Лекция 1.

Парадигмы программирования. Объектно-ориентированное программирование. Объекты и классы. Абстракция и иерархия.

Вся история программирования – это попытка совладать со сложностью окружающего мира. Задачи, встающие перед программистами, становятся все более громоздкими, информация, которую необходимо обработать, непрерывно растет. Как только программисты предлагают более-менее удовлетворительное решение предложенных задач, тут же возникают новые, еще более сложные задачи. Программисты придумывают новые методы программирования, стили, создают новые языки. Некоторые методы и стили становятся общепринятыми и образуют на некоторое время так называемую **парадигму программирования.**

Первые, даже самые простые задачи, написанные в машинных кодах, составляли сотни строк совершенно непонятного текста. Для упрощения и ускорения программирования были придуманы языки программирования высокого уровня, которые возложили рутинные операции по созданию машинного кода на компилятор. Те же программы, переписанные на языках высокого уровня, становились короче и понятнее. Но с возрастанием сложности задач программы снова увеличились в размерах, стали необозримыми.

Возникла идея оформить программу в виде нескольких, по возможности простых, процедур или функций, каждая из которых решает свою определенную задачу. Написать, откомпилировать и отладить небольшую процедуру можно легко и быстро. Остается только собрать все процедуры в нужном порядке в одну программу. Кроме того, один раз написанные процедуры можно затем использовать в других программах как строительные блоки. **Процедурное программирование** стало парадигмой.

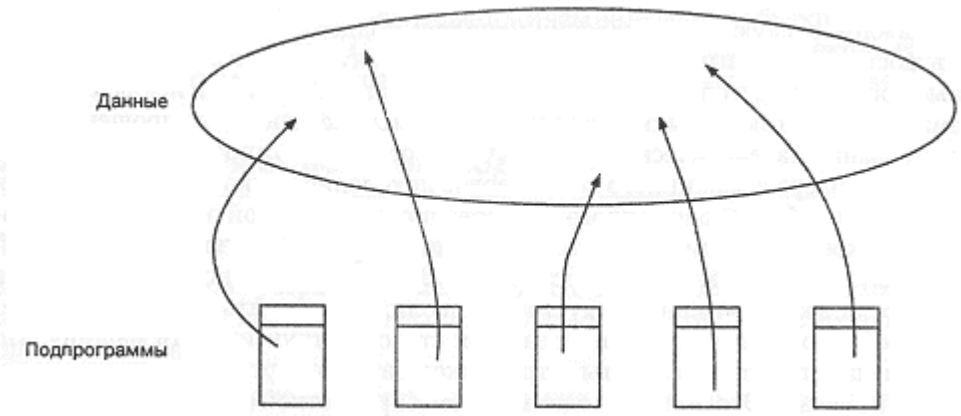


Рис. 1. Процедурная программа.

Встал вопрос о том, как выявить структуру программы, разбить программу на процедуры, какую часть кода выделить в отдельную процедуру, как сделать алгоритм решения задачи простым и наглядным, как удобнее связать процедуры между собой. Были предложены рекомендации, названные структурным программированием. **Структурное программирование** оказалось удобным и стало парадигмой. Появились языки программирования (например, Pascal), на которых удобно писать структурные программы. Более того, на таких языках оказалось сложно писать неструктурные программы.

Усложнение задач привело к тому, что программы стали содержать сотни процедур и опять стали необозримыми. «Строительные блоки» стали слишком маленькими. Потребовался новый стиль программирования. В это же время обнаружилось, что удачная или неудачная структура данных может сильно облегчить или усложнить их обработку. Одни данные удобнее объединять в массив, для других больше подходит список, стек или дерево. Возникла идея объединить исходные данные и все процедуры их обработки в один модуль. Эта идея **модульного программирования** на некоторое время стала парадигмой.

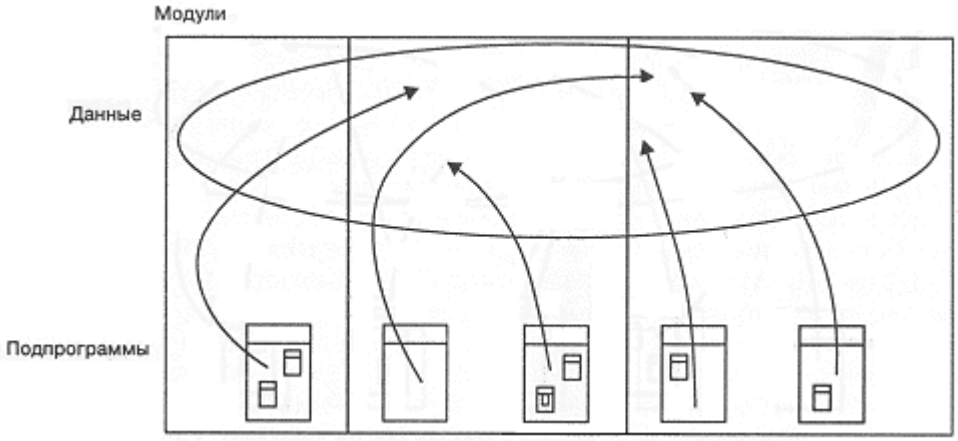


Рис.2. Модульная программа.

Первоначально такой модуль представлял собой просто более или менее случайный набор данных и подпрограмм. В такие модули собирали подпрограммы, которые, как казалось, скорее всего, будут изменяться совместно. Программы составлялись из отдельных модулей. Эффективность таких программ тем выше, чем меньше модули зависят друг от друга. Для того, чтобы обеспечить максимальную независимость модулей, надо четко отделить процедуры, которые будут вызываться другими модулями (*открытые* процедуры), от вспомогательных, которые обрабатывают данные, заключенные в этот модуль (*закрытые* процедуры). Данные, занесенные в модуль, также делятся на открытые и закрытые.

Оказалось удобным разбить программу на модули так, чтобы она превратилась в совокупность взаимодействующих объектов. Так возникло **объектно-ориентированное программирование** (ООП) – современная парадигма программирования. Как известно, все программы состоят из двух элементов: кода и данных. Любая программа может быть концептуально организована либо вокруг ее кода («кодовое воздействие на данные»), либо вокруг данных («управляемый данными доступ к коду»). При первом (процедурном) подходе программу определяет последовательность операторов ее кода. Второй подход организует программу вокруг данных (т.е. вокруг объектов) и набора хорошо организованных интерфейсов (взаимодействий) с этими данными.

Что же такое объектно-ориентированное программирование (object-oriented programming, OOP)? Можно определить его следующим образом:

Объектно-ориентированное программирование - это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

В данном определении можно выделить три части: 1) ООП использует в качестве базовых элементов *объекты,* а не алгоритмы; 2) каждый объект является *экземпляром* какого-либо определенного *класса;* 3) классы организованы *иерархически.*

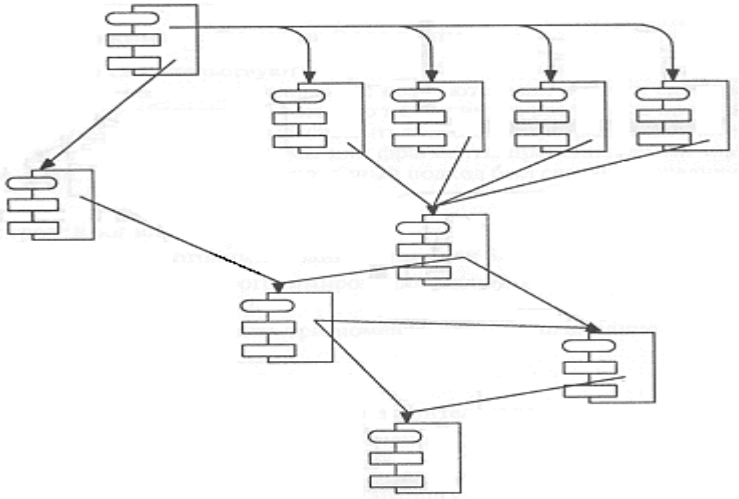


Рис. 3. Иерархическая структура классов

Что же такое объект? С точки зрения восприятия человеком объектом может быть:

- осязаемый и (или) видимый предмет;
- нечто, воспринимаемое мышлением;
- нечто, на что направлена мысль или действие.

Объект обладает определенными характеристиками. Например, в качестве объекта можно представить себе, например, автомобиль, человека, некоторое явление природы (туман, дождь), окно на экране дисплея. Автомобиль может характеризоваться грузоподъемностью, максимальной скоростью. Характеристики окна - это положение его на экране (обычно определяемое координатами верхнего левого угла), ширина, высота, цвет фона и т.д. Объект может производить определенные действия, меняя при этом свои характеристики. Например, окно может перемещаться по экрану, уменьшаться или увеличиваться в размерах. При этом меняются координаты окна, его ширина, высота и т.п.

Итак, объект моделирует часть окружающей действительности и таким образом существует во времени и пространстве.

Описывая поведение какого-либо объекта, мы строим его модель. Модель, как правило, не может описать объект полностью. Реальные объекты слишком сложны. Необходимо отбирать только те характеристики объекта, которые важны для решения поставленной задачи. Например, при описании грузоперевозок для нас несущественен цвет автомобиля. Если же мы хотим определить легковые автомобили, выставленные на продажу, эта характеристика будет весьма важна. Таким образом, мы должны абстрагироваться от некоторых конкретных деталей объекта. Дадим следующее определение абстракции:

Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности поведения от несущественных.

Кроме самых простых ситуаций, число абстракций в системе обычно превышает наши умственные возможности. Значительное упрощение в понимании сложных задач достигается за счет образования из абстракций иерархической структуры. Определим иерархию следующим образом:

Иерархия - это упорядочение абстракций, расположение их по уровням.

Большинство людей видит мир состоящим из объектов, которые связаны друг с другом иерархическим (т.е. организованным «сверху вниз») способом. Хорошо это видно на примере биологии. Например, сиамские кошки являются породой кошек, которые в свою очередь являются представителем семейства кошачьих. Это семейство есть часть класса млекопитающих и т.д. Естественно, на каждом уровне иерархии объекты имеют свой набор свойств. Но одновременно объекты, расположенные в иерархии ниже. Обладают всеми свойствами объектов, расположенных выше по иерархической лестнице. Кошка любой породы обладает чертами, свойственными всем кошкам – шерсть, усы. Любой представитель семейства кошачьих обладает чертами класса млекопитающих – скелет, молочные железы и т.п. Таким образом, имея описание объекта, стоящего на более высокой ступени иерархии, нет необходимости описывать полностью объект, стоящий в иерархии ниже. Достаточно указать лишь его специфические черты.

Описание каждой модели производится в виде одного или нескольких классов. Класс можно считать проектом, чертежом, по которому будут создаваться конкретные объекты. Класс содержит описание переменных и констант, характеризующих объект. Они называются *полями* класса. Процедуры, описывающие поведение объекта, называются *методами* класса. Поля и методы класса являются *членами* класса.

Все языки объектно-ориентированного программирования обеспечивают механизмы, которые позволяют реализовать объектно-ориентированную модель. Они будут описаны в следующей лекции.

Лекция 2.

Принципы объектно-ориентированного программирования.

Все языки объектно-ориентированного программирования обеспечивают механизмы, которые помогают реализовывать объектно-ориентированную модель. К ним относятся *инкапсуляция*, *наследование* и *полиморфизм*.

Буч определяет **инкапсуляцию** следующим образом:

Инкапсуляция, encapsulation - процесс разделения элементов абстракции, которые образуют ее структуру и поведение. Служит для отделения внешних обязательств объекта от его реализации.

Другими словами, инкапсуляция – это механизм, который связывает код вместе с обрабатываемыми им данными и сохраняет их в безопасности как от внешнего влияния, так и от ошибочного использования. При проектировании объектов программист должен решить, какие части объекта должны быть доступны для пользователя, а какие следует изолировать в объекте. Детали класса, остающиеся невидимыми для пользователя, называются инкапсулированными в классе.

Обычно при проектировании объектов стремятся инкапсулировать как можно большую часть класса. Это делается по следующим причинам:

- Закрытые данные объекта не могут изменяться пользователем, что уменьшает вероятность вмешательства в работу объекта или зависимости пользовательского кода от внутренней структуры объекта. Если пользовательский код зависит от структуры объекта, изменение объекта может нарушить работу пользовательского кода.
- При внесении изменений в открытые части объекта необходимо обеспечить совместимость с предыдущими версиями. Чем большую часть объекта видит пользователь, чем меньше можно изменить без нарушения пользовательского кода. Разумная инкапсуляция локализует те особенности проекта, которые могут подвергнуться изменениям. По мере развития системы разработчики могут решить, что какие-то операции выполняются несколько дольше, чем допустимо, а какие-то объекты занимают больше памяти, чем приемлемо. В таких ситуациях часто изменяют внутреннее представление объекта, чтобы реализовать более эффективные алгоритмы или оптимизировать алгоритм по критерию памяти, заменяя хранение данных вычислением. Важным преимуществом ограничения доступа является возможность внесения изменений в объект без изменения других объектов.
- Объемные интерфейсы усложняют всю систему. Доступ к закрытым данным возможен только внутри самого класса; к открытым данным можно обратиться извне через любой объект. Как правило, наличие большого числа открытых членов класса заметно усложняет отладку программы.

