Программирование на C++. Часть 2
Лекция 5

ПМИ Семестр 2

Демяненко Я.М.

2026

Как объединить в одной коллекции объекты различных наследников базового класса?

```
class Person {
   …
public:
  showInform();
  …
};
```

```
class Student: public
Person {
   …
public:
  showInform();
  …
};
```

```
class Professor : public
Person {
   …
public:
  showInform();
  …
};
```

Создать контейнер, содержащий указатели (или ссылки) на объекты базового класса

```
Person *p [5];

p[0] = new Student("Петров", 19, "МГУ", 3);
p[1] = new Student("Чайкин", 18, "ЮФУ", 3);
p[2] = new Professor ("Тьюринг", 32, "ИВЭ");
p[3] = new Professor ("Страуструп", 32, "ПМП");
p[4] = new Student("Вишняк", 20, " ЮФУ ", 3);

for ( int i=0; i<5; ++i)
    p[i]->showInform();
```

Реализация какого из классов функции **showInform()** отработает?

Раннее (статическое) связывание

Это обусловлено тем, что определение, какую из переопределённых функций вызвать, происходит в **момент компиляции** по **типу переменной-объекта или переменной-ссылки (указателя)**

Определение на этапе компиляции вызываемого варианта переопределённой функции называется статическим, или ранним, связыванием

А что хотели увидеть?

Поскольку в классах Student и Professor функция showInform переопределена,
то хотелось бы при вызове

p[0]->showInform() увидеть информацию о студенте Петрове, а при вызове

p[2]->showInform() — о профессоре Тьюринге

Динамическое (позднее) связывание

Для того чтобы обеспечить возможность определения, какая из переопределённых функций должна быть вызвана, не на основе типа переменной-ссылки или указателя, а **на основе типа объекта, на который они ссылаются**, нужен другой механизм — **динамическое, или позднее, связывание**

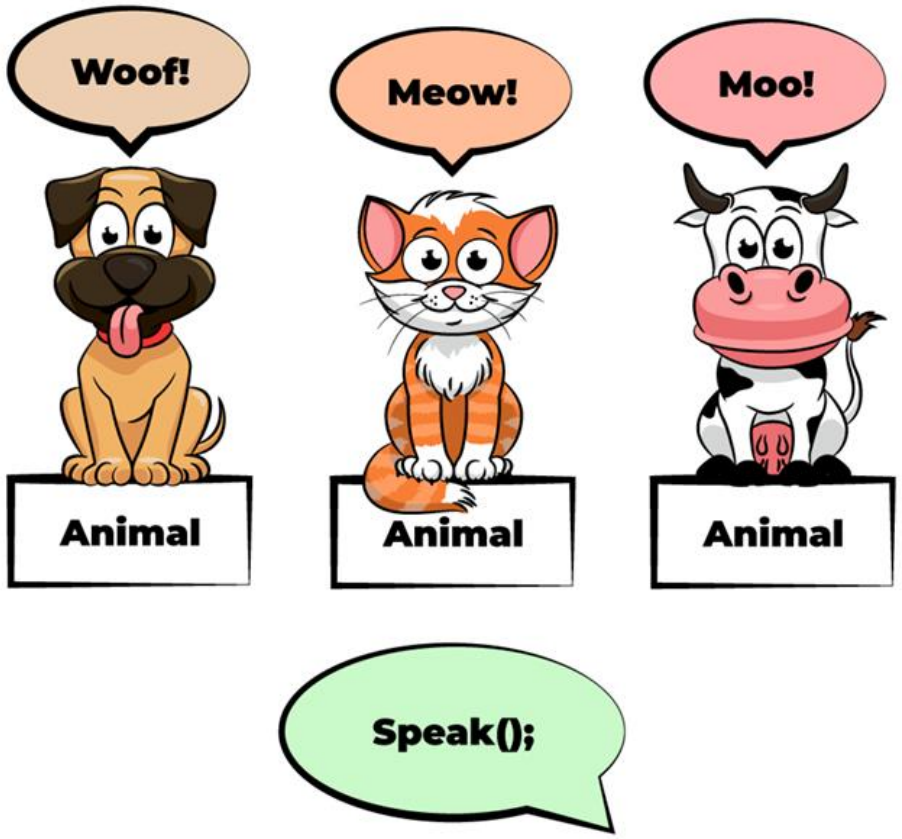
Определение на этапе выполнения вызываемого варианта переопределённой функции называется динамическим, или поздним, связыванием

