

Информатика

Классы. Наследование.

Наследование

Наследование – механизм языка, заключающийся в возможности создания иерархии классов. При этом потомки включают в себя все поля и методы предка.

Класс-наследник может переопределить методы предка и добавить собственные. Т. е. он изменяет и/или расширяет функционал родителя.

Наследование

Класс-предок обычно называется родительским или базовым классом, а класс-наследник – дочерним или производным.

Наследование

Базовые классы объединяют в себе наиболее общие черты для производных классов.

При продвижении по иерархии наследования от базовых классов к производным классы приобретают все больше конкретных черт.

Наследование

Объект производного класса можно трактовать как объект базового класса, с дополнительной информацией, характерной для производного класса.

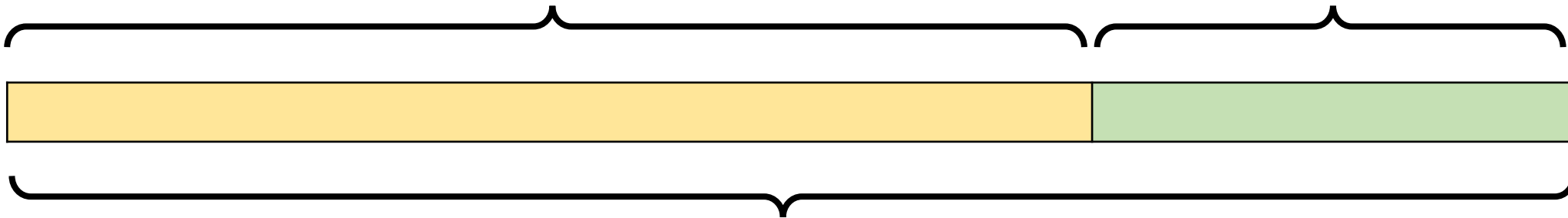
Наследование

```
class Base { ... };
```

```
class Der : public Base { ... };
```

Базовая часть

Данные
производного класса



Объект производного класса

Наследование

Виды наследования:

- Одиночное (простое);
- Множественное.

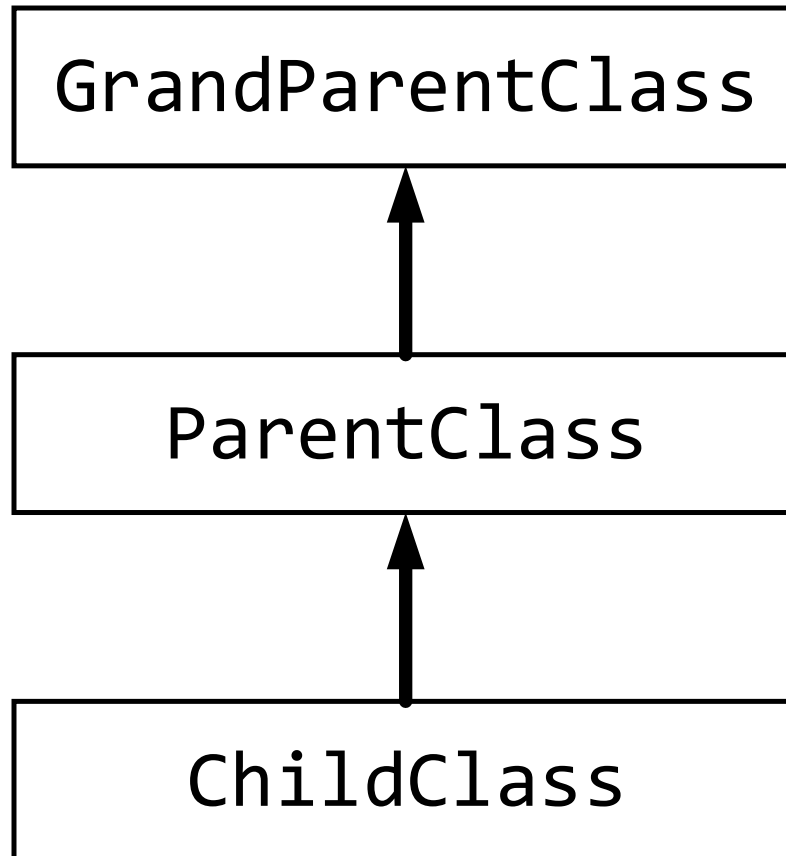
Наследование

При одиночном наследовании каждый производный класс имеет только одного непосредственного предка. При этом число производных классов может быть любым.

При множественном наследовании производный класс имеет двух и более предков. При этом число производных классов может быть любым.

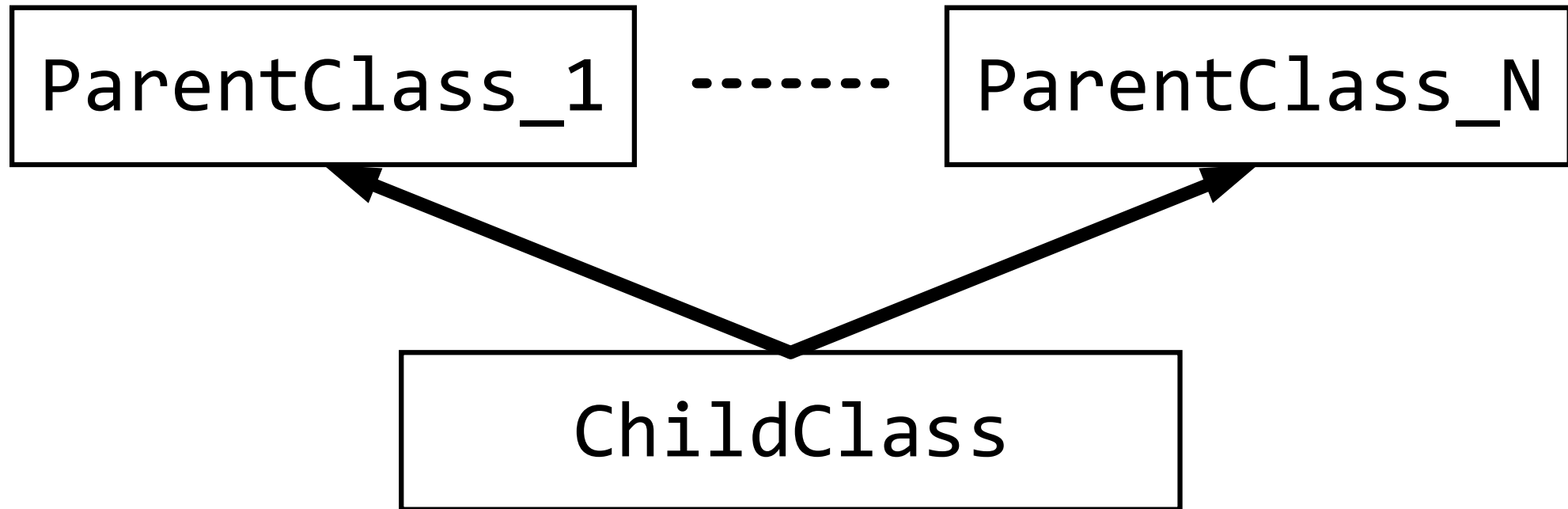
Наследование

Одиночное наследование:



Наследование

Множественное наследование:



Наследование

При описании класса в его заголовке после : перечисляются все классы, являющиеся для него базовыми. Возможность обращения к элементам базовых классов регулируется при помощи спецификаторов доступа `public`, `private` и `protected`.

```
class DerClass : [private|protected|public] BaseClass
{
    ...
}
```

Наследование

Если базовых классов несколько, то они перечисляются через запятую. Спецификатор доступа может стоять перед именем каждого базового класса. Если спецификатор доступа не указан, то считается, что указан `private`.

```
class Der : Base1, public Base2, protected Base3
{
    ...
}
```

Спецификаторы доступа

Спецификатор доступа	Спецификатор доступа в базовом классе	Доступ в производном классе
private	private	Нет
	protected	private
	public	private
protected	private	Нет
	protected	protected
	public	protected
public	private	Нет
	protected	protected
	public	public

Спецификаторы доступа

Вне зависимости от спецификатора доступа `private` поля и методы базового класса не доступны в производном. Обращаться к ним следует через `public` или `protected` методы базового класса.

