

Наследование (Inheritance)

Идея наследования: создание нового класса на основе имеющегося

- переменные и функции родительского класса наследуются
- можно добавить новые переменные
- можно изменить функциональность: добавит новые или переопределить старые функции

Терминология

- Базовый класс, родительский класс: `class Base {...};`
- Производный класс, класс наследник:
`class Derivative: public Base {...};`

Базовый класс: точка на плоскости

```
class Point {
    public:
        Point(double X=0, double Y=0) : x(X), y(Y) {
            cout << "Point("<<x<<","<<y<<")" << endl; }
        ~Point() { cout << "~Point("<<x<<","<<y<<")" << endl; }
        double distance() const {return sqrt(x*x+y*y);}
    private:
        double x,y; // X,Y - coordinates
};
```

Производный класс: вектор на плоскости

```
class Vector : public Point {
    public:
        Vector(double X=0, double Y=0) : Point(X,Y) {           // свой ctor!
            cout << "Vector("<<X<<","<<Y<<")" << endl; }
        ~Vector() { cout << "~Vector" << endl; }                // свой dtor!
        double length() const {return Point::distance();}      // новая функ.
};
```

Пример для классов Point и Vector

```
{  
Point a(1,2);                                // Point(1,2)  
cout << " a.distance= " << a.distance() << endl; // a.distance= 2.23607  
Vector b(3,4);                               // Point(3,4) Vector(3,4)  
cout << " b.length= " << b.length() << endl;    // b.length= 5  
cout << " b.distance= " << b.distance() << endl; // b.distance= 5  
}  
// в конце жизни объектов 'a' и 'b' вызовутся деструкторы:  
// ~Vector ~Point(3,4)  
// ~Point(1,2)
```


Уровни доступа к базовому классу

Добавим в Vector функцию вращения

```
void Vector::rotate(double phi) { // clockwise rotation by angle phi
    double x_new = x*cos(phi)+y*sin(phi);
    double y_new = -x*sin(phi)+y*cos(phi);
    x=x_new;
    y=y_new;
}
```

Ошибка компиляции

```
In member function 'void Vector::rotate(double)':
    error: 'double Point::x' is private
    error: 'double Point::y' is private
```

 переменные из `private` недоступны классу наследнику

```
class Point {  
    ...  
    protected: // меняем private -> protected  
        double x,y;  
};
```

protected, защищенные члены класса

Члены класса помещенные в секцию `protected`:

- открыты для любого класса наследника
- закрыты для остального мира

Тестируем `rotate`:

```
b.rotate(M_PI/2);  
cout << " length= " << b.length() << endl; // length= 5
```

Управление доступом

Три типа наследования:

- ❶ `class D: public B` — все открытые члены `B` остаются открытыми в `D`; защищенные из `B` остаются защищенными в `D`
- ❷ `class D: private B` — открытые и защищенные члены `B` становятся закрытыми в `D`
- ❸ `class D: protected B` — открытые и защищенные члены `B` становятся защищенными в `D`

👉 Все что находится в `private` базового класса `B` недоступно производному классу `D`

Управление доступом, иллюстрирующий пример

```
class A {  
    public:          int x;  
    protected:     int y;  
    private:        int z;  
};  
  
class B : public A {  
    // x is public  
    // y is protected  
    // z is not accessible from B  
};  
  
class C : protected A {  
    // x is protected  
    // y is protected  
    // z is not accessible from C  
};  
  
class D : private A {  
    // x is private  
    // y is private  
    // z is not accessible from D  
};
```

Иерархия классов

- Базовым может быть любой класс, в том числе класс, производный от какого либо другого класса
- Класс может наследовать несколько базовых классов
- «Цепочку» наследования называют иерархией классов и представляют в виде диаграммы

• Производный класс может сам иметь наследника:

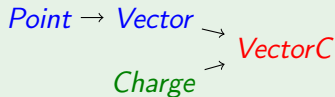
```
class VectorQ : public Vector {  
    public:  
        VectorQ(double X, double Y, double Q) : Vector(X,Y),q(Q) {}  
    private:  
        double q;  
};
```

Иерархия классов: $Point \rightarrow Vector \rightarrow VectorQ$

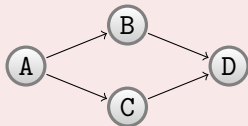
Множественное наследование

```
struct Charge {  
    Charge(int Q) : q(Q) {}  
    int q;  
};  
class VectorC: public Vector, public Charge {  
public:  
    VectorC(double X, double Y, double Q) :  
        Vector(X,Y), Charge(Q) {}  
};
```

