

Лекция 3

Множественное наследование

Множественное наследование

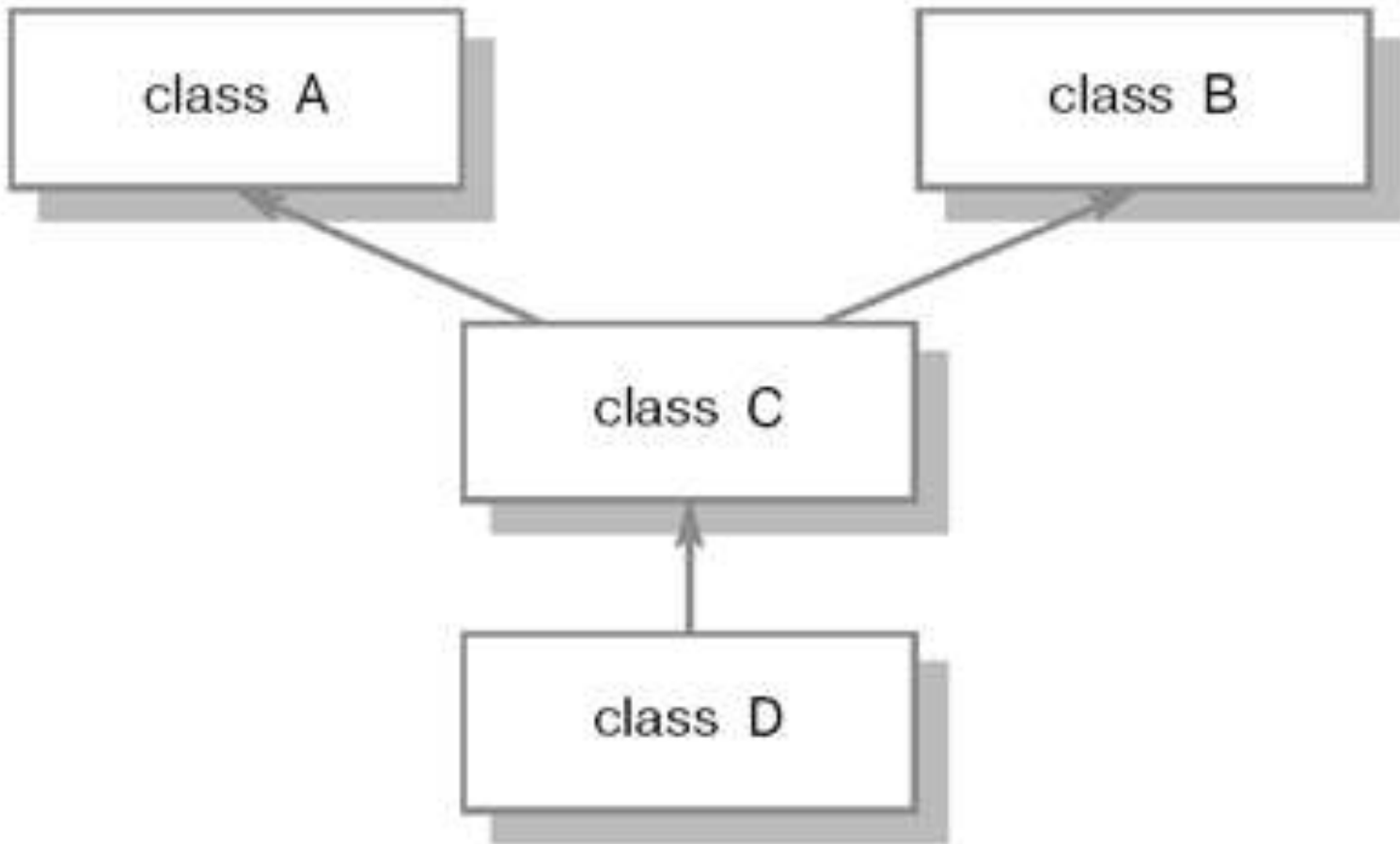
Множественное наследование представляет собой такое наследование, при котором создание производного класса основывается на использовании нескольких непосредственных базовых классов.

