, string strB, int indexB, int length) *public static int Compare(string strA, int indexA, string strB, int indexB, int length, bool ignoreCase)*	Сравнивает части строк *strA* и *strB*. Сравнение начинается со строковых элементов *strA[indexA]* и *strB[indexB]* и включает количество символов, определяемых параметром *length*. Метод возвращает положительное значение, если часть строки *strA* больше части строки *strB*; отрицательное значение, если часть строки *strA* меньше части строки *strB*; и нуль, если сравниваемые части строк *strA* и *strB* равны. Сравнение выполняется с учетом.
CompareOrdinal()	*public static int CompareOrdinal(string strA, string strB)* *public static int CompareOrdinal(string strA, int indexA, string strB, int indexB, int count)*	Делает то же, что и метод Compare(), но без учета локальных установок
Конкатенация (соединение) строк		
Concat()	*public static string Concat(string str0, string str1);* *public static string Concat(params string[] values);*	Комбинирует отдельные экземпляры строк в одну строку (конкатенация)
Поиск в строке		
Contains()	*public bool Contains(string value)*	Метод, который позволяет определить, содержится ли в строке определенная подстрока (*value*)
StartsWith()	*public bool StartsWith(string value)*	Возвращает логическое значение *true*, если вызывающая строка начинается с подстроки *value*. В противном случае возвращается логическое значение *false*.
EndsWith()	*public bool EndsWith(string value)*	Возвращает логическое значение *true*, если вызывающая строка оканчивается подстрокой *value*. В противном случае возвращает логическое значение *false*.
IndexOf()	*public int IndexOf(char value)* *public int IndexOf(string value)*	Находит первое вхождение заданной подстроки или символа в строке. Если искомый символ или подстрока не обнаружены, то возвращается значение -1

	public int IndexOf(char value, int startIndex) *public int IndexOf(string value, int startIndex)* *public int IndexOf(char value, int startIndex, int count)* *public int IndexOf(string value, int startIndex, int count)*	Возвращает индекс первого вхождения символа или подстроки *value* в вызывающей строке. Поиск начинается с элемента, указываемого по индексу *startIndex*, и охватывает число элементов, определяемых параметром *count* (если указан). Метод возвращает значение -1, если искомый символ или подстрока не обнаружен
IndexOfAny()	*public int IndexOfAny(char[] anyOf)* *public int IndexOfAny(char[] anyOf, int startIndex)* *public int IndexOfAny(char[] anyOf, int startIndex, int count)*	Возвращает индекс первого вхождения любого символа из массива anyOf, обнаруженного в вызывающей строке. Поиск начинается с элемента, указываемого по индексу startIndex, и охватывает число элементов, определяемых параметром count (если они указаны). Метод возвращает значение -1, если не обнаружено совпадение ни с одним из символов из массива anyOf. Поиск осуществляется порядковым способом
LastIndexOf()	Перегруженные версии аналогичны методу *IndexOf()*	То же, что *IndexOf*, но находит последнее вхождение символа или подстроки, а не первое
Разделение и соединение строк		
Split	*public string[] Split(params char[] separator)* *public string[] Split(params char[] separator, int count)*	Метод, возвращающий массив *string* с присутствующими в данном экземпляре подстроками внутри, которые отделяются друг от друга элементами из указанного массива *char* или *string*. В первой форме метода *Split()* вызывающая строка разделяется на составные части. В итоге возвращается массив, содержащий подстроки, полученные из вызывающей строки. Символы, ограничивающие эти подстроки, передаются в массиве *separator*. Если массив *separator* пуст или ссылается на пустую строку, то в качестве разделителя подстрок используется пробел. А во второй форме данного метода возвращается количество подстрок, определяемых параметром *count*.
Join()	*public static string Join(string separator, string[] value)* *public static string Join(string separator, string[] value, int startIndex, int count)*	Строит новую строку, комбинируя содержимое массива строк. В первой форме метода *Join()* возвращается строка, состоящая из сцепляемых подстрок, передаваемых в массиве *value*. Во второй форме также возвращается строка, состоящая из подстрок, передаваемых в массиве *value*, но они сцепляются в определенном количестве *count*, начиная с элемента массива *value[startIndex]*. В обеих формах каждая последующая строка отделяется от предыдущей разделительной строкой,

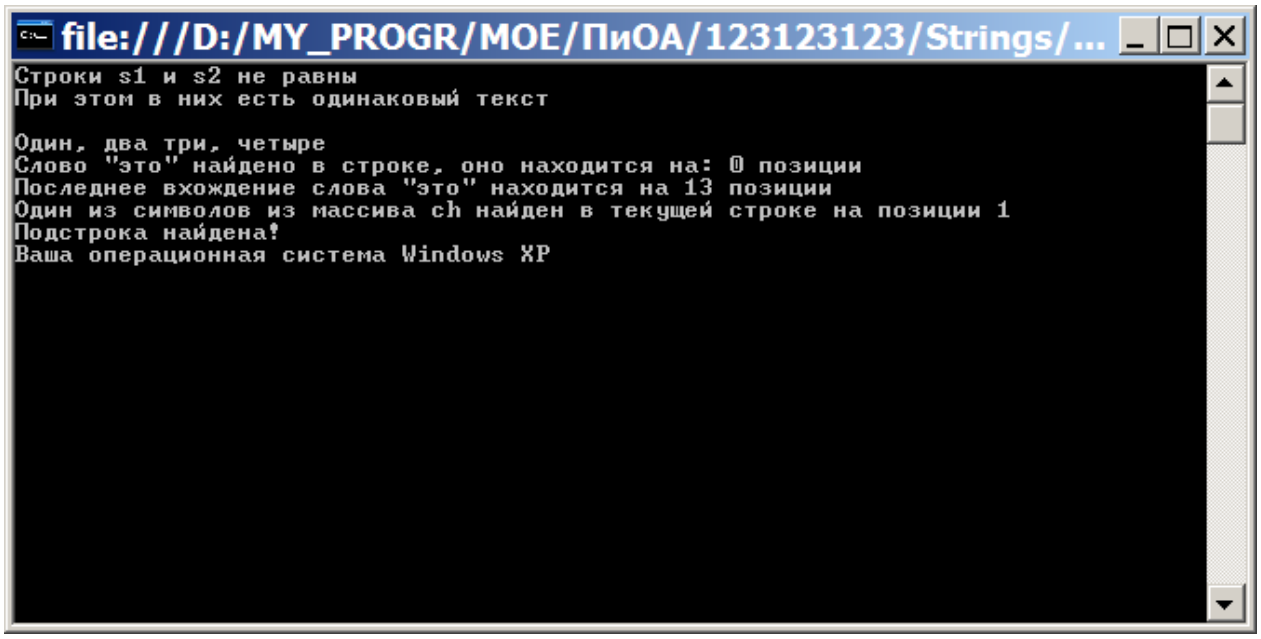
		определяемой параметром *separator*.
Заполнение и обрезка строк		
Trim()	*public string Trim()* *public string Trim(params char[] trimChars)*	Метод, который позволяет удалять все вхождения определенного набора символов с начала и конца текущей строки. В первой форме метода *Trim()* из вызывающей строки удаляются начальные и конечные пробелы. А во второй форме этого метода удаляются начальные и конечные вхождения в вызывающей строке символов из массива *trimChars*. В обеих формах возвращается получающаяся в итоге строка.
PadLeft()	*public string PadLeft(int totalWidth)* *public string PadLeft(int totalWidth, char paddingChar)*	Позволяет дополнить строку символами слева. В первой форме метода *PadLeft()* вводятся пробелы с левой стороны вызывающей строки, чтобы ее общая длина стала равной значению параметра *totalWidth*. А во второй форме данного метода символы, обозначаемые параметром *paddingChar*, вводятся с левой стороны вызывающей строки, чтобы ее общая длина стала равной значению параметра *totalWidth*. В обеих формах возвращается получающаяся в итоге строка. Если значение параметра *totalWidth* меньше длины вызывающей строки, то возвращается копия неизмененной вызывающей строки.
PadRight()	Аналогично *PadLeft()*	Позволяет дополнить строку символами справа.
Вставка, удаление и замена строк		
Insert()	*public string Insert(int startIndex, string value)*	Используется для вставки одной строки в другую, где *value* обозначает строку, вставляемую в вызывающую строку по индексу *startIndex*. Метод возвращает получившуюся в итоге строку.
Remove()	*public string Remove(int startIndex)* *public string Remove(int startIndex, int count)*	Используется для удаления части строки. В первой форме метода *Remove()* удаление выполняется, начиная с места, указываемого по индексу *startIndex*, и продолжается до конца строки. А во второй форме данного метода из строки удаляется количество символов, определяемое параметром *count*, начиная с места, указываемого по индексу *startIndex*.
Replace()	*public string Replace(char oldChar, char newChar)* *public string Replace(string oldValue, string newValue)*	Используется для замены части строки. В первой форме метода *Replace()* все вхождения символа *oldChar* в вызывающей строке заменяются символом *newChar*. А во второй форме данного метода все вхождения строки *oldValue* в вызывающей строке заменяются строкой *newValue*.
Смена регистра		

ToUpper()	*public string ToUpper()*	Делает заглавными все буквы в вызывающей строке.
ToLower()	*public string ToLower()*	Делает строчными все буквы в вызывающей строке.
Получение подстроки из строки		
Substring()	*public string Substring(int startIndex)* *public string Substring(int startIndex, int length)*	В первой форме метода *Substring()* подстрока извлекается, начиная с места, обозначаемого параметром *startIndex*, и до конца вызывающей строки. А во второй форме данного метода извлекается подстрока, состоящая из количества символов, определяемых параметром *length*, начиная с места, обозначаемого параметром *startIndex*.