Полиморфизм

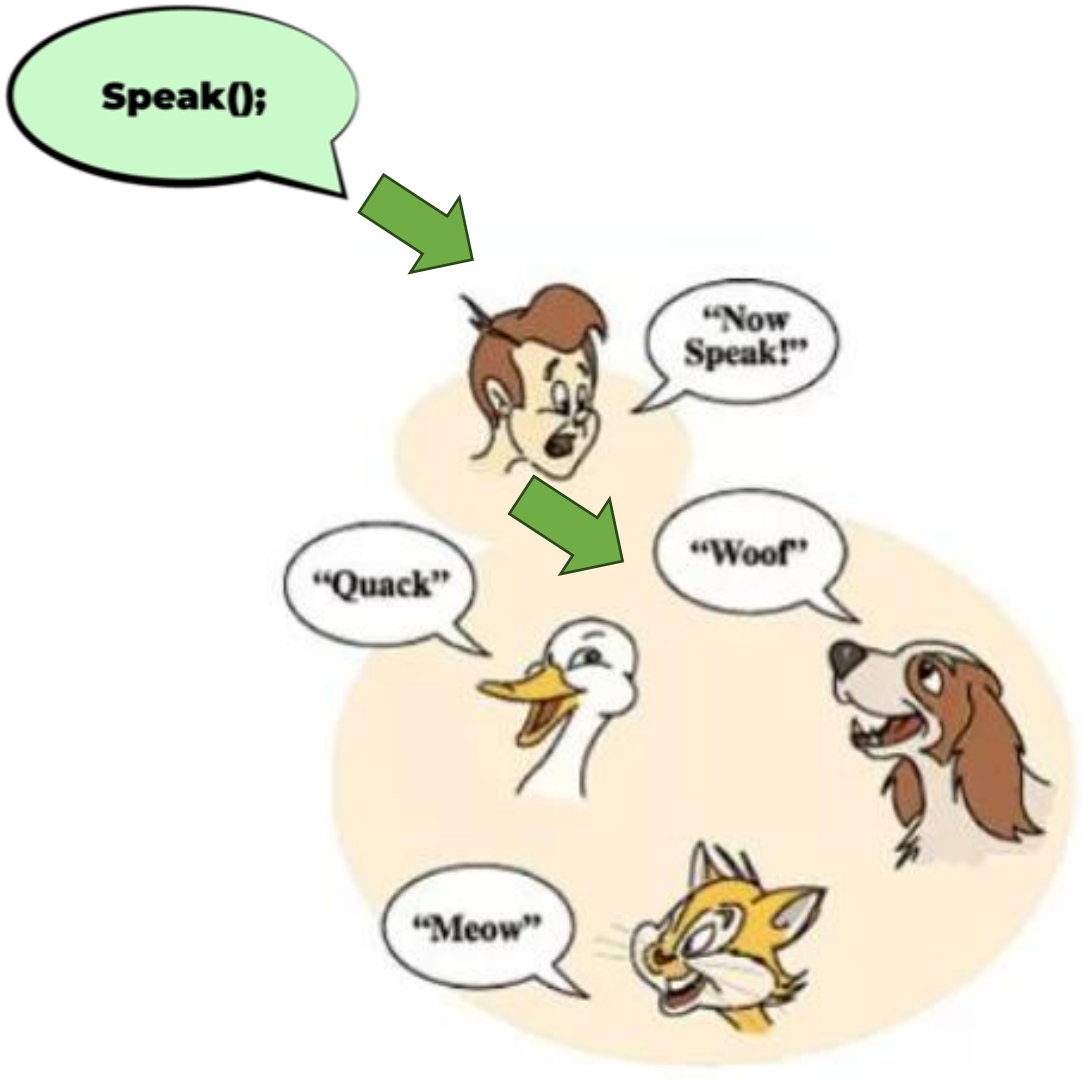
Позднее связывание реализует принцип полиморфизма

Полиморфизм позволяет выбирать **вариант** вызываемой **функции** в ходе **выполнения** программы

