# Объектно–ориентированное программное обеспечение

## Критерии качества программного обеспечения

Для ПО различают 2 различных типа качества. Может быть обнаружен пользователями (быстродействие, интерфейс и т.д.) – это внешние факторы качества. Такие характеристики как модульность, читаемость понятны профессионалам разработчикам и являются внутренними факторами качества. В конечном счёте только внешние факторы имеют значение, но ключ к достижению внешних факторов спрятан во внутренних факторах. Рассмотрим основные внешние факторы качества ПО.

**Correctness** (корректность) – способность ПО точно выполнять задачи так, как они определены в спецификации. Это важнейшее качество, если она не соблюдена, то все остальные качества не имеют особого значения.

Первый шаг к соблюдению корректности – это точная спецификация тех. Требований к системе.

Методы обеспечения корректности являются многоуровневыми:

1. Приложение
2. Транслятор
3. Операционная система
4. Железо

Корректность каждого уровня основывается на том, что ниже лежащие уровни корректны.

**Robustness** (устойчивость) **–** способность ПО соответствующим образом реагировать на аварийные ситуации. Устойчивость дополняет корректность.

Корректность – поведение системы в случаях определенной спецификации. Корректность определяет то, что происходит за пределами спецификации.

Роль требований устойчивость – удостовериться, что в аварийных случаях не произойдет непоправимой ситуации.

Возможные реакции

* Сообщение об ошибке
* Корректное завершение работы
* Переход в режим постепенного вывода из работы

**Extendibility** (расширяемость) – лёгкость адаптации ПО к изменениям спецификации.

Расширяемость – это проблема масштаба. Для небольших программ она несложна. Для больших программ возникают сложности. Для улучшения расширяемости важны 2 принципа:

1. Простота архитектуры
2. Децентрализация

Чем более автономны модули – тем больше шансов, что изменения затронут небольшое их количество.

**Reusability** (повторное использование) – способность элементов ПО служить для построения различных приложений.

Повторное использование влияет на остальные аспекты качества и позволяет прилагать больше усилий к улучшению других факторов.

**Compatibility** (совместимость) – лёгкость сочетания одних программных элементов с другими.

Ключ к совместимости заключается в стандартных соглашениях на коммуникацию между элементами ПО. Стандартные форматы файлов, структуры данных, пользовательский интерфейс.

**Efficiency** (эффективность) – способность ПО как моно меньше зависеть от ресурсов оборудования.

Существуют 2 подхода к эффективности:

1. Сразу тратить усилия на оптимизацию
2. “Сделай правильно, прежде чем делать быстро”

**Portability** (переносимость) – легкость переноса ПО в различные программные аппаратные среды.

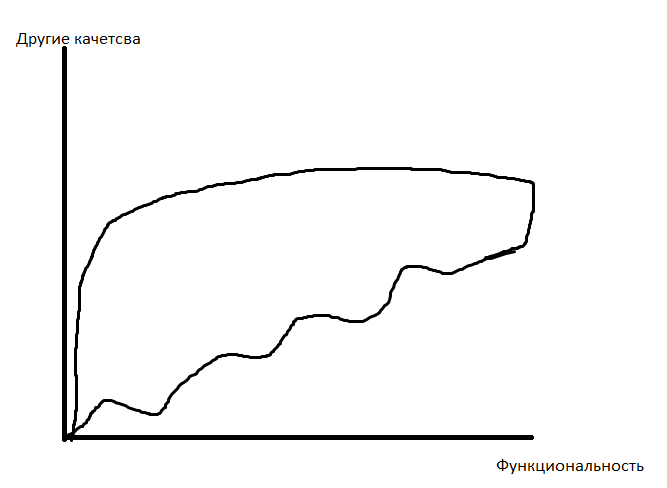
**Easy of use** (быстрота использования) – легкость, с которой с различными знаниями и квалификацией могут научить использовать программное обеспечение и применять его для решения задач. Сюда относится простота установки и текущего контроля.

**Functionality** (функциональность) – степень возможности обеспеченной системой.

Одна из самых сложных проблем – определение достаточной функциональности. Всегда существует желание добавить к проекту все новые и новые свойства.

