



HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

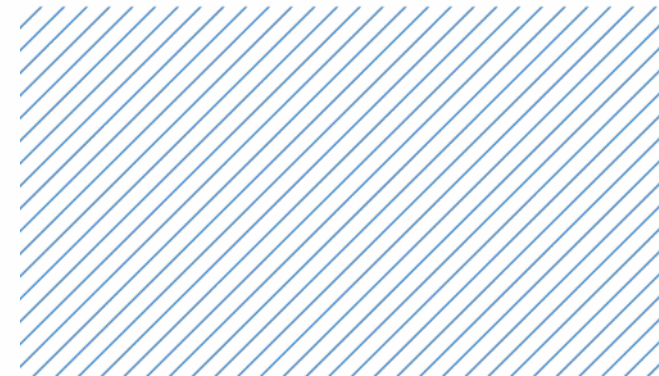
Лекция 4

Наследование

Программирование на языке C++

Константин Леладзе

ФКН ВШЭ





Три ключевых принципа ООП

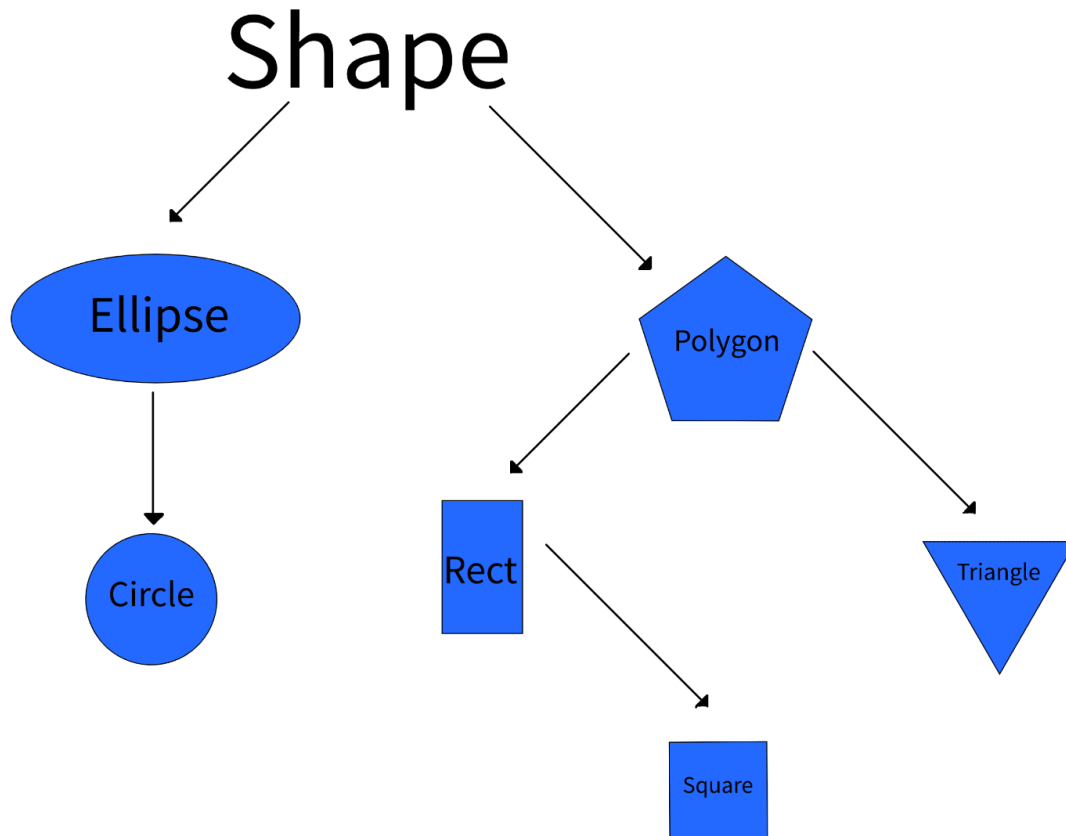
Существует три ключевых принципа ООП. Они присутствуют не только в языке C++, но и во всех объектно-ориентированных языках. Наличие **механизмов реализации** этих принципов в языке и определяет, является ли он объектно-ориентированным, или нет.

Перечислим эти принципы и **механизмы их реализации** в языке C++:

- Инкапсуляция: **классы, спецификаторы доступа**
- Полиморфизм:
 - Статический полиморфизм (compile-time): **шаблоны, наследование**
 - Динамический полиморфизм (runtime): **виртуальные методы и абстракции**
- Наследование: **наследование**

Зеленым цветом помечено то, что мы уже изучили. Синим – то, что будем изучать в этой лекции. Начнем с наследования...

Наследование



Наследование, – одна из ключевых особенностей ООП и C++. Она позволяет нам наследовать свойства (поля) и поведение (методы) классов и организовывать иерархии классовых сущностей.

Класс, наследующий другой, называется наследником, а его предок, – базовым классом.

Наследование – один из механизмов полиморфизма в C++, который позволяет обобщать сущности и писать универсальный код.

На практике, помимо полиморфизма, использование наследования помогает избежать лишнего кода. Без наследования, нам бы приходилось каждый раз писать один и тот же код для того, чтобы поддержать необходимое поведение класса.

Наследование

Объявим следующую иерархию классов и протестируем наследование:

```
struct animal {
    void eat () {
        std::cout << "animal is eating" << std::endl;
    }

    void sleep () {
        std::cout << "animal is sleeping" << std::endl;
    }
};

struct dog : animal {
    void bark () {
        std::cout << "bark bark..." << std::endl;
    }
};

struct cat : animal {
    void meow () {
        std::cout << "meow meow..." << std::endl;
    }
};
```

```
int main () {
    dog dog_instance;

    /// dog can do everything what an animal can:
    dog_instance.eat();
    dog_instance.sleep();
    /// but also dog can bark
    dog_instance.bark();
    /// however, he can't meow:
    dog_instance.meow();

    cat cat_instance;

    /// cat can do everything what an animal can:
    cat_instance.eat();
    cat_instance.sleep();
    /// but also cat can meow
    cat_instance.meow();
    /// however, he can't bark:
    cat_instance.bark();
}
```

Кошка и собака унаследовали способность есть и спать у животного, но у них также есть и свои уникальные способности, – лаять и мяукать соответственно.

Пока что, нам уже очевидна как минимум одна мотивация использования наследования: мы пишем меньше кода. Вернее, мы вообще не пишем лишнего кода, чтобы “научить” собаку и кошку лаять и мяукать соответственно.

Поиск функции по имени при наследовании

```
#include <iostream>

struct base {
    void foo () {
        std::cout << "hello from base::foo" << std::endl;
    }
};

struct derived : base {
    void foo () {
        std::cout << "hello from derived::foo" << std::endl;
    }
};

int main () {
    base b;
    b.foo();

    derived d;
    d.foo();
}
```

Если в наследнике определить метод, имеющий такое же название, что и метод в базовом классе, то ошибки **не** возникнет, несмотря на кажущееся наличие некоей неоднозначности при вызове метода на наследнике.

