



Тема 14

Полиморфизм

Полиморфизм

- **Полиморфизм** (в общем случае) – способность объекта изменять свою форму в процессе его функционирования.

В программировании полиморфизм – свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач.

В C++ полиморфизм реализован:

- механизмом перегрузки функций;
- механизмом шаблонов;
- механизмом переопределения методов класса

Пример полиморфизма – метод Module()

```
class Point2D{  
...  
public:  
    double Module() const;  
...  
};  
  
class Point3D: public Point2D{  
...  
public:  
    double Module() const;  
...  
};
```

Реализация полиморфизма в методе Module()

```
double Point2D::Module() const {  
    return sqrt(x*x + y*y);  
}  
  
double Point3D::Module() const {  
    double x = GetX(), y = GetY()  
    return sqrt(x*x + y*y + z*z);  
}
```

Примеры использования метода Module()

```
Point2D p(3, 4);  
Point3D q(3, 4, 5);  
  
cout << p.Module();  
    // результат - 5  
  
cout << q.Module();  
    // результат - 7.07107  
  
Point2D p1; p1 = q;  
cout << p1.Module();  
    // результат - 5 (правильно, т.к. стандартный  
    // оператор присваивания копирует только поля  
    // класса Point2D, игнорируя остальные)  
  
cout << typeid(p1).name();  
    // результат - class Point2D
```

Странный результат при использовании метода Module()

```
Point2D *cp = new Point3D(3, 4, 5);  
Point3D q(3, 4, 5);  
  
cout << typeid(*cp).name();  
    // результат - class Point3D
```

Однако ...

```
cout << cp->Module();  
    // результат - 5, хотя должен быть 7.07107
```

Причина – ошибка в определении того, какой метод Module() должен использоваться (для класса Point2D или Point3D)

Раннее и позднее связывание

Приложение может быть создано с использованием двух механизмов связывания (определения того, какая из полиморфных функций должна быть вызвана):

- **раннее** связывание – адрес вызываемой функции явно определяется при создании приложения (на этапе линковки);
- **позднее** связывание (применяется только для методов классов) – во время выполнения приложения определяется действительный класс объекта, адрес которого находится в указателе, и вызывается метод нужного класса.

По умолчанию работает раннее связывание. Для включения позднего связывания необходимо объявить соответствующие методы **виртуальными**

Объявление виртуального метода Module()

```
class Point2D{
...
public:
    virtual double Module() const;
...
};

class Point3D: public Point2D{
...
public:
    virtual double Module() const;
...
};
```


Результаты использования виртуального метода Module()

```
void f1(Point2D x) {cout << x.Module() << endl;}
void f2(Point2D* x) {cout << x->Module() << endl;}
void f3(Point2D& x) {cout << x.Module() << endl;}

Point3D q(3, 4, 5);
Point2D *cp = &q;

cout << cp->Module();
    // результат - 7.07107 (правильно)

f1(q); // результат - 5 (правильно, работает
    // конструктор копирования для класса Point2D)

f2(cp); // результат - 7.07107 (правильно, работает
    // механизм вызова виртуальных функций)

f3(q); // результат - 7.07107 (также правильно,
    // поскольку ссылка - тоже указатель)
```

Особенности работы с деструкторами при наследовании

Деструкторы не наследуются так же, как другие методы класса. Вместо этого у класса-потомка создаётся свой собственный деструктор по умолчанию.

При корректном уничтожении объекта работает деструктор его собственного класса, а потом – деструкторы всех его предков (в порядке, обратном иерархии наследования)

Пример работы цепочки деструкторов (описание)

```
// деструкторы включены в состав классов только для  
// иллюстрации их работы!
```

```
class Point2D{
```

```
...
```

```
public:
```

```
    ~Point2D() { cout << "Point2D done" << endl; }
```

```
...
```

```
};
```

```
class Point3D: public Point2D{
```

```
...
```

```
public:
```

```
    ~Point3D() { cout << "Point3D done" << endl; }
```

```
...
```

```
};
```

Пример работы цепочки деструкторов (работа)

```
int main () {  
    Point3D q(3, 4, 5);  
    Point2D p;  
    ...  
    return 0;  
}
```

Результат (уничтожение объектов ведётся в порядке , обратном их созданию):