Пример следующей программы использует несколько из вышеуказанных методов:

```
static void Main(string[] args)
      {
        // Сравним первые две строки
        string s1 = "это строка";
        string s2 = "это текст, а это строка";

        if (String.CompareOrdinal(s1, s2) != 0)
           Console.WriteLine("Строки s1 и s2 не равны");

        if (String.Compare(s1, 0, s2, 13, 10, true) == 0)
           Console.WriteLine("При этом в них есть одинаковый текст");

        // Конкатенация строк
        Console.WriteLine(String.Concat("\n" + "Один, два ","три, четыре"));

        // Поиск в строке
         // Первое вхождение подстроки
         if (s2.IndexOf("это") != -1)
           Console.WriteLine("Слово \"это\" найдено в строке, оно "+
            "находится на: {0} позиции", s2.IndexOf("это"));

         // Последнее вхождение подстроки
         if (s2.LastIndexOf("это") != -1)
            Console.WriteLine("Последнее вхождение слова \"это\" находится "
               + "на {0} позиции", s2.LastIndexOf("это"));

         // Поиск из массива символов
         char[] myCh = {'Ы','х','т'};
         if (s2.IndexOfAny(myCh) != -1)
            Console.WriteLine("Один из символов из массива ch "+
               "найден в текущей строке на позиции {0}", s2.IndexOfAny(myCh));

         // Определяем начинается ли строка с заданной подстроки
         if (s2.StartsWith("это текст") == true)
            Console.WriteLine("Подстрока найдена!");

         // Определяем содержится ли в строке подстрока
         // на примере определения ОС пользователя
         string myOS = Environment.OSVersion.ToString();
         if (myOS.Contains("NT 5.1"))
            Console.WriteLine("Ваша операционная система Windows XP");
         else if (myOS.Contains("NT 6.1"))
           Console.WriteLine("Ваша операционная система Windows 7");
```

```
            Console.ReadLine();
        }

    }
```



Форматирующие строки

Если необходимо, чтобы разрабатываемые программы были дружественными к пользователю, они должны предлагать средства для отображения своих строковых представлений в любом из форматов, которые могут понадобиться пользователю.

Начнем с рассмотрения того, что происходит, когда форматная строка применяется к примитивному типу, а отсюда станет ясно, как следует включать спецификаторы формата для пользовательских классов и структур:

```
decimal d = 12.05;
int i = 5;

Console.WriteLine("Значение переменной d = {0:C}, a i = {1}",d,i);
```

Сама строка формата содержит большую часть отображаемого текста, но всякий раз, когда в нее должно быть вставлено значение переменной, в фигурных скобках указывается индекс. В фигурные скобки может быть включена и другая информация, относящаяся к формату данного элемента, например, та, что описана ниже.

Спецификатор формата предваряется двоеточием. Это указывает, каким образом необходимо отформатировать элемент. Например, можно указать, должно ли число быть форматировано как денежное значение, либо его следует отобразить в научной нотации.

В следующей таблице перечислены часто используемые спецификаторы формата для числовых типов.

Стоит отметить, что полный список спецификаторов формата значительно длиннее, поскольку другие типы данных добавляют собственные спецификаторы.

Спецификаторы формата для чисел

Спецификатор	**Применяется к**	**Значение**	**Пример**
C	Числовым типам	Символ местной валюты	$835.50 (США) £835.50 (Великобритания) 835.50р.(Россия)
D	Только к целочисленным типам	Обычное целое	835
E	Числовым типам	Экспоненциальная нотация	8.35E+002
F	Числовым типам	С фиксированной десятичной точкой	835.50
G	Числовым типам	Обычные числа	835.5
N	Числовым типам	Формат чисел, принятый в данной местности	4,384.50 (Великобритания/США) 4 384,50 (континентальная Европа)
P	Числовым типам	Процентная нотация	835,000.00%
X	Только к целочисленным типам	Шестнадцатеричный формат	1a1f

Форматирование даты и времени

Помимо числовых значений, форматированию нередко подлежит и другой тип данных: *DateTime*. Это структура, представляющая дату и время. Значения даты и времени могут отображаться самыми разными способами.

Форматирование даты и времени осуществляется с помощью спецификаторов формата. Конкретное представление даты и времени может отличаться в силу региональных и языковых особенностей и поэтому зависит от настройки параметров культурной среды. Спецификаторы формата даты и времени сведены в следующей таблице:

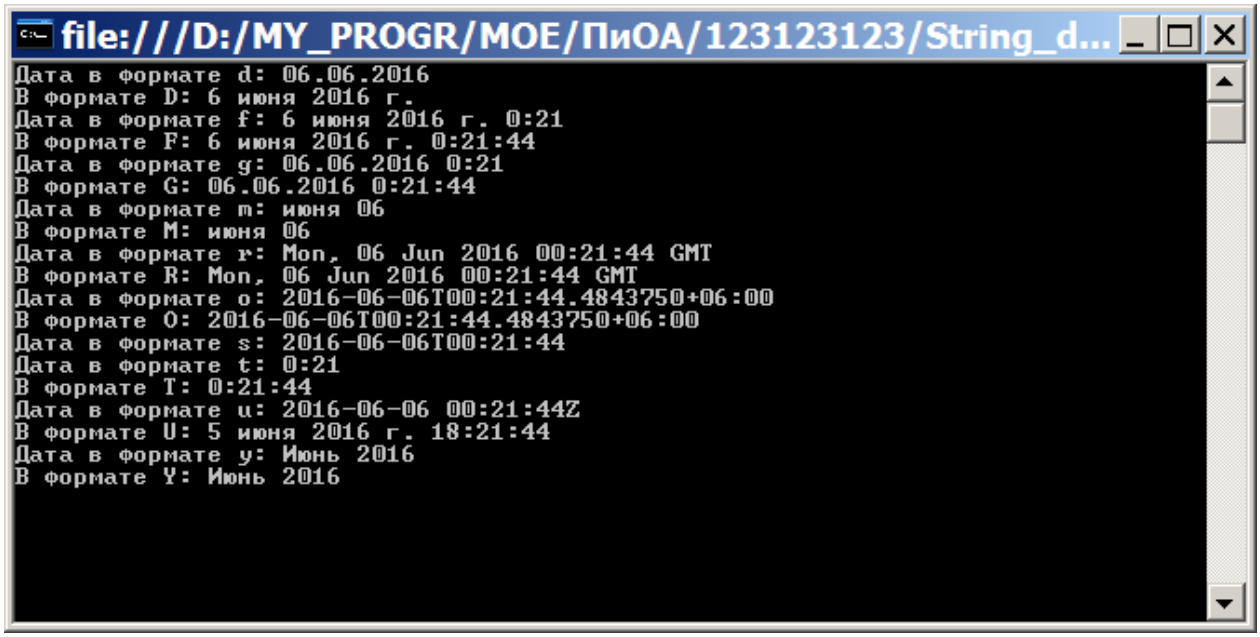
Спецификаторы формата для дат

Спецификатор	Формат
D	Дата в длинной форме
d	Дата в краткой форме
F	Дата и время в длинной форме
f	Дата и время в краткой форме
G	Дата — в краткой форме, время — в длинной
g	Дата и время — в краткой форме
M	Месяц и день
m	То же, что и M
O	Формат даты и времени, включая часовой пояс. Строка, составленная в формате O, может быть преобразована обратно в эквивалентную форму вывода даты и времени. Это так называемый "круговой" формат
R	Дата и время в стандартной форме по Гринвичу
s	Сортируемый формат представления даты и времени
T	Время в длинной форме
t	Время в краткой форме
U	Длинная форма универсального представления даты и времени; время отображается как универсальное синхронизированное время (UTC)
u	Краткая форма универсального представления даты и времени
Y	Месяц и год

В приведенном ниже примере программы демонстрируется применение спецификаторов формата даты и времени:

```
static void Main(string[] args)
      {
        DateTime myDate = DateTime.Now;
        Console.WriteLine("Дата в формате d: {0:d}\nВ формате D: {0:D}",myDate);
```

```
            Console.WriteLine("Дата в формате f: {0:f}\nВ формате F: {0:F}", myDate);
            Console.WriteLine("Дата в формате g: {0:g}\nВ формате G: {0:G}", myDate);
            Console.WriteLine("Дата в формате m: {0:m}\nВ формате M: {0:M}", myDate);
            Console.WriteLine("Дата в формате r: {0:r}\nВ формате R: {0:R}", myDate);
            Console.WriteLine("Дата в формате o: {0:o}\nВ формате O: {0:O}", myDate);
            Console.WriteLine("Дата в формате s: {0:s}", myDate);
            Console.WriteLine("Дата в формате t: {0:t}\nВ формате T: {0:T}", myDate);
            Console.WriteLine("Дата в формате u: {0:u}\nВ формате U: {0:U}", myDate);
            Console.WriteLine("Дата в формате y: {0:y}\nВ формате Y: {0:Y}", myDate);

            Console.ReadLine();
        }
    }
```



Регулярные выражения в C#

Регулярные выражения — это часть небольшой технологической области, невероятно широко используемой в огромном диапазоне программ. Регулярные выражения можно представить себе как мини-язык программирования, имеющий одно специфическое назначение: находить подстроки в больших строковых выражениях.

Это не новая технология, изначально она появилась в среде **UNIX** и обычно используется в языке программирования Perl. Разработчики из Microsoft перенесли ее в Windows, где до недавнего времени эта технология применялась в основном со сценарными языками. Однако теперь регулярные выражения поддерживаются множеством классов .NET из пространства имен *System.Text.RegularExpressions*. Случаи применения регулярных выражений можно встретить во многих частях среды .NET Framework. В частности, вы найдете их в серверных элементах управления проверкой ASP.NET.

Введение в регулярные выражения

Язык регулярных выражений предназначен специально для обработки строк. Он включает два средства:

- Набор управляющих кодов для идентификации специфических типов символов
- Система для группирования частей подстрок и промежуточных результатов таких действий

С помощью регулярных выражений можно выполнять достаточно сложные и высокоуровневые действия над строками:

- Идентифицировать (и возможно, помечать к удалению) все повторяющиеся слова в строке
- Сделать заглавными первые буквы всех слов
- Преобразовать первые буквы всех слов длиннее трех символов в заглавные
- Выделить различные элементы в URL (например, имея http://www.professorweb.ru, выделить протокол, имя компьютера, имя файла и т.д.)

