Объектно-ориентированное программирование. Лекции 6-7. Наследование и полиморфизм

Макаров П. А.

14 и 28 ноября 2024 г.

Содержание

1 Иерархия типов объектов

В достаточно крупных программах часто возникает ситуация, в которой объекты некоторой предметной области могут быть объединены в определённые категории образующие иерархию типов объектов.

Определение 1. ***Иерархия типов объектов*** *— это вид категоризации всей совокупности объектов данной предметной области, при которой:*

1. *категории могут подразделяться на подкатегории;*
2. *каждая категория обладает некоторыми специфическими для неё свойствами, причём этими же свойствами обладают и все объекты из её подкатегорий.*

Из рассмотренного примера (см. рис. 1) видно, что при иерархической организации типов естественно возникают структуры данных, имеющие

- с одной стороны — различный набор полей и обрабатывающих функций,
- а с другой стороны — и некоторые общие поля и операции.

Решением данного противоречия в рамках объектно-ориентированной парадигмы является механизм наследования.

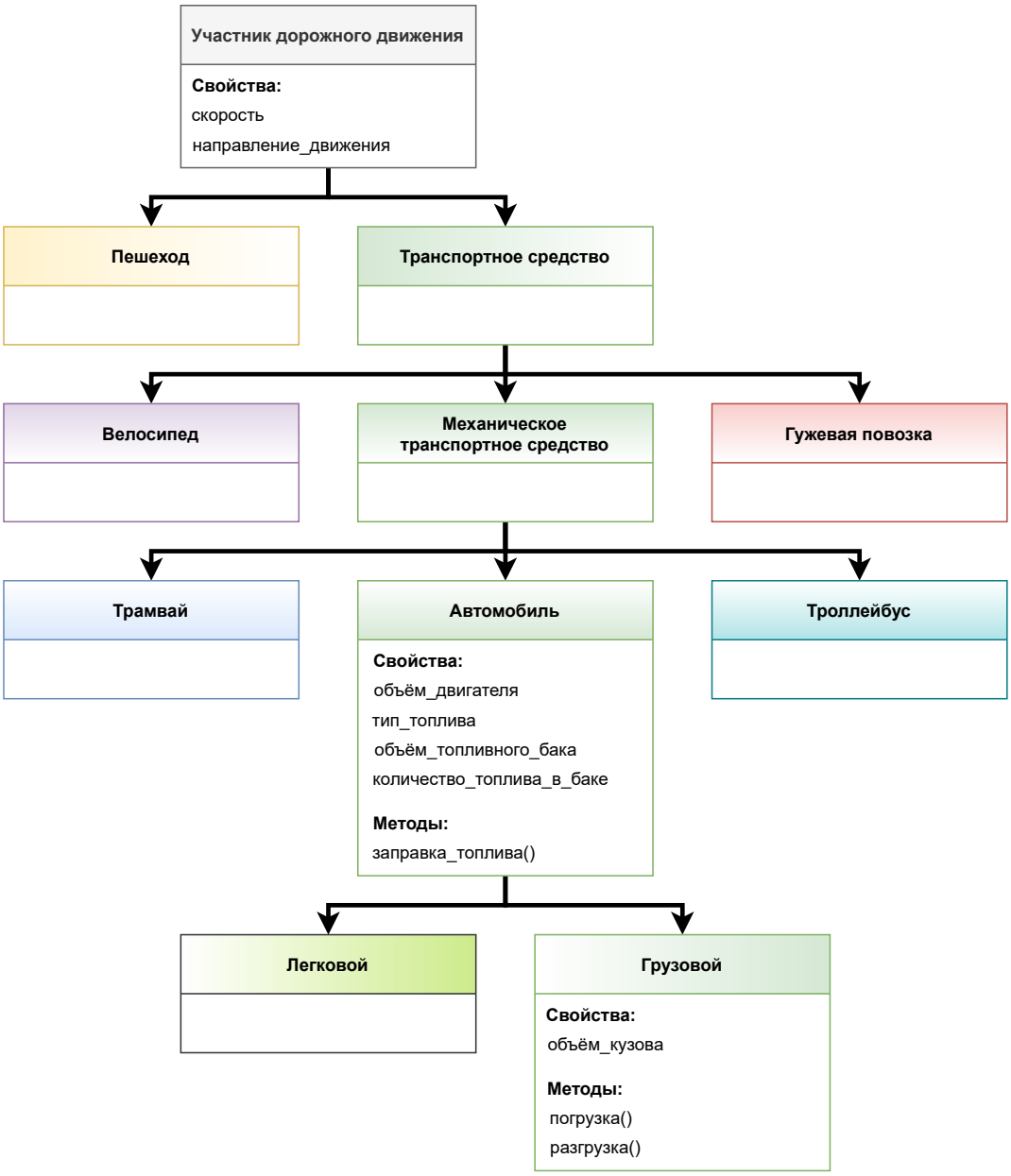

Рис. 1: Пример иерархической категоризации объектов — совокупность типов данных в программе, моделирующей дорожное движение

2 Наследование структур и полиморфизм адресов

С точки зрения структур данных, расположенных в памяти, наследование представляет собой добавление новых полей данных к ранее описанной структуре. Теоретически, такое добавление представляет собой уточнение сведений об описываемом объекте, и, следовательно, **наследование — это переход от общего к частному.**

Пример 1. *Наследование свойств структуры* ***person*** *структурой* ***student***

```
1 struct person {
2     char name[64];
3     char sex;            // 'm' or 'f'
4     int year_of_birth;
5 };
```

Листинг 1: Структура данных, описывающая человека (персону)

```
1 struct student : person {   // Inherit fields of the person
     structure
2     int code;               // specialty code
3     int year;               // year of study
4     double gpa;             // Grade Point Average
5 };
```

Листинг 2: Структура данных, описывающая студента как частный случай персоны

Если теперь описать переменную типа `student`, то она будет иметь все свойства структуры `person` плюс дополнительные поля.

```
1 student s1;
2 strcpy(s1.name, "Ivanov Ivan");
3 s1.sex = 'm';
4 s1.year_of_birth = 2000;
5 s1.code = 335;
6 s1.year = 3;
7 s1.gpa = 4.5;
```

Листинг 3: Допустимые действия со структурой `student`

Определение 2. *Структура* ***person*** *в данном случае называется* ***базовой*** *или* ***родительской****, а структура* ***student*** *—* ***унаследованной****,* ***порождённой*** *или* ***дочерней****. Применяются также термины* ***предок*** *и* ***потомок****.*

Поля, относящиеся к части структуры `student`, унаследованной от `person`, располагаются на тех же местах (по тем же смещениям относительно начала), где они располагались бы в структуре `person` (см. рис. 2).

Таким образом, при необходимости можно работать с переменной `s1` также, как если бы она была переменной типа `person`, а не `student`. Обратное не верно!

Определение 3. *Язык C++* ***разрешает неявное преобразование адресов структур-потомков к адресам структур-предков.***

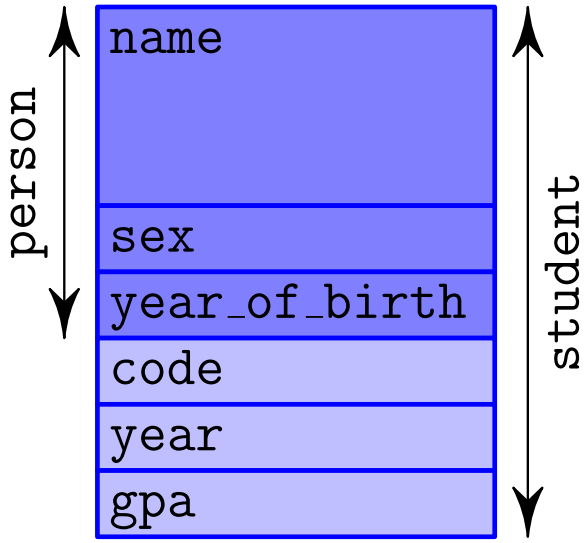

Рис. 2: Свойства унаследованной структуры данных

Такие преобразования разрешены и для указателей, и для ссылок. Это свойство предков, потомков и их адресов называется ***полиморфизмом адресов****.*

Пример 2. *Полиморфизм адресов при наследовании.*

```
1 student s1;
2 person *p;
3 p = &s1;              // alright
4 person &ref = s1;     // alright
5 student *s2 = p;      // error!
```

Листинг 4: Первый пример полиморфизма адресов

Если в программе описана функция, принимающая на вход указатель или ссылку на объект типа `person`, то ей можно передать указатель или ссылку на объект типа `student`.

```
1 void f(person &pers) {
2     // ...
3 }
4 // ...
5 student s1;
6 // ...
7 f(s1);           // alright
```

Листинг 5: Второй пример полиморфизма адресов

3 Защита при наследовании и методы

Несмотря на то, что наследование может применяться как для структур (в классическом понимании языка C), так и для классов, чаще всего этот механизм применяется для структур и классов (напомним, в понимании языка C++ это почти одно и тоже, разница только в модели защиты по умолчанию), имеющих функции-члены (методы). Вообще, можно без ограничений наследовать классы от структур и наоборот, но обычно при описании объектов программисты, работающие в парадигме ООП используют, как правило, именно классы.

