top
КОМПЬЮТЕРНАЯ АКАДЕМИЯ

ОБЪЕКТНО -ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА C++

Урок №11

Наследование

Содержание

Вложенный класс

Здравствуйте!!! Сегодня мы будем знакомиться с различными видами взаимодействий классов между собой (или, иначе говоря механизмами повторного использования). И, начнём мы с вами с самой простой темы — вложенные классы. Итак, для начала определение:

Вложенный (внутренний) класс (inner class) — это класс, полностью определённый внутри другого класса. И, если объект обычного класса, как правило существует самостоятельно, то объект вложенного класса должен быть привязан к объекту класса, в котором этот вложенный класс описан. Класс, в который вкладывается другой класс называют объемлющим.

Основные особенности работы с вложенным классом

1. Вложенный класс является членом объемлющего класса, а его определение может находиться в любой из секций public, private или protected объемлющего класса.

2. Имя вложенного класса известно в области видимости объемлющего класса, но ни в каких других областях. Это означает, что оно не конфликтует с таким же именем, объявленным в объемлющей области видимости. Например, так:

```
//самостоятельный (глобальный) класс A
class A { /* ... */ };
```

```
class B {
public:

    //вложенный класс A инкапсулирован внутри области
    //видимости класса B
    class A {...};

    //Здесь, используется вложенный класс A
    A*obj;
};

void main(){
    //вложенный A невидим в данной области видимости
    //поэтому здесь используется глобальный класс A
    A*obj2;
}
```

3. Для вложенного класса допустимы такие же виды членов, как и для невложенного. Например:

```
class A {

public:
    class B {
        friend class A; //объявление друга
        B( int val=0 ); //конструктор
        B *next;        //указатель на собственный класс
        int value;
    };

private:
    B *obj;
};
```

4. Напомним, что закрытым называется член, который доступен только в определениях членов и друзей класса. У объемлющего класса нет права доступа к закры-

тым членам вложенного. Чтобы в определениях членов A можно было обращаться к закрытым членам B, класс B объявляет A как друга.

В продолжение прошлого утверждения, следует отметить, что вложенный класс также не имеет никаких специальных прав доступа к закрытым членам объемлющего класса. Если бы нужно было разрешить B доступ к закрытым членам класса A, то в объемлющем классе A следовало бы объявить вложенный класс как друга. В приведенном выше примере этого не сделано, поэтому B не может обращаться к закрытым членам A.

5. Объявление B открытым членом (в секции public) класса A означает, что вложенный класс можно использовать как тип во всей программе, в том числе и за пределами определений членов и друзей класса. Например:

```
void main(){
    A::B *ptr;
}
```

Этот приём дает широкую область видимости для вложенного класса. Однако, вложенный класс обычно реализуется только для личных нужд объемлющего класса и не должен быть доступен во всей программе. Поэтому лучше объявить вложенный класс B закрытым членом A, вот так:

```
class A {

public:
    //...
```

```
private:
   class B {
      // ...
   };
   B *obj;

};
```

Теперь тип B доступен только из определений членов и друзей класса A, поэтому все члены класса B можно сделать открытыми. При таком подходе объявление A как друга B становится ненужным. Вот новое определение класса A:

```
class A{

public:
    //...
private:

    //Теперь B закрытый вложенный тип
    class B {
    //а его члены открыты
    public:
       B( int val=0 );
       B *next;
       int value;
    };

    B *obj;

};
```

6. Обратим внимание, что конструктор класса B (в прошлом примере) не задан как встроенный внутри опре-

деления класса и, следовательно, должен быть определен вне него. Открытым остается вопрос: «Где?». Конструктор класса B не является членом A и, значит, не может быть определен в теле последнего. Следовательно, его нужно определить в глобальной области видимости — той, которая содержит определение объемлющего класса. Отсюда, рождается правило:

Когда функция-член вложенного класса не определяется как встроенная в теле, она должна быть определена вне самого внешнего из объемлющих классов.

Вот как могло бы выглядеть определение конструктора B. Однако, показанный ниже синтаксис, в глобальной области видимости некорректен:

```
class A {
public:

private:

    class B {

    public:
        B( int val=0 );

    };
};

//ошибка: B вне области видимости
B:: B( int val ) { ... }
```

Вся проблема заключается в том, что имя B отсутствует в глобальной области видимости. При использовании его таким образом следует указывать, что B — вложенный

класс в области видимости A. Это делается следующим образом:

```
//к имени вложенного класса мы обращаемся
//через имя объемлющего
//это происходит в глобальной области видимости
A::B::B( int val ) {
   value = val;
   next = 0;
}
```

7. Если бы внутри B был объявлен статический член, то его определение также следовало бы поместить в глобальную область видимости. Инициализация этого члена могла бы выглядеть так:

```
int A::B::static_mem = 1024;
```

Примечание: Обратите внимание, что функции-члены и статические данные-члены не обязаны быть открытыми членами вложенного класса для того, чтобы их можно было определить вне его тела. Закрытые члены B также определяются в глобальной области видимости.

8. Вложенный класс разрешается определять вне тела объемлющего. Например, определение B могло бы находиться и в глобальной области видимости:

```
class A {

public:
   //...
```

```
private:
    //здесь объявление необходимо
    class B;
    B *obj;
};


//имя вложенного класса отмечено именем
//объемлющего класса
class A::B {
public:
    B( int val=0 );
    B *next;
    int value;
};
```

В глобальном определении имя вложенного B должно быть отмечено именем объемлющего класса A. Заметьте, что объявление B в теле A опустить нельзя. Определение вложенного класса не может быть задано в глобальной области видимости, если предварительно оно не было объявлено членом объемлющего класса. Но при этом вложенный класс не обязательно должен быть открытым членом объемлющего.

9. Пока компилятор не увидел определения вложенного класса, разрешается объявлять лишь указатели и ссылки на него. Объявления члена obj класса A (в примере выше) правильно несмотря на то, что B определен в глобальной области видимости, поскольку данный член — указатель. Если бы один из них был объектом, то его объявление в классе A привело бы к ошибке компиляции:

```
class A {

public:
   //...

private:
   //объявление необходимо
   class B;
   B *obj;
   B x;   //ошибка: неопределенный вложенный класс B

};
```

Примечание: Зачем определять вложенный класс вне тела объемлющего? Возможно, он поддерживает некоторые детали реализации B, а нам нужно скрыть их от пользователей класса A. Поэтому мы помещаем определение вложенного класса в заголовочный файл, содержащий описание A. Таким образом, определение B может находиться лишь внутри исходного файла, включающего реализацию класса A и его членов.

10. Вложенный класс можно сначала объявить, а затем определить в теле объемлющего. Это позволяет иметь во вложенных классах члены, ссылающиеся друг на друга:

```
class A {

public:
   //...
private:

   //объявление A::B
   class B;
```

```
    class Ref {
       //pli имеет тип A::B*
       B *pli;
    };

    //определение A::B
    class B {
       //pref имеет тип A::Ref*
       Ref *pref;
    };

};
```

Если бы B не был объявлен перед определением класса Ref, то объявление члена pli было бы ошибкой.

11. Вложенный класс не может напрямую обращаться к нестатическим членам объемлющего, даже если они открыты. Любое такое обращение должно производиться через указатель, ссылку или объект объемлющего класса. Например:

```
class A {

public:
   int init( int );

private:

   class A::B {

   public:

      B( int val=0 );
      void mf( const A & );
      int value;

   };
};
```

```
A::B::B { int val )
{
   //A::init() – нестатический член класса A
   //должен использоваться через объект или
   //указатель на тип A

   value = init( val ); //ошибка: неверное
                        //использование init
};
```

При использовании нестатических членов класса компилятор должен иметь возможность идентифицировать объект, которому принадлежит такой член. Внутри функции-члена класса B указатель this неявно применяется лишь к его членам. Благодаря неявному this мы знаем, что член value относится к объекту, для которого вызван конструктор. Внутри конструктора B указатель this имеет тип B*. Для доступа же к функции-члену init() нужен объект типа A или указатель типа A*.

Следующая функция-член mf() является решением проблемы, так как обращается к init() с помощью параметра-ссылки. Таким образом, init() вызывается для объекта, переданного в аргументе функции:

```
void A::B::mf( A & i1 ) {

   //обращается к init() по ссылке
   memb = i1.init();
}
```

В заключение следует отметить, вложенные классы, чаще всего, нужны для выполнения каких-то локальных задач в процессе работы основного класса. Поэтому, мы

вам не советуем злоупотреблять ими. Возможно, в дальнейшем вам вдруг понадобиться использовать ваш класс в другом новом классе. И вместо того, чтобы просто создать объект придется всё писать заново.

Агрегирование и композиция

Агрегирование и композиция. Сейчас мы с вами рассмотрим еще один механизм, который строит отношения между классами. Этот механизм широко используется во многих языковых технологиях, в том числе и C++. Называется он — агрегирование (или агрегация).

Агрегирование — это включение объекта (объектов) одного класса в состав объекта другого класса.

Сейчас мы приведем вам пример агрегации, и, вы будете приятно удивлены. Оказывается, вы делали такое раньше самостоятельно.

```
#include <iostream>
using namespace std;

//класс "точка"
class Point{

    //кооординаты
    int X;
    int Y;
public:

    //конструктор
    Point(){
        X=Y=0;
    }
   //установка координат
```

```
    void SetPoint(int iX,int iY){
        X=iX;
        Y=iY;
    }

    //демонстрация координат
    void Show(){
        cout<<"----------------------------\n\n";
        cout<<X<<"\t"<<Y<<"\n\n";
        cout<<"----------------------------\n\n";
    }
};

//класс фигура
class Figura{

    //агрегация точки
    //(координаты углов)
    Point*obj;

    //количество углов
    int count;
    //цвет фигуры
    int color;

    public:

    //конструктор
    Figura(){
        count=color=0;
        obj=NULL;
    }

    //создание фигуры
    void CreateFigura(int cr,int ct){
        //если углов меньше трех — это не фигура
        if(ct<3) exit(0);
```

```
        //инициализация цвета и количества углов
        count=ct;
        color=cr;
        //выдделение памяти под массив точек
        obj=new Point[count];
        if(!obj) exit(0);

        //установка координат точек
        int tempX,tempY;
        for(int i=0;i<count;i++){
            cout<<"Set X\n";
            cin>>tempX;
            cout<<"Set Y\n";
            cin>>tempY;
            obj[i].SetPoint(tempX,tempY);
        }
    }

    //показ фигуры
    void ShowFigura(){
        cout<<"----------------------------\n\n";
        cout<<"Color"<<color<<"\n\nPoints —
              "<<count<<"\n\n";
        for(int i=0;i<count;i++){
            obj[i].Show();
        }
    }

    //если фигура была очистить память
    ~Figura(){
        if(obj!=NULL) delete[]obj;
    }
};

void main(){

    Figura f;
```

```
    f.CreateFigura(255,3);
    f.ShowFigura();
}
```

Следует отметить, что кроме самой АГРЕГАЦИИ, существует еще частный ее случай под названием композиция. Дадим определение этому явлению:

Композиция — форма агрегации, при которой каждый агрегируемый объект сам может является агрегатом.

Иначе говоря, если к нашему примеру мы добавим класс «рисунок», который будет аккумулировать в себе набор агрегатов-фигур, это и будет композиция.

Композиция и агрегация достаточно сильный аппарат, который конкурирует по своей форме и стилю с предметом наших последующих дискуссий — наследованием. Плюсы и минусы того и другого мы рассмотрим позже в этом уроке.

Наследование. Основные понятия

Начиная рассматривать вопросы наследования, нужно отметить, что любой объект призван собрать вместе свойства и поведение некоторого фрагмента решаемой задачи, связывая в единое целое данные и методы, относящиеся к этому фрагменту.

Вспомним. Каждый объект является конкретным представителем класса. Объекты одного класса имеют разные имена, но одинаковые по типам и внутренним именам данные. Объектам одного класса для обработки своих данных доступны одинаковые функции класса и одинаковые операции, настроенные на работу с объектами класса. Таким образом, класс выступает в роли типа, позволяющего вводить нужное количество объектов, имена (названия) которых программист выбирает по своему усмотрению.

Однако, объекты разных классов и сами классы могут находиться в отношении наследования, при котором формируется иерархия объектов, соответствующая заранее предусмотренной иерархии классов.

Иерархия классов позволяет определять новые классы на основе уже имеющихся. Имеющиеся классы обычно называют базовыми (родительскими). А, новые классы, формируемые на основе базовых — производными (классами-потомками,дочерними классами).

Производные классы получают по наследству данные и методы своих базовых классов и, конечно же, могут

иметь собственные составляющие. При этом, наследуемые члены не описываются в потомке, а остаются только в базовых классах.

При наследовании некоторые имена методов данных базового класса могут быть по-новому определены в производном классе. В этом случае соответствующие компоненты базового класса становятся недоступными из производного класса. Для доступа из производного класса к компонентам базового класса, имена которых повторно определены в производном, используется операция '::' указания (уточнения) области видимости.

Любой производный класс может, в свою очередь, становиться базовым для других классов, и таким образом формируется иерархия классов и объектов. В иерархии производный объект наследует разрешенные для наследования члены всех базовых объектов. Другими словами, у объекта имеется возможность доступа к данным и методам всех своих базовых классов.

Тот тип наследования, о котором мы говорим, называется — одиночное наследование. Однако, в языке C++ допускается, еще один вид — множественное наследование. Множественное наследование — возможность класса наследовать члены нескольких никак не связанных между собой базовых классов. Об этом типе мы поговорим подробнее чуть позже.

Условный пример на наследование

Давайте, для начала рассмотрим пример наследования условно, что называется, «на словах».

Итак, пусть есть класс «точка (позиция) на экране». Будем считать базовым классом, и на его основе постро-

им класс «окно на экране». Данными этого класса будут унаследованные от базового класса координаты точки и две собственные переменные — ширина и высота окна.

Рассмотрим предполагаемые методы потомка, свои и полученные в наследство:

- Функция смещения окна вдоль оси X на DX. СВОЯ
- Функция смещения окна вдоль оси Y на DY. СВОЯ
- Функция получения значения координаты X левого верхнего угла. РОДИТЕЛЬСКАЯ
- Функция получения значения координаты Y левого верхнего угла. РОДИТЕЛЬСКАЯ
- Функция получения размера окна вдоль оси X (ширины). СВОЯ
- Функция получения размера окна вдоль оси Y (высоты). СВОЯ
- Конструктор — создает окно на экране с заданным именем по параметрам, определяющим левый верхний угол окна и его размеры. СВОЙ
- Деструктор — уничтожает окно с заданным именем. СВОЙ

Просто, не правда ли? Но, пока это лишь теория, а нам пора переходить к практике. Давайте только напоследок, дадим чёткое определение тому, что мы изучаем. Итак:

Наследование — это механизм, посредством которого, один объект может получать свойства другого объекта и добавлять к ним черты, характерные только для него.

Спецификаторы доступа при наследовании и синтаксис организации наследования

Спецификаторы доступа

При наследовании классов важную роль играет область доступа членов класса. Для любого класса все его члены лежат в области его действия. Тем самым любая принадлежащая классу функция может использовать любые данные и вызывать любые принадлежащие классу функции. Вне класса в общем случае доступны только те его компоненты, которые имеют статус public.

Напомним, что существуют следующие спецификаторы доступа к членам класса:

1. ***Собственные*** (*private*) — методы и данные доступны только внутри того класса, где они определены.
2. ***Защищенные*** (*protected*) — методы и данные доступны внутри класса, в котором они определены, и дополнительно доступны во всех производных классах.
3. ***Общедоступные*** (*public*) — компоненты класса видимы из любой точки программы, т.е. являются глобальными.

Все вышеперечисленные спецификаторы доступа не только указываются для конкретных членов класса, но и могут быть использованны для определения статуса наследования. Для этого спецификатор устанавливают

при описании производного класса непосредственно перед именем базового класса.

Доступ в базовом классе	Спецификатор доступа перед базовым классом	Доступ в производном классе
public	отсутствует	private
protected	отсутствует	private
private	отсутствует	private
public	public	public
protected	public	protected
private	public	private
public	protected	protected
protected	protected	protected
private	protected	private
public	private	private
protected	private	private
private	private	private

Соглашения о статусах доступа при разных сочетаниях базового и производного классов иллюстрирует следующая таблица:

Синтаксис наследования

И, наконец, финальный пряник. А, именно, синтаксис создания наследуемого класса.

```
class имя_класса : спецификатор_наследования
                                   имя_базового_класса{
    описание_класса;
};
```

Разберём синтаксис более детально:

- **имя_класса** — имя нового создаваемого класса.
- **спецификатор_наследования** — спецификатор доступа к наследуемым членам.
- **имя_базового_класса** — класс, от которого, необходимо отнаследоваться.
- **описание_класса** — тело нового класса.

Кое-что о конструкторах и деструкторах...

Здесь мы приведем несколько утверждений, связанных с работой конструкторов и деструкторов при наследовании.

Конструктор базового класса всегда вызывается и выполняется до конструктора производного класса.

Деструкторы базовых классов выполняются в порядке, обратном перечислению классов в определении производного класса. Таким образом порядок уничтожения объекта противоположен по отношению к порядку его конструирования.

Вызовы деструкторов для объектов класса и для базовых классов выполняются неявно и, почти никогда, не требуют никаких действий программиста.

Ну что ж, багаж знаний упакован — пора попрактиковаться.)

Пример реализации одиночного наследования

А, сейчас реализуем ту конструкцию, которую мы создали условно, в одном из предыдущих разделов.

```
#include <iostream>
using namespace std;

//Класс "точка"
class Point{
protected:
    int x;
    int y;
public:
    Point(){
        x=0;
        y=0;
    }
    //получение x
    int&GetX(){
        return x;
    }
    //получение y
    int&GetY(){
        return y;
    }
};

class MyWindow: public Point{
    int width;
    int height;

public:
    MyWindow(int W,int H){
        width=W;
        height=H;
    }
    //получение ширины
    int&GetWidth(){
        return width;
    }
```

```
    //получение высоты
    int&GetHeight(){
        return height;
    }
    //функции сдвига
    void MoveX(int DX){
        x+=DX;
    }
    void MoveY(int DY){
        y=DY;
    }
    //показ на экран
    void Show(){
        cout<<"--------------\n\n";
        cout<<"X = "<<x<<"\n\n";
        cout<<"Y = "<<y<<"\n\n";
        cout<<"W = "<<width<<"\n\n";
        cout<<"H = "<<height<<"\n\n";
        cout<<"--------------\n\n";
    }
};
void main(){
    //создание объекта
    MyWindow A(10,10);
    A.Show();
    //изменение параметров
    A.GetX()=5;
    A.GetY()=3;
    A.GetWidth()=40;
    A.GetHeight()=50;
    A.Show();
    //сдвиг "окна"
    A.MoveX(2);
    A.MoveY(7);
    A.Show();
}
```

Множественное наследование

В C++ производный класс может быть порождён из любого числа непосредственных базовых классов. Наличие у производного класса более чем одного непосредственного базового класса называется множественным наследованием. Синтаксически множественное наследование отличается от единичного наследования списком баз, состоящим более чем из одного элемента. Например, так:

```
class A
{
    //описание класса A
};

class B
{
    //описание класса B
};

class C : public A, public B
{
    //описание класса C
};
```

Несколько особенностей множественного наследования

При создании объектов-представителей производного класса, порядок расположения непосредственных

базовых классов в списке баз определяет очерёдность вызова конструкторов умолчания. Этот порядок влияет и на очерёдность вызова деструкторов при уничтожении этих объектов.

Более существенным является ограничение, согласно которому одно и то же имя класса не может входить более одного раза в список баз при объявлении производного класса. Это означает, что в наборе непосредственных базовых классов, которые участвуют в формировании производного класса не должно встречаться повторяющихся элементов.

А, теперь, чтобы долго не задерживаться на теории, давайте расмотрим практический пример, описанный в следующем разделе урока.

Пример использования множественного наследования

Классический пример, переходящий из поколения в поколение. Создадим примитивную структуру наследования. Соберем животное — «лось»))).

```
#include <iostream>
#include <string.h>
using namespace std;

//Класс "рога"
class Roga{

protected:

    char color[25];
    int wes;
```

```
public:
    Roga(){

        strcpy(color,"Dirty");
        wes=20;
    }
    Roga(char  *c,int w){
        strcpy(color,c);
        wes=w;
    }
};

//Класс "копыта"
class Kopyta{
protected:

    char forma[25];
    int razmer;

public:
    Kopyta(){
        strcpy(forma,"Big");
        razmer=10;
    }

    Kopyta(char  *c,int w){

        strcpy(forma,c);
        razmer=w;
    }
};

//Класс "Лось", производный от
//классов "рога" и "копыта"
class Los:public Roga,public Kopyta{

    public:
    char name[255];
```

```
    Los(char  *c){
        strcpy(name,c);
    }
    //Функция потомка может обращаться к
    //элементам обоих базовых классов
    void DisplayInfo(){

        cout<<"Name "<<name<<"\n";
        cout<<"Forma "<<forma<<"\n";
        cout<<"Color "<<color<<"\n";
       cout<<"Wes rogov "<<wes<<"\n";
       cout<<"Razmer kopyt "<<razmer<<"\n";
    }
};

void main()
{

    //создание объекта класса-потомка
    Los l("Vasya");
    l.DisplayInfo();
}
```

Обсуждение плюсов и минусов наследования

Итак, сегодня мы с вами рассмотрели несколько способов взаимодействия классов между собой. У этих способов есть свои достоинства и недостатки. Разберем их:

Наследование определяется статически на этапе компиляции, его проще использовать, поскольку оно напрямую поддерживается языком программирования.

В случае наследования упрощается также задача модификации существующей реализации. Если потомок замещает лишь некоторые операции, то могут оказаться затронутыми и остальные операции, т.к. возможно они вызывают замещенные.

Но у наследования есть и минусы:

Во-первых, нельзя изменить унаследованную реализацию во время выполнения.

Во-вторых, родительский класс нередко хотя бы частично определяет физическое представление своих подклассов. Реализации родительского и производного классов сильно связаны.

Композиция объектов определяется динамически во время выполнения за счет того, что объекты получают ссылки на другие объекты. Композицию можно применить, если объекты соблюдают интерфейсы друг друга. Во время выполнения программы любой объект можно заменить другим, лишь бы он имел тот же тип.

Более того, поскольку при реализации объекта кодируются прежде всего его интерфейсы, то зависимости от реализации резко снижается.

Ну, а выбор механизма, как всегда зависит от поставленной задачи.

Наследование шаблонов

Рады вам сообщить, что шаблоны классов, как и сами классы, поддерживают механизм наследования. Все основные идеи наследования при этом остаются неизменными, что позволяет построить иерархическую структуру шаблонов, аналогичную иерархии классов.

Рассмотрим совершенно тривиальный пример, на котором продемонстрируем, каким образом можно создать шаблон класса, производный из другого шаблона.

```
//класс-родитель
template <class T>
class Pair
{
    T a;
    T b;
    public:
    Pair (T t1, T t2);
    //...
};

//конструктор класса родителя
template <class T>
Pair <T>::Pair (T t1, T t2) : a(t1), b(t2)
{}

//класс-потомок

template <class T>
class Trio: public Pair <T>
{
```

```
    T c;
    public:
    Trio (T t1, T t2, T t3);
    //...
};

//Заметьте, что вызов родительского конструктора
//также сопровождается передачей типа T
//в качестве параметра.
template <class T>
Trio<T>::Trio (T t1, T t2, T t3): Pair <T> (t1, t2),
c(t3)
{
}
```

Домашнее задание

1. Создайте класс Student, который будет содержать информацию о студенте. С помощью механизма наследования, реализуйте класс Aspirant (аспирант — студент, который готовится к защите кандидатской работы) производный от Student.
2. Создайте класс Passport (паспорт), который будет содержать паспортную информацию о гражданине Украины. С помощью механизма наследования, реализуйте класс ForeignPassport (загранпаспорт) производный от Passport. Напомним, что загранпаспорт содержит помимо паспортных данных, также данные о визах, номер загранпаспорта.
3. Используя понятие множественного наследования, разработайте класс «Окружность, вписанная в квадрат».
4. В разделе урока «Агрегирование и композиция» приведен пример на альянс класса «точки» и класса «фигура». Ваша задача добавить к программе класс, который будет содержать в себе несколько объектов класса фигура, создавая тем самым композицию.