9. Полиморфизм на уровне объектных переменных

Как известно, объектные переменные являются скрытыми указателями (ссылками) на размещаемые в памяти объекты. Классический механизм указателей в обычных языках предполагает строгое соответствие типов указателей и адресуемых данных: переменные-указатели могут адресовать только данные тех типов, с которыми они были связаны при объявлении.

В объектных языках это правило строгого соответствия типов немного ослаблено: при определенных условиях объектные переменные могут адресовать (ссылаться на) не только свои «родные» объекты, но и **объекты некоторых других классов.** Более конкретно:

Пусть объектная переменная связана при объявлении с некоторым классом. Тогда она может использоваться для доступа не только к объектам этого класса, но и к объектам <u>любых</u> классов, <u>производных</u> от данного.

Пусть, например, для иерархии графических фигур объявлены следующие объектные переменные:

Fig : TFigure; // хоть класс фигур и является абстрактным,

 // разрешается объявлять переменную такого классового типа,

// нельзя лишь использовать ее вместе с конструктором

Circ : TCircle; // доступ к объектам-окружностям

Ellipse : TEllipse; // доступ к объектам-эллипсам

Rect : TRect; // доступ к объектам-прямоугольникам

Пусть также с помощью конструкторов созданы соответствующие объекты (окружность, эллипс, прямоугольник).

Тогда **разрешается** делать следующие присваивания:

Fig := Circ;   // переменная Fig адресует окружность

Fig := Ellipse; // переменная Fig адресует эллипс

Fig := Rect;   // переменная Fig адресует прямоугольник

Circ := Ellipse; // переменная Circ   адресует эллипс

Однако **недопустимыми** будут следующие присваивания:

Circ := Fig; Rect := Fig; Circ := Rect; Ellipse := Circ;

Отсюда следует, что механизм полиморфности указателей работает только в рамках **единой иерархии** классов и только **вниз** по иерархии от заявленного базового класса.

А что будет, если объявить переменную классового типа Object/TObject? Учитывая **особое** положение этого класса как **корневого** для всей иерархии стандартных и нестандартных классов, получим, что такая переменная сможет адресовать объекты **абсолютно любых классов!**

Очень мощным использованием полиморфных объектных переменных является возможность **объединения** их в **единую структуру.** Это позволяет собирать в один массив или список и циклически обрабатывать **объекты разных классов.**

Например, можно создать массив указателей на любые графические объекты в рамках единой иерархии графических фигур с корневым классом TFigure/Figure:

var   Figs : **array** [1..N] **of** TFigure;

Figure [ ]   Figs = **new** Figure [N];

Пример заполнения массива ссылками на конкретные фигуры:

Figs[1] := **TCircle**.Create(…);

Figs[2] := **TRect**.Create(…);

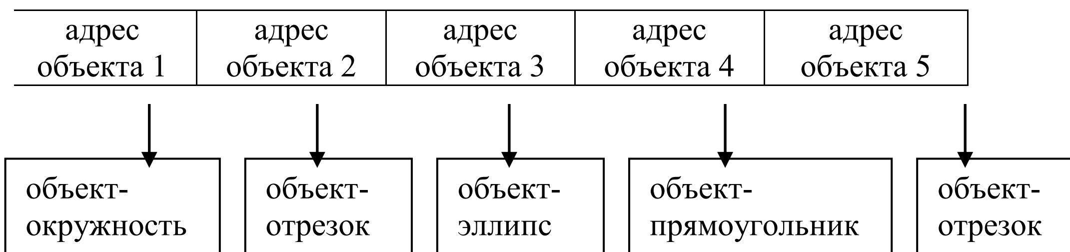
. . . .

Figs[0] = new **Circle**(…);

Figs[1] = new **Rect**(…);

. . . .

Схематичное представление такого массива :



Подобный массив очень удобно можно обрабатывать с помощью циклов. Например — показать все фигуры:

for i := 1 **to** count **do** Figs[I].Show;

for (int i=0; i < count; i++) { Figs[i].Show( ); }

Интересно, что в этих примерах используются **оба** проявления принципа полиморфизма – как полиморфность **объектных переменных**, так и **виртуальные переопределяемые** методы: цикл идет по массиву, для каждого элемента происходит обращение к соответствующему объекту, для которого с помощью его таблицы виртуальных методов определяется **необходимая версия** виртуального метода Show.