Оказывается, что в этом случае, поведение при вызове метода на наследнике схоже с поведением при перегрузке: при наследовании, метод наследника “скрывает” метод базового класса.

```
hello from base::foo
hello from derived::foo
```

```
Process finished with exit code 0
```

Поиск функции по имени при наследовании

Если же требуется вызвать метод именно базового класса, то можно использовать оператор разрешения области видимости:

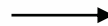
```
int main () {  
    derived d;  
    d.base::foo();  
}
```



```
hello from base::foo  
  
Process finished with exit code 0
```

Более того, это работает и в случае наличия нескольких предков в иерархии наследования:

```
struct granny {  
    ...  
};  
struct mommy : granny {  
    ...  
};  
struct son : mommy {  
    ...  
};  
  
int main () {  
    son s;  
    s.granny::foo();  
    s.mommy::foo();  
    s.son::foo(); /// equivalent to s.foo()  
}
```



```
hello from granny::foo  
hello from mommy::foo  
hello from son::foo  
  
Process finished with exit code 0
```

Физическое устройство наследования

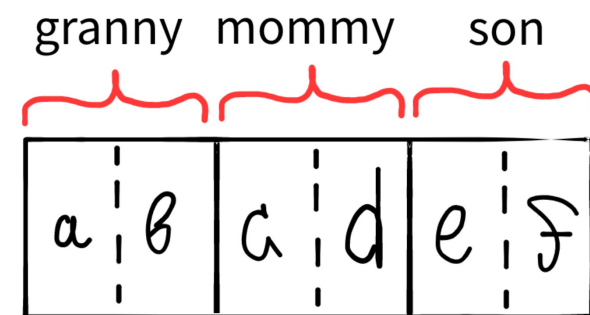
Очевидно, раз мы можем вызвать какой-либо метод родителя, а также обратиться к родительским полям, значит они где-то хранятся физически в памяти, ассоциированный с экземпляром наследника. Изучим физическое устройство следующей иерархии наследников:

```
struct granny {  
    int a, b;  
};  
  
struct mommy : granny {  
    int c, d;  
};  
  
struct son : mommy {  
    int e, f;  
};
```

```
int main () {  
    son s;  
    std::cout <<  
        &s.a << std::endl <<  
        &s.b << std::endl <<  
        &s.c << std::endl <<  
        &s.d << std::endl <<  
        &s.e << std::endl <<  
        &s.f << std::endl;  
}
```

```
0x7ff7bb965848  
0x7ff7bb96584c  
0x7ff7bb965850  
0x7ff7bb965854  
0x7ff7bb965858  
0x7ff7bb96585c
```

Из результатов нашего эксперимента следует, что данные в экземплярах наследника хранятся последовательно и разделены на три блока, ассоциированных с соответствующими классами. Справа видна схема, демонстрирующая устройство памяти внутри любого экземпляра класса **son**.



Примечание: приведенная справа модель хранения не стандартизирована, то есть полагаться на то, что поля будут храниться строго последовательно и без зазоров – нельзя

Инициализация базового класса при наследовании

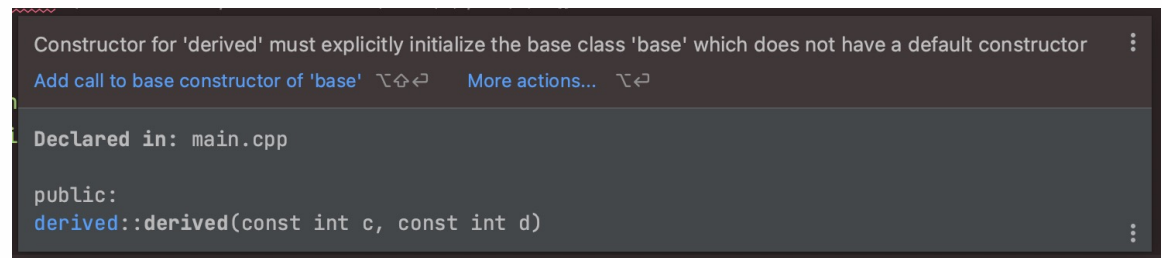
Из предыдущего слайда следует, что на физическом уровне, наследник будто бы хранит “внутри себя” своего предка, а значит этого предка нужно как-то инициализировать.

Изучим приведенный код:

```
struct base {  
    int a, b;  
    base (const int a, const int b): a(a), b(b) {}  
};  
  
struct derived : base{  
    int c, d;  
    derived (const int c, const int d): c(c), d(d) {}  
};  
  
int main () {  
    derived d(1, 2);  
}
```

```
struct derived : base{  
    int c, d;  
    derived (int a, int b, int c, int d): base(a, b), c(c), d(d) {}  
};  
  
int main () {  
    derived d(1, 2, 3, 4);  
}
```

При компиляции, происходит ошибка:



Очевидно, проблема в том, что поля **a** и **b**, относящиеся к базовому классу, но тем не менее присутствующие у наследника нигде не инициализируются.

Ошибка исправляется простым вызовом конструктора базового класса.

Примечание: в этом случае, конструктор **не** является делегирующим, что позволяет нам проинициализировать и поля наследника в том же списке инициализации.

Порядок вызова конструкторов при наследовании

Из предыдущего слайда следует, что инициализация при наследовании происходит в порядке от ребенка к родителю.

Проверим это, проведя простой эксперимент:

```
struct granny {  
    granny () { std::cout << "Hello from granny" << std::endl; }  
};  
  
struct mommy : granny {  
    mommy () { std::cout << "Hello from mommy" << std::endl; }  
};  
  
struct son : mommy {  
    son () { std::cout << "Hello from son" << std::endl; }  
};  
  
int main () {  
    son s;  
}
```




```
Hello from granny  
Hello from mommy  
Hello from son
```

Порядок вызова деструкторов при наследовании

Деструкторы вызываются в обратном порядке.

