Наследование и виртуальные функции

Лекции 6-7

Статические члены класса

Статические члены класса – информационные данные, которые для всех классов должны быть представлены в одном экземпляре.

Пример:

```
class StaticEx{
static int x;
StaticEx(){x++;}
};
```

Какую роль выполняет переменная x в приведенном примере?

Статические члены класса

Статические члены существуют „в отрыве“ от всех экземпляров класса.

Как к ним обращаться? Рассмотрим на примере.

```
StaticEx v1, v2;
v1.x = 100;   v2.x = 200;
StaticEx.x = 300;
int var = v1.x;
```

Чему будет равно значение переменной `var`?

Статические методы класса

Статические методы не могут использовать никаких членов класса, кроме статических.

Не имеет явного параметра this, поскольку является частью класса, а не объекта.

Не может вызвать нестатический метод.

Может быть использован для запрета объявления переменной без инициализации. Тогда конструктор и деструктор помещаются в закрытую область.

Не может быть виртуальным или константным.

Статические методы класса

```
class X{
    X(){}
    ~X(){}
public:
    static X& createX(){
        X* x1 = new X; cout << "X created" << endl;
        return*x1;
    }
    static void destroyX(X& x1){
        delete &x1; cout << "X destroyed" << endl;
    }
};
int main(){
X& xx1 = X::createX();
 . . .
X::destroyX(xx1);
return 0;
}
```

Особенности статических членов класса

Статические члены класса всегда должны быть объявлены(и желательно проинициализированы) вне класса. Это связано с особенностями их хранения в памяти.

тип_перем имя_класса::идентификатор= инициализатор;

Статика пример

```
class B {
   static int i;//статический информ. член класса
public:
   static void f(int j){i = j;}//стат. метод
};
int B::i = 10; // дополнительное внешнее определение
               // статической переменной с
               // инициализацией статического
               // информационного члена класса.
int main(){
    B a;
    ...
   B::f(1);//вызов статической функции-члена класса.
return 0;
}
```

Наследование

Наследование – отношение между классами, при котором один класс повторяет структуру и поведение другого класса или других классов.

Базовый класс – тот, чьи характеристики наследуются.

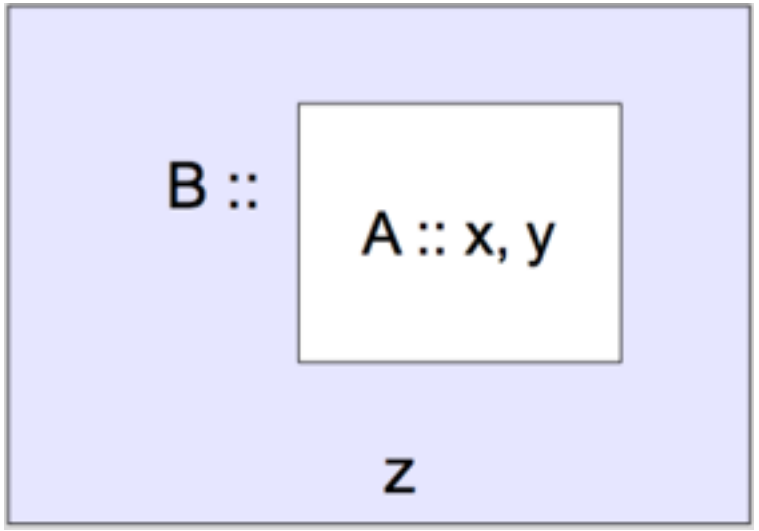
Производный класс – тот, кто наследует характеристики.

class < имя derived> : < способ_насл. > < имя base > {...};

Пример

```
struct A {int x, y; };
struct B: A { int z; };

A a1;
B b1;
b1.x = 1;
b1.y = 2;
b1.z = 3;
a1 = b1;
```



Единичное наследование

Конструкторы, деструкторы и operator= не наследуются!!!

```
struct A { int x; int y;};     struct B : A { int z; };
class C : protected A { int z; };
    .........
A a;                  A * pa;
B b;                  C   c,  * pc = &c;
b.x = 1;        pc -> z;    // ошибка: доступ к закрытому полю
b.y = 2;        pc -> x;    // ошибка: доступ к защищённому
полю
b.z = 3;        pa = ( A * ) pc;
a = b;          pa -> x;    // правильно: поле A::x – открытое


A a, *pa;
B b, *pb;
pb = &b;
pa = pb;
pb = ( B* ) pa;
```

