Лекция 8

Иерархия классов

Можно использовать каждый класс в качестве основы для определения или создания другого класса (подкласса). Можно использовать несколько классов для определения подкласса. Стрелки на Рис. 8.1. всегда указывают на суперкласс(ы).

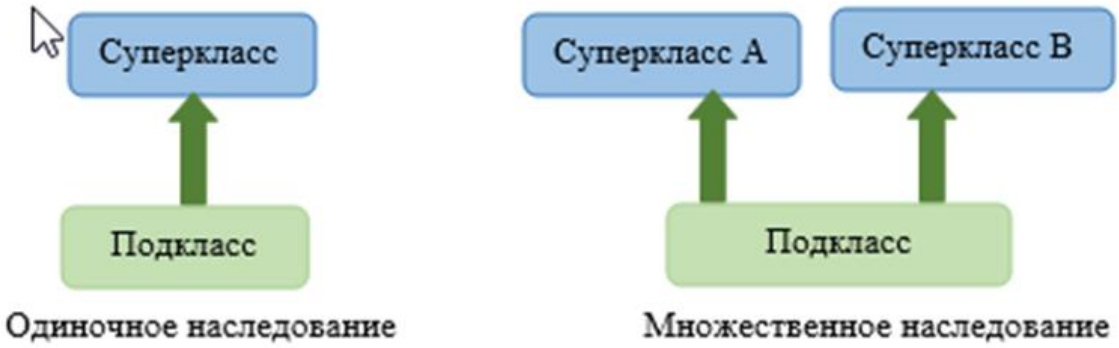

Рис. 8.1. Одиночное и множественное наследование

Пример. Класс будет служить суперклассом

```
#include <iostream>
using namespace std;
class Super {
private:
    int storage;
public:
    void put(int val)
    {
        storage = val;
    }
    int get()
    {
        return storage;
    }
};
int main()
{
    Super object;
    object.put(100);
    object.put(object.get() + 1);
    cout << object.get() << endl;
}
```

Результат: 101

Единичное наследование:

```
class Sub : Super { };
int main()
{
    Sub object;
    object.put(100);
    object.put(object.get() + 1);
    cout << object.get() << endl;
}
```

Определение класса Y как подкласса суперкласса X:

class Y: visibility_specifier X { … };

Нужно:

- поставить двоеточие после имени класса подкласса
- при необходимости разместить спецификатор видимости visibility_specifier
- добавить имя суперкласса

Если существует более одного суперкласса:

class A : X, Y, Z { … };

Определили класс Sub - производный от Super класса.

Класс Sub не вводит новых переменных и новых функций.

Означает ли это, что любой объект класса Sub наследует все черты Super класса, фактически являясь копией объектов Super класса? НЕТ

Код - ошибки компиляции, говорящие, что методы put и get недоступны. Почему?

Без спецификатора видимости компилятор предполагает, что собираемся применить private наследование: все public компоненты суперкласса становятся доступными для частного доступа, а private компоненты суперкласса вообще не будут доступны.

Не разрешается использовать последнее внутри подкласса. Нужно сообщить компилятору, что хотим сохранить ранее использованную политику доступа.

Делаем это с помощью спецификатора видимости public :

```
class Sub : public Super { };
```

Это не означает, что закрытые компоненты Super класса превратятся в общедоступные.

Закрытые компоненты останутся закрытыми, public компоненты останутся public .

Объекты класса Sub могут делать почти то же самое, что и объекты из Super класса.

"Почти", потому что принадлежность к подклассу также означает, что класс потерял доступ к частным компонентам суперкласса. Мы не можем написать функцию-член подкласса, которая могла бы напрямую управлять переменной storage .

Это серьезное ограничение. Есть ли какой-нибудь выход? Да.

Есть третий уровень доступа - “защищенный”.

Ключевое слово protected означает, что такой компонент ведет себя как public компонент при использовании любым из подклассов и выглядит как private компонент для остального мира.

Это справедливо только для public унаследованных классов (таких как Super класс в примере).

```
#include <iostream>
using namespace std;
class Super {
protected:
    int storage;
public:
    void put(int val)
    {
        storage = val;
    }
    int get()
    {
        return storage;
    }
};
class Sub : public Super {
public:
    void print()
    {
        cout << "storage = " << storage << endl;
    }
};

int main()
{
    Sub object;
    object.put(100);
    object.put(object.get() + 1);
    object.print();
}
```

Результат:

```
storage = 101
```

В примере - добавили функцию print в подкласс Sub.Подкласс Sub обращается к переменной storage из суперкласса Super. Это было бы невозможно, если бы переменная была объявлена как частная private. В main области действия функции переменная все равно остается закрытой.

Нельзя писать:

object.storage = 0;

Варианты наследования:

По типу наследования:

- Публичное – public (открытое) наследование
- Частное – private (закрытое, приватное) наследование
- Защищенное – protected наследование

По количеству базовых классов:

- Одиночное наследование (один базовый класс)
- Множественное наследование (два и более базовых класса)

Публичное public наследование - это наследование интерфейсов

При public наследовании поля и методы родительского класса остаются открытыми (public). Производный класс является подтипом родительского

Примеры: «Собака является животным», «Прямоугольник является замкнутой фигурой».

Порожденные классы - это классы, порожденные по правилам наследования.

Синтаксис:

class имя: <public/private/protected> имя базового класса

{ поля и методы порожденного класса }

Основная особенность порожденных классов – это видимость в них полей и методов базового класса, что определяется ключевыми словами: public/private/protected.

По умолчанию из базового класса наследуются все поля и методы со спецификацией public и protected, в порожденном классе они получают статус доступа public, то есть методы порожденного класса могут работать с ними напрямую.

Рис. 8.2. Наследование

В наследнике можно описывать новые поля и методы и переопределять существующие методы несколькими способами.

Если метод в потомке должен работать по-другому, чем в предке, метод описывается в потомке заново. При этом он может иметь другой набор аргументов.

Если требуется внести добавления в метод предка, то в соответствующем методе потомка наряду с описанием дополнительных действий выполняется вызов метода предка с помощью операции доступа к области видимости ::

Если в программе планируется работать одновременно с различными типами объектов иерархии или планируется добавление в иерархию новых объектов, метод объявляется как виртуальный с помощью ключевого слова

virtual. Все виртуальные методы иерархии с одним и тем же именем должны иметь одинаковый список аргументов.

