

# Polymorphism

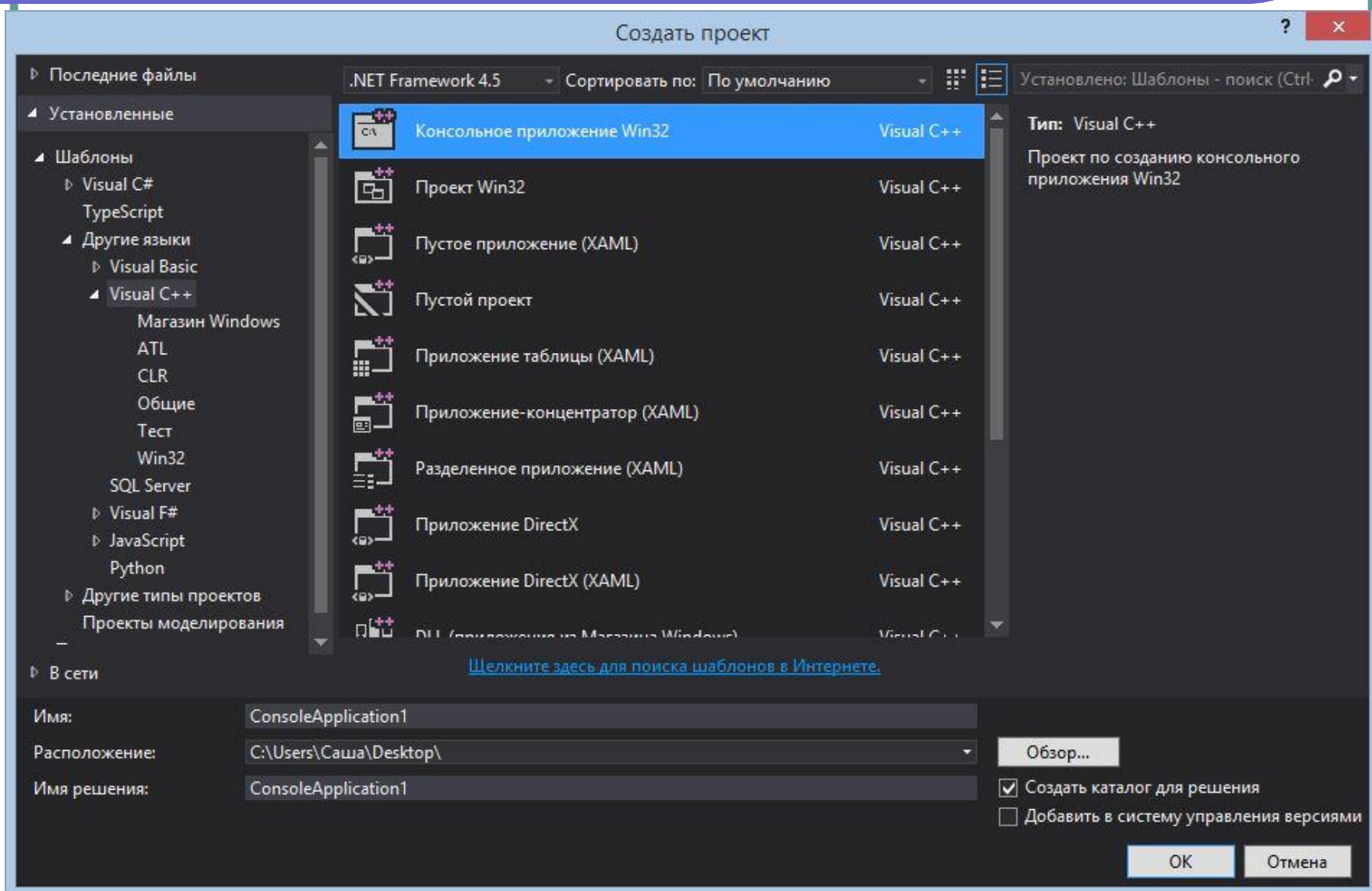


# Предупреждение

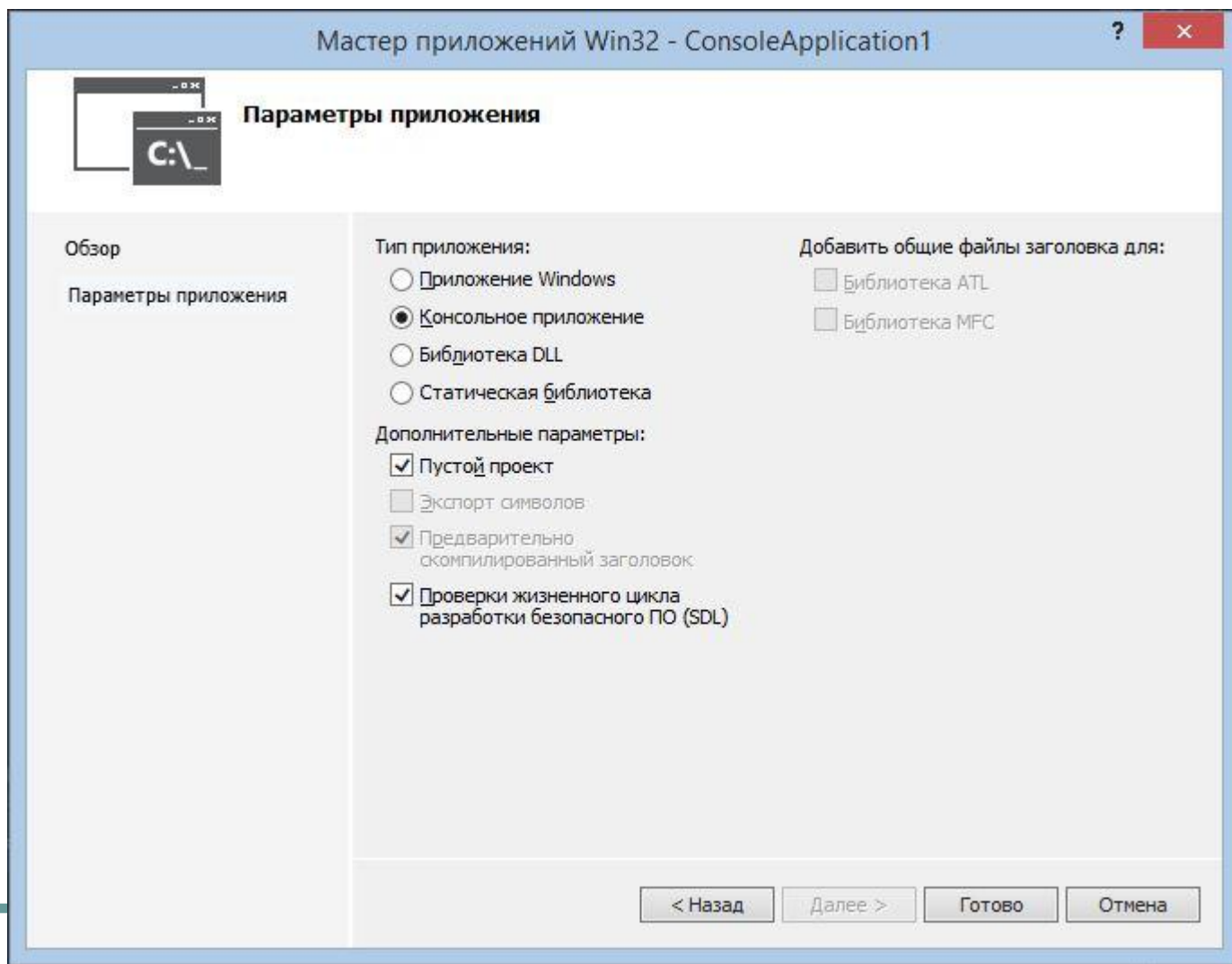
В данной презентации почти все примеры кода написаны на языке C++. Примеры рабочие, и будут запускаться в IDE Microsoft Visual Studio 2013/2015. Язык C++, по сравнению с Java, предоставляет более широкий выбор инструментов для понимания того, что происходит «под капотом» программы. Наличие в нём операторов для получения адресов объектов в памяти и определения точного размера объектов в байтах, простое переключение между ранним и поздним связыванием, работа с таблицей виртуальных методов в отладчике, позволит новую тему во всех деталях.