Поля и методы, объявленные со спецификатором `protected` доступны в производном классе.

Доступ к `public` элементам становится соответствующим спецификатору доступа, указанному при наследовании.

Спецификаторы доступа

При **public** наследовании объект производного класса является и объектом базового класса.

При **private** и **protected** наследовании объект производного класса подобен объекту базового класса.

Пользователи производного класса не имеют доступа к его базовой части. Обращение к **public** полям и методам базового класса возможно лишь из методов производного класса. 15

Спецификаторы доступа

Далее разбирается случай одиночного `public` наследования, если не указано иное.

Спецификаторы доступа

```
class A { ... };  
class B : public A { ... };
```

```
B b;  
A a = b; //С помощью конструктора копирования  
          базового класса A из объекта b  
          производного класса копируется базовая часть
```

```
class C : private A { ... };
```

```
C c;  
A a2 = c;      Ошибка компиляции
```

Иерархия наследования

При создании классов, входящий в иерархию наследования, производные классы могут обращаться к полям и методам базового класса, если это позволяют спецификаторы доступа.

Т. е. эти поля и методы «наследуются» производным классом.

Иерархия наследования

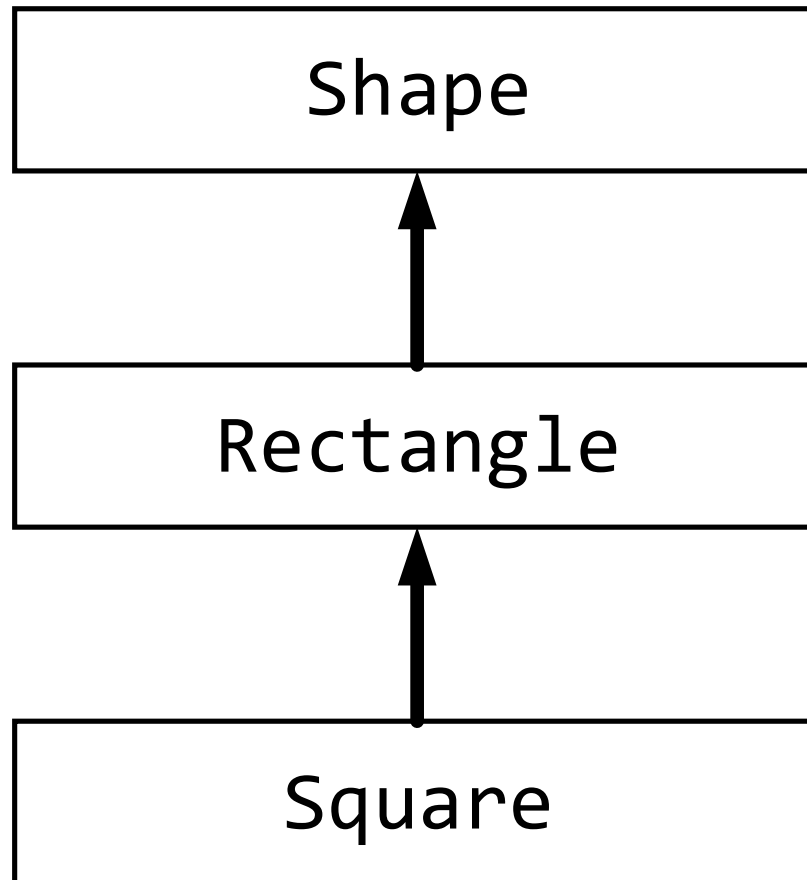
Однако, существует ряд методов, которые не могут наследоваться:

- Конструкторы;
- Оператор присваивания;
- Деструктор.

Поэтому данные методы, при необходимости, должны быть переопределены в производных классах.

Иерархия наследования

Создадим классы, описывающие геометрические фигуры.



Иерархия наследования

```
class Shape {  
protected:  
    double area;  
public:  
    Shape() { this->area = 0; }  
    Shape(double S) { this->area=S; }  
    ~Shape() {};  
};
```

Иерархия наследования

```
class Rectangle : public Shape {  
protected:  
    double a;  
    double b;  
public:  
    Rectangle() { this->a=this->b=0; }  
    Rectangle(double A, double B){    this->a=A;  
                                       this->b=B;  
                                       this->area = A*B;    }  
    ~Rectangle() {}  
};
```

Иерархия наследования

```
class Square : public Rectangle {  
public:  
    Square() {}  
    Square(double C) {this->a = this->b = C;  
                      this->area = C*C;}  
    ~Square() {}  
};
```

Иерархия наследования.

Конструкторы

При создании объекта производного класса сначала вызываются конструкторы базовых классов, начиная с самого верхнего уровня, а затем конструктор производного класса.



Такой вызов конструкторов необходим, потому что в конструкторе производного класса возможно обращение к полям или методам базового класса. Следовательно, на момент вызова конструктора производного класса, объект базового класса должен быть создан и проинициализирован.

Иерархия наследования.

Конструкторы

```
Rectangle r(1,2);  
Square s(3);
```

Порядок вызова конструкторов при создании r:
Shape()  Rectangle(double, double)

Порядок вызова конструкторов при создании s:
Shape()  Rectangle()  Square(double)

Иерархия наследования.

Конструкторы

При создании объекта производного класса, если не прописано иное, вызываются конструкторы по умолчанию базовых классов.

Для того, чтобы вызвать конструкторы с параметрами базовых классов, необходимо после имени конструктора производного класса в списке инициализации указать вызов требуемого конструктора базового класса.

Иерархия наследования.

Конструкторы

```
class Shape {  
    double area;  
public:  
    Shape() { this->area = 0; }  
    Shape(double S) { this->area=S; }  
    ~Shape() {};  
};
```

Иерархия наследования.

Конструкторы

```
class Rectangle : public Shape {  
    double a;  
    double b;  
public:  
    Rectangle() { this->a = this->b = 0;}  
    Rectangle(double A, double B) : Shape(A*B) {  
        this->a=A;  
        this->b=B;  
    }  
    ~Rectangle() {}  
};
```

Иерархия наследования.

Конструкторы

```
class Square : public Rectangle {  
public:  
    Square() {}  
    Square(double C) : Rectangle(C, C) {}  
    ~Square() {}  
};
```

Иерархия наследования.