Главным преимуществом регулярных выражений является использование ***метасимволов*** — специальные символы, задающие команды, а также управляющие последовательности, которые работают подобно управляющим последовательностям C#. Это символы, предваренные знаком обратного слеша (\) и имеющие специальное назначение.

В следующей таблице специальные метасимволы регулярных выражений C# сгруппированы по смыслу:

Метасимволы, используемые в регулярных выражениях C#

Символ	**Значение**	**Пример**	**Соответствует**
Классы символов			
[...]	Любой из символов, указанных в скобках	[a-z]	В исходной строке может быть любой символ английского алфавита в нижнем регистре
[^...]	Любой из символов, не указанных в скобках	[^0-9]	В исходной строке может быть любой символ кроме цифр
.	Любой символ, кроме перевода строки или другого разделителя Unicode-строки		
\w	Любой текстовый символ, не являющийся пробелом, символом табуляции и т.п.		
\W	Любой символ, не являющийся текстовым символом		
\d	Любые ASCII-цифры. Эквивалентно [0-9]		

\D	Любой символ, отличный от ASCII-цифр. Эквивалентно [^0-9]		
Символы повторения			
{n,m}	Соответствует предшествующему шаблону, повторенному не менее n и не более m раз	s{2,4}	"Press", "ssl", "progressss"
{n,}	Соответствует предшествующему шаблону, повторенному n или более раз	s{1,}	"ssl"
{n}	Соответствует в точности n экземплярам предшествующего шаблона	s{2}	"Press", "ssl", но не "progressss"
?	Соответствует нулю или одному экземпляру предшествующего шаблона; предшествующий шаблон является необязательным	Эквивалентно {0,1}	
+	Соответствует одному или более экземплярам предшествующего шаблона	Эквивалентно {1,}	
*	Соответствует нулю или более экземплярам предшествующего шаблона	Эквивалентно {0,}	
Символы регулярных выражений выбора			
|	Соответствует либо подвыражению слева, либо подвыражению справа (аналог логической операции ИЛИ).		
(...)	Группировка. Группирует элементы в единое целое, которое может использоваться с символами *, +, ?, | и т.п. Также запоминает символы, соответствующие этой группе для использования в последующих ссылках.		
Якорные символы регулярных выражений			
^	Соответствует началу строкового выражения или началу строки при многострочном поиске.	^Hello	"Hello, world", но не "Ok, Hello world" т.к. в этой строке слово "Hello" находится не в начале
$	Соответствует концу строкового выражения или концу строки при многострочном поиске.	Hello$	"World, Hello"
\b	Соответствует границе слова, т.е. соответствует позиции между символом \w и	\b(my)\b	В строке "Hello my world" выберет слово "my"

	символом \W или между символом \w и началом или концом строки.		
\B	Соответствует позиции, не являющейся границей слов.	\B(ld)\b	Соответствие найдется в слове "World", но не в слове "ld"

Использование регулярных выражений в C#

Безусловно, задачу поиска и замены подстроки в строке можно решить на C# с использованием различных методов *System.String*. Однако в некоторых случаях это потребует написания большого объема кода C#. Если будем использовать регулярные выражения, то весь этот код сокращается буквально до нескольких строк.

По сути, создается экземпляр объекта ***RegEx***, передается ему строка для обработки, а также само регулярное выражение (строку, включающую инструкции на языке регулярных выражений).

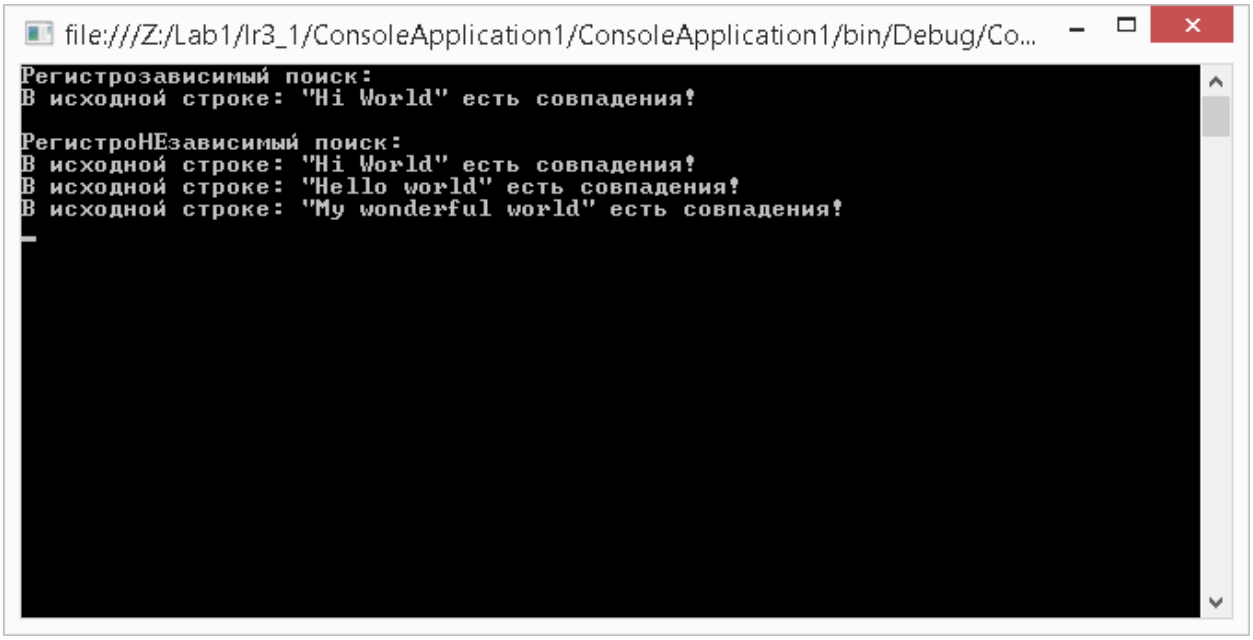
В следующей таблице показана часть информации о перечислении *RegexOptions*, экземпляр которого можно передать конструктору класса *RegEx:*

Структура перечисления RegexOptions

Член	**Описание**
CultureInvariant	Предписывает игнорировать национальные установки строки
ExplicitCapture	Модифицирует способ поиска соответствия, обеспечивая только буквальное соответствие
IgnoreCase	Игнорирует регистр символов во входной строке
IgnorePatternWhitespace	Удаляет из строки не защищенные управляющими символами пробелы и разрешает комментарии, начинающиеся со знака фунта или хеша
Multiline	Изменяет значение символов ^ и $ так, что они применяются к началу и концу каждой строки, а не только к началу и концу всего входного текста
RightToLeft	Предписывает читать входную строку справа налево вместо направления по умолчанию — слева направо (что удобно для некоторых азиатских и других языков, которые читаются в таком направлении)
Singleline	Специфицирует однострочный режим, в котором точка (.) символизирует соответствие любому символу

После создания шаблона регулярного выражения с ним можно осуществить различные действия, в зависимости от того, что необходимо. Можно просто проверить, существует ли текст, соответствующий шаблону, в исходной строке. Для этого нужно использовать метод ***IsMatch()***, который возвращает логическое значение:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Text.RegularExpressions;

namespace ConsoleApplication1
{
    class Program
    {


static void Main(string[] args)
        {
            // Массив тестируемых строк
            string[] test = {
            "Hi World", "Hello world", "My wonderful world"
        };

            // Проверим, содержится ли в исходных строках слово World
            // при этом мы не укажем опции RegexOption
            Regex regex = new Regex("World");

            Console.WriteLine("Регистрозависимый поиск: ");
            foreach (string str in test)
            {
                if (regex.IsMatch(str))
                    Console.WriteLine("В исходной строке: \"{0}\" есть
совпадения!", str);
            }
            Console.WriteLine();

            // Теперь укажем поиск, не зависимый от регистра
            regex = new Regex("World", RegexOptions.IgnoreCase);

            Console.WriteLine("РегистроНЕзависимый поиск: ");
            foreach (string str in test)
            {
                if (regex.IsMatch(str))
                    Console.WriteLine("В исходной строке: \"{0}\" есть
совпадения!", str);
            }

            Console.ReadKey();
        }

        }
}
```



Если необходимо вернуть найденное соответствие из исходной строки, то можно воспользоваться методом ***Match()***, который возвращает объект класса *Match*, содержащий сведения о первой подстроке, которая сопоставлена шаблону регулярного выражения. В этом классе имеется свойство *Success*, которое возвращает значение *true*, если найдено следующее совпадение, которое можно получить с помощью вызова метода *Match.NextMatch()*. Эти вызовы метода можно продолжать пока свойство *Match.Success* не вернет значение *false*.

Например:

```
static void Main(string[] args)
        {

                // Допустим в исходной строке нужно найти все числа,
                // соответствующие стоимости продукта
                string input = "Добро пожаловать в наш магазин, вот наши цены: "
+
                    "1 кг. яблок - 20 руб. " +
                    "2 кг. апельсинов - 30 руб. " +
                    "0.5 кг. орехов - 50 руб.";

                string pattern = @"\b(\d+\W?руб)";
                Regex regex = new Regex(pattern);

                // Получаем совпадения в экземпляре класса Match
                Match match = regex.Match(input);

                // отображаем все совпадения
                while (match.Success)
                {
                    // Т.к. мы выделили в шаблоне одну группу (одни круглые
скобки),
                    // ссылаемся на найденное значение через свойство Groups
класса Match
                    Console.WriteLine(match.Groups[1].Value);

