

Реализация полиморфизма. Полиморфные переменные

- Понятие полиморфизма
- Формы полиморфизма
- Преимущества использования полиморфизма
- Полиморфные переменные
- Проблема связывания сообщений и методов. Виртуальные методы
- Проблема обращения полиморфизма. RTTI

Понятие полиморфизма

- Полиморфизм в языке программирования означает **многозначность** переменных и функций

Формы полиморфизма

- **Полиморфной** называется такая **переменная**, которая может хранить в себе значения различных типов данных
- **Полиморфной функцией** является такая функция, которая может вызываться с аргументами различного типа, а фактический выполняемый код зависит от типа аргументов

Преимущества использования полиморфизма

- Полиморфизм позволяет записывать алгоритмы лишь однажды и затем повторно их использовать для **различных типов** данных, которые, возможно, еще не существуют (**обобщенные** действия или **алгоритмы**)
- Полиморфизм **сужает концептуальное пространство**, т.е. уменьшает количество информации, которое необходимо помнить программисту

Пример обобщенного алгоритма

- Пусть имеется класс `Enumerable`, в котором объявлены операции отношения `>`, `<`, `>=`, `<=`, `==` и `!=`, и имеется свободная функция `sort()`, которая упорядочивает элементы массива типа `Enumerable` по возрастанию
- Тогда всем классам, производным от класса `Enumerable`, будет доступна операция упорядочивания массива по возрастанию

Пример обобщенного алгоритма

```
// Упорядочивание массива методом пузырька
void sort(Enumerable *mass, int count)
{
    for(int i = 0; i < count-1; i++)
    {
        for(int j = i+1; j < count; j++)
        {
            if(mass[i] > mass[j])
            { swap(mass[i], mass[j]); }
        }
    }
}
```

Пример сужения концептуального пространства

- Наличие операций `==` и `!=` для классов `QDate`, `QTime` и `QString` позволяет одинаково работать с экземплярами этих классов в условных выражениях, не особо задумываясь над синтаксисом операций

Пример сужения концептуального пространства

```
bool operator== (const QDate & d) const;  
bool operator!= (const QDate & d) const;  
bool operator== (const QTime & t) const;  
bool operator!= (const QTime & t) const;  
bool operator== (const QString & other) const;  
bool operator!= (const QString & other) const;
```

```
QDate d1(2007, 12, 11), d2(1997, 1, 3);  
QString s("30.01.2007");
```

```
if((d1 == d2) &&  
    (d1.toString("dd.mm.yyyy") == s))  
{  
    puts("Equality");  
}
```


Понятие полиморфной переменной

- Одной из наиболее интересных особенностей объектно-ориентированных языков программирования является тот факт, что фактический тип переменной может не совпадать с типом, заявленным при ее описании
- Полиморфная переменная не только хранит данные фактического типа, но и позволяет использовать методы фактического типа

Понятие полиморфной переменной

- Для обозначения типа, присвоенного переменной при ее описании, будем использовать термин «**статический тип**»
- Термин «**динамический тип**» характеризует тип фактического значения
- Переменная, для которой динамический тип не совпадает (точнее, может не совпадать) со статическим, называется **полиморфной**

Реализация полиморфных переменных в языке Си++

- В языке Си++ полиморфными могут быть только указатели и ссылки
- Полиморфизм возникает, когда указателю (или ссылке) базового класса присваивается указатель на производный класс
- Законность использования свойств и методов динамического типа определяется компилятором на основе статического типа

Пример использования полиморфной переменной

```
class MyGraphicsPrimitive2D
{ // Графический примитив на плоскости
public:
    virtual void draw()
    { puts("MyGraphicsPrimitive2D"); }
};

class MyRectangle: public MyGraphicsPrimitive2D
{ // Прямоугольник
public:
    virtual void draw()
    { puts("MyRectangle"); }

    void setSize(int width, int height)
    { printf("W x H = %d x %d\n", width, height); }
};
```

