Наследование

Классы в C++ содержат не только данные, но и функции для работы с ними. Конструкторы и перегружаемые операторы дают возможность работать с классом примерно так же, как со встроенным типом данных, исключения предоставляют довольно логичное и единообразное средство для обработки ошибок. Еще одно важное свойство, присущее всем объектно-ориентированным языкам - наследование. Сама суть наследования проста - некий класс, допустим, child, может объявить себя наследником другого класса, скажем, parent. В этом случае child принято называть производным классом, а parent - базовым. При этом производный класс унаследует от своего базового класса все поля данных и почти все методы. Такой механизм весьма полезен сразу в нескольких смыслах:

Во-первых, он дает возможность расширять функциональность имеющихся классов без дублирования кода. Если понадобилось еще одно поле данных или новый метод, то не надо ни изменять уже существующий класс, ни копировать все его определение, создавая новый класс с другим именем. Достаточно определить класс-наследник уже имеющегося, добавив в него новые поля и функции.

Во-вторых, при таком подходе отсутствует риск разрушить уже имеющиеся программы, которые использовали базовый класс. Поскольку создается новый класс, никак не затрагивающий начинку уже имеющегося, то для таких программ ничего не меняется.

В-третьих, это позволяет работать со всеми наследниками одного базового класса одинаковым образом. Поскольку каждый класс-наследник несет в себе функциональность базового, то все они будут содержать поля и "отзываться" на методы этого базового класса. Или, как принято говорить, будут поддерживать интерфейс базового класса, точнее, любого из своих базовых классов. Такое свойство, когда объекты класса ведут себя по разному в различных условиях, называют полиморфизмом.

Рассмотрим на примерах, как наследование выглядит в программе на C++. В качестве иллюстрации для этого очень удобно работать с графическими изображениями. Для этого разработаем некую весьма ограниченную по возможностям, но полезную для понимания механизмов наследования, графическую библиотеку.

Для начала создадим класс, каждый объект которого представляет собой точку на экране.

```
class Point {
public:
  int x,y;
  Point(int _x, int _y) : x(_x), y(_y) {};
  void show() {
    // рисуем точку (x,y)
  }
  void hide() {
    // стираем точку (x,y)
  }
  void move(int new_x, new_y) {
    // перемещаем из (x,y) в (new_x, new_y)
    hide();
    x=new_x;
    y=new_y;
    show();
  }
};
```

Каждый объект такого класса - точка с координатами, которую можно "показать" (метод show), спрятать (метод hide) и задать ей новые координаты, "передвинуть" (метод move).
Так, например, в программе, можно нарисовать точку, а затем изменить ее положение на экране:

```
Point p(0,0);
p.show();
p.move(100,100);
```

В этом классе отсутствуют методы, которые позволяли бы узнать координаты точки. Правда, поля x и y размещены в public-секции класса и поэтому доступны напрямую, но этот недочет мы со временем тоже исправим. Добавим недостающие функции получения координат x и y.

```
class Point1 : public Point
{
public:
  Point1(int _x, int _y) : Point(_x,_y) {};

  int get_x() { return x; }
  int get_y() { return y; }
};
```

Разберемся в написанном коде

В первой строке class Point1 : public Point объявляется что класс Point1 является наследником класса Point. Причем перед классом Point стоит ключевое слово public. Вместо public можно было также указать protected или private. Эти ключевые слова влияют на то, на каком уровне доступа окажутся поля данных и методы базового класса в нашем новом, производном классе.

Затем, в самом определении класса, в секции public мы видим конструктор

```
Point1(int _x, int _y) : Point(_x,_y) {};
```

Обратим внимание на то, что в списке инициализации конструктора Point1 стоит имя базового класса, как если бы новый класс содержал базовый в качестве поля данных (по сути, так оно и есть, новый класс - это Point и еще что-то).