Иными словами:

- private элементы базового класса в производном классе недоступны вне зависимости от ключа доступа. Обращение к ним осуществляется только через методы базового класса.
- элементы protected при наследовании с ключом private становятся в производном классе private, в остальных случаях права доступа к ним не изменяются.
- доступ к элементам public при наследовании становится соответствующим ключу доступа.

Если базовый класс наследуется с ключом private, можно выборочно сделать его некоторые элементы доступными в производном классе:

```
class Base{
   ...
   public: void f();
};
class Derived : private Base{
   ...
   public: Base::void f();
};
```

Если конструктор базового класса требует указания параметров, он должен явным образом вызван в конструкторе производного класса в списке инициализации.

Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы.

Порядок вызова конструкторов:

Если в конструкторе потомка явный вызов конструктора предка отсутствует, автоматически вызывается конструктор предка по умолчанию.

Для иерархии, состоящей из нескольких уровней, конструкторы предков вызываются, начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса.

В случае нескольких предков их конструкторы вызываются в порядке объявления.

Выводы: вызов конструкторов в C++

При конструировании экземпляра класса-наследника всегда происходит предварительный вызов конструктора базового класса

Вызов конструктора базового класса происходит до инициализации полей класса наследника

Конструктор класса-наследника может явно передать конструктору базового класса необходимые параметры при помощи списка инициализации

Если вызов конструктора родительского класса не указан явно в списке инициализации, компилятор пытается вызвать конструктор по умолчанию класса-родителя

Деструкторы не наследуется. Если деструктор в производном классе не описан, он формируется автоматически и вызывает деструкторы всех базовых классов

В деструкторе производного класса не требуется явно вызывать деструкторы базовых классов, это будет сделано автоматически. Для иерархии, состоящей из нескольких уровней, деструкторы вызываются в

порядке, строго обратному вызову конструкторов: сначала вызывается деструктор класса, затем – деструкторы элементов класса, а потом деструктор базового класса.

Порядок вызова деструкторов:

В C++ порядок вызова деструкторов всегда обратен порядку вызова конструкторов: сначала вызывается деструктор класса-наследника, затем деструктор базового класса и т.д. вверх по иерархии.

Перегрузка методов в классе-наследнике

В C++ метод производного класса замещает собой все методы родительского класса с тем же именем. Количество и типы аргументов значения не имеют. Для вызова метода родительского класса из метода класса наследника используется метод Base::

Защищенные элементы

При записи полей и методов базового класса можно использовать квалификаторы public, private, protected.

Производный класс может обращаться к public элементам базового класса, как будто они определены в базовом классе, а к private элементам базового класса нельзя обращаться напрямую, для этого в производном классе надо использовать интерфейсные функции базового класса.

Защищенные protected элементы базового класса.

Если в базовом классе элемент protected, то производный класс обращается к нему как к public, а для оставшейся части программы защищенные элементы являются частными (private), то есть обращение к ним идет через интерфейсные функции.

Если protected заменить на private, то обращение из порожденного класса к полям базового возможно только через методы.

Программист сам решает, какими делать поля базового класса private или protected.

Защищенные поля обеспечивают свободный доступ для порожденного класса и защиту всей программы, так как часть, не входящая в производный класс, может обращаться только через интерфейсные функции этого класса.

Если вы порождаете один класс из другого, то возможна ситуация, когда имя элемента в базовом и порожденном классе совпадают.

Все функции производного класса будут использовать поля производного.

Если надо базовое поле, то его определяют через квалификатор класса

имя класса :: имя поля.

Выводы:

Для порожденного класса нужен базовый класс. Для инициализации элементов производного класса вызывают конструктор базового (возможны варианты). Добавляется новый тип элементов – защищенные (protected), которые доступны напрямую и базовому, и производному классам.

Для разрешения конфликтов имен в базовом и порожденном классах используется оператор разрешения контекста (::). Явно изменить статус доступа при наследовании можно с помощью квалификаторов доступа public, private, protected. Любой компонент класса может быть объявлен как:

- публичный public

- частный private
- защищенный protected

Эти три ключевых слова также могут использоваться в совершенно другом контексте для определения модели наследования видимости.

В таблице собраны все возможные комбинации объявления компонента и модели наследования, представляя результирующий доступ к компонентам, когда подкласс полностью определен.

Таблица 7.1. читается (первая строка): если компонент объявлен общедоступным, а его класс наследуется как общедоступный, то результирующий доступ является общедоступным.

Таблица – базовый инструмент для решения проблем, связанных с наследованием компонентов класса.

Табл. 7.1. Таблица вариантов наследования

Объявление компонентов в базовом классе:	Класс наследуется как:	Уровень доступа компонентов в производном класее
public	public	public
protected		protected
private		none
public	protected	protected
protected		protected
private		none
public	private	none
protected		protected
private		public

```
#include <iostream>
using namespace std;
class SuperA {
protected:
    int storage;
public:
    void put(int val)
    {
        storage = val;
    }
    int get()
    {
        return storage;
    }
};

class SuperB {
protected:
    int safe;
public:
    void insert(int val)
    {
        safe = val;
    }
    int takeout()
    {
        return safe;
    }
};

class Sub : public SuperA, public SuperB {
public:
    void print()
    {
        cout << "storage = " << storage << endl;
        cout << "safe    = " << safe << endl;
    }
int main(void) {
    Sub object;
    object.put(1);  object.insert(2);
    object.put(object.get() + object.takeout());
    object.insert(object.get() + object.takeout());
    object.print();
    return 0;
}
storage = 3
safe    = 5
```

Совместимость типов

Каждый новый класс представляет собой новый тип данных. Каждый объект, построенный на основе такого класса, подобен значению нового типа. Это означает, что любые два объекта могут быть (или не быть) совместимы в смысле их типов.

Пример.

Объекты класса Cat несовместимы с объектами класса Dog , хотя структура обоих классов идентична. Назначения не допустимые, приведут к ошибке компилятора:

```
a_dog = a_cat;

a_cat = a_dog;
```

Классы Dog и Cat не имеют ничего общего в смысле наследования – они оба полностью независимы.

Можно сказать, что объекты, производные от классов, которые лежат в разных ветвях дерева наследования, всегда несовместимы.