Проверим это, проведя простой эксперимент:

```
struct granny {  
    granny () { std::cout << "Hello from granny" << std::endl; }  
    ~granny () { std::cout << "Hello from ~granny" << std::endl; }  
};  
  
struct mommy : granny {  
    mommy () { std::cout << "Hello from mommy" << std::endl; }  
    ~mommy () { std::cout << "Hello from ~mommy" << std::endl; }  
};  
  
struct son : mommy {  
    son () { std::cout << "Hello from son" << std::endl; }  
    ~son () { std::cout << "Hello from ~son" << std::endl; }  
};  
  
int main () {  
    son s;  
}
```



```
Hello from granny  
Hello from mommy  
Hello from son  
Hello from ~son  
Hello from ~mommy  
Hello from ~granny
```

Спецификаторы доступа наследования

```
struct rectangle {
    rectangle (const double a, const double b): a(a), b(b) {}
    double get_a () const { return a; }
    double get_b () const { return b; }

private:
    double a;
    double b;
};

struct square : rectangle {
    square (const double side): rectangle(side, side) {}
    double get_side () const { return side; }

private:
    double side;
};

int main () {
    square sq(2.);
    sq.get_side(); /// OK
    /// But we shouldn't be allowed to call these methods by-design
    sq.get_a();
    sq.get_b();
}
```

В этом примере, использование наследования приводит к тому, что все унаследованные методы становятся доступны внешнему пользователю, однако это не всегда удобно.

Например, в этом случае, интерфейс класса **square** кажется пользователю излишне перегруженным. Действительно, методы **get_a** и **get_b** бесполезны, ведь их функционал заменен единым методом **get_side**.

Нам бы хотелось управлять поведением нашего наследования и, самое важное, управлять группами доступа унаследованных методов.

Например, в нашем случае проблема решилась бы просто, если бы мы изменили доступ унаследованных методов с **public** на **private**.

Спецификаторы доступа наследования

```
struct rectangle {
    rectangle (const double a, const double b): a(a), b(b) {}
    double get_a () const { return a; }
    double get_b () const { return b; }

private:
    double a;
    double b;
};

struct square : private rectangle {
    square (const double side): rectangle(side, side) {}
    double get_side () const { return side; }

private:
    double side;
};

int main () {
    square sq(2.);
    sq.get_side(); // OK
    // But we shouldn't be allowed to call these methods by-design
    sq.get_a();
    sq.get_b();
}
```

get_a is a private member of 'rectangle' constrained by private inheritance here member is declared here

Приватное наследование позволяет скрыть все унаследованные методы от пользователя: они просто становятся приватными.

Помимо **private** наследования, существуют **public** и **protected** наследования.

Public-наследование мы уже рассматривали до текущего момента.

***Важно:** по-умолчанию, **структуры** наследуются со спецификатором **public**, если не указано иного. Классы же наследуются со спецификатором **private**, если не указано иного. Это и есть второе и последнее отличие классов от структур.*

Прежде чем рассмотреть **protected**-наследование, изучим что же это за модификатор доступа, и в чем его предназначение...

Модификатор доступа protected

Представим ситуацию: существует базовый класс, у которого есть приватное поле, которое мы хотим использовать в наследнике. Возникает проблема: **приватные поля и методы не наследуются ни при каких обстоятельствах**:

```
struct base {  
    void set_a (const int value) { a = value; }  
  
private:  
    int a;  
};  
  
struct derived : base {  
    int get_a () { return a; }  
};  
  
int main () {  
    derived d;  
    d.set_a(123);  
    std::cout << d.get_a() << std::endl;  
}
```

'a' is a private member of 'base'
[declared private here](#)

Простым решением было бы просто сделать поле публичным, однако это идет вразрез с принципом инкапсуляции данных...

Модификатор доступа protected

В этом случае, на помощь нам приходит модификатор доступа **protected**. Отличие **protected** от **private** в том, что **protected** поле все же наследуется, однако внешнему пользователю оно по-прежнему недоступно.

```
struct base {  
    void set_a (const int value) { a = value; }  
protected:  
    int a;  
};  
  
struct derived : base {  
    int get_a () { return a; }  
};  
  
int main () {  
    derived d;  
    d.set_a(123);  
    std::cout << d.get_a() << std::endl;  
}
```

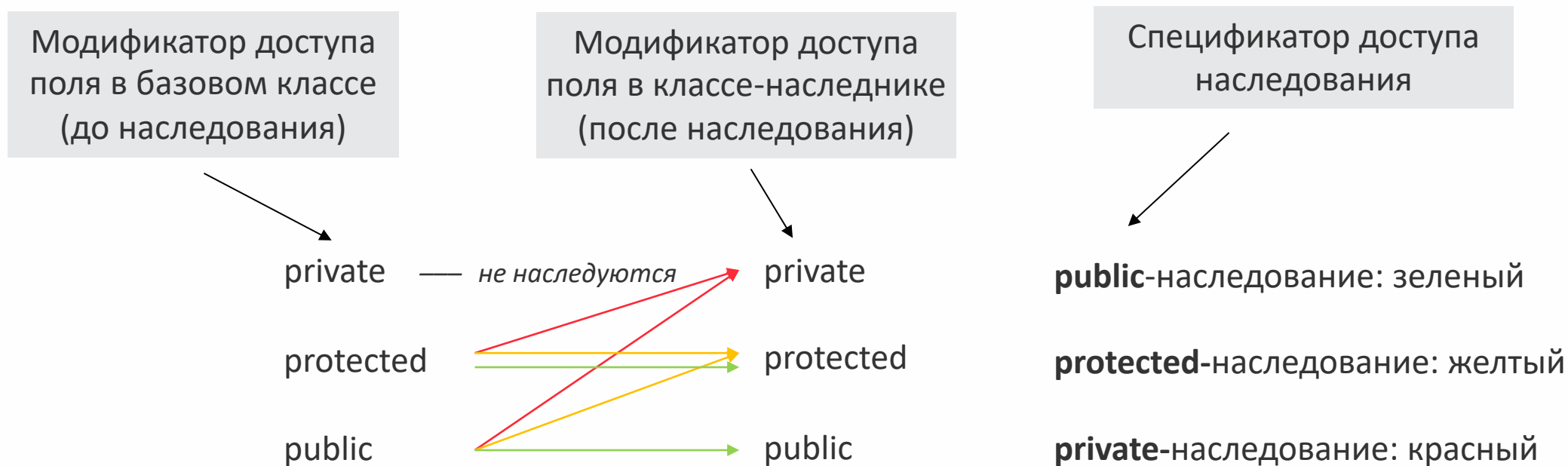
Рассмотрим сводную таблицу всех модификаторов доступа:

Модификатор доступа, которым помечено поле	Доступно ли поле пользователю извне класса	Наследуется ли поле
public	+	+
protected	—	+
private	—	—

Наконец, вернемся к спецификатором доступа наследования...

Спецификаторы доступа наследования

Спецификаторы доступа наследования позволяют вам изменить модификаторы доступа полей наследника:



Приведенная схема объясняет, почему добавление спецификатора **private** к наследованию в примере с прямоугольником и квадратом привело к тому, что методы `get_a` и `get_b` стали приватными.

Напоминание: если не указывать спецификатора наследования у *структуры*, то оно будет **public**. Аналогично, у класса – **private**.

