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

П. А. Макаров

СЫКТЫВКАРСКИЙ
ЛЕСНОЙ
ИНСТИТУТ

17 ноября и 01 декабря 2023 г.

Содержание

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.*

Иерархия типов объектов

Пример иерархической категоризации объектов

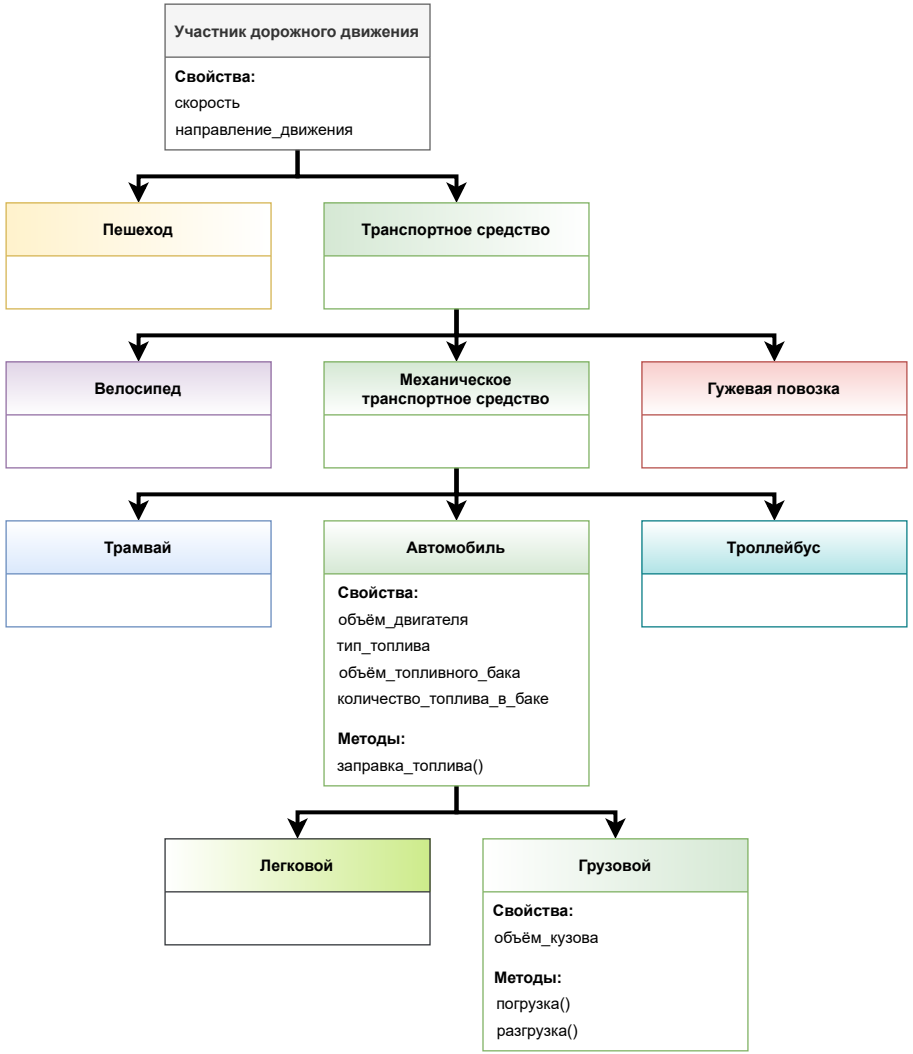

Рис. 1: Совокупность типов данных в программе, моделирующей дорожное движение

Проблема

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.