                    // Переходим к следующему совпадению
                    match = match.NextMatch();
                }
                Console.ReadKey();
        }
```

Извлечь все совпадения можно и более простым способом, используя метод ***Regex.Matches().*** который возвращает объект класса *MatchCollection*, который, в свою очередь, содержит сведения обо всех совпадениях, которые обработчик регулярных выражений находит во входной строке. Например, предыдущий пример может быть переписан для вызова метода *Matches* вместо метода *Match* и метода *NextMatch*:

```
static void Main(string[] args)
        {
            // Допустим в исходной строке нужно найти все числа,
            // соответствующие стоимости продукта
            string input = "Добро пожаловать в наш магазин, вот наши цены: " +
                "1 кг. яблок - 20 руб. " +
                "2 кг. апельсинов - 30 руб. " +
                "0.5 кг. орехов - 50 руб.";

            string pattern = @"\b(\d+\W?руб)";
            Regex regex = new Regex(pattern);

            // Достигаем того же результата что и в предыдущем примере,
            // используя метод Regex.Matches() возвращающий MatchCollection
            foreach (Match match in regex.Matches(input))
            {
                Console.WriteLine(match.Groups[1].Value);
            }
            Console.ReadKey();
        }

    }
```

Наконец, можно не просто извлекать совпадения в исходной строке, но и заменять их на собственные значения. Для этого используется метод ***Regex.Replace().*** В качестве замены методу *Replace()* можно передавать как строку, так и шаблон замены. В следующей таблице показано как формируются метасимволы для замены:

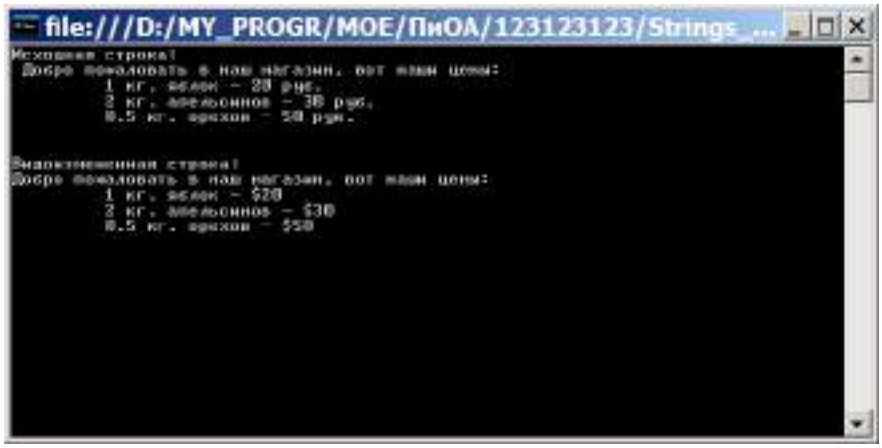
Метасимволы замены в регулярных выражениях C#

Символ	**Описание**	**Пример шаблона**	**Пример шаблона замены**	**Результат (входная -> результирующая строки)**
$ number	Замещает часть строки, соответствующую группе number	\b(\w+)(\s)(\w+)\b	$3$2$1	"один два" -> "два один"
$$	Подставляет литерал "$"	\b(\d+)\s?USD	$$$1	"103 USD" -> "$103"
$&	Замещает копией полного соответствия	(\$*(\d*(\.+\d+)?){1})	**$&	"$1.30" -> "**$1.30**"
$`	Замещает весь текст входной строки до соответствия	B+	$`	"AABBCC" -> "AAAACC"
$'	Замещает весь текст входной строки после соответствия	B+	$'	"AABBCC" -> "AACCCC"
$+	Замещает последнюю захваченную группу	B+(C+)	$+	"AABBCCDD" -> "AACCDD"
$_	Замещает всю входную строку	B+	$_	"AABBCC" -> "AAAABBCCCC"

Давайте рассмотрим метод *Regex.Replace()* на примере:

```
  static void Main(string[] args)
        {

            {
                // Допустим в исходной строке нужно заменить "руб." на "$",
                // а стоимость переместить после знака $
                string input = "Добро пожаловать в наш магазин, вот наши цены:
\n" +
                    "\t 1 кг. яблок - 20 руб. \n" +
                    "\t 2 кг. апельсинов - 30 руб. \n" +
                    "\t 0.5 кг. орехов - 50 руб. \n";

                Console.WriteLine("Исходная строка:\n {0}", input);

                // В шаблоне используются 2 группы
```

```
            string pattern = @"\b(\d+)\W?(руб.)";

            // Строка замены "руб." на "$"
            string replacement1 = "$$$1";     // Перед первой группой
ставится знак $,
            // вторая группа удаляется без замены

            input = Regex.Replace(input, pattern, replacement1);
            Console.WriteLine("\nВидоизмененная строка: \n" + input);
        }
        Console.ReadKey();
    }
```

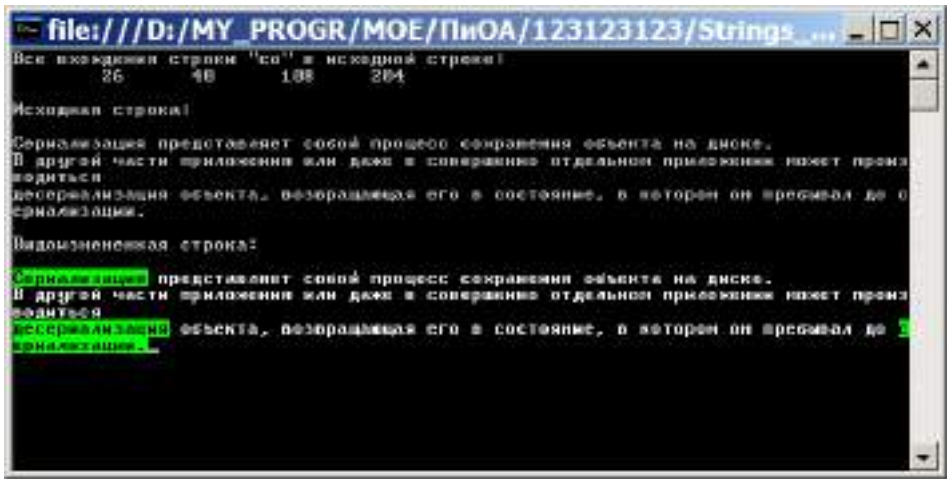


Для закрепления темы давайте рассмотрим еще один пример использования регулярных выражений, где будем искать в исходном тексте слово «сериализация» и его однокоренные слова, при этом выделяя в консоли их другим цветом:

```
static void Main(string[] args)
        {
            string myText = @"Сериализация представляет собой процесс сохранения
объекта на диске.
В другой части приложения или даже в совершенно отдельном приложении может
производиться
десериализация объекта, возвращающая его в состояние, в котором он пребывал до
сериализации.";

            const string myReg = "со";
            MatchCollection myMatch = Regex.Matches(myText,myReg);

            Console.WriteLine("Все вхождения строки \"{0}\" в исходной строке:
",myReg);
            foreach (Match i in myMatch)
                Console.Write("\t"+i.Index);

            // Усложним шаблон регулярного выражения
            // введя в него специальные метасимволы

            const string myReg1 = @"\b[с,д]\S*ериализац\S*";
            MatchCollection match1 =
Regex.Matches(myText,myReg1,RegexOptions.IgnoreCase);
            findMyText(myText,match1);

            Console.ReadLine();
        }

        static void findMyText(string text, MatchCollection myMatch)
```

```
        {
            Console.WriteLine("\n\nИсходная строка:\n\n{0}\n\nВидоизмененная
строка:\n",text);

            // Реализуем выделение ключевых слов в консоли другим цветом
            for (int i = 0; i < text.Length; i++)
            {
                foreach (Match m in myMatch)
                {
                    if ((i >= m.Index) && (i < m.Index+m.Length))
                    {
                        Console.BackgroundColor = ConsoleColor.Green;
                        Console.ForegroundColor = ConsoleColor.Black;
                        break;
                    }
                    else
                    {
                        Console.BackgroundColor = ConsoleColor.Black;
                        Console.ForegroundColor = ConsoleColor.White;
                    }
                }
                Console.Write(text[i]);
            }
           Console.ReadKey();
        }
    }
```

Результаты работы данной программы:



Для проверки гибкости работы регулярных выражений, подставьте в исходный текст еще несколько слов «сериализация», и увидите, что они будут автоматически выделены зеленым цветом в консоли

Собака означает, что в всё что в кавычках форматированию не подлежит, то есть использовать "как есть"

Например:

С @:

"Kuka\nreku" =Kuka\nreku

Без:

"Kuka\nreku"= Kuka

reku

Лекция 5
СТРУКТУРЫ

Кроме базовых простых типов данных в C# имеется и составной тип данных, который называется структурой. Структуры могут содержать в себе обычные переменные и методы.

Для примера создадим структуру *Book*, в которой будут храниться переменные для названия, автора и года издания книги. Кроме того, структура будет содержать метод для вывода информации о книге на консоль:

```
struct Book
{
    public string name;
    public string author;
    public int year;

    public void Info()
    {
        Console.WriteLine("Книга '{0}' (автор {1}) была издана в {2} году",
name, author, year);
    }
}
```

Чтобы можно было использовать переменные и методы структуры из любого места программы необходимо ставить перед переменными и методом модификатор доступа *public.*

Покажем пример использования структуры на практике:

```
using System;

namespace Structures
{
    class Program
    {
        static void Main(string[] args)
        {
            Book book;
            book.name = "Война и мир";
            book.author = "Л. Н. Толстой";
            book.year = 1869;

            //Выведем информацию о книге book на экран
            book.Info();

            Console.ReadLine();
        }
    }

    struct Book
    {
        public string name;
        public string author;
        public int year;

        public void Info()
        {
            Console.WriteLine("Книга '{0}' (автор {1}) была издана в {2} году",
name, author, year);
        }
    }
}
```

Структуру можно задать как внутри пространства имен (как в данном случае), так и внутри класса, но не внутри метода.

По сути структура *Book* представляет новый тип данных. Можно также использовать массив структур:

```
Book[] books=new Book[3];

books[0].name = "Война и мир";
books[0].author = "Л. Н. Толстой";
books[0].year = 1869;

books[1].name = "Преступление и наказание";
books[1].author = "Ф. М. Достоевский";
books[1].year = 1866;

books[2].name = "Отцы и дети";
books[2].author = "И. С. Тургенев";
books[2].year = 1862;

foreach (Book b in books)
{
    b.Info();
}
```

Конструкторы в структурах

Кроме обычных методов структура может содержать специальный метод - конструктор, который выполняет некую начальную инициализацию объекта, например, присваивает всем полям некоторые значения по умолчанию. В принципе для структуры необязательно использовать конструктор:

```
Book book1;
```

Однако в этом случае, чтобы использовать структуру, необходимо будет первоначально проинициализировать все ее поля:

```
book1.name = "Война и мир";
book1.author = "Л. Н. Толстой";
book1.year = 1869;
```

Вызов же конструктора по умолчанию позволяет автоматически проинициализировать поля структуры значениями по умолчанию (например, для числовых данных - это число 0):

```
Book book2 = new Book(); // использование конструктора
```

Вызов конструктора имеет следующий синтаксис:
new название_структуры ([список_параметров]).

Теперь попробуем определить свой конструктор:

```
struct Book
{
    public string name;
    public string author;
    public int year;

    // конструкор
    public Book(string n, string a, int y)
    {
        name = n;
        author = a;
        year = y;
```

```
    }
    public void Info()
    {
        Console.WriteLine("Книга '{0}' (автор {1}) была издана в {2} году",
name, author, year);
    }
}
```

Конструктор по сути является обычным методом, только не имеет возвращаемого значения, и его название всегда совпадает с именем структуры или класса. Теперь используем его:

```
Book book = new Book("Война и мир", "Л. Н. Толстой", 1869);
book.Info();
```

Теперь нет необходимости вручную присваивать значения полям структуры - их инициализацию выполнил конструктор.

Типы значений и ссылочные типы

Ранее были рассмотрены следующие элементарные типы данных: *int, byte, double, string* и др. Также есть сложные типы: структуры, классы и др.

Все эти типы данных можно разделить на:

- типы значений, еще называемые значимыми типами, (value types) и
- ссылочные типы (reference types).

Важно понимать между ними различия.

Примеры типов значений:

- Целочисленные типы (byte, sbyte, char, short, ushort, int, uint, long, ulong)
- Типы с плавающей запятой (float, double)
- Тип decimal
- Тип bool
- Структуры (struct)
- Примеры ссылочных типов:
- Тип object
- Тип string
- Классы (class)

В чем же между ними различия? Для этого надо понять организацию памяти в .NET. Здесь память делится на два типа: стек и куча (heap).

Все типы значений являются производными от типа *System.ValueType* и размещают свое значение в стеке. Стек представляет собой структуру данных, которая растет снизу-вверх: каждый новый добавляемый элемент помещаются поверх предыдущего. Время жизни переменных таких типов ограничено их контекстом. Физически стек - это некоторая область памяти в адресном пространстве.

Когда программа только запускается на выполнение, в конце блока памяти, зарезервированного для стека, устанавливается указатель стека. При помещении данных в стек, указатель переустанавливается таким образом, что снова указывает на новое свободное место.

Например:

```
private void SomeMethodVal(int t)
{
    int x = 5;
    int y = 6;
    int z = x * y * t;
}
```

При вызове этого метода в стек будут помещаться переменные *t, x, y* и *z*. Они определяются в контексте данного метода. Когда метод отработает, все эти переменные уничтожаются, и память очищается.

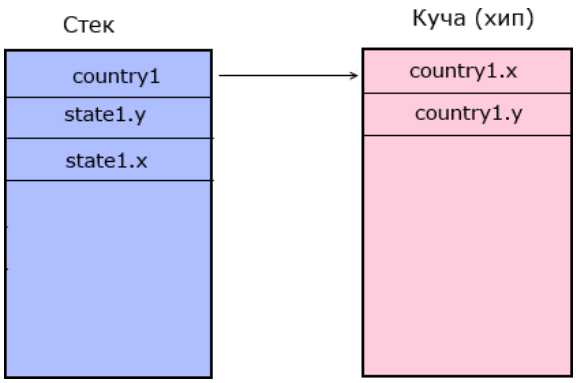
Ссылочные типы хранятся в куче, которую можно представить как неупорядоченный набор разнородных объектов. Физически это остальная часть памяти, которая доступна процессу.

При создании объекта ссылочного типа в стеке помещается ссылка на адрес в куче. Когда объект ссылочного типа перестает использоваться, то ссылка из стека удаляется, и память очищается. После этого в дело вступает автоматический сборщик мусора: он видит, что на объект в куче нет больше ссылок, и удаляет этот объект и очищает память.

Чтобы разобраться, создадим небольшой пример, где у нас будет сразу и тип значений в виде структуры и ссылочный тип в виде класса.

```
class Program
{
    private static void Main(string[] args)
    {
        State state1 = new State();

// State - структура, ее данные размещены в стеке

        Country country1 = new Country();

// Country - класс, в стек помещается ссылка на адрес в куче
// а в куче располагаются все данные объекта country1

    }

}

struct State
{
    public int x;
    public int y;
}
class Country
{
    public int x;
    public int y;
}
```

Здесь в методе *Main* в стеке выделяется память для объекта *state1*. Далее в стеке создается ссылка для объекта *country1* (*Country country1*), а с помощью конструктора выделяется место в куче (*new Country()*). Ссылка в стеке для объекта *country1* будет представлять адрес на место в куче, по которому размещен данный объект.



Таким образом, в стеке окажутся все поля структуры *state1* и адрес на все поля объекта *country1* в куче.

Копирование значений

Тип данных надо учитывать при копировании значений. При присвоении данных объекту значимого типа он получает копию данных. При присвоении данных объекту ссылочного типа происходит присвоение не копии объекта, а его адреса.

Например:

```
private static void Main(string[] args)
{
    State state1 = new State(); // Структура State
    State state2 = new State();
    state2.x = 1;
    state2.y = 2;
    state1 = state2;
    state2.x = 5; // state1.x=1 по-прежнему
    Console.WriteLine(state1.x); // 1
    Console.WriteLine(state2.x); // 5

    Country country1 = new Country(); // Класс Country
    Country country2 = new Country();
    country2.x = 1;
    country2.y = 4;
    country1 = country2;
    country2.x = 7; // теперь и country1.x = 7, так как обе ссылки и country1 и
country2
                    // указывают на один объект в куче
    Console.WriteLine(country1.x); // 7
    Console.WriteLine(country2.x); // 7

    Console.Read();
}
```

Так как *state1* - структура, то при присвоении *state1 = state2* она получает копию структуры *state2*. А объект класса *country1* при присвоении *country1 = country2*; получает адрес ссылки на тот же объект, на который указывает *country2*. Поэтому с изменением *country2*, так же будет меняться и *country1*.

Обработка исключений

Иногда при выполнении программы возникают ошибки, которые трудно предусмотреть или предвидеть, а иногда и вовсе невозможно. Например, при передачи файла по сети может неожиданно оборваться сетевое подключение. такие ситуации называются исключениями.

Язык C# предоставляет разработчикам возможности для обработки таких ситуаций. Для этого в C# предназначена конструкция try...catch...finally. При возникновении исключения ищется блок catch, который может обработать данное исключение. Если такого блока не найдено, то пользователю отображается сообщение о необработанном исключении, а дальнейшее выполнение программы останавливается. И чтобы подобной остановки не произошло, и надо использовать блок try..catch. Например:

```
static void Main(string[] args)
{
    int[] a = new int[4];

    try
    {
        a[5] = 4; // тут возникнет исключение, так как у нас в массиве только 4
элемента
        Console.WriteLine("Завершение блока try");
    }

    catch (Exception ex)
    {
        Console.WriteLine("Ошибка: " + ex.Message);
    }

    finally
    {
        Console.WriteLine("Блок finally");
    }
    Console.ReadLine();
}
```

При использовании блока try...catch..finally вначале выполняются все инструкции между операторами try и catch. Если между этими операторами вдруг возникает исключение, то обычный порядок выполнения останавливается и переходит к инструкции catch. В данном случае явно возникнет исключение в блоке try, так как пытаемся присвоить значение шестому элементу массива в то время, как в массиве всего 4 элемента. И дойдя до строки a[5] = 4;, выполнение программы остановится и перейдет к блоку catch

Инструкция catch имеет следующий синтаксис:
catch (тип_исключения имя_переменной).

В нашем примере объявляется переменная *ex*, которая имеет тип *Exception*. Но если возникшее исключение не является исключением типа, указанного в инструкции catch, то оно не обрабатывается, а программа просто зависает или выбрасывает сообщение об ошибке.

Однако так как тип *Exception* является базовым классом для всех исключений, то выражение *catch (Exception ex)* будет обрабатывать практически все исключения. Вся обработка исключения в нашем случае сводится к выводу на консоль сообщении об исключении, которое имеется в свойстве *message* класса *Exception*.

Далее в любом случае выполняется блок finally. Однако этот блок необязательный, и его можно при обработке исключений опускать. Если же в ходе программы исключений не возникнет, то программа не будет выполнять блок catch, сразу перейдет к блоку finally, если он имеется.

Обработка нескольких исключений

При необходимости можно разграничить обработку различных типов исключений, включив дополнительные блоки catch:

