

X

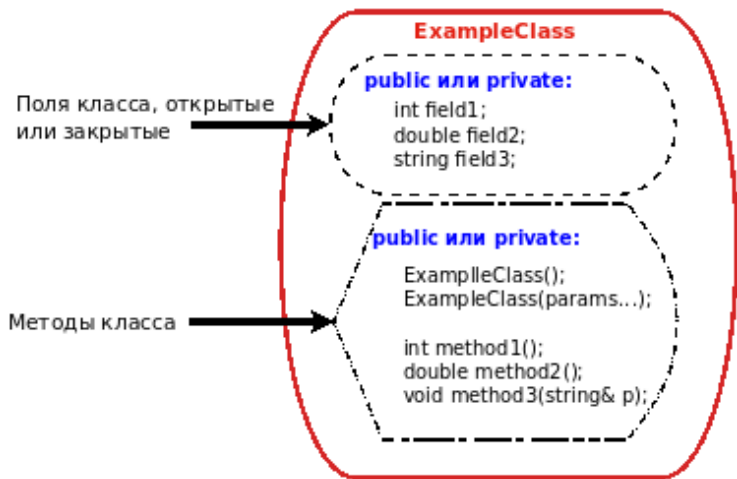
# Наследование в C++


Как проектируются типы (речь в основном об самых общих составных типах в C++: *классы* и *структуры*)?


- есть группа значений, которые логически связаны между собой – они становятся *полями* класса и, технически, образуют хранилище (storage) каждого его объекта (переменной);
- придумывается «словарь» типа для изменения всех или части значений в конкретном объекте – так появляются *методы* класса.

# Наследование в C++

Схематичное представление составного типа:



Подобная схема очень произвольно нарисована. Для проектирования сложных компьютерных систем был разработан специальный *язык моделирования*, представляющий собой набор диаграмм для отображения типов программы, их связей и взаимодействий между друг другом. Этот язык получил название – <sup>1</sup> (*унифицированный язык моделирования*, англ. unified modelling language).

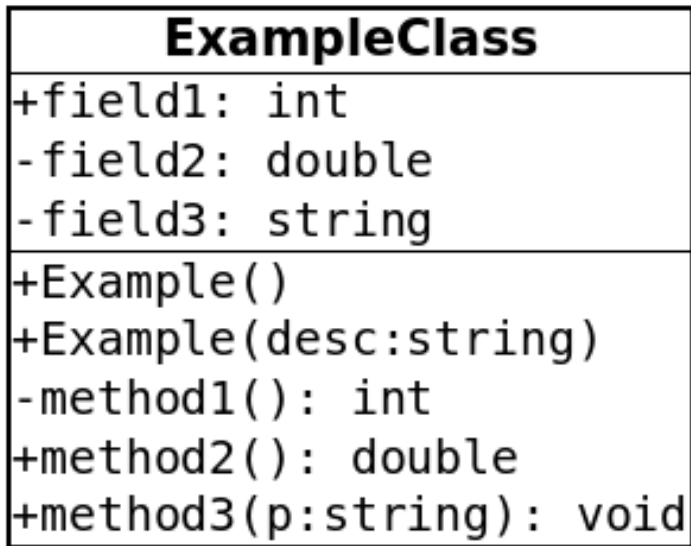
Его освоение не входит в задачи данного курса, но отсюда мы возьмём диаграмму для представления <sup>2</sup>

---

<sup>1</sup><https://ru.wikipedia.org/wiki/UML>

<sup>2</sup>[https://ru.wikipedia.org/wiki/Диаграмма\\_классов](https://ru.wikipedia.org/wiki/Диаграмма_классов)

Пример диаграммы класса на языке UML



На предыдущем слайде:

- 1 в первом блоке указано имя типа: **ExampleClass** в примере;
- 2 во втором блоке идёт перечисление *полей* составного типа.
  - знак «+» означает открытое поле;
  - знак «-» — закрытое поле;

Типы полей указываются справа от их названий.