Иерархия классов:



Проблема ромбовидной (diamond) иерархии



- Как наследуется объект *A* в *D*?
- Возможна неопределенность вызова виртуальных функций

Доступ к членам и функциям базовых классов

☞ Доступ к элементам базового класса осуществляется с помощью оператора `::`

```
class VectorV : public Vector {
public:
    VectorV(double X, double Y, Vector V) :
        Vector(X,Y),v(V) {} // X,Y - координаты базового вектора
    double length() const {return v.distance();} //переопределили length()
private:
    Vector v; // член класса, объект типа Vector
};

Vector b(3,4);
VectorV VV(1,1,b);
cout << VV.length() << endl; // 5 функция из VectorV
cout << VV.Vector::length() << endl; // 1.41421 функция из Vector
```

Виртуальные функции

Мотивация

- Имеется набор классов производных от `Base`
- Объекты этих классов удобно хранить в виде набора указателей (или ссылок) на базовый класс: `Base* p[100]`
- но `p[i]->Fun()` вызовет функцию `Base::Fun()`, а не от «реального» класса
- Язык C++ реализует механизм виртуальных функций позволяющий для `p[i]->Fun()` вызвать функцию из класса наследника

👉 В каждом классе своя функция `Fun()`, а интерфейс вызова один!

Простой пример

```
struct Base {  
    virtual void fun() {cout << "I am Base" << endl;}  
};  
struct Derived : public Base {  
    virtual void fun() {cout << "I am Derived" << endl;}  
};  
Base* array[2] = { new Base, new Derived };  
for( int i = 0; i < 2; i++ ) array[i]->fun();
```

I am Base

I am Derived

Простой пример

```
struct Base {  
    virtual void fun() {cout << "I am Base" << endl;}  
};  
struct Derived : public Base {  
    virtual void fun() {cout << "I am Derived" << endl;}  
};  
Base* array[2] = { new Base, new Derived };  
for( int i = 0; i < 2; i++ ) array[i]->fun();
```

```
I am Base  
I am Derived
```

Всю работу делает объявление virtual

↪ если убрать `virtual`:

```
I am Base  
I am Base
```

Полиморфизм

Пояснения к примеру

- В базовом классе определена **виртуальная** функция `fun()`, а в классе наследнике эта функция переопределена
- Виртуальность гарантирует, что во время исполнения будет вызвана функция того класса от которого в действительности взят указатель; это называется поздним связыванием
- Если функции не виртуальные, то работает «статическое связывание» и всегда вызывается функция соответствующая точному типу указателя

Формальное определение полиморфизма

- 👉 Объекты с одинаковой спецификацией могут иметь различную реализацию: указатели на базовый класс `Base* array[2]` при вызове `array[i]->fun()` ведут себя по разному

Пример: виртуальная функция area()

```
class Point {  
    ...  
    virtual double area() const {return 0;}  
};  
class Circle : public Point {  
public:  
    Circle(double X=0,double Y=0,double R=0) : Point(X,Y),rad(R) {}  
    virtual double area() const {return M_PI*rad*rad;}  
private:  
    double rad;  
};  
class Rectangle : public Point {  
public:  
    Rectangle(double X=0,double Y=0,double H=0,double W=0):  
        Point(X,Y), height(H), width(W) {}  
    virtual double area() const {return height*width;}  
private:  
    double height, width;  
};
```

... продолжение

```
Point* shapes[3];  
shapes[0] = new Circle(0,0,1.2);  
shapes[1] = new Point(1,2);  
shapes[2] = new Rectangle(1,1,2,3);  
for( int i = 0; i < 3; i++ )  
    cout << " area is " << shapes[i]->area() << endl;
```

area is 4.52389

area is 0

area is 6

Обратите внимание:

👉 **virtual** достаточно указать только в базовом классе, но второе определение в производном классе совершенно безвредно и используется исключительно для улучшения читабельности

Абстрактные классы

- Если мы имеем дело с набором объектов типа «точка», «линия», «окружность» и т.д., то приходим к абстрактному понятию «фигура»
- Фигура настолько абстрактное понятие, что её нельзя ни нарисовать, ни вычислить площадь и «площадь фигуры», функция `area()`, тоже абстрактна
- В C++ можно задать класс с абстрактной виртуальной функцией, и такой класс называется абстрактным, так как **нельзя создать объект этого класса**
- 👉 Абстрактный класс задает интерфейс: список функций которые обязан иметь наследник этого класса