Использование инкапсуляции дает возможность использовать внутреннюю структуру объекта, не соответствующую представлению об этом объекте. Рассмотрим, например, автоматическую трансмиссию автомобиля. Она инкапсулирует информацию о машине, например, о величине ускорения, позицию рукоятки передач и т.п. Как пользователь, вы имеете только один метод воздействия – перемещение рукоятки передач. Невозможно, например, повлиять на трансмиссию, используя сигнал поворота. Таким образом, рукоятка – интерфейс к трансмиссии.

Определим интерфейс следующим образом:

Интерфейс (interface) - внешний вид класса, объекта или модуля, выделяющий его существенные черты и не показывающий внутреннего устройства и секретов поведения.

Вернемся к трансмиссии. Все, что происходит внутри трансмиссии, никак не влияет на ее внешние объекты. Например, смещение рычага передач никак не сказывается на сигналах поворота. Именно потому, что автоматическая трансмиссия инкапсулирована, масса производителей автомобилей может реализовать ее любым способом, которой им покажется

наиболее правильным. Однако с точки зрения пользователей (водителей) все трансмиссии будут работать одинаково.

Та же самая идея применяется и в программировании. Например, проектируя систему платежей, можно прийти к необходимости использовать объект «Деньги». С точки зрения пользователя объекты такого класса должны представляться состоящими из двух частей – рублей и копеек. Внутреннее представление может быть совершенно другим. Понятно, что внешнее представление неудобно для расчетов. Можно, например, использовать внутренне представление, использующее только копейки. Операции на копейками будут определяться проще и выполняться быстрее. И только если пользователю потребуется использовать результаты расчетов, внутреннее представление будет переводиться во внешнее.

Для того чтобы скрыть реализацию класса, каждое поле и каждый метод класса должен быть помечен как **private** (частный или локальный) или **public** (общий). Методы и поля с модификатором *private* могут быть доступны только в коде, являющемся членом данного класса. Модификатор *public* указывает на все, что нужно или можно знать пользователям класса. При этом настоятельно не рекомендуется использовать в классах *public* поля. Доступ к полям должен всегда контролироваться с использованием соответствующих *public* методов.

Определим **наследование** следующим образом:

Наследование, inheritance - отношение между классами, при котором класс использует структуру или поведение другого (одиночное наследование) или других (множественное наследование) классов. Наследование вводит иерархию "общее/частное", в которой подкласс наследует от одного или нескольких более общих суперклассов. Подклассы обычно дополняют или переопределяют унаследованную структуру и поведение.

Наследование есть процесс, с помощью которого один объект приобретает свойства другого. Как уже упоминалось в предыдущей лекции, наибольшая часть знаний становится управляемой только с помощью иерархических классификаций. Без применения классификаций каждый объект нуждался бы в явном определении всех своих характеристик. При использовании наследования объект нуждается в определении только тех качеств, которые делают его уникальным в собственном классе. Он может наследовать общие свойства от своего родителя. Родительский класс, на основе которого строится новый класс, называется **базовым** классом (base class), а потомок – **производным** классом (derived class).

Базовый класс всегда является боле общим, чем производный. Производный класс использует члены базового класса, но может также изменять и дополнять их. Например, класс «кошки» может наследовать метод «спать» от класса «животные». При этом он может иметь собственную улучшенную версию метода «питаться».

Наследование исключительно важно, так как позволяет объектно-ориентированным программам расти по сложности линейно, а не геометрически. Производный класс наследует описание базового кода, делая ненужным повторную разработку и тестирование кода.

Буч определяет **полиморфизм** так:

Полиморфизм, polymorphism - положение теории типов, согласно которому имена (например, переменных) могут обозначать объекты разных (но имеющих общего родителя) классов. Следовательно, любой объект, обозначаемый полиморфным именем, может по-своему реагировать на некий общий набор операций.

Концепцию полиморфизма часто выражают фразой «один интерфейс, много методов».

Рассмотрим, например, стек. Может потребоваться программа, в которая требует три типа стеков. Один, например, будет использоваться для хранения целых чисел, второй – для чисел с плавающей точкой, третий – для символов. Алгоритм, который реализует стек – один и тот же, хотя хранимые данные различны. В не объектно-ориентированном языке программирования потребовалось бы создать три набора стековых функций, каждая из которых имела бы собственное имя. Полиморфизм позволяет специфицировать общий для всех типов данных набор стековых функций, использующих одно и то же имя. Это позволяет уменьшить сложность программирования. Забота компилятора – выбрать специфическое действие (т.е. метод) для его

использования в каждой конкретной ситуации. Программист не должен делать этот выбор «вручную». Он должен только помнить и использовать общий интерфейс.

Применение полиморфизма позволяет решить проблему добавления новой функциональности. Нетрудно представить себе ситуацию, когда нам может потребоваться еще один тип стека для хранения строковых данных. Используя полиморфизм, нетрудно добавить некоторый набор функций, позволяющий выполнять операции вталкивания и извлечения из стека строковых данных. При этом существующий программный код не подвергается никаким изменениям. Во-первых, новый код добавляется к уже существующему в разработанном классе. Не требуется отладка уже написанного ранее кода класса. Во-вторых, уже существующий код, использующий этот класс, будет оставаться работоспособным.

При правильном применении полиморфизм, инкапсуляция и наследование комбинируются так, что создают некую среду программирования, которая обеспечивает намного более устойчивые масштабируемые программы. Удачно спроектированная иерархия классов является базисом для повторного используемого кода. Инкапсуляция позволяет реализациям мигрировать во времени без разрушения кода, который зависит от public-интерфейса классов. Полиморфизм позволяет создавать ясный и читабельный код.

Хотя мы и рассмотрели принципы объектно-ориентированного программирования по отдельности, работают они вместе и практически не существуют отдельно друг от друга. Для того, чтобы разработать систему, необходимо создать некоторую иерархию классов. При этом базовый класс должен быть спроектирован с учетом возможного наследования. Ведь если базовый класс изменится, это повлечет за собой соответствующие изменения в функционировании производных классов. Понятно, что внутренняя структура как базового, так и производных классов должна быть скрыта от пользователя классов. Таким образом, совместно работают инкапсуляция и полиморфизм. Некоторые методы в производных классах могут уточняться (переопределяться). Имена этих методов будут одинаковы как в базовом, так и в производных классах. Таким образом, начинают совместно работать наследование и полиморфизм.

Рассмотрим один из примеров, который демонстрирует взаимодействие всех принципов объектно-ориентированного программирования и проектирования – автомобиль. При анализе нетрудно выделить некий базовый класс «автомобиль», который обладает всеми чертами автомобилей (наличие колес, коробки передач, рамы, двигателя и т.п.). Все производные классы («грузовой автомобиль», «легковой автомобиль») будут обладать всеми свойствами, присущими базовому классу. При вождении разных типов (производных классов) транспортных средств все водители полагаются на наследование. Понятно, что вне зависимости от типа водителю приходится иметь дело с рулем, тормозами, акселератором.

Люди постоянно взаимодействуют с инкапсулированными свойствами автомобиля. Нажимая на педаль газа, мы обычно не задумываемся о том, какие процессы в автомобиле происходят – нам нет дела до этих процессов. Мы можем и не знать, какие именно части автомобиля при этом оказываются задействованы. Таким образом, педаль тормоза можно считать очень простым интерфейсом взаимодействия автомобиля и водителя.

Полиморфизм ясно отражен в способности производителей автомобилей предлагать широкий набор возможностей для примерно одинаковых транспортных средств. Например, мы можем использовать различный набор резины (зимняя, летняя), использовать различные тормозные системы (например, противоблокировочную). В любом случае, если мы хотим остановиться, будем использовать педаль тормоза. То есть, один и тот же интерфейс может быть использован для управления различными реализациями.

Как можно видеть, именно через применение инкапсуляции, наследования и полиморфизма отдельные части трансформируются в объект, известный как автомобиль. То же самое справедливо и для компьютерных программ. Применяя объектно-ориентированные принципы, разные части сложной программы можно объединить вместе в форму взаимосвязанного, устойчивого и легко обслуживаемого целого.

Лекция 3.

Пример проектирования совокупности классов. Абстрактные типы данных.

Попробуем разобраться, как можно было бы спроектировать совокупность классов для представления списков.

Предположим, что для нас важно уметь записывать данные в списки, стеки и очереди. Каждая из этих структур данных достаточно хорошо известна, для каждой имеется набор операций, необходимый для работы.

Рассмотрим каждую из этих структур: как они должны быть устроена и какие услуги мы предоставляем пользователю этих структур.

Список – это последовательность элементов, имеющая начало. Для пользователей в принципе неважно, как внутри устроен список. Для него в первую очередь важна возможность хранения информации внутри этого списка. При этом список должен по возможности хранить любую информацию.

Для списка нам необходимо определить следующие основные операции:

- добавить новый узел в список в определенной позиции;
- удалить узел из списка;
- найти первый элемент списка;
- найти следующий элемент списка;
- подсчитать число элементов в списке;
- очистить список.

В качестве дополнительных услуг можно было бы предложить следующие операции:

- найти последний элемент списка;
- найти предыдущий элемент списка.

Исходя из возможностей, предоставляемых пользователю, можно представить себе три возможных структуры хранения данных:

1. Массив указателей на структуру, хранящую данные;
2. Классический однонаправленный список, в котором каждый узел имеет указатель на следующий узел и указатель на структуру хранения;
3. Классический двунаправленный список, в котором помимо указателя на следующий узел имеется указатель и на предыдущий.

Реализация класса с использованием первого варианта потребует более сложного программирования, так как операции по вставке и удалению узлов будут значительно затруднены из-за возможного сдвига в массиве достаточно большого количества указателей. Операции же по поиску данных в такой структуре выполняются очень эффективно. Использование второго или третьего вариантов дает возможность относительно простого выполнения операций вставки и удаления (особенно третий вариант). Возможности поиска будут резко ограничены.

Обратим также внимание, что реализация второго и третьего варианта требуют особой структуры хранения со своим специфическим набором операций. Таким образом, мы приходим к необходимости создания некоторого закрытого класса (класса, к которому имеется доступ только из класса «список») «узел».

Кроме структуры хранения типа «узел» нам могут потребоваться также некоторые вспомогательные поля:

- максимальное количество элементов в списке;
- текущее количество элементов в списке;
- указатель на первый элемент.

В зависимости от реализации нам может потребоваться хранение и указателя на последний элемент.

Стек – это последовательность элементов, которые могут добавляться и удаляться только с одного конца. Опять же для пользователей неважно, как внутри устроен стек. Важен набор операций над этим стеком. Мы можем себе представить его следующим образом:

- поместить элемент в стек;

- извлечь элемент из стека;
- определить количество элементов в стеке;
- очистить стек.

Очередь – это последовательность элементов, которые могут добавляться только с одного конца, а удаляться с другого. Заметим, что набор операций, определенных для стека и для очереди, идентичен. Единственное отличие – в операции «извлечь». Да и структура хранения для стека и очереди может быть одна и та же.

Таким образом, в случае стека и очереди мы можем определить один общий базовый класс (назовем его «упорядоченный список»), в котором каким-то образом определим структуру хранения и необходимый набор операций. Для каждого наследуемого класса нам останется только переопределить правильным образом операцию «извлечь».

Разберемся теперь со структурой хранения. Нетрудно заметить, что для хранения данных мы можем использовать описанный нами класс «список». Действительно, каждую из описанных для «упорядоченного списка» операций можно легко осуществить с помощью операций для списка:

Операция «поместить элемент» легко осуществляется с помощью операций «найти первый» и «добавить узел в список».

В зависимости от производного класса операция «извлечь элемент» будет соответствовать последовательности операций «найти первый» и «удалить узел» или «найти последний» и «удалить узел».

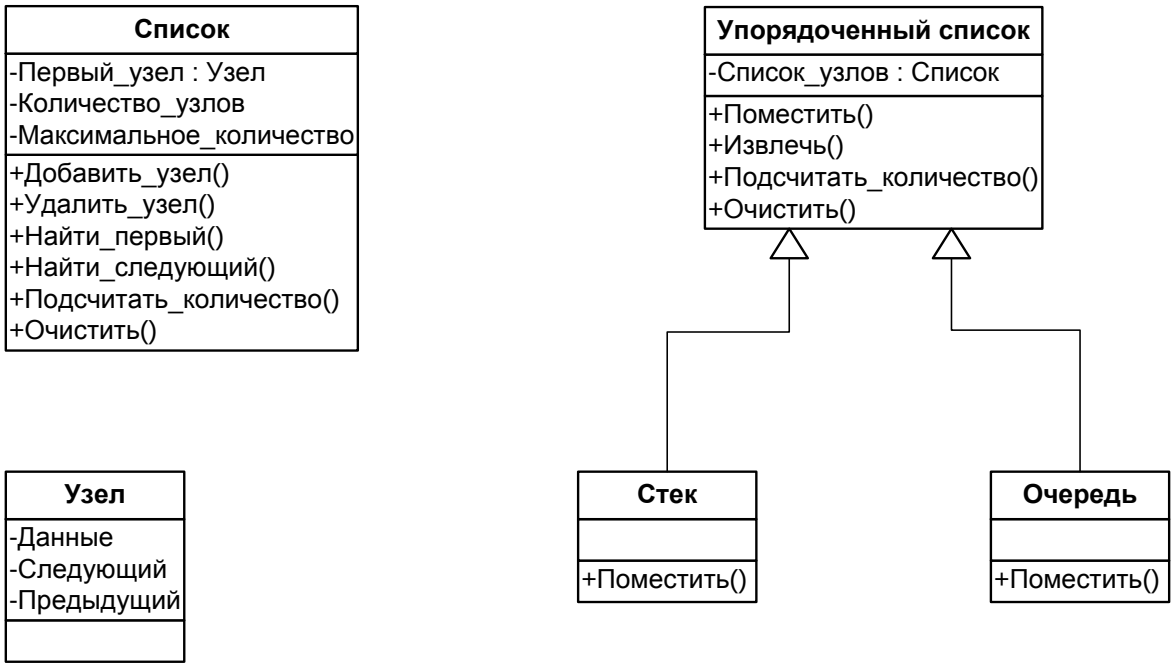
Операции «определить количество элементов» и «очистить» могут определяться через соответствующие методы класса «список».