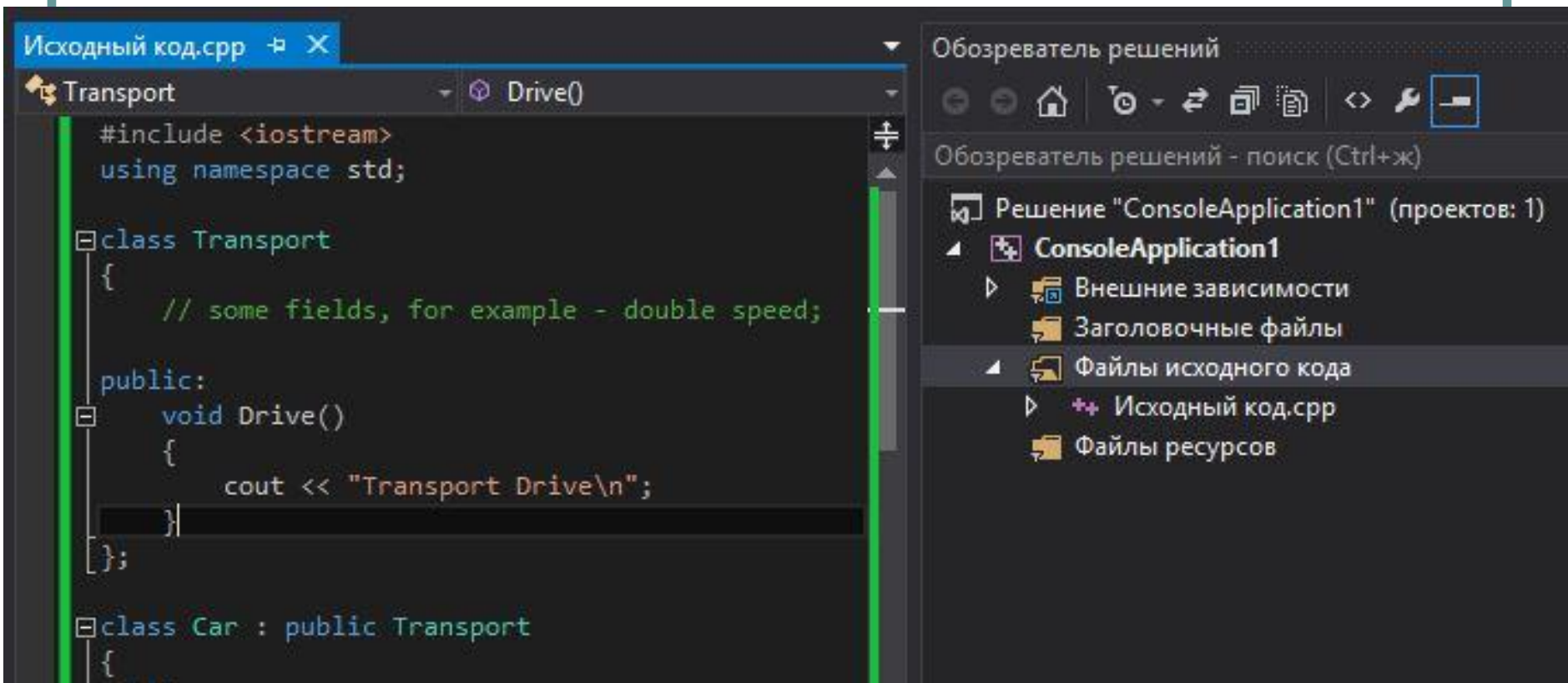
# Создание проекта



# Ставим чекбокс «Пустой проект»



# Добавляем сpp-файл



The image shows a screenshot of the Visual Studio IDE. The main window displays the source code for a C++ program. The code defines a base class `Transport` and a derived class `Car`. The `Transport` class has a `Drive()` method that prints "Transport Drive\n". The `Car` class inherits from `Transport`.

```
Исходный код.cpp [X]
Transport Drive()
#include <iostream>
using namespace std;

class Transport
{
    // some fields, for example - double speed;

public:
    void Drive()
    {
        cout << "Transport Drive\n";
    }
};

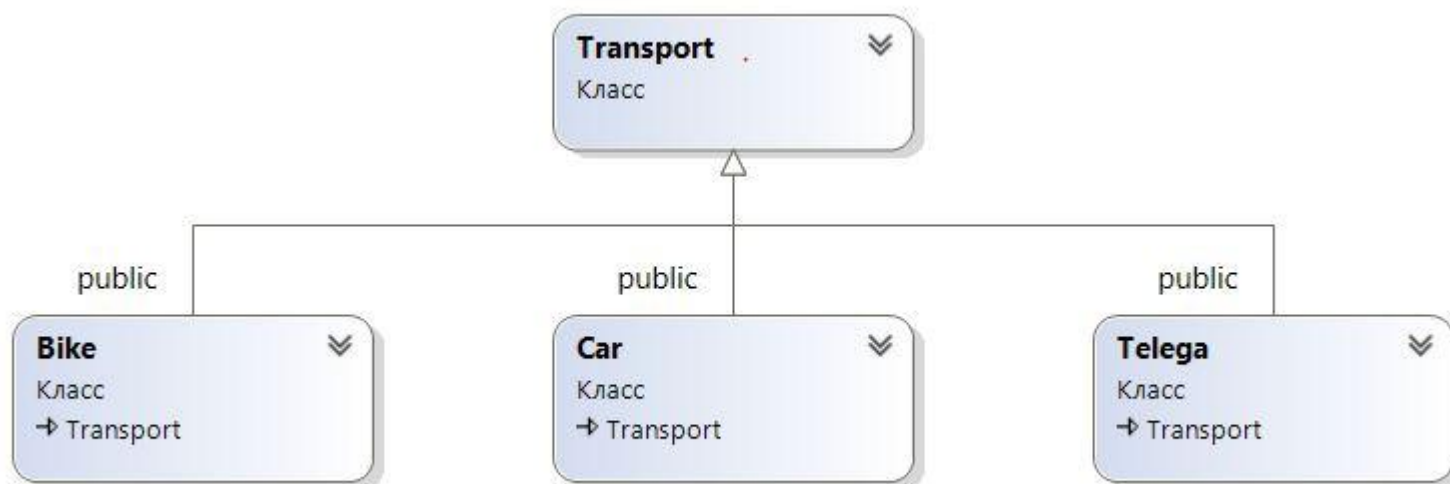
class Car : public Transport
{
```

On the right side, the Solution Explorer shows the project structure for "ConsoleApplication1". The tree view includes:

- Решение "ConsoleApplication1" (проектов: 1)
  - ConsoleApplication1
    - Внешние зависимости
    - Заголовочные файлы
    - Файлы исходного кода
      - Исходный код.cpp
    - Файлы ресурсов

# Транспортные средства

<https://git.io/vrqav>



# Работа примера

Каждый класс, отнаследованный от `Transport`, получает метод `Drive`. (т.е., наследники обладают общим интерфейсом). Однако, каждое конкретное транспортное средство будет ехать по-своему (разные реализации, т.к. методы переопределены).