Спецификаторы доступа наследования

Посмотрим на все возможные комбинации спецификаторов доступа полей классов и спецификаторов доступа наследования:

```
class base {  
public:  
    int a;  
  
protected:  
    int b;  
  
private:  
    int c;  
};  
  
class derived_public : public base {};  
class derived_protected : protected base {};  
class derived_private : private base {};
```

```
int main () {  
    derived_public public_instance;  
    public_instance.a; /// public  
    public_instance.b; /// protected  
    public_instance.c; /// isn't inherited from base  
  
    derived_protected protected_instance;  
    protected_instance.a; /// protected  
    protected_instance.b; /// protected  
    protected_instance.c; /// isn't inherited from base  
  
    derived_private private_instance;  
    private_instance.a; /// private  
    private_instance.b; /// private  
    private_instance.c; /// isn't inherited from base  
}
```


Финальный класс

Помимо спецификаторов доступа наследования, существует еще несколько полезных спецификаторов. Посмотрим на первый из них, спецификатор **final**.

Этот спецификатор позволяет запретить наследоваться от класса, помеченного им:

```
struct granny {};  
struct mommy final : granny {};  
struct son : mommy {};
```

Base 'mommy' is marked 'final'
'mommy' declared here

Make struct 'mommy' non-final

Этот спецификатор полезен, если вы пишете библиотечный код, и не хотите, чтобы пользователь имел возможность унаследовать функционал вашего класса, и, например, получить доступ к `protected` полям и методам вашего класса.

Множественное наследование

С++ позволяет классу наследовать **напрямую** сразу несколько других классов.
При множественном наследовании, потомку доступны поля и методы **всех** его базовых классов.

```
struct first_base {  
    void foo () { std::cout << "hi from first_base" << std::endl; }  
};  
  
struct second_base {  
    void goo () { std::cout << "hi from second_base" << std::endl; }  
};  
  
struct derived : first_base, second_base {  
    void boo () { std::cout << "hi from derived" << std::endl; }  
};  
  
int main () {  
    derived d;  
    d.foo();  
    d.goo();  
    d.boo();  
}
```



```
hi from first_base  
hi from second_base  
hi from derived
```

Множественное наследование

В случае множественного наследования, конструкторы сразу нескольких родительских все так же можно вызвать в списке инициализации:

```
struct first_base { int a; };
struct second_base { int b; };

struct derived : first_base, second_base {
    int c;
    derived (const int a, const int b, const int c): first_base{a}, second_base{b}, c(c) {}
};

int main () {
    derived d(1, 2, 3);
}
```

***Примечание:** в этом случае, безусловно, я вызвал не конструктор, а использовал синтаксис инициализации из списка, но сути дела это не меняет*



Множественное наследование

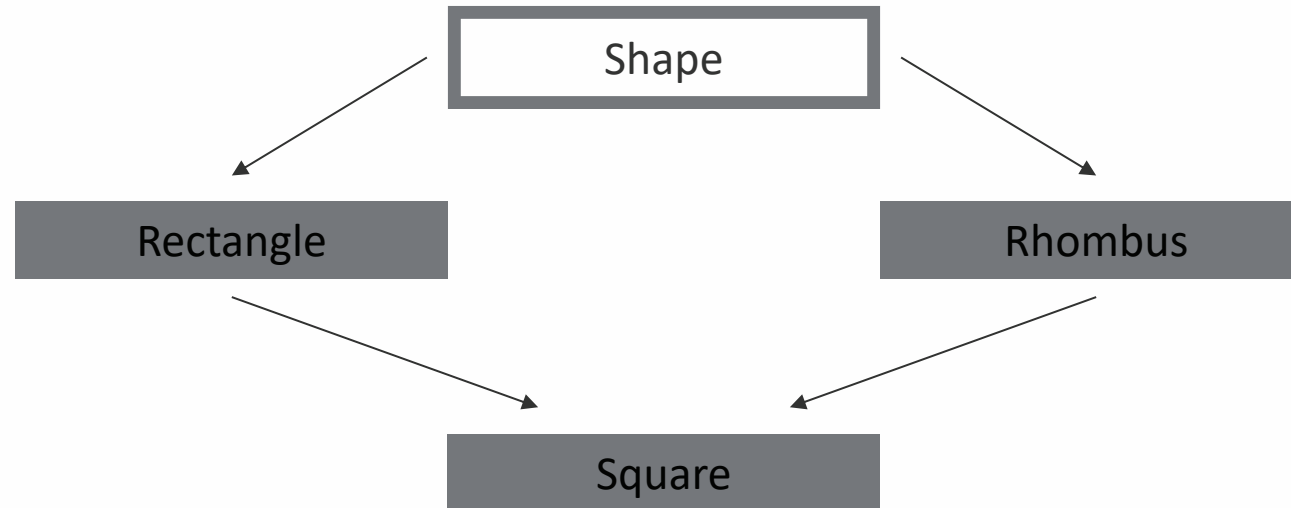
Также, синтаксис множественного наследования позволяет использовать разные модификаторы для наследования разных классов:

```
class derived :  
    public first_base,  
    protected second_base,  
    private third_base  
{ /* ... */ };
```

Ромбовидное наследование

Применение множественного наследования на практике достаточно проблемно, так как в случае так называемого “ромбовидного наследования” возникают неоднозначности.

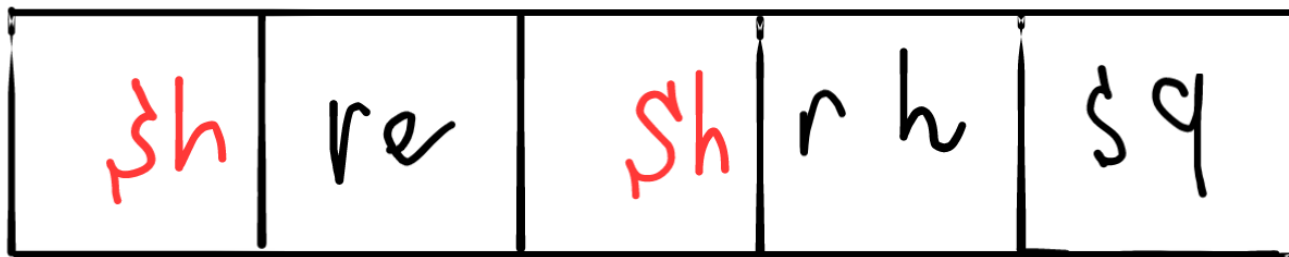
Изучим вопрос, рассмотрев приведенную конфигурацию классов:



Посмотрим на то, к каким проблемам такая модель наследования приводит на практике. Изучит [example-1.cpp](#)

Ромбовидное наследование

Проблема возникает из-за того, что в коде класса **square**, базовый класс **shape** встречается дважды, ведь он унаследован и через класс **rectangle**, и через класс **rhombus**:




Проблему можно было бы решить, если бы у нас был какой-то инструмент, который позволяет исключать дубликаты классов из иерархии при наследовании...