Возникает вопрос: а может ли непосредственно использоваться класс «упорядоченный список»? По-видимому, ответ на этот вопрос должен быть отрицательным. Ведь в случае использования этого класса непонятно, как будет работать метод «извлечь элемент». В базовом классе эта функция никак не будет определена. Ее реализация возможна только в производном классе. Мы можем запретить использование пользователями этого класса.

Следует различать запрет на использование класса «узел» и класса «упорядоченный список». Класс «узел» - это наш дополнительный класс, необходимый для работы всех остальных классов. При этом внутри класса «список» мы спокойно можем создавать новые объекты типа «узел», пользоваться соответствующими методами этого класса для создания нужной нам структуры. Мы только ограничиваем извне доступ к этому классу.

Для класса «упорядоченный список» ситуация противоположная. Мы не ограничиваем доступ к этому классу. Но при этом мы должны запретить создание объектов такого класса. Действительно, не может быть создан объект, в котором хотя бы один из методов неопределен.

Таким образом, можно представить себе такую структуру классов:



Конечно, эта структура далеко не полна. В ней нет указаний, как осуществляется доступ к классу «узел». Мы не отметили, какие методы и поля являются открытыми, а какие – закрытыми. Мы не определили типы полей и типы возвращаемых методами значений. Нет и указания, какие параметры мы должны передавать в функции. Но данная схема уже позволяет начинать реализацию этих классов в одном из объектно-ориентированных языков программирования.

Вспомним, что объектно-ориентированное программирование (ООП) – это программирование, сфокусированное на данных, причем данные и методы их обработки (поведение) неразрывно связаны. Вместе данные и поведение представляют собой класс. ООП рассматривает вычисления как моделирование поведения. То, что моделируется, является объектами, представленными вычислительной абстракцией. В ООП объекты – это экземпляры класса.

Объектно-ориентированное программирование позволяет легко создавать и использовать абстрактные типы данных. Определим их следующим образом:

Абстрактный тип данных (АТД, abstract data type) – это определяемое пользователем расширение исходных типов языка. АТД состоит из набора значений и операций, которые могут влиять на эти значения.

Реализация АТД может быть осуществлена таким образом, что пользователь такого АТД будет обращаться с ними так же, как и с исходными типами данных языка. Рассмотрим, например, класс «деньги». Понятно, что в большинстве языков программирования такой класс отсутствует. Для данного класса определен свой набор операций:

- сложение;
- вычитание;
- умножение на действительное число

и др.

Естественно, пользователю языка хотелось бы иметь механизм, который позволял обращаться с этими типами данных как, например, с обычными числами:

```
currency a, b;
a=12;
b=13.2;
a+=b;
a*=1.05;
```

Использование ООП во многих случаях предоставляет такую возможность. Нетрудно заметить, что многие операции полиморфны по своей сути. Например, операция умножения производится для разных базовых типов данных: целых чисел, действительных и т.д. Используя возможность создания класса, используя полиморфные возможности языка, мы можем создать класс, в котором переопределим необходимые нам операции.

Лекция 4.

Классы и объекты в языке C++. Создание, уничтожение объектов и классы памяти.

В Лекции 1 уже давалось понятие о классах и объектах. Напомним, что класс – это некоторое множество объектов, имеющих общую структуру и общее поведение. Напомним также, что объектно-ориентированная программа представляет собой совокупность взаимодействующих объектов. Возникает вопрос – как же мы можем описать класс и создать объекты в конкретном объектно-ориентированном языке программирования.

Рассмотрим, как это делается в языке C++.

Класс в языке C++ - это агрегатный тип, который объединяет поля и методы в единую языковую конструкцию. Класс объявляется с использованием ключевого слова **class**. В простейшем виде класс описывается следующим образом:

```
class classname {
type variable1;
type variable2;
…
access_specifier:
type methodname1(parameter_list) {
      //тело метода
}
type methodname2(parameter_list) {
      //тело метода
}
      …
}
```

Приведем пример:

```
class pair
{
 int first;
 char second;
public:
 bool compare(pair c)
 {
      if (first==c.first && second==c.second)
           return true;
      else return false;
 }
 void setfirst(int f)
 {
      first=f;
 }
 void setsecond(char s)
 {
      second=s;
 }
};
```

Здесь описан некоторый класс с именем pair, который имеет два поля – first и second и несколько методов. При этом мы указываем, что все методы имеют доступ public, т.е. мы можем обращаться к нашим методам вне класса. Для полей не указан никакой спецификатор метода доступа. Дело в том, что по умолчанию, если спецификатор метода доступа не указан, предполагается доступ private. Поэтому указанная выше декларация идентична следующей:

```
class pair
{
private:
 int first;
 char second;
public:
…
};
```

Описанная выше декларация класса является удобной только в случае очень небольшого класса. Если же методов достаточно много, а сами они сложны и объемны, имеет смысл отделить описание класса от реализации. Это тем более важно, так как язык C++ допускает раздельную компиляцию модулей. Поэтому, например, описание класса может быть выделено в отдельный, заголовочный модуль, а реализация методов будет производиться в другом, основном модуле. Поэтому наш класс может быть записан следующим образом:

```
class pair
{
private:
 int first;
 char second;
public:
 bool compare(pair c);
 void setfirst(int f);
 void setsecond(char s);
};
bool pair::compare(pair c)
{
      if (first==c.first && second==c.second)
           return true;
      else return false;
}
void pair::setfirst(int f)
{
 first=f;
}
void pair::setsecond(char s)
{
 second=s;
}
```

Таким образом, для указания того, что некоторая функция или переменная являются частью класса, вводится **оператор разрешения области видимости**. Он имеет вид:

classname::variable

тем самым мы указываем, что функции compare, setfirst, setsecond принадлежат классу pair. Боле того, в одном и том же методе мы можем использовать переменные с одним именем, но принадлежащие разным областям видимости. Например, функцию setfirst можно было бы объявить следующим образом:

```
void pair::setfirst(int first)
{
 pair::first=first;
}
```

Здесь first – абсолютно разные переменные. Одна является полем класса pair, а вторая – передается в качестве параметра в метод класса.

Описание класса не означает создание объекта. Класс – это только некоторая схема, описание типа данных. Для создания объекта необходимо предпринимать дополнительные усилия, которые зависят от того, каким образом мы хотим создать наш объект и как мы его

будем использовать. Для того, чтобы разобраться в методах создания объектов, необходимо разобраться в **классах памяти**. Заметим, что классы памяти никоим образом не имеют отношения к классам как типам данных.

Объекты могут быть созданы в следующих формах:

- **глобальные объекты**. Создаются в начале создаваемой программы и разрушаются при ее завершении.
- **автоматические объекты**. Создаются, когда их объявление встречается в выполняемой программе, и разрушаются, когда блок программы, в котором они были объявлены, разрушается или удаляется из памяти.
- **статические объекты**. Создаются один раз при запуске программы и разрушаются тоже один раз при завершении выполнения программы.
- **Объекты в динамической памяти**. Создаются в процессе выполнения программы оператором **new** и разрушаются оператором **delete.**
- **Объекты-компоненты классов**. Создаются при построении объекта и разрушаются при разрушении этого объекта.

Рассмотрим на примере, каким образом создаются в программе автоматические и динамические объекты и как осуществляется доступ к методам объектов в том и другом случае.

```
void main()
{
 pair a;
 pair *b;
 a.setfirst(10);
 a.setsecond('q');
 b=new pair;
 b->setfirst(10);
 b->setsecond('q');
 if (a.compare(*b))
     cout<<"equal";
 else
     cout<<"different";
 a.setfirst(11);
 if (b->compare(a))
     cout<<"equal";
 else
     cout<<"different";
 delete b;
}
```

Данный пример демонстрирует два принципиально разных подхода. Объект *a* создается **автоматически** в начале программы (в данном случае) и будет уничтожен, когда программа закончит свою работу. В общем случае автоматические объекты создаются в блоке, в котором они объявлены и уничтожаются при выходе из этого блока. Объект *b* создается **динамически** тогда, когда будет выполнен оператор *new*. Уничтожение такого объекта происходит с использованием, когда будет выполнен оператор *delete*. Заметим, что и при создании и при уничтожении объекта мы используем **указатель** на объект.

По разному происходит и обращение к методам (и, если такое возможно, к полям) класса. Для автоматических объектов доступ к компонентам класса производится с помощью точечной нотации:

```
object.method
```

Именно такой доступ был продемонстрирован ранее при описании метода compare класса pair.

Для динамических объектов доступ производится с помощью указателя:

```
pointer->method
```

Лекция 5.

Статические члены класса. Конструкторы.

Статические переменные создаются один раз при запуске программы и разрушаются при завершении программы. Членам класса также могут быть статические переменные. Такая переменная создается только один раз, и все объекты данного класса используют этот экземпляр переменной.

Обратиться к статической переменной можно с помощью операции точка с использованием любого объекта данного класса. Однако этот способ не является предпочтительным. Более удобным является доступ с помощью операции разрешения видимости:

Имя_класса::идентификатор_статической_переменной

Доступ к статической переменной может осуществляться как через экземпляр класса, так и непосредственно, минуя применение конкретного экземпляра класса. Статическая переменная только описывается в определении класса. Для реального выделения памяти она должна быть описана еще раз во внешней части программы и без использования модификатора класса памяти static.

Статическими могут быть не только переменные, но и функции-члены класса. Экземпляр такой функции создается один раз и используется всеми экземплярами класса.

Рассмотрим программу, использующую статические переменные и методы класса.

```
#include <iostream.h>
class pair {
private:
  int first;
  char second;
  static int count;
public:
  bool compare(pair c);
  void setfirst(int first);
  void setsecond(char second);
  int getfirst() { return first;}
  char getsecond() {return second;}
  static void inccount() {count++;}
  static void deccount() {count--;}
  static int getcount();
};
bool pair::compare(pair c) {
        return (first==c.first && second==c.second);
  }
void pair::setfirst(int first) {
  pair::first=first;
}
void pair::setsecond(char second) {
  pair::second=second;
}
int pair::getcount() {
  return count;}
int pair::count=0;
void main() {
  pair a,b;
  a.setfirst(0);
  a.setsecond('a');
```

```
  pair::inccount();
  cout <<pair::getcount()<<endl;
  b.setfirst(1);
  b.setsecond('b');
  pair::inccount();
  cout <<pair::getcount()<<endl;
}
```

В этой программе мы использовали одну статическую переменную (смысл ее - в подсчете количества работающих в настоящий момент экземпляров данного класса) и три статические функции, позволяющие работать с этой переменной. Отметим, что данная программа не лишена недостатков, так как не связывает автоматически создание и уничтожение экземпляра класса с изменением счетчика членов класса. Такая связь может быть осуществлена с применением конструкторов и деструкторов.

Часто требуется инициализировать некоторые переменные – члены объекта при его создании. Для этого вида инициализации предназначены специальные инструменты. При определении класса можно определить функции-члены специального вида, называемые конструкторами. **Конструктор** – это функция-член класса, которая автоматически вызывается при объявлении объекта этого класса или при динамическом создании объекта. Конструктор создает значения типа своего класса. Этот процесс включает в себя инициализацию членов данных и зачастую распределение свободной памяти.

В языке C++ конструкторы определяются так же, как и любые другие функции за исключением двух моментов:

- конструктор должен иметь то же имя, что и сам класс
- конструктор не может возвращать значение. Более того, в прототипе или заголовке определения конструктора никакой тип (даже void) не указывается.

В следующем примере продемонстрирован конструктор, который выполняет следующие действия:

- выделяет память под строку символов
- присваивает начальные значения переменным – членам класса
- увеличивает счетчик числа экземпляров класса на 1.