Если создать несколько объектов разных подклассов, то компилятор поступит вполне предсказуемо, и вызовет метод `Drive` из класса `Car` для объекта типа `Car`, и метод `Drive` из класса `Bike` для объекта типа `Bike`.

# Раннее связывание

На что при этом ориентируется компилятор? В данном случае, на тип указателя (ссылки), который содержит адрес объектной переменной. Причём тип указателя точно известен на этапе компиляции, а это означает, что связывание вызова метода через этот указатель на объект с кодом реализации метода Drive происходит на этапе построения приложения. Такой процесс называется **раннее связывание** (static dispatch).



# Моделирование

Предположим, в программе необходимо смоделировать поведение различных видов транспорта на перекрёстке. Всё просто: как только на светофоре загорится зелёный - все машинки должны поехать.



# Объекты разных типов

Однако, следует учесть, что транспортные средства будут разные, и ехать они должны по-разному... К тому же, заранее неизвестно, сколько всего машин, мотоциклов и телег будет у светофора, т.е. их общее количество определяется динамически, уже на этапе выполнения программы.

# Проблема

Обычно для работы с группой объектов используются массивы либо другие коллекции, вроде списков или деревьев. Но ведь транспортные средства у нас будут с разными типами! А в коллекциях все элементы всегда однотипные...