Пример использования полиморфной переменной

```
// Создаем указатель на любой графический примитив
MyGraphicsPrimitive2D *primitive;

// Создаем прямоугольник
primitive = new MyRectangle();

// Отрисовываем прямоугольник. Обращение к методу
// MyRectangle::draw() возможно, т.к. аналогичный
// метод заявлен в базовом классе как virtual
primitive->draw();    // "MyRectangle"

// Пытаемся задать размеры прямоугольника.
// Это приведет к ошибке, т.к. метод не заявлен
// в базовом классе
// primitive->setSize(15, 20);
```

Преобразование статического типа к динамическому в языке Си++

- В рассмотренном выше примере метод `setSize()` не доступен через указатель базового класса, т.к. законность использования методов динамического типа определяется компилятором на основе статического типа
- Доступ к нему можно получить только после явного **приведения** статического типа к динамическому

Преобразование статического типа к динамическому в языке Си++

- Для приведения статического типа к динамическому рекомендуется использовать два оператора языка Си++ из системы **RTTI** (про RTTI будет сказано позже):

dynamic_cast <type-id> (expression)

static_cast <type-id> (expression)

- Главное достоинство указанных операторов заключается в том, что они возвращают **NULL** (для указателей), если приведение невозможно

Пример использования оператора `dynamic_cast`

```
MyEllipse    *ellipse = new MyEllipse;
MyRectangle  *rectangle;

// Пытаемся преобразовать эллипс в прямоугольник
rectangle = dynamic_cast<MyRectangle *>(ellipse);

if(rectangle == NULL)
{ puts("Convert is not possible"); }
```


Пример использования полиморфной переменной

```
// Создаем указатель на любой графический примитив
MyGraphicsPrimitive2D *primitive;

// Создаем прямоугольник
primitive = new MyRectangle();

// Задаем размеры прямоугольника
dynamic_cast<MyRectangle *>
    (primitive)->setSize(15, 20); // "W x H = 15 x 20"

// Отрисовываем прямоугольник
primitive->draw();                // "MyRectangle"
```

Понятие полиморфного контейнера

- Среди полиморфных переменных можно выделить полиморфные контейнеры
- **Полиморфным** называется **контейнер**, при объявлении которого используется статический тип, а хранятся элементы динамического типа

Пример полиморфного контейнера

```
// Создаем массив для хранения ЛЮБЫХ
// 3-х графических примитивов
MyGraphicsPrimitive2D *primitives[3];

// Заполняем массив РАЗНОРОДНЫМИ
// графическими примитивами
primitives[0]= new MyCircle();
primitives[1]= new MyEllipse();
primitives[2]= new MyRectangle();

// Отрисовываем все примитивы
for(int i = 0; i < 3; i++)
{ primitives[i]->draw(); }
```

Проблемы, связанные с полиморфными переменными

- Проблема **связывания** сообщений и методов
- Проблема **обращения** полиморфизма

Проблема связывания сообщений и методов

- Переменная может рассматриваться как с точки зрения ее описания (т.е. статически), так и с точки зрения ее текущего значения (т.е. динамически)
- Это различие становится важным, когда встречается метод, который определен в родительском классе и переопределен в дочернем, т.е. имеется одно сообщение и несколько адекватных ему методов

Проблема связывания сообщений и методов

- Чаще всего мы ожидаем, что **связывание** сообщения должно происходить исходя из **динамического** типа данных (как в примере с графическими примитивами)
- Однако существуют ситуации, где противоположный подход также полезен

Связывание сообщений и методов в языке Си++

- В языке Си++ для **обычных** переменных (не указателей или ссылок) связывание методов осуществляется всегда **статически**, т.к. они не являются полиморфными
- Но когда объекты обозначаются с помощью **указателей** или **ссылок** используется **динамическое** связывание при условии, что метод объявлен как **виртуальный**

Динамическое связывание в языке Си++ и виртуальные методы

- Метод, который объявлен с ключевым словом `virtual`, называется **виртуальным**
- Если он объявлен именно так, то поиск метода начинается с динамического типа, если нет – со статического
- **Тип** является **полиморфным**, если в нем имеются **виртуальные методы**

Пример использования динамического СВЯЗЫВАНИЯ

```
class MyGraphicsPrimitive2D
{
public:
    void setPos(int x, int y)
    { printf("Pos = (%d,%d)\n"); }

    virtual void draw()
    { puts("MyGraphicsPrimitive2D"); }
};

class MyEllipse: public MyGraphicsPrimitive2D
{
public:
    virtual void draw()
    { puts("MyEllipse"); }
};
```