И такой инструмент у нас есть!

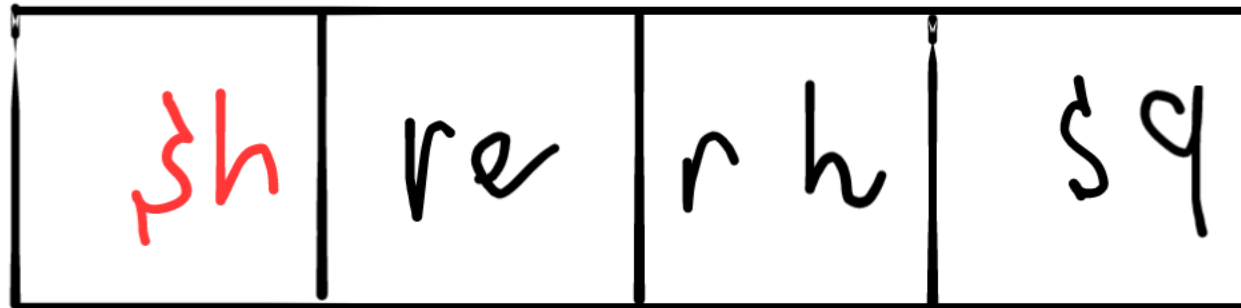
Виртуальное наследование

Виртуальное наследование (не путать с виртуальными функциями) позволяет запретить дублирование базового класса при наследовании и исключить лишние копии.



```
class rhombus : public virtual shape_2d {...};
```

Если виртуально унаследовать **Shape**, то на физическом уровне это будет выглядеть примерно так:



Изучим код в примере [example-2.cpp](#)

Виртуальное наследование

Важно: если мы наследуем n “сиблингов”, **лишь k** из которых наследуют родительский класс виртуально, то только они объединяются в одну группу, не приводящую к созданию дубликатов родительского класса лишь в k случаях (для остальных $n - k$ сиблингов, родительский класс все будет скопирован).

Рассмотрим пример:

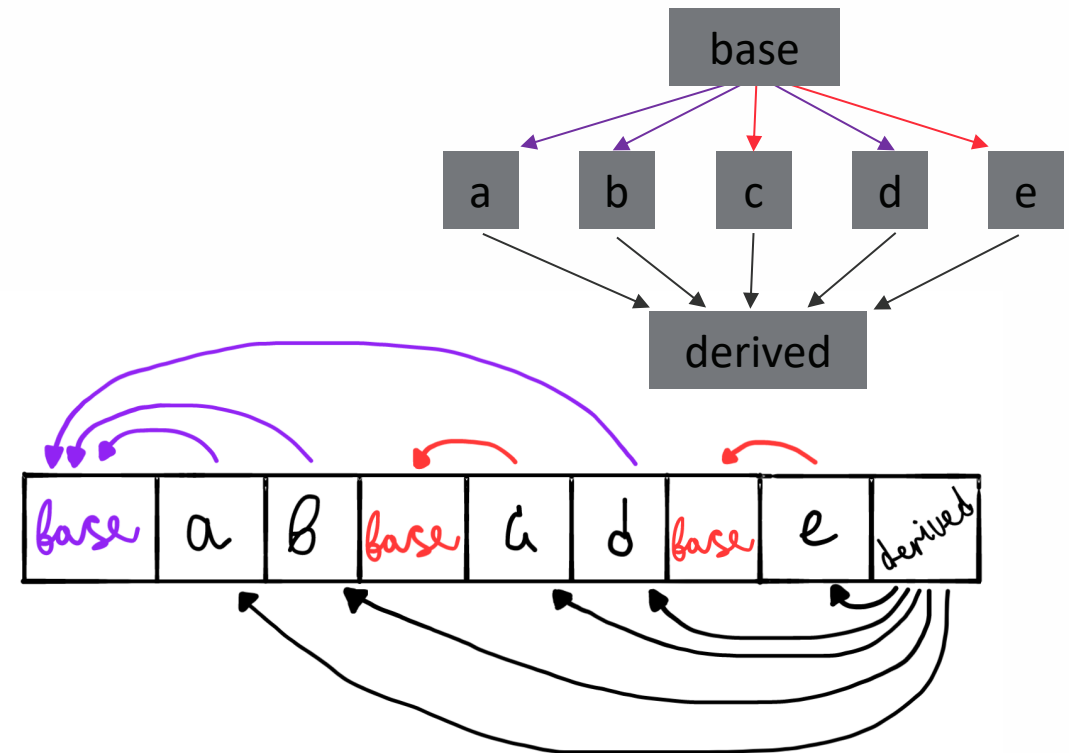
```
struct base { void f() { std::cout << "base" << std::endl; } };

struct a : virtual base { void f() { std::cout << "a" << std::endl; } };
struct b : virtual base { void f() { std::cout << "b" << std::endl; } };
struct c : base { void f() { std::cout << "c" << std::endl; } };
struct d : virtual base { void f() { std::cout << "d" << std::endl; } };
struct e : base { void f() { std::cout << "d" << std::endl; } };

struct derived : a, b, c, d, e { void f() { std::cout << "derived" << std::endl; } };

int main () {
    derived d;
    d.base::f();
}
```

Ambiguous conversion from derived class 'derived' to base class 'base':
struct derived -> struct a -> struct base
struct derived -> struct c -> struct base
struct derived -> struct e -> struct base



Для всех сиблингов, наследовавших **base виртуально** был выделен **единый** кусок памяти для хранения базы. Для **остальных** сиблингов, было создано по одной копии.

Обрезание наследников при передаче

Основная идея полиморфизма заключается в том, что одна сущность (например, функция), может работать с произвольным типом. Наследование реализует полиморфизм в C++.

Например, мы можем передать класс-наследник в функцию, которая ожидает базовый класс (**ведь класс-наследник является и базовым классом**).

В этом случае произойдет так называемое “обрезание” (**trimming**) инстанса. То есть внутри функции, которая приняла класс-наследник по типу его базы, вам будут доступны исключительно те методы и поля, которые доступны базе:

```
struct base {  
    void foo () const { std::cout << "hello from base" << std::endl; }  
};  
  
struct derived : base {  
    void foo () const { std::cout << "hello from first_derived" << std::endl; }  
};  
  
void greet (const base& b) {  
    b.foo();  
}  
  
int main () {  
    derived d;  
    greet(d);  
}
```



```
hello from base
```

```
Process finished with exit code 0
```

Обрезание наследников при передаче

Обрезание происходит не только при передаче в функцию, но и при обычном присваивании со взятием ссылки или указателя:

```
int main () {  
    derived d;  
  
    base& b = d;  
    b.foo();  
  
    /// reinterpret_cast is not needed in this case!  
    /// (because of the inheritance)  
    static_cast<base*>(&d)->foo();  
}
```



```
hello from base  
hello from base  
  
Process finished with exit code 0
```

Важно: обрезание не происходит на физическом уровне. То есть с самим экземпляром ничего не происходит, поля и методы наследника **не удаляются**, будучи обрезанными, а просто скрываются от пользователей.