# Решение

Для решения этой проблемы придумали одну очень хитрую вещь: разрешается делать ссылку на объект с типом базового класса, и в дальнейшем присваивать ей адреса объектов производного типа (но не наоборот!)

```
Transport t = new Car();
```

```
// Transport* t = new Car(); // код C++
```

# Массив ссылок на объекты

Теперь можно будет создать целый массив ссылок типа базового класса, и поочерёдно присвоить им адреса объектов различных производных типов. Таким образом решается проблема хранения разнотипных объектов (однако имеющих общего предка!) в виде массива.

```
Transport** ar = new Transport*[2];  
ar[0] = new Bike();  
ar[1] = new Telega();
```

# Доверяй, но проверяй

Итак, попробуем применить новые знания на практике:

<https://git.io/vrqyZ>



**ПОЕХАЛИ!**

# Что-то пошло не так...

Упс! При попытке моделирования ситуации на светофоре, программа сработала не совсем так, как хотелось бы. Всеми виной – то самое раннее связывание. Ну в самом деле, метод Drive вызывается через указатель `traffic[i]`, а ведь это указатель с типом `Transport`... Соответственно, компилятор берёт и вызывает метод именно из класса `Transport`. В итоге, и мотоциклы, и телеги, и машины поедут каким-то общесхематическим образом (таким, как это определено в классе `Transport`).

# Позднее связывание

Для того, чтобы в C++ сменить механизм с раннего связывания на позднее, достаточно пометить метод `Drive` в базовом классе `Transport` как **virtual**. Метод станет **виртуальным**, и связывание вызова метода через указатель (ссылку) на объект с кодом реализации метода будет происходить уже **на этапе выполнения программы**, а не на этапе компиляции.



# Правило виртуальности

Получается, что наличие в коде ключевого слова `virtual` решило все проблемы по работе с разнотипными объектами!

Существует **правило виртуальности**: метод, объявленный виртуальным в некотором классе, остаётся таким во всех классах-потомках. Но для наглядности в C++ рекомендуется писать ключевое слово `virtual` и в классах-наследниках, чтобы код оставался читабельным и понятным.

# Определение

Виртуальный м. - это метод класса, который может быть переопределён в классах-наследниках так, что конкретная реализация метода для вызова будет подбираться во время исполнения. Таким образом, программисту необязательно знать точный тип объекта для работы с ним через виртуальные методы: достаточно лишь знать, что объект принадлежит наследнику класса, в котором метод объявлен.

# Полиморфизм

Виртуальные методы - это один из важнейших приёмов реализации **полиморфизма**. Они позволяют создавать общий код, который может работать как с объектами базового класса, так и с объектами любого его класса-наследника. При этом базовый класс определяет наличие способа работы с объектами, а любые его наследники могут предоставлять конкретную реализацию этого способа.

# Важнейшая концепция ООП

**Полиморфизм** – важнейшая концепция в ООП. Большинство лучших практик и решений основаны на полиморфизме и глубокое понимание принципов и тонкостей работы данного механизма является обязательным для построения гибкой и надёжной архитектуры ПО!

# Определение

**Полиморфизм** – это принцип, согласно которому есть возможность использовать одну и ту же запись для работы с объектами различных типов данных.  
Кратко: «**один интерфейс, множество реализаций**». Полиморфизм позволяет единообразно работать с объектами различных типов, подменяя только сами объекты, но не код по их обработке.

# Полиморфная строка кода

```
for (int i = 0; i < count; i++)  
    traffic[i]->Drive();
```

# Виды полиморфизма

В узком смысле полиморфизм разделяют на **статический** и **динамический**. Однако, в большинстве ситуаций под полиморфизмом понимают именно динамический полиморфизм.

**Статический полиморфизм** – это механизм, при котором одна и та же инструкция может быть использована для работы с объектами разных типов, но конкретный тип и инструкции по работе с ним уже известны на этапе компиляции!

# Статический полиморфизм

- **Ad-hoc полиморфизм.** Реализуется через механизм перегрузки методов – эта тема вам уже хорошо знакома.
- **Параметрический полиморфизм.** Реализуется через механизм обобщений (generics, дженериков) – с этим будем разбираться после темы «интерфейсы».



# Динамический полиморфизм

**Динамический полиморфизм** – это механизм, при котором одна и та же инструкция может быть использована для работы с объектами разных типов, но **конкретный тип и инструкции по работе с ним НЕ известны на этапе компиляции, а определяются на этапе выполнения** (реализуется т.н. полиморфное поведение).