Различные объекты используются одинаковым образом



Большая глубина



Виртуальные функции

Позднее связывание реализуется с помощью виртуальных функций

```
class Person{
  public:
    //все, как было описано выше, кроме функции showInform
    ...
    virtual void showInform();
  private:
    //все, как было описано выше
};
```

Реализация функции showInform как для класса Person, так и для классов-наследников остаётся без изменений

Вызов виртуальных функций

```
for ( int i=0; i<5; ++i)
    p[i]->showInform();
```

Виртуальная функция всегда виртуальна

Если объявление функции в базовом классе начинается с ключевого слова virtual, то это делает функцию виртуальной для **базового** класса и **всех** классов, **производных** от базового класса. Виртуальная функция всегда виртуальна.

Рекомендуется всегда использовать спецификатор virtual в объявлении виртуальных функций, независимо от их расположения в иерархии наследования. Это улучшает читаемость кода.

Предупреждение

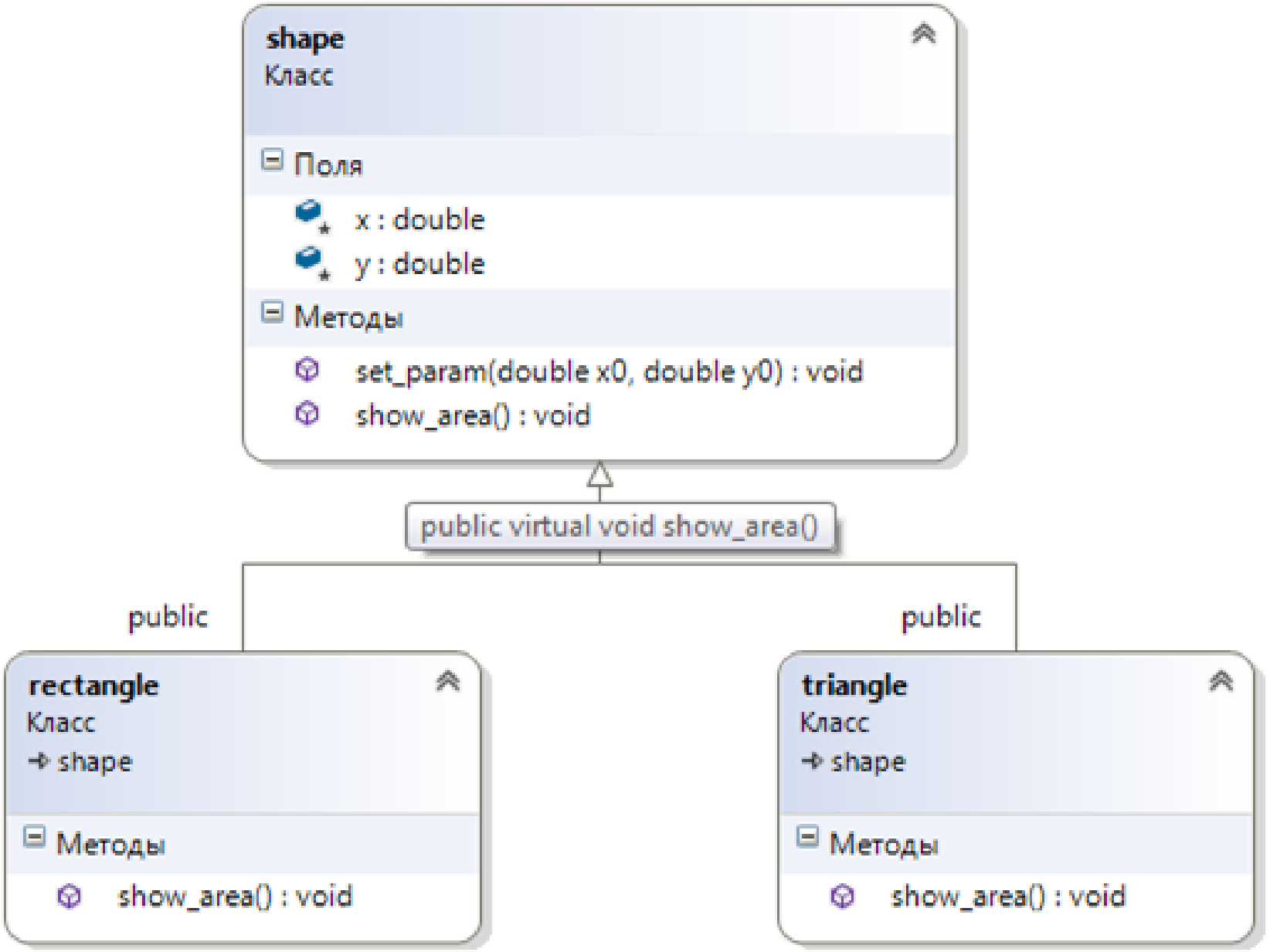
```
Person p;
Student s("Вишняк", 20, " ЮФУ ", 3);

p = s;
p.showInform();
```

