

Лекция 3- Наследование классов и полиморфизм ООП

Правила наследования классов

Классы в объектно-ориентированных программах используются для моделирования информационных сущностей реальной и/или программной предметной области. Взаимодействие этих сущностей предметной области решает поставленную задачу. Взаимодействие сущностей в отношении *родитель/потомок* — это отношение **наследования**, именуемое также отношением *обобщение/специализация (is-a)*. Наследование является одним из основных механизмов в объектно-ориентированных языках программирования.

Наследование — отношение такое между классами, при котором один класс повторяет структуру и поведение другого класса (*единочное наследование*) или других (*множественное наследование*) классов.

Класс, поведение и структура которого наследуется, называется базовым классом, а класс, который наследует — производным классом.

Общий синтаксис создания производного класса при одиночном наследовании:

```
class имя : ключ_доступа имя_базового_класса {
    // тело класса
};
```

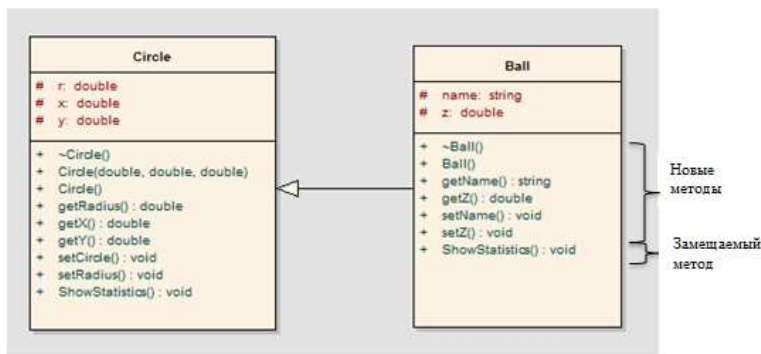


Рис.1 Базовый класс Circle и производный класс Ball

Производный класс Ball наследует **все** члены базового класса Circle, а также замещает (переопределяет) и добавляет новые методы (Рис.1). Ядром класса Ball будет полный состав класса Circle (Рис.2).

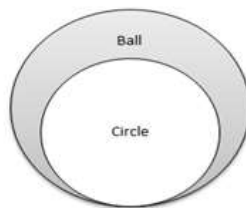


Рис.2 Производный класс Ball

Пример 1. Объявление класса Ball как производного от класса Circle на языке C++.

```
class Circle
{
public:
    Circle(double vr, double vy, double vx);

    Circle();

    ~Circle();

    double getRadius();
```

```
double getX();

double getY();

void setCircle();

void setRadius();

void ShowStatistics();
```

protected:

```
double r;

double x;

double y;

};
```

//производный класс Ball получен открытым наследованием от базового

// класса Circle

```
class Ball : public Circle
```

```
{
```

public:

```
Ball();

~Ball();

string getName();

double getZ();

void setName();

void setZ();

void ShowStatistics();
```

protected:

```
string name;

double z;

};
```

В случае **множественного наследования** после двоеточия перечисляются через запятую все базовые классы со своими ключами доступа к ним. Производный класс, в свою очередь, сам может служить базовым классом. Такой набор связанных классов традиционно называется **иерархией классов**. Иерархия чаще всего является деревом, но может иметь и более общую структуру графа.

Проблема неоднозначности при множественном наследовании. Пусть от базового класса Parent, имеющего поле *x*, наследуются два класса HeirA и HeirB, а класс HeirAB - производный от этих двух классов (множественное наследование). Если попытаться обратиться к элементу *x* из методов класса HeirAB, то компилятор воспримет выражение HeirAB.: *x* как неоднозначное и прекратит работу. Для решения проблемы в C++ предусмотрен механизм, благодаря которому в класс HeirAB будет включена только одна копия класса Parent. Достигается это добавлением спецификатора *virtual* перед ключами доступа к Parent в объявлениях классов HeirA и HeirB.