Взаимодействие механизма наследования с механизмом защиты — отдельная достаточно непростая тема. Это связано с тем, что сам факт наследования одного класса от другого может в некоторых случаях (хотя это и не обязательно) рассматриваться как *частная деталь реализации* данного класса. В таких случаях необходимо *сокрытие этого фрагмента программы от остального кода*. В связи с этим в C++ различают два основных типа[1] наследования:

- открытое (public);
- закрыте (private).

Модель защиты для конкретного случая наследования указывается в заголовке класса непосредственно перед названием класса-предка.

```
1 class B : public A {
2     // ...
3 };
4
5 class C : private A {
6     // ...
7 };
```

Листинг 6: Открытое и закрытое наследование

В первом случае все свойства класса `A` (его открытые методы и поля, если таковые имеются) будут доступны для объектов класса `B` отовсюду. Во втором случае — только из методов класса `C`, а вся остальная программа вообще не будет знать, что класс `C` унаследован от `A`.

Замечание 1. *Тип наследования можно не указывать, как это было сделано в примерах раздела 2. В этом случае будут действовать умолчания: для структуры наследование по-умолчанию открытое (public), для класса — закрытое (private).*

[1] На самом деле типов наследования три, но мы опишем это несколько позже.

На практике необходимость закрытого наследования возникает крайне редко.

Для базового класса унаследованный класс ничем не отличается от текста всей остальной программы и, как любой недружественный фрагмент кода, не должен иметь доступа к деталям реализации класса. Поэтому закрытые поля и методы базового класса не будут доступны порождённым классам.

В реальных задачах часто возникает необходимость в таких деталях интерфейса класса, которые предназначены только и исключительно для его потомков. Именно для таких случаев, наряду с режимами `public` и `private`, вводится третий режим защиты — защищённый (`protected`).

Определение 4. *Поля и методы класса, отмеченные словом `protected`, будут доступны в самом классе (т. е. в его методах), в дружественных функциях, а также в методах потомков данного класса. Во всей остальной программе такие поля и методы недоступны.*

Замечание 2. *Поля и методы, имеющие защищённый режим защиты (`protected`), не являются в полном смысле слова деталями реализации, которые не касаются никого, кроме данного класса.*

Особенности реализации класса, отмеченные ключевым словом `protected`, могут использоваться в самых непредсказуемых ситуациях: достаточно кому-либо где-то описать потомка данного класса. Поэтому (в отличие от приватных полей и методов) поля и методы с режимом `protected` обязательно должны документироваться, а изменять их стоит столь же осторожно, как и открытую (`public`) часть класса. В некотором смысле `protected` — это особый вид *открытых* особенностей класса.

В связи с появлением третьего режима защиты, возникает и третий вариант наследования. Таким образом, в C++ существует

- Public-наследование. Не изменяет режима доступа к элементам базового класса из производного. При таком наследовании общедоступные (`public`) элементы базового класса останутся общедоступными элементами в производном классе. Закрытые (`private`) элементы станут частью производного класса, но к ним можно будет обращаться только посредством общедоступных и защищенных методов базового класса. Защищенные (`protected`) элементы базового класса будут также защищенными и в производном классе.

- Private-наследование. Осуществляет изменение режима доступа к элементам базового класса: общедоступные и защищенные элементы базового класса становятся закрытыми элементами в производном классе. Это означает, что методы базового класса уже не являются частью общедоступного интерфейса производного класса и

могут использоваться только внутри функций производного класса. Часто такое наследование используют, чтобы ограничить использование методов базового класса в производном.

- Protected-наследование. Является разновидностью закрытого наследования. В этом случае общедоступные элементы базового класса становятся защищенными. Основное отличие `private` и `protected`-наследования проявляется при создании нового класса из производного. При `private`-наследовании класс третьего поколения не будет иметь доступа к методам класса первого поколения, т. к. они становятся `private`-методами в классе второго поколения.

```
1  class A {
2  public:          // inherited with public, protected, private
3      int a1;
4  protected:       // inherited with public, protected, private
5      int a2;
6  private:         // not inherited
7      int a3;
8  };
9
10 class B : public A {
11 public:
12     int b;       // In addition, it includes a1 as public and
           a2 as protected
13 };
14
15 class C1: protected A {
16 public:
17     int c1;      // In addition, it includes a1 and a2 as
           protected
18 };
19
20 class D1: private A {
21 public:
22     int d1;      // In addition, it includes a1 and a2 as
           private
23 };
```

Листинг 7: Различные режимы наследования

Если объекту-потомку доступен (т. е. не скрыт механизмом защиты) метод, введённый в его классе-предке, то такой метод можно вызвать для объекта-потомка точно так же, как и для объекта-предка. При этом, если не предпринять специальных мер, то методы базового класса ничего не будут знать о том, что их вызывают для порождённого объекта, т. е. будут работать абсолютно также как если бы их вызывали для объекта-

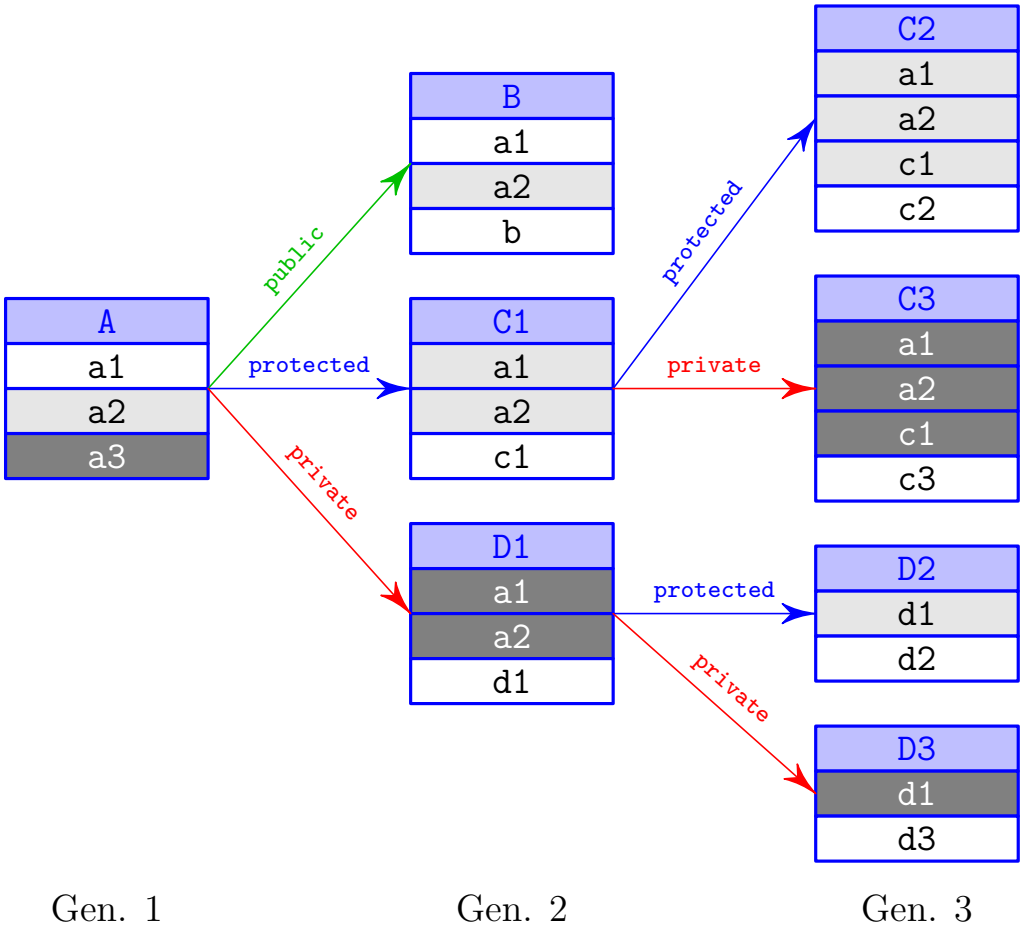

Рис. 3: Различные режимы наследования для трёх поколений классов

предка. В этом проявляется описанный в предыдущем разделе адресный полиморфизм в применении к параметру `this`.