Конструктор для нового класса, пусть и несложный, пришлось написать, никакое наследование не помогло - в самом деле, конструктор должен уметь правильно создавать объект своего типа, так что никакие конструкторы других классов для этой цели не годятся.

Следом, также в общедоступной секции, стоят те две функции, ради которых мы новый класс и определен:

```
int get_x() { return x; }
int get_y() { return y; }
```

Замечательно то, что мы пользуемся полями x и y в новом классе, как своими собственными - они унаследованы из базового класса. Точно так же унаследованы и все три метода базового класса - show, hide и move. Так что код рисования и перемещения точки

```
Point1 p(0,0);
p.show();
p.move(100,100);
```

по прежнему будет работать (мы в нем заменили только тип с Point на Point1), но в дополнение к этому мы теперь можем, написав

```
int x = p.get_x();
int y = p.get_y();
```

получить координаты точки p.

Процесс можно продолжить, написав класс, производный уже от Point1.

```
class Point2 : public Point1
{
public:
  Point2(int _x, int _y) : Point1(_x,_y) {};
  void set_x(int new_x) { x = new_x; }
  void set_y(int new_y) { y = new_x; }
};
```

Естественно, при необходимости в новый класс можно добавлять не только функции, но и поля данных.

Следует обратить внимание на то, что последний класс является наследником класса Point1, а тот в свою очередь - наследник Point. В таких случаях говорят, что Point1 - непосредственный базовый класс для Point2 (direct base), а Point - косвенный базовый (indirect base).

В C++ у производного класса может быть несколько базовых - при этом такой объект будет наследовать признаки всех своих базовых классов. Например, если мы захотим объединить точки в список (например, для рисования фигур), то такие объекты вполне естественно наделить еще и свойствами элементов списка:

```
class LinkedPoint
  : public Point, public LinkedListItem
{
  ...
};
```

Определение базового класса LinkedListItem в данном случае осталось за кадром, но ясно, что в нем должны содержаться поля и методы для объединения элементов в связный список.

Уровни доступа к базовому классу

Разберемся, что означает ключевое слово public (а также и другие уровни защиты - protected, private) применительно к базовому классу. А заодно исправим недочет класса Point, убрав поля x и y из общедоступной секции.
Определение самого первого класса выглядело следующим образом

```
class Point {
public:
  int x,y;
  …  };
```

По канонам объектно-ориентированного языка делать общедоступными поля данных нехорошо. Однако, если мы попытаемся разместить x,y в личной секции

```
class Point {
private:
  int x,y;
public:
  …  };
```

а потом определить производный класс

```
class Point1 : public Point {
public:
  ...
  int get_x() { return x; }
  int get_y() { return y; }
};
```

то транслятор выдаст сообщение об ошибке - нет доступа к личным полям x и y.
Личные есть личные - никто кроме самого класса Point не имеет права их трогать. Если же мы хотим, чтобы поля были недоступны внешнему миру, но при этом производные классы все-таки могли ими пользоваться, в классе Point надо использовать другое ключевое слово - protected (защищенные).

```
class Point {
protected:
  int x,y;
public:
  ...
};
```

Теперь класс-наследник сможет работать с x и y напрямую.

Однако будут ли доступны эти поля следующему классу, производному не от Point, а от Point1? Это как раз и определяется тем, какое ключевое слово поставлено перед указанием базового класса.

Когда мы пишем

```
class Point1 : public Point
```

то уровни защиты базовых полей и методов в производном классе не меняются - protected-члены класса Point становятся protected-членами Point1, а public-члены так и остаются общедоступными.
Если бы мы написали

```
class Point1 : private Point
```

то все члены класса Point стали бы личными членами Point1. В частности, мы в программе могли бы пользоваться только функциями get_x, get_y, а функции show, hide, move стали бы недоступными.

Третий вариант

class Point1 : protected Point

усиливает защиту на одну ступеньку, превращая защищенные поля в личные, а общедоступные делая защищенными.
Какой именно уровень защиты приписать базовому классу, определяет, исходя из своих нужд, разработчик производного класса.