Сокрытие имён

```
struct A {
        int f ( int x , int y);
        int g ();
        int h;
};
struct B : public A {
        int x;
        void f ( int x );
        void h ( int x );
};
...
A  a,  *pa;
B  b,  *pb;
pb = &b;
pb -> f (1);  // вызывается B::f(1)
pb -> g ();  // вызывается A::g()
pb -> h = 1; // Err.! функция h(int) – не L-value выражение
pa = pb;   pa -> f (1); // Err.! функция A::f(1) имеет 2 пар-ра
pb = &a;      // Err.!  расширяющее присваивание // pb = (B*)&a
pb -> f (1); // Возможна Err, если в f(1) используется x  из B
```

Видимость и доступность имен

```
int x;
void f (int a) { cout << " :: f " << a << endl; }

class A {
      int x;
public:
      void f (int a) { cout << " A:: f " << a << endl; }
};
class B : public A {
public:
   void f (int a) { cout << " B:: f " << a << endl; }
      void g ();
};

void B::g() {
      f(1);         // вызов  B::f(1)
      A::f(1);
      ::f(1);       // вызов глобальной void f(int)
   //x = 2;  //ошибка!!! – осущ. доступ к закрытому члену
   класса A
}
```

Вызов конструкторов базового и производного классов. Пример.

```
class A { ... };

class B : public A {
public:
   B ();
   B (const B &); //есть явно описанный конструктор копирования
   ...
};
class C : public A {
public:
   // нет явно описанного конструктора копирования
   ...
};
int   main ( ) {
   B b1;       //  A (),   B ()
   B b2 = b1; //  A (),   B (const B &)
   C c1;              //  A (),   C ()
   C c2 = c1; //  A (const A &),   C (const C &)
   ...
}
```

Классы student и student4

```
class student {
    char * name;
    int year;
    double est;
public:
    student ( char* n, int y, double e);
    void print () const;
     ~student ();
};
```

```
class student4: public student {
        char * diplom;
        char * tutor;
public:
        student4 ( char*n, double e, char*d, char* t);
         void print () const;
        //эта print  скрывает print из базового класса
        ~student4 ();
};
```

```
student4:: student4( char* n, double e, char* d, char* t) : student (n, 4, e) {
    diplom = new char [strlen (d) + 1];
    strcpy (diplom, d);
    tutor = new char [strlen (t) + 1];
    strcpy (tutor, t);
}
```

```
student4 :: ~student4 () {

    delete [ ] diplom;
    delete [ ] tutor;
}
```

```
void  student4 :: print () const {
        student :: print ();   // name, year, est
                cout << diplom << endl;
        cout << tutor << endl:
}
```

Использование классов student и student

```
void f ( ) {
  student  s (“Kate”, 2, 4.18),  * ps = & s;
  student4  gs (“Moris”, 3.96, “DIP”, “Nick”);
  student4 * pgs = & gs;

  ps -> print();    // student :: print ();
  pgs -> print();   // student4 :: print ();

  ps = pgs;     // base = derived – допустимо с
          // преобразованием по умолчанию.

  ps -> print(); // student :: print () –
    //функция выбирается статически по типу
    //указателя.
```

Виртуальные методы

Метод называется **виртуальным**, если при его объявлении в классе используется квалификатор **virtual**.

Класс называется **полиморфным**, если содержит хотя бы один виртуальный метод.

Объект полиморфного класса называют **полиморфным** объектом.

Виртуальные методы

Чтобы динамически выбирать функцию print () по типу объекта, на который ссылается указатель, переделаем классы:

```
class student {…
public:
    ...
    virtual void print ( ) const ;
};
class student4 : public student {…
public:
    ...
    [virtual] void print ( ) const ;
};
```

Тогда: ps = pgs;
ps -> print(); // student4 :: print () – ф-я выбирается
// динамически по типу объекта, чей адрес
// в данный момент хранится в указателе

Виртуальные деструкторы

для полиморфных классов делайте деструкторы виртуальными!!!

```
void f () {
   student * ps = new  student4 (“Moris”, 3.96,
   “DIP”, “Nick”);
       …
   delete ps;//вызовется ~student: высвобождается не все!
}
```

Но если:

virtual ~student (); и
[virtual] ~student4 ();

то вызовется ~student4(), т.к. сработает динамический полиморфизм.

Механизм виртуальных функций (механизм динамического полиморфизма)

1. !Виртуальность функции, описанной с использованием служебного слова **virtual** не работает сама по себе, она начинает работать, когда появляется класс, производный от данного, с функцией с **таким же прототипом**.

2. Виртуальные функции выбираются по типу объекта, на который ссылается указатель (или ссылка).
3. У виртуальных функций должны быть одинаковые прототипы. Исключение составляют функции с одинаковым именем и списком формальных параметров, у которых тип результата есть указатель или ссылка на себя (т.е. соответственно на базовый и производный класс).

4. Если виртуальные функции отличаются только типом результата (кроме случая выше), генерируется ошибка.

5. Для виртуальных функций, описанных с использованием служебного слова **virtual,** с разными прототипами работает механизм сокрытия имен.