В терминологии ООП это означает, что потомок умеет отвечать на все виды сообщений, предусмотренные для его предков. Для того, чтобы объекту можно было отправить такое сообщение (т. е. вызвать метод предка), достаточно иметь доступ к соответствующему методу (т. е. он не должен быть скрыт защитой). Другими словами, с объектом-наследником можно работать точно так же как с объектом-предком, не обращая никакого внимания на тот факт, что это объекты разных типов. В связи с этим часто говорят, что **объект порождённого класса также является и объектом базового класса.** С другой стороны, **объект порождённого класса содержит в себе объект базового класса в качестве своей части.** В этом нет никакого противоречия, т. к. это две различных модели восприятия: первая — точка зрения логики проектирования программы (пользовательская семантика), вторая — точка зрения реализации средств языка (реализаторская семантика).

4 Конструкторы и деструкторы при наследовании

В контексте наследования особо следует рассмотреть такие методы как конструкторы и деструкторы. С какой бы точки зрения ни рассматривался порождённый объект, очевидно, что в момент его создания также создаётся и объект базового класса (неважно, считаем мы его «частью» наследника или «другой его ипостасью»). Аналогично, при уничтожении порождённого объекта исчезает и базовый объект. Следовательно, при создании объекта-потомка должен отработать и конструктор предка, а при уничтожении наследника — деструктор его родительского класса. Несложно понять, что базовый объект должен быть проинициализирован раньше, а уничтожен позже, чем порождённый объект.

С деструктором всё происходит очевидно — компилятор сначала вызывает тело деструктора порождённого класса, а затем — тело деструктора базового класса.

С конструктором дело обстоит сложнее, т. к. в общем случае конструкторы могут требовать входных параметров (мы с этим неоднократно сталкивались при обсуждении конструкторов в Лекциях №3–4). Решается эта проблема в данном случае абсолютно аналогично тому, как это происходило в разделе 9 Лекций №3–4 (см. Листинги 58 и 59): в описании конструктора порождённого класса между заголовком и телом вставляется список инициализаторов, начинающийся с инициализатора базового класса с указанием всех нужных параметров конструктора.

Пример 3. *Конструирование предка*

Пусть `A` — базовый класс, конструктор которого требует двух параметров типа `int`, `B` — класс, унаследованный от `A`, имеющий конструктор по умолчанию и имеющий поле `int f`:

```
1  class A {
2      // ...
3  public:
4      A(int a, int b) {
5          // ...
6      }
7      // ...
8  };
9
10 class B : public A {
11     int f;
12 public:
13     B();
14     // ...
```

```
15 };
```

В этом случае описание конструктора класса `B` может выглядеть так:

```
1 B::B() : A(1, 2), f(3) {
2     // ...
3 }
```

5 Виртуальные функции и динамический полиморфизм

До сих пор предполагалось, что объект-потомок будет реагировать на сообщения, определённые для его предка абсолютно так же, как на них реагировал бы объект-предок. В определённых случаях такое поведение не отвечает потребностям моделируемой предметной области: бывает так, что на некоторые сообщения потомок должен реагировать иначе, чем предок. Решить эту проблему позволяет механизм виртуальных функций.

Определение 5. *Механизм* ***виртуальных функций*** *позволяет автору класса-предка предоставить возможность классам-потомкам частично или полностью модифицировать поведение отдельных методов.*

Пример 4. *Рассмотрим некоторую задачу, связанную с компьютерной графикой.*

Пусть требуется описать сцену (т. е. набор графических элементов) в виде некоторого списка или массива объектов, задающих графические объекты различного типа, например, отдельные пиксели, линии, окружности и т. п.

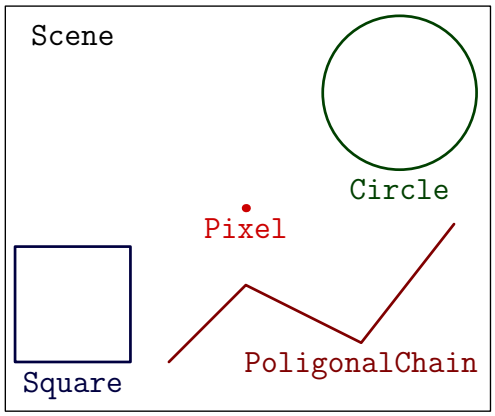

Рис. 4: Элементы графической сцены

Опишем класс, объекты которого будут представлять на нашей сцене простейшие графические примитивы — отдельные пиксели. Пиксель будем характеризовать двумя координатами (числа с плавающей точкой) и цветом (целое число). Будем считать, что основные действия с графическим объектом — это:

- показать объект на экране (метод `Show`);
- убрать его с экрана (метод `Hide`);
- и переместить его с на новую позицию (метод `Move`).

```
1  class Pixel {
2      double x, y;
3      int color;
4  public:
5      Pixel(double ax, double ay, int acolor)
6          : x(ax), y(ay), color(acolor) {}
7      void Show();
8      void Hide();
9      void Move(double nx, double ny);
10 };
```

Листинг 8: Первый вариант заголовка класса `Pixel`

Конкретика описания тел методов `Show` и `Hide` зависит от используемой графической библиотеки, правил пересчёта координат и тому подобных деталей, на которых мы не будем останавливаться (просто предположим, что эти функции описаны в файле реализации тем или иным способом). С помощью функций `Show` и `Hide` очень просто описать метод `Move`, так как перемещение объекта состоит в том, что его сначала надо убрать с экрана, после чего необходимо изменить его координаты и снова показать.

```
1 void Pixel::Move(double nx, double ny) {
2     Hide();
3     x = nx;
4     y = ny;
5     Show();
6 }
```

Листинг 9: Реализация метода `Move`

Теперь предположим, что нам необходим объект «окружность». Такой объект также обладает свойствами координаты центра и цвет, но в дополнение к ним характеризуется ещё и радиусом. Поэтому можно использовать уже описанный класс `Pixel` в качестве базового, и унасле-

довать от него класс `Circle`[2]. При этом, естественно, методам класса `Circle` потребуется доступ к полям координат и цвета. Решим вопрос доступа к этим полям самым простым (хотя и не самым лучшим способом): сделаем поля базового класса защищёнными (`protected`), а не закрытыми:

```
1 class Pixel {
2 protected:
3     double x, y;
4     // everything else is unchanged
5 };
```

Теперь можно перейти к описанию класса `Circle`.

```
1 class Circle : public Pixel {
2     double radius;                                  // additional
        field
3 public:
4     Circle(double x, double y, double r, int color)
5         : Pixel(x, y, color), radius(r) {}      // constructor
6     void Show();
7     void Hide();
8 };
```

Листинг 10: Заголовок класса `Circle`

Как и в классе `Pixel`, тела функций `Show` и `Hide` зависят от применяемой графической библиотеки, поэтому их описание приводить не будем. Отметим только, что эти функции, естественно, будут отличаться от соответствующих функций из класса `Pixel`.

Теперь надо было бы описать метод `Move`, аналогичный тому, что был описан выше для класса `Pixel`. Вместо этого обсудим следующее. Функция `Move` для порождённого класса окажется абсолютно такой же, как и для аналогичная функция родительского класса. Таким образом, придётся написать не только такой же заголовок функции, но и точно такое же тело функции!