- 3 в третьем блоке перечислены методы класса (в том числе и конструкторы). Смысл знаков «плюса» и «минуса» — полностью аналогичен второму пункту. Параметры указываются в виде «имя»:«тип», тип возвращаемого значения ставится после списка параметров.

Теперь добираемся до **наследования**

## Термин

**Наследование** – специальный механизм в языках программирования, позволяющий создавать новые типы данных на основе уже имеющихся.

Основные понятия:

- класс, на основе которого *создаётся другой тип*, называется **базовым** классом (др. встречающиеся термины – *родительский класс, суперкласс*);
- *создаваемый класс* на основе другого типа называется **производным** (класс-потомок, подкласс).



Вместе с данной лекцией идут

- дополнительное описание наследование в C++ в виде нескольких некрупных заметок – [▶ начать отсюда](#)<sup>3</sup>. Очень желательно для ознакомления всем, кто хочет въехать в этот механизм;
- пример с лекции, [▶ доступный тут](#). Далее часть деталей реализации конкретных методов приводится не будет для сосредоточения на сути, а не на деталях.

---

<sup>3</sup><https://github.com/posgen/OmsuMaterials/wiki/OOP:-inheritance-part-1>

UML-диаграмма базового класса, с которым будем изучать механизм наследования в C++:

<b>Base</b>
-details_ : string
+Base(details:string) +details(): string +set_details(details:string): void +about(): string

# Наследование в C++

И его объявление:

```
1 class Base
2 {
3 public:
4     Base(const string& details);
5
6     string details() const;
7     void set_details(const string& new_details);
8     string about() const
9     { return "[Base] with details: " + details_; }
10
11 private:
12     string details_;
13 };
```

Достаточно простой класс – одно поле, три метода.

Приведена реализация только метода **about**, который по задумке просто сообщает нам о *состоянии* конкретного объекта данного типа.

# Наследование в C++

Создадим *производный* класс:

```
1 class Derived : public Base
2 {
3 public:
4     Derived(string details, string description) :
5         Base{details}, description_{description}
6     {}
7
8     string about() const
9     { return Base::about() +
10         "\n[Derived] with description: "
11         + description_; }
12
13 private:
14     string description_;
15 };
```

## Синтаксические особенности:

- базовый класс указывается сразу после объявления производного типа через двоеточие (**строка 1**);
- **public Base** – означает *открытое* наследование (уточним термин далее);
- при создании (конструировании) объекта класса **Derived** всегда есть возможность явно вызвать конструктор базового класса (**строка 5**).

## Смысловые (семантические) особенности:

- класс **Derived** сможет пользоваться *интерфейсом* базового класса за исключением метода **about**;
- для метода **about** класс **Derived** хочет предоставить собственную реализацию, отличающуюся от таковой в базовом типе.

При наследовании всегда ставится вопрос – что тип **Derived** получил от своего базового класса **Base**?

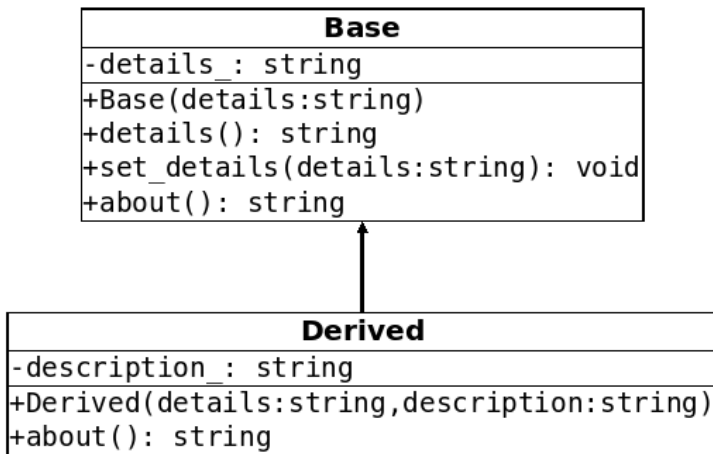
Со стороны технической части, при наследовании в производном классе **всегда** появляются поля базового объекта. В нашем примере это означает, что при создании объекта типа **Derived**, хранилище под его данные будет состоять из двух частей: первая часть – для поля **details\_** из класса **Base**, вторая – для **description\_** из класса **Derived**. Причём, C++ гарантирует такой порядок следования полей в хранилище, что сначала идут поля базового (базовых, в общем случае) типа, а затем уже поля собственно производного класса.