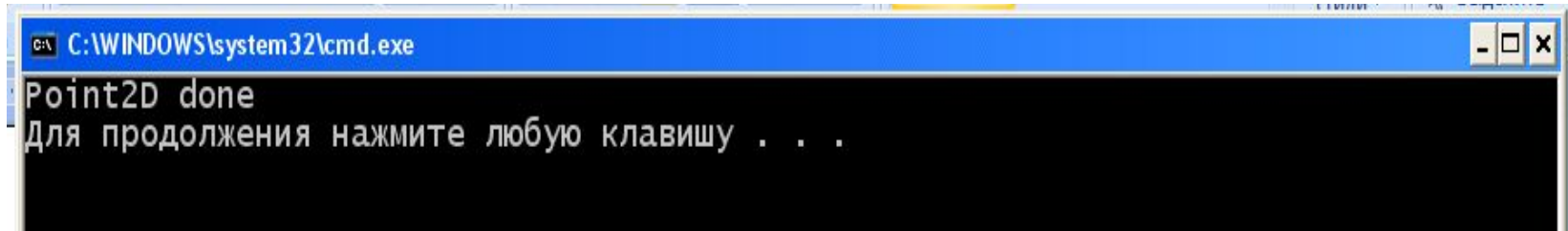
C:\WINDOWS\system32\cmd.exe

```
Point2D done  
Point3D done  
Point2D done  
Для продолжения нажмите любую клавишу . . .
```

Неправильная работа цепочки деструкторов

```
int main () {  
    Point2D *cp = new Point3D(1,2,3);  
    ...  
    delete cp;  
    return 0;  
}
```

Результат - деструктор для класса Point3D не вызывается



```
C:\WINDOWS\system32\cmd.exe  
Point2D done  
Для продолжения нажмите любую клавишу . . .
```

Причина – деструктор не объявлен как виртуальный!

Правильное объявление деструкторов при наследовании

```
// виртуальные деструкторы должны быть  
// включены в состав классов,  
// даже если их тело пусто!
```

```
class Point2D{  
...  
public:  
    virtual ~Point2D() {}  
...  
};
```

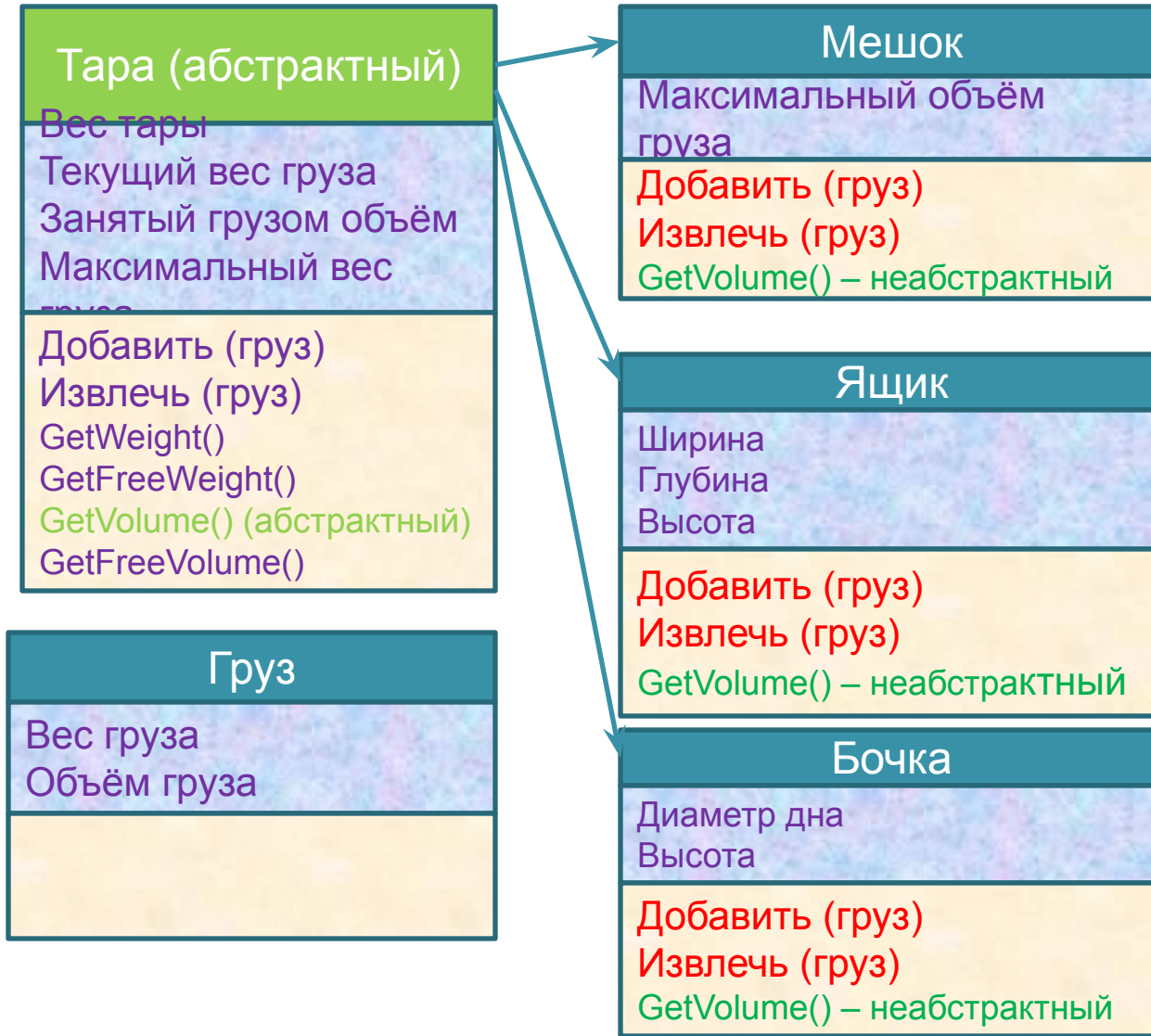
```
class Point3D: public Point2D{  
...  
public:  
    virtual ~Point3D() {}  
...  
};
```