Возникает вопрос: можно ли для класса `Circle` использовать ту же самую функцию `Move`, которая уже описана для класса `Pixel`, не описывая новой её версии для класса `Circle`? Легко понять, что простой вызов функции `Move` для объекта класса `Circle`, конечно, возможен (т. к. это открытый метод базового класса и наследование выполнено по открытой схеме), **но не приведёт к корректному результату.** Причина этого состоит в следующем. Во время трансляции тела функции `Pixel::Move` компилятор может ничего не знать о существовании потомков у класса `Pixel`, которые при этом вводят свои версии методов `Show` и `Hide`.

[2]Строго говоря, окружность — это не частный случай точки, поэтому такой подход не соответствует принципам ООП и может рассматриваться только как простейший пример наследования.

Поэтому компилятор, естественно, вставит в машинный код функции `Pixel::Move` обычные вызовы функций[3] `Pixel::Show` и `Pixel::Hide`.

Теперь очевидно, что при вызове функции `Move` для объекта класса `Circle`, она для стирания с экрана объекта вызовет метод `Hide` в той его версии, в которой он описан для класса `Pixel`. Это приведёт к стиранию пиксела вместо окружности. Аналогичная проблема возникнет и для функции `Show`. Очевидно, что происходящее в описываемой ситуации, как минимум, некорректно.

Вместе с тем, описывать для класса `Circle` метод `Move`, слово в слово повторяющий аналогичный метод из базового класса тоже некорректно. Для решения подобных проблем как раз и используется механизм виртуальных функций. Как уже было сформулировано в Определении 5, виртуальная функция (или виртуальный метод) — это функция, описывающая такое действие над объектом, относительно которого предполагается, что аналогичное действие будет определено и для объектов-потомков, но, возможно, для потомков оно будет выполняться иначе.

Определение 6. *В теории ООП говорят, что* ***виртуальным методом*** *задаётся реакция объекта класса на некторый тип сообщений в случае, если:*

1. *предполагается, что у данного класса будут классы-потомки;*
2. *объекты классов-потомков будут способны получать сообщения того же типа;*
3. *объекты некоторых или всех потомков будут реагировать на эти сообщения иначе, чем это делает объект класса-предка.*

Замечание 3. *В C++ виртуальной можно объявить любую функцию-метод, кроме конструкторов и статических методов. Для этого перед заголовком функции необходимо написать ключевое слово* `virtual`.

В отличие от вызовов обычных функций, вызовы виртуальных функций обрабатываются компилятором в предположении, что тип объекта, для которого вызывается функция, может отличаться от базового класса, и что для этого объекта может потребоваться вызов другой функции, нежели для базового класса. В частности, если в классе `Pixel` отметить функции `Show` и `Hide` как виртуальные, то при компиляции тела функции `Move` компилятор будет знать, что объект, на который указывает параметр `this`, может быть как объектом самого класса `Pixel`, так и объектом

[3] То есть инструкции `call` с жёстко заданными исполнительными адресами, не подразумевающие никаких изменений.

любого его класса-потомка (причём для такого объекта может потребоваться вызов других версий функций `Show` и `Hide`, введённых в соответствующем потомке). Код, сгенерированный компилятором, при этом будет несколько сложнее, чем для обычного вызова функции, но зато он будет вызывать именно ту версию функции, которая соответствует типу объекта.

Технически это достигается следующим образом. Если в классе описана хотя бы одна виртуальная функция, то компилятор вставляет во все объекты этого класса невидимое поле, называемое **указателем на таблицу виртуальных методов (vmtp — virtual method table pointer).** Для всего класса создаётся в одном экземпляре неизменяемая **таблица виртуальных методов,** содержащая указатели на каждую из описанных в классе виртуальных функций. Когда компилятор встречает вызов виртуальной функции, он вставляет в объектный код следующее:

1. инструкцию извлечь из объекта значение поля `vmtp`,
2. обратиться по полученному адресу к таблице виртуальных методов и из неё взять адрес требуемой функции,
3. используя полученный адрес, обратиться к искомой функции.

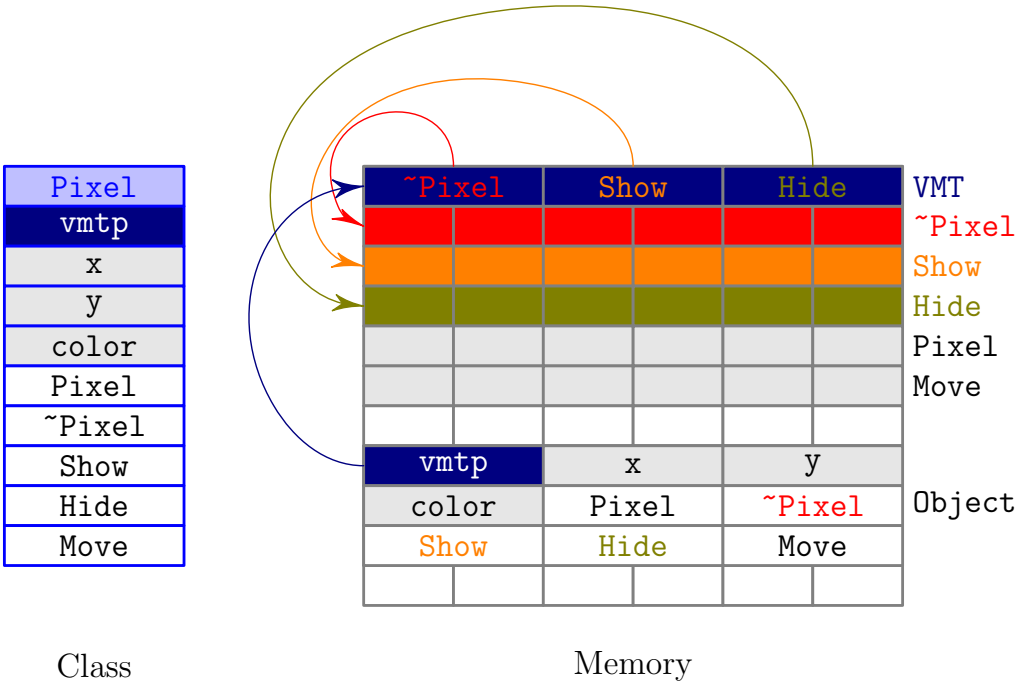

Рис. 5: Реализация идеи виртуальных методов на примере класса `Pixel`

Объект класса-потомка содержит все поля, присущие классу-предку. Это относится и к полю `vmtp`, однако объекты класса-потомка имеют

в этом поле иное значение, нежели объекты класса-предка. Для каждого класса-потомка компилятор создаёт (в единственном экземпляре) собственную таблицу виртуальных методов, содержащую адреса соответствующих версий виртуальных функций (адрес именно этой таблицы заносится в поле `vmtp`). За инициализацию `vmtp` отвечает конструктор класса, в начало которого компилятор вставляет соответствующие команды.

Итак, объявим функции `Show` и `Hide` виртуальными в базовом классе. Это позволит использовать функцию `Move` для любого из классов, унаследованных от `Pixel`, если только для этих классов описаны собственные версии функций `Show` и `Hide`.

Замечание 4. *Если в классе описана хотя бы одна виртуальная функция, то рекомендуется всегда явно описывать деструктор и объявлять его виртуальным*[4].

```
1  class Pixel {
2  protected:
3      double x, y;
4      int color;
5  public:
6      Pixel(double ax, double ay, int acolor)
7          : x(ax), y(ay), color(acolor) {}
8      virtual ~Pixel() {}
9      virtual void Show();
10     virtual void Hide();
11     void Move(double nx, double ny);
12 };
```

Листинг 11: Второй вариант заголовка класса `Pixel`

```
1  class Circle : public Pixel {
2      double radius;
3  public:
4      Circle(double x, double y, double r, int color)
5          : Pixel(x, y, color), radius(r) {}
6      virtual ~Circle() {}
7      virtual void Show();
8      virtual void Hide();
9  };
```

Листинг 12: Второй вариант заголовка класса `Circle`

Замечание 5. *Ключевое слово* ***`virtual`*** *в описании функций* ***`Show`*** *и* ***`Hide`*** *в классе* ***`Circle`*** *можно опустить, т. к. функции, совпадающие по заголовку с виртуальными функциями базового класса, объявляются виртуальными автоматически.*

[4]Смысл этого будет описан немного позднее в разделе 9.