Со стороны логической части, при наследовании производный класс получает частичный доступ к полям/методам базового класса (т.е. возможность обращаться к ним напрямую по идентификатору). Базовые правила тут следующие:

- к закрытым (**private**) полям и методам базового класса *производный* тип доступа **не имеет**;
- конструкторы и деструктор **никогда не передаются** производному классу при наследовании;
- производный класс может иметь доступ ко всем открытым полям и методам базового типа.

# Наследование в C++

Между делом, диаграмма **UML** для наследования в нашем случае имеет вид:





## Суть

В рассматриваемом примере наследование реализует так называемое отношение «is-a» (или «является»). Так, объекты класса **Derived** одновременно *являются* объектами базового класса **Base** и могут быть использованы во всех случаях, где требуются копии, ссылки или указатели на тип **Base**. Не без нюансов, конечно же, как уж без них.

Пример использования созданных классов:

```
1 Base b1{"red, green, blur"};  
2 cout << b1.about() << endl << endl;  
3  
4 Derived d1{"one, two, three", "something ↵  
    important"};  
5 cout << d1.about() << endl << endl;
```

Здесь идут обычные вызовы методов, вывод, надеюсь, не содержит каких-либо неожиданностей.

# Наследование в C++

Выше говорилось о том, что производный класс не имеет доступа к *закрытым* полям базового типа. На практике это означает невозможность следующего кода для **Derived**:

```
1 class Derived : public Base
2 {
3     public:
4     ...
5     // метод приведёт к ошибке компиляции
6     void update_details(string more)
7     { details_ += more; }
8     ...
9 };
```

Несмотря на то, что поле **details\_** присутствует в типе **Derived**, доступ к нему ни для получения, ни для изменения значения невозможен.

Но в некоторых случаях доступ к закрытым полям и/или метода базового класса может быть предоставлен с помощью ещё одного *модификатора доступа* – **protected**:. Для примера, добавим в **Base** ещё одно поле:

```
1 class Base
2 {
3     ...
4     protected:
5         string type_;
6 };
```

По отношению к объектам типа **Base** – это обычное закрытое поле, доступ к которому есть только внутри методов данного класса.

# Наследование в C++

А вот для производного класса это поле становится доступным по прямому имени:

```
1 class Derived : public Base
2 {
3     public:
4     ...
5     string type() const
6     { return type_; }
7
8     void update_type(string new_type)
9     { type_ = new_type; }
10    ...
11 };
```

Никаких ошибок, всё будет работать. В русско-язычной литературе для описания **protected** полей и методов часто используется термин – **защищённые** поля и методы.

И ещё одно небольшое отступление – на диаграмме *UML* **protected**-поле (или метод) обозначаются вот так:<sup>4</sup>

<b>Base</b>
-details_: string
#type_: string
+Base(details:string)
+details(): string
+set_details(details:string): void
+ <i>about(): string</i>

---

<sup>4</sup>Диаграмма создана с помощью [программки Dia](#). Почему метод **about** стал выделен курсивом – дальше по слайдам будет объяснение.