Пример 2. Решение проблемы неоднозначности при множественном наследовании: иерархия классов Parent, HeirA, HeirB, HeirAB.

```
#include <iostream>

using namespace std;
```

```
class Parent {  
  
protected:  
  
    int x;  
  
public:  
  
    Parent(int _x = 0) { x = _x; }  
  
};  
  
class HeirA: virtual public Parent {  
  
public:  
  
    void AddA(int y) { Parent::x += y; }  
  
};  
  
class HeirB: virtual public Parent {  
  
public:  
  
    void AddB(int y) { Parent::x += y; }  
  
};  
  
class HeirAB: public HeirA, public HeirB {  
  
public:  
  
    void ShowX() { cout << "x = " << Parent::x << endl; }  
  
};  
  
int main()  
{  
  
    HeirAB d;  
  
    d.ShowX();  
  
    d.AddA(10); d.ShowX();  
  
    d.AddB(5); d.ShowX();  
  
    return 0;  
}
```

Программа выдаст:

x=0

x=10

x=15

В случае использование **шаблона** базового класса производные классы также будут шаблонами.

Пример 3. Синтаксис описания шаблонами базового и производных классов Примера 2.

```
#include <iostream>
```

```
using namespace std;
```

```

template <class T>

class Parent {

protected:

    T x;

public:

    Parent(T _x = 0) { x = _x; }

};

```

```

template <class T>

class HeirA : virtual public Parent<T> {

public:

    void AddA(T y) { Parent<T>::x += y; }

};

```

```

template <class T>

class HeirB : virtual public Parent<T> {

public:

    void AddB(T y) { Parent<T>::x += y; }

};

```

```

template <class T>

class HeirAB : public HeirA<T>, public HeirB<T> {

public:

    void ShowX() { cout << "x = " << Parent<T>::x << endl; }

};

```

```

int main()

{

    HeirAB<int> d;

    d.ShowX();

    d.AddA(10); d.ShowX();

    d.AddB(5); d.ShowX();

    return 0;

}

```

До сих пор доступ к элементам класса регулировался с помощью двух спецификаторов: `private` – закрытая часть класса, `public` – открытая часть класса. Для базовых классов возможен еще один спецификатор доступа – **`protected`**, который определяет так называемую **защищенную** часть класса (рис.3). Смысл «защиты» заключается в том, что элементы этой части класса являются доступными для любого производного класса, но в то же время они недоступны извне классов данной иерархии.



Рис.3 Доступ к защищённым членам класса

Доступ к членам базового класса в производном классе регулируется **ключом доступа**, описанному в объявлении производного класса. Этот ключ определяет вид наследования: *открытое* (*public*), *защищенное* (*protected*) или *закрытое* (*private*).

Синтаксис описания:

```
class имя : [private|protected|public] базовый_класс {
    // тело класса
    ...
};
```

Открытое наследование сохраняет статус доступа всех элементов базового класса, *защищенное* – понижает статус доступа public элементов базового класса до protected, и наконец, *закрытое* – понижает статусы доступа public и protected элементов базового класса до private. В Таблице 1 показан спецификатор доступа к членам базового класса из производного при разных видах наследования.

Таблица 1

Спецификатор доступа в производном классе к членам базового класса

Ключ доступа			
Доступ в базовом классе	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Замещение методов базового класса. Иногда в производном классе требуется несколько иная реализация метода, унаследованного из базового класса. Замещение (переопределение) метода производится путем объявления в производном классе метода с таким же именем, но с другим алгоритмом обработки. Если понадобится все-таки вызвать из потомка метод предка, используется операция доступа к области видимости (::). В Примере 1 замещаемым является метод ShowStatistics().

Пример 4. В примере замещаемой является функция Show.

```
class Base {
public:
    void Show() { cout << "Это Работа 4!" << endl; }
};

class Mess: public Base
{
public:
    void Show() {
        Base::Show();    //Вызов метода базового класса
        cout << "Открытое наследование?" << endl;
    }
}
```

```
};

class SubMess : public Mess {
public:
    void Show() {
        Mess::Show();    //Вызов метода базового класса
        cout << "Отлично!" << endl;
    }
};

int main()
{
    SubMess sd;
    sd.Show();
}
```

Эта программа выведет на экран:

Это Работа 4!

Открытое наследование?

Отлично!