Одноименные поля в производном и базовых классах

Итак, в классе Point у нас были поля x, y, которые мы использовали во всех производных классах. А что бы случилось, если в производном классе мы бы определили свои поля с такими же именами?

```
class Point {
public:
  int x, y;
  ...
};

class OtherPoint : public Point {
public:
  int x, y;
  void set_x(int _x) { x=_x; }
};
```

В такой ситуации методы производного класса стали бы работать со своими полями, а не с полями базового класса. Однако при необходимости мы смогли бы добраться и до базовых полей. Вот как бы это выглядело в производном классе

```
class OtherPoint : public Point {
public:
  int x, y;
  void set_x(int _x) { x=_x; }
  void set_base_x(int _x) { Point::x=_x; }
};

OtherPoint p;

// меняем поле OtherPoint::x
p.x = 1;

// меняем базовое поле Point::x
p.Point::x = 1;
```

По умолчанию работа идет со "своими" полями, но, указав перед именем поля имя базового класса, становится возможным добраться и до унаследованных полей.

Виртуальные функции

Теперь рассмотрим, как ведут себя наследуемые функции.

Прежде всего, мы можем в производном классе переопределить унаследованный метод - написать другую функцию с такой же сигнатурой. При этом ситуация будет такая же, как с унаследованными полями - по умолчанию вызываться будет функция производного класса, но, указав явно имя базы, мы сможем добраться и до оригинальной унаследованной функции:

```
class Point { public:   …   void show() { ... };   … };

class OtherPoint : public Point {
public:
  ...
  void show() {
    // Замена для базовой show()
    ...
    // вызов базовой show()
    Point::show();
  }
};
```

Но это еще не самое интересное. Гораздо интереснее и полезнее разобраться с вопросом, как функции ссылаются друг на друга.

Вспомним полное определение класса Point (оно нам сейчас понадобится):

```
class Point {
public:
  int x,y;
  Point(int _x, int _y) : x(_x), y(_y) {};
  void show() {
    // рисуем точку (x,y)
  }
  void hide() {
    // стираем точку (x,y)
  }
  void move(int new_x, new_y) {
    // перемещаем из (x,y) в (new_x, new_y)
    hide();
    x=new_x;
    y=new_y;
    show();
  }
```

Теперь представим, что нужно написать класс, который рисует окружность. Что при этом можно унаследовать от класса Point? Координаты точки можем. А вот функции нам придется переписать - окружность и рисовать, и стирать надо по-другому:

```
class Circle : public Point
protected:
  int r;
public:  Circle(int _x, int _y, int _r)  : r(_r), Point(_x,_y)   {};

  void show() {
    // Вариант show для окружности
  }
  void hide() {
    // Вариант hide для окружности
  }
  void move(int new_x, int new_y) {
    hide();
    x = new_x;
    y = new_y;
    show();
  }
};
```

С функциями show и hide никуда не деться - рисуем не точку, окружность.

Обратим внимание на move - в ней точно такой же код, как и в функции Point::move. Однако нам пришлось переписать и ее. Если бы мы воспользовались наследуемой функцией, она бы, конечно, вызвала hide и show - но только не новые, а из базового класса. Соответствующие вызовы были вставлены в тело Point::move еще на этапе компиляции - это называется ранним связыванием (early binding).

Вопрос: нельзя ли все-таки сделать так, чтобы мы работали с унаследованной функцией move, но она при этом определяла в момент вызова, с каким именно классом работает, и вызывала правильные варианты функций? Оказывается, можно. Для этого надо всего-навсего сделать функции show и hide виртуальными, поставив в определении базового класса перед ними ключевое слово virtual:

```
class Point {
public:
  int x,y;
  Point(int _x, int _y) : x(_x), y(_y) {};

  virtual void show() {
    // рисуем точку (x,y)
  }

  virtual void hide() {
    // стираем точку (x,y)
  }

  void move(int new_x, new_y) {
    // перемещаем из (x,y) в (new_x, new_y)
    hide();
    x=new_x;
    y=new_y;
    show();
  }
};
```