# Диспетчеризация

Для реализации динамического полиморфизма используется **subtype polymorphism**, то есть ДП реализуется только через механизм наследования. Для понимания реализации полиморфного поведения, необходимо как-то выяснить, каким образом выбирается конкретная реализация, которую нужно вызвать для объекта. Для определения конкретного метода при вызове используется механизм **связывания** (диспетчеризация, dispatch).

# Ещё раз о раннем связывании

Присваивание ссылок разных типов данных возможно только тогда, когда слева от оператора присваивания находится ссылка на базовый класс, а справа - адрес объекта одного из производных классов. Через ссылку на базовый класс можно работать с объектом производного класса, но только с той его частью, которая была унаследована из базового. При раннем связывании при работе с объектом производного класса через ссылку на базовый класс связывание вызова метода с самим кодом метода происходит на этапе компиляции программы. То есть вызывается метод класса, соответствующий типу указателя (ссылки), а не типу объекта, который адресуется через данный указатель.

# Механизм позднего связывания

При работе через ссылку базового типа с объектом производного класса, часто требуется, чтобы связывание вызова метода с самим кодом метода происходило именно на этапе выполнения программы. То есть, чтобы вызывался метод в соответствии с настоящим типом объекта, а не типом ссылки, которая содержит адрес данного объекта. Для решения данной проблемы в базовом классе (в C++) переопределяемый метод помечается как **виртуальный**. А в производных классах, этот виртуальный метод просто переопределяется.

# Как это всё работает?

Для начала, рассмотрим пример:

<https://git.io/vr1BB>

Пример демонстрирует, как можно получить адреса объектов, их полей и методов.

# Добавим наследование

Теперь примерно то же самое, но с наследованием:

<https://git.io/vr1RT>

# Время экспериментов

- Теперь попробуйте сделать метод `Guard` виртуальным (`virtual` можно писать как до типа возвращаемого значения, так и после него). Что изменилось?
- А теперь сделайте виртуальным ещё и метод `Bark`. Что-то изменилось?

# Загадочные 4 байта

По всей видимости, пометка хотя бы одного, или пусть даже нескольких методов в классе как `virtual`, приводит к тому, что размер каждого объекта класса будет увеличен на 4 байта. Откуда они берутся? Вообще, 4 – оптимальное количество байт для хранения адреса какого-нибудь объекта. Запустим отладчик VS 2015.



# Скрин отладчика

```
PugDog d;  
cout << "Адрес объекта d типа PugDog: ";  
cout << &d << "\n";  
  
cout << "Адрес первого поля объекта d: ";  
cout << &d.name << "\n";  
  
cout << "Адрес второго поля объекта d: ";  
cout << &d.age << "\n";  
  
cout << "Адрес третьего поля объекта d: ";  
cout << &d.some_field << "\n";  
}
```

100 %

Видимые

Имя	Значение	Тип
d	{some_field=0 }	PugDog
Dog	{name=0x00000000 <NULL> age=0 }	Dog
_vfptr	0x009874ec { ConsoleApplication1.exe!const PugDog::`vftable' } {0x00982850 {C	void **
[0]	0x00982850 { ConsoleApplication1.exe!PugDog::Guard(void) }	void *
[1]	0x009827b0 { ConsoleApplication1.exe!PugDog::Bark(void) }	void *
name	0x00000000 <NULL>	char *
age	0	int

Видимые Локальные Контрольные значения 1

Готово

# Таблица виртуальных методов

Итак, наличие виртуального метода в классе привело к появлению поля под названием `__vfptr`. Название это расшифровывается как **virtual functions pointer**, или «указатель на таблицу виртуальных методов». На самом деле, это скорее не таблица, а самый обычный одномерный массив, в котором хранятся адреса всех виртуальных методов класса.

# Один класс – одна таблица

Важно понять, что на каждый класс, в котором заявлены виртуальные методы, будет по одной таблице VM. Так, например, компилятор создаёт одну таблицу для класса Dog, и ещё одну таблицу для класса PugDog. В то время, как у каждого объекта этих классов будет по одному указателю на определённую таблицу VM.