Выход

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

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

```
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**.*

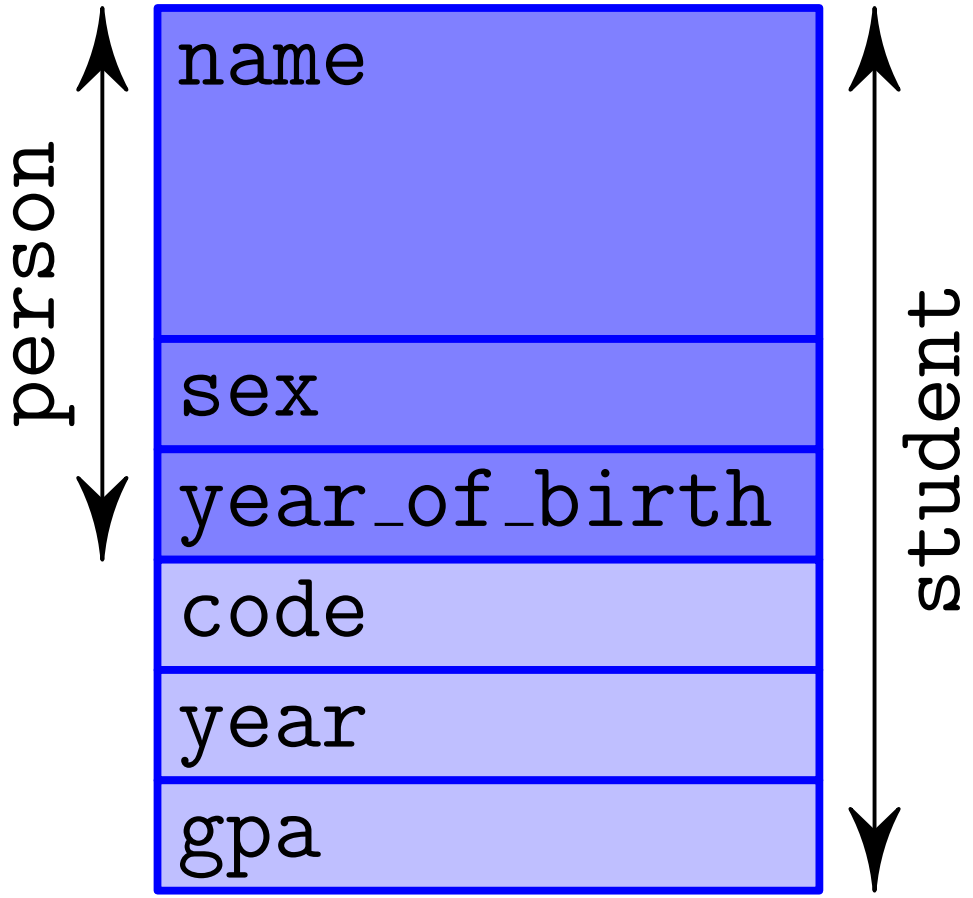

Рис. 2: Свойства унаследованной структуры данных

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.*** *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

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

Листинг 5: Второй пример полиморфизма адресов

Сам факт наследования одного класса от другого в некоторых случаях может рассматриваться как *частная деталь реализации* данного класса. В таких случаях необходимо *сокрытие этого фрагмента программы от остального кода*.

Два основных типа наследования:

- Открытое (public);
- Закрыте (private).

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

Листинг 6: Открытое и закрытое наследование

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).

Проблема

Для базового класса унаследованный класс ничем не отличается от текста всей остальной программы и **не должен иметь доступа к деталям реализации,** однако часто возникает необходимость в таких деталях интерфейса класса, которые должны быть доступны его потомкам.

Решение

Защищённый (`protected`) режим защиты.

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.

- **Public-наследование.** Не изменяет режима доступа к элементам базового класса из производного. При таком наследовании открытые (`public`) элементы базового класса останутся открытыми элементами в производном классе. Закрытые (`private`) элементы станут частью производного класса, но к ним можно будет обращаться только посредством открытых и защищенных методов базового класса.
- **Private-наследование.** Осуществляет изменение режима доступа к элементам базового класса: открытые и защищенные элементы базового класса становятся закрытыми элементами в производном классе.
- **Protected-наследование.** В этом случае открытые элементы базового класса становятся защищенными. Основное отличие `private` и `protected`-наследования проявляется при создании нового класса из производного.