Расширение системы приводит к 2 проблемам:

1. Потеря непротиворечивости
2. Слишком большое внимание к одним свойствам в ущерб другим качествам программы



*Кривые Осмонта*

**Timeliness** (своевременность) **–** выпуск ПО в нужный момент. Т.е. тогда или незадолго до того, когда у пользователей появилась соответствующая потребность.

**Verifiability** (верифицируемость) – Лёгкость подготовки процедур приёмки, процедур получения информации об отладке и трассировке ошибок.

**Integrity** (целостность) – способность ПО защищать свои компоненты от несанкционированного доступа и модификации.

**Reparability** (восстанавливаемость) – способность устранять наличие дефектов.

**Economy** (экономичность) – способность проекта завершиться, оставаясь в рамках бюджета.

### Компромиссы

Некоторые элементы конфликтуют друг с другом. Например, целостность с простотой использования, экономичность и функциональность, эффективность и переносимость. Повторное использование расширяет границы заданной спецификации и т.д. Поэтому приходится прибегать к различного рода компромиссам. Компромиссы могут затрагивать все факторы качества, кроме одного – корректности.

## Парадигма программирования

Парадигма программирования – это способ построения программ, основанный на определенных принципах программирования.

В структурной или процедурной парадигме основной абстракцией является алгоритм.

В логической парадигме основной абстракцией являются цели, обычно выраженные в терминах исчисления предикатов.

В объектно-ориентированной парадигме основной абстракцией являются классы и объекты.

Для объектной парадигмы концептуальная база - объектная модель.

Из 7 основных принципов, которые 4 являются главными (абстрагирование, инкапсуляция, модульность и иерархия) и 3 дополнительными (типизация, параллелизм и сохраняемость)

### Абстрагирование

Является одним из основных методов используемых в решениях сложных задач

Абстракция – выделяет существенные характеристики некоторого объекта, отличающие его от всех видов других проблем 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 объектов.

Клиентом называется любой объект, использующий ресурсы другого объекта (сервера)

Такой подход концентрирует внимание на внешних проявлениях объекта и приводит к идеи контрактной модели программирования, внешнее проявление объекта рассматривается с точки зрения его контракта с другими объектами в соответствии с этим должно быть выполнено и его внутреннее устройство.

Контракт фиксирует все обязательства, которые объект-сервер имеет перед объектом-клиентом.

Каждая операция, предусмотренная контрактом, однозначно определяется формальными параметрами и типом возвращаемого значения.

Полный набор операций вместе с правильным порядком вызова, называется протоколом.

Протокол отражает все возможные способы, которыми объект может действовать или подвергаться воздействию полностью определяя внешнее поведение абстракции со статической и динамической точек зрения.

Центральная идея абстракции является понятие инварианта.

Инвариант – это некоторое логическое условие, значение которого должно сохранятся

Для каждой операции объекта можно задать предусловие (инварианты предполагаемой итерации) и постусловие (инварианты которым удовлетворяет операция).

Изменение инварианта нарушает контракт связанный с итерацией. Если нарушено условие, то клиент не соблюдает свои обязательства и сервер не может выполнить задачу правильно, если нарушено постусловие, то свои обязательства нарушил сервер.

При нарушении какого-то условия вызывается исключительная ситуация.

Все абстракции обладают как статическими, так и динамическими свойствами

Например, файл имеет имя, размер и определённое содержание, эти атрибуты являются статическими, а конкретные значение этих атрибутов динамичны и меняются в процессе выполнения объекта.

Абстракция и инкапсуляция дополняют друг друга. Абстрагирование направлено на наблюдаемое поведение объекта, а инкапсуляция занимается внутренним устройством. Чаще всего инкапсуляция выполняется по средствам скрытия информации или маскировкой всех внутренних деталей, не влияющих на поведение объекта. Обычно вскрывают и внутреннюю структуру объекта, и реализацию его методов.

Инкапсуляция – это процесс отделение друг от друга элементов объекта, определяющие его устройство и поведение.

Инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

ПРИМЕР

Одной из ключевых абстракций в тепличном хозяйстве является нагреватель, можно ограничиться 3 действиями с этим объектом (включение/выключение/запрос состояния). Нагреватель не должен отвечать за поддержание температуры, эта абстракция более высокого уровня, совместно реализуемая датчиком, нагревателем и регулятором.

Во многих языках попытка обращения к закрытым данным выявляется на этапе компиляции, но путем манипуляции адресами, можно забраться во внутрь закрытой части объекта.

«Инкапсуляция защищает от ошибок, но не от жульничества»

## Инкапсуляция в Java script

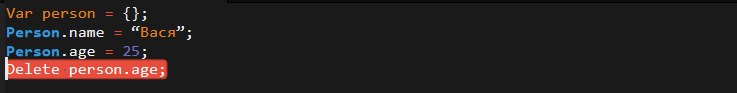
Пустой объект можно создать 2 способами:



В любой момент после создания объекта можно добавить к нему свойства путем простого присваивания.

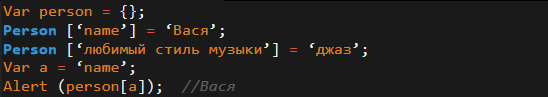


Можно удалить свойство:

Существует возможность обратиться по чтению к отсутствующему свойству, при этом ошибка не возвращается.

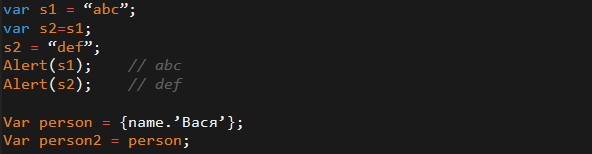


Существует альтернативный способ работы со свойствами использующий квадратные скобки.

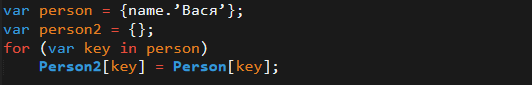


Объект можно заполнить значениями при создании:

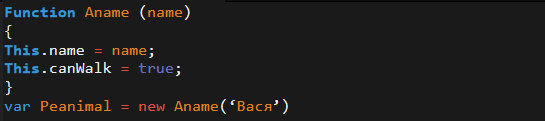
Объекты в отличии от других примитивов (строки, числа) копируются по ссылке:



Следует создать клон объекта



Часто необходимо создать группу одинаковых объектов для этого используют функции конструктора. Конструктором становиться любая функция, запущенная оператором new.



Функция, запущенная через new:

1. Создаётся пустой объект
2. This получает ссылку на этот объект
3. Функция выполняется
4. Возвращается this

В функции конструктора можно объявлять локальные переменные и вложенные функции, которые будут видны только изнутри.

## Иерархия

Значительное упрощение в понимании сложных задач достигается за счет образования из абстракций и иерархической структуры.

Иерархия – это упорядочение абстракций и расположение их по уровням.

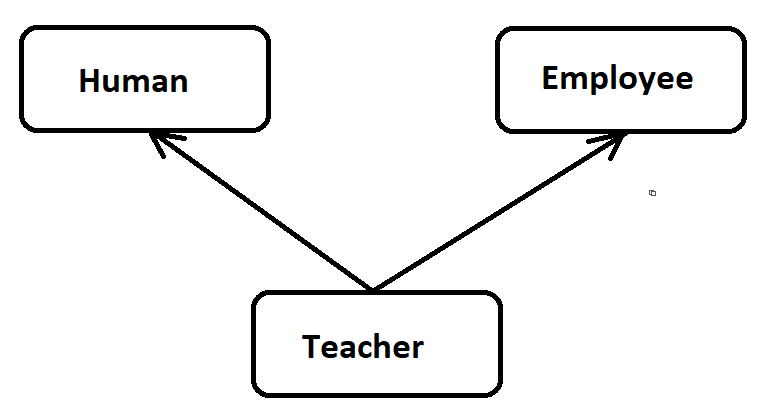
Существует 2 основных вида иерархических структур:

1. Иерархия классов (структура классов)
2. Иерархия объектов (структура объектов)

Важным элементом объектно-ориентированных систем и основным видом иерархии ИСЭ, является концепция наследования.