Кроме того, в программе продемонстрирована работа деструктора, назначение и описание которого будет рассмотрено позже.

```
#include <iostream.h>
#include <cstring>
class s_pair {
private:
  int first;
  char* second;
  static int count;
public:
  s_pair(int first, char* string);
  ~s_pair();
  void setfirst(int first) {s_pair::first=first;}
  int getfirst() { return first;}
  void setsecond(char* string) {
        int len=strlen(string);
        delete [] second;
        second=new char[len+1];
        strcpy(second,string);
  }
  char* getsecond() {
        char* str=new char[strlen(second)+1];
        strcpy(str,second);
```

```
        return str;
  }
  static int getcount() {return count;}
};
int s_pair::count=0;
s_pair::s_pair(int first,char* string) {
  s_pair::first=first;
   int len=strlen(string);
  second=new char[len+1];
  strcpy(second,string);
  count++;
}
s_pair::~s_pair() {
  count--;
  delete [] second;
}

void main() {
  s_pair a(0,"it is string"), *b;
  cout <<a.getfirst()<<" "<<a.getsecond()<<endl<<"count="<<s_pair::getcount()<<endl;
  b=new s_pair(1,"it is outher string");
  cout<<b->getfirst()<<" "<<b->getsecond()<<endl<<"count="<<s_pair::getcount()<<endl;
  delete b;
  cout <<"count="<<s_pair::getcount()<<endl;
}
```

Вывод этой программы:

```
0 it is string
count=1
1 it is outher string
count=2
count=1
```

Существует специальный синтаксис для инициализации полей объекта с помощью конструктора. Инициализаторы конструктора для членов класса могут быть заданы через запятую в списке, который идет после двоеточия за списком параметров и предшествует телу. При применении инициализатора наш конструктор примет вид:

```
s_pair::s_pair(int first,char* string) : first(first){
  int len=strlen(string);
  second=new char[len+1];
  strcpy(second,string);
  count++;
}
```

При использовании инициализатора отпала необходимость в присваивании. Заметим, что инициализация, если она возможна, предпочтительней присваивания.

Для одного класса может быть определено несколько конструкторов. Все они должны иметь одно и то же имя, совпадающее с именем класса и различаться в аргументах. C++ предоставляет такую возможность, используя перегрузку функций. Определение нескольких функций с одним именем называется перегрузкой имени функции. При перегрузке функции различные определения должны либо содержать разное количество параметров, либо иметь различия в типах параметров. При обращении к функции компилятор использует то определение функции, количество и тип формальных параметров которого совпадают с количеством и типом аргументов в обращении функции.

Лекция 6.

Конструктор по умолчанию. Копирующий конструктор. Деструктор.

Как упоминалось в предыдущей лекции, C++ позволяет определить несколько конструкторов. Среди них особое место занимают два – конструктор по умолчанию и копирующий конструктор. Важность их связана с тем, что именно эти два конструктора предоставляет компилятор в отсутствии определения конструкторов класса.

Конструктор по умолчанию – это конструктор, не требующий аргументов. Это может быть конструктор с пустым списком аргументов, или конструктор, у которого все аргументы имеют значения по умолчанию.

Ранее демонстрировались примеры классов, в которых конструктор явно не определен. Однако это не означает, что для данного класса конструктор вообще отсутствует. Компилятор предоставляет в таком случае свой конструктор по умолчанию, основные действия которого сводятся к выделению памяти под члены класса. Но если требуется произвести какие-либо дополнительные действия (выделение динамической памяти, увеличение счетчика экземпляров класса и т.п.), необходимо явно определить конструктор по умолчанию.

Заметим, что если в программе явно определен хотя бы один конструктор, компилятор не предоставляет свой конструктор по умолчанию. В таком случае возникают проблемы с размещением массива переменных типа данного класса. Поэтому, если в процессе проектирования класса предполагается, что может быть создан массив экземпляров класса, конструктор по умолчанию должен быть явно определен.

Для класса c_pair из предыдущей лекции конструктор по умолчанию может быть определен следующим образом:

```
s_pair:s_pair() : first(0) {
  second=new char;
  *second='\0';
  count++;
}
```

Данный конструктор не имеет параметров, а переменные first и second инициализирует 0 и пустой строкой соответственно. При этом увеличивается счетчик числа экземпляров класса.

Вторым важным видом конструкторов является **копирующий конструктор**. Как и конструктор по умолчанию, копирующий конструктор предоставляется компилятором. Однако в отличие от конструктора по умолчанию, даже если в классе имеется явно определенный конструктор, но нет определения копирующего конструктора, компилятор предоставляет свой копирующий конструктор (это утверждение справедливо не для всех компиляторов).

Сигнатура копирующего конструктора такова:

имя_класса::имя_класса(const имя_класса& source);

Когда же используется копирующий конструктор? Самое простое, явное и не главное его применение – инициализация одного экземпляра класса с помощью другого. Наиболее важным является его использование, когда экземпляр класса передается по значению в качестве аргумента функции или возвращается в качестве значения функции.

Если в программе определена функция, имеющая в качестве аргумента экземпляр класса, компилятор использует копирующий конструктор для создания локальной копии экземпляра класса в функции. Если копирующий конструктор не определен, компилятор использует свой копирующий конструктор, который инициализирует члены класса один за другим. Проблема может возникнуть, если такой класс в качестве члена имеет указатель. В таком случае копирующий конструктор, предоставляемый компилятором, не будет выделять память под динамические переменные, а только перепишет ссылки. Таким образом, в локальной копии ссылки будут указывать на ту же область памяти, что и в источнике, что может нарушить нормальную работу программы. Поэтому при определении класса желательно определять явно копирующий конструктор.

Ниже рассматривается пример, в котором явно определены и конструктор по умолчанию, и копирующий конструктор. Заметим, что явно вызов копирующего конструктора не происходит. Компилятор использует его в тех случаях, когда происходит вызов функции, использующей в качестве аргумента объект данного класса.

```
#include <iostream.h>
#include <cstring>
class s_pair {
private:
  int first;
  char* second;
  static int count;
public:
  s_pair(int first, char* string);
  s_pair();
  s_pair(const s_pair& source);
  ~s_pair();
  void setfirst(int first) {s_pair::first=first;}
  int getfirst() { return first;}
  void setsecond(char* string) {
        int len=strlen(string);
        delete [] second;
        second=new char[len+1];
        strcpy(second,string);
  }
  void getsecond(char* string) {
        strcpy(string,second);
  }
  static int getcount() {return count;}
};
int s_pair::count=0;
s_pair::s_pair(int first,char* string) : first(first){
  int len=strlen(string);
  second=new char[len+1];
  strcpy(second,string);
  count++;
}
s_pair::s_pair() : first(0) {
  second=new char;
  *second='\0';
  count++;
}
s_pair::s_pair(const s_pair& source): first(source.first) {
  int len=strlen(source.second);
  second=new char[len+1];
  strcpy(second,source.second);
  count++;
}
s_pair::~s_pair() {
  count--;
  delete [] second;
  second=0;
}
void print(s_pair element) {
```

```
  char str[20];
  element.getsecond(str);
  cout <<element.getfirst()<<" "<<str<<endl;
}
void main() {
  s_pair a(0,"it is string"), *b;
  cout <<"count="<<s_pair::getcount()<<endl;
  print(a);
  s_pair c[2];
  cout <<"count="<<s_pair::getcount()<<endl;
  print(c[0]);
  print(c[1]);
  b=new s_pair(1,"it is outher string");
  cout <<"count="<<s_pair::getcount()<<endl;
  b->setsecond("replace");
  print(*b);
  delete b;
  b=0;
  cout <<"count="<<s_pair::getcount()<<endl;
}
```

Вывод этой программы будет следующим:

```
count=1
0 it is string
count=3
0
0
count=4
1 replace
count=3
```

Предположим теперь, что копирующий конструктор отсутствует (заметим, что не все компиляторы позволяют это делать). Немного изменим функцию main для демонстрации эффектов, возникающих при отсутствии копирующего конструктора:

```
void main() {
  char str[20];
  s_pair a(12,"it is string"), *b;
  cout <<"count="<<s_pair::getcount()<<endl;
  print(a);
  a.getsecond(str);
  cout<<a.getfirst()<<" "<<str<<endl;
  s_pair c[2];
  cout <<"count="<<s_pair::getcount()<<endl;
  print(c[0]);
  print(c[1]);
  b=new s_pair(1,"it is outher string");
  cout <<"count="<<s_pair::getcount()<<endl;
  b->setsecond("replace");
  print(*b);
  delete b;
  b=0;
  cout <<"count="<<s_pair::getcount()<<endl;
}
```

Вывод: этой программы таков:
count=1
12 it is string
12 _____0
count=2
0
0
count=1
1 replace

После этого в программе возникает ошибка и программа аварийно заканчивает работу. Как мы видим, в программе неверно подсчитывается количество экземпляров класса и перестают работать указатели на строки. Связано это с тем, что по окончании работы функции print локальные копии уничтожаются. Производит это так называемый деструктор. А при отсутствии копирующего конструктора, как уже говорилось ранее, создаются копии указателей. При вызове деструктора уничтожаются не только локальные копии, но и информация, на которую указывают данные указатели.

В описанной выше программе присутствует еще одна функция-член – деструктор. **Деструктор** представляет собой начинающееся с тильды имя класса. Деструктор предназначен для уничтожения объектов данного класса. Как и конструкторы, деструктор не имеет возвращаемого значения. Заметим, что деструктор не имеет аргументов и не может быть перегружен.

Вызов деструктора происходит явно, когда применяется оператор delete и неявно, когда программа выходит за область видимости объекта (при выходе из блока или функции, в которой объявлен объект).

В описанной выше программе деструктор выполняет следующие действия:

- освобождает динамически выделенную память
- уменьшает счетчик числа экземпляров класса на 1

Лекция 7.

Перегруженные методы. Дружественные функции.

Конкретная функция при перегрузке выбирается в зависимости от соответствия списка аргументов при вызове функции списку параметров в объявлении функции. Когда выбирается перегруженная функция, компилятор должен иметь алгоритм для выбора надлежащей функции. В общем, этот алгоритм состоит из следующих шагов:

1. Использовать строгое соответствие (если возможно). Этот вариант является предпочтительным, так как подразумевает, что список аргументов в точности совпадает со списком параметров.

2. Попробовать стандартное повышение типа. Повышение – это, например, преобразование float в double, а также bool, char, short в int.

3. Попробовать стандартное преобразование типа. К ним, например, относится преобразование указателей.

4. Попробовать определяемое пользователем преобразование.

5. Использовать, если возможно, соответствие эллипсису (…)

Рассмотрим, что такое собственное преобразование типов. Оно может быть сделано для пользовательских типов данных. Нельзя переопределить стандартные преобразования типов данных. Для определения преобразования одного типа данных в другой используются конструкторы с единственным параметром. Например, в приведенном ниже примере конструктор с одним параметром позволяет автоматически преобразовывать тип double к типу ruMoney.

```
#include <iostream.h>
#include <cstring>
class ruMoney {
private:
long allcop;
public:
ruMoney(long rub, int cop);
ruMoney() {allcop=0;}
ruMoney(double money);
long getrub() {return static_cast<long>(allcop*0.01);}
int getcop() {return allcop-100*static_cast<long>(0.01*allcop);}
};
ruMoney::ruMoney(long rub, int cop) {
allcop=100*rub+cop;
}
ruMoney::ruMoney(double money) {
allcop=static_cast<long>(100*money);
}
void main() {
ruMoney a(11,0), c;
cout<<a.getrub()<<" rub "<<a.getcop()<<" cop\n";
c=11.01;
cout<<c.getrub()<<" rub "<<c.getcop()<<" cop\n";
}
```

Вывод этой программы:

```
11 rub 0 cop
11 rub 1 cop
```

Существуют случаи, когда было бы желательно функциям - не членам класса предоставить доступ к скрытым членам класса. Это возможно для так называемых **дружественных** функций. Для описания дружественной функции она должна быть объявлена внутри класса со спецификатором *friend*. Функция может быть описана в любой части класса, как закрытой, так и открытой. Это не влияет на ее сущность, так как она не принадлежит классу. Естественно, и доступ к такой функции осуществляется как к обычной функции (без точки).

Одна из причин использования дружественных функций состоит в том, что она нуждается в привилегированном доступе к более чем одному классу. Рассмотрим пример. Пусть в дополнение к уже имеющемуся классу ruMoney определен класс usMoney и требуется написать функцию, которая позволяет сравнивать значения этих классов.

```
#include <iostream.h>
#include <cstring>
class usMoney;
class ruMoney {
private:
long allcop;
public:
friend int compare(ruMoney rus, usMoney amr, double kurs);
ruMoney(long rub, int cop);
ruMoney() {allcop=0;}
ruMoney(double money);
long getrub() {return static_cast<long>(allcop*0.01);}
int getcop() {return allcop-100*static_cast<long>(0.01*allcop);}
};
ruMoney::ruMoney(long rub, int cop) {
allcop=100*rub+cop;
}
ruMoney::ruMoney(double money) {
allcop=static_cast<long>(100*money);
}
class usMoney {
private:
long allcent;
public:
friend int compare(ruMoney rus, usMoney amr, double kurs);
amMoney(long dol, int cent);
amMoney() {allcent=0;}
amMoney(double money);
long getdol() {return static_cast<long>(allcent*0.01);}
int getcent() {return allcent-100*static_cast<long>(0.01*allcent);}
};
usMoney:: usMoney (long dol, int cent) {
allcent=100*dol+cent;
}
usMoney:: usMoney (double money) {
allcent=static_cast<long>(100*money);
}
int compare(ruMoney rus, usMoney amr, double kurs) {
if (rus.allcop<amr.allcent*kurs)
return -1;
else if(rus.allcop*kurs==amr.allcent)
return 0;
else
```

```
return 1;
}

void main() {
ruMoney a(32,95);
double kurs=29.73;
usMoney b=1.11;
cout<<a.getrub()<<" rub "<<a.getcop()<<" cop\n";
cout<<b.getdol()<<" dol "<<b.getcent()<<" cent\n";
cout<<compare(a,b,kurs)<<endl;
}
```

Вывод этой программы

```
32 rub 95 cop
1 dol 11 cent
-1
```

Заметим, что дружественная функция, имеющая доступ к обоим классам, в обоих классах и должна быть определена как дружественная.