```
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

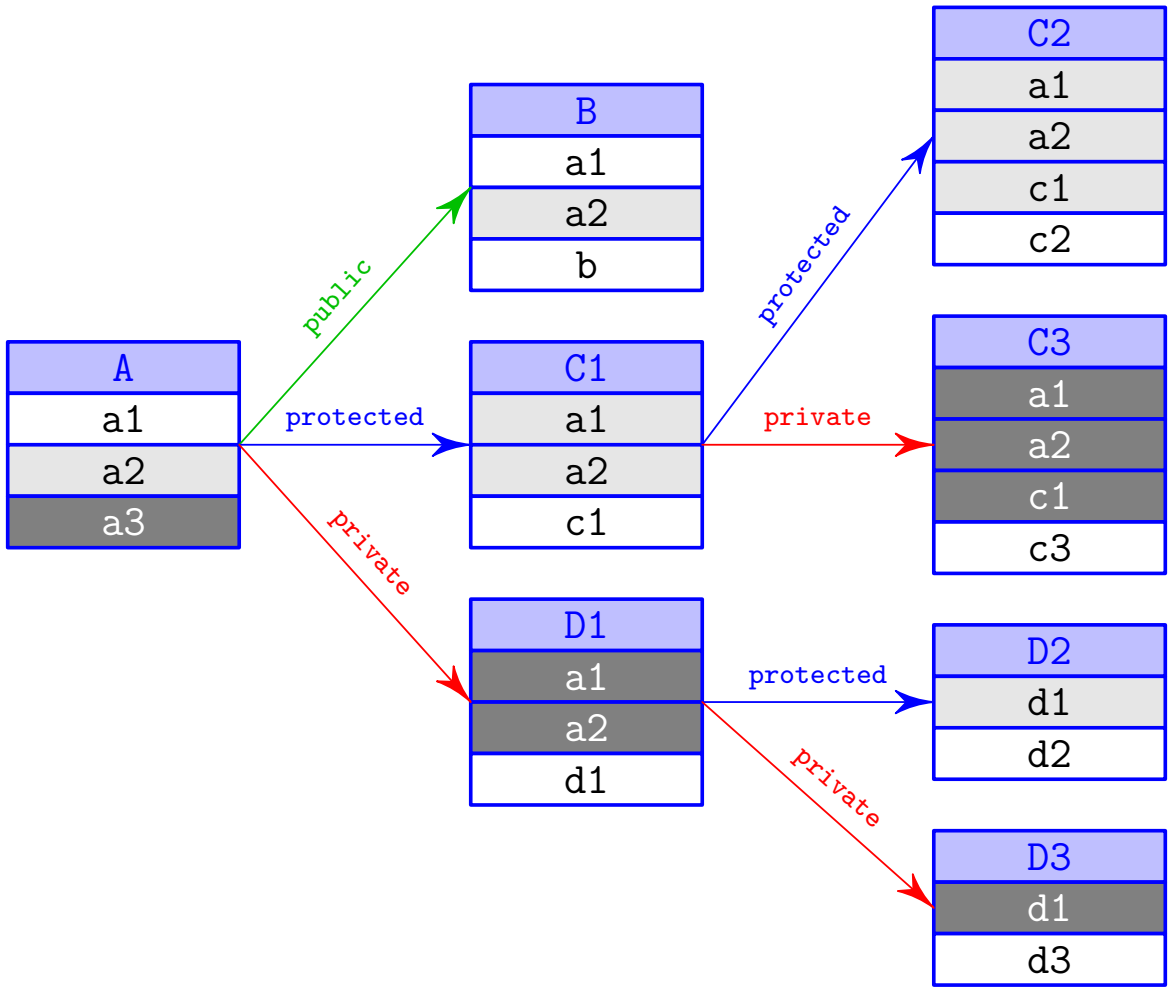

Рис. 3: Различные режимы наследования для трёх поколений классов

Если объекту-потомку доступен метод, введённый в его классе-предке, то такой метод можно вызвать для объекта-потомка точно так же, как и для объекта-предка.