Крайним случаем массива полиморфных указателей является массив, в который можно собрать указатели на объекты **абсолютно любых** классов. Для этого достаточно объявить массив с типом Object/TObject. На практике применение таких "сверхуниверсальных" массивов должно быть оправданным и обоснованным.

Другим важным применением полиморфных объектных переменных является использование их в качестве **параметров методов.** Например, объявим метод с формальным параметром классового типа фигур:

procedure SomeMet (aFigs : **TFigure**);

void SomeMet (**Figure** aFigs);

В качестве **фактического** значения при вызове этого метода можно передавать указатель на объект **любого дочернего** графического класса:

SomeMet (Circle); // выполнение кода метода для окружности

SomeMet (Rect); // выполнение кода метода для прямоугольника

Аналогично "сверхуниверсальном" массиву можно объявить метод со “сверхуниверсальным” параметром:

```
procedure  SuperMet (aObj : TObject);
void   SuperMet (Object   aObj);
```

Такой метод может принимать в качестве фактического значения ссылку на объект **любого** класса!

К сожалению, при использовании полиморфных объектных переменных часто возникает ряд **проблем.** Поскольку полиморфный указатель может **динамически** менять свой тип, возникает необходимость ответа на вопрос:

Объект какого класса адресует полиморфная объектная переменная в данный момент работы приложения?

Например, для массива указателей на графические фигуры при использовании конструкций типа Figs[i] возникает **неопределенность**: какой графический объект адресуется этим элементом?

Для ответа на подобные вопросы используются специальные механизмы, которые **для каждого объекта в каждый момент времени работы программы позволяют получать информацию о его текущем классе.** Эта информация оформляется в виде таблиц и хранится в оперативной памяти в специальном внутреннем формате. Подобные таблицы часто называют таблицами **метаданных**. Метаданные – это данные о данных, т.е. данные, которые описывают структуру других данных, в нашем случае – структуру классов.

Несмотря на то, что конкретный состав метаданных различен для разных языков, можно указать наиболее часто используемые компоненты:

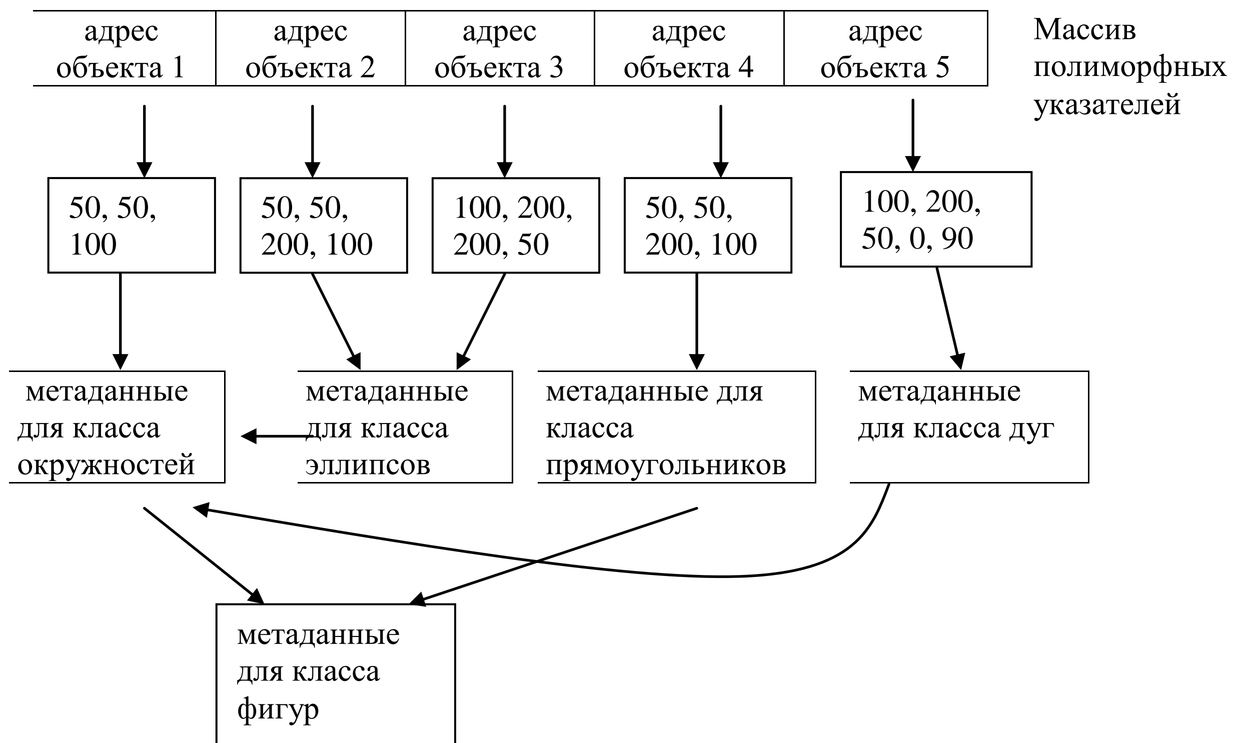
- имя класса
- указатель на таблицу метаданных родительского класса (тем самым, переходя последовательно к родительским классам, можно для любого класса восстановить всю цепочку наследования)
- байтовый размер объекта

