

```

void Point::Show()
{
    Visible = true;
    putpixel (X,Y,getcolor());
}

void Point::Hide()
{
    Visible = false;
    putpixel (X,Y,getbkcolor());
}

void Point::MoveTo (int newX, int newY)
{
    Hide ();
    X = newX;
    Y = newY;
    Show ();
}

```

что произойдет, если мы не будем перегружать этот метод?

```

void Circle::Show ()
{
    Visible = true;
    circle (X,Y, Radius);
}

void Circle::Hide ()
{
    Visible = false;
    unsigned int tempColor = getcolor ();
    setcolor (getbkcolor());
    circle (X,Y, Radius);
    setcolor (tempColor);
}

void Circle::Expand (int deltaR)
{
    Hide();
    Radius += deltaR;
    Show();
}

void Circle::Contract (int deltaR)
{
    Expand (-deltaR);
}

void Circle::MoveTo (int newX, int newY)
{
    Hide ();
    X = newX;
    Y = newY;
    Show ();
}

```

Раннее или позднее связывание

- В случае раннего связывания адреса всех функций и процедур известны в тот момент, когда происходит компоновка программы.
- В случае позднего связывания адрес процедуры не связывается с обращением к ней до того момента, пока обращение не произойдет фактически, т. е. во время выполнения программы.

Виртуальные функции

- *Виртуальный метод* – метод, который может быть переопределён в классах-наследниках так, что конкретная реализация метода для вызова будет определяться во время исполнения.
- Виртуальные методы описываются с помощью ключевого слова *virtual* в базовом классе
- Виртуальные методы не могут быть статическими
- Виртуальные методы позволяют создавать различные версии функции для базового и производных классов, однако должны иметь всегда одинаковое число и одинаковые типы параметров. В противном случае функция будет перегруженной, а не виртуальной.
- Когда виртуальная функция вызывается с указанием области видимости, например, `Point::Show()`, виртуальный механизм не применяется.

Полиморфизм и виртуальные методы

```
class Point
{
    ...
public:
    ...
    virtual void Show ();
    virtual void Hide ();
    void MoveTo (int newX, int newY);
};
...
реализация методов класса Point
...
class Circle: public Point
{
    ...
public:
    ... // без метода MoveTo()
    virtual void Show ();
    virtual void Hide ();
};
...
реализация методов класса Circle
...
```

```
Point pointObj (100,20); // объект базового класса
Circle circleObj (20,30,10); // объект производного класса
Point *pointPtr; // указатель базового класса
pointPtr = & pointObj; // указывает на объект базового класса
pointPtr = & circleObj; // указывает на объект производного класса
pointPtr = & pointObj; pointPtr->MoveTo(10,10);
pointPtr = & circleObj;
pointPtr -> Expand(12);
pointPtr = & pointObj;
pointPtr->Show(10,10); // ВЫЗОВ Show() объекта pointObj класса
    Point
pointPtr = & circleObj;
pointPtr->Show(10,10); // ВЫЗОВ Show() объекта circleObj класса
    Circle

void JumpFigure (Point* AnyFigure, int h)
{
    int oldX = AnyFigure->GetX();
    int oldY = AnyFigure->GetY();
    delay(100); // временная задержка на 0.1 сек
    AnyFigure->MoveTo (oldX, oldY-h); // "прыжок"
    delay(100); // временная задержка на 0.1 сек
    AnyFigure->MoveTo (oldX, oldY); // на исходную позицию
}
```

Абстрактный класс

- **Полиморфный класс** - класс, содержащий хотя бы один виртуальный метод.

- **Чисто виртуальные методы** не определяются в базовом классе. У них нет тела, а есть только декларации об их существовании.

Для чисто виртуальной функции используется общая форма

virtual тип имя_функции (список параметров) = 0;

- Класс, содержащий хотя бы один виртуальный метод, называется **абстрактным классом**.

- Абстрактные классы не бывают изолированными, т.е. всегда абстрактный класс должен быть наследуемым. Поскольку у чисто виртуального метода нет тела, то создать объект абстрактного класса невозможно.

• Соберите все общие атрибуты в один абстрактный класс, и определите иерархию классов так, чтобы все общие элементы использовались из **ЭТОГО** класса

```
class Point
{
protected:
    int X; int Y; Boolean Visible;
public:
    int GetX(void) { return X; }
    int GetY(void) { return Y; }
    Boolean isVisible () { return Visible; }
    Point (int newX =0, int new Y =0);
    virtual void Show() = 0; // чисто виртуальная функция
    virtual void Hide() = 0; // чисто виртуальная функция
    void MoveTo (int newX, int newY)
    {
        Hide();
        X = newX; Y = newY;
        Show();
    }
};
```

Принцип инверсии зависимостей

- Высокоуровневые модули не должны зависеть от низкоуровневых. И те, и другие должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей; вместо этого детали должны зависеть от абстракций.