В терминологии ООП это означает, что потомок умеет отвечать на все виды сообщений, предусмотренные для его предков.

Пользовательская семантика

Объект порождённого класса также является и объектом базового класса.

Реализаторская семантика

Объект порождённого класса содержит в себе объект базового класса в качестве своей части.

1. Конструкция объекта в роли предка;
2. Конструкция объекта в роли потомка;
3. Деструкция объекта в роли потомка;
4. Деструкция объекта в роли предка.

Пример 3

Конструирование предка

```
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 };
```

Листинг 8: Проблема

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

Листинг 9: Решение

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.

Определение 5

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

Пример 4

Рассмотрим задачу, связанную с компьютерной графикой.

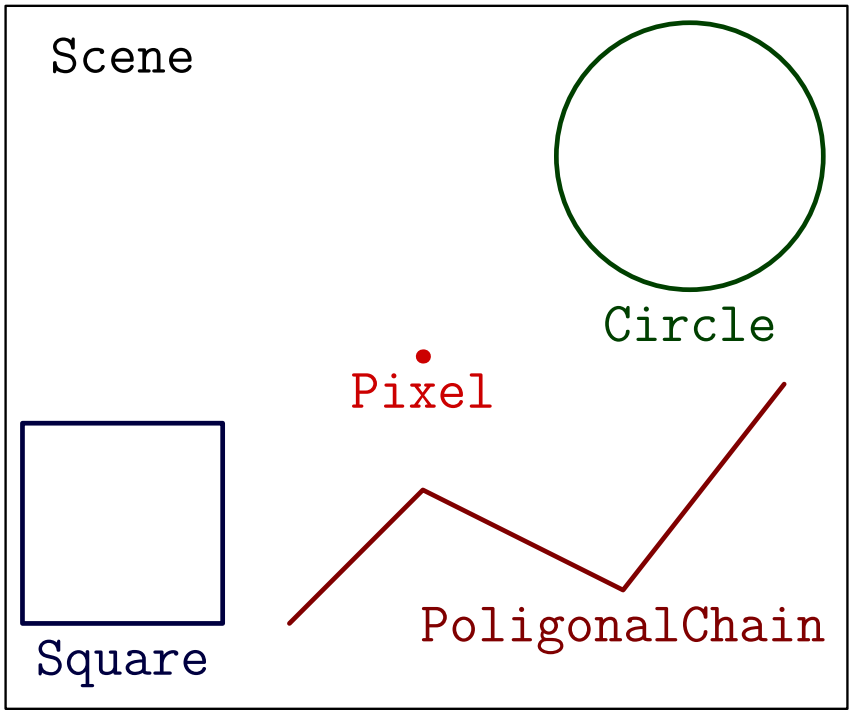

Рис. 4: Элементы графической сцены

Опишем класс, объекты которого будут представлять на сцене простейшие графические примитивы — пиксели.

Свойства класса `Pixel`

- две координаты (числа с плавающей точкой);
- цвет (целое число).

Основные действия с графическим объектом

- 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 10: Первый вариант заголовка класса Pixel

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

Листинг 11: Реализация метода Move

Виртуальные функции и динамический полиморфизм

Простейший пример – наследование класса `Circle` от класса `Pixel`

Виртуальные функции и динамический полиморфизм

Простейший пример – наследование класса Circle от класса Pixel

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

Виртуальные функции и динамический полиморфизм

Простейший пример – наследование класса Circle от класса Pixel

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

```
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 12: Заголовок класса Circle

Виртуальные функции и динамический полиморфизм

Простейший пример – наследование класса Circle от класса Pixel

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

```
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 12: Заголовок класса Circle

Почему в классе Circle не описан метод Move?

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`. Поэтому компилятор, естественно, вставит в машинный код функции `Pixel::Move` обычные вызовы функций `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`.

Определение 6

В теории ООП говорят, что ***виртуальным методом*** *задаётся реакция объекта класса на некоторый 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`.