Абстрактные классы

Абстрактным называется класс, содержащий хотя бы одну **чистую виртуальную** функцию.

Чистая виртуальная функция имеет вид:

virtual тип_рез имя (сп_фп) **= 0**;

Абстрактные классы

```
class shape {
public:
      virtual double area () = 0;
};

class rectangle: public shape {
      double height, width;
public:
      double area () {
            return    height * width;
      }
};

class circle: public shape {
      double radius;
public:
      double area () {
            return    3.14 * radius * radius;
      }
};
```

```
#define N 100
....
        shape* p [ N ];

        double total_area = 0;
                ....
        for (int i =0; i < N; i++)
            total_area += p[i] -> area();
                ....
```

Интерфейсы

Интерфейсами называют абстрактные классы, которые

- не содержат нестатических полей-данных, и
- все их методы являются открытыми чистыми виртуальными функциями.

Виртуальные функции. Пример 1.

```
class X {
public:
    void g ( ) {
        cout << "X::g\n";
        h ( );
    }
    virtual void f() {
        g ( );
        h ( );
    }
    virtual void h ( ) {
        cout << "X::h\n";
    }
};
```

```
class Y : public X {
    public:
        void g ( ) {
        cout << "Y::g\n";
        h ( );
    }
        virtual void f ( ) {
        g ( );
        h ( );
    }
        virtual void h ( ) {
         cout << "Y::h\n";
    }
};
```

```
int main () {
        Y b;
        X  *px = &b;
        px -> f();        //  ???
        px -> g();        //  ???
        return 0;
}
```

Виртуальные функции. Пример 1.

```
class X {
public:
    void g ( ) {
        cout << "X::g\n";
        h ( );
    }
    virtual void f() {
        g ( );
        h ( );
    }
    virtual void h ( ) {
        cout << "X::h\n";
    }
};
```

```
class Y : public X {
    public:
        void g ( ) {
        cout << "Y::g\n";
        h ( );
    }
        virtual void f ( ) {
        g ( );
        h ( );
    }
        virtual void h ( ) {
         cout << "Y::h\n";
    }
};
```

```
int main () {
        Y b;
        X  *px = &b;
        px -> f();      //  Y::g    Y::h    Y::h
        px -> g();      //  ???
        return 0;
}
```

Виртуальные функции. Пример 1.

```
class X {
public:
    void g ( ) {
        cout << "X::g\n";
        h ( );
    }
    virtual void f() {
        g ( );
        h ( );
    }
    virtual void h ( ) {
        cout << "X::h\n";
    }
};
```

```
class Y : public X {
    public:
        void g ( ) {
        cout << "Y::g\n";
        h ( );
    }
        virtual void f ( ) {
        g ( );
        h ( );
    }
        virtual void h ( ) {
         cout << "Y::h\n";
    }
};
```

```
int main () {
        Y b;
        X  *px = &b;
        px -> f();      //  Y::g    Y::h    Y::h
        px -> g();      //  X::g    Y::h
        return 0;
}
```

Виртуальные функции. Пример 2.

```
struct A {
    virtual int f (int x, int y) {
        cout << "A : :f (int, int) \n";
        return x + y;
    }
    virtual void f ( int x ) {
        cout << "A :: f( ) \n";
    }
};

struct B :  A {
    void f ( int x ) {
        cout << "B :: f( ) \n";
    }
};

struct C :  B {
        virtual int f (int x, int y) {
        cout << "C :: f (int, int) \n";
        return x + y;
    }
};
```

```
int main () {
        B b,  *pb = &b;
        C c;
        A  * pa = &b;
        pa -> f (1);      // B::f();
                pa -> f (1, 2);  // A::f(int, int)
    //pb -> f (1, 2);  // Err.! Эта f не видна

        A & ra = c;
        ra.f(1,1);        // C::f(int, int)

                B & rb = c;
    //rb.f(0,0);   // Err.! Эта f не видна
         return 0;
}
```

Виртуальные функции. Пример 3.

```
struct B {
    virtual B& f () { cout << " f ( ) from B\n"; return *this;}
    virtual void g (int x, int y = 7) { cout << "B::g\n"; }
};

struct D : B {
    virtual D& f ( ) { cout << " f ( ) from D\n"; return *this; }
    virtual void g (int x, int y) { cout << "D::g y = " << y << endl; }
};
    int main () {
        D  d;
        B  b1,  *pb = &d;
        pb -> f();              // f ( ) from D
        pb -> g(1);             // D::g  y = 7
        pb -> g(1,2);           // D::g  y = 2
        return 0;
    }
```

Если значение параметра y по умолчанию будет в функции g класса D, а в базовом не будет, компилятор на вызов pb - > g(1) выдаст ошибку.