Виртуальные функции можно вызвать не только из других методов того же класса, как это происходит в рассматриваемом нами примере.

Правило 1. *Компилятор сгенерирует код для вызова через таблицу виртуальных методов при любом обращении к виртуальной функции, если тип объекта, для которого её вызывают хотя бы теоретически может меняться. Из этого правила есть только одно важное исключение, которое будет обсуждаться далее в разделе 13.*

В тривиальном случае, когда объект описан в виде обычной переменной типа класс и мы обращаемся к его методу (пусть и виртуальному) напрямую через имя этой переменной и точку

```
1 A a;
2 // ...
3 a.func();    // is always called f()
4              // virtuality mechanism is not involved
```

— то такое обращение бессмысленно реализовывать через виртуальность, так как тип переменной известен во время компиляции и другим стать не может. Иное дело, если обращение к виртуальной функции записано через адрес объекта (указатель или ссылку) (например, при вызове методов из тел других методов, где объект идентифицируется указателем `this`). Ещё один пример, иллюстрирующий этот факт:

```
1 void f(const Pixel& p) {
2     // ...
3     p.Show();
4     // ...
5 }
6 // ...
7 Circle C(0, 0, 1, 5);
8 // ...
9 f(C);            // correct!
```

6 Чисто виртуальные методы и абстрактные классы

Наследование класса `Circle` от класса `Pixel` нарушает принципы ООП, т. к. окружность — это не частный случай точки. Однако и точка, и окружность — это частные случаи *геометрических фигур.* Каждая геометрическая фигура характеризуется цветом и имеет координаты *точки привязки.*

Такое представление об абстрактном понятии геометрической фигуры позволяет указать единый для всех фигур алгоритм передвижения фи-

гуры по экрану: стереть фигуру с экрана, изменить координаты точки привязки, отрисовать фигуру в новом месте.

Опишем класс для представления абстрактной геометрической фигуры `GraphObject` и унаследуем от него оба класса `Pixel` и `Circle`. Однако перед тем как это сделать, отметим ещё один важный момент. Для абстрактной геометрической фигуры тела методов `Show` и `Hide` описать *невозможно в принципе.*

Несмотря на это, мы точно знаем как написать функцию `Move` в предположении, что для классов-потомков данного класса будут правильно описаны методы `Show` и `Hide`. Другими словами, мы знаем, что все объекты классов-потомков данного класса должны уметь получать сообщение определённого типа, но при описании базового класса мы не можем задать какую бы то ни было реакцию на такие события, т. к. она полностью зависит от конкретного типа потомка. При этом для описания некоторых других (более общих) методов базового класса нам требуется знание о том, что реакция на соответствующие события будет предусмотрена во всех классах-потомках.

Специально для таких случаев в Си++ введены ***чисто виртуальные функции*** (*pure virtual functions*). Описывая в классе чисто виртуальный метод, программист сообщает компилятору о том, что метод с таким профилем будет существовать во всех классах-потомках, и под него нужно зарезервировать позицию в таблице виртуальных функций, но при этом само тело такого метода для базового класса описываться не будет, так что значение адреса этого метода в таблице виртуальных функций следует оставить нулевым. Синтаксис описания чисто виртуальной функции:

```
1 class A {
2     // ...
3     virtual void f() = 0;
4     // ...
5 };
```

Листинг 13: Синтаксис чисто виртуальной функции

Специальная лексическая последовательность «`= 0`» сообщает компилятору о том, что данная функция чисто виртуальная.

Определение 7. *Класс, в котором есть хотя бы одна чисто виртуальная функция называется* ***абстрактным классом.***

Замечание 6. *Компилятор не позволяет создавать объекты абстрактных классов.*

Единственное назначение абстрактного класса — служить базисом для порождения других классов, в которых все чисто виртуальные мето-

ды будут конкретизированы. Если в порождаемом классе не описан хотя бы один метод, объявленный в базовом классе как чисто виртуальный, то компилятор будет считать, что метод остался чисто виртуальным, а сам класс-потомок (как и его предок) представляет собой абстрактный класс.

```
1  class GraphObject {
2  protected:
3      double x, y;
4      int color;
5  public:
6      GraphObject(double ax, double ay, int acolor)
7          : x(ax), y(ay), color(acolor) {}
8      virtual ~GraphObject() {}
9      virtual void Show() = 0;
10     virtual void Hide() = 0;
11     void Move(double nx, double ny);
12 };
```

Листинг 14: Заголовок абстрактного класса `GraphObject`

Описание функции `Move` слово в слово повторяет описание метода `Pixel::Move()` приведённое раньше.

Заголовки классов `Pixel` и `Circle` теперь будут выглядеть следующим образом.

```
1  class Pixel : public GraphObject {
2  public:
3      Pixel(double x, double y, int color)
4          : GraphObject(x, y, color) {}
5      virtual ~Pixel() {}
6      virtual void Show();
7      virtual void Hide();
8  };
```

Листинг 15: Третий вариант заголовка класса `Pixel`

```
1  class Circle : public GraphObject {
2      double radius;
3  public:
4      Circle(double x, double y, double r, int color)
5          : GraphObject(x, y, color), radius(r) {}
6      virtual ~Circle() {}
7      virtual void Show();
8      virtual void Hide();
9  };
```

Листинг 16: Третий вариант заголовка класса `Circle`

Замечание 7. *Для этих классов, в отличие от класса* ***`GraphObject`****,*

необходимо описать конкретные тела методов `Show` *и* `Hide`*, иначе программа не пройдёт этап линковки.*

7 Виртуальность в конструкторах и деструкторах

Вызов виртаульных функций в телах конструкторов и деструкторов имеет одну нетривиальную особенность, связанную с конструированием и последующим деструктированием в объекте невидимого поля, содержащего адрес таблицы виртуальных функций. Это поле заполняет конструктор, но, очевидно, что он делает это *после того, как остальные подобъекты рассматриваемого объекта уже сконструированы,* в том числе сконструирован и предок. При этом, конструктор предка тоже содержит код, инициализирующий указатель на таблицу виртуальных методов, а т. к. про предков он ничего не знает, в поле `vmtp` он занесёт адрес *своей* таблицы.

Таким образом, если в классе вообще имеются виртуальные функции, то во время выполнения тела конструктора вызываются те виртуальные функции, которые описаны для данного класса, вне зависимости от того, какого класса на самом деле конструируется объект. Аналогичный эффект присутствует также в телах деструкторов — после завершения своего тела деструктор деинициализирует указатель на виртуальную таблицу — реально это означает, что в поле `vmtp` записывается адрес таблицы виртуальных методов предка.

В частности, из конструкторов и деструкторов вообще нельзя вызывать методы, описанные в данном объекте как чисто виртуальные.

Правило 2. *При возникновении любых сомнений на этот счёт лучше вообще воздержаться от обращения к виртуальным методам из конструкторов и деструкторов.*

8 Наследование ради конструктора

На практике часто применяется упрощённый случай наследования, при котором класс-потомок отличается от предка только набором конструкторов, то есть он не вводит ни новых свойств, ни новых методов. Такие классы создаются из соображений экономии объёма кода, чтобы не повторять одни и те же действия при конструировании однотипных объектов. Для иллюстрации этого случая введём сначала ещё одного потомка класса `GraphObject`, представляющего собой ломаную линию, а

затем опишем графический объект «квадрат» как частный случай ломаной линии.