Конструкторы, деструкторы и операции присваивания базового класса не наследуются, поэтому при создании производного класса встает вопрос, нужны ли эти методы и как они должны быть реализованы.

«Решая вопрос о конструкторах производного класса, руководствуйтесь следующими правилами:

1. Если в базовом классе вообще нет конструктора или есть конструктор по умолчанию, то производному классу конструктор нужен только в том случае, когда требуется инициализировать поля, введенные в этом классе.
2. Если вы не определили ни одного конструктора, компилятор самостоятельно создаст конструктор по умолчанию, из которого будет вызван конструктор по умолчанию базового класса.
3. Если в базовом классе есть конструктор с аргументами, то в производном классе, как правило, требуется задать конструктор со списком аргументов, включающим значения для передачи конструктору базового класса; этот конструктор надо вызвать в списке инициализации» [2].

Необходимость в деструкторе для производного класса определяется тем, нужно ли освобождать какие-либо ресурсы, выделенные в конструкторе. Если такой необходимости нет, то можно доверить компилятору создать деструктор по умолчанию. В нем обеспечивается вызов деструктора базового класса.

На этапе выполнения программы при создании объекта производного класса сначала вызываются конструкторы базовых классов, начиная с самого верхнего уровня, затем конструкторы объектов-элементов класса, и в последнюю очередь — конструктор класса. При уничтожении объекта (например, когда покидается область его видимости) деструкторы вызываются в порядке, обратном вызову конструкторов.

Наследование повышает модульность программного кода и способствует повторному использованию компонентов. Группу хорошо сконструированных классов общего назначения можно организовать в библиотеку. Интерфейс таких библиотечных классов следует опубликовать в заголовочный файл модуля, а реализацию — инкапсулировать в файл реализации модуля. Библиотечные классы могут специализироваться путем создания новых производных классов. В этих классах к элементам данных и функциям базового класса добавляются новые данные. Подобный метод широко используется для создания графических пользовательских интерфейсов. Классы приложения наследуют свойства из библиотечных классов — окон, диалоговых блоков, графических командных кнопок. Программисты приложения применяют эти свойства, реализованные в библиотечных классах, добавляют специфические свойства, которые определяют, как именно должна вести себя в приложении конкретная кнопка, диалоговый блок или окно. В процессе такой специализации вносить изменения в базовые библиотечные классы не требуется. Следовательно, нет необходимости в их редактировании и перекомпиляции [3].

Оглавление

[1. Наследование классов](#)

[2. Класс как полиморфный тип](#)

1. Наследование классов

Правила наследования классов

Классы в объектно-ориентированных программах используются для моделирования информационных сущностей реальной и/или программной предметной области. Взаимодействие этих сущностей предметной области решает поставленную задачу. Взаимодействие сущностей в отношении *родитель/потомок* — это отношение **наследования**, именуемое также отношением *обобщение/специализация (is-a)*. Наследование является одним из основных механизмов в объектно-ориентированных языках программирования.

Наследование — отношение такое между классами, при котором один класс повторяет структуру и поведение другого класса (*единочное наследование*) или других (*множественное наследование*) классов.

Класс, поведение и структура которого наследуется, называется базовым классом, а класс, который наследует — производным классом.

Общий синтаксис создания производного класса при одиночном наследовании:

```
class имя : ключ_доступа имя_базового_класса {
    // тело класса
};
```

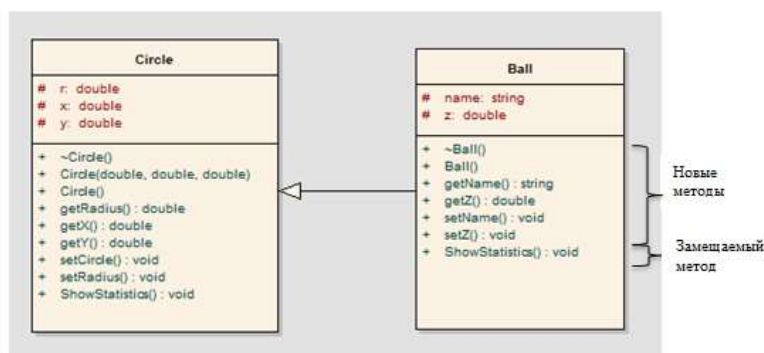


Рис.1 Базовый класс Circle и производный класс Ball

Производный класс Ball наследует **все** члены базового класса Circle, а также замещает (переопределяет) и добавляет новые методы (Рис.1). Ядром класса Ball будет полный состав класса Circle (Рис.2).