Пример

```
#include <iostream>
using namespace std;
class Cat {
public:
    void make_sound()
    {
        cout << "Meow! Meow!" << endl;
    }
};
class Dog {
public:
    void make_sound()
    {
            cout << "Woof! Woof!" << endl;
    }
};


 int main()
 {
     Cat *a_cat = new Cat();
     Dog *a_dog = new Dog();

     a_cat -> make_sound();
     a_dog -> make_sound();
 }
```

```
Meow! Meow!
Woof! Woof!
```

Перестроили пример. Есть два предыдущих класса Dog and Cat, но поместили их в одно и то же дерево наследования.

Классы Dog и Cat теперь являются потомками общего базового класса Pet.

У всех классов есть конструкторы, это позволяет давать уникальные имена всем объектам, которые мы собираемся создать в будущем. Наши домашние животные также умеют бегать.

Итог - создали:

объекты, производные от класса Pet , могут запускаться

объекты, производные от классов Dog и Cat , могут бегать (они наследуют эту способность от своего суперкласса); они могут издавать звуки (этот навык недоступен для объектов класса домашних животных Pet).

То есть:

Объекты для Cat и Dog могут делать все, что могут делать домашние животные Pet

Pet не могут делать все то, что могут делать Cat и Dog

В общем:

объекты подкласса обладают, по крайней мере, теми же возможностями, что и объекты суперкласса

объекты суперкласса могут не обладать теми же возможностями, что и объекты подкласса

Вывод:

- объекты суперкласса совместимы с объектами подкласса
- объекты подкласса несовместимы с объектами суперкласса

Это означает, что можно сделать:

```
a_pet = new Dog("Huckleberry");

a_pet -> run();
```

но нельзя делать подобного: a_pet -> make_sound();

потому что Pets не умеют издавать звуки

не разрешается делать: a_dog = new Pet("Strange pet");

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; }
    void Run(void) { cout << Name << ": I'm running" << endl; }
};
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) {};
    void MakeSound(void) { cout << Name << ": Woof! Woof!" << endl; }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) {};
    void MakeSound(void) { cout << Name << ": Meow! Meow!" << endl; }
};
```

```
int main(void) {
    Pet *a pet1 = new Cat("Tom");
    Pet *a pet2 = new Dog("Spike");
    a pet1 -> Run();
    // 'a pet1 -> MakeSound();' is not allowed here!
    a pet2 -> Run();
    // 'a pet2 -> MakeSound();' is not allowed here!
    return 0;
}
```

```
Tom: I'm running
Spike: I'm running
```

Обратите внимание на то, как указатель на объекты суперкласса (указатель типа Pet *) может служить указателем на объекты подкласса.

Вопрос: почему не разрешается приказывать питомцу издавать звуки? Проблема возникает из-за статических проверок, выполняемых компилятором в процессе компиляции. Компилятор убежден, что домашние животные не могут издавать звуки и не позволит это сделать.

Для решения проблемы - использовать cast операторы.

Первый - static_cast, осуществляет явное допустимое приведение типа данных.

Синтаксис: static_cast<target_type>(an_expression)

target_type - это имя типа, которое мы хотим, чтобы компилятор использовал при оценке значения an_expression.

Например, форма: static_cast<Dog *>(a_pet) заставляет компилятор предполагать, что a_pet (временно) преобразуется в указатель типа Dog *. Это означает, что наша Dog обретает способность лаять...

Применим оператор static_cast, чтобы питомцы могли снова издавать свои звуки.

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; }
    void Run(void) { cout << Name << ": I'm running" << endl; }
};
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) {};
    void MakeSound(void) { cout << Name << ": Woof! Woof!" << endl; }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) {};
    void MakeSound(void) { cout << Name << ": Meow! Meow!" << endl; }
};
int main(void) {
    Pet *a pet1 = new Cat("Tom");
    Pet *a pet2 = new Dog("Spike");
    a pet1 -> Run();
    static cast<Cat *>(a pet1) -> MakeSound();
    a pet2 -> Run();
    static cast<Dog *>(a pet2) -> MakeSound();
    return 0;
}
```

```
int main(void) {
    Pet *a pet1 = new Cat("Tom");
    Pet *a pet2 = new Dog("Spike");
    a pet1 -> Run();
    static cast<Cat *>(a pet1) -> MakeSound();
    a pet2 -> Run();
    static cast<Dog *>(a pet2) -> MakeSound();
    return 0;
}
```

```
Tom: I'm running
Tom: Meow! Meow!
Spike: I'm running
Spike: Woof! Woof!
```

Не следует злоупотреблять возможностями оператора static_cast и использовать его бездумно.

Пример (плохой). Пытались относиться к кошке как к собаке и наоборот. Результат программы:

Spike: I'm running

Spike: Meow! Meow!

Tom: I'm running

Tom: Woof! Woof!

Этот эффект вызван тем, что компилятор не может проверить, совместим ли преобразуемый указатель с объектом, на который он указывает. Компилятор вынужден признать указатель допустимым – на самом деле у него нет другого выбора. Он просто должен верить, что мы знаем, что делаем.

Полная проверка правильности указателя возможна тогда и только тогда, когда программа выполняется (во время выполнения). В языке C++ есть второй оператор преобразования, разработанный специально для этого случая - dynamic_cast.

Преобразование выполняется динамически относительно текущего состояния всех созданных объектов. Это означает, что преобразование может (или не может) быть успешным, что приведет к остановке программы, если она захочет, чтобы какая-либо собака мяукала.

Пример.

```
#include <iostream>
using namespace std;
class Pet {
protected:
    string name;
public:
    Pet(string n)
    {
        name = n;
    }
    void run()
    {
        cout << name << ": I'm running" << endl;
    }
};
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) {};
    void make sound()
    {
        cout << name << ": Woof! Woof!" << endl;
    }
};
```

```
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) {};
    void make sound()
    {
        cout << name << ": Meow! Meow!" << endl;
    }
};
int main()
{
    Pet *a pet1 = new Cat("Tom");
…
}
```

```
Результат:

Tom: I'm running
Tom: Meow! Meow!
Spike: I'm running
Spike: Woof! Woof!
```

Правило: объекты, находящиеся на более высоких уровнях, совместимы с объектами на более низких уровнях иерархии классов, работает с произвольно длинной иерархией наследования.