Множественное наследование должно выражать отношение *is-a* (**“является объектом”**).

Это механизм наследования, характерный для C++. Во многих других объектно-ориентированных языках (Delphi, Java) он отсутствует.

Множественное наследование – мощное средство C++.

Иерархия классов при множественном наследовании



Рекомендуется

Используйте

множественное наследование в тех случаях, когда в классе необходимо применять поля и методы, объявленные в разных классах

Не рекомендуется

Не используйте

множественное наследование в тех случаях, когда можно обойтись одиночным наследованием

Формат описания класса при множественном наследовании

```
class имя_производного_класса:  
    [ключ доступа] имя_базового_класса_1,  
        ...,  
    [ключ доступа] имя_базового_класса_N  
{  
    тело класса  
};
```

Пример:

```
class A { ... };  
class B { ... };  
class C { ... };  
class D: A, protected B, public C  
    { ... };
```


Конструкторы при множественном наследовании

- ▶ При определении конструктора производного класса должны вызываться конструкторы ВСЕХ непосредственных базовых классов с передачей им необходимых параметров.
- ▶ Конструктор без параметров требует вызова также конструкторов без параметров.
- ▶ Если создается иерархия классов, то при вызове конструктора производного класса вызываются также и конструкторы наследуемых классов (классов (последовательно снизу вверх по иерархии)).

Порядок выполнения конструкторов

1. Инициализируются (в порядке объявления) базовые классы.
2. Инициализируются (в порядке объявления) члены класса.
3. Выполняется тело конструктора.

Деструкторы вызываются в порядке, *обратном* выполнению конструкторов.



Проблемы множественного наследования

- 1) Возможный конфликт имен методов или полей нескольких базовых классов.

Для устранения возможной неоднозначности в использовании имен, одинаковых в базовых классах, используется обычная операция расширения области видимости (::)

Проблемы множественного наследования

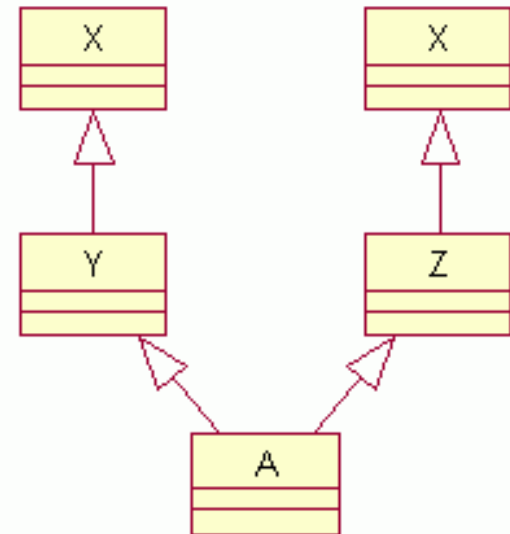
Пример:

```
class Tc1{  
  public:  
    int a;  
  
  ...  
};  
  
class Tc2{  
  public:  
    float a;  
  
  ...  
};  
  
class Tc3: public Tc1, public Tc2 {...};  
void main() {  
    Tc3 c3;  
    cout << c3.c2::a;  
}
```

Проблемы множественного наследования

2) Несколько базовых классов в свою очередь наследуются от одного и того же класса-прародителя (*ромбовидное наследование*).

```
class X {....};  
class Y : public X {...};  
class Z : public X {...};  
class A : public Y, public Z  
        {...};
```