👉 Абстрактная виртуальная функция задаётся инициализацией `=0`

```
virtual double area() const = 0; // pure virtual function
```

Пример: реализации абстрактного класса Figure

```
class figure {
public:
    figure(double X=0, double Y=0) : x(X), y(Y) {}
    virtual ~figure() {}
    virtual void draw() const = 0;
    virtual void hide() const = 0;
    virtual void move(double newx, double newy) {
        hide();
        x = newx; y = newy;
        draw();
    }
    virtual double area() const = 0;
protected:
    double x,y; // X,Y - coordinates
};
```

☞ В классах наследниках все абстрактные виртуальные функции должны быть определены!

```
class point : public figure {
public:
    point(double X, double Y) : figure(X,Y) {}
    virtual void draw() const {cout << "draw point" << endl;}
    virtual void hide() const {cout << "hide point" << endl;}
    virtual double area() const {return 0;}
};

class circle : public figure {
public:
    circle(double X, double Y, double R) : figure(X,Y),rad(R) {}
    virtual void draw() const {cout << "draw circle" << endl;}
    virtual void hide() const {cout << "hide circle" << endl;}
    virtual double area() const {return M_PI*rad*rad;}
private:
    double rad;
};
```

Пример использования

```
figure Ft; // ERROR: abstract type 'figure'  
figure* fig[10];  
fig[0] = new point(1,2);  
fig[1] = new circle(1,1,3);  
for( int i = 0; i < 2; i++ ) {  
    cout << " area is " << fig[i]->area()  
        << endl;  
    fig[i]->move(i+1,i+1);  
}
```

Output

```
area is 0  
hide point  
draw point  
area is 28.2743  
hide circle  
draw circle
```

👉 функции `hide()` и `draw()` вызваны для правильных классов

Оператор `dynamic_cast<>` (Run Time Type Information)

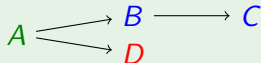
- преобразует указатель (ссылку) базового типа на указатель (ссылку) производного класса:

```
Derivative* pd = dynamic_cast<Derivative*>(pb);
```

```
Derivative& rd = dynamic_cast<Derivative&>(rb);
```

- `dynamic_cast<>` опирается на RTTI, то есть на информацию о каждом полиморфном классе **во время выполнения программы**
- если преобразование невозможно — возвращает нулевой указатель, а для ссылок возбуждает исключение

Пример:



- Объекты всех классов хранятся как набор указателей на базовый класс **A**
- В классах **B** и **C** имеется функция **m()**, которой нет ни в **A** ни **D**
- Как вызвать **m()** для подходящих объектов наиболее простым способом?

...реализации указанной иерархии

```
struct A {  
    virtual void Iam() { cout << "I am A" << endl;}  
};  
  
struct B : public A {  
    virtual void Iam() { cout << "I am B" << endl;}  
    virtual void m() { cout << "fun m() from B" << endl;}  
};  
  
struct C : public B {  
    virtual void Iam() { cout << "I am C" << endl;}  
    virtual void m() { cout << "fun m() from C" << endl;}  
};  
  
struct D : public A {  
    virtual void Iam() { cout << "I am D" << endl;}  
};
```

... вызываем функцию `Iam()` для всех объектов

✓ Механизм виртуальных функций:

```
vector<A*> pa { new A, new B, new C, new D };  
for(auto p: pa) {  
    p->Iam();  
}
```

```
I am A  
I am B  
I am C  
I am D
```

... вызываем функцию `m()` только для ветки *B*

✓ Оператор `dynamic_cast<>`:

```
for(auto p: pa) {  
    B* pb = dynamic_cast<B*>(p);  
    if( pb ) pb->m();  
}
```

```
fun m() from B  
fun m() from C
```

Оператор typeid() (Run Time Type Information)

- `typeid(X)` возвращает объект с информацией о фактическом типе `X`
- возвращаемый объект имеет тип: `std::type_info`
- 👉 необходим заголовочный файл `<typeinfo>`

Пример использования typeid()

```
#include <typeinfo> // for typeid()
for(auto p: pa) {
    cout << typeid(*p).name() << " ";
    if(typeid(*p) == typeid(B)) ((B*)(p))->m();
    else if(typeid(*p) != typeid(D)) p->Iam();
    else cout << endl;
}
```