Теперь мы можем не повторять код функции move в производном классе, а воспользоваться наследуемой:

```
class Circle : public Point
protected:
  int r;
public:
  Circle(int _x, int _y, int _r)
    : r(_r), Point(_x,_y)
  {};

  void show() {
    // Вариант show для окружности
  }
  void hide() {
    // Вариант hide для окружности
  }
};
```

Теперь, если где-нибудь в программе мы напишем

```
Circle c(10,10);
Point p(20,20);
c.move(50,50);
p.move(70,70);
```

то и в третьей, и в четвертой строке сработает функция Point::move. Однако благодаря ключевому слову virtual она в третьей строке вызовет Circle::show и Circle::hide, а в четвертой - Point::show и Point::hide. Это замечательное свойство - во время выполнения программы определять, функцию из какого именно класса надо использовать - называется поздним связыванием (late binding).

Благодаря виртуальным функциям объекты получают еще одно замечательное качество. Вы ведь можете работать не с самими объектами, а с указателями или ссылками на них. А по правилам языка ссылка или указатель на базовый тип совместима со ссылкой или указателем на производный. То есть, работая с указателями, можно написать, например, такой цикл:

```
Point *a[2];
Circle c;
Point p;
a[0] = &c;
a[1] = &p;

for (int i=0; i<2; i++)
  a[i]->show();
```

и не задумываться о том, на какой именно тип объекта указывает конкретный элемент массива - благодаря тому, что функция show объявлена виртуальной, для точки будет вызвана Point::show, а для окружности - Circle::show.

Это и есть *полиморфизм* - способность объекта вести себя по-разному в зависимости от того, как им пользуются. Если с ним работают через ссылку или указатель на базовый класс, то он и ведет себя как базовый (разумеется, объект Circle рисовать будет окружность, а не точку, но по *интерфейсу*, то есть, по набору доступных полей и методов, это будет именно объект базового класса.

Абстрактные классы. Чистые виртуальные функции.

Сейчас наша сеть наследования выглядит так - базовый класс Point, его производный класс - Circle. Можно расширить эту сеть - например, написать еще производных от Point классов - Rectangle, Polygon, и так далее. Однако такая схема наследования не совсем логична. Point - это уже некий реальный объект, который помимо свойств, характерных для всех фигур (в нашем утрированном примере это координаты x,y и метод move) содержит еще и специфические только для своего типа детали - методы show и hide. Гораздо логичнее выделить все общее в отдельный класс, например, в класс Figure, а специфичные для конкретной фигуры детали определять непосредственно в производных классах.

С координатами при таком подходе проблем нет. Но нам надо внести в базовый класс метод move, который вызывает show и hide. А show и hide уже относятся к тем самым специфическим особенностям, которые мы хотим удалить из базового класса. Решить эту проблему в C++ помогают чистые виртуальные функции (pure virtual functions). Вот как могло бы выглядеть соответствующее определение базового класса:

```
class Figure {
protected:
  int x, y;
public:
  Figure(int _x, int _y) : x(_x), y(_y) {};

  void move(int new_x, int new_y) {
    hide();
    x = new_x;
    y = new_y;
    show();
  };

  virtual void show() = 0;
  virtual void hide() = 0;
};
```

Обратим внимание, как записаны в определении класса show и hide:

```
virtual void show() = 0;
virtual void hide() = 0;
```

Это и есть чистые виртуальные функции. Класс, в котором есть хотя бы одна чистая виртуальная функция, непригоден напрямую для использования - транслятор просто не позволит вам создавать объекты типа Figure. Однако он вполне подходит (как и задумано) для базового класса. Кроме того, указатель (или ссылка) на Figure может содержать адрес или ссылаться на объект любого производного от него класса. Такие классы с чистыми виртуальными функциями называют абстрактными.