Возвращаемся к исследованию наследования. Более полезен при проектировании иерархии типов *случай*, когда объект производного класса может выступать в качестве базового. Напомним, в общем случае, в C++ есть три способа использования объектов – по значению (копия), по ссылкам и по указателю. Скажем, условно у нас могли бы быть три функции для исследования наследования:

```
1 void pass_by_value(Base obj);  
2 void pass_by_ref(Base& obj);  
3 void pass_by_ptr(Base* obj);
```

# Наследование в C++

Не будет определять отдельные функции, чтобы не увеличивать размер примеров, сосредоточимся на следующей ситуации:

```
1 Base b_copy = d1;    // (1)
2 Base& b_ref = d1;    // (2)
3 Base* b_ptr = &d1;   // (3)
4
5 cout << "obj-to-obj" << b_copy.about() << endl
6     << "-----" << endl;
7
8 cout << "ref-to-obj" << b_ref.about() << endl
9     << "-----" << endl;
10
11 cout << "ptr-to-obj" << b_ptr->about() << endl
12     << "-----" << endl;
```



При выполнении примера с предыдущего слайда, трижды будет вызван метод **about** базового класса.

При этом, это логично только для случая **(1)** (копирование объекта **d1** в **b\_copy**). Копирование объектов производного типа в базовый класс возможно благодаря технической особенности создания хранилища для объектов производного класса (см. выше).

Для случаев **(2)** и **(3)** хотелось бы другого поведения: вроде бы, копирования тут не происходит; и ссылке, и указателю подставляем объект **d1**. Хотелось бы, чтобы и метод **about** вызывался от производного, а не от базового класса.

Такое возможно, только если сделать тип **Base** – **полиморфным** классом.

## Термин

В самой общей форме, под **полиморфизмом** в языках программирования понимают предоставление *единого интерфейса* для сущностей (объектов<sup>a</sup>) различных типов (термин восходит к утхтсггве языку: *polys* – много, многие; *morphe* – форма, вид.

---

<sup>a</sup>в более общем смысле, чем экземпляров конкретных классов

**Полиморфизм** может быть классифицирован различными категориями. Например, в программировании различают **статический** и **динамический** полиморфизм. Пример *статического полиморфизма* в C++ – это перегрузка функций и шаблоны. *Динамический* – разбирается далее на примере наследования.

Есть другая классификация:

- 1 **«ad-hoc» полиморфизм** (иногда известен как *мнимый полиморфизм*) – возможность одной функцией обрабатывать входящие данные разных типов. Опять же, та самая **перегрузка** функций в C++;
- 2 **параметрический полиморфизм** – это шаблонные функции и типы в C++;
- 3 **полиморфизм подтипов** – а это как раз механизм (*открытого*) наследования.

# Наследование в C++

Для того, чтобы сделать *полиморфным* класс **Base**, в него надо добавить хотя бы один метод, который будет объявлен со спецификатором **virtual**

```
1 class Base
2 {
3 public:
4     Base(const string& details);
5
6     string details() const;
7     void set_details(const string& new_details);
8     virtual string about() const
9     { return "[Base] with details: " + details_; }
10
11 private:
12     string details_;
13 };
```

Производный тип **Derived** поменяется следующим образом:

```
1 class Derived : public Base
2 {
3     ...
4     string about() const override
5     { return Base::about() +
6       "\n[Derived] with description: "
7       + description_; }
8     ...
9 };
```

Ключевое слово **override** необязательно, но оно вынуждает компилятор проверить, что мы действительно переопределили *виртуальную* функцию базового класса.

# Наследование в C++

И теперь приведённый ранее пример сможет по ссылке и указателю *на базовый класс* вызвать метод **about** уже реального объекта, на который они ссылаются:

```
1 Base& b_ref = d1;
2 Base* b_ptr = &d1;
3
4 // будет вызван метод Derived::about
5 cout << "ref-to-obj" << b_ref.about() << endl
6     << "-----" << endl;
7 // будет вызван метод Derived::about
8 cout << "ptr-to-obj" << b_ptr->about() << endl
9     << "-----" << endl;
```

Виртуальный метод **about** используемый инструмент для построения диаграмм отображает курсивом:

Base
-details_: string
+Base(details:string)
+details(): string
+set_details(details:string): void
+ <i>about(): string</i>