Пример - иерархия из 3х классов. Указатель на верхний суперкласс работает и для нижних объектов.

```
#include <iostream>
using namespace std;
class Pet {
protected:
    string name;
public:
    Pet(string n)
    {
        name = n;
    }
    void run()
    {
        cout << name << ": I'm running" << endl;
    }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) {};
    void make sound()
    {
        cout << name << ": Meow! Meow!" << endl;
    }
};

class Persian : public Cat {
public:
    Persian(string n) : Cat(n) {};
};
int main()
{
    Pet     *a pet;
    Persian *a persian;
    a pet = a persian = new Persian("Mr. Bigglesworth");
    a persian -> make sound();
…
}
```

```
Mr. Bigglesworth: Meow! Meow!
Mr. Bigglesworth: Meow! Meow!
```

Полиморфизм и виртуальные методы

Когда подкласс объявляет метод с именем, ранее известным в его суперклассе, исходный метод переопределяется. Это означает, что подкласс скрывает предыдущее значение идентификатора метода, и любой вызов в подклассе, будет ссылаться на более новый метод.

Пример – показывает два важных аспекта переопределения.

1 - эффекты переопределения при вызове make_sound из объектов a_cat и a_dog . “Услышим” звук, соответствующий классу животных, которые их издают.

2 - эффекты переопределения могут быть отменены (аннулированы), если использовать оператор static_cast в обратном порядке.

```
#include <iostream>
using namespace std;
class Pet {
protected:
	string name;
public:
	Pet(string n)
	{
		name = n;
	}
	void make_sound()
	{
		cout << name << " the Pet says: Shh! Shh!" << endl;
	}
};

class Cat : public Pet {
public:
	Cat(string n) : Pet(n) { }
	void make_sound()
	{
		cout << name << " the Cat says: Meow! Meow!" << endl;
	}
};
```

```
class Dog : public Pet {
public:
      Dog(string n) : Pet(n) { }
      void make_sound()
      {
            cout << name << " the Dog says: Woof! Woof!" << endl;
      }
};

int main()
{
      Cat *a_cat = new Cat("Kitty");;
      Dog *a_dog = new Dog("Doggie");;

      a_cat -> make_sound();
      static_cast<Pet *>(a_cat) -> make_sound();
      a_dog -> make_sound();
      static_cast<Pet *>(a_dog) -> make_sound();
}
```

Программа-пример выведет строки (кошка и собака “запомнили” звуки, издаваемые их предшественниками):

```
Kitty the Cat says: Meow! Meow!
Kitty the Pet says: Shh! Shh!
Doggie the Dog says: Woof! Woof!
Doggie the Pet says: Shh! Shh!
```

Второй пример программы показывает другую сторону того же эффекта.

Не используем static_cast, но некоторые указатели явно объявлены как указывающие на общий суперкласс (Pet). Сами классы остались такими же. Изменили то, как мы относимся к указателям.

Ожидаемые (и полученные) эффекты почти одинаковы – код выведет на экран:

```
Kitty the Pet says: Shh! Shh!
Kitty the Cat says: Meow! Meow!
Doggie the Pet says: Shh! Shh!
Doggie the Dog says: Woof! Woof!
```

```
#include <iostream>
using namespace std;
class Pet {
protected:
	string name;
public:
	Pet(string n)
	{
		name = n;
	}
	void make_sound()
	{
		cout << name << " the Pet says: Shh! Shh!" << endl;
	}
};

class Cat : public Pet {
public:
	Cat(string n) : Pet(n) { }
	void make_sound()
	{
		cout << name << " the Cat says: Meow! Meow!" << endl;
	}
};


class Dog : public Pet {
public:
	Dog(string n) : Pet(n) { }
	void make_sound()
	{
		cout << name << " the Dog says: Woof! Woof!" << endl;
	}
};

int main()
{
	Pet *a_pet1, *a_pet2;
	Cat *a_cat;
	Dog *a_dog;

	a_pet1 = a_cat = new Cat("Kitty");
	a_pet2 = a_dog = new Dog("Doggie");
	a_pet1 -> make_sound();
	a_cat  -> make_sound();
	a_pet2 -> make_sound();
	a_dog  -> make_sound();
}
```

```
class Dog : public Pet {
public:
      Dog(string n) : Pet(n) { }
      void make_sound()
      {
            cout << name << " the Dog says: Woof! Woof!" << endl;
      }
};

int main()
{
      Pet *a_pet1, *a_pet2;
      Cat *a_cat;
      Dog *a_dog;

      a_pet1 = a_cat = new Cat("Kitty");
      a_pet2 = a_dog = new Dog("Doggie");
      a_pet1 -> make_sound();
      a_cat  -> make_sound();
      static_cast<Pet *>(a_cat) -> make_sound();
      a_pet2 -> make_sound();
      a_dog  -> make_sound();
      static_cast<Pet *>(a_dog) -> make_sound();
}
```

Программа–пример выведет несколько неожиданный фрагмент текста:

```
Kitty the Cat says: Meow! Meow!
Kitty the Cat says: Meow! Meow!
Kitty the Cat says: Meow! Meow!
Doggie the Dog says: Woof! Woof!
Doggie the Dog says: Woof! Woof!
Doggie the Dog says: Woof! Woof!
```

main функция: вызывали метод make_sound 6 раз: 2 раза для объектов класса Pet (a_dog и a_cat) и 4 раза для их суперкласса (a_pet1 и a_pet2).

Класс Pet не всегда один и тот же – любой из его подклассов может изменить свое поведение.

То есть - не можем предсказать поведение класса, содержащего виртуальную функцию, если не знаем всех его подклассов. Это также может означать, что некоторые аспекты класса определены вне класса.

Так работает полиморфизм.

Пример: привязка между источником виртуальной функции (внутри суперкласса) и ее заменой (определенной в подклассе) создается динамически во время выполнения программы.

Вызываем метод make_sound как часть конструктора Pet . Знаем, что метод полиморфно заменен новыми реализациями, представленными подклассами Cat и Dog . Пока не знаем, когда произойдет замена.

Результат:

```
Kitty the Pet says: Shh! Shh!
Doggie the Pet says: Shh! Shh!
```

Это означает, что привязка между исходными функциями и их полиморфными реализациями устанавливается при полном создании объекта подкласса, а не раньше

```
#include <iostream>
using namespace std;
class Pet {
protected:
	string name;
public:
	Pet(string n)
	{
		name = n;
		make_sound();
	}
	virtual void make_sound()
	{
		cout << name << " the Pet says: Shh! Shh!" << endl;
	}
};

class Cat : public Pet {
public:
	Cat(string n): Pet(n) { }
	void make_sound()
	{
		cout << name << " the Cat says: Meow! Meow!" << endl;
	}
};
```

```
class Dog : public Pet {
public:
      Dog(string n) : Pet(n) { }
      void make_sound()
      {
            cout << name << " the Dog says: Woof! Woof!" << endl;
      }
};

int main()
{
      Cat *a_cat = new Cat("Kitty");
      Dog *a_dog = new Dog("Doggie");

      a_cat -> make_sound();
      a_dog -> make_sound();
}
```

Виртуальный метод может быть вызван не только извне класса, но и изнутри.