Абстрактные методы и классы

При построении иерархии классов классы, находящиеся на верхних уровнях иерархии, могут не быть реализованы. Это означает, что нельзя создать ни одного объекта, принадлежащего этому классу. Такие классы называются **абстрактными**.

Необходимость создания абстрактных классов состоит в том, чтобы на верхнем уровне описать полиморфные методы, общие для всех классов-потомков. На уровне абстрактного класса эти методы (абстрактные методы) не могут быть реализованы, они переопределяются в неабстрактных классах-потомках.

Пример иерархии с абстрактными классами



Описание абстрактных методов и их переопределение в неабстрактных классах

```
class Container {
private:
    double used_volume;
...
public:
    virtual double GetVolume() const = 0;
...
};

class Bag: public Container {
private:
    double volume;
...
public:
    virtual double GetVolume() const ;
...
};
```

Вызов абстрактных методов в абстрактных классах, их переопределение

```
double Container:: GetVolume() const {}

double Container:: GetFreeVolume() const {
    return GetVolume() - used_volume;
}

double Bag:: GetVolume() const {
    return volume;
}

double Crate:: GetVolume() const {
    return width * depth * height;
}

double Barrel:: GetVolume() const {
    return 3.1415926 * radius * radius * height;
}
```

Наличие тела в абстрактных методах

```
class Animal {  
  private:  
    string name;  
  ...  
  public:  
    virtual void eats() const = 0;  
  ...  
};  
  
class Panda: public Animal {  
  ...  
  public:  
    virtual void eats() const;  
  ...  
};
```

Наличие тела в абстрактных методах (продолжение)

```
void Animal::eats() const {  
    cout << name << " eats ";  
}  
  
void Panda::eats() const {  
    Animal::eats();  
    cout << ", shoots and leaves" << endl;  
}
```

```
Panda p("My panda");  
p.eats();
```

```
// Результат: My panda eats, shoots and leaves
```

Проблемы при работе с классом exception

Использование только класса exception при выбросе исключений приводит к тому, что анализ перехваченного исключения и его правильная обработка становится сложной задачей:

```
void f1 () { ...
    throw exception("Error");
... }
void f2 () { ...
    throw exception("Error");
... }
try {
    f1 ();
    f2 ();
}
catch (const exception& e) {
    // непонятно, где выброшено исключение
}
```

Построение иерархии ИСКЛЮЧЕНИЙ

Одно из решений этой проблемы состоит в создании собственных классов-исключений – потомков класса `exception`. Работа этих исключений ничем не отличается от работы `exception`, поэтому достаточно написать конструкторы (которые не наследуются):

```
class my_exception: public exception {  
public:  
    my_exception(const char* const message)  
        : exception(message) {}  
    my_exception(const my_exception &right)  
        : exception(right) {}  
};
```

Обработка иерархии исключений

Важен порядок перехвата исключений: вначале обрабатываются потомки, а потом – предки!

Несоблюдение этого правила приведёт к тому, что специфическая обработка исключений-потомков будет пропущена.

```
try {  
    ...  
}  
catch (const my_exception& e) {  
    ...  
}  
catch (const exception& e) {  
    ...  
}  
catch (...) {  
    ...  
}
```