Конечно, можно было бы написать функцию, выполняющую те же самые действия, но не являющуюся дружественной, используя при этом определенные в классе функции доступа. Однако можно заметить, что в нашем случае такая функция была бы крайне неэффективной, потому что для сравнения потребовался бы перевод во-первых, из внутреннего представления во внешний (это делают функции доступа), а во-вторых, обратный перевод в некоторое представление, позволяющее сравнивать значения. Для дружественной функции мы использовали только внутреннее представление без всяких дополнительных преобразований.

Другая причина использования дружественных функций – перегрузка операторов. Язык C++ позволяет перегружать почти все операторы. Это сделано для того, чтобы пользователь класса мог использовать стандартный вид операций, определенный для стандартных типов данных. Логично, например, определить операции +, *, - и т.д. для класса денег.

В программе выше использовался стандартный вид вывода для классов ruMoney и usMoney. Логично было бы переопределить функцию вывода так, чтобы она могла выводить объект нашего класса. Это делается с помощью дружественной функции. Наша программа примет вид:

```
#include <iostream.h>
#include <cstring>
class usMoney;
class ruMoney {
private:
long allcop;
public:
friend int compare(ruMoney rus, usMoney amr, double kurs);
friend ostream& operator<<(ostream& out, ruMoney rus);
ruMoney(long rub, int cop);
ruMoney() {allcop=0;}
ruMoney(double money);
long getrub() {return static_cast<long>(allcop*0.01);}
int getcop() {return allcop-100*static_cast<long>(0.01*allcop);}
};
ruMoney::ruMoney(long rub, int cop) {
allcop=100*rub+cop;
}
ruMoney::ruMoney(double money) {
```

```
allcop=static_cast<long>(100*money);
}
ostream& operator<<(ostream& out, ruMoney rus) {
return (out<<rus.getrub()<<" rub "<<rus.getcop()<<" cop");
}
class usMoney {
private:
long allcent;
public:
friend int compare(ruMoney rus, usMoney amr, double kurs);
friend ostream& operator<<(ostream& out, usMoney amr);
usMoney (long dol, int cent);
usMoney () {allcent=0;}
usMoney (double money);
long getdol() {return static_cast<long>(allcent*0.01);}
int getcent() {return allcent-100*static_cast<long>(0.01*allcent);}
};
usMoney:: usMoney (long dol, int cent) {
allcent=100*dol+cent;
}
usMoney:: usMoney (double money) {
allcent=static_cast<long>(100*money);
}
ostream& operator<<(ostream& out, usMoney amr) {
return (out<<amr.getdol()<<" dol "<<amr.getcent()<<" cent");
}
int compare(ruMoney rus, usMoney amr, double kurs) {
if (rus.allcop<amr.allcent*kurs)
return -1;
else if(rus.allcop*kurs==amr.allcent)
return 0;
else
return 1;
}

void main() {
ruMoney a(32,95);
double kurs=29.73;
usMoney b=1.11;
cout<<a<<endl<<b<<endl;
cout<<compare(a,b,kurs)<<endl;
}
```

Вывод ее останется неизменным.

Лекция 8.

Наследование. Открытое и закрытое наследование.

Наследование – это механизм, позволяющий получить новый класс на основе существующего. Существующий класс может быть изменен или дополнен для создания нового класса. Класс, на основе которого создается новый класс, называется базовым (суперклассом). Наследуемый класс называется производным (или подклассом).

Буч определяет наследование следующим образом:

*Наследование - это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (**одиночное наследование**) или других (**множественное наследование**) классов.*

Наследование есть процесс, с помощью которого один объект приобретает свойства другого объекта. При использовании наследования объект нуждается в определении только тех качеств, которые делают его уникальным в собственном классе. Он может наследовать общие свойства от своего родителя. Поэтому именно механизм наследования дает возможность одному объекту быть специфическим экземпляром общего случая. Наследование взаимодействует также с инкапсуляцией. Если данный класс инкапсулирует некоторые атрибуты, то любой подкласс будет иметь те же атрибуты плюс атрибут, который он добавляет как часть своей специализации. Эта ключевая концепция, которая позволяет объектно-ориентированным программам расти по сложности линейно, а не в геометрически. Новый подкласс наследует все атрибуты всех его предков. Он не имеет непредсказуемых взаимодействий с большинством остальных кодов в системе.

Класс можно сделать производным от существующего с использованием следующей формы:

```
class имя_класса : (public | protected | private) имя_базового_класса {
объявления членов класса
};
```

Ключевые слова public, protected, private используются для указания того, насколько члены базового класса будут доступны из производного. Использование в заголовке производного класса public означает, защищенные и открытые члены базового класса должны наследоваться как защищенные и открытые члены производного класса. Это означает, что и защищенные (*protected*), и открытые (*public*) члены базового класса доступны в производном классе и также являются защищенными и закрытыми. Закрытые (*private*) члены базового класса недоступны для производного класса. Заметим, что закрытые и защищенные члены базового класса недоступны из классов, не являющихся производными базового. Открытое наследование, называемое также интерфейсным наследованием, означает, что производный тип является подтипом базового (отношение ISA). Каждый объект производного класса является объектом базового типа.

Например, класс млекопитающих является базовым для класса собак, класс собак – базовым для класса болонок. Т.е. каждая болонка – собака, каждая собака – млекопитающее, но не наоборот.

Пример открытого одиночного наследования:

```
#include <iostream.h>
class rectangle {
protected:
  float x,y,height,width;
public:
  rectangle(float xbase, float ybase, float h, float w):
    x(xbase), y(ybase), height(h), width(w) { }
```

```
    float area() { return height*width; }
  void printarea() {
          cout<<"rectangle area="<<area()<<endl;
  }
};
class colorsquare : public rectangle {
private:
  long color;
public:
  colorsquare(float xbase, float ybase, float h, long c):
    rectangle(xbase,ybase,h,h), color(c) { }
    long getcolor() {return color;}
    void printarea() {
            cout<<"square area="<<area()<<endl;
    }
};
void main() {
  rectangle a(0,0,10,20),*refa;
  refa=&a;
  refa->printarea();
  colorsquare b(0,0,10,0xffffffff), *refb;
  refb=&b;
  refb->printarea();
  refa=refb;
  refa->printarea();
  //refa->getcolor();
}
```

Результат работы программы:

```
rectangle area=200
square area=100
rectangle area=100
```

В этом примере мы создаем базовый класс *rectangle*, имеющий защищенные члены класса, определяющие параметры прямоугольника. Заметим, что мы объявили их защищенными, а не закрытыми именно с той целью, чтобы они были доступны для производного класса и не доступны извне. Класс *colorsquare* (цветной квадрат) является подтипом базового класса. Это означает, что производный класс наследует все доступные (открытые и защищенные) свойства и методы базового класса. В дополнение производный класс добавлено свойство *color* и метод, позволяющий с этим свойством работать. В примере видно, что конструктор базового класса используется конструктором производного класса как часть списка инициализации. Это понятно, ведь объект класса *colorsquare* является одновременно объектом класса *rectangle*.

В производном классе мы можем переопределить методы базового класса. В качестве примера мы переопределяем функцию *printarea* – для разных классов эта функция выдает разные сообщения.

При открытом наследовании указатель на базовый класс может быть также указателем на производный класс. В программе мы создаем два указателя – один на базовый класс *rectangle* и другой – на производный класс *colorsquare*. Правило преобразования указателей состоит в том, что указатель на открытый производный класс может быть неявно преобразован к указателю на его базовый класс. Присваивая указатель на производный класс указателю на базовый класс, не возникает ошибки. Компилятор связывает вызов функции с тем ее вариантом, который отвечает классу, указанному при объявлении указателя, а не тому, на объект которого в данный момент направлен указатель. Фактически нам становится доступен объект базового класса,

образованный в процессе создания объекта производного класса. Заметим, что методы и свойства производного класса с помощью такого указателя недоступны.

В дальнейшем будет показано, что необходимо сделать для того, чтобы выбор вызываемой функции определялся на этапе выполнения в зависимости от того, на что направлен указатель.

Язык C++ позволяет создавать классы одновременно на основе нескольких базовых классов (множественное наследование). Заметим, что это позволяют не все объектно-ориентированные языки программирования. Например, Java не позволяет использовать множественное наследование.

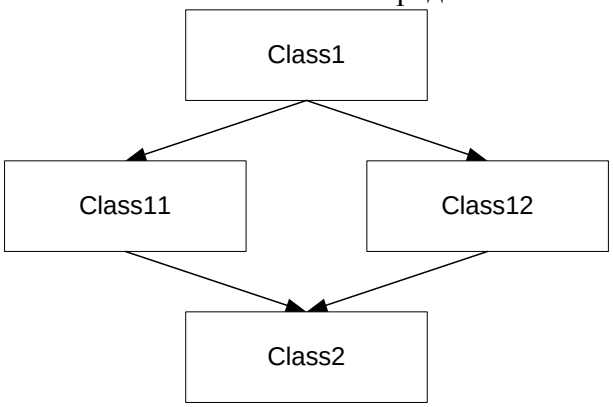
В случае множественного наследования класс определяется следующим образом:

class имя_класса : (public | protected | private) имя_базового_класса1, (public | protected | private) имя_базового_класса2, … {

объявления членов класса

};

Использование множественного наследования может создавать неоднозначности. Например, можно представить себе следующую ситуацию: *Class1* является базовым для классов *Class11* и *Class12*. Они в свою очередь являются базовыми для класса *Class2*.



Предположим теперь, что в классах *Class11* и *Class12* переопределяется некоторый метод *method1* базового класса *Class1*. Возникает вопрос, какой же экземпляр метода будет использоваться в классе *Class2*? Решить эту проблему можно, добавляя в качестве префиксов имя класса-источника: *Class12::method1*.

Закрытое наследование не носит характера отношения подтипов. При закрытом наследовании мы повторно используем код базового класса, но не предполагаем рассматривать объекты производного класса как объекты базового. Закрытое наследование называется отношением LIKEA, или наследованием реализации. Закрытое и защищенное наследование не создает иерархии типов. Поэтому, если класс *colorsquare* будет закрытым наследником класса *rectangle*, преобразования указателей невозможны. В основном такое наследование используется для повторного использования кода.

Практически закрытое наследование не используется.

Лекция 9.

Виртуальные функции. Абстрактные классы.

Как было продемонстрировано ранее, перегруженная функция-член вызывается с учетом соответствия типов. Компилятор связывает вызов функции с тем ее вариантом, который соответствует классу, указанному при объявлении указателя, а не тому, на объект которого направлен указатель. В языке C++ имеется возможность динамически (в процессе выполнения) выбирать перегруженную функцию-член среди функций базового и производного классов. Рассмотрим несколько измененный вариант программы из предыдущей лекции.

```
#include <iostream.h>
class rectangle {
protected:
  float x,y,height,width;
public:
  rectangle(float xbase, float ybase, float h, float w):
    x(xbase), y(ybase), height(h), width(w) { }
    float area() { return height*width; }
  virtual void printarea() {
          cout<<"rectangle area="<<area()<<endl;
  }
};
class colorsquare : public rectangle {
private:
  int color;
public:
  colorsquare(float xbase, float ybase, float h, int c):
    rectangle(xbase,ybase,h,h), color(c) { }
    int getcolor() {return color;}
    void printarea() {
            cout<<"square area="<<rectangle::area()<<endl;
    }
};
void main() {
  rectangle a(0,0,10,20),*refa;
  refa=&a;
  refa->printarea();
  colorsquare b(0,0,10,0xffffffff), *refb;
  refb=&b;
  refb->printarea();
  refa=refb;
  refa->printarea();
}
```

Единственное отличие – в объявлении функции *printarea* базового класса. Ключевое слово **virtual** служит спецификатором функции и как раз предоставляет механизм динамического выбора перегруженных функций. Результат работы данной программы следующий:

```
rectangle area=200
square area=100
square area=100
```

Как видно, в отличие от предыдущего результата, выбор функции *printarea* зависит не от того, указатель какого класса указывает на данный объект, а от класса самого объекта.

Виртуальность является частью полиморфизма. Механизм виртуальных функций основан на позднем (динамическом) связывании (раннее – связывание на этапе компиляции). Если в некотором классе имеется хотя бы одна виртуальная функция, то все объекты этого класса содержат указатель на связанную с их классом виртуальную таблицу указателей функций этого класса. Доступ к виртуальной функции осуществляется посредством косвенной адресации – через этот указатель и соответствующую таблицу. Тем самым использование виртуальных функций снижает быстродействие и увеличивает размер объектов класса.

