

Реализация наследования

- Синтаксис наследования
- Поля класса при наследовании
- Конструкторы и деструкторы при наследовании
- Расширение структуры и поведения родительского класса. Общедоступное наследование, добавление полей и методов
- Ограничение структуры и поведения родительского класса. Защищенное и закрытое наследование

Реализация наследования

- Переопределение поведения родительского класса. Переопределение и уточнение методов при наследовании
- Отложенные методы и абстрактно-специфицированные классы
- Множественное наследование и связанные с ним проблемы. Виртуальный базовый класс. Интерфейсы
- Терминальные классы и методы

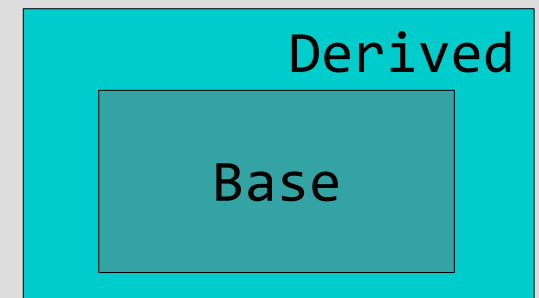
Синтаксис наследования

```
class Base
{
    // Описание базового класса
};

class Derived: спецификатор_доступа Base
               [, спецификатор_доступа Base2, ...]
{
    // Описание производного класса
};
```

Поля класса при наследования

- При создании экземпляра производного класса сначала создаются и инициализируются **все поля базового** класса, а затем поля, добавленные в **производном** классе
- **Логически** можно считать, что производный класс **агрегирует** в себе базовый класс или в производном классе «**упакован**» базовый класс
- Для инициализации **полей базового** класса используется **конструктор базового** класса



Конструкторы при наследовании

- Как и другие методы конструкторы наследуются, но их использование имеет ряд особенностей
- Если в конструкторе производного класса явно не вызывается конструктор базового класса, то компилятор сам вызывает конструктор по умолчанию базового класса

Конструкторы при наследовании

- Если необходимо вызвать конструктор базового класса **с параметрами**, то конструктор указывается в **строке инициализации**
- Тело конструктора **базового** класса всегда выполняется **раньше** тела конструктора **производного** класса

Деструкторы при наследовании

- Как и другие методы деструкторы **наследуются**, но их использование имеет ряд особенностей
- Деструктор производного класса **не требует** явно вызывать деструктор базового класса. В деструкторе производного класса компилятор **автоматически** генерирует вызовы базовых деструкторов
- Тело деструктора **производного** класса всегда выполняется **раньше** тела деструктора **базового** класса

Пример вызова конструкторов и деструкторов при наследовании

```
// Базовый класс
class Base
{
public:
    // Конструктор по умолчанию
    Base()
    { _property = -1;    puts("Default constructor Base"); }

    // Конструктор с параметрами
    Base(int value)
    { _property = value; puts("Constructor Base"); }

    // Деструктор
    ~Base()                { puts("Destructor Base"); }

private:
    int _property;
};
```


Пример вызова конструкторов и деструкторов при наследовании

```
// Производный класс
class Derived: protected Base
{
public:
    // Конструктор по умолчанию
    Derived()          { puts("Default constructor Derived"); }

    // Конструктор с параметрами
    Derived(int value)
        :Base(value)   { puts("Constructor Derived"); }

    // Деструктор
    ~Derived()          { puts("Destructor Derived"); }
};
```

Пример вызова конструкторов и деструкторов при наследовании

```
int _tmain(int argc, _TCHAR* argv[])
{
    Base *b;
    Derived *d;

    b = new Base;          delete b;    puts("");
    b = new Base(2);        delete b;    puts("");
    d = new Derived;        delete d;    puts("");
    d = new Derived(3);     delete d;    puts("");

    getch();
    return 0;
}
```

Пример вызова конструкторов и деструкторов при наследовании

```
Default constructor Base  
Destructor Base
```

```
Constructor Base  
Destructor Base
```

```
Default constructor Base  
Default constructor Derived  
Destructor Derived  
Destructor Base
```

```
Constructor Base  
Constructor Derived  
Destructor Derived  
Destructor Base
```

Расширение структуры и поведения родительского класса

- Расширение структуры и поведения родительского класса осуществляется за счет **общедоступного** наследования и **добавления** новых свойств и методов
- При общедоступном наследовании в дочернем классе полностью **сохраняется** интерфейс родительского класса

Уровень доступа к элементу в базовом классе

private
protected
public

Спецификатор доступа при наследовании

public

Уровень доступа к элементу в производном классе

не доступен
protected
public

Добавление методов при наследовании

- Под добавлением методов понимается такая ситуация, когда в производном классе объявляются **новые** методы, которые **отличаются** от методов базового класса названием и/или набором параметров
- При этом интерфейс родительского класса **расширяется**
- Добавление методов используется при «**расширении**» базового класса, но может встречаться при его «**обобщении**» и «**варьировании**»

Пример добавления методов

```
// Базовый класс
class Base
{
public:
    void draw()                { printf("####"); }
};

// Производный класс
class Derived: public Base
{
public:
    void gotoxy(int x, int y) { ... }

private:
    int _x, _y;
};
```

Пример добавления методов

```
int _tmain(int argc, _TCHAR* argv[])
{
    Base b;
    Derived d;

    b.draw();                // распечатается ####

    d.gotoxy(10, 15);
    d.draw();                // в позиции (10, 15)
                           // распечатается ####

    getch();
    return 0;
}
```

Ограничение структуры и поведения родительского класса

- Под ограничением понимается частичное или полное **сокрытие** свойств и методов родительского класса в дочернем классе
- Соккрытие свойств и методов класса выполняется для реализации таких форм наследования как «**ограничение**» и «**конструирование**»
- Чаще всего для ограничения структуры и поведения родительского класса используют **закрытое** и **защищенное** виды наследования, которые **полностью скрывают** интерфейс родительского класса

Защищенное и закрытое наследование

Уровень доступа к
элементу в базовом
классе

private

protected

public

private

protected

public

Спецификатор
доступа при
наследовании

protected

private

Уровень доступа к
элементу в
производном классе

не доступен

protected

protected

не доступен

private

private

Частичное сокрытие свойств и методов базового класса

- Иногда требуется **частичное** сокрытие свойств и методов базового класса
- В языке Си++ для отдельных свойств и методов можно задать **свои** спецификаторы доступа, **отличные** от вида наследования
- При этом уровень доступа **не может быть повышен** по сравнению с уровнем доступа, указанным в базовом классе. Самый **привилегированный** уровень доступа — это **private**

Пример частичного сокрытия свойств и методов базового класса

```
// Базовый класс
class Base
{
public:
    Base()                { _property = -1; }

    void setProperty(int value)
        { _property = checkValue(value) ? value : -1; }

    int property()         { return _property; }

protected:
    bool checkValue(int value) { return value > 0; }

private:
    int _property;
};
```

Пример частичного сокрытия свойств и методов базового класса

```
// Производный класс (защищенное наследование)
class Derived: protected Base
{
public:
    Derived(int value)
    {
        // _property = value;           // так нельзя
        setProperty(value);             // так можно
    }
    // Делаем int Base::property() общедоступным методом
    Base::property;

private:
    // Делаем метод void Base::setProperty(int value)
    // закрытым
    Base::setProperty;
};
```

Пример частичного сокрытия свойств и методов базового класса

```
int _tmain(int argc, _TCHAR* argv[])
{
    Derived d(2);                // d._property = 2

    printf("%d", d.property());

    // d._property = 3;          // так нельзя
    // d.setProperty(3);         // так тоже нельзя

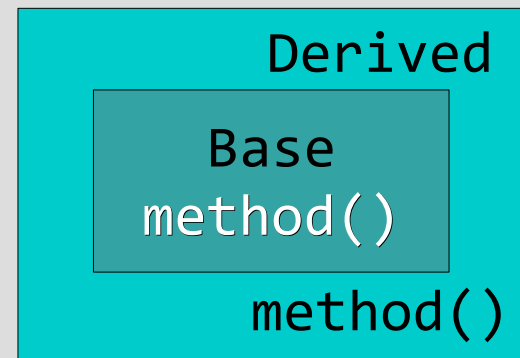
    getch();
    return 0;
}
```

Переопределение поведения родительского класса

- Переопределить поведение родительского класса можно за счет использования общедоступного наследования и одновременного **переопределения/замещения** его методов
- Переопределение/замещение методов используется при «**специализации**», «**спецификации**», «**обобщении**», «**конструировании**» и «**варьировании**» базового класса
- **Различие** между переопределением и замещением будет рассмотрено при изучении **полиморфизма**

Переопределение/замещение методов при наследовании

- Ситуация, когда в дочернем классе **объявляется метод, совпадающий** по названию и набору параметров с методом родительского класса, называется переопределением/замещением метода
- Для пользователя дочернего класса переопределяемый/замещаемый метод базового класса уже **не виден**, но он существует в производном классе
- Метод базового класса может быть вызван путем **явного** указания принадлежности к базовому классу



Пример замещения методов

```
// Базовый класс
class Base
{
public:
    void draw()                { printf("####"); }
};

// Производный класс
class Derived: public Base
{
public:
    void draw()                { printf("****"); }
    void gotoxy(int x, int y) { ... }

private:
    int _x, _y;
};
```


Пример замещения методов

```
int _tmain(int argc, _TCHAR* argv[])
{
    Base b;
    Derived d;

    b.draw();                // распечатается ####

    d.gotoxy(10, 15);
    d.draw();                // в позиции (10, 15)
                           // распечатается ****

    d.Base::draw();         // распечатается ####

    getch();
    return 0;
}
```

Уточнение методов при наследовании

- Уточнение метода — это **специализированная** форма переопределения/замещения
- В этом случае замещающий метод **вызывает** замещаемый метод родительского класса
- Таким образом, родительское поведение **сохраняется и присоединяется**
- Уточнение методов используется при «**специализации**», «**обобщении**», «**конструировании**» и «**варьировании**» базового класса

Пример уточнения методов

```
class Base                                // Базовый класс
{
public:
    void draw()                          { printf("####"); }
};

class Derived: public Base                // Производный класс
{
public:
    Derived(int x, int y) { _x = x; _y = y; }
    void draw()          { gotoxy(_x, _y); Base::draw(); }

protected:
    void gotoxy(int x, int y) { ... }

private:
    int _x, _y;
};
```

Пример уточнения методов

```
int _tmain(int argc, _TCHAR* argv[])
{
    Base b;
    Derived d(10, 15);

    b.draw();           // распечатается ####

    d.draw();           // в позиции (10, 15)
                       // распечатается ####

    d.Base::draw();     // распечатается ####

    getch();
    return 0;
}
```

Отложенные методы

- **Отложенный** метод (абстрактный метод) – это частный случай переопределения, когда метод базового класса **не имеет реализации**, а любая полезная деятельность задается в методе дочернего класса
- Использование отложенных методов **гарантирует единый** интерфейс у всех потомков базового класса

Отложенные методы и абстрактные классы в языке Си++

- В языке Си++ отложенный метод должен быть описан в **явном** виде с ключевым словом **virtual**
- Тело отложенного метода не определяется, вместо этого функции «**приписывается**» значение **0**

virtual <заголовок метода> = 0;

- Отложенные методы в языке Си++ называются **чисто виртуальными**
- В языке Си++ класс, который содержит чисто виртуальные методы, называется **абстрактным**

Свойства абстрактных классов

- Компилятор **не разрешает** пользователю создавать **экземпляры** абстрактного класса
- Только после того, как в производных классах будут **переопределены** отложенные методы, можно будет создавать экземпляры подклассов

Пример отложенного метода

```
class Base                                // Базовый класс - абстрактный
{
public:
    virtual void draw() = 0;
};

class Derived: public Base                // Производный класс
{
public:
    Derived(int x, int y)                { _x = x; _y = y; }
    void draw() { gotoxy(_x, _y); printf("####"); }

protected:
    void gotoxy(int x, int y) { ... }

private:
    int _x, _y;
};
```

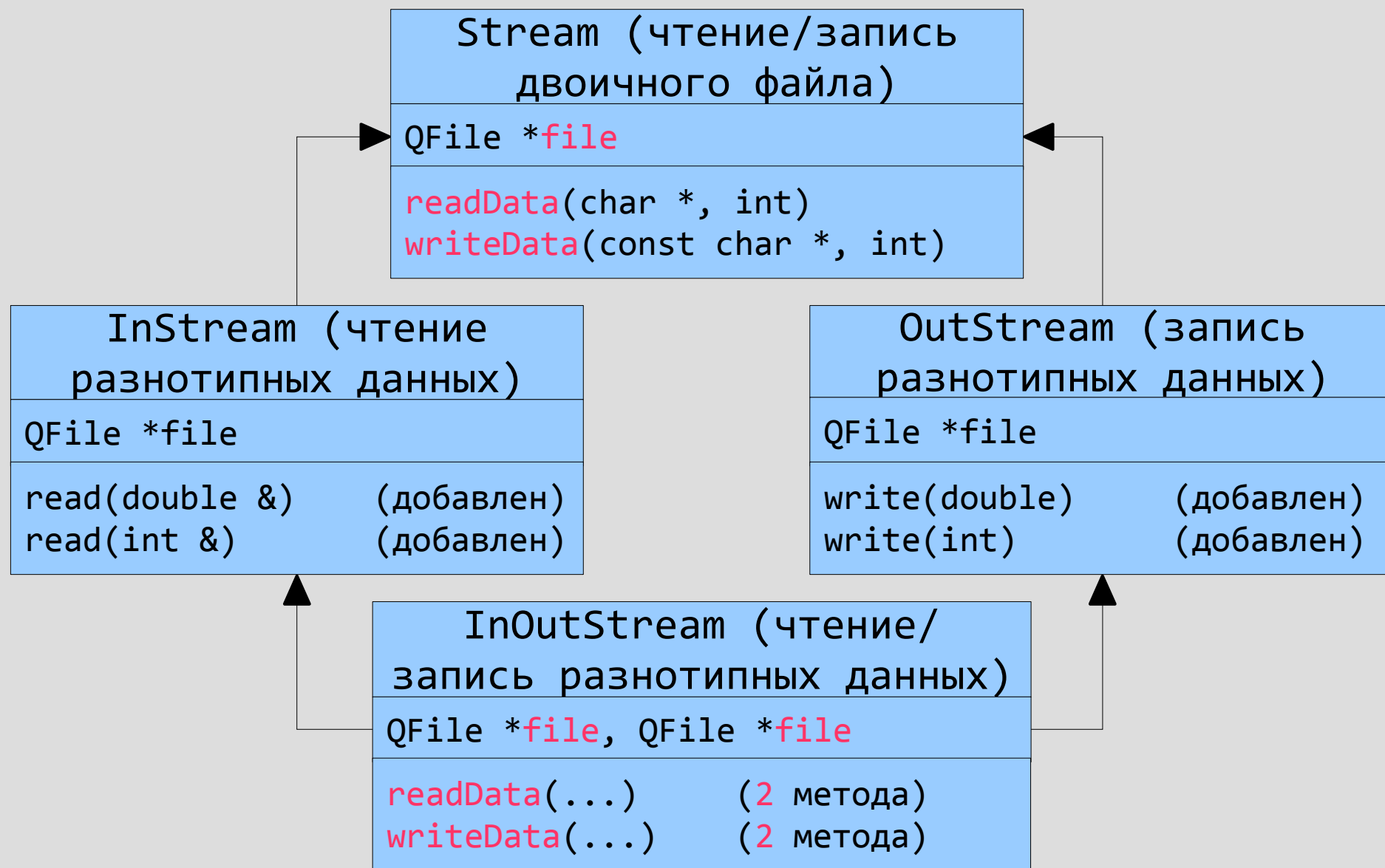

Множественное наследование

- Создание класса на основе **двух и более** базовых классов называется множественным наследованием
- В этом случае производный класс обладает свойствами и методами сразу **нескольких** базовых классов
- Через множественное наследование реализуется форма наследования «**комбинирование**»
- Правильная форма «комбинирования» подразумевает, что производный класс может выступать в **роли любого** из своих родителей

Проблемы, связанные с множественным наследованием

- Проблема **двусмысленности** имен возникает, когда в базовых классах имеются **одноименные** методы
- Проблема **наследования через общих предков** возникает, когда диаграмма наследования имеет **ромбовидную** форму. В этом случае к проблеме двусмысленности имен добавляется проблема **дублирования** полей базового класса

Проблема наследования через общих предков



Решение проблемы двусмысленности имен

- Решение проблемы заключается в **одновременном** использовании **замещения** и **переименования** методов

Пример решения проблемы двусмысленности имен

```
class Base1      // Базовый класс
{
public:
    Base1()      { f_value = 1; }
    int value()  { return f_value; }
private:
    int f_value;
};

class Base2      // Другой базовый класс
{
public:
    Base2()      { f_value = 2; }
    int value()  { return f_value; }
private:
    int f_value;
};
```

Пример решения проблемы двусмысленности имен

```
// Производный класс, который содержит ДВА поля f_value
// и ДВА метода value(). Каждый метод value() будет
// обращаться к своему полю
class Derived: private Base1, private Base2
{
    public:
        int value1()      { return Base1::value(); }
        int value2()      { return Base2::value(); }
};
```

Пример решения проблемы двусмысленности имен

```
int _tmain(int argc, _TCHAR* argv[])
{
    Derived d;

    // Ошибка, т.к. непонятно какой из методов вызывать
    // d.value();

    // Ошибка, т.к. закрытое наследование
    // d.Base1::value();

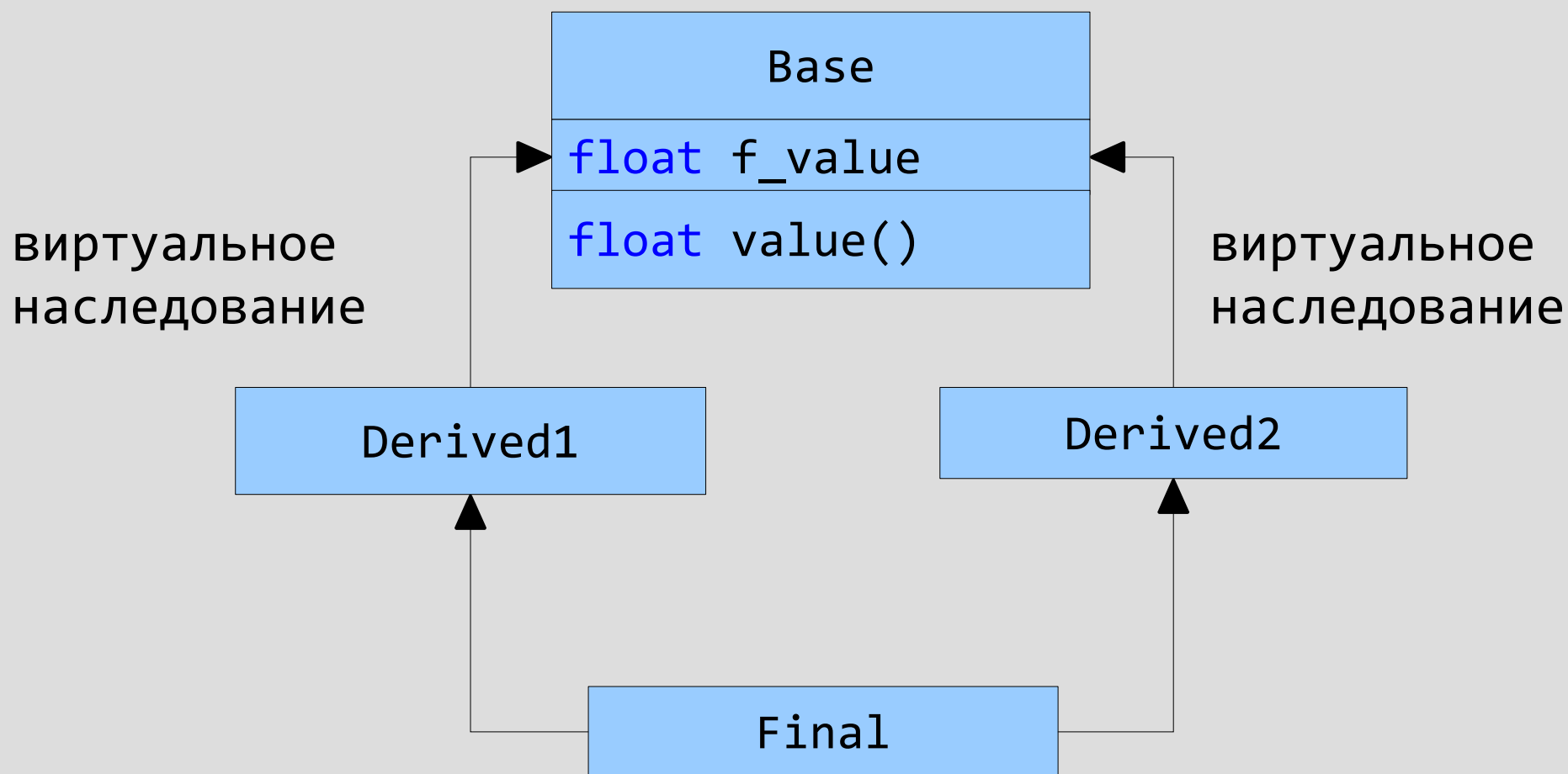
    // Верно. Результат: 1, 2
    printf("%d, %d", d.value1(), d.value2());

    return 0;
}
```

Решение проблемы наследования через общих предков

- Если мы хотим иметь только одну копию полей **прародительского** класса, то должны наследовать его как виртуальный базовый класс
- Если **базовый класс** определен как **виртуальный**, то в производных классах создается единственная копия его полей. Кроме того, только один раз будет вызван конструктор по умолчанию прародительского класса

Пример использования виртуального базового класса



Пример использования виртуального базового класса

```
// Базовый класс - прародитель
class Base
{
public:
    Base()                { f_value = 1.0; puts("1.0"); }
    float value()         { return f_value; }

protected:
    float f_value;
};
```

Пример использования виртуального базового класса

```
// Промежуточный производный класс
class Derived1 : virtual public Base
{
public:
    Derived1()          { f_value = 1.1; puts("1.1"); }
};

// Промежуточный производный класс
class Derived2 : virtual public Base
{
public:
    Derived2()          { f_value = 1.2; puts("1.2"); }
};
```

Пример использования виртуального базового класса

```
// Производный класс, который содержит ОДНО поле  
// f_value и ОДИН метод value(), который будет  
// обращаться к этому полю
```

```
class Final: private Derived1, private Derived2  
{  
};
```

```
int main(int argc, char *argv[])  
{  
    Final f;  
  
    printf("value = %g", f.value());  
  
    return 0;  
}
```

Пример использования виртуального базового класса

```
1.0  
1.1  
1.2  
value = 1.2
```

- Полю `value` присвоено значение `1.2`, т.к. последним вызывался конструктор класса `Derived2`

Уровни доступа при наследовании через общих предков

- Спецификаторы доступа при наследовании могут **отличаться** в разных родительских классах
- Следовательно, одни и те же элементы прародительского класса будут иметь **различные** спецификаторы доступа в производном классе
- В таком случае наиболее жесткий уровень защиты игнорируется и используется **менее ограничительная** категория

Рекомендации по использованию множественного наследования

- Из-за проблем, возникающих при множественном наследовании классов, многие языки программирования его **не поддерживают**
- В современных ООП-языках вместо множественного наследования классу разрешено реализовать **несколько интерфейсов**
- Использование интерфейсов **решает** проблему наследования через общих предков

Интерфейсы

- Интерфейс — это множество **требований** (requirements), предъявляемых к соответствующему классу
- Требования выражаются в **объявлении методов**
- Объявление интерфейса на языке **Java**

```
public interface <имя интерфейса>  
{  
    // заголовки методов  
}
```


Свойства интерфейсов (применительно к языку Java)

- В интерфейсе не может быть ни полей, ни статических методов, но **могут** объявляться **статические константы**
- **Нельзя** создать **экземпляр** интерфейса, но **можно** объявить интерфейсную **переменную**
- Интерфейсные переменные должны ссылаться на объект класса, реализующего данный интерфейс
- Аналогично классам интерфейсы могут образовывать **иерархию** наследования

Реализация интерфейсов (применительно к языку Java)

- Реализация интерфейса выполняется каким-либо классом путем **определения** методов, указанных в интерфейсе

```
class <имя класса> implements <имя интерфейса>
{
    // описание класса, в том числе, методов,
    // указанных в интерфейсе
}
```

Пример интерфейсов (применительно к языку Java)

```
interface canFight {           // Может убивать
    void fight();
}

interface canSwim {           // Может плавать
    void swim();
}

interface canFly {            // Может летать
    void fly();
}

class ActionCharacter {        // Действующее лицо
    public void fight() { ... }
}
```

Пример интерфейсов (применительно к языку Java)

```
class Hero extends ActionCharacter
    implements canFight, canSwim, canFly { /* Герой
– это действующее лицо, которое может плавать и летать */
    public void swim()      { ... }
    public void fly()       { ... }
}
```

```
public class Advanture { // Приключение
    public static void t(canFight x)      { x.fight(); }
    public static void u(canSwim x)       { x.swim(); }
    public static void v(canFly x)        { x.fly(); }
    public static void w(ActionCharacter x) { x.fight(); }
    public static void main(Strings args[])
    {
        Hero h = new Hero();
        t(h); u(h); v(h); w(h);
    }
}
```

Миксеры

- В рассмотренном примере используется множественное наследование в «**СМЕШАННОМ**» виде
- Базовый класс `ActionCharacter` описывает родительский объект, а интерфейсы `canFight`, `canSwim`, `canFly` описывают **ВСПОМОГАТЕЛЬНЫЕ** свойства
- В этом случае интерфейсы выполняют роль **МИКСЕРОВ** (mix-ins) — они «**примешивают**» к базовому классу дополнительные свойства

Интерфейсы и абстрактные классы

- Интерфейсы и абстрактные классы — это **схожие** понятия, но имеются и **различия** между ними
- Абстрактный класс **может иметь** состояние и поведение (хотя бы частично), а интерфейс только **заявляет** поведение
- Интерфейс, чаще всего, определяет некоторое **свойство**, а абстрактный класс определяет «полноценную» **сущность**
- Злоупотреблять интерфейсами не следует, т.к. они вводят много **мелких** понятий

Терминальные классы и методы

- Классы, которые **не** могут иметь **ПОТОМКОВ**, называются терминальными
- Данный механизм отсутствует в языке Си++, но имеется в языке **Java**

final class <имя класса> { ... }

- Так же неизменными можно сделать **отдельные** методы класса

final <имя метода> { ... }

Использование терминальных классов и методов

- Трудно предусмотреть все возможности повторно-го использования класса, особенно для классов общего назначения. Определяя класс или метод как `final`, вы **блокируете** возможность использования класса в проектах других программистов только потому, что сами не могли предвидеть такую возможность
- Обычно терминальные классы и методы используются для обеспечения **безопасности**