Пример.

Добавили новый метод в класс Pet для выполнения всех действий, выполняемых домашним животным, когда оно просыпается. Предположили, что тогда каждый питомец должен издавать звук.

Метод make_sound вызывается косвенно, через метод wake_up , который определяется один раз в суперклассе и больше нигде не переопределяется.

Результат:

```
Kitty the Cat says: Meow! Meow!
Doggie the Dog says: Woof! Woof!
```

Объекты как параметры и динамическое приведение

Любой объект может использоваться в качестве параметра функции и, наоборот, любая функция может иметь параметр в качестве объекта любого класса.

Пример - два способа передачи объекта в функцию: по указателю и по ссылке.

Передача указателя - это просто определенная форма передачи значения (позволяет изменять состояние объекта внутри функции).

В примере две функции с двумя параметрами. Первая функция объявляет указатель, вторая - ссылку. Целевой класс один и тот же: “ Pet ”.

Создаем два объекта в main: p1 обрабатывается указателем и создается с помощью ключевого слова new. p2 является автоматической переменной.

Вызываем каждую из функций дважды, 1-ый раз в соответствии с их объявлениями, 2-ой раз, заменяя способ обработки объектов: p2 подчиняется оператору &; p1 разыменован оператором *.

```
#include <iostream>
using namespace std;
class Pet {
protected:
	string name;
public:
	void name_me(string name)
	{
		this -> name = name;
	}
	void make_sound()
	{
		cout << name << " says: no comments" << endl;
	}
};

void play_with_pet_by_pointer(string name, Pet *pet)
{
	pet -> name_me(name);
	pet -> make_sound();
}
```

```
#include <iostream>
using namespace std;
class Pet {
protected:
	string name;
public:
	void name_me(string name)
	{
		this -> name = name;
	}
	void make_sound()
	{
		cout << name << " says: no comments" << endl;
	}
};

void play_with_pet_by_pointer(string name, Pet *pet)
{
	pet -> name_me(name);
	pet -> make_sound();
}
```

Результат:

```
anonymous says: no comments
no_name says: no comments
no_name says: no comments
anonymous says: no comments
```

Объекты могут передаваться по значению как обычные значения базовых встроенных типов (int, float). Последствия любых операций, влияющих на объект, будут происходить с копией объекта, а не с самим объектом.

Пример - три возможных варианта использования объекта в функции. Все три функции пытаются «назвать» свой параметр (вызвать его метод с именем name_me()).

После завершения вызова функции активируется метод make_sound для проверки - было ли присвоение имени успешным и может ли оно наблюдаться вне функции.

Результаты: присвоение объекту “ Alpha ” повлияло на копию объекта pet, а не на сам объект.

Неудобство возникает, когда передаем по значению объект подкласса класса, указанного в качестве типа параметра. Это может привести к “исчезновению” части объекта. Передача объекта по значению, как правило, не является хорошей идеей.

Результат:

```
no_name says: no comments
Beta says: no comments
Gamma says: no comments
```

```
#include <iostream>
using namespace std;
class Pet {
protected:
	string name;
public:
	void name_me(string name)
	{
		this -> name = name;
	}
	void make_sound()
	{
		cout << name << " says: no comments" << endl;
	}
};

void name_pet_by_value(string name, Pet pet)
{
	pet.name_me(name);
}
```

```
#include <iostream>
using namespace std;
class Pet {
protected:
	string name;
public:
	Pet(string n)
	{
		name = n;
	}
	virtual void make_sound()
	{
		cout << name << " the Pet says: Shh! Shh!" << endl;
	}
	void wake_up()
	{
		make_sound();
	}
};

class Cat : public Pet {
public:
	Cat(string n) : Pet(n) { }
	void make_sound()
	{
		cout << name << " the Cat says: Meow! Meow!" << endl;
	}
};




class Dog : public Pet {
public:
	Dog(string n) : Pet(n) { }
	void make_sound()
	{
		cout << name << " the Dog says: Woof! Woof!" << endl;
	}
};

int main()
{
	Cat *a_cat = new Cat("Kitty");
	Dog *a_dog = new Dog("Doggie");

	a_cat -> wake_up();
	a_dog -> wake_up();
}
```

```
void name_pet_by_pointer(string name, Pet *pet)
{
      pet -> name_me(name);
}

void name_pet_by_reference(string name, Pet &pet)
{
      pet.name_me(name);
}

int main()
{
      Pet pet;
      pet.name_me("no_name");
      name_pet_by_value("Alpha", pet);
      pet.make_sound();
      name_pet_by_pointer("Beta", &pet);
      pet.make_sound();
      name_pet_by_reference("Gamma", pet);
      pet.make_sound();
}
```

Более сложная иерархия классов.

Существует функция play_with_pet, готовая получить один параметр типа Pet & (ссылка на объект Pet). Функция вызывается с четырьмя объектами, взятыми с разных уровней иерархии классов.

Это возможно, потому что объект суперкласса совместим по типу с объектами любого из подклассов. Можно объявить формальный параметр типа суперкласс и передать фактический параметр любого из подклассов формального параметра.

Можете ли предсказать результат? Является ли функция make_sound виртуальной?, т.е. имеем ли мы дело с полиморфизмом или нет?