будет выполнено раннее связывание, несмотря на то, что функция showInform() виртуальная

Вывод:
Полиморфизм C++ работает **только через указатели и ссылки** на объекты базового класса

Пример. Создать базовый класс shape. Создать два класса-потомка rectangle (две стороны) и triangle (сторона и высота к ней)

В этом примере наследование **не** имеет целью **расширение** базового класса.
Каждый из классов-наследников **реализует свое собственное поведение** на основе виртуальной функции, объявленной в базовом классе.


shape
Класс
Поля
x : double
y : double
Методы
set_param(double x0, double y0) : void
show_area() : void
public virtual void show_area()
public
public
rectangle
Класс
shape
Методы
show_area() : void
triangle
Класс
shape
Методы
show_area() : void

Базовый класс shape

```
class shape {
  protected:
    double x,y;
  public:
    //конструкторы не создаём, так как используется автоматический конструктор по умолчанию
    void set_param(double x0, double y0) {
      x=x0;
      y=y0;
    }
    virtual void show_area(){
      cout<<"Area calculation for this class is undefined";
      cout <<endl;
    }
};
```

Потомки rectangle и triangle

```
class triangle: public shape {
  public:
    //используется автоматический конструктор по умолчанию
    //вызывающий конструктор по умолчанию базового класса
    void show_area() {
      cout<<endl<<"triangle with height "<<x<<" and base "<<y;
      cout<<endl<<"has an area "<<x*0.5 *y<<endl;
    }
};

class rectangle : public shape {
  public:
    //используется автоматический конструктор по умолчанию
    //вызывающий конструктор по умолчанию базового класса
    void show_area() {
      cout<<endl<<"rectangle with sides "<<x<<" * "<<y;
      cout<<endl<<"has an area "<<x*y<<endl;
    }
};
```

Проанализируем

```
int main() {
  shape *p[2];
  triangle t;
  rectangle r;

  p[0]=&t;
  p[0]->set_param(10.0,5.0);
  p[0]->show_area();

  p[1]=&r;
  p[1]->set_ param (10.0,5.0);
  p[1]->show_area();

  for (auto x: p)
    tellMeAboutYourself(x);
  return 0;
}
```

```
void tellMeAboutYourself(shape *s) {
  s-> show_area();
}
```

auto

```
for (переменная: массив)
  statement;
```

```
for (auto x: p)
  tellMeAboutYourself(x);
```

Для пользовательских типов данных, используемых в качестве параметра цикла, рекомендуется применять **auto**

До C++ 11 auto использовалось для явного указания, что переменная имеет автоматический класс памяти (время существования ограничено блоком, в котором она определена).
Однако начиная с C++ 11 все переменные по умолчанию имеют автоматический класс памяти

Начиная с C++ 11, ключевое слово auto используется для вывода типа (автоматического определения типа)

Цена виртуальности

Полиморфизм в C++ реализуется с помощью таблиц виртуальных функций — Virtual Methods Table(VMT)

В каждом объекте появляется **дополнительный указатель** vptr на **таблицу виртуальных методов**.

Для каждого класса создается таблица виртуальных методов, которая содержит **адреса всех виртуальных методов этого класса и всех его предков**.

Если в классе и его предках нет виртуальных методов, то в его объекте поле vptr отсутствует (реализуется принцип: не платим за то, что не используем).