Корнями иерархий должны быть только абстрактные классы. Абстрактные базовые классы должны беспокоиться об определении функциональности, но не о ее реализации.

Три фундаментальных преимущества при проектировании:

- *Повышение надежности.*
- *Повышение гибкости*
- *Хорошая модульность.*

Перекрытие виртуальных функций

- Определение в производном классе перекрытия, которое может быть неуспешным (например, генерировать исключения), будет корректно только в том случае, когда в базовом классе не объявлено, что данная операция всегда успешна.
- Никогда не изменяйте аргумент по умолчанию при перекрытии

```
class Base {  
    // ...  
    virtual void Foo( int x = 0 );  
};  
class Derived : public Base {  
    // ...  
    virtual void Foo( int x = 1 ); // Лучше так не делать...  
};
```

```
Derived *pD = new Derived;  
pD->Foo(); // Вызов pD->Foo(l)  
Base *pB = pD;
```


- Желательно добавлять ключевое слово `virtual` при перекрытии функций, несмотря на его избыточность — это сделает код более удобным для чтения и понимания.

- Не забывайте о том, что перекрытие может скрывать перегруженные функции из базового класса

```
class Base {  
    // ...  
    virtual void Foo( int );  
    virtual void Foo( int, int );  
    void Foo( int, int, int );  
};  
class Derived : public Base {  
    // ...  
    virtual void Foo( int ); // Перекрывает Base::Foo(int), скрывая остальные функции  
};
```

```
Derived d; d.Foo( 1 ); // Все в порядке  
d.Foo( 1, 2 ); // Ошибка  
d.Foo( 1, 2, 3 ); // Ошибка
```

- Если перегруженные функции из базового класса должны быть видимы, воспользуйтесь объявлением `using` для того, чтобы повторно объявить их в производном классе:

```
class Derived : public Base {  
    // ...  
    virtual void Foo( int ); // Перекрытие Base::Foo(int)  
    using Base::Foo; // Вносит все прочие перегрузки Base::Foo в область видимости  
};
```

- Виртуальные функции стоит делать неоткрытыми, а открытые — не виртуальными (исключение - деструкторы)

Открытая виртуальная функция по своей природе решает две различные параллельные задачи:

Она определяет интерфейс. Будучи открытой, такая функция является непосредственной частью интерфейса класса, предоставленного внешнему миру.

Она определяет детали реализации. Будучи виртуальной, функция предоставляет производному классу возможность заменить базовую

- Деструкторы базовых классов стоит делать открытыми и виртуальными либо защищенными и не виртуальными

Пример 1. Базовые классы с полиморфным удалением. Если должно быть разрешено полиморфное удаление, то деструктор должен быть открытым (в противном случае вызывающий код не сможет к нему обратиться) и должен быть виртуальным (в противном случае его вызов приведет к неопределенному поведению).

Пример 2. Базовые классы без полиморфного удаления (например, классы стратегий). Если полиморфное удаление не должно быть разрешено, деструктор должен не быть открытым (чтобы вызывающий код не мог к нему обратиться) и не должен быть виртуальным (потому что виртуальность ему не нужна).

- Избегайте вызовов виртуальных функций в конструкторах и деструкторах

При таком вызове функции теряют виртуальность. Более того, вызов чисто виртуальной нереализованной функции приводит к неопределённому поведению.

Множественное наследование

```
class task
{
public:
void trace ();
// ...
};
class displayed
{
public:
void trace ();
// ...
};
class my_task: public task {
// эта задача не изображается
// на экране, т.к. не содержит класс displayed
// ...
};
class my_displayed: public displayed
{
// а это не задача
// т.к. не содержит класс task
// ...
};
```

```
class my_displayed_task:public displayed, public task
{
// в этом классе trace () не определяется
};
void g ( my_displayed_task * p )
{
p -> trace (); // ошибка: неоднозначность
}
```

```
class my_displayed_task:public displayed, public task
{
// ...
public:
void trace ()
{
// текст пользователя
displayed::trace (); // ВЫЗОВ trace () из displayed
task::trace (); // ВЫЗОВ trace () из task
}
// ...
};
void g ( my_displayed_task * p )
{
p -> trace (); // теперь нормально
}
```

Устранение неопределённости

```
class A
{
public:
void show() { cout << "Класс A\n"; }
};
```