Нет. Ожидаемый результат:

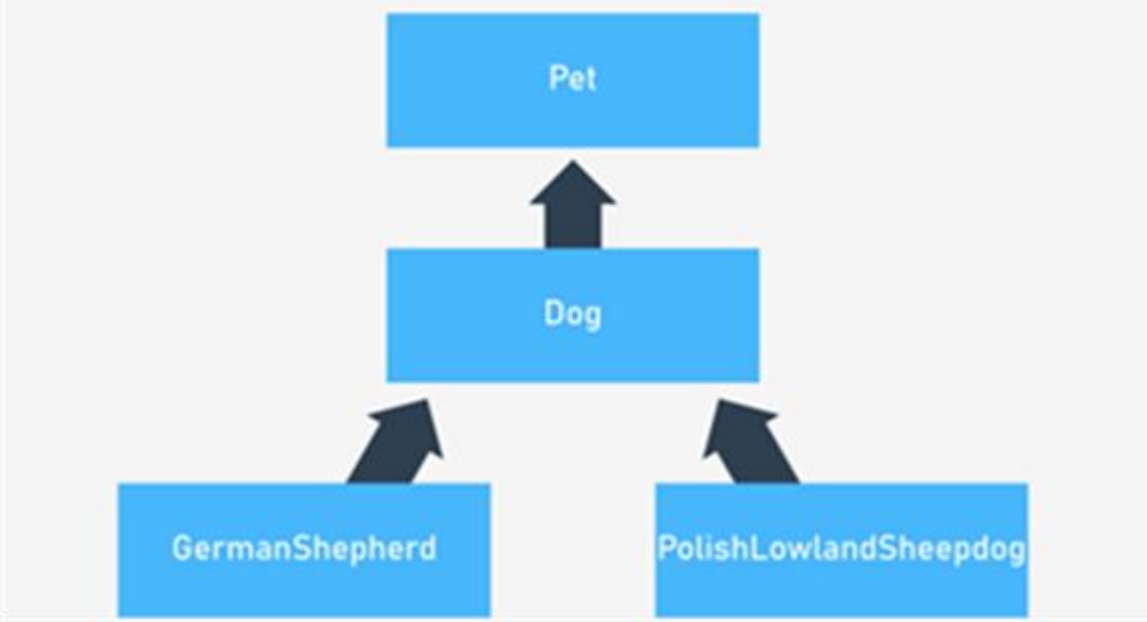

Рис. 8.3. Иерархия классов

```
creature is silent :(
Dog is silent :(
Hund is silent :(
Perro is silent :(
```

```
#include <iostream>
using namespace std;
class Pet {
protected:
	string name;
public:
	Pet(string name)
	{
		this -> name = name;
	}
	void make_sound()
	{
		cout << name << " is silent :(" << endl;
	}
};
class Dog : public Pet {
public:
	Dog(string name) : Pet(name) {}
	void make_sound()
	{
		cout << name << " says: Woof!" << endl;
	}
};
```

```
class GermanShepherd : public Dog {
public:
	GermanShepherd(string name) : Dog(name) {}
	void make_sound()
	{
		cout << name << " says: Wuff!" << endl;
	}
};
class MastinEspanol : public Dog {
public:
	MastinEspanol(string name) : Dog(name) {}
	void make_sound()
	{
		cout << name << " says: Guau!" << endl;
	}
};
void play_with_pet(Pet &pet)
{
	pet.make_sound();
}

int main()
{
	Pet pet("creature");
	Dog dog("Dog");
	GermanShepherd shepherd("Hund");
	MastinEspanol mastin("Perro");
	play_with_pet(pet);
	play_with_pet(dog);
	play_with_pet(shepherd);
	play_with_pet(mastin);
}
```

Изменили предыдущую программу. Поведение не изменилось, результат точно такой же.

Изменили интерфейс функции play_with_pet().

Теперь функция принимает указатель, а не ссылку. Изменили основную функцию.

Результат:

```
. . .
void play_with_pet(Pet *pet)
{
      pet -> make_sound();
}

int main()
{
      Pet *pet                          = new Pet("creature");
      Dog *dog                          = new Dog("Dog");
      GermanShepherd *shepherd = new GermanShepherd("Hund");
      MastinEspanol  *mastin       = new MastinEspanol("Perro");

      play_with_pet(pet);
      play_with_pet(dog);
      play_with_pet(shepherd);
      play_with_pet(mastin);
}
```

```
creature is silent :(
Dog is silent :(
Hund is silent :(
Perro is silent :(
```

Оператор dynamic_cast

Реорганизовали наш «питомник» - пример в файле 1. Изменения:

Изменили метод make_sound внутри класса верхнего уровня – теперь он virtual. Сделали дерево классов глубже - добавили два дополнительных уровня. Самый низкий уровень состоит из двух ветвей, содержащих специфические виды: German Shepherd и Mastin Espanol (немецкая овчарка и испанский мастифф). Эти собаки лают на своих родных языках и также «бегают» на своих языках. Методы make_sound и два новых метода: laufen и ejecutar.

Функция play_with_pet получает указатель типа Pet * (указатель на объекты класса верхнего уровня).

Эта функция:

- вызывает virtual метод make_sound; поэтому объекты смогут издавать свои собственные звуки;
- пытается распознать природу полученного указателя и заставить указанный объект вести себя в соответствии с его происхождением – это момент, когда оператор dynamic_cast становится незаменимым

Правило:

Если оператор dynamic_cast используется следующим образом:

dynamic_cast<pointer_type>(pointer_to_object)

и преобразование pointer_to_object в тип pointer_type возможно, тогда результатом преобразования будет новый указатель, который можно использовать.

В противном случае результат преобразования равен noneptr.

Этот механизм удобно позволяет распознавать природу указателя.

GermanShepherd *shepherd = dynamic_cast<GermanShepherd *>(pet);

if(shepherd != nullptr)

shepherd -> laufen();

Если указатель pet идентифицирует объект класса GermanShepherd или любого из его подклассов, используем преобразованный указатель, хранящийся в переменной shepherd. В противном случае мы ничего не делаем.

Проанализируйте этот пример, скомпилируйте и запустите.

Изменим программу.

Функция play_with_pet имеет не указатель, а ссылку.

Две части программы также изменены:

• функция main вызывает play_with_pet другим способом;

• форма использования dynamic_cast другая:

dynamic_cast<pointer_type>(pointer_to_object)

```
void play_with_pet(Pet &pet)
{
    pet.make_sound();
    dynamic_cast<GermanShepherd &>(pet).laufen();
    dynamic_cast<MastinEspanol &>(pet).ejecutar();
}
```

```
int main()
{
    Pet pet("creature");
    Dog dog("Dog");
    GermanShepherd shepherd("Hund");
    MastinEspanol mastin("Perro");
. . .
}
```

Если оператор dynamic_cast не сможет выполнить свою задачу, то программа может завершиться с ошибкой выполнения.

Есть способ защитить себя от последствий неудач – использование try-catch (в следующей лк) Используя его можно “попробовать” некоторые рискованные действия и “поймать” ошибки, которые появляются во время выполнения.