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.

Замечание 4

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

```
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 13: Второй вариант заголовка класса Pixel

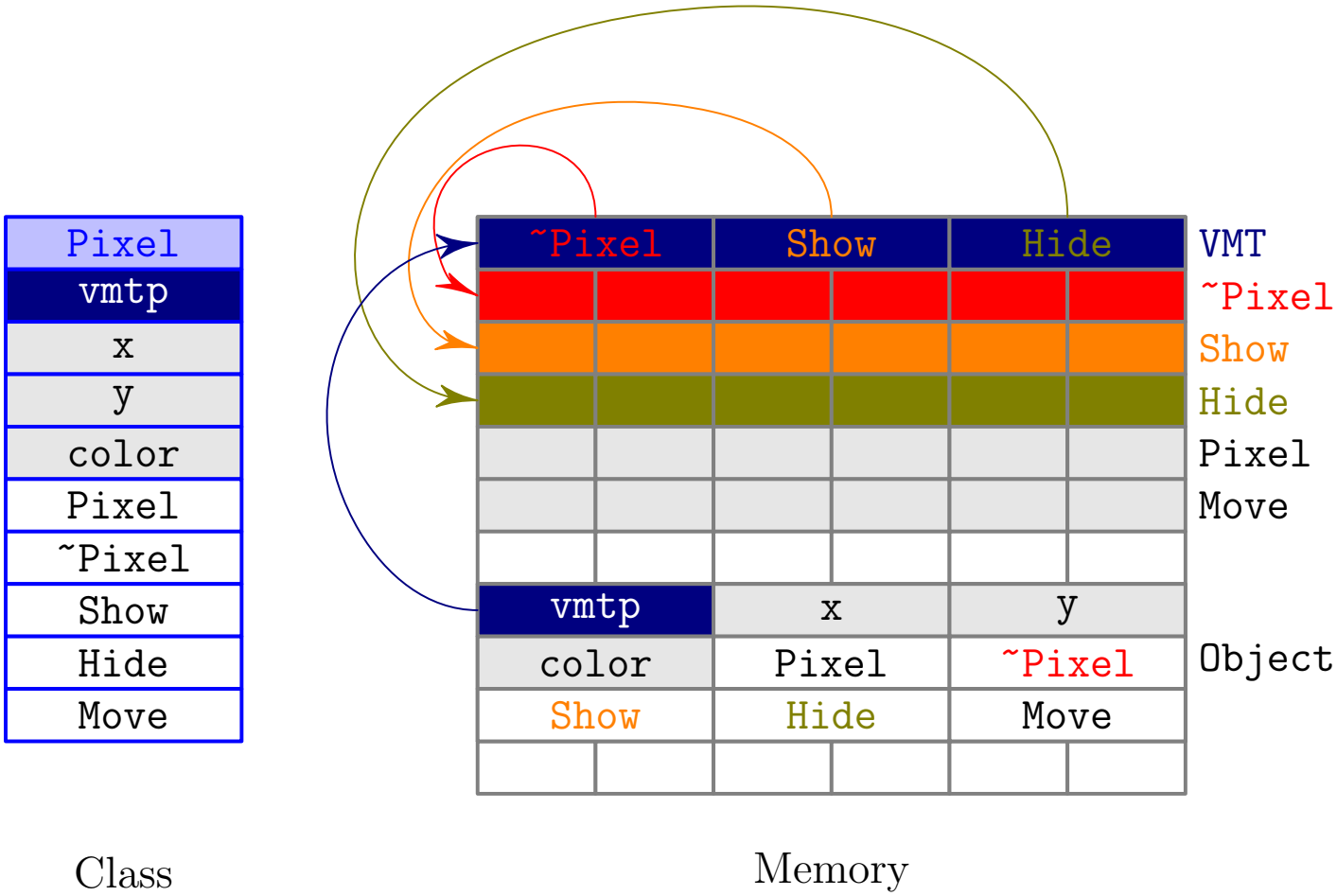

Рис. 5: Реализация идеи виртуальных методов на примере класса `Pixel`

```
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 14: 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.