Конструкторы

```
Rectangle r(1,2);
```

```
Square s(3);
```

Порядок вызова конструкторов при создании r:

Shape(double)  Rectangle(double, double)

Порядок вызова конструкторов при создании s:

Shape(double)  Rectangle(double, double)  Square(double)₃₀

Иерархия наследования. Конструкторы

При наличии в составе производного класса полей, являющихся объектами, сначала вызывается конструктор базовых классов, затем конструкторы для полей производного класса, а затем конструктор производного класса.

Иерархия наследования.

Конструкторы

```
class A {  
    ...};
```

```
class B {  
    ...};
```

```
class C {  
    A a;  
    B b;  
    ...};
```

```
class D : public C {  
    A a;  
    ...};
```


Иерархия наследования. Конструкторы

D d;

Порядок вызова конструкторов:

A() -> B() -> C() -> A() -> D()



Создается объект C

Иерархия наследования. Конструктор копирования

Также как и для классов не входящих в иерархию наследования, для классов в иерархии по умолчанию создаются конструктор копирования и оператор присваивания, которые выполняют побайтовое копирование содержимого объекта.

Иерархия наследования.

Конструктор копирования

При необходимости создания собственного конструктора копирования нужно иметь ввиду, что при этом автоматически вызывается не конструктор копирования для базовой части, а конструктор по умолчанию базового класса.

Позаботиться о вызове конструктора копирования для базовой части класса должен программист.

Иерархия наследования.

Конструктор копирования

```
class A {  
    ...  
public:  
    A() { ... }  
    A(int Val) { ... }  
    A(const A& a) { ... }  
};  
  
class B : public A {  
    ...  
public:  
    B() { ... }  
    B(int Val) { ... }  
    B (const B& b) : A(b) { ... }  
};
```

Иерархия наследования. Оператор присваивания

Аналогично классам, не входящим в иерархию наследования, для классов в иерархии в некоторых случаях по умолчанию создаются оператор присваивания, которые выполняют побайтовое копирование содержимого объекта.

Иерархия наследования. Оператор присваивания

При необходимости создания собственного оператора присваивания нужно иметь в виду, что при этом не вызывается оператор присваивания для базовой части класса.

Позаботиться о вызове оператора присваивания для базовой части класса должен программист.

Иерархия наследования. Оператор присваивания

```
A& A::operator=(const A& a) {  
    if (this != &a) {  
        ...  
    }  
    return *this;  
}
```

Иерархия наследования. Оператор присваивания

```
B& B::operator=(const B& b) {  
    if (this != &b) {  
        static_cast<A&>(*this) = b;           //Присваивание  
                                                базовой части  
  
        //Эквивалентно  
        //(A&)(*this) = b;  
        //A::operator=(b);  
        //*static_cast<A*>(this) = b;  
        ...  
    }  
    return *this;  
}
```


Иерархия наследования.

Деструкторы

Деструкторы для объекта производного класса вызываются в порядке, обратном вызову конструкторов. Т. е. сначала вызывается деструктор производного класса, а затем – деструктор базового.


Иерархия наследования. Деструкторы

Такой порядок вызова деструкторов необходим, потому что в деструкторе производного класса возможно обращение к полям и методам базового класса. Поэтому на момент вызова деструктора производного класса, объект базового класса должен существовать.

Иерархия наследования. Деструкторы

```
{  
    Rectangle r(1,2);  
    Square s(3);  
}
```

Порядок вызова деструкторов для r:

~Rectangle()  ~Shape()

Порядок вызова деструкторов для s:

~Square()  ~Rectangle()  ~Shape()

Иерархия наследования. Доступ к полям и методам базового класса

К полям и методам базового класса, объявленных со спецификаторами `public` или `protected` можно обратиться из методов производного класса.

При этом, если поля или методы с одинаковыми именами присутствуют и в базовом и в производном классе, то для обращения к ним нужно перед именем поля/метода указать имя класса и оператор разрешения области видимости.

При этом после инициализации объекта возможен прямой вызов конструктора для базовой части класса.

Иерархия наследования. Доступ к полям и методам базового класса

```
class A {  
protected:  
    int a;  
public:  
    A() { this->a = 1; }  
    A(int Val) { this->a = Val; }  
    void print(){std::cout<<"A::a="<<this->a<<std::endl;}  
};
```

Иерархия наследования. Доступ к полям и методам базового класса

```
class B : public A {  
    int a;  
  
public:  
    B() { this->a = 2; this->A::a = 3; }  
    B(int Val) { this->a = val; this->A::A(2); }  
    void print() {  
        this->A::print();  
        std::cout << "B::a=" << this->a << std::endl;  
    }  
};
```

Иерархия наследования. Доступ к полям и методам базового класса

```
int main() {  
    B b;  
    b.print();           A::a=3  
                          B::a=2  
  
    b.A::A(5);  
    b.A::print();        A::a=5  
    return 0;  
}
```

Иерархия наследования. Доступ к полям и методам базового класса

Можно запретить прямое создание объектов базового класса, поместив конструкторы в секцию `protected` или `private`.

При этом, если конструкторы объявлены со спецификатором `protected`, то возможно создание объектов базового класса посредством методов производного класса.

Иерархия наследования. Доступ к полям и методам базового класса

```
class A {  
protected:  
    int a;  
    A() { this->a = 1; }  
    A(int Val) { this->a = Val; }  
public:  
    void print(){std::cout<<"A::a="<<this->a<<std::endl;}  
};
```

Иерархия наследования. Доступ к полям и методам базового класса

```
class B : public A {  
    int a;  
public:  
    B() { this->a = 2; this->A::a = 3; }  
    B(int Val) { this->a = Val; this->A::A(2); }  
    void print() {  
        this->A::print();  
        std::cout << "B::a=" << this->a << std::endl;  
    }  
    A GetA() { return (A)(*this); }  
};
```

Иерархия наследования. Доступ к полям и методам базового класса

```
int main() {  
    B b;  
    b.print();           A::a=3  
  
    b.A::print();        B::a=2  
    A a = b.GetA();      A::a=3  
    a.print();           A::a=3  
    return 0;  
}
```

Размер классов, входящих в иерархию наследования

Поскольку при наследовании любой производный класс содержит базовую часть (т. е. объект родительского класса), то при вычислении его размера, учитывается также размер родительского класса.

Таким образом, размер производного класса не меньше суммы входящих в него полей и размера базового класса.

Размер классов, входящих в иерархию наследования

```
class Shape {  
    double area;  
... };
```

```
class Rectangle : public Shape {  
    double a;  
    double b;  
...};
```

```
class Square : public Rectangle { ... };
```

Размер классов, входящих в иерархию наследования

```
std::cout << sizeof(Shape) << " "  
<< sizeof(Rectangle) << " "  
<< sizeof(Square) << std::endl;
```

Результат:

8 24 24

Полиморфизм

Полиморфизм – способность объектов во время выполнения вести себя различным образом.

Полиморфизм можно представить как свойство классов, входящих в иерархию наследования, иметь одинаковые методы, которые будут работать по-разному в зависимости от конкретного типа объекта.

Полиморфизм

Полиморфизмом могут обладать только методы класса.

Механизм полиморфизма задействуется только при вызове метода посредством адреса объекта (т. е. с помощью указателя или ссылки).

При вызове метода посредством объекта компилятор генерирует обычный вызов метода класса.

Использование полиморфизма может привести к увеличению объема необходимой памяти и времени выполнения.

Полиморфизм. Зачем это нужно?