```
static void Main(string[] args)
{
    try
    {

    }
    catch (FileNotFoundException e)
    {
        // Обработка исключения, возникшего при отсутствии файла
    }
    catch (IOException e)
    {
        // Обработка исключений ввода-вывода
    }
    Console.ReadLine();
}
```

Если возникает исключение определенного типа, то оно переходит к соответствующему блоку catch.

При этом более частные исключения следует помещать в начале, и только потом более общие классы исключений. Например, сначала обрабатывается исключение *IOException*, и только потом *Exception* (так как *IOException* наследуется от класса *Exception*).

Обработка исключений и условные конструкции

Ряд исключительных ситуаций может быть предвиден разработчиком. Например, пусть программа предусматривает ввод числа и вывод его квадрата:

```
static void Main(string[] args)
{
    Console.WriteLine("Введите число");
    int x = Int32.Parse(Console.ReadLine());

    x *= x;
    Console.WriteLine("Квадрат числа: " + x);
    Console.Read();
}
```

Если пользователь введет не число, а строку, какие-то другие символы, то программа выпадет в ошибку. С одной стороны, здесь как раз та ситуация, когда можно применить блок try..catch, чтобы обработать возможную ошибку. Однако гораздо оптимальнее было бы проверить допустимость преобразования:

```
static void Main(string[] args)
{
    Console.WriteLine("Введите число");
    int x;
    string input = Console.ReadLine();
    if (Int32.TryParse(input, out x))
    {
        x *= x;
        Console.WriteLine("Квадрат числа: " + x);
    }
    else
    {
        Console.WriteLine("Некорректный ввод");
    }
    Console.Read();
}
```

Метод *Int32.TryParse()* возвращает *true*, если преобразование можно осуществить, и *false* - если нельзя. При допустимости преобразования переменная *x* будет содержать введенное число. Так, не используя try...catch можно обработать возможную исключительную ситуацию.

С точки зрения производительности использование блоков try..catch более накладно, чем применение условных конструкций. Поэтому по возможности вместо try..catch лучше использовать условные конструкции на проверку исключительных ситуаций.

Лекция 6
РАБОТА С ФАЙЛАМИ

Работа с потоками и файловой системой

Большинство задач в программировании так или иначе связаны с работой с файлами и каталогами. При создании программного приложения может потребоваться прочитать текст из файла или наоборот произвести запись, удалить файл или целый каталог.

Фреймворк .NET предоставляет большие возможности по управлению и манипуляции файлами и каталогами, которые по большей части сосредоточены в пространстве имен *System.IO*. Классы, расположенные в этом пространстве имен (такие как *Stream*, *StreamWriter*, *FileStream* и др.), позволяют управлять файловым вводом-выводом.

Файл – это набор данных, который хранится на внешнем запоминающем устройстве (например, на жестком диске). Файл имеет имя и расширение. Расширение позволяет идентифицировать, какие данные и в каком формате хранятся в файле.

Под работой с файлами подразумевается:

- создание файлов;
- удаление файлов;
- чтение данных;
- запись данных;
- изменение параметров файла (имя, расширение…);
- другое.

В Си# есть пространство имен *System.IO*, в котором реализованы все необходимые классы для работы с файлами. Чтобы подключить это пространство имен, необходимо в самом начале программы добавить строку *using System.IO*. Для использования кодировок еще добавим пространство *using System.Text*;

Работа с дисками

Работу с файловой системой начнем с самого верхнего уровня - дисков. Для представления диска в пространстве имен *System.IO* имеется класс *DriveInfo*.

Этот класс имеет статический метод *GetDrives*, который возвращает имена всех логических дисков компьютера. Также он предоставляет ряд полезных свойств:

AvailableFreeSpace: указывает на объем доступного свободного места на диске в байтах

DriveFormat: получает имя файловой системы

DriveType: представляет тип диска

IsReady: готов ли диск (например, DVD-диск может быть не вставлен в дисковод)

Name: получает имя диска

TotalFreeSpace: получает общий объем свободного места на диске в байтах

TotalSize: общий размер диска в байтах

VolumeLabel: получает или устанавливает метку тома

Получим имена и свойства всех дисков на компьютере:

```
using System;
using System.Collections.Generic;
using System.IO;

namespace FileApp
{
    class Program
    {
        static void Main(string[] args)
        {
            DriveInfo[] drives = DriveInfo.GetDrives();
```

```
                foreach (DriveInfo drive in drives)
                {
                    Console.WriteLine("Название: {0}", drive.Name);
                    Console.WriteLine("Тип: {0}", drive.DriveType);

                    if (drive.IsReady)
                    {
                        Console.WriteLine("Объем          диска:          {0}",
drive.TotalSize);
                        Console.WriteLine("Свободное    пространство:    {0}",
drive.TotalFreeSpace);
                        Console.WriteLine("Метка: {0}", drive.VolumeLabel);
                    }
                    Console.WriteLine();
                }

                Console.ReadLine();
            }
        }
    }
```

Работа с каталогами (папками)

Для работы с каталогами в пространстве имен *System.IO* предназначены сразу два класса: *Directory* и *DirectoryInfo*.

Основной класс – *DirectoryInfo*, он не имеет статичных методов и предназначен для работы с конкретными каталогами. Второй класс – *Directory*, является вспомогательным и содержит часто используемые статичные методы, функционал которых, по сути, реализован через класс *DirectoryInfo*.

Класс *DirectoryInfo* при инициализации принимает путь к каталогу, который он реализует.

Класс *Directory*

Класс *Directory* предоставляет ряд статических методов для управления каталогами. Некоторые из этих методов:

CreateDirectory(path): создает каталог по указанному пути path

Delete(path): удаляет каталог по указанному пути path

Exists(path): определяет, существует ли каталог по указанному пути path. Если существует, возвращается true, если не существует, то false

GetDirectories(path): получает список каталогов в каталоге path

GetFiles(path): получает список файлов в каталоге path

Move(sourceDirName, destDirName): перемещает каталог

GetParent(path): получение родительского каталога

Класс *DirectoryInfo*

Данный класс предоставляет функциональность для создания, удаления, перемещения и других операций с каталогами. Во многом он похож на *Directory*. Некоторые из его свойств и методов:

Create(): создает каталог

CreateSubdirectory(path): создает подкаталог по указанному пути path

Delete(): удаляет каталог

Свойство *Exists*: определяет, существует ли каталог

GetDirectories(): получает список каталогов

GetFiles(): получает список файлов

MoveTo(destDirName): перемещает каталог

Свойство *Parent*: получение родительского каталога

Свойство *Root*: получение корневого каталога

Посмотрим на примере применение этих классов
Получение списка файлов и подкаталогов

```
string dirName = "C:\\";

if (Directory.Exists(dirName))
{
    Console.WriteLine("Подкаталоги:");
    string[] dirs = Directory.GetDirectories(dirName);
    foreach (string s in dirs)
    {
        Console.WriteLine(s);
    }
    Console.WriteLine();
    Console.WriteLine("Файлы:");
    string[] files = Directory.GetFiles(dirName);
    foreach (string s in files)
    {
        Console.WriteLine(s);
    }
}
```

Обратите внимание на использование слешей в именах файлов. Либо мы используем двойной слеш: "C:\\", либо одинарный, но тогда перед всем путем ставим знак @: @"C:\Program Files"

Создание каталога

```
string path = @"C:\SomeDir";
string subpath = @"program\avalon";
DirectoryInfo dirInfo = new DirectoryInfo(path);
if (!dirInfo.Exists)
{
    dirInfo.Create();
}
dirInfo.CreateSubdirectory(subpath);
```

Работа с файлами. Классы *File* и *FileInfo*

Для работы с файлами предназначена пара классов *File* и *FileInfo*. С их помощью можно создавать, удалять, перемещать файлы, получать их свойства и многое другое.

Некоторые полезные методы и свойства класса *FileInfo*:
CopyTo(path): копирует файл в новое место по указанному пути *path*
Create(): создает файл
Delete(): удаляет файл
MoveTo(destFileName): перемещает файл в новое место
Свойство *Directory*: получает родительский каталог в виде объекта *DirectoryInfo*
Свойство *DirectoryName:* получает полный путь к родительскому каталогу
Свойство *Exists*: указывает, существует ли файл
Свойство *Length*: получает размер файла
Свойство *Extension*: получает расширение файла
Свойство *Name*: получает имя файла
Свойство *FullName*: получает полное имя файла

Класс *File* реализует похожую функциональность с помощью статических методов:
Copy(): копирует файл в новое место
Create(): создает файл

Delete(): удаляет файл
Move: перемещает файл в новое место
Exists(file): определяет, существует ли файл

Получение информации о файле

```
string path = @"C:\apache\hta.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
    Console.WriteLine("Имя файла: {0}", fileInf.Name);
    Console.WriteLine("Время создания: {0}", fileInf.CreationTime);
    Console.WriteLine("Размер: {0}", fileInf.Length);
}
```

Удаление файла

```
string path = @"C:\apache\hta.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
   fileInf.Delete();
   // альтернатива с помощью класса File
   // File.Delete(path);
}
```

Перемещение файла

```
string path = @"C:\apache\hta.txt";
string newPath = @"C:\SomeDir\hta.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
   fileInf.MoveTo(newPath);
   // альтернатива с помощью класса File
   // File.Move(path, newPath);
}
```

Копирование файла