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.

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

```
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!
```

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.*

Опишем класс для представления абстрактной геометрической фигуры `GraphObject` и унаследуем от него оба класса `Pixel` и `Circle`. Однако отметим один важный момент, а именно — 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`.

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.

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

Листинг 15: Синтаксис чисто виртуальной функции

Определение 7

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

Замечание 6

Компилятор не позволяет создавать объекты абстрактных классов.

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 методы 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 16: Заголовок абстрактного класса GraphObject

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

```
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 17: 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 18: 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.

Особенность вызова виртуальных функций в телах конструкторов и деструкторов связана с конструированием и последующим деструктированием в объекте поля `vmtp`.

Если в классе 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 имеется также для деструкторов — после завершения своего тела деструктор деинициализирует указатель на виртуальную таблицу.

Правило 2

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

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.

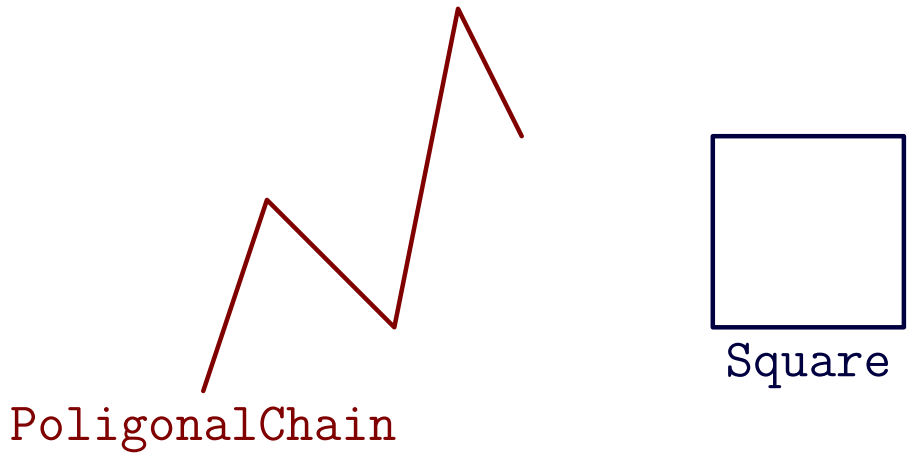

Рис. 6: Пример объектов подобного рода

Ломаную линию 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.

```
1  class PolygonalChain : public GraphObject {
2      struct Vertex {
3          double dx, dy;
4          Vertex *next;
5      };
6      Vertex *first;
7  public:
8      PolygonalChain(double x, double y, int color)
9          : GraphObject(x, y, color), first(0) {}
10     virtual ~PolygonalChain();
11     void AddVertex(double adx, double ady);
12     virtual void Show();
13     virtual void Hide();
14 };
```

Листинг 19: Заголовок класса PolygonalChain

```
1 void PolygonalChain::AddVertex(double ax, double ay) {
2     Vertex *tmp = new Vertex;
3     tmp->dx = ax;
4     tmp->dy = ay;
5     tmp->next = first;
6     first = tmp;
7 }
```

Листинг 20: Реализация функции AddVertex

```
1 PolygonalChain::~PolygonalChain() {
2     while(first) {
3         Vertex *tmp = first;
4         first = first->next;
5         delete tmp;
6     }
7 }
```

Листинг 21: Деструктор класса PolygonalChain

Как и ранее, методы Show и Hide считаются написанными.

Наследование ради конструктора

Класс Square — наследник класса PolygonalChain

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.

```
 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 22: Класс Square

Единственное отличие классов Square и PolygonalChain только в конструкторе.

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.