Замечание:
В C++ нет общего главного класса-предка, как например, Object в PascalABC.Net, C#, Java.
Это делается в целях повышения эффективности.
Так как общий предок предоставляет набор виртуальных функций (методов), что приводит к созданию VMT для каждого класса.

Последовательность выполнения

Как работает **new**: сначала выделяется **память**, потом вызывается **конструктор**.

Как работает **delete**: сначала вызывается **деструктор**, потом возвращается **память**.

Что будет?

```
Person * pp = new Student("Петров", 19, "МГУ", 3);
pp -> showInform();
delete pp;
```

Будет ...

```
Person * pp = new Student("Петров", 19, "МГУ", 3);
pp -> showInform();
delete pp;
```

В данном случае, при выполнении delete pp; будет вызван деструктор ~Person(), что плохо.

Деструкторы в C++ тоже могут быть виртуальными, однако по умолчанию они таковыми не являются.

Как исправить ситуацию?

Сделаем деструктор для класса Person виртуальным

```
class Person {
    …
    virtual void Print() { ... }
    virtual ~Person() { ... }
}
```

Теперь, при освобождении памяти, на которую ссылается pp, вызовется деструктор ~Student().

Если деструктор у предка виртуальный, и если при этом для класса Student не написать деструктор, то он сгенерируется автоматическим и будем виртуальным.

Деструкторы и полиморфизм

Правило:

Если в классе есть **хотя бы одна виртуальная функция**, тогда обязательно делаем его **деструктор виртуальным**.

Если базовый класс не имеет явного деструктора, а у потомков класса появятся явные деструкторы, например, освобождающие динамическую память, то возможна ошибка утечки памяти.

Важно гарантировать, чтобы при уничтожении объекта был вызван деструктор именно того класса-наследника, к которому он относится.

Если базовый класс не требует выполнения явного деструктора, не следует полагаться на деструктор по умолчанию.

Вместо этого необходимо описать виртуальный деструктор с пустой реализацией.

Интерфейсы

Вспомним

```
void tellMeAboutYourself(shape *s) {
  s-> show_area();
}
```

Между функцией tellMeAboutYourself и наследниками класса shape устанавливается **соглашение по возможностям взаимодействия**.

Такое соглашение называется **интерфейсом**.

А есть ли смысл в функции show_area для класса shape?...

```
class shape {
  protected:
    double x,y;
  public:
    //конструкторы не создаём, так как используется автоматический конструктор по умолчанию
    void set_param(double x0, double y0) {
      x=x0;
      y=y0;
    }

    //нет смысла, лишь способ корректно сообщить об ошибке использования
    virtual void show_area(){
      cout<<"Area calculation for this class is undefined";
      cout <<endl;
    }
};
```

Виртуальная функция без реализации

Объект класса shape без конкретизации типа фигуры **не может иметь площади**.
Для того чтобы не прибегать к такому искусственному приёму, имеется возможность определять **виртуальную функцию без реализации**.

Виртуальная функция без реализации называется чисто виртуальной.

Чисто виртуальная функция

Синтаксис объявления чисто виртуальной функции — **virtual <тип> <имя> (<список параметров>) = 0;**

```
class shape {
  protected:
    double x,y;
  public:
    //конструкторы не создаём, так как используется автоматический конструктор по умолчанию
    void set_param(double x0, double y0) {
      x=x0;
      y=y0;
    }

     virtual void show_area() =0;
};
```

Если класс имеет хотя бы одну чисто виртуальную функцию, его называют абстрактным классом.

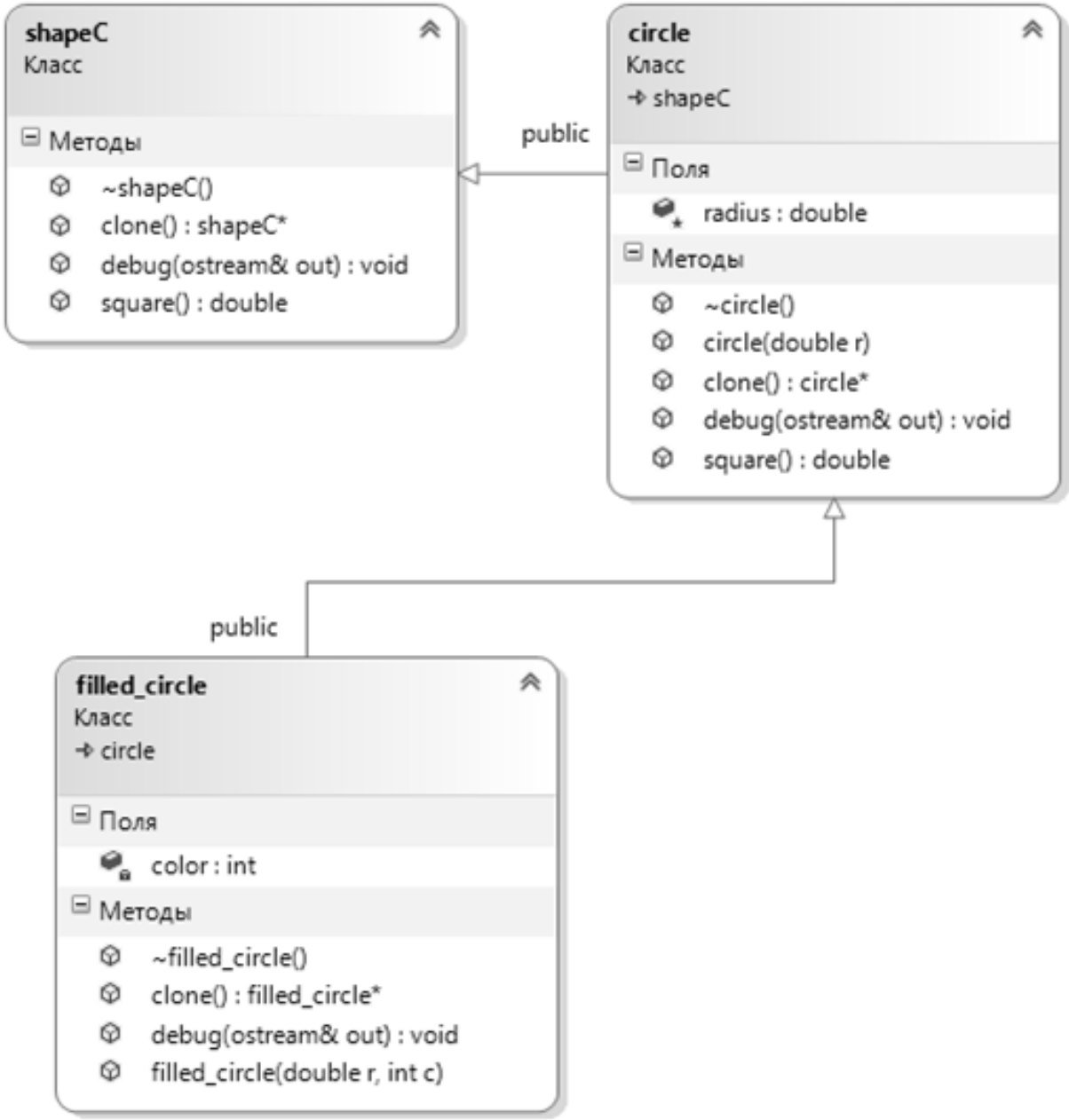
Абстрактный класс

- Для абстрактного класса **не могут быть созданы объекты**
- Такой класс может служить **только в качестве базового в системе наследования** и для создания указателей и ссылок, которые будут использованы при реализации полиморфизма.
- Если в базовом классе имеется **чисто виртуальная функция**, **производный** класс **должен иметь определение её** собственной **реализации**.
- Если **реализация** хотя бы одной из чисто виртуальных функций **не будет выполнена**, **производный** класс, в свою очередь, **останется абстрактным**

Интерфейсы и абстрактные классы

Абстрактные классы посредством чисто виртуальных функций описывают интерфейс, который можно использовать в других классах или функциях, ожидая, что наследники реализуют эти функции.