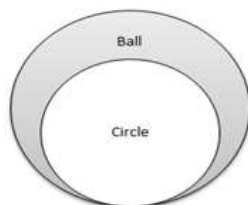


Рис.2 Производный класс Ball

Пример 1. Объявление класса Ball как производного от класса Circle на языке C++.

```
class Circle
{
public:
    Circle(double vr, double vy, double vx);

    Circle();

    ~Circle();

    double getRadius();

    double getX();

    double getY();

    void setCircle();
```



```

    void setRadius();

    void ShowStatistics();

protected:

    double r;

    double x;

    double y;

};

//производный класс Ball  получен открытым наследованием от базового

// класса Circle

class Ball : public Circle

{

public:

    Ball();

    ~Ball();

    string getName();

    double getZ();

    void setName();

    void setZ();

    void ShowStatistics();

protected:

    string name;

    double z;

};

```

В случае **множественного наследования** после двоеточия перечисляются через запятую все базовые классы со своими ключами доступа к ним. Производный класс, в свою очередь, сам может служить базовым классом. Такой набор связанных классов традиционно называется **иерархией классов**. Иерархия чаще всего является деревом, но может иметь и более общую структуру графа.

Проблема неоднозначности при множественном наследовании. Пусть от базового класса Parent, имеющего поле x , наследуются два класса HeirA и HeirB, а класс HeirAB - производный от этих двух классов (множественное наследование). Если попытаться обратиться к элементу x из методов класса HeirAB, то компилятор воспримет выражение HeirAB:: x как неоднозначное и прекратит работу. Для решения проблемы в C++ предусмотрен механизм, благодаря которому в класс HeirAB будет включена только одна копия класса Parent. Достигается это добавлением спецификатора *virtual* перед ключами доступа к Parent в объявлениях классов HeirA и HeirB.

Пример 2. Решение проблемы неоднозначности при множественном наследовании: иерархия классов Parent, HeirA , HeirB, HeirAB.

```

#include <iostream>

using namespace std;

class Parent {

protected:

    int x;

```

```
public:
```

```
    Parent(int _x = 0) { x = _x; }
```

```
};
```

```
class HeirA: virtual public Parent {
```

```
public:
```

```
    void AddA(int y) { Parent::x += y; }
```

```
};
```

```
class HeirB: virtual public Parent {
```

```
public:
```

```
    void AddB(int y) { Parent::x += y; }
```

```
};
```

```
class HeirAB: public HeirA, public HeirB {
```

```
public:
```

```
    void ShowX() { cout << "x = " << Parent::x << endl; }
```

```
};
```

```
int main()
```

```
{
```

```
    HeirAB d;
```

```
    d.ShowX();
```

```
    d.AddA(10); d.ShowX();
```

```
    d.AddB(5); d.ShowX();
```

```
return 0;
```

```
}
```

Программа выдаст:

```
x=0
```

```
x=10
```

```
x=15
```

В случае использования **шаблона** базового класса производные классы также будут шаблонами.

Пример 3. Синтаксис описания шаблонами базового и производных классов Примера 2.

```
#include <iostream>
```

```
using namespace std;
```

```
template <class T>
```

```
class Parent {
```

protected:

T x;

public:

Parent(T _x = 0) { x = _x; }

};

template <class T>

class HeirA : virtual public Parent<T> {

public:

void AddA(T y) { Parent<T>::x += y; }

};

template <class T>

class HeirB : virtual public Parent<T> {

public:

void AddB(T y) { Parent<T>::x += y; }

};

template <class T>

class HeirAB : public HeirA<T>, public HeirB<T> {

public:

void ShowX() { cout << "x = " << Parent<T>::x << endl; }

};

int main()

{

HeirAB<int> d;

d.ShowX();

d.AddA(10); d.ShowX();

d.AddB(5); d.ShowX();

return 0;

}

До сих пор доступ к элементам класса регулировался с помощью двух спецификаторов: `private` – закрытая часть класса, `public` – открытая часть класса. Для базовых классов возможен еще один спецификатор доступа – **`protected`**, который определяет так называемую **защищенную** часть класса (рис.3). Смысл «защиты» заключается в том, что элементы этой части класса являются доступными для любого производного класса, но в то же время они недоступны извне классов данной иерархии.