Как было определено в разделе 6, каждая геометрическая фигура в нашей системе имеет точку привязки. Ломаную линию при этом проще всего представлять в виде списка координатных пар, задающих смещение каждой вершины относительно точки привязки. Для организации такого списка опишем в закрытой части класса структуру, задающую элемент списка. Изначально будем создавать ломаную, не имеющую ни одной вершины, а для добавления новых вершин будем применять метод `AddVertex`[5].

```
class PolygonalChain : public GraphObject {
    struct Vertex {
        double dx, dy;
        Vertex *next;
    };
    Vertex *first;
public:
    PolygonalChain(double x, double y, int color)
        : GraphObject(x, y, color), first(0) {}
    virtual ~PolygonalChain();
    void AddVertex(double adx, double ady);
    virtual void Show();
    virtual void Hide();
};
```

Листинг 17: Заголовок класса `PolygonalChain`

Для простоты, функция `AddVertex` будет добавлять новую вершину в начало списка, а не в его конец. Линия, конечно, будет отрисовываться при этом в обратном порядке, но по существу это ничего не изменит.

```
void PolygonalChain::AddVertex(double ax, double ay) {
    Vertex *tmp = new Vertex;
    tmp->dx = ax;
    tmp->dy = ay;
    tmp->next = first;
    first = tmp;
}
```

Листинг 18: Реализация функции `AddVertex`

Так как объекты класса `PolygonalChain` используют динамическую память для хранения списка вершин, то обязательно необходимо описать деструктор.

```
PolygonalChain::~PolygonalChain() {
    while(first) {
```

[5]Такая организация, во всяком случае, относительно проста, и позволяет не использовать динамических структур данных для хранения массива координат неизвестного заранее числа вершин.

```
3           Vertex *tmp = first;
4           first = first->next;
5           delete tmp;
6       }
7   }
```

Листинг 19: Деструктор класса `PolygonalChain`

Также как это было и ранее, функции `Show` и `Hide` предполагаются написанными так, как того требует используемая графическая библиотека.

Теперь создадим класс для представления квадрата, стороны которого параллельны осям координат, а длина стороны задаётся параметром конструктора. Очевидно, что такой квадрат — это частный случай ломаной, описываемой классом `PolygonalChain`. Если в качестве привязки выбрать нижнюю левую вершину квадрата, а длину его стороны обозначить символом a, то соответствующая ломаная должна начаться в точке привязки (что отвечает координатам $(0, 0)$), пройти через точки $(a, 0)$, (a, a), $(0, a)$ и вернуться в точку $(0, 0)$. Таким образом, всего ломаная содержит пять вершин и оказывается замкнутой, поскольку первая и последняя вершины совпадают.

Для того, чтобы не приходилось каждый раз при описании квадрата писать шесть строк кода (одну для создания объекта `PolygonalChain` и пять — для добавления вершин), можно описать класс `Square`, унаследовав его от `PolygonalChain`.

```
1  class Square : public PolygonalChain {
2  public:
3      Square(double x, double y, double a, int color)
4          : PolygonalChain(x, y, color) {
5          AddVertex(0, 0);
6          AddVertex(a, 0);
7          AddVertex(a, a);
8          AddVertex(0, a);
9          AddVertex(0, 0);
10     }
11 };
```

Листинг 20: Класс `Square`

Единственное отличие классов `Square` и `PolygonalChain` только в конструкторе. Для квадрата больше ничего дополнительно описывать не нужно, так как для всего необходимого уже есть методы базового класса.

9 Виртуальный деструктор

В замечании 4 было отмечено, что в классе, имеющем хотя бы одну виртуальную функцию, деструктор также необходимо объявить виртуальным. Разберём в данном разделе причины этого.

Для этого заметим, что при активном использовании полиморфизма могут возникать ситуации, когда нужно применить оператор `delete` к указателю, имеющему тип «указатель на базовый класс». В то же время, принятое в C++ соглашение о преобразовании адресов по закону полиморфизма (см. определение 3 в разделе 2) разрешает данному указателю в действительности адресовать объект не базового, а порождённого класса. В этом случае, очевидно, необходимо вызвать деструктор, соответствующий типу уничтожаемого объекта, а не указателя. Приведём конкретный пример подобной ситуации:

Пример 5. *Иллюстрация необходимости виртуальных деструкторов*

```
1 GraphObject *ptr;
2 // ...
3 ptr = new Square(0, 0, 10, 0x00FF00);   // correct!
4 // ...
5 delete ptr;                 // correct, because destructor
6                             // of GraphObject class defined
7                             // as virtual
```

В приведённом примере указатель `ptr` имеет тип «`GraphObject*`», но в действительности он адресует объект класса `Square`. Пятая строка данного листинга абсолютно корректна, т. к. деструктор класса `GraphObject` виртуальный и компилятор произведёт вызов деструктора через таблицу виртуальных методов уничтожаемого объекта. В результате этого будет вызван неявный деструктор для класса `Square`, который вызовет деструктор класса `PolygonalChain`, а тот, в свою очередь, — деструктор класса `GraphObject`.

Полезно подумать над тем, что произойдёт в данном примере, если бы деструктор класса `GraphObject` не был объявлен как виртуальный. С точки зрения синтаксиса языка C++ в этом случае всё тоже было бы абсолютно корректно, но из-за допущенного логического просчёта такое решение привело бы к неконтролируемому «засорению» динамической памяти.

Рассмотренный пример объясняет причину замечания 4. Это позволяет сформулировать следующее достаточно общее правило.

Правило 3. *Деструктор любого класса, имеющего хотя бы одну виртуальную функцию, следует объявлять как виртуальный, не за-*

думываясь о том, понадобится ли это в программе или нет. Многие компиляторы выдают предупреждение, если этого не сделать.

10 Практическое использование динамического полиморфизма

Пример 6. *Продолжение Примера 4 — использование полиморфизма на графической сцене*

Пусть мы создаём графическую сцену, состоящую из разных графических объектов, представляемых объектами классов, унаследованных от `GraphObject`. При этом на момент написания программы заранее не известно, сколько и каких именно объектов будет в сцене (например, это может стать известно при чтении конфигурационного файла или сгенерировано случайным образом во время исполнения программы).

Допустим, что, так или иначе, в какой-то момент времени программе становится известно, сколько объектов будет содержать сцена. Это даёт возможность использовать для хранения всей сцены динамически создаваемый массив указателей на объекты потомков `GraphObject`.

```
1 int scene_size;
2 GraphObject **scene;
```

Когда переменная `scene_size` получит своё значение, можно будет завести массив:

```
1 scene = new GraphObject*[scene_size];
```

Теперь, благодаря полиморфизму адресов оказываются корректны (при условии, что `i < scene_size`) следующие присваивания:

```
1 scene[i] = new Pixel(1, 1, 0x0000FF);
2 // ...
3 scene[i] = new Circle(5, 5, 1, 0xFF0000);
4 // ..
5 scene[i] = new Circle(-1, -1, 3, 0x00FF00);
6 // ...
```

и тому подобные. Таким образом, мы создали массив указателей типа `GraphObject*`, каждый из которых в действительности адресует некоторый объект класса-потомка. Интересно, что нам в принципе может никогда не потребоваться знать, на объекты какого типа указывают конкретные указатели в нашем массиве. Вне зависимости от конкретных типов мы можем перемещать объекты по экрану, гасить их и снова отрисовывать, т. к. методы `Move`, `Hide` и `Show` доступны для класса `GraphObject`, а

значит, могут быть вызваны по указателю типа `GraphObject*` без уточнения типа объекта. Таким же образом благодаря наличию виртуального деструктора можно уничтожить все объекты сцены, а затем и саму сцену:

```
for(int i = 0; i < scene_size; i++)
    delete scene[i];
delete [] scene;
```

Ситуации, подобные описанным в данном разделе, достаточно часто возникают в относительно сложных программах.

Определение 8. *Если конкретные методы, которые нужно вызывать, становятся известны только во время исполнения программы, то такой полиморфизм называется* ***динамическим полиморфизмом.***

Замечание 8. *Динамический полиморфизм основан на механизме виртуальных функций и является его предназначением.*

11 Приватные и защищённые деструкторы

Убирая деструктор класса из открытого доступа можно получить специфические эффекты, иногда используемые на практике.

- Так, если описать деструктор в секции `private`, это будет означать, что уничтожение объекта данного класса возможно только в его методах и дружественных функциях.
- Деструктор в секции `protected` добавляет к этому списку потомков данного класса.