Предположим, что мы хотим нарисовать фигуру. При этом хотелось бы написать общую функцию для всех фигур вида:

```
void Draw(Shape* Sh) {  
    Sh->draw();  
}
```

А вызывать её уже для конкретных типов фигур:

```
Rectangle r(1, 2);
```

```
Square s(3);
```

```
Draw(&r);
```

```
Draw(&s);
```

Проблема в том, что тогда будет вызываться метод `draw` из класса `Shape`, а не конкретного объекта (`Rectangle` или `Square`), который мы передали. Чтобы исправить эту ситуацию раньше нам пришлось бы перегружать функцию `Draw` для каждого производного класса.

Но делать нам этого не придётся, потому что есть позднее связывание и виртуальные функции!

Раннее и позднее связывание

Язык C++ поддерживает механизмы раннего и позднего связывания.

Раннее связывание - объект и вызов функции связываются между собой на этапе компиляции. Вся необходимая информация для того, чтобы определить, какая именно функция будет вызвана, известна на этапе компиляции программы.

Раннее связывание

Примеры раннего связывания:

- Вызов методов класса;
- Вызов перегруженных методов класса.

Достоинства:

- Более высокое быстродействие в сравнении с поздним связыванием;
- Меньший объем необходимой памяти в сравнении с поздним связыванием.

Недостатки:

- Меньшая гибкость в сравнении с поздним связыванием;
- В некоторых случаях более сложная логика программы.

Раннее связывание

```
class Shape {  
    ...  
public:  
    void drawNV() { std::cout << "Shape:: drawNV" << std::endl; }  
};  
  
class Rectangle : public Shape {  
    ...  
public:  
    void drawNV() { std::cout << "Rectangle:: drawNV" << std::endl; }  
};  
  
class Square : public Rectangle {  
    ...  
public:  
    void drawNV() { std::cout << "Square:: drawNV" << std::endl; }  
};
```

Раннее связывание

Поскольку все объекты, входящие в иерархию наследования могут быть безопасно приведены к типу базового класса, то для общности в функцию можно передавать указатель или ссылку на объект базового класса.

```
void DrawNV(Shape* Sh) {  
    Sh->drawNV();  
}
```

Раннее связывание

Однако, при такой реализации функции вследствие раннего связывания всегда будет вызываться метод `drawNV` для класса `Shape`.

```
Rectangle r(1, 2);
```

```
Square s(3);
```

```
DrawNV(&r);
```

```
DrawNV(&s);
```

```
Shape::drawNV
```

```
Shape::drawNV
```

Раннее связывание

Тогда, в функцию помимо указателя (или ссылки) на объект нужно передавать значение, описывающее его тип:

```
enum ObjType { SHAPE, RECTANGLE, SQUARE };
```

```
void DrawNV2(Shape* Sh, ObjType Type) {  
    switch (Type) {  
        case SHAPE:      Sh->drawNV(); break;  
        case RECTANGLE:  static_cast<Rectangle*>(Sh)->drawNV(); break;  
        case SQUARE:     static_cast<Square*>(Sh)->drawNV(); break;  
    }  
}
```

Раннее связывание

```
Rectangle r(1, 1);
```

```
Square s(2);
```

```
DrawNV2(&r, RECTANGLE);    Rectangle::drawNV
```

```
DrawNV2(&s, SQUARE);       Square::drawNV
```

Такой способ вызова функций неудобен,
поскольку нужно знать точный тип объекта.

Позднее связывание

Упростить реализацию можно с использованием механизма позднего связывания.

Позднее связывание - объект и вызов функции связываются между собой на этапе выполнения программы.

Достоинства:

- Упрощение логики программы;
- Большая гибкость в сравнении с ранним связыванием.

Недостатки:

Увеличение объема необходимой памяти;

Снижение быстродействия по сравнению с ранним связыванием.

Виртуальные функции

Для того, чтобы метод класса стал полиморфным при его объявлении необходимо указать ключевое слово `virtual`. Такие методы называются виртуальными функциями.

Виртуальные функции должны иметь одинаковые имена, список аргументов и тип возвращаемого значения. Реализация данных функций может быть различна.

Виртуальные функции

Если объявление и реализация виртуальной функции разнесены, то при реализации ключевое слово `virtual` указывать не нужно.

Если функция объявлена в базовом классе как виртуальная, то она будет таковой и в производных классах. При объявлении такой функции в производном классе указывать ключевое слово `virtual` не обязательно.

Виртуальная функция может не иметь реализации в том классе, где она объявлена. Тогда предполагается, что её реализация обязательно есть у производных классов.

Виртуальные функции

Реализация виртуальной функции может отсутствовать в каком-либо производном классе. При этом будет использована соответствующая виртуальная функция родительского класса.

Виртуальная функция не может быть статическим методом класса.

Конструкторы не могут быть виртуальными.

Виртуальные функции

```
class Shape {  
    ...  
public:  
    virtual void draw() { std::cout << "Shape::draw" << std::endl; }  
    virtual void print() { std::cout << "Shape::print" << std::endl; }  
};  
  
class Rectangle : public Shape {  
    ...  
public:  
    void draw() { std::cout << "Rectangle::draw" << std::endl; }  
};  
  
class Square : public Rectangle {  
    ...  
public:  
    void draw() { std::cout << "Square::draw" << std::endl; }  
};
```

Виртуальные функции

```
void Draw(Shape* Sh) {  
    Sh->draw();  
}
```

...

```
Rectangle r(1, 2);
```

```
Square s(3);
```

```
Draw(&r);
```

```
Draw(&s);
```

```
Rectangle::draw
```

```
Square::draw
```

Виртуальные функции

Виртуальная функция, так же как и любой другой метод базового класса может быть вызвана посредством указания имени класса и оператора разрешения области видимости.

```
void Draw2V(Shape* Sh) {  
    Sh->Shape::draw();  
    Sh->draw();  
}
```

...

```
Rectangle r(1, 2);  
Draw2V(&r);
```

```
Shape::draw  
Rectangle::draw
```

Виртуальные функции

Работа полиморфных методов обеспечивается за счет создания в классе таблицы виртуальных функций. При этом в классе создается поле, недоступное программисту напрямую, в котором содержится указатель на данную таблицу.

Таблица виртуальных функций создается при объявлении в классе виртуальной функции.

Таблица виртуальных функций создается для класса, в котором объявлена виртуальная функция и его производных классов.

Виртуальные функции

Таблица виртуальных функций создается для текущего класса в единственном экземпляре.

В таблицу виртуальных функций заносятся адреса виртуальных функций.

Перегруженные виртуальные функции имеют одинаковые смещения (индексы в таблице) в таблицах виртуальных функций для классов, входящих в иерархию наследования.