Рис.3 Доступ к защищённым членам класса

Доступ к членам базового класса в производном классе регулируется **ключом доступа**, описанному в объявлении производного класса. Этот ключ определяет вид наследования: *открытое* (*public*), *защищенное* (*protected*) или *закрытое* (*private*).

Синтаксис описания:

```
class имя : [private|protected|public] базовый_класс {
    // тело класса
    ...
};
```

Открытое наследование сохраняет статус доступа всех элементов базового класса, *защищенное* – понижает статус доступа public элементов базового класса до protected, и наконец, *закрытое* – понижает статусы доступа public и protected элементов базового класса до private. В Таблице 1 показан спецификатор доступа к членам базового класса из производного при разных видах наследования.

Таблица 1

Спецификатор доступа в производном классе к членам базового класса

Ключ доступа			
Доступ в базовом классе	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Замещение методов базового класса. Иногда в производном классе требуется несколько иная реализация метода, унаследованного из базового класса. Замещение (переопределение) метода производится путем объявления в производном классе метода с таким же именем, но с другим алгоритмом обработки. Если понадобится все-таки вызвать из потомка метод предка, используется операция доступа к области видимости (::). В Примере 1 замещаемым является метод ShowStatistics().

Пример 4. В примере замещаемой является функция Show.

```
class Base {
public:
    void Show() { cout << "Это Работа 4!" << endl; }
};

class Mess: public Base
{
public:
    void Show() {
        Base::Show();    //Вызов метода базового класса
        cout << "Открытое наследование?" << endl;
    }
}
```

```
};

class SubMess : public Mess {
public:
    void Show() {
        Mess::Show();    //Вызов метода базового класса
        cout << "Отлично!" << endl;
    }
};
```

```
int main()
{
    SubMess sd;

    sd.Show();
}
```

Эта программа выведет на экран:

Это Работа 4!

Открытое наследование?

Отлично!

Конструкторы, деструкторы и операции присваивания базового класса не наследуются, поэтому при создании производного класса встает вопрос, нужны ли эти методы и как они должны быть реализованы.

«Решая вопрос о конструкторах производного класса, руководствуйтесь следующими правилами:

1. Если в базовом классе вообще нет конструктора или есть конструктор по умолчанию, то производному классу конструктор нужен только в том случае, когда требуется инициализировать поля, введенные в этом классе.
2. Если вы не определили ни одного конструктора, компилятор самостоятельно создаст конструктор по умолчанию, из которого будет вызван конструктор по умолчанию базового класса.
3. Если в базовом классе есть конструктор с аргументами, то в производном классе, как правило, требуется задать конструктор со списком аргументов, включающим значения для передачи конструктору базового класса; этот конструктор надо вызвать в списке инициализации» [2].

Необходимость в деструкторе для производного класса определяется тем, нужно ли освобождать какие-либо ресурсы, выделенные в конструкторе. Если такой необходимости нет, то можно доверить компилятору создать деструктор по умолчанию. В нем обеспечивается вызов деструктора базового класса.

На этапе выполнения программы при создании объекта производного класса сначала вызываются конструкторы базовых классов, начиная с самого верхнего уровня, затем конструкторы объектов-элементов класса, и в последнюю очередь — конструктор класса. При уничтожении объекта (например, когда покидается область его видимости) деструкторы вызываются в порядке, обратном вызову конструкторов.