# Пример

<https://git.io/vr16A>

Почитать дома:

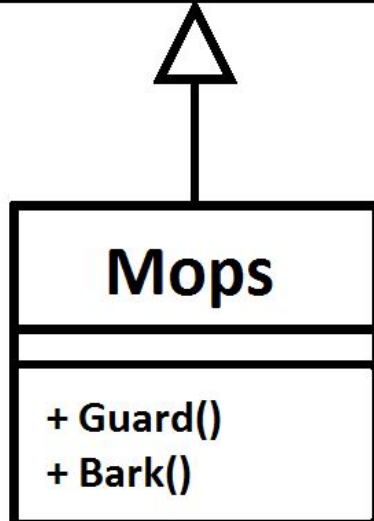
<https://habrahabr.ru/post/51229/>

[https://en.wikipedia.org/wiki/Virtual\\_method\\_table](https://en.wikipedia.org/wiki/Virtual_method_table)

# UML-диаграмма



Method	Implementation
Bark()	Dog::Bark()
Guard()	Dog::Guard()
...	...



Method	Implementation
Bark()	Mops::Bark()
Guard()	Mops::Guard()
...	...

# За всё приходится платить

Виртуальный вызов требует выполнения такой операции, как индексированное разыменование. Поэтому вызов виртуальных методов по сути медленнее, чем вызов не виртуальных. опыты показывают, что примерно 6-13% времени исполнения тратится просто на поиск соответствующего метода.

# virtual в Java

Так как на практике чаще всего ожидается вызов метода именно из класса объекта, а не из класса ссылки на объект, то в Java механизм позднего связывания реализован по умолчанию для всех методов, и помечать их как **virtual** необходимости нет – всё и так работает, как надо. Но «под капотом» всё работает точно также, как и в C++!

# Позднее связывание в Java

Тот же пример, переписанный уже на языке Java, демонстрирует факт, что позднее связывание работает без каких-либо дополнительных действий со стороны программиста:

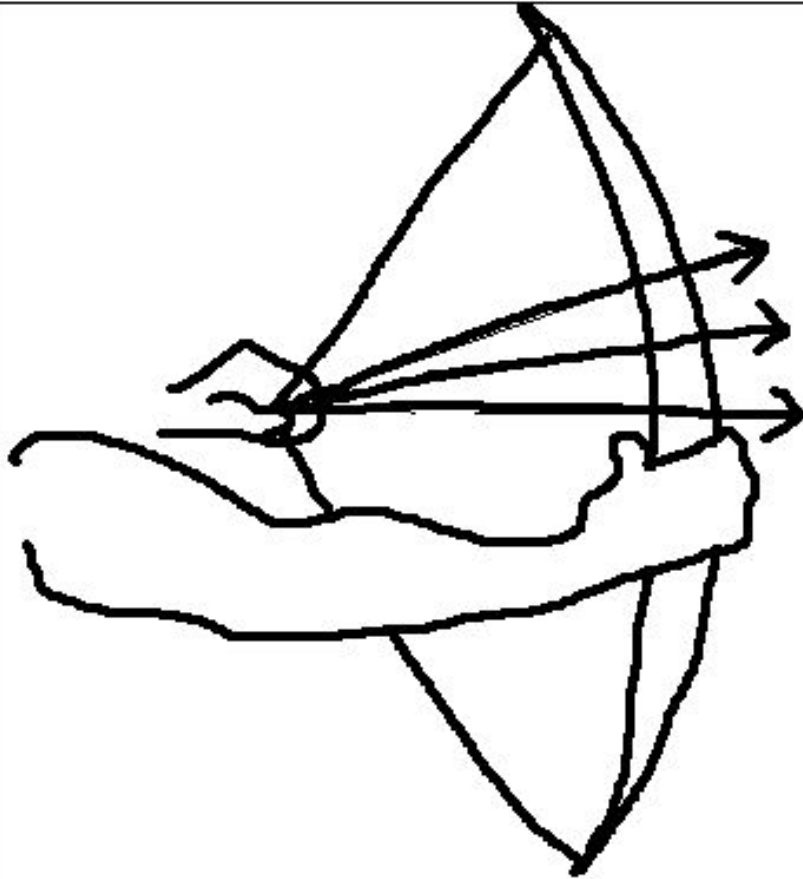
<https://git.io/vr1yz>



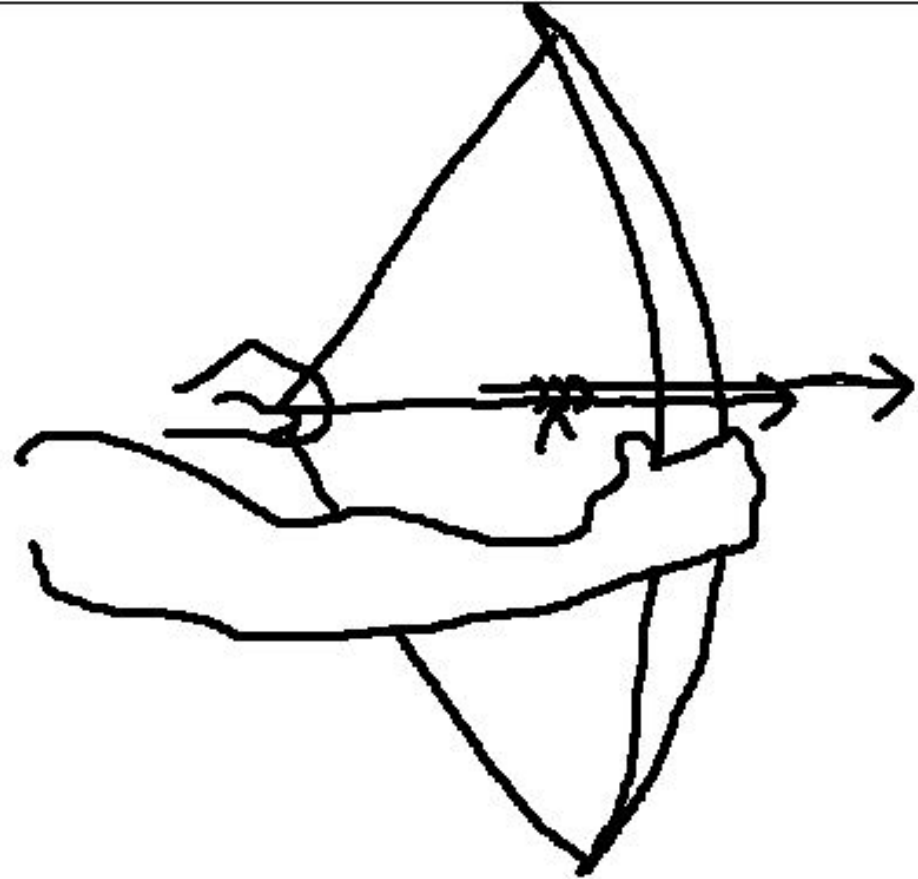
# Запрет переопределения

Существует возможность запретить переопределение метода, пометив его как **final**. Сделано это для того, чтобы гарантированно зафиксировать задуманное поведение метода без возможности его изменения в будущем. А статические методы вообще не участвуют в процессе переопределения (проверить это, пометив метод как **static**).

# overload vs override



**Overloading**



**Overriding**

# Формальное преобразование

Механизм наследования классов предусматривает возможности преобразования типов между суперклассом и подклассом.

Преобразование типов в каком-то смысле является **формальным**. Сам объект при таком преобразовании не изменяется, преобразование относится только к типу ссылки на объект.

# Upcasting и downcasting

Формальное преобразование, от подкласса к суперклассу (**upcasting**):

```
Object o = new Dog();
```

Понижающее преобразование, от суперкласса к подклассу (**downcasting**):

```
Dog d = (Dog)o;
```

# Ограничения downcasting

- Downcasting может задаваться только явно, при помощи операции преобразования типов
- Объект, подвергаемый преобразованию, реально должен быть того класса, к которому он преобразуется. Если это не так, то возникнет исключение **ClassCastException**.

# instanceof

В Java для проверки типа объекта есть операция **instanceof**. Она часто применяется при понижающем преобразовании (downcasting). Эта операция проверяет отношение левого операнда к классу, заданному правым операндом.

```
if (o instanceof Dog) return true;
```

Оператор **instanceof** относится к механизму **динамической идентификации типа данных** (run-time type information, run-time type identification), который позволяет определить тип данных объекта во время выполнения программы.

[https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F\\_%D0%B8%D0%B4%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F\\_%D1%82%D0%B8%D0%BF%D0%B0\\_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%B8%D0%B4%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F_%D1%82%D0%B8%D0%BF%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)