Виртуальные функции

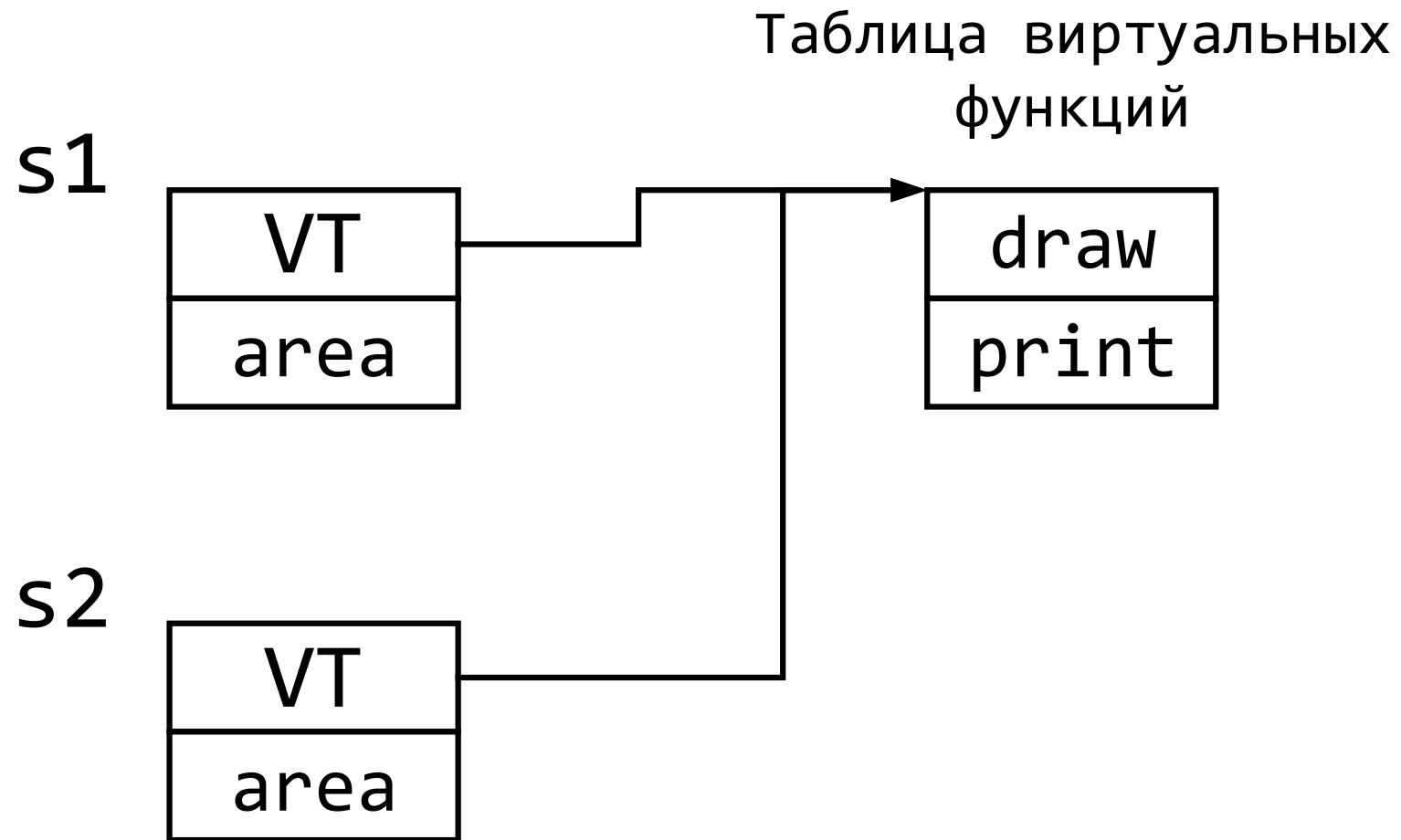
Вызов виртуальной функции посредством указателя на объект:

- Зная адрес объекта, расположение поля, содержащего указатель на таблицу виртуальных функций (VT) и индекс (idx) функции в таблице, компилятор получает адрес вызываемой виртуальной функции $\text{addr} = \text{VT}[\text{idx}]$.
- Происходит вызов функции, располагающейся по заданному адресу. Поскольку вызываемая функция является нестатическим методом класса, то ей передается адрес объекта, для которого она вызывается.

Виртуальные функции

Shape s1;

Shape s2;

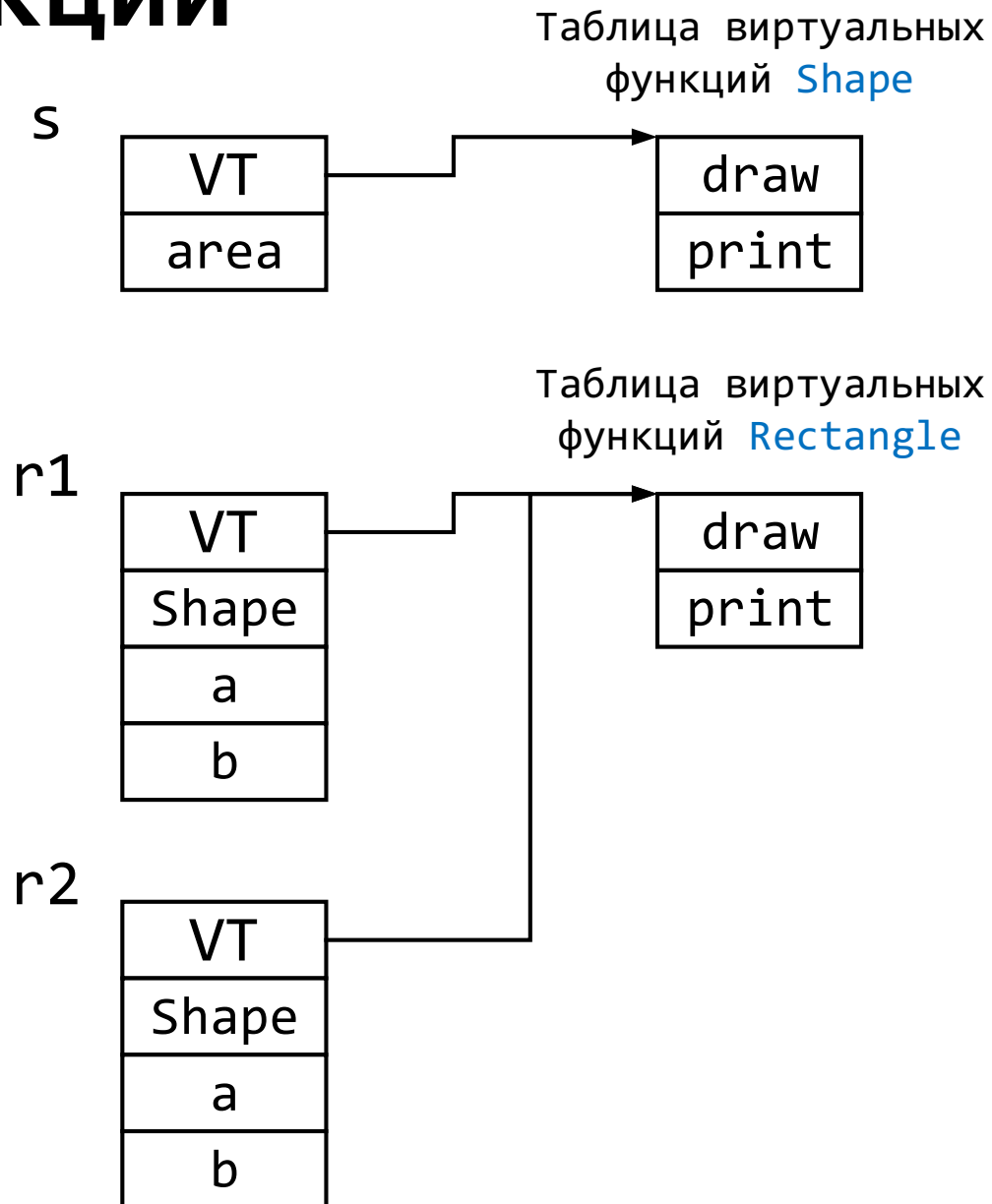


Виртуальные функции

Shape s;

Rectangle r1;

Rectangle r2;



Виртуальные функции

При вызове виртуальной функции компилятору не известен тип объекта.