В примере улавливаем ошибку и ничего не делаем, возникающая ошибка просто игнорируется

```
void play_with_pet(Pet &pet) {
    pet.make_sound();
    try {
        dynamic_cast<GermanShepherd &>(pet).laufen();
    }
    catch(...) {}
    try {
        dynamic_cast<MastinEspanol &>(pet).ejecutar();
    }
    catch(...) {}
}
```

Конструктор копирования - это особая форма конструктора, предназначенная для создания более или менее буквальной копии объекта. Вы можете узнать этот Конструктор можно узнать по его различимому заголовку.

Конструктор копирования класса A будет объявлен: A(A &)

Конструктор ожидает, что один параметр будет ссылкой на объект, содержимое которого предназначено для копирования во вновь созданный объект.

Нет никаких обязательств объявлять свой собственный конструктор копирования в любом из классов. Если в классе нет явного конструктора копирования, вместо него будет использоваться неявный конструктор. Неявный конструктор просто копирует (бит за битом) исходный объект, создавая его двойную копию.

В большинстве случаев он работает удовлетворительно, но требует внимания при использовании с объектами, которые содержат другие объекты или указатели.

Пример.

Класс Class не имеет явного конструктора копирования. Будет использоваться неявный конструктор. Он будет активирован дважды – во время инициализации объектов o2 и o3.

Пример напоминает о двух возможных способах указания объявления объекта: используя оператор = в классической нотации и используя функциональную нотацию. Объекты o2 и o3 будут созданы как двойные копии объектов o2 и o3 соответственно, но эти три объекта не имеют ничего общего.

У каждого из них есть свое поле data. Все три объекта живут своей собственной жизнью. Приращение, примененное к объекту o1, не влияет на o2 и o3.

```
#include <iostream>
using namespace std;
class Class {
    int data;
public:
    Class(int value) : data(value) {}
    void increment()
    {
        data++;
    }
    int value()
    {
        return data;
    }
};
int main()
{
    Class o1(123);
    Class o2 = o1;
    Class o3(o2);

    o1.increment();
    cout << o1.value() << endl;
    cout << o2.value() << endl;
    cout << o3.value() << endl;
}
```

Результат:

```
124
123
123
```

Конструктор неявного копирования создает двойную копию объекта.

Это относится к объекту, а не к сущностям, существующим вне объекта. Это означает, что:

- поле data будет скопировано во вновь созданный объект, но фактически оно будет указывать на тот же фрагмент памяти.

- у разных объектов может быть что-то общее - они могут делиться данными между собой.

Нельзя сказать, что это всегда неправильно и означает проблемы. Если вы делаете это намеренно, это может быть полезно и удобно. Но если вы сделали это случайно, это может быть сложно и трудно диагностировать.

Пусть: не хотим, чтобы объекты обменивались данными. Хотим изолировать объекты и держать их все отдельно.

Добавим в класс конструктора явного копирования.

Конструктор будет нести ответственность за создание копии самостоятельно.

Никакие автоматические действия не будут выполняться, если в классе есть конструктор копирования.

```
#include <iostream>
using namespace std;
class Class {
    int *data;
public:
    Class(int value) {
        data = new int;
       *data = value;
    }
    Class(Class &source) {
        data = new int;
       *data = source.value();
    }
    void increment(void) {  (*data)++; }
    int value(void) { return *data; }
};
int main(void) {
    Class o1(123);
    Class o2 = o1;
    Class o3(o2);
    o1.increment();
    cout << o1.value() << endl;
    cout << o2.value() << endl;
    cout << o3.value() << endl;
    return 0;
}
```

Конструктор копирования выделил новый фрагмент памяти и скопировал в него исходное содержимое данных.

Это гарантирует, что объект будет отдельным и не будет иметь общих частей.

Код выведет на экран строки:

```
124
123
123
```

Композиции

Наследование - не единственная форма сосуществования класса и объекта.

Наследование совершенно бесполезно, когда мы выражаем многие сложные отношения, существующие в реальном мире.

Например, можно представить класс, предназначенный для сбора разных видов собак. Создание класса и размещение его в нижней части дерева наследования породы собак не является хорошей идеей. Реальная роль класса – он должен собрать несколько собак, по сути, он будет состоять из собак (как питомник).

Это означает, что класс должен включать собак, а не наследовать их. Питомник имеет очень мало общего с собакой (например, питомник не может лаять).

Любая сложная структура состоит из более простых элементов (пример: автомобиль состоит из двигателя, подвески и т.д.). Если представить эти части как классы, увидим класс car как композицию, которая не имеет ничего общего с наследованием.

Построение класса называется композицией.

Пример - простой фиктивный класс Compo, состоящий из двух объектов двух независимых классов.

```
#include <iostream>
using namespace std;
class A {
public:
    void do it()
    {
        cout << "A is doing something" << endl;
    }
};
class B {
public:
    void do it()
    {
        cout << "B is doing something" << endl;
    }
};
class Compo {
public:
    A f1;
    B f2;
};
int main()
{
    Compo  co;
    co.f1.do it();
    co.f2.do it();
}
```

Результат:

```
A is doing something
B is doing something
```

Пример - тот же класс, но его компоненты изменены и содержат конструкторы копий. Они только выдают сообщение, что конструктор был вызван. Добавлены конструкторы по умолчанию. У класса Compo нет конструктора копирования. Это означает, что компилятор создаст неявный конструктор копирования для класса. Этот конструктор копирует объекты по частям.

Скомпилировали и запустили код, результат:

```
copying A...
copying B...
A is doing something
B is doing something
```

Это означает, что, помимо своей обычной деятельности (клонирования), конструктор копирования учитывает все существующие конструкторы копирования (неявные и явные), определенные в объектах, используемых для создания класса. Это дает возможность скопировать компоненты наиболее подходящим способом.

```
#include <iostream>
using namespace std;
class A {
public:
    A(A &src) { cout << "copying A..." << endl; }
    A(void) { }
    void Do(void) { cout << "A is doing something" << endl; }
};
class B {
public:
    B(B &src) { cout << "copying B..." << endl; }
    B(void){ }
    void Do(void) { cout << "B is doing something" << endl; }
};
class Compo {
public:
    Compo(void) { } ;
    A f1;
    B f2;
```

```
int main(void) {
    Compo  co1;
    Compo  co2 = co1;
    co2.f1.Do();
    co2.f2.Do();
    return 0;
}
```

```
copying A...
copying B...
A is doing something
B is doing something
```

Пример – изменили класс Compo. Теперь у него есть свой собственный конструктор копирования. Он будет неявно вызываться 1 раз в течение срока действия программы в момент создания объекта co2.