Наследование означает такое отношение между классами, когда один класс заимствует структурную и функциональные части одного или нескольких других классов. Иными словами, наследование создает такую абстракцию, в которой под классы наследуют строение от одного или нескольких суперклассов, изменяя и(или) добавляя новые элементы.

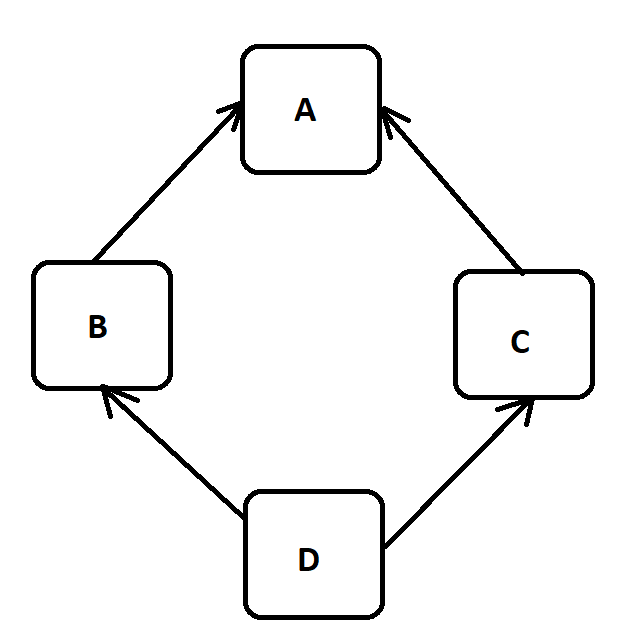
Суперклассы отображают более общие, а подклассы более специализированные абстракции. Принципы инкапсуляции и иерархии находятся в некотором противоречии. Инкапсуляция скрывает внутреннее строение объекта, а принцип наследования требует открыть туда доступ (компромиссом является protected(в плюсах)).



*#include* <string>  
  
*using namespace* std;  
  
*class* Human{  
 string m\_name;  
 *int* m\_age;  
*public*:  
 Human(string name, *int* age){  
 m\_name = name;  
 m\_age = age;  
 }  
 *int* getName(){  
 *return* m\_name;  
 }  
 *int* getAge(){  
 *return* m\_age;  
 }  
};  
  
Class Employee{  
 string m\_employez;  
 *double* m\_wage;  
*public*:  
 Employe(String employes, *double* wage){  
 m\_employez = employez;  
 m\_wage = wage;  
 }  
   
 string getEmployez(){  
 *return* m\_employez;  
 }  
   
 *double* getWage(){  
 *return* m\_wage;  
 }  
};  
  
Class Teacher: *public* Human, *public* Employee{  
 *int* m\_teachersGrade;  
*public*:  
 Teacher(string name, *int* age, string employez, *double* wage, *int* TeachersGrade) :  
 Human(name, age), Employee(employez, wage), m\_teachersGrade(teachersGrade){}  
};

Множественное наследование может привести к ряду проблем, которые могут заметно увеличить сложность программы и сделать кошмаром дальнейшую поддержку кода.

1. Может возникнуть неоднозначность, когда несколько родительских классов имеют одноименные методы.
2. Ромбовидное наследование или “Алмаз смерти”



## Наследование в JavaScript

### Функциональное наследование

В JS наследование можно реализовать 2 способами 1 из которых функциональное наследование или наложение конструкторов. Рассмотрим пример, определением базовый класс машина и определим кофеварку.