Для вызова компилятору достаточно знать адрес объекта, расположение указателя на таблицу виртуальных функций и индекс вызываемой функции в данной таблице.

Виртуальные функции

При обладании информацией о расположении поля, содержащего указатель на таблицу виртуальных функций и индексе функции в таблице, можно узнать расположение таблицы виртуальных функций и вызвать функцию посредством указателя.

При использовании компилятора Microsoft поле, содержащее адрес таблицы виртуальных функций является первым полем класса.

Виртуальные функции

```
class Shape {  
    ...  
public:  
    virtual void draw() { ... }  
    virtual void print() { ... }  
};
```

Виртуальные функции

Способ для x86 приложений

```
typedef void(__thiscall *VIRTUAL_FUN)(void*);
```

```
int main() {  
    Shape s;  
    char* s_addr = (char *)&s;  
    void** VT = (void**)(*(unsigned long*)s_addr);           адрес таблицы  
                                                             виртуальных функций  
    void* draw_addr = VT[0];                                  адрес метода draw  
    void* print_addr = VT[1];                                  адрес метода print  
  
    ((VIRTUAL_FUN)draw_addr)(&s);                             Shape::draw  
    ((VIRTUAL_FUN)print_addr)(&s);                             Shape::print  
}
```


Виртуальные функции

Данный способ будет работать даже если виртуальная функция объявлена со спецификатором доступа `private` или `protected`.

Спецификаторы доступа предназначены для контроля доступа при «честном» вызове методов и не защищают от вызова функции по адресу.

Виртуальные функции

При наличии в классе виртуальных функций каждый экземпляр класса имеет служебное поле, хранящее указатель на таблицу виртуальных функций.

Поэтому при наличии в классе виртуальных функций, его размер увеличивается на размер указателя. При этом, так же как и для обычных классов, учитывается выравнивание полей.

Виртуальные функции

```
std::cout << sizeof(Shape) << " "  
<< sizeof(Rectangle) << " "  
<< sizeof(Square) << std::endl;
```

Результат:

16 32 32

Виртуальные функции

Зачастую при построении иерархии наследования базовый класс является настолько обобщенным, что объекты этого класса не существуют.

Так, например, экземпляр класса `Shape` не имеет особого самостоятельного смысла. Практический интерес представляют производные от него классы `Rectangle`, `Square` и т.д., описывающие конкретные геометрические фигуры.

Виртуальные функции

Класс `Shape` хранит общую для всех фигур информацию (в данном случае площадь). Однако, реализацию некоторых методов, например, отрисовки, можно реализовать только для конкретных фигур. (В предыдущих примерах для иллюстративных целей была реализована «заглушка», выводящая на экран имя вызванного метода и класса, которому он принадлежит)

Создавать экземпляры такого класса на практике не имеет особого смысла.

Виртуальные функции

Существует механизм, использование которого позволяет сообщить компилятору, что базовый класс является абстракцией, вследствие чего создавать объекты данного класса нельзя, а некоторые методы нужно реализовать в производном классе.

Это достигается за счет использования чисто виртуальных функций.

Виртуальные функции

Объявление чисто виртуальной функции:

```
class Shape {  
    ...  
public:  
    virtual void draw() = 0;  
};
```

Виртуальные функции

Абстрактный класс – это класс, содержащий хотя бы одну чисто виртуальную функцию.

Свойства абстрактных классов:

- Экземпляры абстрактных классов создавать нельзя. При попытке создания будет выдана ошибка компиляции.
- Абстрактный класс может использоваться только в качестве базового для других классов.
- В производном классе должна быть реализована функция, объявленная в абстрактном как чисто виртуальная. Если в производном классе отсутствует ее реализация он также является абстрактным.
- Можно пользоваться указателем на абстрактный класс.

Виртуальные функции

```
class Rectangle : public Shape {  
    ...  
public:  
    void draw() { std::cout << "Rectangle::draw" << std::endl; }  
};
```

```
class Square : public Rectangle {  
    ...  
public:  
    void draw() { std::cout << "Square::draw" << std::endl; }  
};
```

Виртуальные функции

```
void Draw(Shape* Sh) {  
    Sh->draw();  
}
```

```
void Draw2(Shape& Sh) {  
    Sh.draw();  
}
```

...

```
Rectangle r(1, 2);
```

```
Square s(3);
```

```
Draw(&r);
```

```
Draw2(s);
```

```
Rectangle::draw
```

```
Square::draw
```

Виртуальные функции

Поскольку все объекты, входящие в иерархию наследования, являются и объектами базового класса, то возможны операции с объектами производных классов посредством указателя на базовый класс.

Например, передача в функцию по адресу или создание объектов производных классов через указатель на базовый класс.

Виртуальные функции

Такой подход удобен при работе с объектами, входящими в иерархию наследования, когда точный тип объекта известен только на этапе выполнения программы.

В примере с геометрическими фигурами, отрисовка сцены (совокупности фигур) при наличии виртуальных методов отрисовки конкретной фигуры выполняется посредством функции, принимающей указатель на базовый класс [Shape](#).

Виртуальные функции

```
struct SinglyLinkedList {  
    Shape* Data;  
    SinglyLinkedList* Next;  
};
```

```
void Add(SinglyLinkedList** PListHead, Shape* Val) { ... }
```

```
void DrawScene(SinglyLinkedList* Lst) {  
    SinglyLinkedList* curr = Lst;  
    while (curr != NULL) {  
        curr->Data->draw();  
        curr = curr->Next;  
    }  
}
```

Виртуальные функции

```
Rectangle r(1, 2);
```

```
Square s(3);
```

```
SinglyLinkedList* Head = NULL;
```

```
Add(&Head, &r);
```

```
Add(&Head, &s);
```

```
DrawScene(Head);
```

Виртуальные функции

При этом часто возникает необходимость и в обобщенном, т. е. через указатель на базовый класс, создании объектов производных классов в динамической памяти.

```
Shape* ptr = new Rectangle(1, 2);
```

```
Shape* ptr2 = new Square(3);
```

Виртуальные функции

При этом, при освобождении памяти оператором `delete` будет вызван деструктор класса `Shape`. Деструктор для объекта производного класса вызван не будет.

Для предотвращения подобной ситуации и корректного освобождения ресурсов деструкторы в объектах, входящих в иерархию наследования рекомендуется делать виртуальными.