* список всех полей данных с их атрибутами
* список всех методов с их атрибутами (открытый, защищенный, закрытый, абстрактный, виртуальный и т.д.)

Для доступа к метаданным можно использовать специальные методы, а сам процесс получения этой информации во время работы программы обозначают термином **reflection** (**рефлексия**, **отражение**).

Все объекты **одного и того же** класса связаны со **своей** таблицей метаданных, для чего в состав каждого объекта включается скрытое **адресное** поле.

Например, для массива указателей на графические фигуры связь соответствующих структур (сам массив, адресуемые объекты и соответствующие таблицы метаданных) можно схематично представить следующим образом:



Из этой схемы видно, что ни элемент массива указателей, ни сам объект **не определяют** классовый тип данных, этот тип определяется таблицей

метаданных. Например, второй элемент в массиве указателей на графические объекты (как это видно из рисунка) адресует набор из четырех целых чисел, которые должны быть интерпретированы как параметры объекта-эллипса, тогда как другая четверка чисел, адресуемая четвертым элементом массива определяет уже другой объект, а именно – прямоугольник.

С другой стороны, в любой момент можно изменить классовый тип любого элемента массива указателей – достаточно присвоить ему адрес другого объекта.

Теперь можно вернуться к ответу на вопрос о проверке динамического типа полиморфных объектных переменных. Для этих целей в объектных языках имеются специальные проверочные операторы:

- оператор **is** в языках C# и Delphi / FreePascal
- оператор **instanceof** в языке Java

Оператор **is** является логическим и поэтому чаще всего используется в условном операторе следующим образом:

if (объект **is** класс) **then** . . .

Оператор **is** возвращает истину, если объектная переменная слева от него адресует в данный момент объект класса справа от него ИЛИ объекты <u>**производного**</u> от него класса. Можно сказать, что оператор выполняет проверку типа объектной переменной «с точностью до подиерархии», т.е. фактически проверяется «попадание» объектной переменной в заданную подиерархию, корнем которой является указанный в операторе класс.

Например, для иерархии графических фигур условный оператор вида

if (Figs[i] **is** TCircle) **then** …

возвращает истинное значение, если полиморфная переменная Figs[i] в данный момент адресует окружность, эллипс, дугу или какой-то другой объект дочернего для окружности класса. Если же Figs[i] адресует что-то «не

округлое» (например, отрезки или прямоугольники), оператор **is** вернет ложное значение.

Этот прием можно использовать для **выборочной** обработки объектов некоторой подиерархии. Например, для массива полиморфных указателей на графические объекты можно организовать перемещение только криволинейных объектов, т.е. окружностей, эллипсов и дуг:

```
for i := 1 to N do
    if (Figs[i] is TCircle) then Figs[i].MoveTo (dx, dy);
```

```
for (int i = 0; i < N; i++)
    if (Figs[i] is Circle) Figs[i].MoveTo (. . .);
```

Аналогичная проверочная конструкция в языке Java выглядит так:

```
if (Figs[i] instanceof Circle) …..
```

При использовании полиморфных объектных переменных возникает еще один любопытный вопрос: если объектная переменная адресует объект «**чужого»** класса, то можно ли ее использовать для вызова методов, которых нет в ее «**родном**» классе, но которые есть в текущем динамическом классе?