Пример. Создать абстрактный класс shapeC и его наследников: классы circle и filled_circle.


shapeC
Класс
Методы
~shapeC()
clone() : shapeC*
debug(ostream& out) : void
square() : double
public
circle
Класс
shapeC
Поля
radius : double
Методы
~circle()
circle(double r)
clone() : circle*
debug(ostream& out) : void
square() : double
public
filled_circle
Класс
circle
Поля
color : int
Методы
~filled_circle()
clone() : filled_circle*
debug(ostream& out) : void
filled_circle(double r, int c)

Абстрактный класс shapeC

```
class shapeC {
  public:
    //используется автоматический конструктор по умолчанию
    virtual ~shapeC() {}
    virtual double square() const =0;
    virtual shapeC* clone() const =0;
    virtual void debug(ostream &out) const=0;
};
```

Наследники circle и filled_circle

```
class circle: public shapeC {
  public:
    circle(double r=0): radius(r) { }
    ~circle() {}
    double square() const {
      return 3.14*radius*radius;
    }
    circle* clone() const {
      return new circle(radius);
    }
    void debug (ostream & out) const {
      out<<" radius = "<<radius <<endl;
    }
  protected:
    double radius;
};
```

```
class filled_circle: public circle {
  public:
    filled_circle (double r, int c): circle(r), color(c) { }
    ~filled_circle() {}
    filled_circle* clone() const {
      return new filled_circle(radius, color);
    }
    void debug (ostream & out) const {
      circle::debug(out);
      out<<" color = "<<color <<endl;
    }
  private:
    int color;
};
```

Полиморфный контейнер

```
int main() {
shapeC *p[4];

  circle t(5);
  filled_circle r(10, 255);

  p[0] = &t;
  p[1] = &r;

  p[2] = p[0]->clone();
  p[3] = p[1]->clone();

  for (auto x : p)   {
    x->debug(cout);
  }
  return 0;
}
```

Массив **p** указателей на shapeC, представляет собой **полиморфный контейнер**, поскольку он может содержать указатели на circle и filled_circle.

Имея объекты наследников класса shapeC, можно их адреса присвоить элементам массива p.

Копии элементов полиморфного контейнера

```
p[0] = &t;
p[1] = &r;
```

Если потребуется создать копии двух элементов полиморфного контейнера, операция присваивания не поможет, поскольку будет выполнено присваивание адресов.

```
p[2] = p[0];
p[3] = p[1];
```

В этом случае p[2] и p[0] будут ссылаться на один и тот же объект t, а p[3] и p[1] — на объект r.

Решение проблемы копирования

Решением данной проблемы могло бы быть создание виртуального конструктора, однако **конструкторы в C++ не могут быть виртуальными**.

Поэтому для решения этой проблемы следует использовать функцию **clone()**.

Клонирование является полиморфным, так как объект должен клонировать себя, а не объект базового типа.

```
p[2] = p[0]->clone();
p[3] = p[1]->clone();
```

Функция clone()

virtual shapeC* clone() const =0;

```
class circle: public shapeC {
  public:
...
  circle* clone() const {
    return new circle(radius);
  }
. ..
};
class filled_circle: public shapeC {
  public:
...
  filled_circle* clone() const {
    return new filled_circle(radius, color);
  }
...
};
```

```
p[2] = p[0]->clone();
p[3] = p[1]->clone();
```

Ограничения

Конструкторы не могут быть виртуальными.
Производный класс не наследует конструкторы базового класса, поэтому бессмысленно делать их виртуальными.

Статические функции также не могут быть виртуальными.

Виртуальное наследование

Виртуальное наследование (virtual inheritance) **предотвращает** появление **множественных объектов базового класса** в иерархии наследования.

```
class Device {
    public:
        Device() {
            cout << "Device constructor called" << endl;
        }
        void turn_on() {
            cout << "Device is on." << endl;
        }
};
```

```
class Computer: virtual public Device {
    public:
        Computer() {
            cout << "Computer constructor called" << endl;
        }
};
```

```
class Monitor: virtual public Device {
    public:
        Monitor() {
            cout << "Monitor constructor called" << endl;
        }
};
```

```
class Laptop: public Computer, public Monitor {};

int main() {
    Laptop Laptop_instance;
    Laptop_instance.turn_on();
    return 0;
}
```

Конструктор базового класса Device будет вызван только единожды,
а обращение к методу turn_on() без его переопределения в дочернем классе не будет вызывать ошибку при компиляции.