Виртуальные методы

На практике, такое поведение может быть не всегда полезно...

Рассмотрим пример [example-3.cpp](#), в котором мы хотим написать функцию, которая выводит площадь произвольной фигуры. Из-за обрезания, нам не доступны методы **area** у наследников, а доступен лишь метод **area** базового класса.

Вот бы у нас был инструмент, при помощи которого мы бы могли сохранить информацию о том, что наш обрезанный до базового класс-наследник **все же является наследником**, дабы была возможность отличить его от базового класса...

И такой инструмент у нас есть! Этот инструмент – **виртуальный** метод.

Примечание: слово **виртуальный** плохо описывает поведение такого метода. На самом деле, куда более корректным словом могло бы быть слово “**персистентный**”, либо “**динамический**”. Действительно, виртуальный метод будто бы **привязывается к самому** экземпляру (*динамический контекст*), а не к типу объекта (*статический контекст*).

Рассмотрим следующий пример...

Виртуальные методы

Добавив спецификатор **virtual**, мы можем изменить поведение “обрезания”, продемонстрированное несколькими слайдами ранее:

```
struct base {  
    void foo () const { std::cout << "hello from base" << std::endl; }  
};  
  
struct derived : base {  
    void foo () const { std::cout << "hello from first_derived" << std::endl; }  
};  
  
void greet (const base& b) {  
    b.foo();  
}  
  
int main () {  
    derived d;  
    greet(d);  
}
```

hello from base

Process finished with exit code 0



```
struct base {  
    virtual void foo () const { std::cout << "hello from base" << std::endl; }  
};  
  
struct derived : base {  
    void foo () const { std::cout << "hello from derived" << std::endl; }  
};  
  
void greet (const base& b) {  
    b.foo();  
}  
  
int main () {  
    derived d;  
    greet(d);  
}
```

hello from derived

Process finished with exit code 0

Теперь у инстансов **любого** наследника base, определяющего метод **foo**, будет вызываться метод, определенный наследником, а не базовым классом (то есть **derived::foo**, а не **base::foo**).

Виртуальные методы

Заметим, что спецификатор **virtual** мы применили только к методу базового класса. Действительно, **никакой** необходимости помечать метод наследника спецификатором **virtual** нет, так как он **уже** трактуется как виртуальный во всей **последующей** иерархии наследования, поэтому настоятельно рекомендуется помечать спецификатором **virtual** только метод базового класса.

```
struct granny {  
    virtual void foo () const { std::cout << "hello from granny" << std::endl; }  
};  
  
struct mommy : granny {  
    /// explicitly virtual, base method is granny::foo  
    void foo () const { std::cout << "hello from mommy" << std::endl; }  
};  
  
struct son : mommy {  
    /// explicitly virtual, base method is granny::foo  
    void foo () const { std::cout << "hello from son" << std::endl; }  
};  
  
void greet (const granny& b) {  
    b.foo();  
}  
  
int main () {  
    mommy m;  
    greet(m);  
    son s;  
    greet(s);  
}
```



```
hello from mommy  
hello from son  
  
Process finished with exit code 0
```



Виртуальные методы

Поведение, при котором одноименные методы всех наследников неявно становятся виртуальными – не очень наглядно, из-за чего ваш код может стать трудночитаемым и повысится вероятность ошибки.

Действительно, представим, что вы определили базовый класс, содержащий виртуальный метод, в отдельном от наследников файле. В этом случае, код наследников не содержит никакой информации о том, что в нем присутствует виртуальный метод, перегружающий метод базового класса. Можно попробовать решить эту проблему, добавив спецификатор **virtual** на метод наследника, однако это решение не лучшее, ведь в этом случае мы все еще не имеем достаточно информации для того, чтобы визуально определить, что этот метод перегружает метод базового класса, а не просто является виртуальным методом.

Для таких случаев был придуман специальный спецификатор **override**, который применим исключительно к виртуальным методам, которые перегружают какой-то виртуальный метод базового класса.

Улучшим код предыдущего примера, добавив спецификатор **override**...

Виртуальные методы

Теперь нам сразу же понятно, что метод является виртуальным, и перегружает какой-то базовый виртуальный метод.

```
struct granny {
    virtual void foo () const { std::cout << "hello from granny" << std::endl; }
};

struct mommy : granny {
    void foo () const override { std::cout << "hello from mommy" << std::endl; }
};

struct son : mommy {
    void foo () const override { std::cout << "hello from son" << std::endl; }
};

void greet (const granny& b) {
    b.foo();
}

int main () {
    mommy m;
    greet(m);
    son s;
    greet(s);
}
```

Более того, добавление спецификатора **override** позволяет помочь вам избежать ошибки, связанной с опечаткой в названии метода, либо с забытым спецификатором **virtual** в методе базового класса:

```
struct base {
    void foo () {}
};

struct derived : base {
    void goo () override {} // whoops, made a typo...
    void foo () override {} // whoops, forgot to mark base::foo as virtual
};
```

Only virtual member functions can be marked 'override' ⋮

Remove 'override' ↵⌘⌘ More actions... ↵⌘

Виртуальные методы

Существует также еще один специальный спецификатор, который позволяет **запретить дальнейшее переопределение виртуального метода в иерархии наследования**. Этот спецификатор, – **final**:

```
struct a {  
    virtual void foo () {}  
};  
  
struct b : a {  
    virtual void foo () override {}  
};  
  
struct c : b {  
    virtual void foo () final {}  
};  
  
struct d : c {  
    virtual void foo () override {}  
};
```

Declaration of 'foo' overrides a 'final' function
overridden virtual function is here

Make c::foo non-final More actions...

Этот спецификатор достаточно полезен, если вы пишете какой-то библиотечный код, и не хотите, чтобы ваш виртуальный метод можно было переопределить наследованием от вашего класса.

Таким образом, последовательность модификаторов виртуальных методов в иерархии наследования получается такой:

virtual -> override -> ... -> override -> [final]

Естественно, **final** – опционален.



Таблица виртуальных методов

Разберемся, как работает ключевое слово **virtual**, и почему поведение виртуальных методов таковое.

Типы, имеющие таблицу виртуальных методов будем называть **виртуальными**.

Внутри виртуального класса (класса, имеющего хотя бы один виртуальный метод), инициализируется указатель на **таблицу виртуальных методов**.

Эта таблица, в свою очередь, хранит соответствие *“название виртуального метода” <-> “указатель на верную версию метода для вызова”*.