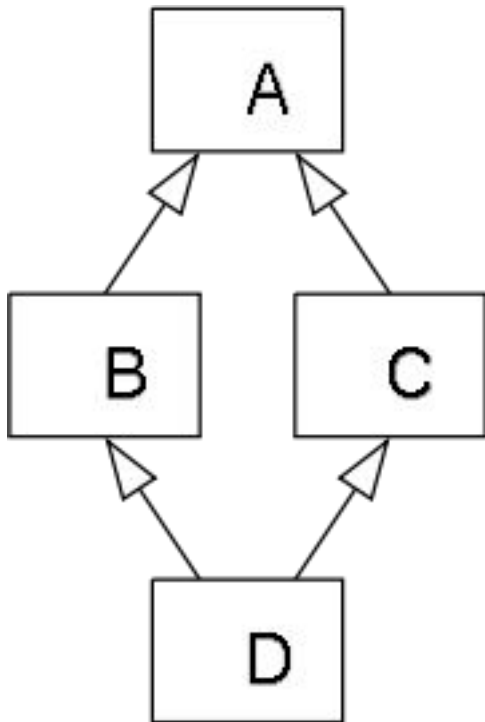
```
class B
{
public:
void show() { cout << "Класс B\n"; }
};
```

```
class C : public A, public B
{
};
```

```
C objC; // объект класса C
// objC.show(); // так делать нельзя - программа не скомпилируется
objC.A::show(); // так можно
objC.B::show(); // так можно
```

Ромбовидное наследование

Проблема ромба
("diamond problem")



- Члены-данные класса A в объектах класса D в двух экземплярах
- Обращение к членам класса A в классе D невозможно без уточнения непосредственного наследника (B или C)
- Порядок наследования может привести к изменению семантики
- Решение

```
class B : virtual public A
```

Таблица виртуальных функций

```
class A{
    int foo();
    int a1;
    int a2;
    int a3;
}

class B{
    int bar();
    int b1;
    int b2;
    int b3;
}

class D : public A, public B{
    int foo();
    int d1;
    int d2;
    int d3;
}

/* псевдокод
struct D{  int a1;    int a2;    int a3;
           int b1; int b2;    int b3;    int d1;
           int d2; int d3;
}; */

A a;
B b;
D d;
A *pA = new D;

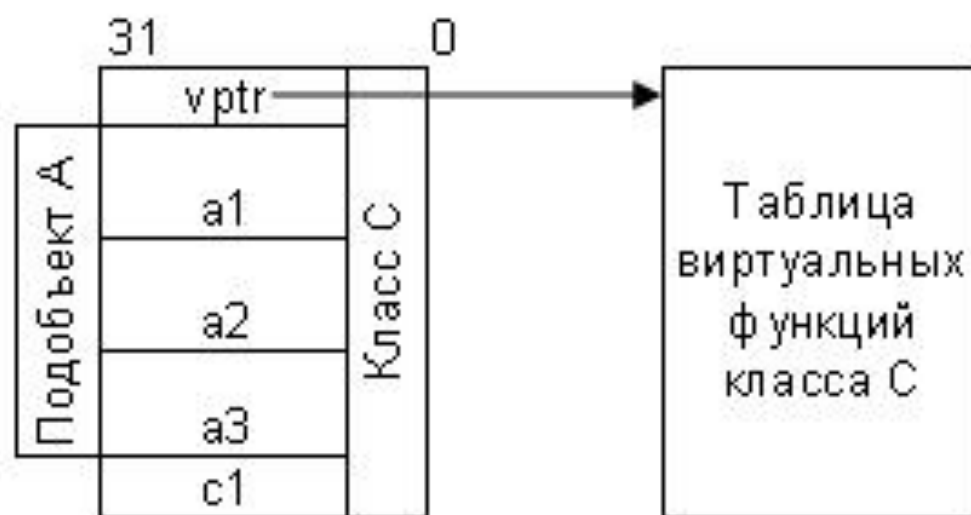
a.foo();
b.bar();
d.foo();
d.bar(); // псевдокод B::bar(&d);
          //ВЫЗОВ B *p = (B*)&d;
pA->foo(); // псевдокод A::foo(&d);
```

- Компилятор добавляет во все полиморфные классы один дополнительный указатель. Этот указатель содержит адрес таблицы виртуальных методов (ТВМ) - массив, каждый элемент которого содержит указатель на функцию.

```
class Vbase{
    virtual void foo();
    virtual void bar();
    void do() { foo();}
    int v;
} // в псевдокоде class Vbase{ void *m_pVptr; int v;}
```

```
class V : public Vbase{
    virtual void foo();
    virtual void alpha();
}
```

```
V v; // Vbase() // V()
Vbase *pbase = &v;
v.foo();
pbase->foo();
v.bar(); // Vbase::do() //this->foo()
pbase->do();
```

Виртуальные конструкторы и деструкторы?

Попробуйте найти ошибку в коде:

```
1. class A{  
public:  
    A(){foo();};  
    virtual void foo() = 0;  
};  
class C : public A{  
public:  
    C() : A(){};  
    virtual void foo(){};  
};
```

```
2. A *pa = new B; // A содержит неvirtуальный деструктор  
delete pa;
```