Как следствие, объект класса, имеющего приватный деструктор **нельзя создать в виде простой (локальной или глобальной) переменной** за пределами его методов и «друзей». Этот эффект используют, чтобы создать тип объекта, всегда размещаемого в динамической памяти. Обычно такие объекты в тех или иных обстоятельствах удаляют себя сами. Для этого можно, например, предусмотреть в классе специальный метод:

```
class Samurai {
    ~Samurai() {}      // destructor is private
public:
    // ...
    void Harakiri() {
        delete this;
    }
};
```

Замечание 9. *Приватный деструктор запрещает возможность наследования от такого класса, за исключением единственной ситуации, когда все унаследованые классы описаны в качестве дружественных.*

Замечание 10. *В противоположность предыдущему замечанию, защищённый деструктор явным образом указывает на то, что данный класс спроектирован как основа для создания наследников, а использование его самого по себе не предполагается.*

Если все наследники класса с защищённым деструктором тоже будут вводить свои деструкторы в защищённой части, то мы получим целую полиморфную иерархию классов, объекты которых могут существовать только в динамической памяти.

12 Перегрузка функций и сокрытие имён

Допустим, имеется класс `A` от которого унаследован класс `B`, и при этом в обоих классах есть поля или методы, названные одним и тем же идентификатором (например, `x`)[6].

Правило 4. *Введение поля или метода с именем* **`x`** *в порождённом классе* ***скрывает*** *любые поля или методы базовых классов, имеющие такое же имя.*

Замечание 11. *Вообще, мы не рассматриваем в наших лекциях тему множественного наследования, но можно отметить, что если поля или методы с одинаковыми именами появились в двух базовых классах одного порождённого класса, то в таком порождённом классе сокрытию подвергнутся имена из обоих базовых классов.*

Если в обоих классах (базовом и порождённом) одинаковы имена полей или методы с идентичными профилями (т. е. одинаковым количеством и типами параметров), правило сокрытия очевидно. Также очевидно оно и для случая, когда в базовом классе имеется метод с именем `x`, а в порождённом вводится поле `x`, или наоборот — причина этого в том, что перегрузка имён полей в C++ не предусмотрена.

```
1 class A {
2     // ...
3 public:
```

[6] Конечно, это пример плохой логической организации кода, за исключением специального случая, когда переопределяются виртуальные функции, но стандарт языка C++ допускает такие решения, поэтому полезно о них знать.

```
4     void f(int a, int b);
5 };
6
7 class B : public A {
8     double f;        // method f(int, int) now is hidden
9 };
```

Листинг 21: Первый пример сокрытия имен при наследовании

Если теперь создать объект класса `B`, вызвать метод `f` обычным путём не удастся:

```
1 B b;
2 b.f(1, 2);       // error! method f is hidden.
```

Надо понимать, что «скрыт» не значит «недоступен»! В классе `B` по-прежнему присутствует метод `f`, унаследованный от класса `A`, однако его вызов необходимо выполнять с явным указанием области видимости:

```
1 b.A::f(1, 2);       // alright!
```

Замечание 12. *Наиболее неочевидным следствием правила 4 становится то, что появившаяся в порождённом классе функция с тем же именем, что и метод базового класса* ***скрывает метод базового класса, даже если они различаются профилем.*** *Перегрузка имен функций здесь не работает!*

```
1 class A {
2     // ...
3 public:
4     void f(int a, int b);
5 };
6
7 class B : public A {
8 public:
9     double f(const char *str);  // method f(int, int) is
          hidden
10 };
11
12 B b;
13 double t = b.f("Gibberish!");    // correct
14 b.f(2, 3);                      // error!
15 b.A::f(2, 3);                   // alrigth
```

Листинг 22: Второй пример сокрытия имен при наследовании

Правило 5. *Перегрузка имён функций действует только в рамках одной области видимости. Если имена введены в различных (даже пересекающихся!) областях видимости, то принципы перегрузки на них не распространяются.*

13 Вызов в обход механизма виртуальности

Явное указание области видимости (имени класса), использовавшееся в предыдущем разделе для обращения к скрытым именам, имеет в C++ ещё один эффект.

Правило 6. *При вызове виртуального метода с явным указанием имени класса отключается механизм виртуальности.*

```
1  class A {
2  public:
3      virtual void f() { printf("first\n"); }
4      void g() { f(); }          // method f is called depending
5                                 // on the actual object type
6      void h() { A::f(); }       // method f of class A
7                                 // is always called
8  };
9
10 class B : public A {
11 public:
12     virtual void f() { printf("second\n"); }
13 }
```

Листинг 23: Описание методов с указанием области видимости

```
1 B b;
2 b.f();       // "second"
3 b.A::f();    // "first"
4 b.g();       // "second"
5 b.h();       // "first"
6
7 A *pa = &b;
8 pa->f();     // "second"
9 pa->A::f(); // "first"
```

Листинг 24: Пример вызовов виртуальных метода с использованием и в обход виртуальности

Встречается эта возможность на практике относительно редко, но иногда бывает полезной. Наиболее часто такая возможность используется, когда из «новой» версии виртуального метода, введенной классом-потомком, требуется вызвать «старую» версию, описанную для класса-предка.

14 Наследование как сужение множества

При изучении и использовании объектно-ориентированного программирования используют различные варианты терминологии.

Таблица 1: Примеры эквивалентных терминов в ООП

Теория	**Практика**
передача сообщения объекту	вызов метода
множество объектов, удовлетворяющих некоторым условиям	класс
подмножество объектов	унаследованный класс

Одно из проявлений полиморфизма состоит в том, что объект порождённого класса является одновременно и объектом порождённого, и объектом базового класса. Обратное, вообще говоря, не верно!

Можно считать что, наследование — это уточнение свойств объекта, т. е. переход от общего к частному. Множество частных случаев есть подмножество множества общих случаев. В терминах теории множеств наследование описывается отношением включения $B \subseteq A$.

Определение 9. *Следствием такого рассмотрения являются термины* ***подкласс*** *(subclass) для обозначения порождённого класса и* ***суперкласс*** *(superclass) для — базового.*

Замечание 13. *Такая терминология иногда вызывает некоторую путаницу. Это связано с двумя причинами. Во-первых, объект порождённого класса (подкласса) заведомо занимает не меньше (а как правило, — больше) памяти, чем объект базового класса (суперкласса). Во-вторых, объект суперкласса оказывается подобъектом объекта подкласса.*

Эта проблема вызвана тем, что в разных терминологиях рассматриваются два разных отношения вложенности, направленных противоположно. Чисто технически, объект базового класса вложен в объект порождённого класса. В то же время сами классы (как множества объектов), вложены наоборот — порождённый в базовый.

15 Операции приведения типа

На практике нередко приходится изменять тип выражения. Иногда это происходит *неявно* (пример: вычисление выражения $3 + 2.0$), но достаточно часто требуется *явно указывать компилятору новый тип выражения* (например, при изменении типа указателя). В языке C для этого используется **унарная операция преобразования типа,** символ которой — это имя типа, заключённое в круглые скобки:

```
1 int x;
2 char *p = (char*)&x;
```

Замечание 14. *Операция приведения типа* ***опасна*** *тем, что её применение позволяет при желании обойти любые ограничения, вводимые системой типизации, включая, например, запреты на запись в константные области памяти и даже защиту данных в классах.*

Необдуманное применение преобразования типов приводит к запутыванию программы и, в конечном счёте, к труднообнаруживаемым ошибкам.

Для уменьшения негативных эффектов операции приведения типов, а также поддержки полиморфного программирования в языке C++ вводится четыре дополнительные операции. Синтаксис этих операций достаточно нетривиальный и имеет следующий вид:

```
KeyWord<NewType>(expression)
```

Здесь `KeyWord` — это ключевое слово, задающее операцию (`static_cast`, `dynamic_cast`, `const_cast` или `reinterpret_cast`), `NewType` — это имя нового типа, а `expression` — это само выражение, тип которого требуется изменить.

```
1 Square *Sptr = static_cast<Square*>(scene[i]);
```

Листинг 25: Пример использования операции `static_cast`

В отличие от операции приведения типов в языке C, которая применялась для любых случаев явного изменения типа, каждая из операций C++ предназначена для своей ситуации. Так, операция `const_cast` позволяет снять или, наоборот, установить сколько угодно модификаторов `const` и/или `volatile`[7]. Попытка использовать `const_cast` для любых других изменений типа приведёт к ошибке компиляции.

```
1 int *p;
2 const int *q;
3 const char *s;
4 // ...
5 q = p;                          // allow without cast
6 p = q;                          // error!
7 p = const_cast<int*>(q);        // correct
8 p = const_cast<int*>(s);        // error!
```

[7]Волатильные переменные имеют большое значение в аппаратном и многопоточном программировании, когда нельзя гарантировать сохранения значения переменной после предыдущего обращения к ней. В нашем курсе эти вопросы не затрагиваются.