Виртуальные функции

```
class Shape {  
    ...  
public:  
    virtual ~Shape() { ... }  
};  
  
class Rectangle : public Shape {  
    ...  
public:  
    virtual ~Rectangle() { ... }  
};  
  
class Square : public Rectangle {  
    ...  
public:  
    virtual ~Square() { ... }  
};
```

Виртуальные функции

Если оператор перегружен глобальной функцией, то он не будет полиморфным.

```
class Shape { ... };  
class Rectangle : public Shape { ... };  
  
void operator!(Shape& Sh) {  
    std::cout << "Shape::operator!" << std::endl;  
}  
  
void operator!(Rectangle& R) {  
    std::cout << "Rectangle::operator!" << std::endl;  
}
```

Виртуальные функции

```
Rectangle r;
```

```
!r;                                Rectangle::operator!
```

```
Shape* ptr = new Rectangle(1, 2);
```

```
!(*ptr);                           Shape::operator!
```

Виртуальные функции

При перегрузке виртуальным методом класса оператор ! становится полиморфным:

```
class Shape {  
    virtual void operator!() {  
        std::cout << "Shape::operator!" << std::endl;  
    }  
};  
  
class Rectangle : public Shape {  
    void operator!() {  
        std::cout << "Rectangle::operator!" << std::endl;  
    }  
};
```

Виртуальные функции

```
Rectangle r;
```

```
!r;                                Rectangle::operator!
```

```
Shape* ptr = new Rectangle(1, 2);
```

```
!(*ptr);                           Rectangle::operator!
```

Виртуальные функции. Интерфейсы

С помощью абстрактных классов, содержащих только чисто виртуальные функции можно реализовать такую сущность, как интерфейс.

Интерфейс – это сущность, используемая для описания совокупности возможностей (т.е. методов), предоставляемых классом.

Т.е. интерфейс определяет прототипы методов (имя, тип возвращаемого значения, тип и количество параметров), которые должны быть реализованы в классе-наследнике.

Говорят, что класс реализует или поддерживает некоторый интерфейс, если данный класс является публичным классом-наследником данного интерфейса и предоставляет реализацию всех чисто виртуальных функций, определенных в интерфейсе.

Виртуальные функции. Интерфейсы

Пусть некоторые объекты должны поддерживать отрисовку. Для этого удобно определить интерфейс `IDrawable`, который должны будут поддерживать объекты, которым нужна отрисовка.

```
class IDrawable
{
public:
    void draw() = 0;
};
```

Виртуальные функции. Интерфейсы

```
class Rectangle : public IDrawable
{
public:
    void draw() { ... }
};
```

```
class FunctionGraph : public IDrawable
{
public:
    void draw() { ... }
};
```


Виртуальные функции. Интерфейсы

Классы `Rectangle` и `FunctionGraph` поддерживают интерфейс `IDrawable`. Как следствие, объекты данных типов могут быть нарисованы с использованием некоторого обобщенного алгоритма, который опирается лишь на поддержку интерфейса (т.е. наличие метода с заданным прототипом), а не на конкретный тип объекта.

```
void Draw(IDrawable& Item)
{
    Item.draw();
}
```

Финальный класс

Если необходимо запретить дальнейшее изменение класса или его полей/методов, то нужно объявить их с модификатором `final` (начиная с C++11).

Финальный класс

```
class B final : public A {  
    int a;  
public:  
    B() { ... }  
    B(int Val) { ... }  
    void print() { ... }  
    A GetA() { ... }  
};
```

Финальный класс

При попытке создания производного класса от финального будет сгенерирована ошибка времени компиляции.

```
class C : public B
{
    ...
};
```

ERROR

Финальные функции

Если виртуальная функция в некотором производном классе и его потомках не должна переопределяться, то она может быть объявлена с ключевым словом **final** (начиная с C++11).

Переопределение такой функции в производном классе вызовет ошибку компиляции.

Финальные функции

```
class A {  
    ...  
public:  
    virtual void print() { ... }  
};
```

```
class B : public A {  
public:  
    void print() final { ... }  
};
```

```
class C : public B {  
    ...  
public:  
    void print() { ... }  
};
```

ERROR

Оператор `dynamic_cast`

Если класс входит в иерархию наследования и обладает свойством полиморфизма, то на этапе выполнения программы может быть осуществлено приведение типа к корректному указателю или ссылке при помощи оператора `dynamic_cast`.

Данный оператор применяется к указателям и ссылкам.

При попытке приведения указателя, который содержит адрес объекта базового класса, к указателю типа производного класса, будет возвращен нулевой указатель.

Оператор `dynamic_cast`

```
Shape* ptr = new Rectangle(1, 2);
```

```
Shape* ptr2 = new Square(3);
```

```
Rectangle* newPtr1 = dynamic_cast<Rectangle*>(ptr);
```

```
newPtr1 = 0x11223344
```

```
Rectangle* newPtr2 = dynamic_cast<Rectangle*>(ptr2);
```

```
newPtr2 = 0x55667788
```

```
Square* newPtr3 = dynamic_cast<Square*>(ptr);
```

```
newPtr3 = NULL
```


Оператор `dynamic_cast`

`Shape*` arr[OBJ_COUNT] = { `NULL` }; массив указателей на `Shape`

```
for (int i = 0; i < OBJ_COUNT; i++) {  
    if (rand() % 2) arr[i] = new Rectangle();  
    else arr[i] = new Square();  
}  
  
int rCount = 0, sCount = 0;  
for (int i = 0; i < OBJ_COUNT; i++) {  
    if (dynamic_cast<Square*>(arr[i]) != NULL)  
        sCount++;  
    else  
        rCount++;  
}
```

Множественное наследование

Множественное наследование – наследование, при котором производный класс имеет двух и более родителей.

Множественное наследование

Примеры множественного наследования:



Множественное наследование

Базовые классы при множественном наследовании перечисляются через запятую. Спецификатор доступа может стоять перед именем каждого базового класса. Если спецификатор доступа не указан, то считается, что указан `private`.

```
class Der : Base1, public Base2, protected Base3
{
    ...
}
```

Множественное наследование

Порядок вызова конструкторов:

При создании объекта производного класса сначала вызываются конструкторы базовых классов в том порядке, в котором они перечислены при объявлении класса.

Порядок вызова деструкторов:

При уничтожении объекта производного класса деструкторы вызываются в порядке, обратном порядку вызова конструкторов.

Множественное наследование

```
class A {  
protected:  
    int val;  
public:  
    A(int Val) { std::cout << "A(int)" << std::endl; this->val = Val; }  
    ~A() { std::cout << "~A()" << std::endl; }  
};
```

```
class B {  
protected:  
    int val;  
public:  
    B(int Val) { std::cout << "B(int)" << std::endl; this->val = Val; }  
    ~B() { std::cout << "~B()" << std::endl; }  
};
```

Множественное наследование

```
class C : public A, public B {  
public:  
    C(int Val) : B(Val), A (Val) {  
        std::cout << "B(int)" << std::endl;  
    }  
    ~C() { std::cout << "~C()" << std::endl; }  
};  
...  
{  
    C c(5);  
}  
  
A(int) B(int) C(int)  
~C() ~B() ~A()
```

Множественное наследование

Поскольку в обоих базовых классах присутствует поле с одним и тем же именем, то при попытке обращения к нему из методов производного класса C будет неоднозначным, что вызовет ошибку компиляции.