Помимо статического типа, у виртуальных типов есть еще и **динамический**, который привязывается к самому экземпляру объекта в момент инстанцирования и не изменяется даже при обрезании этого объекта.

Таким образом, виртуальная таблица содержит указатель на нужный метод, который и будет вызван при необходимости, причем она сама хранится внутри базового класса, то есть при обрезании, **она никуда не пропадает**. Это и позволяет таблице виртуальных методов вызвать нужный метод, исходя из динамического типа объекта.

Таблица виртуальных методов

class base

vtable

foo	0x582dab3c
goo	0x81052a3c

virtual method foo

virtual method goo

non-virtual method boo

derived : base

vtable:

foo	0x5952a3bc
goo	0x1bc713ca

virtual method foo

virtual method goo

non-virtual method boo

method foo override

method goo override

base

derived

Таким образом, даже если тип derived будет обрезан до base, виртуальная таблица сохранит корректные указатели на методы, которые необходимо вызвать (эти методы, несмотря на обрезание, физически будут существовать, и их спокойно выйдет вызвать по указателю).



Абстракции

Исправим, наконец, предыдущий пример, и рассмотрим [example-4.cpp](#).

Добавив спецификаторы **virtual** и **override**, мы изменили поведение функции **print_area**, и теперь она работает **полиморфно**, то есть результат ее работы отличается в зависимости от **динамического** типа переданного ей объекта (статический тип получаемого объекта зафиксирован – **shape**).

Однако, в нашем коде все еще остаются несколько недочетов. Разберемся с первым. Проблема заключается в том, что то, как мы реализовали метод **shape::area**, в общем случае – костыль. Действительно, возвращать из метода какое-то значение-заглушку не есть хороший паттерн проектирования. Вместо этого, мы бы хотели иметь возможность **вообще не реализовывать** тело этого метода, ведь **shape** – это чистая абстракция, не имеющая площади, в отличие от **вполне себе конкретных** прямоугольника и ромба.

Для таких случаев, в языке предусмотрен специальный вид виртуального метода: **pure virtual method** (чистый виртуальный метод)...

Абстракции

Объявим метод `shape::area` как чистый виртуальный:

```
struct shape {  
    virtual double area () const = 0;  
};
```

***Примечание:** На первый взгляд, может показаться, что название и синтаксис этого вида виртуальной функции не очень удачные... Действительно, куда более подходящим названием было бы “**абстрактный метод**”, да и в синтаксис можно было бы ввести слово **abstract** для объявления такого типа метода...*

Определение: класс, содержащий (или наследующий без перегрузки) как минимум один чистый виртуальный метод называется **абстрактным**.

В примере сверху, класс `shape` – абстрактный.

Важное свойство абстрактных классов заключается в том, что нельзя создавать их экземпляры, ведь попросту неясно как у них можно было бы вызвать неопределенный чистый виртуальный метод.



Абстракции

На первый взгляд может показаться, что абстрактные классы – усложнение, которое приводит к тому, что в вашем коде появляются классы, которые нельзя инстанцировать. Кажется, что концепт абстракции ограничивает вас, и не позволяет удобно работать с объектами в программе. В действительности же, абстракции существуют в языке для того, чтобы у нас была возможность выделить **интерфейс** какой-то сущности.

Интерфейс – это описание (перечисление) публичных методов и их сигнатур, которые должен иметь какой-то класс. Например, любой **shape** должен уметь возвращать число типа **double** из своего метода **area**. Помимо этого, мы можем добавить еще один метод – **perimeter**, который также будет возвращать число типа **double**.

Наличие возможности выделить интерфейс какой-то сущности является жизненно важной функцией объектно-ориентированного языка, позволяющей реализовать полиморфизм, ведь с его помощью, мы можем создавать функции, классы, и другие сущности, которые работают с абстрактными объектами, используя **определенный интерфейс**. Например, в рассмотренном примере [example-4.cpp](#), функция **print_area** так себя и ведет.

Абстракции

Попробуем создать вектор, который будет содержать несколько экземпляров, реализующих абстракцию **shape**:

```
int main () {  
    std::vector<shape> vec({  
        rectangle(1., 2.),  
        rhombus(1., std::nu  
    });  
}
```

Allocating an object of **abstract** class type 'const shape' :
unimplemented pure virtual method 'area' in 'shape'

К сожалению, у нас ничего не выйдет, так как **shape** – абстракция, а значит создать вектор из нее не выйдет.

***Примечание:** вектор не понимает, что работает с наследниками, реализующими абстракцию, а думает, что создает вектор экземпляров самой абстракции, что запрещено.*

Попробуем решить эту проблему динамическим выделением памяти: инстанцируем экземпляры на куче, а в сам вектор сохраним указатели на них, не забыв удалить выделенную память после использования...

Абстракции

```
int main () {  
    std::vector<shape* const> vec({  
        new rectangle(1., 2.),  
        new rhombus(1., std::numbers::pi / 4.)  
    });  
  
    for (const shape* const s : vec)  
        print_area(*s);  
  
    for (const shape* const s : vec)  
        delete s;  
}
```

Delete called on 'shape' that is abstract but has non-virtual destructor

Казалось бы, все теперь хорошо, однако статический анализатор выводит какую-то ошибку, связанную с деструкторами. Добавим вывод сообщений в консоль из деструкторов классов **rectangle** и **rhombus**, и протестируем, вызываются ли они...

Абстракции

```
class rectangle : public shape {
public:
    rectangle(const double a, const double b): a(a), b(b) {}
    double area() const override { return a * b; }
    ~rectangle () { std::cout << "~rectangle" << std::endl; }

private:
    double a;
    double b;
};

class rhombus : public shape {
public:
    rhombus(const double side, const double angle): side(side), angle(angle) {}
    double area() const override { return std::pow(side, 2.) * std::sin(angle); }
    ~rhombus () { std::cout << "~rhombus" << std::endl; }

private:
    double side;
    double angle;
};
```

2

0.707107

Process finished with exit code 0

Видно, что вызова деструкторов не произошло, а значит, в нашей программе есть утечка памяти...
В чем же причина?

Дело в том, что деструктор — это тоже метод, а значит и он должен быть виртуальным, иначе по аналогии с обычным методом, будет вызван только деструктор базового класса.

Абстракции

Объявим деструктор базового класса как **pure virtual**:

```
struct shape {  
    virtual double area () const = 0;  
    virtual ~shape () = 0;  
};
```

Попробуем запустить код и получим ошибку:

```
Undefined symbols for architecture x86_64:  
  "shape::~~shape()", referenced from:  
      rectangle::~rectangle() in main.cpp.o  
      rhombus::~rhombus() in main.cpp.o  
ld: symbol(s) not found for architecture x86_64  
clang: error: linker command failed with exit code 1 (use -v to see invocation)  
ninja: build stopped: subcommand failed.
```

Дело в том, что деструктор – это единственный метод, который не должен быть объявлен как **чистый** виртуальный в абстрактном классе. Деструктор базового класса обязательно должен быть реализован, иначе деструкторам наследников будет некого вызывать (вспомним, что наследник должен вызвать деструктор базового класса при собственной деструктуризации).