```
string path = @"C:\apache\hta.txt";
string newPath = @"C:\SomeDir\hta.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
   fileInf.CopyTo(newPath, true);
   // альтернатива с помощью класса File
   // File.Copy(path, newPath, true);
}
```

Метод *CopyTo* класса *FileInfo* принимает два параметра: путь, по которому файл будет копироваться, и булевое значение, которое указывает, надо ли при копировании перезаписывать файл (если *true*, как в случае выше, файл при копировании перезаписывается). Если же в качестве последнего параметра передать значение *false*, то если такой файл уже существует, приложение выдаст ошибку.

Метод *Copy* класса *File* принимает три параметра: путь к исходному файлу, путь, по которому файл будет копироваться, и булевое значение, указывающее, будет ли файл перезаписываться.

Чтение и запись файла. Класс *FileStream*

Кроме того, чтобы читать/записывать данные в файл с Си# можно использовать потоки.

Поток – это абстрактное представление данных (в байтах), которое облегчает работу с ними. В качестве источника данных может быть файл, устройство ввода-вывода, принтер.

Класс *Stream* является абстрактным базовым классом для всех потоковых классов в Си. Для работы с файлами понадобится класс *FileStream* (файловый поток).

FileStream - представляет поток, который позволяет выполнять операции чтения/записи в файл.

Класс *FileStream* представляет возможности по считыванию из файла и записи в файл. Он позволяет работать как с текстовыми файлами, так и с бинарными.

Рассмотрим наиболее важные его свойства и методы:

Свойство *Length*: возвращает длину потока в байтах

Свойство *Position*: возвращает текущую позицию в потоке

Метод *Read*: считывает данные из файла в массив байтов. Принимает три параметра:

int Read(byte[] array, int offset, int count)

и возвращает количество успешно считанных байтов. Здесь используются следующие параметры:

array - массив байтов, куда будут помещены считываемые из файла данные

offset представляет смещение в байтах в массиве array, в который считанные байты будут помещены

count - максимальное число байтов, предназначенных для чтения. Если в файле находится меньшее количество байтов, то все они будут считаны.

Метод *long Seek(long offset, SeekOrigin origin)*: устанавливает позицию в потоке со смещением на количество байт, указанных в параметре offset.

Метод *Write*: записывает в файл данные из массива байтов. Принимает три параметра:

Write(byte[] array, int offset, int count)

array - массив байтов, откуда данные будут записываться в файла

offset - смещение в байтах в массиве array, откуда начинается запись байтов в поток

count - максимальное число байтов, предназначенных для записи

FileStream представляет доступ к файлам на уровне байтов, поэтому, например, если необходимо считать или записать одну или несколько строк в текстовый файл, то массив байтов надо преобразовать в строки, используя специальные методы. Поэтому для работы с текстовыми файлами применяются другие классы.

В то же время при работе с различными бинарными файлами, имеющими определенную структуру, *FileStream* может быть очень даже полезен для извлечения определенных порций информации и ее обработки.

Посмотрим на примере считывания-записи в текстовый файл:

```
Console.WriteLine("Введите строку для записи в файл:");
string text = Console.ReadLine();

// запись в файл
using (FileStream fstream = new FileStream(@"C:\SomeDir\noname\note.txt",
FileMode.OpenOrCreate))
{
    // преобразуем строку в байты
    byte[] array = System.Text.Encoding.Default.GetBytes(text);

    // запись массива байтов в файл
    fstream.Write(array, 0, array.Length);
    Console.WriteLine("Текст записан в файл");
```

```
}


// чтение из файла
using (FileStream fstream = File.OpenRead(@"C:\SomeDir\noname\note.txt"))
{
    // преобразуем строку в байты
    byte[] array = new byte[fstream.Length];

    // считываем данные
    fstream.Read(array, 0, array.Length);

    // декодируем байты в строку
    string textFromFile = System.Text.Encoding.Default.GetString(array);
    Console.WriteLine("Текст из файла: {0}", textFromFile);
}

Console.ReadLine();
```

Разберем этот пример. И при чтении, и при записи используется оператор *using*. Не надо путать данный оператор с директивой *using*, которая подключает пространства имен в начале файла кода. Оператор *using* позволяет создавать объект в блоке кода, по завершению которого вызывается метод *Dispose* у этого объекта, и, таким образом, объект уничтожается. В данном случае в качестве такого объекта служит переменная *fstream*.

Объект *fstream* создается двумя разными способами: через конструктор и через один из статических методов класса *File*.

Здесь в конструктор передается два параметра: путь к файлу и перечисление *FileMode*. Данное перечисление указывает на *режим доступа к файлу* и может принимать следующие значения:

Append: если файл существует, то текст добавляется в конец файла. Если файла нет, то он создается. Файл открывается только для записи.

Create: создается новый файл. Если такой файл уже существует, то он перезаписывается

CreateNew: создается новый файл. Если такой файл уже существует, то приложение выдает ошибку

Open: открывает файл. Если файл не существует, выдается ошибка (исключение)

OpenOrCreate: если файл существует, он открывается, если нет - создается новый

Truncate: если файл существует, то он перезаписывается. Файл открывается только для записи.

Статический метод *OpenRead* класса *File* открывает файл для чтения и возвращает объект *FileStream*.

Конструктор класса *FileStream* также имеет ряд перегруженных версий, позволяющий более точно настроить создаваемый объект. Все эти версии можно посмотреть на *msdn*.

И при записи, и при чтении применяется объект кодировки *Encoding.Default* из пространства имен *System.Text*. В данном случае используем два его метода: *GetBytes* для получения массива байтов из строки и *GetString* для получения строки из массива байтов.

В итоге введенная строка записывается в файл *noname/note.txt*. По сути это бинарный файл (не текстовый), хотя если в него запишем только строку, то сможем посмотреть в удобочитаемом виде этот файл, открыв его в текстовом редакторе. Однако если в него запишем случайные байты, например:

```
fstream.WriteByte(13);
fstream.WriteByte(103);
```

то могут возникнуть проблемы с его пониманием. Поэтому для работы непосредственно с текстовыми файлами предназначены отдельные классы - *StreamReader* и *StreamWriter*.

Произвольный доступ к файлам

Нередко бинарные файлы представляют определенную структуру. И, зная эту структуру, можно взять из файла нужную порцию информации или наоборот записать в определенном месте файла определенный набор байтов. Например, в wav-файлах непосредственно звуковые данные начинаются с 44 байта, а до 44 байта идут различные метаданные - количество каналов аудио, частота дискретизации и т.д.

С помощью метода *Seek()* можно управлять положением курсора потока, начиная с которого производится считывание или запись в файл.

long Seek(long offset, SeekOrigin origin)

Этот метод принимает два параметра: *offset* (смещение) и позиция в файле. Позиция в файле описывается тремя значениями:

SeekOrigin.Begin: начало файла

SeekOrigin.End: конец файла

SeekOrigin.Current: текущая позиция файла

Курсор потока, с которого начинается чтение или запись, смещается вперед на значение *offset* относительно позиции, указанной в качестве второго параметра. Смещение может быть отрицательным, тогда курсор сдвигается назад, если положительное - то вперед.

Рассмотрим на примере:

```
using System.IO;
using System.Text;

class Program
{
    static void Main(string[] args)
    {
        string text = "hello world";

        // запись в файл
        using (FileStream fstream = new FileStream(@"D:\note.dat",
FileMode.OpenOrCreate))
        {
            // преобразуем строку в байты
            byte[] input = Encoding.Default.GetBytes(text);

            // запись массива байтов в файл
            fstream.Write(input, 0, input.Length);
            Console.WriteLine("Текст записан в файл");

            // перемещаем указатель в конец файла, до конца файла- пять
байт
            fstream.Seek(-5, SeekOrigin.End); // минус 5 символов с конца
потока

            // считываем четыре символов с текущей позиции
            byte[] output = new byte[4];
            fstream.Read(output, 0, output.Length);

            // декодируем байты в строку
            string textFromFile = Encoding.Default.GetString(output);
            Console.WriteLine("Текст из файла: {0}", textFromFile); //
worl

            // заменим в файле слово world на слово house
```

```
                string replaceText = "house";
                fstream.Seek(-5, SeekOrigin.End); // минус 5 символов с конца
потока
                input = Encoding.Default.GetBytes(replaceText);
                fstream.Write(input, 0, input.Length);

                // считываем весь файл
                // возвращаем указатель в начало файла
                fstream.Seek(0, SeekOrigin.Begin);

                output = new byte[fstream.Length];
                fstream.Read(output, 0, output.Length);
                // декодируем байты в строку
                textFromFile = Encoding.Default.GetString(output);
                Console.WriteLine("Текст из файла: {0}", textFromFile); //
hello house
            }
            Console.Read();
        }
    }
```

Консольный вывод:

Вызов *fstream.Seek(-5, SeekOrigin.End)* перемещает курсор потока в конец файла назад на пять символов:

То есть после записи в новый файл строки "*hello world*" курсор будет стоять на позиции символа "*w*".

После этого считываем четыре байта, начиная с символа "*w*". В данной кодировке 1 символ будет представлять 1 байт. Поэтому чтение 4 байтов будет эквивалентно чтению четырех символов: "*worl*".

Затем опять же перемещаемся в конец файла, не доходя до конца пять символов (то есть опять же с позиции символа "*w*"), и осуществляем запись строки "*house*". Таким образом, строка "*house*" заменяет строку "*world*".

Закрытие потока

В примерах выше для закрытия потока применяется конструкция *using*. После того как все операторы и выражения в блоке *using* отработают, объект *FileStream* уничтожается. Однако можно выбрать и другой способ:

```
FileStream fstream = null;
try
{
    fstream = new FileStream(@"D:\note3.dat", FileMode.OpenOrCreate);
    // операции с потоком
}
catch(Exception ex)
{
```

```
}
finally
{
    if (fstream != null)
        fstream.Close();
}
```

Если не используется конструкция *using*, то надо явным образом вызвать метод *Close()*:

fstream.Close()

Чтение и запись текстовых файлов. *StreamReader* и *StreamWriter*

Класс *FileStream* не очень удобно применять для работы с текстовыми файлами. К тому же для этого в пространстве *System.IO* определены специальные классы: *StreamReader* и *StreamWriter*.

Чтение из файла и StreamReader

Класс *StreamReader* позволяет легко считывать весь текст или отдельные строки из текстового файла. Среди его методов можно выделить следующие:

Close: закрывает считываемый файл и освобождает все ресурсы

Peek: возвращает следующий доступный символ, если символов больше нет, то возвращает -1

Read: считывает и возвращает следующий символ в численном представлении. Имеет перегруженную версию:

Read(char[] array, int index, int count),

где *array* - массив, куда считываются символы, *index* - индекс в массиве array, начиная с которого записываются считываемые символы, и *count* - максимальное количество считываемых символов

ReadLine: считывает одну строку в файле

ReadToEnd: считывает весь текст из файла

Считаем текст из файла различными способами:

```
string path= @"C:\SomeDir\hta.txt";

try
{
    Console.WriteLine("******считываем весь файл********");
    using (StreamReader sr = new StreamReader(path))
    {
        Console.WriteLine(sr.ReadToEnd());
    }

    Console.WriteLine();


    Console.WriteLine("******считываем построчно********");
    using (StreamReader sr = new StreamReader(path,
System.Text.Encoding.Default))
    {
        string line;
        while ((line = sr.ReadLine()) != null)
        {
            Console.WriteLine(line);
        }
    }
```

```
            Console.WriteLine();


            Console.WriteLine("******считываем блоками********");
            using      (StreamReader      sr      =      new      StreamReader(path,
System.Text.Encoding.Default))
            {
                char[] array = new char[4];
                // считываем 4 символа
                sr.Read(array, 0, 4);

                Console.WriteLine(array);
            }
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
```

Как и в случае с классом *FileStream* здесь используется конструкция *using*.

В первом случае разом считываем весь текст с помощью метода *ReadToEnd()*.

Во втором случае считываем построчно через цикл *while*:

while ((line = sr.ReadLine()) != null) - сначала присваиваем переменной *line* результат функции *sr.ReadLine()*, а затем проверяем, не равна ли она *null*. Когда объект *sr* дойдет до конца файла и больше строк не останется, то метод *sr.ReadLine()* будет возвращать *null*.

В третьем случае считываем в массив четыре символа.

Обратите внимание, что в последних двух случаях в конструкторе *StreamReader* указывалась кодировка *System.Text.Encoding.Default*. Свойство *Default* класса *Encoding* получает кодировку для текущей кодовой страницы *ANSI*. Также через другие свойства можно указать другие кодировки. Если кодировка не указана, то при чтении используется *UTF8*.

Запись в файл и *StreamWriter*

Для записи в текстовый файл используется класс *StreamWriter*. Свою функциональность он реализует через следующие методы:

Close: закрывает записываемый файл и освобождает все ресурсы

Flush: записывает в файл оставшиеся в буфере данные и очищает буфер.

Write: записывает в файл данные простейших типов, как int, double, char, string и т.д.

WriteLine: также записывает данные, только после записи добавляет в файл символ окончания строки

Рассмотрим запись в файл на примере:

```
        string readPath= @"C:\SomeDir\hta.txt";
        string writePath = @"C:\SomeDir\ath.txt";

        string text = "";
        try
        {
            using      (StreamReader      sr      =      new      StreamReader(readPath,
System.Text.Encoding.Default))
            {
                text=sr.ReadToEnd();
            }
            using   (StreamWriter   sw   =   new   StreamWriter(writePath,   false,
System.Text.Encoding.Default))
            {
                sw.WriteLine(text);
            }
```

```
        using (StreamWriter sw = new StreamWriter(writePath, true,
System.Text.Encoding.Default))
        {
            sw.WriteLine("Дозапись");
            sw.Write(4.5);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
```

Здесь сначала считываем файл в переменную *text*, а *затем* записываем эту переменную в файл, а затем через объект *StreamWriter* записываем в новый файл.

Класс *StreamWriter* имеет несколько конструкторов. Здесь использовали один из них:

new StreamWriter(writePath, false, System.Text.Encoding.Default).

В качестве первого параметра передается путь к записываемому файлу. Второй параметр представляет булевую переменную, которая определяет, будет файл дозаписываться или перезаписываться. Если этот параметр равен *true*, то новые данные добавляются в конце к уже имеющимся данным. Если *false*, то файл перезаписывается. Третий параметр указывает кодировку, в которой записывается файл.

Работа с бинарными файлами.

BinaryWriter и *BinaryReader*

Для работы с бинарными файлами предназначена пара классов *BinaryWriter* и *BinaryReader*. Эти классы позволяют читать и записывать данные в двоичном формате.

Основные метода класса *BinaryWriter*

Close(): закрывает поток и освобождает ресурсы

Flush(): очищает буфер, дописывая из него оставшиеся данные в файл

Seek(): устанавливает позицию в потоке

Write(): записывает данные в поток

Основные метода класса *BinaryReader*

Close(): закрывает поток и освобождает ресурсы

ReadBoolean(): считывает значение *bool* и перемещает указать на один байт

ReadByte(): считывает один байт и перемещает указать на один байт

ReadChar(): считывает значение *char*, то есть один символ, и перемещает указать на столько байтов, сколько занимает символ в текущей кодировке

ReadDecimal(): считывает значение *decimal* и перемещает указать на 16 байт

ReadDouble(): считывает значение *double* и перемещает указать на 8 байт

ReadInt16(): считывает значение *short* и перемещает указать на 2 байта

ReadInt32(): считывает значение *int* и перемещает указать на 4 байта

ReadInt64(): считывает значение *long* и перемещает указать на 8 байт

ReadSingle(): считывает значение *float* и перемещает указать на 4 байта

ReadString(): считывает значение *string*. Каждая строка предваряется значением длины строки, которое представляет 7-битное целое число

С чтением бинарных данных все просто: соответствующий метод считывает данные определенного типа и перемещает указатель на размер этого типа в байтах, например, значение типа *int* занимает 4 байта, поэтому *BinaryReader* считает 4 байта и переместит указать на эти 4 байта.

Посмотрим на реальной задаче применение этих классов. Попробуем с их помощью записывать и считывать из файла массив структур:

```
struct State
```

```
    {
        public string name;
        public string capital;
        public int area;
        public double people;

        public State(string n, string c, int a, double p)
        {
            name = n;
            capital = c;
            people = p;
            area = a;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            State[] states = new State[2];
            states[0] = new State("Германия", "Берлин", 357168, 80.8);
            states[1] = new State("Франция", "Париж", 640679, 64.7);

            string path= @"C:\SomeDir\states.dat";

            try
            {
                // создаем объект BinaryWriter
                using (BinaryWriter writer = new BinaryWriter(File.Open(path,
FileMode.OpenOrCreate)))
                {
                    // записываем в файл значение каждого поля структуры
                    foreach (State s in states)
                    {
                        writer.Write(s.name);
                        writer.Write(s.capital);
                        writer.Write(s.area);
                        writer.Write(s.people);
                    }
                }
                // создаем объект BinaryReader
                using (BinaryReader reader = new BinaryReader(File.Open(path,
FileMode.Open)))
                {
                    // пока не достигнут конец файла
                    // считываем каждое значение из файла
                    while (reader.PeekChar() > -1)
                    {
                        string name = reader.ReadString();
                        string capital = reader.ReadString();
                        int area = reader.ReadInt32();
                        double population = reader.ReadDouble();

                        Console.WriteLine("Страна: {0}  столица: {1}  площадь
{2} кв. км   численность населения: {3} млн. чел.",
                            name, capital, area, population);
                    }
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            Console.ReadLine();
        }
    }
```

Итак, имеется структура *State* с некоторым набором полей. В основной программе создаем массив структур и записываем с помощью *BinaryWriter*. Этот класс в качестве параметра в конструкторе принимает объект *Stream*, который создается вызовом *File.Open(path, FileMode.OpenOrCreate).*

Затем в цикле пробегаемся по массиву структур и записываем каждое поле структуры в поток. В том порядке, в каком они значения полей записываются, в том порядке они и будут размещаться в файле.

Затем считываем из записанного файла. Конструктор класса *BinaryReader* также в качестве параметра принимает объект потока, только в данном случае устанавливаем в качестве режима *FileMode.Open: new BinaryReader(File.Open(path, FileMode.Open))*

В цикле *while* считываем данные. Чтобы узнать окончание потока, вызываем метод *PeekChar()*. Этот метод считывает следующий символ и возвращает его числовое представление. Если символ отсутствует, то метод возвращает -1, что будет означать, что достигнут конец файла.

В цикле последовательно считываем значения поле структур в том же порядке, в каком они записывались.

Таким образом, классы *BinaryWriter* и *BinaryReader* очень удобны для работы с бинарными файлами, особенно когда известна структура этих файлов.

Работа с XML- файлами.

http://metanit.com/sharp/tutorial
http://kbyte.ru/ru/Programming/Articles.aspx?id=74&mode=art
Обработка больших XML при помощи XmlReader
Console
http://vscode.ru/prog-lessons/sozdanie_chtenie-xml-c-sharp.html
WF приложение
http://www.realcoding.net/article/view/1810
(Windows-forms приложение)
http://csharpprogramming.ru/xml/xml-dom-2
DOM-модель
http://www.studfiles.ru/preview/4392937/

Структура XML-файла

```
<?xml version="1.0" encoding="utf-16" ?>
<Заказы>
    <Заказ Адрес="Уфа" Дата="21.04.2004">
        <Товар Название="Товар_А" Цена="100" />
        <Товар Название="Товар_Б" Цена="150" />
        <Товар Название="Товар_В" Цена="370" />
     </Заказ>
    <Заказ Адрес="Москва" Дата="24.04.2004">
        <Товар Название="Товар_Г" Цена="400" />
    </Заказ>
    <Заказ Адрес="Омск" Дата="28.04.2004">
        <Товар Название="Товар_Д" Цена="255" />
    </Заказ>
</Заказы>
```

Для работы с XML применяются XML-парсеры. Существует два основных типа парсеров:

Simple API for XML (SAX) и Document Object Model (DOM).

SAX основан на курсорах и событиях, возникающих при проходе по узлам XML документа. SAX-прасеру не требуется большого количества памяти для разбора даже больших документов (т.к. ему не нужно загружать в память весь документ), но его существенным ограничением является то, что можно перемещаться по документу только в одном направлении.

DOM полностью загружает документ в память и представляет его в виде дерева, поэтому можно произвольно перемещаться по XML-документу.