При отсутствии функции-члена производного типа по умолчанию используется виртуальная функция базового класса. В нашем примере при отсутствии определения функции *printarea* в классе *colorsquare* всегда будет вызываться эта же функция базового класса.

Спецификатор **virtual** может применяться только к нестатическим функциям-членам. Конструкторы не могут быть виртуальными. Виртуальность наследуется. Если функция в базовом классе объявлена как виртуальная, то функция в производном классе также будет виртуальной. При этом нет необходимости в производном классе еще раз объявлять ее как виртуальную.

Деструкторы могут быть виртуальными. Если класс имеет хотя бы одну виртуальную функцию, рекомендуется деструкторы также объявлять виртуальными.

Дополним базовый класс rectangle деструктором:

```
virtual ~rectangle() {cout<<"in rectangle\n";}
```

Также добавим деструктор в производный класс (виртуальность в нем объявлять уже нет необходимости):

```
~colorsquare() {cout<<"in colorsquare\n";}
```

Определим главную функцию таким образом, чтобы продемонстрировать динамическое создание и уничтожение объектов с помощью операторов *new* и *delete*:

```
void main() {
  rectangle *refa;
  refa=new rectangle(0,0,10,20);
  colorsquare *refb;
  refb=new colorsquare(0,0,10,0xffffffff);
  delete refa;
  refa=refb;
  delete refa;
}
```

Результат работы программы:

```
in rectangle
in colorsquare
in rectangle
```

Видно, что при уничтожении объекта *rectangle* вызывается его деструктор, а при уничтожении объекта *colorsquare* – его деструктор, который в свою очередь вызывает деструктор базового класса. В ситуации, когда деструктор не объявлен виртуальным, результат работы программы изменится:

```
in rectangle
in rectangle
```

Здесь при уничтожении объекта производного класса вызывается деструктор только базового класса.

Иерархия типов обычно имеет корневой класс, содержащий некоторое число виртуальных функций. При этом они в большинстве случаев являются фиктивными функциями. Они имеют пустое тело в корневом классе, но в производных классах этим функциям придают смысл (в терминологии ООП это называется отложенным методом). Для таких функций в C++ введено понятие «**чисто виртуальные функции**» – это виртуальные функции, тело которых не определено. Объявляются они следующим образом:

virtual прототип_функции=0;

Класс, имеющий хотя бы одну виртуальную функцию, называется **абстрактным** классом. Для абстрактных классов нельзя создавать объекты. Они используются, во-первых, чтобы описать интерфейс без конкретной реализации, и, во-вторых, для объявления указателей, имеющих доступ к объектам производных классов.

Если абстрактная функция не определена в производном классе, он также является абстрактным. Приведем пример:

```
class shape {
protected:
  float dim1,dim2;
public:
  virtual float area()=0;
  shape(float d1,float d2) : dim1(d1), dim2(d2) { }
};
class rectangle : public shape{
protected:
  float x,y;
public:
  rectangle(float xbase, float ybase, float h, float w):
    shape(w,h), x(xbase), y(ybase) { }
    float area() { return dim1*dim2; }
  virtual void printarea() {
          cout<<"rectangle area="<<area()<<endl;
  }
  virtual ~rectangle() {cout<<"in rectangle\n";}
};
```

Мы использовали предыдущую программу со следующими изменениями:

- добавили абстрактный класс *shape*, имеющий виртуальную функцию *area* и конструктор
- изменили класс *rectangle*, сделав его производным от *shape*. При этом изменился конструктор класса и по-новому определена функция *area*.

Если бы мы не определили функцию *area* в производном классе, он так и остался бы быть абстрактным.

Лекция 10.

Шаблоны функций. Шаблонные классы.

Часто приходится иметь дело с случаем, когда несколько функций должны выполнять схожие действия с данными разных типов. В таких случаях мог бы быть использован механизм перегруженных функций, т.е. мы могли бы написать несколько вариантов одной и той же функции с различными значениями аргументов. Это применимо, если алгоритмы обработки для разных типов данных несколько отличаются. А как быть, если независимо от типа аргументов функции должны выполнять одни и те же действия? Например, копирование одного массива в другой, обмен значениями переменных и т.д. выполняются одинаково для разных типов данных. В таких случаях в языке C++ предусмотрен механизм шаблонов. Определение шаблонных функций делает возможным повторное использование кода безопасным с точки зрения типов образом, что позволяет компилятору автоматизировать процесс инстанцирования типа (фактический тип заменяет тип-параметр, присутствующий в коде шаблона). Механизм шаблонов обеспечивает параметрический полиморфизм.

Шаблон функции определяется следующим образом:

template<шаблонные_аргументы> объявление

Шаблонными аргументами служат

class идентификатор

и/или

объявление_аргумента

Аргументы *class идентификатор* инстанцируются типом. Другие аргументы объявления инстанцируются постоянными выражениями не с плавающей точкой и могут быть функцией или адресом объекта с внешней компоновкой. Например:

template<class T,int n>

(Заметим, что *n* должна быть использована в объявлении).

Шаблон может быть написан с использованием параметров по умолчанию (только для классов):

template<class T=double, int n=100>

Аргументом шаблона может быть другой шаблон.

Рассмотрим пример определения шаблонной функции:

```
#include <iostream.h>
#include <string.h>
#define MAX_LEN 255
template<class T>
void shiftleft(T a[],int start, int end) {
  cout<<"in template shiftleft"<<endl;
  T temp;
  temp=a[start];
  for (int i=start; i<end; i++)
        a[i]=a[i+1];
  a[end]=temp;
}
void shiftleft(char a[][MAX_LEN],int start, int end) {
  cout<<"in string shiftleft"<<endl;
  char temp[MAX_LEN];
  strcpy(temp,a[start]);
  for (int i=start; i<end; i++)
        strcpy(a[i],a[i+1]);
  strcpy(a[end],temp);
}
void main() {
  int i;
```

```
  char str[]="abcdef";
        char words[][MAX_LEN]=
{"word00","word01","word02","word03","word04","word05"};
  int dim[]={0,1,2,3,4,5};
  shiftleft(str,0,2);
  cout<<str<<endl;
  shiftleft(dim,0,2);
  for (i=0;i<6;i++)
        cout<<dim[i]<<' ';
  cout<<endl;
  shiftleft(words,0,2);
  for (i=0;i<6;i++)
        cout<<words[i]<<' ';
  cout<<endl;
}
```

В данном примере определен шаблон функции *shiftleft*, которая сдвигает циклически влево элементы массива с номерами от *start* до *end*. Такая функция может работать с различными типами данных – *char, int, double* и т.д. Но эта функция не может использоваться, например, при работе с массивами строк. Для придания ей такой функциональности необходимо перегрузить функцию и для массивов строк.

Результат работы программы:

```
in template shiftleft
bcadef
in template shiftleft
1 2 0 3 4 5
in string shiftleft
word01 word02 word00 word03 word04 word05
```

Видно, что шаблон функции вызывается два раза для различных типов данных: в первый раз для массива символов, а второй раз – для массива целых чисел. При использовании массива строк вызывается перегруженный вариант функции.

Выбор перегруженной функции производится по следующему алгоритму:

1. Строгое соответствие (при допущении некоторых тривиальных преобразований) для нешаблонных функций.
2. Строгое соответствие для шаблонов функций
3. Обычное разрешение аргументов для нешаблонных функций.

Таким образом, компилятор сначала пытается найти подходящий вариант функции, не использующей шаблона, а уже потом – используя его.

Заметим, что функция может иметь несколько шаблонов. Например, можно написать следующую параметризованную функцию для разобранной выше программы (предполагающую, что начальный индекс равен *0*):

```
template<class T>
void shiftleft(T a[], int end) {
  cout<<"in template shiftleft without start"<<endl;
  T temp;
  temp=a[0];
  for (int i=0; i<end; i++)
        a[i]=a[i+1];
  a[end]=temp;
}
```

Механизм шаблонов может быть применен и при определении классов. Например, при определении таких классов как стек, список, дерево и т.п. нам безразлично, какую природу имеют элементы. Было бы нелогично описывать стек отдельно для целых чисел, отдельно – для действительных, отдельно – для символов. Проще использовать, как и в случае функций, шаблоны. Как и для функций, объявление класса предваряет объявление шаблона.

Рассмотрим пример простого параметризованного класса *pair*. Заметим, что в классе используются сразу два параметра типа.

```
#include <iostream.h>
template<class T1, class T2>
class pair {
protected:
  T1 first;
  T2 second;
public:
  pair(T1 f,T2 s):first(f),second(s) { }
  void setfirst(T1 f) { first=f;}
  T1 getfirst() {return first;}
  void setsecond(T2 s) {second=s;}
  T2 getsecond();
};
template<class T1, class T2>
T2 pair<T1,T2>::getsecond() {return second;}
class charpair:public pair<char,char> {
public:
  charpair(char c1, char c2):pair<char,char>(c1,c2) { }
};
void main() {
  pair<char,int> a('a',1);
  cout<<"first="<<a.getfirst()<<" second="<<a.getsecond()<<endl;
  charpair b('b','c');
  cout<<"first="<<b.getfirst()<<" second="<<b.getsecond()<<endl;
}
```

Из примера видно, что параметризованный класс может быть базовым классом при создании производного класса (*charpair* в примере). Производный класс также может быть параметризованным. Например, мы можем изменить производный класс *charpair*, обобщив его так:

```
template<class T>
class eqpair:public pair<T,T> {
public:
  eqpair(T c1, T c2):pair<T,T>(c1,c2) { }
};
```

При этом конкретное объявление объекта класса будет производиться следующим образом:

```
  eqpair<char> b('b','c');
```

Шаблонные класс могут иметь дружественные функции. При этом дружественная функция, не использующая спецификацию шаблона, - универсальна. Это означает, что имеется единственный ее экземпляр для всех инстанцирований шаблонного класса. Если функция задействует аргументы шаблона – она специфична для каждого инстанцирования класса.

Статические переменные не универсальны, они специфичны для каждого инстанцирования. Например, если бы класс *eqpair* содержал бы объявление

```
 private:
  static int count;
```

то переменные

eqpair<char>::count

eqpair<int>::count

различались бы между собой.

Лекция 11.

Простейшая программа для Windows с использованием MFC

Операционная система Windows обладает следующими особенностями, представляющими интерес для разработчиков приложений:

- графический интерфейс, обеспечивающий пользователю удобство в работе и привлекательное графическое изображение;
- приложение выполняется в собственных окнах. Каждое приложение располагает, по крайней мере, одним собственным окном. Через окна приложения выполняется ввод-вывод информации пользователя. С точки зрения пользователя приложение отождествляется с главным окном. Для программиста окно – визуальный интерфейс;
- Работа в Windows ориентирована на события. Приложение в большинстве случаев выполняется отдельными фрагментами. После того как будет решена отдельная подзадача, управление передается Windows, которая может вызывать по мере надобности другие приложения. Какой-либо ввод со стороны пользователя (активация окна, вызов команды меню и т.д.) инициирует событие Windows. Событие обрабатывается и программное управление передается в соответствующее приложение. Происходит это при помощи сообщений, описывающих изменения в окружающей приложение среде. Каждое сообщение связано с конкретным окном, с каждым из которых связана конкретная оконная процедура – специальная функция, отвечающая за обработку поступающих сообщений. Для приложения это означает, что оно вызывается для обработки события, соответственно на него реагирует и затем возвращается в неактивное состояние. Для сравнения: DOS-программа, будучи однажды вызвана, выполняется до тех пор, пока не закончится или не будет прервана.

Для программирования в среде Windows разработан **API** (Application Programming Interface – интерфейс прикладного программирования). Он представляет набор функций, при помощи которых любое приложение может взаимодействовать с Windows. API содержит примерно 2000 функций, несколько сот сообщений, предопределенных констант, макросов. Для облегчения программирования разработана библиотека классов **MFC** (Microsoft Foundation Classes). Она представляет иерархически упорядоченное множество классов, охватывающее большую часть функциональных возможностей Windows. MFC предоставляет разработчику механизмы, которые, не нарушая идеологию операционной системы, существенно ее расширяют и упрощают. Конструкторы классов, их методы берут на себя большую часть рутинной работы.

Любая программа, использующая MFC для работы под Windows, должна определить два класса. Первый – производный класс от класса **CWinApp**, позволяющий создать приложение. Второй – производный класс от класса, определяющего окно. В большинстве случаев это класс **CFrameWnd.**

Класс **CWinApp** является базовым классом, из которого формируется обязательный объект - приложение. Основными задачами этого объекта являются инициализация и создание главного окна, а затем опрос системных сообщений. Инициализация окна производится путем переопределения виртуальной функции **InitInstance**. В этой функции создается объект класса, производного от одного из оконных классов Windows. В ней же вызываются функции, показывающие окно (**ShowWindow**) и прорисовывающую окно (**UpdateWindow**).

Само окно создается в конструкторе класса, производного от оконного класса. Это делается, например, с помощью функции **Create**. В этой функции определяется заголовок окна, его стиль, координаты, и некоторые другие параметры.