Программа выдает результат:

```
Copying Compo...
A is doing something
B is doing something
```

Конструктор явного копирования (написанный нами) не вызвал ни одного из конструкторов копирования компонента. Определение собственного конструктора копирования означает, что вы берете на себя полную ответственность за все действия, необходимые для выполнения ответственного копирования. Это означает, что нужно изменить конструктор. Один из способов сделать это - добавить подобную строку:

```
Compo(Compo &src) : f1(src.f1), f2(src.f2)
{
    cout << "Copying Compo..." << endl;
}
```

Вместо:

```
Compo(Compo &src)
{
    cout << "Copying Compo..." << endl;
}
```

Измененная программа:

```
#include <iostream>
using namespace std;
class A {
public:
    A(A &src) { cout << "copying A..." << endl; }
    A(void) { }
    void Do(void) { cout << "A is doing something" << endl; }
};
class B {
public:
    B(B &src) { cout << "copying B..." << endl; }
    B(void){ }
    void Do(void) { cout << "B is doing something" << endl; }
};
class Compo {
public:
    Compo(Compo &src) { cout << "Copying Compo..." << endl; }
    Compo(void) { } ;
    A f1;
    B f2;
};
int main(void) {
    Compo  co1;
    Compo  co2 = co1;
    co2.f1.Do();
    co2.f2.Do();
    return 0;
}
```

```
copying A...
copying B...
Copying Compo...
A is doing something
B is doing something
```

«Друзья» в C++

Друг класса может быть:

- класс class (friend class)
- функция function (friend function - дружественная функция)

Друзьям разрешается получать доступ к закрытым и защищенным компонентам класса или использовать их.

Дружба в этом смысле подобна разрешению и выражению доверия.

Если есть два класса A и B, и если A хочет, чтобы B мог получить доступ к своим private компонентам, он должен объявить, что B является его другом. Объявление работает только в одном направлении. Класс B не может просто сказать: “Я друг A” – это не сработает. (Это точно так же, как в настоящем мире. Наши друзья - это те, кого мы называем друзьями. Люди, утверждающие, что они наши друзья, могут быть лжецами.)

```
#include <iostream>
using namespace std;
class Class {
friend class Friend;
private:
    int field;
    void print()
    {
        cout << "It's a secret, that field = " << field << endl;
    }
};
class Friend {
public:
    void doit(Class &c)
    {
        c.field = 100;
        c.print();
    }
};
int main()
{
    Class o;
    Friend f;

    f.doit(o);
}
```

Класс с именем Class объявил, что Friend является его другом.

Объект класса Friend способен манипулировать private частными полями Class и вызывать его private методы.

Не имеет значения, где добавлять объявление о дружбе, строка:

friend class ... ;

может существовать внутри любой из частей класса (общедоступной, частной или защищенной), но должна быть размещена вне любой функции или агрегата.

Программа выводит:

```
It's a secret, that field = 100
```

Дополнительные правила, которые необходимо учитывать:

- класс может быть другом многих классов
- в классе может быть много друзей
- друг моего друга - это не мой друг
- дружба не наследуется – подкласс должен определять свои собственные дружеские отношения

Функция также может быть другом класса. Такая функция может получить доступ ко всем закрытым и/или защищенным компонентам класса.

Правила:

объявление о дружбе должно содержать полный прототип функции друга (включая тип возвращаемого значения); функция с тем же именем, но несовместимая в смысле соответствия параметров, не будет признана другом;

- у класса может быть много дружественных функций;
- функция может быть другом многих классов;
- класс может распознавать в качестве друзей глобальные функции и функции-члены;

Пример. В классе " A " есть три друга:

класс B

глобальная функция DoIt()

функция-член dec() (из класса C)

Строка class A; ничего не делает, кроме как сообщает компилятору, что класс с именем A будет использоваться. Отсутствие строки приведет к ошибкам компиляции, так как компилятор не будет знать о существовании класса A при анализе тела класса C.

```
#include <iostream>
using namespace std;
class A;
class C {
public:
    void dec(A &a);
};
class A {
friend class B;
friend void C::dec(A&);
friend void DoIt(A&);
private:
    int field;
protected:
    void print(void) { cout << "It's a secret, that field = " << field << endl; }
};
void C::dec(A &a) { a.field--; }
class B {
public:
    void DoIt(A &a) { a.print(); }
};
void DoIt(A &a) {
    a.field = 99;
}
int main(void) {
    A a; B b; C c;

    DoIt(a);
    b.DoIt(a);
    return 0;
}
It's a secret, that field = 99
```

Метод класса может быть объявлен виртуальным, если допускается его альтернативная реализация в порожденном классе.

При вызове виртуальной функции через указатель или ссылку на объект базового класса будет вызвана реализация данной функции, специфичная для фактического типа объекта.

Виртуальные функции обозначаются в объявлении класса при помощи слова virtual.

Виртуальные функции позволяют использовать полиморфизм - позволяет осуществлять работу с разными реализациями через один и тот же интерфейс.

В C++ функции, объявленные в базовом классе виртуальными, остаются виртуальными в классах-потомках.

Использовать слово virtual в классах – наследниках необязательно (но желательно).

В C++ виртуальные функции не являются виртуальными, если они вызваны в конструкторе или деструкторе данного класса. Такое поведение специфично для механизма инициализации и разрушения объектов C++ (в других ЯП м.б. по-другому).

Виртуальный деструктор

Деструктор класса, имеющего наследника, всегда должен явно объявляться виртуальным. Это обеспечивает корректный вызов деструктора нужного класса при вызове оператора delete с указателем на базовый класс.

Деструктор, не объявленный явно виртуальным, а также автоматически сгенерированный деструктор, является не виртуальным.

Классы без виртуальных деструкторов не предназначены для расширения.

Всегда используем виртуальный деструктор:

- в базовых классах

- в классах, от которых возможно наследование в будущем (в классах с виртуальными методами)

Не используем виртуальные деструкторы в классах, от которых не планируется создавать производные классы в будущем

В базовом классе возможно объявить защищенный (protected) невиртуальный деструктор объекты данного класса удалить напрямую невозможно – только через указатель на класс-наследник данный деструктор будет доступен классам – наследникам.