Без виртуального наследования

//class Laptop : public Computer, public Monitor, public Device {};

Ошибка (активно) E0266 "Laptop::turn_on" не является однозначным

Предупреждение C4584 Laptop: базовый класс "Device" уже является базовым классом для "Computer"

Ошибка C2385 неоднозначный уровень доступа "turn_on"

Одна цепочка виртуальности или две?

```
class B {
public:
   virtual void f(short) { std::cout << "B::f" << std::endl; }
};

class D : public B {
public:
   virtual void f(int) { std::cout << "D::f" << std::endl; }
};
```

А теперь?

```
class B  {
public:
  virtual void f(int) const {std::cout << "B::f " << std::endl;}
};

class D : public B  {
public:
  virtual void f(int) {std::cout << "D::f" << std::endl;}
};
```

virtual, override и final

virtual — начало цепочки виртуальных методов

override — метод является переопределением виртуального метода в базовом классе

final — конец цепочки виртуальных методов (производный класс не должен переопределять виртуальный метод)

Использование override

```
class B {
public:
   virtual void f(short) { std::cout << "B::f" << std::endl; }
};

class D : public B {
public:
   virtual void f(int) override { std::cout << "D::f" << std::endl; }
};
```

или

```
class D : public B {
public:
    void f(int) override { std::cout << "D::f" << std::endl; }
};
```

Ошибка C3668 D::f: метод со спецификатором переопределения "override" не переопределяет какие-либо методы базового класса

Рекомендации по virtual и override

Отмечайте **виртуальные** функции **или** как **virtual (начало цепочки)**, **или** как **override (переопределение)**, но не то и другое одновременно.

```
class B {
public:
   virtual void f(int) { std::cout << "B::f" << std::endl; }
};

class D : public B {
public:
    void f(int) override { std::cout << "D::f" << std::endl; }
};
```

Использование final

```
class B {
public:
   virtual void f(int) { std::cout << "B::f" << std::endl; }
};

class D : public B {
public:
   virtual void f(int) override final { std::cout << "D::f" << std::endl; }
};

class F : public D {
public:
   virtual void f(int) override { std::cout << "F::f" << std::endl; }
};
```

Ошибка C3248 D::f: функцию, объявленную как "final", нельзя переопределить с помощью "F::f"

Рекомендации

```
class B {
public:
   virtual void f(int) { std::cout << "B::f" << std::endl; }
};

class D : public B {
public:
     void f(int) override { std::cout << "D::f" << std::endl; }
};

class F : public D {
public:
      void f(int) final { std::cout << "F::f" << std::endl; }
};
```

Что не так?

```
class B {
public:
   virtual void f(int) { std::cout << "B::f" << std::endl; }
};

class D : public B {
public:
    void f(int) override { std::cout << "D::f" << std::endl; }
};

class F : public D {
public:
   int f(int) final { std::cout << "F::f" << std::endl; }
};
```

```
class B {
public:
    virtual void f(int) { std::cout << "B::f" << std::endl; }
};

class D : public B {
public:
    void f(int) override { std::cout << "D::f" << std::endl; }
};

class F : public D {
public:
    int f(int) final { std::cout << "F::f" << std::endl; }
};
```

Ошибка C2555 F::f: возвращаемый тип переопределенной виртуальной функции отличается от "D::f" и не является **ковариантным**

Ковариантные возвращаемые типы

Если **тип возвращаемого** значения **виртуальной** функции является **указателем** или **ссылкой** на класс, **переопределяющие** функции могут возвращать **указатель** или **ссылку** на **производный** класс.
Это называется **ковариантными возвращаемыми типами**.

Ковариантные возвращаемые типы. Пример

```
class circle: public shapeC {
  public:
    circle(double r=0): radius(r) { }
    ~circle() {}
    double square() const {
      return 3.14*radius*radius;
    }
    circle* clone() const override {
      return new circle(radius);
    }
    void debug (ostream & out) const {
      out<<" radius = "<<radius <<endl;
    }
  protected:
    double radius;
};
```

```
class filled_circle: public circle {
  public:
    filled_circle (double r, int c): circle(r),
color(c) { }
    ~filled_circle() {}
    filled_circle* clone() const override {
      return new filled_circle(radius, color);
    }
    void debug (ostream & out) const {
      circle::debug(out);
      out<<" color = "<<color <<endl;
    }
  private:
    int color;
};
```