Output

```
1A I am A
1B fun m() from B
1C I am C
1D
```

- 👉 Наиболее часто `typeid` используют для сравнения с `typeid()` тестируемого класса

Обратите внимание

- В `typeid()` можно использовать имя класса: `typeid(*p)==typeid(A)`
- Константность игнорируется: `typeid(const T) == typeid(T)`
- Функция `name()` возвращает текст включающий имя класса:
содержимое текста зависит от компилятора
- Тип указатель на класс и тип самого класса различаются:
`typeid(*p).name()` вернет `1A`
`typeid(p).name()` вернет `P1A`
- `typeid(p)` для нулевого указателя возбуждает исключение `bad_typeid`

Замечания к использованию `dynamic_cast<>` и `typeid()`

- RTTI может не поддерживаться: `-fno-rtti`
- В случае сложного разветвленного наследования их вызовы «дороги»
- Вместо RTTI рекомендуется использовать полиморфизм и шаблоны

Виртуальный деструктор

👉 в базовом классе **рекомендуется** объявлять деструктор со спецификацией **virtual**

Пример

```
class Base {  
    // ~Base();          // 1-st case  
    virtual ~Base(); // 2-nd case - recommended  
};  
  
class Derived : public Base {  
    ~Derived() { // Do some important cleanup }  
};  
  
Base *ptr = new Derived();  
...  
delete ptr;    // 1) only ~Base() will be called  
               // 2) and ~Derived() and ~Base() will be called
```

«Виртуальные» friend-функции

- ✎ Виртуальными могут быть только **не статические** функции члены класса
- ✎ Конструкторы, статические и friend-функции **нельзя объявить виртуальными**

Иногда требуется подобная функциональность

Например для оператора вывода:

```
for( int i = 0; i < 2; i++ ) {  
    fig[i]->move(i+1,i+1);  
    cout << " fig[" << i <<"]= " << *fig[i] << endl;  
}
```

- ✗ Определив `operator«()` для `figure` мы сможем увидеть только «базовую» информацию

✎ Тем не менее решение существует ...

«Виртуальный» operator«()

```
class figure {  
    friend ostream& operator<<(ostream& out, const figure& f) {  
        out << f.ToStr(); }  
private: // или в public  
    virtual string ToStr() const = 0;  
};  
  
class point : public figure {  
    virtual string ToStr() const {  
        stringstream ss;  
        ss << "point(" << x << "," << y << ")";  
        return ss.str(); }  
};  
  
class circle : public figure {  
    virtual string ToStr() const {  
        stringstream ss;  
        ss << "circle(" << x << "," << y << ":" << rad << ")";  
        return ss.str(); }  
};
```

Проверяем

```
figure* fig[10];  
fig[0] = new point(1,-1);  
fig[1] = new circle(1,1,3);  
for( int i = 0; i < 2; i++ ) {  
    fig[i]->move(i+1,i+1);  
    cout << "fig[" << i <<"]= " << *fig[i]  
        << endl;  
}
```

Output

```
hide point  
draw point  
fig[0]= point(1,1)  
hide circle  
draw circle  
fig[1]= circle(2,2:3)
```

Обратите внимание:

- 👉 приватную виртуальную функцию можно переопределить в производном классе

Контроль за виртуальными функциями

(C++11)

Служебное слово `override` в декларации функции гарантирует:

- ☞ функция переопределяет существующую виртуальную функцию из базового класса

Пример: гарантированное переопределение функций с `override`

```
class Base {  
    virtual int fun1(double a);  
    virtual void fun2() const;  
    void fun3();  
};  
  
class Derived : public Base { // обратите внимание где стоит override  
    void fun1(double x) override; // OK! override Base::fun1  
    int fun1(int i) override; // ERROR: signature mismatch  
    void fun2() override; // ERROR: Base::fun2 is const  
    void fun3() override; // ERROR: Base::fun3 is not virtual  
};
```

Прерывание «наследования»: `final`

👉 `final` в декларации функции:

- 1 проверяет, что функция переопределяет виртуальную функцию
- 2 функция не может быть переопределена в производных классах

👉 `final` в декларации класса: этот класс не может быть базовым

Пример: обратите внимание где стоит `final`

```
class Base {  
    virtual void fun();  
};  
class A : public Base {  
    void fun() final; // OK: fun is overridden and it is the final!  
void fun2() final; // ERROR: non-virtual function  
};  
class B final : public A {  
void fun() override; // ERROR: A::fun() cannot be overridden  
};  
class C : public B {}; // ERROR: B is final
```