Например, можно ли с помощью полиморфного указателя Figs[i] вызвать **специфический** метод класса окружностей, такой как “Изменение радиуса” (ясно, что в родном классе фигур ничего подобного нет и быть не может)?

Для положительного ответа на этот вопрос **необходимо**:

1. Убедиться (например, с помощью оператора **is**), что **текущее** значение указателя является **правильным** с точки зрения вызова метода (например, Figs [i] адресует окружность)

2. Выполнить специальную операцию – **приведение типов**, т.е. привести общий родительский указатель к **конкретному дочернему** типу

В языках C# и Delphi/FreePascal для явного приведения типов используется специальный оператор **as**, который включается в следующую синтаксическую конструкцию:

(объект **as** класс),

в результате чего объектная переменная приводится к типу указанного справа класса. А вот после этого уже можно вызывать **уникальные** методы того класса, к которому приведена данная объектная переменная, используя обычный (т.е. через точку) синтаксис вызова метода:

(объект **as** класс).SomeChildMethod ( );

Возвращаясь опять к примеру с массивом указателей на графические объекты, можно поставить следующую задачу: пройти по всем элементам массива, проверить для каждого текущий классовый тип и для всех криволинейных объектов выполнить изменение радиуса, а для всех прямоугольных объектов – операцию поворота. Вот эта конструкция:

if (Figs[i] **is** TCircle) **then** (Figs[i] **as** TCircle).ChangeRad (…)
    else if (Figs[i] **is** TRect) **then** (Figs[i] **as** TRect).Rotate (…);

Здесь конструкция (Figs[i] **as** TCircle) выполняет явное приведение типа родительского указателя к классу TCircle, после чего с помощью этого указателя выполняется обращение к методу изменения радиуса класса окружностей, тогда как аналогичная конструкция (Figs[i] **as** TRect) выполняет приведение уже к другому классу, а именно – к классу прямоугольников.

Аналогичная конструкция в языке Java выглядит так:

if (Figs[i] **instanceof** Circle) (Circle (Figs[i])).ChangeRad (…);

Для более глубокого проникновения к метаданным можно использовать методы специальных классов, входящих в состав стандартных библиотек. В языке Java такой класс носит простое и скромное имя **Class,** а в языке C# - имя **Type**. Объекты этих классов создаются автоматически при создании первого объекта каждого используемого в программе класса. Все

последующие объекты будут связаны с уже созданным объектом класса **Class** или **Type.** Можно сказать, что доступ к метаданным осуществляется на объектном уровне.

В языке Java для доступа к объектам класса **Class** можно либо использовать встроенное поле с именем **class,** которым автоматически снабжаются объекты всех классов, либо метод **getClass**( ), который объявлен в корневом классе **Object** и наследуется всеми его потомками. В классе **Class** реализовано несколько методов, наиболее полезными из которых являются:

- **forName**(String name) – возвращает указатель на объект класса **Class** для класса с именем name; метод является статическим (классовым) и поэтому может использоваться **БЕЗ** создания базового объекта;
- **getName**( ) – возвращает имя класса для вызывающего объекта;
- **getSuperclass**( ) – возвращает указатель на родителя вызывающего объекта;

Кроме того, предусмотрены методы, которые формируют списки всех полей, методов, конструкторов и интерфейсов класса.

Аналогичные функции в языке C# обеспечиваются стандартным классом **Type.** Он включает целый ряд свойств и методов, таких как:

- GetFields ( ) – получение списка полей указанного класса
- GetMethods ( ) – получение списка методов класса
- GetConstructors ( ) – получение списка конструкторов
- GetEvents ( ) – получение списка событий
- GetProperties ( ) – получение списка свойств

Для использования всех этих методов прежде всего надо получить объект класса **Type** для интересующего нас класса. Для этого можно использовать специальный оператор **typeof** (ИмяКласса). Например, если нам нужна информация по классу с именем MyClass, то объект-носитель этой информации инициируется следующим образом:

Type MyTypeObject = **typeof** (MyClass);

После этого конструкция MyTypeObject.Name позволит получить имя класса, а конструкция MyTypeObject.GetMethods ( ) – список всех его методов.

Все это позволяет динамически во время выполнения программы получать всю информацию о любом используемом объекте программы. Важность этого механизма определяется тем, что он активно используется при компонентной разработке приложений в рамках визуальных инструментов быстрого создания программ.