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
```

Правило 3

Деструктор любого класса, имеющего хотя бы одну виртуальную функцию, следует объявлять как виртуальный, не задумываясь о том, понадобится ли это в программе или нет. Многие компиляторы выдают предупреждение, если этого не сделать.

Пример 6

Продолжение Примера 4 — использование полиморфизма на графической сцене

Пусть создаётся графическая сцена, состоящая из графических объектов, представляемых классами — наследниками `GraphObject`, и на момент написания программы не определено, сколько и каких именно объектов будет в сцене. Допустим, что в какой-то момент становится известно, сколько объектов будет содержать сцена. В этой ситуации для хранения всей сцены можно использовать динамический массив указателей на объекты потомков `GraphObject`.

```
1 int scene_size;
2 GraphObject **scene;
3 // ...
4 scene = new GraphObject*[scene_size];
```

Благодаря полиморфизму адресов корректно следующее:

```
1 scene[i] = new Pixel(1, 1, 0x0000FF);
2 scene[j] = new Circle(5, 5, 1, 0xFF0000);
3 scene[k] = new Circle(-1, -1, 3, 0x00FF00);
```

Из-за наличия виртуального деструктора разрешено и это:

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

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.

- 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.

Приватные и защищённые деструкторы

Пример деструкции объекта с приватным деструктором

```
1 class Samurai {
2     ~Samurai() {}      // destructor is private
3 public:
4     // ...
5     void Harakiri() {
6         delete this;
7     }
8 };
```

```
(base) pavel@m-ThinkPad-T410 ~ $ ./harakiri Joe Max
Start
Joe, Healt: 3, Hit: 5
Max, Healt: 7, Hit: 1

Joe is tired and resting.

Max hits Joe!
Joe, Healt: 2, Hit: 5

Joe hits Max!
Max, Healt: 2, Hit: 1

Max hits Joe!
Joe, Healt: 1, Hit: 5

Joe hits Max!
Max is killed by self
(base) pavel@m-ThinkPad-T410 ~ $
```

Рис. 7: Забавный пример

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.

Пусть имеется класс `A` от которого унаследован класс `B`, и при этом в обоих классах есть поля или методы, названные одним и тем же идентификатором `x`.

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

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.

```
1 class A {
2     // ...
3 public:
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 23: Первый пример сокрытия имен при наследовании

Обычный вызвов метода f для объекта класса B не удастся:

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

Однако «скрыт» — не значит «недоступен»!

```
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!*

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.

```
 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 24: Второй пример сокрытия имен при наследовании

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 25: 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 26: 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.

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.

1. *Объект порождённого класса (подкласса) заведомо занимает не меньше (а как правило, — больше) памяти, чем объект базового класса (суперкласса).*
2. *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.

Два варианта изменения типа выражения

- Неявный (пример: вычисление выражения $3 + 2.0$);
- Явный (пример: изменение типа указателя).

Унарная операция преобразования типа

```
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.*

Общий синтаксис и пример использования

`KeyWord<NewType>(expression)`

`KeyWord` — это ключевое слово, задающее одну из операций:

- `static_cast`,
- `dynamic_cast`,
- `const_cast`,
- `reinterpret_cast`.

`NewType` — это имя нового типа.
`expression` — это само выражение, тип которого требуется изменить.

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

Листинг 27: Пример использования операции `static_cast`

```
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!
```

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?

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.

```
 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 28: Примеры использования static_cast

Операция `reinterpret_cast` позволяет произвести любое преобразование (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`.

Операции приведения типа

Операция dynamic_cast

Операция dynamic_cast предполагает проведение проверки **во время исполнения** программы и предназначена для преобразования адресов объектов в направлении, противоположном закону полиморфизма. В случае применения dynamic_cast, она проводит проверку и в случае, если преобразование некорректно (т. е. по заданному адресу в памяти не находится объект нужного типа), возвращает нулевой указатель (NULL).

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.

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++.

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

- `throw any_type` $\longrightarrow$ `catch(any_type &);`
- `throw any_type *` $\longrightarrow$ `catch(const any_type *);`
- `throw any_type &` $\longrightarrow$ `catch(const any_type &).`

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

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 в некоторой программе

- 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!*

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?