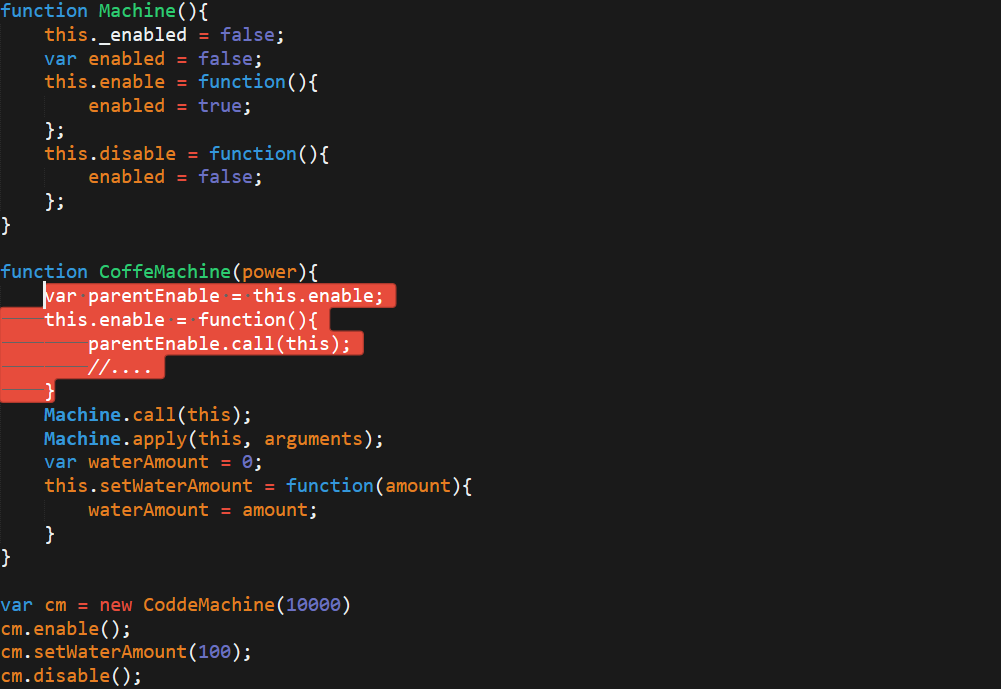
В данном примере есть проблема, наследник не имеет доступа к приватному свойству “enabled” в классе машина. Чтобы получить доступ к свойству – его нужно записать в this, а, чтобы подчеркнуть, что свойство является внутренним его имя, начинают с подчеркивания. Подчеркивание – общепринятый знак что свойство является внутренним предназначенным лишь для доступа из самого объекта и его наследников.



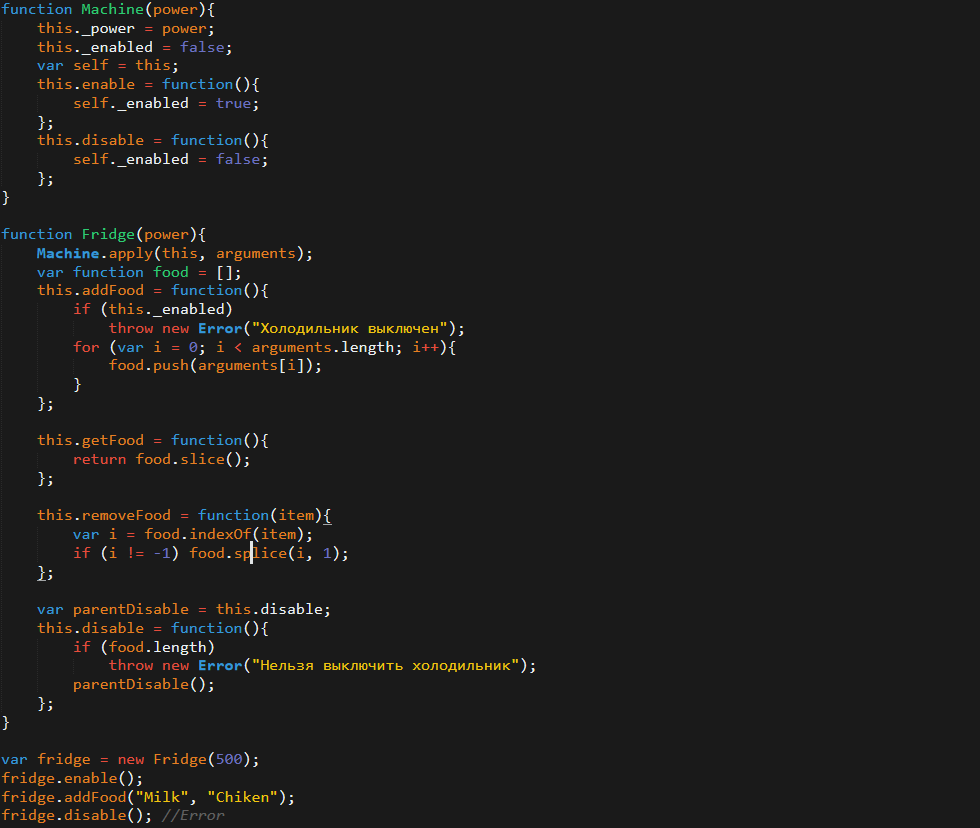
Предположим, что необходимо переопределить метод родителя, например, для кофеварки необходимо переопределить метод enable.