Этого достаточно, чтобы на экран выводилось окно определенного нами стиля. Однако это окно не будет обрабатывать никаких сообщений и не будет ничего выводить на экран. Для этого необходимо сделать следующее:

- определить карту сообщений для нашего окна (MESSAGE MAP)
- написать обработчики сообщений.

В нашей программе мы обрабатываем два сообщения – сообщение о щелчке левой кнопкой мыши (ON_WM_LBUTTONDOWN) и сообщение о необходимости перерисовки

окна(ON_WM_PAINT). Для них предусмотрены обработчики с соответствующими именами – **OnLButtonDown** и **OnPaint**.

Поясним, для чего необходим обработчик **OnPaint**. Забота о сохранении содержимого окна ложится на программиста. Если окно оказалось частично или полностью перекрыто другим окном, содержимое окна теряется. Когда закрытая часть окна снова становится видимой, Windows посылает сообщение WM_PAINT о необходимости прорисовки окна заново, что и должен сделать соответствующий обработчик.

Коротко рассмотрим, как устроен графический вывод в системе Windows. Весь вывод реализован на принципе унификации работы с физически различными устройствами (экран, принтер и т.п.). Для всех устройств приложение использует одни и те же функции. Система сама распознает устройство вывода и активизирует соответствующий драйвер. Для реализации такого подхода в Windows предусмотрен специальный объект, называемый контекстом устройства. Именно он хранит информацию как об устройстве вывода, так и о параметрах рисования. В представленной ниже программе используются два контекста:

- **CPaintDC** используется для вывода в окно в обработчике сообщения о необходимости перерисовки окна
- **CClientDC** используется для вывода в клиентскую (рабочую) область окна. Может использоваться для графического вывода в любой функции.

В программе также используется собственный класс, производный от класса **CPoint** (хранит информацию о точке). Мы дополнили его строкой, хранящей некоторое сообщение и методами, позволяющими менять параметры. Заметим, что объекты нашего класса мы описали как глобальные, видимые всем функциям нашей программы. Альтернативный вариант – описать их как часть класса, производного от класса окна.

Файл simple.h

```
#include <afxwin.h>
class simpleApp: public CWinApp {
public:
  virtual BOOL InitInstance();
};
class simpleWin: public CFrameWnd {
public:
  simpleWin();
  DECLARE_MESSAGE_MAP();
protected:
  void OnLButtonDown(UINT,CPoint);
  void OnPaint();
};
```

Файл simple.cpp

```
#include "simple.h"
#include "myclass.h"
myPoint *p,pm;
bool click=false;
simpleApp App;
simpleWin::simpleWin() {
  LPCTSTR classname=NULL;
  Create(classname,"Простая программа",WS_OVERLAPPEDWINDOW | WS_VSCROLL,
CRect(200,200,600,400),0,0,0,0);
  p=new myPoint;
  p->setXY(100,100);
  p->setMessage(".Эта точка(100,100) определена нами");
  pm.setMessage("Здесь щелкнули мышкой");
}
BOOL simpleApp::InitInstance() {
  simpleWin *pMainWnd = new simpleWin;
```

```
  m_pMainWnd=pMainWnd;
  m_pMainWnd->ShowWindow(m_nCmdShow);
  m_pMainWnd->UpdateWindow();
  return TRUE;
}
BEGIN_MESSAGE_MAP(simpleWin,CFrameWnd)
  ON_WM_LBUTTONDOWN()
  ON_WM_PAINT()
END_MESSAGE_MAP()
void simpleWin::OnLButtonDown(UINT nFlags,CPoint point) {
  click=true;
  CClientDC dc(this);
  CRect rect;
  GetClientRect(&rect);
  dc.FillRect(rect,WHITE_BRUSH);
  dc.TextOut(p->getX(),p->getY(),p->getMessage());
  pm.setXY(point.x,point.y);
  dc.TextOut(pm.getX(),pm.getY(),pm.getMessage());
}
void simpleWin::OnPaint() {
  CPaintDC dc(this);
  dc.TextOut(p->getX(),p->getY(),p->getMessage());
  if (click) {
          dc.TextOut(pm.getX(),pm.getY(),pm.getMessage(););
  }
}
```

Файл myclass.h

```
#include <afxwin.h>
class myPoint: public CPoint{
private:
  char message[100];
public:
  myPoint(int x=0, int y=0,char* mes="") :CPoint(x,y) {
          strcpy(message,mes);
  }
  void setXY(int x, int y) {
          CPoint::x=x; CPoint::y=y;
  }
  void setMessage(char *mes) {
          strcpy(message,mes);
  }
  char* getMessage() {
          char *mes=new char[100];
          strcpy(mes,message);
          return mes;
  }
  int getX() {return CPoint::x;}
  int getY() {return CPoint::y;}
};
```

Лекция 12.

Краткий обзор языка Java. Классы и объекты в языке Java.

Язык Java был создан в 1995 г. сотрудниками компании Sun Microsystems. Первоначальным стимулом для появления была потребность в независимом от платформы языке, который мог использоваться для создания программного обеспечения с целью внедрения в электронные устройства различных потребителей. В процессе разработки всплыл второй, более важный фактор, который позднее стал играть ключевую роль в языке Java. Это был Web. Для программ, разрабатываемых для WWW, должна решаться проблема мобильности – переносимости на другие платформы и другие операционные системы. Свойства языка Java, с одной стороны, помогли развитию Интернет, а с другой стороны, Интернет помог развитию языка.

Java получил многие свойства из C и C++. Синтаксис языка Java близок к этим языкам. Он так же является объектно-ориентированным. Но это не дальнейшее развитие C++, приспособленное для Интернет, а самостоятельный язык программирования, не совместимый ни с C, ни с C++.

Язык Java мобилен. Дело в том, что выход компилятора Java является не выполняемым кодом, а байт-кодом – оптимизированным набором команд, предназначенных для выполнения специальной системой, которая называется виртуальной машиной Java (JVM). Поэтому единственным условием переноса программ в другую среду является наличие там JVM.

Из вышесказанного следует, что программа Java интерпретируется (следует иметь ввиду, что интерпретируется не исходный текст, а байт-код). Это помогает сделать ее более безопасной. JVM может как содержать программу, так и оберегать ее от побочных эффектов вне системы.

Исходный текст программы на Java – это один или несколько файлов, содержащие определения классов. Обычно один файл содержит одно определение класса. В Java ничто не может существовать вне класса. Рассмотрим простейшую программу. Программа представляет собой класс Example, который имеет один метод - *main*. Этот метод имеет спецификатор доступа *public*, и использует метод стандартного класса *System*. Метод позволяет напечатать на стандартном устройстве вывода переменную *outstring*, объявленную ранее. Заметим, что переменная outstring является объектом стандартного класса *String*. Имя файла, в котором записан данный класс, должно совпадать с именем класса и обычно имеет расширение *java*: **Example.java**.

```
public class Example {
public static void main(String args[]) {
 String outstring="Мы в мире Java-программ";
 System.out.println(outstring);
}
}
```

Для того, чтобы использовать программу, ее необходимо предварительно скомпилировать. Для компиляции можно использовать стандартный компилятор фирмы Sun **javac**. Например, чтобы компилировать написанный выше класс, необходимо осуществить из командной строки запуск следующим образом:

```
javac Example.java
```

Следует иметь ввиду, что может потребоваться указание полного пути к файлу.

На основе этого текста компилятор Java создает файл Example.class. Для Java весьма существенно, что название файла совпадает с именем класса. Добавим, что каждый класс компилируется отдельно и для каждого класса создается отдельный файл, содержащий байт-код. Обычно этот код имеет очень небольшие размеры, так как все стандартные функции, вызываемые в программе, подключаются виртуальной машиной только на этапе выполнения, а не включаются в байт-коды. Таким образом, программы на языке Java подразумевают **динамическую компоновку**.

Для выполнения программы необходимо указать виртуальной машине точку входа – место, с которого программа выполняться. Такой точкой входа служит метод main. Этот метод может содержаться в любом классе программы. Более того, каждый класс может содержать метод main. Следует иметь ввиду, что в классе может быть только один метод main, и он обязательно должен быть объявлен как открытый (public), статический (static), не должен иметь возвращаемого значения (void). Его аргументами обязательно должен быть массив строк. Через этот массив в программу можно передать параметры. В отличие от C++, так как имя класса и имя файла всегда совпадают, в программу не передается имя файла. Поэтому массив аргументов содержит только те параметры, которые были переданы в программу.

Для выполнения программы с использованием средств разработки фирмы Sun требуется использовать интерпретатор java:

```
java Example
```

При этом не указывается расширение class, так это подразумевается. Если интерпретатор и программа не располагаются в одной директории, может потребоваться указание пути к классу. Обычно, если интерпретатор не находит класс, выдается ошибка «Класс не найден». Для указания пути к классу можно воспользоваться опцией *classpath*:

```
java –classpath путь_к_классу Exaple
```

Разберем, как же принципы ООП реализованы в Java. Как уже было сказано, в основе языка Java лежит класс. Общая форма объявления класса показана ниже.

```
class classname {
 type instance-variable1;
 type instance-variable2;
 …
 type methodname1(parameter-list) {
      //тело метода
 }
 type methodname2(parameter-list) {
      //тело метода
 }
 …
}
```

Данные или переменные, определенные в классе, называются экземплярными переменными. Код содержится внутри методов, которые определяют, как могут использоваться данные. Все вместе, переменные и методы, называются членами класса. Заметим, что методы класса описываются **только** внутри описания самого класса. А в связи с тем, что Java – язык со строгой типизацией, **все** переменные должны быть описаны, и каждая переменная должна быть определенного типа.

Приведем пример определения класса. Здесь описан класс, имеющий одну переменную *length* стандартного типа *double* и два метода – *volume* и *show*. Как мы видим, язык Java позволяет указать метод доступа к переменным и к методам. В нашем случае мы указываем, что переменная *length* является закрытой переменной, т.е. вне класса к этой переменной обратиться невозможно. Методы, напротив, описаны как отрытые. К ним мы можем обратиться из любой части программы (разумеется, соблюдая правила видимости). Данный класс имеет два конструктора – один с параметром, а другой – по умолчанию.

```
class Cube {
private double length;
Cube(double length) {
 this.length=length;
}
Cube() {
```

```
 length=1;
}
public double volume() {
 return length*length*length;
}
public void show() {
 System.out.println("length="+length+" volume="+this.volume());
}
public static void main(String args[]) {
 Cube c1;
 c1=new Cube();
 c1.show();
 Cube c2=new Cube(2.5);
 double v=c2.volume();
 System.out.println(" volume="+v);
}
}
```

В Java класс – это только объявление некоторого типа данных. Получение объектов класса в программе – двухуровневый процесс. Сначала должна быть объявлена переменная типа «класс». Она не определяет объект, а только ссылается на него. Для получения самого объекта необходимо использовать операцию *new*. Например, получение экземпляров приведенного выше класса (и работа с экземплярами) может выглядеть, как показано в методе *main*. Сначала мы определяем переменную *c1*, которая будет указывать на экземпляр данного класса. Сам экземпляр создается следующим оператором. Этот процесс можно совместить: создать объект одновременно с его объявлением.

Лекция 13.

Наследование в языке Java. Класс Object. Абстрактные методы и классы. Финальные методы и классы.

Рассмотрим, как реализованы в языке принципы наследования. Опишем, например, класс WeightCube, являющийся подклассом класса Cube из предыдущей лекции.

```
class WeightCube extends Cube {
 double weight;
 WeightCube(double length, double weight) {
      super(length);
      this.weight=weight;
 }
WeightCube() {
      super();
      weight=1;
 }
 public void show() {
      System.out.println("length="+length+"
      volume="+super.volume()+" weight="+weight);
 }
 public static void main(String args[]) {
 Cube c1;
 c1=new Cube();
 Cube c;
 c=c1;
 c.show();
 WeightCube w1=new WeightCube(1.25,2);
 c=w1;
 c.show();

 }
}
```

Данный класс является подклассом класса *Cube*. Об этом свидетельствует ключевое слово **extends** после имени определяемого класса WeightCube. В конструкторах подкласса мы используем конструктор суперкласса. Нет необходимости вызывать конструктор суперкласса по имени – это делается с помощью ключевого слова **super**. Это возможно, потому то язык Java не допускает множественного наследования – подкласс имеет только одного родителя (разумеется, родитель может в свою очередь иметь своего родителя и т.д.).

Класс *WeightCube* мы дополнили одной переменной *weight* стандартного типа *double*. Мы не определили права доступа к этой переменной. Язык Java может это сделать за нас, определив права доступа по умолчанию. В классе переопределен метод *show* суперкласса. Этот метод будет работать правильно только в том случае, если переменная *length* в суперклассе определена с методом доступа *protected* (или *public*), т.е. закрытым для всех, кроме методов подкласса. Если переменная *length* останется с методом доступа *private*, как это было определено ранее, программа окажется неработоспособной.