Абстракции

Объявим деструктор как **default** , и исправим наш код, наконец:

```
struct shape {  
    virtual double area () const = 0;  
    virtual ~shape () = default;  
};
```

```
2  
0.707107  
~rectangle  
~rhombus  
  
Process finished with exit code 0
```

Полный код находится в файле [example-5.cpp](#).

***Примечание:** конечно же, работать с экземплярами абстрактных классов через указатели – неудобно... Для упрощения нашей жизни, в язык были введены так называемые **умные указатели**, которые мы рассмотрим через несколько лекций.*



Динамическое преобразование типов

Напомню, помимо статического типа, у виртуальных типов есть еще и **динамический**, который привязывается к самому экземпляру объекта в момент инстанцирования и не изменяется даже при обрезании этого объекта. Это и позволяет таблице виртуальных методов вызвать нужный метод, исходя из динамического типа объекта.

Может потребоваться преобразовать виртуальный объект к его изначальному типу (например, после передачи по ссылке на родительский класс в какую-то функцию). Для этого (**и только для этого**) следует использовать оператор **dynamic_cast**.

Оператор **dynamic_cast** работает исключительно с указателями и ссылками. Преобразовать объект по-значению им не выйдет.

Оператор **dynamic_cast** выполняет преобразование в runtime, то есть это чисто динамический оператор. Из-за этого, у нас нету способа понять, сработает ли преобразование или нет на этапе компиляции, поэтому оператор **dynamic_cast** – **побуе** преобразовать объект к требуемому типу, а не гарантированно делает это.

Динамическое преобразование типов

Добавим методы `rectangle::get_diagonal` и `rhombus::get_diagonals` в предыдущий пример. Пусть теперь мы хотим написать **универсальный** код, который получает абстрактную фигуру `shape`, и выводит в консоль длину ее большей диагонали. Проблема в том, что методы `rectangle::get_diagonal` и `rhombus::get_diagonals` имеют различную сигнатуру, и унифицировать их посредством выделения общей интерфейсной функции не выйдет.

Значит, нам нужен какой-то способ отличить прямоугольник от ромба прямо внутри функции `print_greatest_diagonal`. Воспользуемся оператором `dynamic_cast`.

Если `dynamic_cast` работает с указателем, то он возвращает указатель на требуемый тип в случае успешного преобразования, иначе – `nullptr`. Посмотрим на пример [example-6.cpp](#).

Если же `dynamic_cast` работает со ссылкой, то в случае неуспешного преобразования выбрасывается **исключение**.

Посмотрим на пример [example-7.cpp](#), попробуем его запустить и убедимся в том, что программа падает с ошибкой `std::bad_cast` из-за неуспешной попытки преобразования ромба к прямоугольнику. Про исключения мы будем говорить на следующей лекции, ну а пока что, будем использовать указатели для динамического преобразования типов



Идиома CRTP

Напоследок, разберем идиому CRTP: **Curiously Recurring Template Pattern**. Обойдемся без определений и сразу же взглянем на пример кода, демонстрирующий **CRTP** в общем виде:

```
template <typename vt>
struct base {
    /// ...
};

struct derived : base<derived> {
    /// ...
};
```

Кажется, что в этом случае должна возникнуть ошибка компиляции, ведь мы используем класс при его же определении, то есть, присутствует какая-то рекурсия...

На самом деле, никакой рекурсии здесь нет, ведь подстановка параметров шаблонных классов происходит в compile-time.



Идиома CRTP

CRTP обычно используется в такой конфигурации: существует шаблонный класс, определяющий некий интерфейс, открытый к расширению и зависящий от типа наследника.

Например, мы можем написать шаблонный абстрактный класс **comparable**, от которого унаследуем классы **vector_2d** и **number**, в которых в свою очередь определим метод **compare_to**, благодаря чему получим возможность сравнивать их экземпляры. Наконец, напомним, функцию **compare**, которая принимает два объекта произвольного типа, реализующих интерфейс **comparable** и сравнивает их, возвращая результат. Используем эту функцию для сравнения как экземпляров **vector_2d**, так и экземпляров **number**.

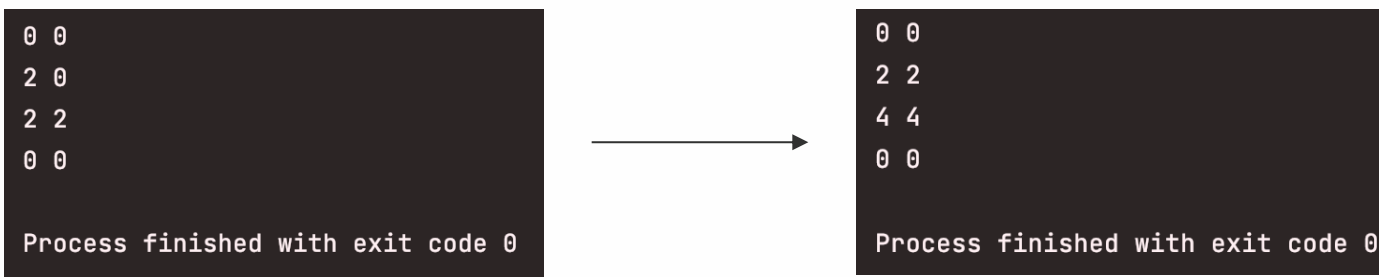
Посмотрим код в файле [example-8.cpp](#)

Идиома CRTP

Также CRTP можно комбинировать и со **static**-полями. Например, можно сделать удобный класс-помощник, унаследовав который, можно будет считать количество копий наследника.

Посмотрим пример [example-9.cpp](#).

На первый взгляд может показаться, что раз **vt** (шаблонный параметр) не используется в коде, то и идиома CRTP здесь нам не нужна, и счетчик **counter** можно сделать нешаблонным классом. Однако, если попробовать так сделать (*попробуйте*), то счетчик станет общим для всех типов, чего нам **не** хотелось бы:



То есть в этом случае, шаблонный параметр позволяет создать отдельный класс-счетчик для каждого типа, унаследовавшего его. Это работает так, потому что во время компиляции, шаблонные параметры подставляются в код шаблонного класса и создают отдельные классы. То есть коды классов **counter<vector_2d>** и **counter<vector_3d>** – разные, а значит и счетчики в них.



HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

Лекция 4

Наследование

Программирование на языке C++

Константин Леладзе

ФКН ВШЭ