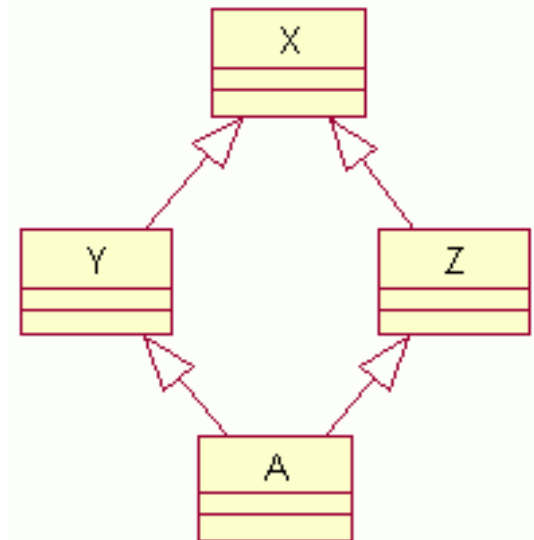


Проблемы множественного наследования

Разрешение неоднозначности:

для устранения дублирования объектов
непрямого базового класса при
множественном наследовании, этот
базовый класс объявляют *виртуальным*
(virtual) :

```
class X {...};  
class Y : virtual public X {...};  
class Z : public virtual X {...};  
class A : public Y, public Z  
    {...};
```



Новые правила конструктора для виртуальных классов

1) Обеспечение уникальности заполнения полей класса-прародителя, пример:

```
class TEmployee {  
    static int ID_Repository;  
    const int ID;  
    static int NewID() {  
        return (ID_Repository++);  
    }  
public:  
    TEmployee():ID(NewID()) {};  
};  
static int TEmployee::ID_Repository = 10000;
```

Новые правила конструктора для виртуальных классов

```
class TTemporary: virtual public TEmployee {  
    int ContractEnd;  
public:  
    TTemporary(): ContractEnd(1000) {};  
};
```

```
class TDriver: virtual public TEmployee {  
    char LicenceType [6];  
public:  
    TDriver() {strcpy(LicenceType, "B");}  
};
```

```
class TTempDriver: public TTemporary, TDriver{...};
```

Новые правила конструктора для виртуальных классов

2) По возможности виртуальные классы должны создаваться с помощью конструкторов без параметров, что позволяет избежать многих проблем, возникающих при ромбовидном наследовании.

При наличии конструкторов с параметрами может возникнуть конфликт при инициализации полей.

Пример:

```
class X {
    int w;
public:
    X() { w=0; }
    X(int w_) : w(w_) {}
};

class Y: public virtual X {
    char c ;
public:
    Y(char c_,int w_) : X(w_), c(c_) {}
};

class Z: public virtual X {
    float d ;
public:
    Z(float d_,int w_) : X(w_), d(d_) {}
};

class A: public Y, public Z {
public:
    A(char c_,float d_,int w_) :
        Y(c_,w_),Z(d_,w_) {}
};
```

В конструкторе класса А вызываются конструкторы для непосредственных базовых классов Y и Z. Эти конструкторы должны передавать информацию своим базовым классам (поле w класса X).

Возникает неопределенность: из какого класса (Y или Z) будет формироваться w?

Во избежание этого потенциального конфликта C++ отключает автоматическую передачу информации через промежуточный класс к базовому классу, если базовый класс виртуален.

Как же тогда определяется значение w?

При создании объекта класса А должен быть создан сначала компонент базового объекта (класса Х), поэтому вызывается конструктор базового класса Х, определяемый по умолчанию.

Если в базовом классе Х отсутствует конструктор по умолчанию (без параметров)—будет ошибка компиляции.

Новые правила конструктора для виртуальных классов

- 3) Если класс имеет косвенный виртуальный базовый класс и для него желательно использовать конструктор с параметрами, то нужно вызвать базовый конструктор ЯВНО:

```
class A: public Y, public Z {  
public:  
    A(char c_, float d_, int w_) :  
        X(w_), Y(c_, w_), Z(d_, w_) {}  
};
```

Для неvirtуальных базовых классов это
ЗАПРЕЩЕНО!

Вложенные классы

Определение

Класс, объявленный внутри другого класса, называется ***ВЛОЖЕННЫМ КЛАССОМ***.

```
class A {  
    public:  
        A();  
        ~A();  
        class B {    // Вложенный класс  
            B() {};  
            ~B() {};  
            ...  
        };  
    ...  
};
```

Определение

Реализация вложения классов
НЕЭКВИВАЛЕНТНА выполнению включения
has-a (“имеет объект”).

Примеры включения:

```
class X { ... };
```

1) объект

```
class A
{
    X x;
    ...
};
```

2) указатель или ссылка

```
class A
{
    X *p;
    X &r;
    ...
};
```

Определение

Включение – отношение между *объектами*.

Вложение классов представляет собой отношение между *классами*.

Вложение определяет *тип*, который известен *локально* классу, содержащему вложенное объявление класса.

Методы класса, содержащего объявление вложенного класса, могут создавать и использовать объекты вложенного класса.


Преимущества вложения

- 1) управление именами;
- 2) управление доступом.

Доступ к компонентам вложенного класса возможен с помощью оператора определения диапазона доступа (::).

В примере имя В вложено внутрь класса А, оно известно исключительно как А::В и не конфликтует с другими классами, имеющими имя А.

Особенности реализации вложенных классов

- 1) Во вложенных классах можно определить статические методы и статические переменные.
 - 2) Нестатические методы и переменные доступны либо через диапазон доступа (::), либо через объект вложенного класса, созданного внутри внешнего.
 - 3) Методы вложенного класса определяются только внутри класса.
- 

Пример 1.

```
class OuterClass {
public:
    class InnerClass {
public:
        static double d; //статические переменные
        void pr() //внутренний метод
        { cout<<d<<endl; }
    }; //конец объявления вложенного класса

    void pr1() {
        InnerClass::pr(); //обращение к методу вложенного класса
    } // через ::
    InnerClass c;
    void pr2() {
        c.pr(); //обращение к методу вложенного класса через объект
    }
}; // конец объявления внешнего класса
double OuterClass::InnerClass::d=5.32; // определение статической
// переменной

void main() {
    OuterClass oc;
    oc.c.pr(); // нестатический член вложенного класса
    cout << OuterClass::InnerClass::d <<endl; // статический член
    // вложенного класса
}
```

Особенности реализации вложенных классов


- 4) Внутри вложенных классов можно использовать следующие компоненты внешнего класса:
 - – статические (**static**) переменные;
 - – внешние (**extern**) переменные,
 - – внешние функции и элементы перечислений.
- 5) Вложенный класс не имеет никакого особого права доступа к членам внешнего класса, т.е. он может обращаться к ним только через объект этого класса (так же как и внешний класс не имеет каких-либо особых прав доступа к вложенному классу).
- 6) В то же время нельзя использовать автоматические переменные из указанной области.

Пример 2.

```
class Global {  
public:  
    int i;  
    static float f;  
    class Internal {  
        void func(Global &glob)  
        {  
            i = 3;           // Ошибка: используется имя нестатического  
                             // данного из внешнего класса  
            f = 3.5;         // Правильно: f - статическая переменная  
            ::i = 5;         // Правильно: i - внешняя (по отношению к  
                             // классу) переменная  
            glob.i = 3;      // Правильно: обращение к членам внешнего  
                             // класса через объект этого класса  
            n = 7;           // Ошибка: обращение к private-члену  
                             // внешнего класса  
        }  
    };  
protected:  
    static int n;  
};
```

Вложенные классы и доступ

Где и как вложенный класс может использоваться, зависит от:

- диапазона доступа – берется у внешнего класса, и определяет, насколько доступен вложенный класс, т.е. устанавливает, какие части программы могут создавать объекты вложенного класса;
 - управления доступом – берется у вложенного класса и регламентирует доступ к элементам вложенного класса по тем же самым правилам, которые регулируют доступ к обычному классу.
- 

Свойства диапазонов доступа вложенного класса

Место объявления во вложенном классе	Доступность для вложенного класса	Доступность для классов, производных от вложенного	Доступность для внешнего мира
private	да	нет	нет
protected	да	да	нет
public	да	да	да

Пример

```
class LinkedList {
public:
    class Iterator // вложенный класс
    {
    public:
        void insert( int x );
        int erase();
        ...
    };
    ...
private:
    class Link // вложенный класс
    {
    public:
        Link *next;
        int data;
    };
    ... };
```

Вложенные классы и наследование

Ограничений в наследовании вложенных классов НЕТ:

- внешний класс может наследовать от вложенного класса;
- вложенный класс может наследовать от вложенного класса.

Нужно следить только за видимостью базового класса в точке наследования.

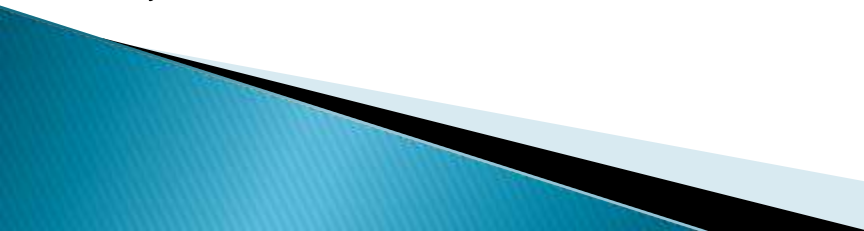
Пример

```
class A {}; // внешний класс

class B {
public:
    class C: public A {}; // вложенный класс наследует
}; // от внешнего

class D: public B::C {}; // внешний класс наследует
// от вложенного

class E {
    class F: public B::C {}; // вложенный класс
    // наследует от вложенного
    class G: public F {}; // вложенный класс наследует
    // от вложенного
};
```



Иерархия исключений

Использование классов для обработки исключений

Исключения – это механизм обработки ошибок с использованием систематического, объектно-ориентированного подхода.

Причиной исключительных ситуаций могут быть как ошибки в программе, так и неправильные действия пользователя, неверные данные и др.

Особый интерес представляют исключения, сгенерированные самим пользователем.

При генерации собственных исключений имеет смысл создать специальный класс для работы с ними.



Пример


```
class TMyException {
    char* msg;
    void operator =(const TMyException& e) {}
public:
    TMyException(const char* str) {
        assert(str != NULL);
        msg = new char[strlen(str) + 1];
        strcpy(msg, str);
    }
    TMyException(const TMyException& e) {
        msg = new char [strlen(e.msg) + 1];
        strcpy(msg, e.msg);
    }
    const char* getMessage() const { return msg; }

    ~TMyException() { delete [] msg; }
};
```

Перегруженная операция присваивания в описании, защищенном спецификатором доступа **private**, блокирует выполнение этой операции.

Функция **assert** аварийно завершает работу программы, если выражение, заданное в параметре, ложно (вызывает **abort()**).

Написание конструктора копирования обязательно – ведь он может неявно вызываться в блоке **catch**.



Пример

Генерация исключения из описанного класса:

```
throw TMyException  
    ("Extra data in input file");
```

Обработчик исключения:

```
try {  
    // здесь может быть сгенерировано  
    // собственное исключение  
}  
  
catch (TMyException e) {  
    cout << "ERROR! " << e.getMessage()  
        << endl;  
}
```

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 MyException: public exception {  
private:  
    MyException();  
public:  
    MyException(const char * const s): exception(s) {}  
    MyException(const MyException& e): exception(e) {}  
};
```

Обработка исключения

```
try {  
...  
    throw(MyException("Unknown internal error!"));  
...  
}  
catch (MyException e)  
    {cout << e.what() << endl;}
```


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 MyException& e) {  
    ...  
}  
catch (const exception& e) {  
    ...  
}  
catch (...) {  
    ...  
}
```

Спасибо за внимание