Пример использования динамического СВЯЗЫВАНИЯ

```
class MyRectangle: public MyGraphicsPrimitive2D
{
public:
    void setPos(int x, int y)
    { printf("Rectangle pos = (%d,%d)\n"); }

    virtual void draw()
    { puts("MyRectangle"); }
};

class MyCircle: public MyEllipse
{
public:
    virtual void draw() { puts("MyCircle"); }
};
```

Пример использования динамического СВЯЗЫВАНИЯ

```
int _tmain(int argc, _TCHAR* argv[])
{
    // Создаем массив для хранения ЛЮБЫХ
    // 3-х графических примитивов
    MyGraphicsPrimitive2D *primitives[3];

    // Заполняем массив разнородными графическими
    // примитивами. Динамический тип элемента массива
    // не совпадает со статическим
    primitives[0]= new MyCircle();
    primitives[1]= new MyEllipse();
    primitives[2]= new MyRectangle();
    . . . . .
```

Пример использования динамического связывания

.....

```
// Отрисовываем все примитивы. Так как в  
// контейнере указатели на объекты и  
// метод draw() объявлен как виртуальный,  
// то используется метод динамического типа
```

```
for(int i = 0; i < 3; i++)  
{  
    primitives[i]->draw();  
}
```

.....

```
MyCircle  
MyEllipse  
MyRectangle
```

Пример использования статического связывания

.....

```
// Хотя переменная primitives[2] полиморфная, но  
// происходит СТАТИЧЕСКОЕ связывание, т.к.  
// метод не объявлен как виртуальный  
primitives[2]->setPos(30, 20);
```

```
    getch();  
    return 0;  
}
```

```
Pos = (30, 20)
```

Сравнение статического и динамического СВЯЗЫВАНИЯ

поиск при
статическом
связывании

MyRectangle

```
MyRectangle::setPos()  
MyGraphicsPrimitive2D::setPos()
```

поиск при
динамическом
связывании

MyCircle

```
MyCircle::draw()  
MyEllipse::draw()  
MyGraphicsPrimitive2D::draw()
```

Рекомендации по использованию полиморфных переменных в языке Си++

- Для использования полиморфной переменной необходимо:

- 1) **Объявить** переменную типа **указатель** или **ссылку** на базовый класс
- 2) Сделать базовый класс **полиморфным**, т.е. объявить в нем хотя бы один **виртуальный** метод
- 3) Виртуальными должны быть объявлены все методы, которые будут **переопределены** в производном классе и к которым будет **обращение** через полиморфную переменную

Проблема обращения полиморфизма

- Чаще всего, данная проблема возникает в связи с использованием **полиморфных контейнеров**
- Мы помещаем в полиморфный контейнер разнотипные объекты (например, различные примитивы), однако **при извлечении** объекта **не можем сказать**, какой перед нами объект и какими специфическими свойствами и специфическим поведением он обладает

Пример проблемы обращения полиморфизма

```
// Создаем массив для хранения ЛЮБЫХ
// 3-х графических примитивов
MyGraphicsPrimitive2D *primitives[3];

// Заполняем массив разнородными графическими
// примитивами .....

// Пытаемся изменить радиус у окружностей,
// возможны проблемы, т.к. не все графические
// примитивы являются окружностями
for(int i = 0; i < 3; i++)
{
    dynamic_cast<MyCircle *>(primitives[i])
        ->setRadius(30);
}
```

Решение проблемы обращения полиморфизма в языке Си++: RTTI

- В Си++ имеются средства для определения динамического типа переменной
- Они образуют систему **RTTI** (**R**untime **T**ype **I**dentification – идентификация типа во время выполнения)
- В системе **RTTI** каждый класс имеет связанную с ним структуру типа **type_info**, которая кодирует различную информацию о классе

Решение проблемы обращения полиморфизма в языке Си++: RTTI

- Для получения структуры `type_info` используется оператор `typeid`, который имеет следующий синтаксис:

`typeid(type-id)`

`typeid(expression)`

Операции и методы, поддерживаемые структурой `type_info`

operator <code>==</code>	сравнивает два типа данных	<pre>class A a; class B b; if(typeid(a) == typeid(b)) { puts("Type are equivalent"); } else { puts("Type are not equivalent"); }</pre>
operator <code>!=</code>	сравнивает два типа данных	<pre>class A a; class B b; if(typeid(a) != typeid(b)) { puts("Type are not equivalent"); } else { puts("Type are equivalent"); }</pre>
<code>name()</code>	возвращает имя класса в виде текстовой строки	<pre>class A a; class B b; puts(typeid(a).name()); // "class A" puts(typeid(b).name()); // "class B"</pre>

Особенность работы метода typeid::name()

- Метод `typeid::name()` возвращает **динамический** тип переменной, если статический тип данных является **полиморфным**
- В противном случае всегда возвращается статический тип переменной
- **Тип** является **полиморфным**, если в нем имеются **виртуальные методы**

Пример использования RTTI: тип не является полиморфным

```
// Тип не является полиморфным
```

```
class MyGraphicsPrimitive2D  
{  
public:  
    void draw() {...}  
};
```

```
// Производные типы также не являются полиморфными
```

```
class MyEllipse:    public MyGraphicsPrimitive2D {...};  
class MyRectangle: public MyGraphicsPrimitive2D {...};  
class MyCircle:    public MyEllipse                {...};
```

```
// Создаем массив для хранения ЛЮБЫХ
```

```
// 3-х графических примитивов
```

```
MyGraphicsPrimitive2D *primitives[3];
```

```
.....
```

Пример использования RTTI: тип не является полиморфным

```
.....  
// Заполняем массив разнородными графическими  
// примитивами  
primitives[0]= new MyCircle();  
primitives[1]= new MyEllipse();  
primitives[2]= new MyRectangle();  
  
// Определяем тип элементов массива  
for(int i = 0; i < 3; i++)  
{ puts(typeid(*primitives[i]).name()); }
```

```
class MyGraphicsPrimitive2D  
class MyGraphicsPrimitive2D  
class MyGraphicsPrimitive2D
```

Пример использования RTTI: тип является полиморфным

```
// Тип является полиморфным
class MyGraphicsPrimitive2D
{ public: virtual void draw() {} };

// Производные типы также являются полиморфными
class MyEllipse:    public MyGraphicsPrimitive2D {};
class MyRectangle: public MyGraphicsPrimitive2D {};
class MyCircle:    public MyEllipse             {};

// Создаем массив для хранения ЛЮБЫХ
// 3-х графических примитивов
MyGraphicsPrimitive2D *primitives[3];
.....
```


Пример использования RTTI: тип является полиморфным

```
.....  
// Заполняем массив разнородными графическими  
// примитивами  
primitives[0]= new MyCircle();  
primitives[1]= new MyEllipse();  
primitives[2]= new MyRectangle();  
  
// Определяем тип элементов массива  
for(int i = 0; i < 3; i++)  
{ puts(typeid(*primitives[i]).name()); }
```

```
class MyCircle  
class MyEllipse  
class MyRectangle
```

Пример обращения полиморфизма

```
// Создаем массив графических примитивов
// primitives[3] и заполняем его разнородными
// графическими примитивами .....

// Изменяем радиус только у окружностей
for(int i = 0; i < 3; i++)
{
    if(strcmp(typeid(*primitives[i]).name(),
               "MyCircle") == 0)
    {
        MyCircle *circle=
            dynamic_cast<MyCircle *>(primitives[i]);
        circle->setRadius(30);
    }
}
```

Особенности использования операторов

RTTI: `dynamic_cast` и `static_cast`

- Как говорилось ранее, для преобразования статического типа к динамическому используются два оператора: `dynamic_cast` и `static_cast`
- Предпочтительнее использовать оператор `dynamic_cast`, но он работает только с полиморфными типами

Рекомендации по использованию полиморфных переменных в языке Си++

- Если производный класс не расширяет базовый класс, то **нет необходимости** в приведении типов
- Если же через полиморфную переменную требуется обратиться к **дополнительным** свойствам и методам производного класса, то необходимо **явное приведение** типа с помощью операции `dynamic_cast`
- Если существует **несколько** производных классов со своим набором уникальных свойств и методов, то предварительно необходимо **распознать** класс с помощью метода `typeid::name()`