Теперь мы можем более логично построить нашу сеть наследования, сделав класс Point наравне с другими производным от Figure:

```
class Point: public Figure {
public:
  Point(int _x, int _y) : Figure(_x,_y) {};
  void show() { /* show для точки */ };
  void hide() { /* hide для точки */ };
};

class Circle: public Figure {
protected:
  int r;
public:
  Circle(int _x, int _y, int _r)
    : Figure(_x,_y), r(_r)
  {};
  void show() { /* show для окружности */ };
  void hide() { /* hide для окружности */ };
};

class Section: public Figure {
protected:
  int len;
public:
  Section(int _x, int _y, int _len)
    : Figure(_x,_y), len(_len)
  {};
  void show() { /* show для отрезка */ };
  void hide() { /* hide для отрезка */ };
};
```

Соответственно, если нас не интересуют специфические детали конкретных объектов, мы можем работать с ними по интерфейсу Figure:

```
Figure *ptr[3];

Point p(0,0);
Circle c(10,10, 20);
Section s(20,20,2);

ptr[0] = &p;
ptr[1] = &c;
ptr[2] = &s;

// Прячем все фигуры в массиве,
// независимо от типа
for (int i=0; i<3; i++)
  ptr[i]->hide();
```

Следует обратить внимание на цикл в последнем примере. Это снова иллюстрация *полиморфизма* - очень полезного свойства имеющих общую базу объектов. Все они могут работать по интерфейсу любого своего базового класса.

Подобный стиль работы широко применяется в объектно-ориентированном программировании. Например, в C++ стиле ввода-вывода можно считывать данные с терминала, из файла, даже из массива - и при этом будут использоваться одними и теми же функциями.

В C++ у производного класса может быть несколько базовых классов. Если бы мы написали, например

```
class Circle
  : public Point, public ListItem
{ ...; }
```

то смогли бы работать с указателями/ссылками на Circle и как с точками, и как с элементами списка.
Полиморфизм - мощное средство, широко применяемое в объектно-ориентированном программировании. Однако вам следует запомнить - чтобы получить полиморфное поведение, необходимо работать не с самим объектом, а с указателем или ссылкой на него.

Виртуальные конструкторы

Сначала, чтобы не возникло путаницы, следует сделать оговорку - виртуальных конструкторов не существует, они запрещены правилами языка. То, о чем пойдет речь сейчас - как с помощью обычных функций добиваться того же результата, какой можно было бы ожидать от виртуальных конструкторов. Но прежде всего попробуем разобраться, для чего нужны виртуальные конструкторы.

В чистом виде виртуальный конструктор - вещь нелепая. Раз уж мы создаем объект, то не абы какой, а какого-то конкретного типа. Другое дело, что мы можем не знать, какого именно - просто у нас есть (какой-то) объект, и мы хотим сделать точно такой же. Или даже не просто сделать, но и скопировать в него начинку старого. Первая операция похожа на конструктор по умолчанию, вторая - на копирующий конструктор. Но, поскольку точный тип образца нам неизвестен, приходится проделывать это с помощью обычных виртуальных функций.

Так что, когда говорят о виртуальных конструкторах, всегда имеют в виду задачу создания объектов по существующему образцу. Ведь у каждого образцового объекта есть какой-то вполне определенный тип, даже если мы и не знаем, какой именно.

Подобного рода задачи возникают при необходимости продублировать большой набор объектов, хранящийся, например, в виде дерева. Одно из применений - в каком-нибудь графическом редакторе мы создавали чертеж из отрезков, фигур, строк и так далее. Все это - объекты разного типа, но они вполне могут быть наследниками одного базового класса, скажем, TElement, и все изображение хранится в виде связного списка указателей на этот базовый тип. Затем в какой-то момент мы хотим скопировать чертеж (например, для вывода на печать). Но про каждый из объектов мы знаем только то, что он является наследником TElement.