Наследование повышает модульность программного кода и способствует повторному использованию компонентов. Группу хорошо сконструированных классов общего назначения можно организовать в библиотеку. Интерфейс таких библиотечных классов следует опубликовать в заголовочный файл модуля, а реализацию — инкапсулировать в файл реализации модуля. Библиотечные классы могут специализироваться путем создания новых производных классов. В этих классах к элементам данных и функциям базового класса добавляются новые данные. Подобный метод широко используется для создания графических пользовательских интерфейсов. Классы приложения наследуют свойства из библиотечных классов — окон, диалоговых блоков, графических командных кнопок. Программисты приложения применяют эти свойства, реализованные в библиотечных классах, добавляют специфические свойства, которые определяют, как именно должна вести себя в приложении конкретная кнопка, диалоговый блок или окно. В процессе такой специализации вносить изменения в базовые библиотечные классы не требуется. Следовательно, нет необходимости в их редактировании и перекомпиляции [3].

2. Класс как полиморфный тип

Виртуальные методы и абстрактные классы

Одним из главных достоинств объектно-ориентированного подхода является возможность описать класс как полиморфный тип.

Полиморфизм – это возможность использовать одинаковые имена для разных методов, входящих в различные классы иерархии наследования. Концепция полиморфизма обеспечивает в случае применения метода к объекту использование именно того метода, который соответствует классу объекта.

☺ Полиморфизм позволяет повысить процент повторного использования кода и сократить тем самым размер программы и временные затраты на её написание.

Для корректного доступа через указатель на базовый класс к переопределенным методам иерархии классов используют механизм **виртуальных методов**. Виртуальный механизм работает только при использовании указателей и ссылок на объекты. Объект, определенный через указатель или ссылку и содержащий виртуальные методы, называется **полиморфным**. Механизм виртуальных методов заключается в том, что, результат вызова виртуального метода с использованием указателя или ссылки зависит не от того, на основе какого типа создан указатель, а от типа объекта, на который указывает этот указатель.

Виртуальный метод объявляется спецификатором **virtual**. Во всех классах-наследниках наследуемый виртуальный метод остается виртуальным или **полиморфным**. Таким образом, все типы-наследники полиморфного типа являются полиморфными типами.

Класс, содержащий хотя бы один виртуальный метод, называется **полиморфным типом (классом)**, а объект этого типа – **полиморфным объектом**.

По отношению ко всем виртуальным методам компилятор применяет стратегию *позднего, или динамического, связывания*. При вызове виртуального метода через указатель на полиморфный объект осуществляется динамический выбор метода в зависимости от текущего объекта, а не от типа указателя на объект. И поэтому метод выбирается на этапе выполнения, а не компиляции. В этом проявляется динамический полиморфизм.

Для реализации динамического связывания компилятор создает *таблицу виртуальных методов (vtbl)*, а к каждому объекту добавляет скрытое поле ссылки (*vptr*) на таблицу *vtbl*. Это несколько снижает эффективность программы, поэтому рекомендуется делать виртуальными только те методы, которые переопределяются в производных классах или являются потенциальными кандидатами на такое переопределение.

Эффективнее всего работать с объектами одной иерархии через указатель на базовый класс. Дело в том, что при открытом (public) наследовании можно присваивать такому указателю адрес объекта как базового класса, так и любого производного класса.

Пример 1. Рассмотрим следующую иерархию классов.

```
#include <iostream>

using namespace std;

class Base {
public:
    Base(int X = 10) { x = X; }
    void ShowX() { cout << "x = " << x << "; "; }
    void ModifyX() { x *= 2; }
protected:
    int x;
};

class Derived : public Base {
public:
    void ModifyX() { x /= 2; }
};

int main() {
    Base b;        Derived d;
    b.ShowX();     d.ShowX();
    Base* pB;
    pB = &b;      pB->ModifyX();  pB->ShowX();
    pB = &d;      pB->ModifyX();  pB->ShowX();
    return 0;
}
```

Эта программа выведет на экран:

x = 10; x = 10; x = 20; x = 20;

Второй вызов метода ModifyX() происходит также из базового класса, так как указатель имеет тип базового класса. Это статическое или раннее связывание на этапе компиляции по типу переменной.

Для реализации полиморфизма или динамического связывания нужно модифицировать приведенную выше программу добавлением спецификатора *virtual* перед заголовком метода ModifyX() в базовом классе, так что теперь определение метода примет вид: