

Наследование

2013

- ▶ Наследование в C++ – это механизм, посредством которого один класс может получать свойства другого.
- ▶ Наследование позволяет строить иерархию классов, переходя от более общих к более специальным.
- ▶ Класс, который наследуется, называют **базовым классом**. Наследующий класс называется **производным классом**.

- ▶ В языке C++ производный класс наследует все элементы базового класса, за исключением :
 - конструкторов,
 - деструктора,
 - перегруженной операции присваивания,
 - определения друзей класса.

- ▶ Производный класс содержит в себе все поля и методы базового класса, добавляя к ним новые поля и методы, определенные в нем самом.
- ▶ Кроме того, производный класс может изменять (переопределять) любой наследуемый метод.

Новый классификатор доступа

protected

Доступность для разных разделов

Раздел базового класса	Сам класс, и его друзья	Методы классов - наследников	Другие классы и функции
Открытый (public)	да	да	да
Защищенный (protected)	да	да	нет
Закрытый (private)	да	нет	нет

Описание производного класса

```
class <имя производного класса>:  
    {public|protected|private}  
    <имя базового класса>  
{  
    . . .  
};
```

Открытое наследование

- ▶ Реализует отношение «*is-a*» между базовым и производным классами (является объектом типа. . .)
- ▶ Открытые и защищенные члены базового класса остаются, соответственно, открытыми и защищенными членами производного класса

Открытое наследование

- ▶ При открытом наследовании все, что характеризует объекты класса-предка, является справедливым и для объектов класса-наследника.
- ▶ Это свойство называется *совместимостью типов* объектов.
- ▶ Благодаря этому объект производного класса можно применять вместо объекта базового класса, но не наоборот.

```
class A
{
// базовый класс
};
class B : public A
{
// public наследование
};
A* a = new B(); // допустимо при открытом
                // наследовании
```

```
class Base
{
private:
    int i, j;
public:
    Base() { cout<<"Base consructor"<<endl;}
    void setParams(int a, int b) { i = a; j = b; }
    void showParams() { cout << i << " " << j << endl; }
};
```

```
// inheritance class
class Derived : public Base
{
private:
    int k;
public:
Derived (int x)    // Derived (int x) : Base()
    { k = x; }
void showK() { cout << k << "\n"; }
};
```

```
int main()
{
    Derived ob(3);
    ob.setParams(1, 2); // access member of Base
    ob.showParams();    // access member of Base
    ob.showK();         // uses member of Derived
    return 0;
}
```

- ▶ Добавим в Derived

```
void setAll(int a, int b, int c)
```

```
{  
    setParams(a,b); // i = a; j = b; нельзя. . . почему?  
    k = c;  
}
```

Изменим базовый класс

```
class Base
{
protected: // вне класса действует как private
    int i, j;
public:
Base() { cout<<"Base constructor"<<endl;}
void setParams(int a, int b) { i = a; j = b; }
void showParams() { cout << i << " " << j << endl; }
};
```

- ▶ теперь возможно

```
void setAll (int a, int b, int c)
{
    i = a;    j = b;    k = c;
}
```

- ▶ а можно и так //переопределение - overload

```
void setParams (int a, int b, int c)
{
    i = a;    j = b;    k = c;
}
```


Правило для конструкторов

- ▶ Конструкторы не наследуются
- ▶ Перед вызовом конструктора производного класса будет вызван конструктор базового
- ▶ Для указания, какой именно из конструкторов базового типа следует вызвать в каждом конкретном конструкторе класса-потомка, используется синтаксис списка инициализаторов
- ▶ Если конструктор базового класса отсутствует в списке инициализации, используется конструктор базового класса по умолчанию

```
class Base
{
    . . .
public:
    Base() {
        cout<<"Base empty constructor"<<endl;
        i=j=0;
    }
    Base(int a, int b) : i(a), j(b) {
        cout<<"Base param constructor"<<endl;
    }
}
```

```
// inheritance class
class Derived : public Base
{
    . . .
public:
    Derived ( ) : Base( ), k(0)
        { cout<<"Derived empty constructor"<<endl;}
    Derived (int a, int b, int c) : Base(a, b), k(c)
        { cout<<"Derived param constructor"<<endl;}
};
```

- ▶ Если в классе–потомке не определен ни один конструктор, то будет создан конструктор по умолчанию, который вызовет конструкторы по умолчанию всех предков
- ▶ Важно, чтобы в иерархии классов **всегда были определены конструкторы по умолчанию**

Правила для деструкторов

- ▶ деструкторы всегда вызываются в порядке, обратном вызову конструкторов. Сначала – деструктор производного класса, затем – базового класса
- ▶ вызов деструкторов осуществляется автоматически и не требует явного указания

- ▶ В производном классе можно переопределять функции базового класса
- ▶ При переопределении функции базового класса в производном классе списки параметров могут не совпадать
- ▶ Перегрузка при этом не происходит, так как перегрузка возможна только в одном пространстве имен. Каждый класс имеет свое пространство имен

```
class Base
{
    ...
public:
void showParams() { cout << i << " " << j << endl; }
};
Derived : public Base
{
    ...
public:
void showParams() { cout << i << " " << j << " " << k << endl; }
};
```

- ▶ если нужно использовать метод базового класса

```
Derived : public Base
{
    ...
public:
void showParams() {
    Base::showParams();
    cout << k<< endl; }
};
```


Полиморфизм

аналогично перегрузке

```
Base a;
```

```
Derived b;
```

```
...
```

```
a.showParams( ); // метод класса Base
```

```
b.showParams( ); // метод класса Derived
```

Полиморфизм

```
Base * p, *q;
```

```
p = new Base(1,2);
```

```
p->showParams(); // метод класса Base
```

```
q = new Derived(1,2,3);
```

```
q->showK(); // нельзя, т.к. showk не определен в  
//Base
```

```
q->showParams(); // метод класса Base
```

```
//– нет полиморфизма!!!
```

Когда нужен такой полиморфизм

- ▶ массивы ссылок на базовый класс, но указывающих на объекты разных классов-наследников (is-a)
- ▶ параметры функций, передаваемые по ссылке, если функция должна одинаково работать с базовыми и производными классами

Почему не работает?

- ▶ Определение, какую из функций вызвать, происходит в момент компиляции по типу переменной-объекта или переменной-ссылки (указателя)
- ▶ Определение на этапе компиляции вызываемого варианта переопределенной функции называется *статическим*, или *ранним связыванием*

- ▶ Для того чтобы обеспечить возможность определения, какой из переопределенных методов должен быть вызван, не на основе типа переменной-ссылки или указателя, а на основе типа объекта, на который они ссылаются, нужен другой механизм – *динамическое* или *позднее связывание*

Как добиться?

- ▶ виртуальные функции (методы)

```
class Base
```

```
{
```

```
...
```

```
public:
```

```
virtual void showParams() { cout << i << " " << j << endl; }
```

```
};
```

```
Derived : public Base
```

```
{
```

```
...
```

```
public:
```

```
[virtual] void showParams()
```

```
    { cout << i << " " << j << " " << k << endl; }
```

```
};
```

Теперь

```
Base *q;
```

```
p = new Base(1,2);
```

```
p->showParams(); // метод класса Base
```

```
q = new Derived(1,2,3);
```

```
q->showParams(); // метод класса Derived
```

- ▶ Если объявление метода в базовом классе начинается с ключевого слова `virtual`, то это делает функцию виртуальной для базового класса и всех классов, производных от базового класса, – «виртуальная функция всегда виртуальна»

- ▶ конструкторы никогда не могут быть виртуальными
- ▶ деструктор базового класса рекомендуется всегда делать виртуальным

Чисто виртуальная функция

- ▶ Если в базовом классе нужно создать виртуальную функцию, которая не может иметь реализацию, то ее можно сделать «чисто виртуальной»

```
class Figure {  
    protected:  
        double x,y;  
    public:  
        void set_dim (double x0, double y0) { x=x0;    y=y0; }  
        virtual double show_area()=0;  
};
```

Абстрактный класс

- ▶ Если класс имеет хотя бы одну чисто виртуальную функцию, его называют *абстрактным классом*
- ▶ Для абстрактного класса не могут быть созданы объекты
- ▶ Такой класс используется только в качестве базового в системе наследования и для создания указателей и ссылок, которые будут использованы при реализации динамического полиморфизма

Figure a; // недопустимо для абстрактного класса
Figure *p; // возможно

- ▶ Если в базовом классе имеется чисто виртуальная функция, производный класс должен иметь определение собственной реализации этой функции
- ▶ Если реализация хотя бы одной из чисто виртуальных функций не будет выполнена, производный класс, в свою очередь, будет абстрактным

```
class rectangle : public figure {  
    public:  
        double show_area() { return x*y; }  
};
```

```
class triangle: public figure {  
    public:  
        double show_area() { return 0.5 *x*y; }  
};
```

```
int main() {  
    figure *p;  
    triangle t;  
    rectangle r;  
  
    p=&t;  
    p->set_dim(10.0,5.0);  
    cout << p->show_area()<<endl ;  
  
    p=&r;  
    p->set_dim(10.0,5.0);  
    cout<< p->show_area()<< endl;  
    return 0;  
}
```