Таким образом, у нас возникает необходимость скопировать весь список с сохранением типов объектов, но самих типов мы не знаем.

Функции, которые помогают решить эту задачу, и называют виртуальными конструкторами.

Различают два вида виртуальных конструкторов - по умолчанию и копирующий. Означают эти термины то же, что и в случае обычных конструкторов. Конструктор по умолчанию создает "пустой" объект (такого же, как и у образца, типа). Копирующий конструктор еще и копирует во вновь созданный объект содержимое образца.

Делается это следующим образом: - в базовом классе определяются две виртуальные (в примере ниже это makeobject и copyobject), которые создают объект (copyobject еще и копирует в новый объект содержимое образца) и возвращают указатель на

```
#include <string>
#include <iostream>
using namespace std;

class A {
protected:
  int val;
public:
  A(int v=0) : val(v) {};
  A(const A& src) : val(src.val) {};

  // makeobject() is a virtual default ctor
  virtual A* makeobject() {
    A* ptr=new A;
    return ptr;
  }

  // copyobject() is a virtual copy ctor
  virtual A* copyobject() {
    A* ptr=new A(*this);
  }

  virtual void show() {
    cout << "Object of type A, val=" << val << endl;
  }

};  // End of class A definition
```

Затем, при разработке производного класса вы замещаете эти две функции другими - создающими и копирующими объект вашего нового типа, но по прежнему возвращающими указатель на базовый класс:

```
class B : public A {
  string s;
public:
  B(const char *str) : s(str) {};

  // Copy ctor
  B(const B& src) : A(src), s(src.s) {};

  // Default ctor
  B() : s("") {};

  // virtual default ctor for class B
  A* makeobject() {
    B* ptr=new B;
    return ptr;
  }

  // virtual copy ctor for class B.
  A* copyobject() {
    B* ptr=new B(*this);
    return ptr;
  }

  virtual void show() {
    cout << "Object of type B, s=" << s << endl;
  }

};  // end of class B definition
```

Проверить, как это работает, поможет небольшая программа

```
main() {

        A a(1);
        B b("Hello");

        A* psrc[2] = {&a, &b};
        A* pmake[2];
        A* pcopy[2];

        for (int i=0; i<2; i++) {
          pmake[i] = psrc[i]->makeobject();
          pcopy[i] = psrc[i]->copyobject();

          psrc[i] ->show();
          pmake[i]->show();
          pcopy[i]->show();
        }

        return 0;
      }
```

```
Object of type A, val=1
      Object of type A, val=0
      Object of type A, val=1
      Object of type B, s=Hello
      Object of type B, s=
      Object of type B, s=Hello
```

Как видно, благодаря полиморфизму, вызвав функции makeobject или copyobject через указатель на базовый класс, даже не зная типа объекта-образца, можно создать объекты точно такого же типа.

dynamic_cast и RTTI

Даже при правильном проектировании и с учетом всех возможностей полиморфизма иногда возникает потребность узнать конкретный тип какого-то объекта. Для этой цели предназначены информация времени исполнения о типах (*RTTI*, Run Time Type Information) и *dynamic_cast*, который позволяет безопасно преобразовывать (во время выполнения программы) совместимые по сути наследования типы.

Рассмотрим пример. Немного раньше был приведен в качестве примера чертеж, все элементы которого - наследники базового класса TElement.

Если они к тому же наследники класса Point, то мы можем убрать чертеж с экрана в одном цикле. Правда, для этого нам придется явно использовать приведение типа:

```
TElement *drawing[100];
...
for (int i=0; i<100; i++) {
  Point *p = (Point*)drawing[i];
  p->hide();
}
```

Проблема этого кода в том, что подобное приведение типа сработает в любом случае - даже если реальный объект не имеет к Point никакого отношения. А ведь в чертеже могут храниться не только объекты-фигуры, но и какая-нибудь служебная информация - например, объекты string с датой создания или фамилией автора. А поскольку у string нет метода hide(), показанный выше код приведет к ошибке при выполнении программы.