Методы подкласса могут использовать не только открытые и защищенные переменные суперкласса, но и открытые и защищенные методы. Например, для переопределенного в подклассе метода *show* показан вызов метода суперкласса *volume*. Для вызова метода суперкласса здесь использовалось уточнение с помощью ключевого слова **super**. В данном случае это не имеет никакого значения. так как мы не переопределяли этот метод в подклассе. Но в общем случае с помощью такой конструкции в подклассе можно выбрать - какой метод должен применяться.

Заметим, что в языке Java решение о применимости переопределенного метода принимается не на этапе компиляции, а на этапе выполнения (т.е. фактически все методы в языке Java являются виртуальными). Это называется динамической диспетчеризацией методов. Ее демонстрирует метод *main*. В языке Java ссылочная переменная суперкласса может обращаться к объекту подкласса. Поэтому в данном примере ссылочная переменная *c* сначала обращается к методу *show* класса *Cube* (так как ссылается на объект этого класса), а потом – к тому же методу класса *WeightCube* (переменная *c* ссылается уже на объект данного класса).

Для класса *Cube* мы не указывали никакого суперкласса. Но это не значит, что он стоит на вершине иерархии. На самом деле, если не пишется имя суперкласс, он считается унаследованным от стандартного класса **Object**, который и стоит в вершине иерархии классов языка Java. Поэтому все классы содержат некоторые стандартные методы:

equals – позволяет сравнивать два объекта;

toString – преобразует содержание класса в строку.

Желательно в своем собственном классе переопределять эти методы, так как стандартная реализация этих методов не всегда приводит к нужным результатам. Например, мы можем для класса *WeightCube* переопределить этот метод следующим образом:

```
    public String toString() {
     return        "length="+length+"        volume="+super.volume()+"
weight="+weight;
    }
```

При этом уже не возникнет необходимости в методе *show*. Вывод в виде:

```
System.out.println(c);
```

даст точно такой же эффект.

Язык Java поддерживает также концепцию абстрактных классов. В ситуации, когда необходимо лишь определить структуру некоторой абстракции без законченной реализации всех методов, мы можем определить класс как абстрактный. При этом нереализованные методы также объявляются как абстрактные. Мы лишь определяем их природу. Невозможно создать объекты абстрактного класса. Мы можем лишь использовать его как суперкласс для некоторых других классов. При этом в подклассах, созданных на основе абстрактного суперкласса, должны быть переопределены все абстрактные методы. Пример определения абстрактного класса и созданного на его основе подкласса показан ниже.

```
abstract class Shape {
 protected double dim1;
 protected double dim2;
 abstract public double area();
 Shape(double d1, double d2) {
 dim1=d1;
 dim2=d2;
 }
}
class Rectangle extends Shape {
 Rectangle(double width, double height) {
 super(width,height);
 }
 public double area() {
 return dim1*dim2;
 }
}
```

Мы определили абстрактный класс *Shape*. Он содержит две защищенные переменные *dim1* и *dim2* и конструктор класса. Также абстрактный класс содержит абстрактный метод *area*, который должен быть переопределен в подклассе. Подкласс *Rectangle* класса *Shape* имеет конструктор, использующий конструктор суперкласса, а также переопределенный метод *area*,

подсчитывающий площадь прямоугольника. Разумеется, класс *Rectangle* может содержать и другие методы, не определенные в суперклассе.

Язык Java позволяет не только создавать иерархию классов, но и запрещать в некоторых случаях создавать иерархию и переопределять методы. Делается это с помощью ключевого слова **final**. Метод, помеченный этим ключевым словом, не может быть переопределен в подклассе. Например, в классе *Rectangle* таким образом мы могли бы пометить метод *area*:

```
public final double area() …
```

Теперь мы можем быть уверены, что ни в каком подклассе этот метод не может быть изменен – ведь метод вычисления площади прямоугольника остается тем же. Это может быть использовано для большей безопасности программ.

Таким же образом мы можем запретить расширение всего класса. Для этого достаточно также описать его как **final**. Подкласс на основе такого класса создать уже невозможно. Это полезно, если, например, класс инкапсулирует какие-либо расчетные методы, которые не должны быть изменены. Например, стандартный класс *Math* в языке Java объявлен как **final** – было бы странно иметь возможность самому переопределить математические функции.

Этим же модификатором могут быть помечены переменные. В этом случае изменить эти переменные невозможно – они превращаются по существу в константы, которые обычно записываются прописными буквами:

```
public final int MIN_VALUE=0;
```

Лекция 14.

Пакеты. Управление доступом в пакетах. Интерфейсы.

При разработке собственного класса и использовании классов других разработчиков надо быть уверенным, что выбранное для класса имя будет уникальным. В противном случае это имя может вступить в противоречие с уже использующимся именем. Соответственно, компилятор языка не поймет, какой именно класс должен быть использован. Таким образом, возникает необходимость организовать свое собственное уникальное пространство имен. Кроме того, необходим способ каким-то образом структурно упорядочить созданные классы. Если создаваемый набор состоит из десятков или сотен классов, им невозможно управлять без создания какой-то внутренней структуры.

Язык Java обеспечивает специальный механизм для разделения пространства имен классов на управляемые части. Этот механизм называется «пакеты» (**packages**). Пакет является механизмом как именования, так и управления видимостью. Все классы Java распределяются по пакетам. Кроме классов, пакеты могут включать в себя интерфейсы и вложенные подпакеты, образуя древовидную структуру.

Эта структура в точности отображается на структуру файловой системы. Все классы, принадлежащие одному пакету, хранятся в одном каталоге файловой системы. Подпакеты собираются в подкаталоги этого каталога.

Каждый пакет образует одно пространство имен. Это означает, что все имена классов, подпакетов и интерфейсов в пакете должны быть уникальны. Имена в разных пакетах могут совпадать. Таким образом, ни один класс, интерфейс или подпакет не могут оказаться одновременно в двух пакетах.

Для того, чтобы создать пакет, необходимо в начале файла с исходным кодом включить оператор:

```
package имя_пакета;
```

Тем самым создается пакет с указанным именем. Все классы, объявленные в пределах этого файла, попадут в соответствующий пакет. Повторяя этот же оператор в начале каждого файла, можно включить в пакет новые классы.

Точно таким же образом создается и иерархия пакетов:

```
package имя_пакета.имя_подпакета;
```

Для создания подпакета необходимо отделить имя подпакета от стоящего выше по иерархии с помощью операции «точка».

Полное имя класса, попадающего в пакет, состоит из имени пакета и имени класса. Например, если в пакете `MyPack.MySubpack` объявлен класс `MyClass`, то его полное имя будет `MyPack.MySubpack.MyClass`.

Компилятор ищет классы только в пределах пакета, который указан в начале файла. Заставить компилятор искать класс в другом пакете можно, указав его полное имя. Если полное имя длинное, а класс используется часто, становится утомительным каждый раз писать его. Для сокращения имени используется оператор **import**:

```
import MyPack.MySubpack.MyClass;
```

Этот оператор (или операторы, если требуется включить несколько классов) в исходном файле всегда следует после оператора package и до определения классов пакета.

Включив оператор **import**, мы в дальнейшем можем использовать имя класса без имени пакета. Если требуется включить несколько классов из одного пакета, можно использовать другую форму оператора:

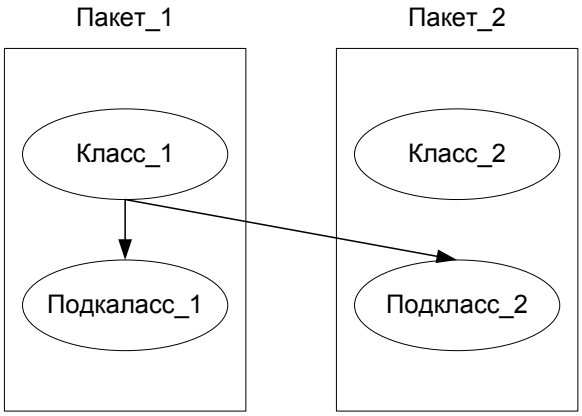
```
import MyPack.MySubpack.*;
```

Тем самым мы указываем компилятору, в каком пакете (а в данном случае – в каком подпакете) следует искать необходимые классы. Импорту подлежат только те классы пакета, котрые объявлены как **public**.

Пакет создается даже в том случае, когда оператор **package** не используется. В таком случае компилятор создает неименованный пакет (**unnamed package**), которому соответствует текущий каталог файловой системы. Поэтому при компиляции файла без определения пакета

class-файл попадает в тот же каталог, что и исходный файл. Если же пакет определен, компилятор сам создает для него каталог и помещает туда скомпилированный файл.

Пакеты добавляют еще одно измерение к управлению доступом. Пакеты действуют как контейнеры для классов и других зависимых пакетов. Классы действуют как контейнеры для данных и кода. Класс – самый мелкий модуль абстракции Java. Рассмотрим права доступа к членам класса.



Из-за взаимодействия между классами и пакетами Java адресует четыре категории видимости для членов класса:

- подкласс в том же пакете (для нашего рисунка – доступ членов класса Подкласс_1 из Класс_1);
- неподклассы в том же пакете (доступ членов Подкласс_2 из Класс_2);
- подклассы в различных пакетах (доступ членов Подкласс_2 из Класс_1);
- классы, которые не находятся в том же пакете и не являются подклассами (доступ членов Класс_2 из Класс_1).

Рассмотрим таблицу, описывающую все многообразие вариантов доступа:

	Private	**Без модификатора**	**Proteced**	**Public**
Тот же класс	True	True	True	True
Подкласс того же пакета	False	True	True	True
Неподкласс того же пакета	False	True	True	True
Подкласс другого пакета	False	False	True	True
Неподкласс другого пакета	False	False	False	True

Таким образом, все, объявленное как **public**, может быть доступно отовсюду. Все, объявленное как **private**, не может быть доступно кроме как собственного класса. Если спецификатор доступа не объявлен, он видим в своем пакете. Это – доступ по умолчанию. Члены класса, объявленные как **protected**. видимы в своем пакете и во всех подклассах, даже если они находятся в других пакетах.

Сам класс может иметь только два возможных уровня доступа: **public** и по умолчанию. Класс, объявленный как **public**, доступен всюду. Класс, не имеющий спецификатора доступа, доступен только в пределах собственного пакета. Заметим, что определение уровня доступа к классу накладывает определенные ограничения на распределение классов по файлам. В одном файле может быть определено сразу несколько классов, но только один из них может иметь доступ public. Все остальные классы файла не должны иметь спецификатора доступа.

Язык Java не допускает множественное наследование. Это было сделано для того, чтобы не возникало проблем, связанных со случаями, когда класс порождается от нескольких классов, которые в свою очередь порождаются от единого предка. Что же делать, когда класс все-таки должен сочетать в себе свойства нескольких классов. В этом случае применяется еще одна конструкция языка – интерфейс. **Интерфейс** (interface) содержит только константы и сигнатуры

методов, без их реализации. Интерфейсы размещаются в тех же пакетах и подпакетах, что и классы, и компилируются также в class-файлы. Определение интерфейса во многом подобно определению класса:

```
access interface interface_name {
 return_type method1(parameter_list1);
 …
 return_type methodN(parameter_listN);
 type final-variable1 = value1;
 …
 type final-variableM = valueM;
}
```

Здесь **access** – спецификатор доступа (или **public**, или без спецификатора). Если спецификатор доступа не используется, интерфейс виден только членам пакета. Если спецификатор доступа включен, интерфейс виден из любого класса.

Объявленные в интерфейсе методы не имеют тел. Это, по существу, абстрактные методы, хотя указывать это специально не нужно. Константы, определенные в интерфейсе – всегда статические и финальные (и это тоже не требуется указывать). Все методы и переменные интерфейсов неявно общие (**public**).

Интерфейсы допускают наследование: после имени интерфейса может стоять слово **extends** и через запятую список интерфейсов-предков. Таким образом, интерфейсы могут порождать свою, независимую от классов, иерархию, в которой допускается множественное наследование.

Приведем пример определения простого интерфейса:

```
interface MyInterface {
 void myMethod(int param);
}
```

Когда интерфейс определен, он может реализовываться одним или несколькими классами. Для реализации интерфейса в определение класса включают предложение implements, за которым через запятую следуют имена интерфейсов:

```
access class class_name [extends superclass] [ implements interface_name1 [, interface_name2 …]] {
// тело класса
}
```

Методы, которые реализуют интерфейс, должны быть объявлены как public. приведем пример:

```
class MyClass implements MyInterface {
 public void myMethod(int param) {
      System.out.println("param="+param);
 }
}
```

Если в классе реализуются не все методы интерфейса, он должен быть объявлен как абстрактный.

Интерфейсы позволяют средствами языка реализовывать чистое объектно-ориентированное проектирование, не отвлекаясь на вопросы реализации проекта: мы можем записать проект в виде иерархии интерфейсов, а затем построить иерархию классов, учитывая видимость членов класса и одиночное наследование.

Мы можем создавать ссылки на интерфейсы. Эти ссылки, разумеется, могут указывать только на реализацию интерфейса. Например:

```
MyInterface m=new MyClass();
m.myMethod(1);
```

Понятно, что ссылочная переменная не может обращаться к методам класса, не реализующим интерфейс. Но с ее помощью можно обращаться к одним и тем же методам в разных классах (если эти методы реализуют данный интерфейс). Таким образом, интерфейсы дают еще один способ организации полиморфизма.