Но чаще необходимо доопределить метод.

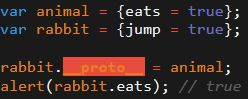


Рассмотрим более крупный пример “Холодильник”



### Наследование в прототипном стиле

Объекты JS можно организовывать в цепочки так, чтобы свойство, ненайденное в 1 объекте автоматически искалось по-другому. Связующим звеном выступает специальное свойство \_\_proto\_\_. Если 1 объект имеет ссылку \_\_proto\_\_ на другой объект, то при чтении свойства из него, если свойство отсутствует в самом объекте, оно ищется в объекте \_\_proto\_\_.

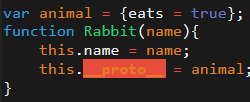


Объект, на который указывает ссылка \_\_proto\_\_ является прототипом.

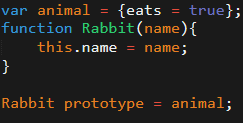
Прототип используется исключительно при чтении. Запись значения работает напрямую с объектом.

Используя \_\_proto\_\_ можно строить достаточно длинные цепочки объектов.

Рассмотрим задание прототипа при создании объекта через функцию конструктора

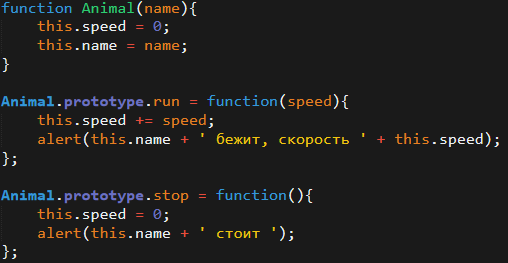


С этим способом были проблемы в Internet explorer. Поэтому есть 2 вариант



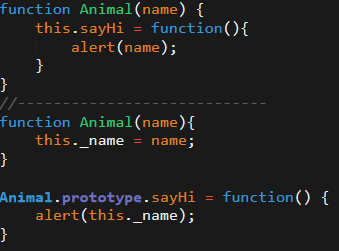


Функциональный стиль



Прототипный стиль

В результате в объекте хранятся только свойства, а общие методы в прототипе (достоинства портативного стиля). Функциональный стиль записывает свойства и методы, а прототипный только свойства. Поэтому прототипный стиль стребует меньше ресурсов. При создании методов через прототип теряется возможность использовать локальные переменные как приватные свойства. У них нет общей видимости с конструктором.



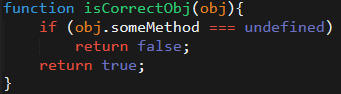
Если иерархия is a определяет отношение обобщения специализация, то отношение part of вводит иерархию агрегация. Агрегация есть во всех языках, использующих структуры и записи, состоящие из разнотипных данных. Агрегация позволяет физически сгруппировать. Логически связанные структуры, а наследование копирует эти общие группы в различные абстракции. В связи с агрегацией возникает проблема владения или принадлежности объектов.

Рассмотрим огород и растущие в нем растения. При уничтожении огорода возможны 2 решения.

1. Уничтожить растения
2. Растения можно пересадить в этом случае огород и растения имеют свои отдельные и независимые сроки жизни.

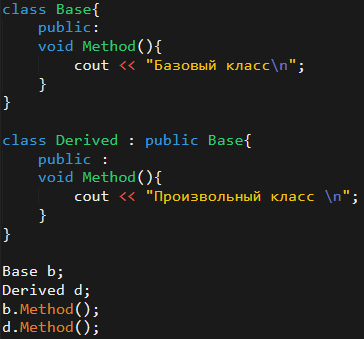
## Типизация

Это способ защититься от использования объектов 1 класса вместо другого или управлять таким использованием. Языком программирования может иметь сильную или слабую типизацию, оставаясь объектно-ориентированным. В сильнотипизированных языках нарушение согласования типов обнаруживается во время трансляции, в слаботипизированных обычно проявляется ошибка исполнения (или не проявляется).