Решить эту проблему позволяет специальный оператор *dynamic_cast*, который позволяет безопасно приводить один тип к другому. Выглядит он следующим образом:

```
Type1* p1;
Type2* p2 = dynamic_cast<Type2*>(p1);  или

Type1& ref1;
Type2& ref2 = dynamic_cast<Type2&>(ref1);
```

Первый вариант приводит указатель типа Type1* к типу Type2*, второй проделывает то же самое со ссылками.

Почему же этот вариант безопасен? Дело в том, что если с его помощью попытаться преобразовать один тип в другой, несовместимый с первым, dynamic_cast вернет 0 (нулевой указатель). Так что теперь мы могли бы убрать с экрана чертеж без риска "сломать" программу:

```
TElement *drawing[100];
...
for (int i=0; i<100; i++) {
  Point *p = dynamic_cast<Point*>drawing[i];
  if (p != 0)
    p->hide();
}
```

Естественно, проверить на 0 можно только указатель, так что при работе со ссылками проверка выполняется другим способом - при попытке приведения между несовместимыми типами dynamic_cast возбуждает исключение *bad_cast*. Соответственно, код нашего цикла выглядел бы примерно так:

```
TElement *drawing[100];
...
for (int i=0; i<100; i++) {

  try {
    Point& p = dynamic_cast<Point&> *drawing[i];
    p->hide();
  }
  catch (bad_cast) {};

}
```

Следует подчеркнуть, что dynamic_cast не в состоянии помочь, если требуется узнать точный тип объекта. Он лишь сообщает о том, можно ли, преобразовав указатель или ссылку к другому типу, безопасно использовать объект в новом качестве. Например, он позволит привести Circle* к Point*. Позволит и обратное преобразование, если мы имеем дело с объектом типа Circle. Однако даже в последнем случае так и не удастся узнать, действительно ли имеется объект Circle, или у него другой, производный уже от Circle тип.

Как быть, если нужно точно узнать тип своего объекта? Такая потребность все-таки иногда возникает даже при правильном проектировании программ. Для этого в C++ есть другое средство - typeid. Вот как выглядит программа, использующая этот оператор для вывода на печать имени используемого типа:

```
#include <typeinfo>
#include <iostream>
using namespace std;

class A { public: virtual void l() {};} a;
class B : public A {} b;
class C : public B {} c;

void f(A& ref) {
  cout << typeid(ref).name() << endl;
}

main() {
  f(a);
  f(b);
  f(c);
}
```

Такая программа напечатает следующее:

1A
1B
1C

В базовом классе добавлена "пустая" виртуальная функция. Дело в том, что typeid "предпочитает" полиморфные классы, а самый простой способ сделать класс полиморфным - добавить в него виртуальную функцию.

использование результата typeid напоминает использование объекта класса. Так и есть на самом деле - typeid возвращает объект типа type_info, содержащийся в каждом полиморфном классе и хранящий информацию о его типе. А name() - метод класса type_info, который возвращает текстовую строку, уникальную для класса (как видно из вывода программы, она включает в себя имя класса).
Метод name() - не единственный в классе type_info. Два других полезных метода - это операторы != и ==, позволяющие проверять, совпадает ли тип нашего объекта с ожидаемым. Интересуй нас объекты именно типа A, мы могли бы изменить код функции f в примере выше следующим образом:

```
void f(A& ref) {
  cout << typeid(ref).name() << endl;
  if (typeid(ref)==typeid(A))
    cout << "match" << endl;
  else
    cout << "mismatch" << endl;
}
```

и при этом программа напечатала бы

```
1A
match
1B
mismatch
1C
mismatch
```

Есть в type_info и другие методы, но наиболее полезны именно эти три, name, == и !=.

На ошибки (например, при попытке определить тип нулевого указателя), оператор typeid реагирует, возбуждая исключение bad_typeid.