Множественное наследование

Для разрешения неоднозначности необходимо явно указать к полю какого класса происходит обращение:

```
int C::GetVal() { return this->A::val; }
```

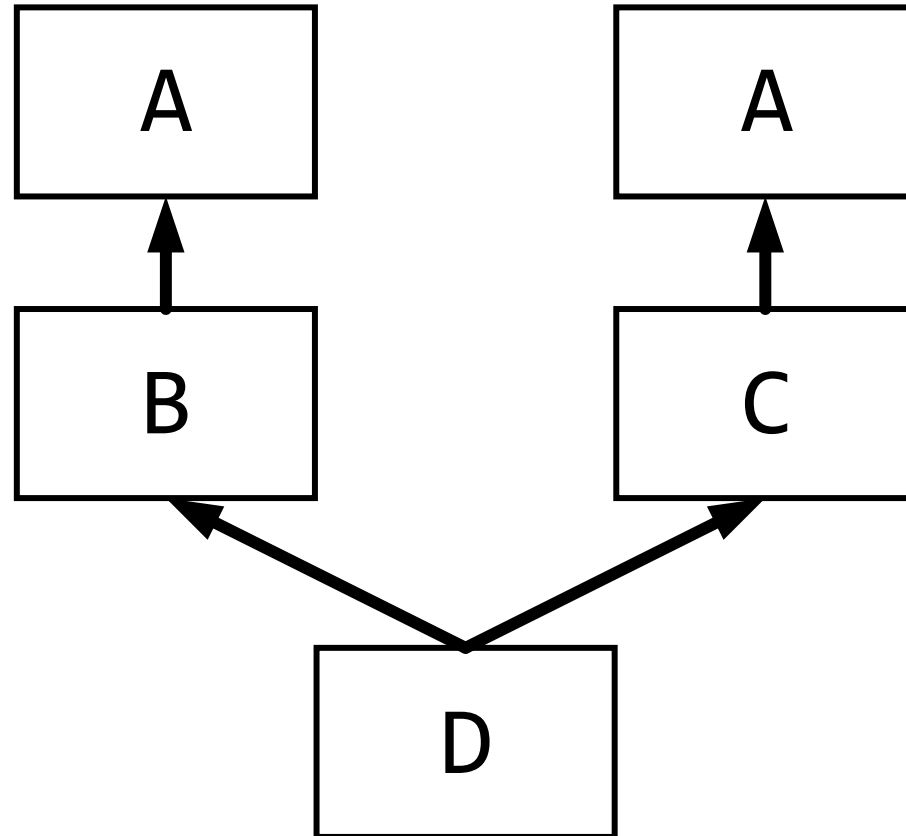
Множественное наследование.

Повторяющиеся базовые классы

При множественном наследовании может возникнуть ситуация, когда какой-либо класс несколько раз окажется базовым для некоторого производного класса.

Такая ситуация приводит к наличию в производном классе нескольких объектов одного и того же базового класса. Как следствие, возникают дополнительные затраты памяти и неоднозначность при обращении к полям и методам повторяющегося базового класса.

Множественное наследование. Повторяющиеся базовые классы



Множественное наследование. Повторяющиеся базовые классы

```
class A {  
protected:  
    int val;  
public:  
...};
```

```
class B : public A {  
    int bVal;  
public:  
    B() { this->val = 1; this->bVal = 0; }  
    int getAVal() { return this->val; }  
    int getVal() { return this->bVal; }  
};
```

Множественное наследование. Повторяющиеся базовые классы

```
class C : public A {  
    int cVal;  
public:  
    C() { this->val = 2; this->cVal = 0;}  
    int getAVal() { return this->val; }  
    int getVal() { return this->cVal; }  
};
```

```
class D : public B, public C {  
    int val;  
public:  
    D() {  
        this->val = 0;           //Используется поле val класса D  
        this->B::A::val = 3;     //Неоднозначность. Поле A::val есть и в классе B и в C  
    }  
};
```

Множественное наследование. Повторяющиеся базовые классы

```
D d;
```

```
int val1 = d.B::getAVal();           val1 = 3
```

```
int val2 = d.C::getAVal();           val2 = 2
```

```
A a = d;           //Ошибка. Неоднозначность, т.к.  
                   А есть и в В, и в С.
```

Множественное наследование.

Виртуальный базовый класс

Избежать неоднозначности при обращении к полям и методам базового класса **A** базового класса можно при помощи виртуального наследования.

Для этого, при объявлении базового класса для **B** и **C** необходимо использовать ключевое слово **virtual**.

Каждый виртуальный базовый класс в производном классе представлен одним и тем же (совместно используемым) объектом.

Множественное наследование.

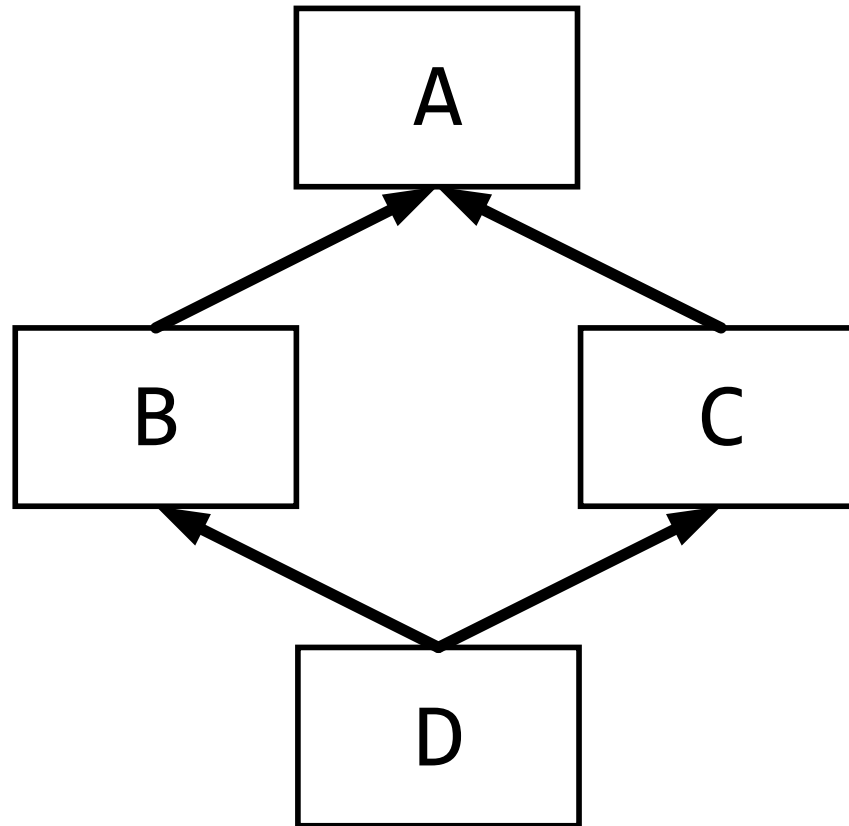
Виртуальный базовый класс

```
class B : virtual public A { ... };
class C : virtual public A { ... };

class D : public B, public C {
    int val;
public:
    D() {
        this->val = 0;           //Используется поле val класса D
        this->A::val = 3;        //Нет неоднозначности
    }
};
```


Множественное наследование. Виртуальный базовый класс

При этом иерархия наследования имеет вид:



Множественное наследование. Виртуальный базовый класс

```
D d;  
int val1 = d.B::getAVal();           val1 = 3  
int val2 = d.C::getAVal();           val2 = 3  
  
A a = d;           //Нет неоднозначности
```

Множественное наследование.

Виртуальный базовый класс

Однако такой вариант иерархии наследования не всегда применим. Например, если классы **В** и **С** должны меняться независимо друг от друга.

При применении «ромбического» наследования при изменении базовой части класса **В** будет меняться и базовая часть класса **С**. При неvirtуальном наследовании эти части меняются независимо друг от друга.