Замечание 15. *Использование операции* `const_cast` ***не отменяет опасности*** *такого преобразования!*

Реальная необходимость в обходе константной защиты возникает крайне редко. Прежде чем применять такое преобразование, подумайте, — всё ли вы правильно понимаете и делаете? Может быть, вы забыли пометить словом `const` метод, не изменяющий состояние объекта? **Для применения операции** `const_cast` **должны быть очень веские причины**[8].

Операция `static_cast` предназначена для работы с наследуемыми объектами и позволяет преобразовать указатель или ссылку в направлении, противоположном закону полиморфизма, т. е. от базового класса к порождённому. Попытка призвести любое другое преобразование приведёт к ошибке.

```
1  class A { /* ... */ };
2  class B : public A { /* ... */ };
3  class C { /* ... */ };
4  // ...
5  A *ap;
6  B *bp;
7  C *cp;
8  // ...
9  ap = bp;                        // allow without cast
10 bp = ap;                        // error!
11 bp = static_cast<B*>(ap);       // correct
12 cp = static_cast<C*>(ap);       // error! classes A and C are
13                                 // not related by inheritance
```

Листинг 26: Примеры использования `static_cast`

Очевидно, делать это следует только когда мы действительно уверены, что по данному адресу расположен объект именно того типа, к которому мы намерены преобразовать. В противном случае мы столкнёмся с трудновыявляемыми ошибками.

Операция `reinterpret_cast` позволяет произвести любое перобразование (чего угодно во что угодно), если только компилятор понимает, как это сделать (в частности, преобразовать объекты структур разных типов друг к другу не получится). Фактически, эта операция представляет собой эквивалент унарной операции явного приведения типа `(type)` языка C, и рекомендуется применять именно `reinterpret_cast`.

Особое место занимает операция `dynamic_cast`. Три предыдущие операции служат для управления системой контроля типов, т. е. для управления компилятором, и не порождают действий, осуществляемых во

[8] В некоторых коллективах программистов на каждое применение `const_cast` требуется получать личное разрешение руководителя разработки.

время исполнения программы[9]. Операция `dynamic_cast`, в отличие от остальных, предполагает проведение проверки **во время исполнения** программы и, подобно операции `static_cast`, предназначена для преобразования адресов объектов в направлении, противоположном закону полиморфизма, т. е. от адреса предка к адресу потомка. В случае `static_cast` ответственность за корректность такой операции возлагается на программиста. В случае применения `dynamic_cast`, она сама проведёт проверку и в случае, если преобразование некорректно (т. е. по заданному адресу в памяти не находится объект нужного типа), вернёт нулевой указатель (`NULL`). На момент написания программы в общем случае тип объекта неизвестен, так что проверка проводится во время исполнения (т. е. динамически, откуда и название операции).

Проверка типа выполняется на основании значения указателя на таблицу виртуальных функций. Это связано с тем, что такая таблица уникальна для каждого класса, имеющего виртуальные методы, т. е. её адрес однозначно идентифицирует класс объекта. Как следствие, `dynamic_cast` может работать только с классами/структурами, имеющими виртуальные функции[10].

Замечание 16. *Обычно реализации* `dynamic_cast` *работают достаточно медленно.*

Замечание 17. *Преобразование* `dynamic_cast` *умеет работать не только с указателями, но и со ссылками, однако понятия «нулевой ссылки» не существует, поэтому при отрицательном результате проверки эта операция выбрасывает «стандартное исключение», для обработки которого необходимо подключить заголовочный файл стандартной библиотеки C++.*

16 Иерархии исключений

В разделе 7 Лекции № 5 указывалось, что в обработчиках исключений компилятором C++ допускаются следующие преобразования:

- `throw any_type` ⟶ `catch(any_type &);`
- `throw any_type *` ⟶ `catch(const any_type *);`
- `throw any_type &` ⟶ `catch(const any_type &).`

[9] Вообще говоря, здесь есть исключения, но мы не будем их рассматривать.

[10] Иногда такие классы называются **полиморфными,** но это не вполне корректно, т. к. полиморфизм в определённом смысле имеет место и для классов, не имеющих виртуальных функций.

В действительности, допускается ещё один вид преобразований — преобразование адреса (т. е. указателя или ссылки) объекта-потомка к соответствующему адресному типу объекта-предка.

Например, если описать два класса, унаследовав один от другого:

```
1 class A { /* ... */ };
2 class B : public A { /* ... */ };
```

— то обработчик вида

```
1 catch(const A & ex) { /* ... */ }
```

сможет ловить исключения **обоих** типов, т. е. результат как оператора `throw A(...);`, так и `throw B(...);`.

Это свойство используется для создания **иерархии исключительных ситуаций.**

Пример 7. *Иерархия исключений*

- Ошибки, возникающие по вине пользователя:
  - синтаксические ошибки при вводе;
  - неверное имя файла;
  - неправильно введённый пароль;
  - недопустимая комбинация требований.
- Ошибки, обусловленные средой выполнения:
  - переполнение диска;
  - отсутствие файлов, необходимых для работы;
  - недостаток оперативной памяти;
  - ошибки при работе с сетью.
- Ситуации, указывающие на ошибки в самой программе и требующие её исправления.

Для работы со всей совокупностью возможных исключений в данном случае, создадим класс `Error`, отвечающий понятию «любая ошибка». Унаследуем от него подклассы `UserError` (класс пользовательских ошибок), `ExternalError` (класс ошибок среды выполнения) и `Bug` (класс внутренних ошибок программы). От класса `UserError` можно унаследовать классы `IncorrectInput`, `WrongFileName`, `IncorrectPassword` и т. д.

После этого отбаботчик вида

```
1 catch(const IncorrectPassword & ex) { /* ... */ }
```

будет обрабатывать только исключения, связанные с неправильным паролем, тогда как обработчик вида

```
1 catch(const UserError & ex) { /* ... */ }
```

будет реагировать на любые ошибки пользователя. Обработчик

```
1 catch(const Error & ex) { /* ... */ }
```

сможет "поймать" вообще любое исключение из рассматриваемой иерархии.

Замечание 18. *Такое преобразование работает* ***только для адресов,*** *а не для объектов как таковых!*

17 Контрольные вопросы и задания

1. Вернитесь к рис. 3 и изучите его подробно ещё раз совместно с листингом 7. Как можно уточнить данный рисунок? Действительно ли поле `a3` недоступно в классах-наследниках? Чем сокрытие отличается от приватности?

2. Изобразите диаграмму, аналогичную изображённой на рис. 5 для класса `Circle`, унаследованного от класса `Pixel` согласно схеме, описанной в разделе 5. Как изменятся эти диаграммы при переходе к схеме наследования, основанной на абстрактном классе `GraphObject` и обсуждавшейся в разделе 6?

3. Используя такие инструменты, как отладчик `gdb`, а также утилиты `hexdump` и `objdump`, исследуйте на практике реализацию механизмов наследования и виртуальных функций. Для этого напишите ряд несложных программ, использующих наследование, и изучите их машинный код: определите размеры родительского и наследуемых объектов; изучите структуру и адресацию полей и методов этих объектов; определите наличие у объектов поля `vtmp` и его свойства; зафиксируйте расположение в памяти и состав таблицы виртуальных методов.

4. Почему объект класса, имеющего приватный деструктор нельзя создать в виде простой (локальной или глобальной) переменной за пределами его методов и «друзей»?

5. Изучите самостоятельно концепцию множественного наследования. Какие у данного механизма преимущества и недостатки?