Следует различать сильную и статическую типизации. Сильная типизация следит за соответствием типов. А статическая определяет время, когда имена связываются с типами. Статическая связь означает, что типы всех переменных и выражений известны при компиляции, а динамическое (позднее) связывание означает, что типы не известны до времени выполнения программы.

Пример

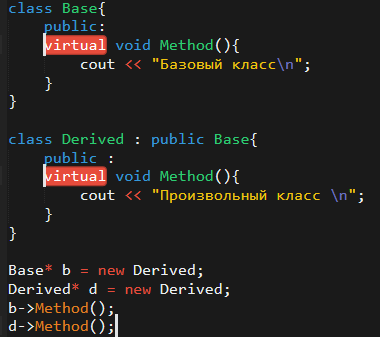


На этапе компиляции память выделяется для 2 версий функции Method и адреса привязываются к именам Base::Method() и Derived::Method().



Вывод будет таким же.

Чтобы сработало позднее связывание нужно функцию член класса метод объявить как виртуальную.



Теперь дважды будет выведен произвольный класс.

Для виртуальных методов память выделяется так же на этапе компиляции, но адреса их не привязываются к именам. Когда в базовом классе объявляется хотя бы одна виртуальная функция, то для всей иерархии создается таблица виртуальных функций. Это одномерный массив указателей на функции, количество элементов массива = количеству виртуальных функций в классе. При создании объекта таблица заполняется адресами виртуальных методов в соответствии с созданным типом.

Способность родительского класса представлять любого из своих наследников называется полиморфизмом. Полиморфизм возникает там, где взаимодействует наследование и динамическое связывание.

## Параллелизм

Есть задачи, в которых много событий должны обрабатываться одновременно. В других случаях при вычислении единственной задачи требуется большая вычислительная мощность. В каждой из этих задач естественно использовать несколько компьютеров, либо многозадачность на одном компьютере. Процесс(потокоуправление) это фундаментальная единица действия в системе. Каждая программа имеет по крайней мере 1 поток управления, параллельная система – имеет множество потоков. Реальная многозадачность достигается только на многопроцессорных системах. А в однопроцессорных только имитируется за счёт алгоритмов деления выполнения. Следует различать “тяжелую” и “легкую” потребность. Тяжелые задачи выполняются независимо друг от друга, под них выделяется защищенное адресное пространство. Легкие сосуществуют в едином адресном пространстве. Параллелизм основное внимание уделяет абстрагированию и синхронизации процесса. Каждый объект, полученный из абстракции реального мира может представлять собой отдельный поток управления(абстракцию процесса). Такой объект называют активным. Для систем, построенных на основе ООД (д-дизайн(проектирование)) – мир может быть представлен как совокупность взаимодействующих объектов, часть з которых может быть активной и выступает в роли независимых вычислительных центров. На этой основе можно дать определение параллелизма. Параллелизм позволяет различным объектам действовать одновременно. Параллелизм – это свойство, отличающее активные объекты от пассивных. ООД есть 3 подхода к параллелизму:

1. Параллелизм – внутреннее свойство языков программирования.
2. Использовать библиотеку классов, реализующих параллелизм.
3. Реализация многозадачности с использованием прерывания.

Как только в систему введен параллелизм возникает проблема синхронизации объектов.

## Сохраняемость

Любой программный объект существует в памяти и живет во времени. Время жизни может быть совершенно различным. Например: Промежуточные результаты вычисления выражения, локальные переменные подпрограмм, глобальные переменные и динамические данные. Данные, сохраняющиеся между сеансами выполнения программы, данные, сохраняемые при переходе на новую версию программы, данные, которые переживают программу.

Сохраняемость – способность объекта существовать во времени, переживать породившие его процессы или перемещаясь из 1 адресного пространства в другое.

## Качество классов и объектов