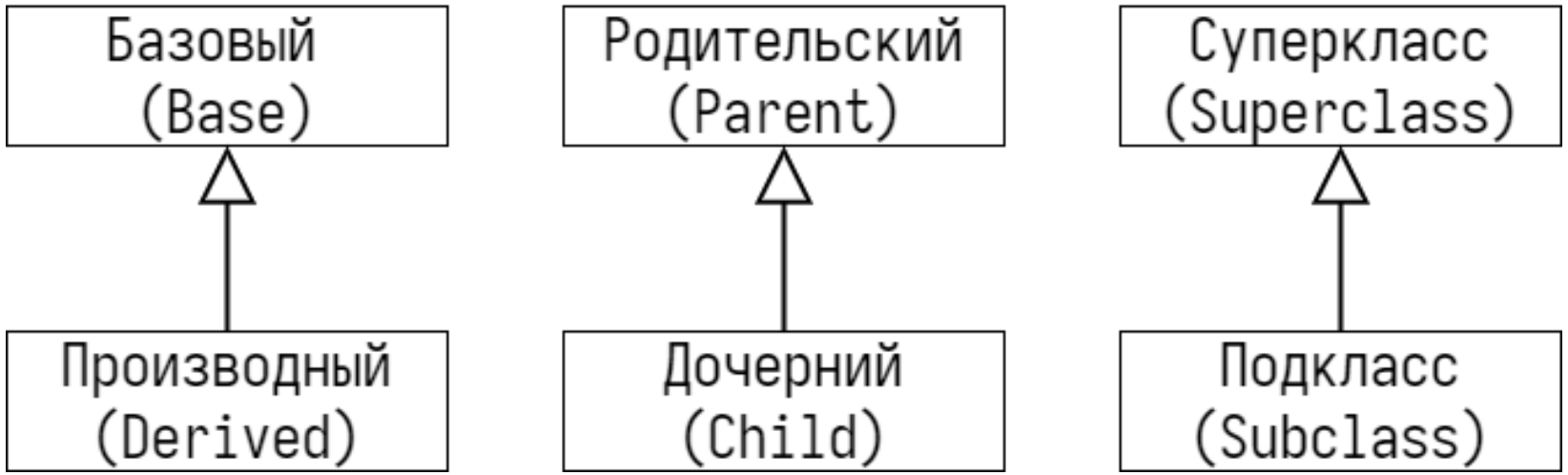
Лекция 3

Наследование

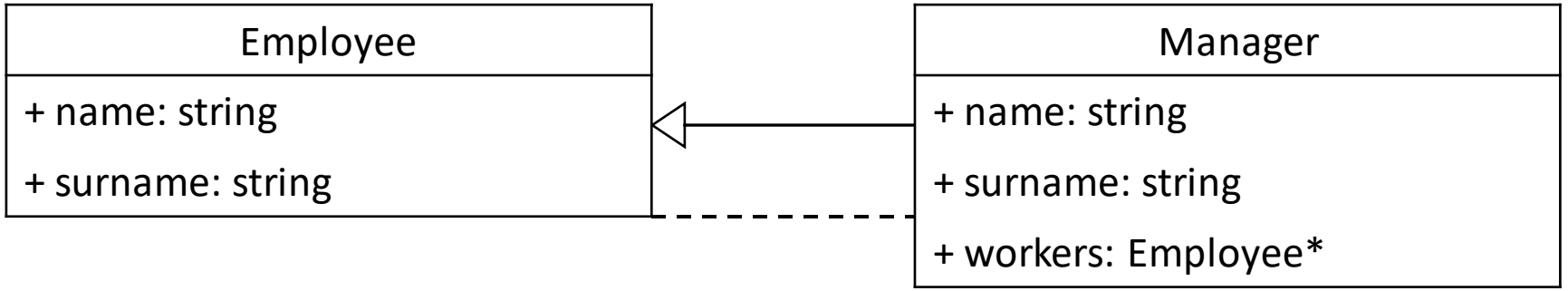
Наследование

- Один из основных механизмов ООП
- Позволяет создавать классы на основе существующих
- Изменяют или расширяют функционал тех классов, на основе которых создаются
- В C++ допускается множественное наследование



Расположение в памяти

- Дополнительная память только для новых полей
- Внутренний объект родительского класса располагается в начале дочернего объекта
- Ссылку или указатель на объект дочернего класса можно использовать везде, где допустимо использование ссылки или указателя на объект родительского класса



Порядок определения

- Использование класса в качестве базового эквивалентно созданию неименованного поля в дочернем объекте
- Для использования в качестве базового, класс должен быть объявлен перед этим
- Наследование может отличаться модификатором доступа

```
class Base; //только объявление без определения

class Deriv: public Base{
    //Определение
};
```

Приведение по ссылке или указателю

- Это возможно за счет одинакового расположения полей

```
class Base{

};

class Deriv: public Base{

};

int main(){
    Base* obj1 = new Deriv();
    Deriv obj2;
    Base& obj3 = obj2;
}
```

Типы наследования

- Базовый класс может быть объявлен с одним из следующих модификаторов доступа:
  - `private`
  - `protected`
  - `public`

- Приватные члены базового класса недоступны в дочернем ни при каком типе наследования

- Конструкторы и деструкторы не наследуются

Доступ к полям при наследовании

- Происходит понижение к самому ограничивающему модификатору доступа

Область видимости базового класса	**Тип наследования**		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Ограничение доступа при protected

```
class Base{
private:
    void f(){std::cout << "Private";}
protected:
    void g(){std::cout << "Protected";}
public:
    void h(){std::cout << "Public";}
};

class Deriv: protected Base{

};

int main(){
    Deriv obj;
    obj.f();
    obj.g();
    obj.h();
}
```

Ограничение доступа при public

```
class Base{
private:
    void f(){std::cout << "Private";}
protected:
    void g(){std::cout << "Protected";}
public:
    void h(){std::cout << "Public";}
};

class Deriv: public Base{

};

int main(){
    Deriv obj;
    obj.f();
    obj.g();
    obj.h();
}
```

Порядок конструирования объектов

- Объекты создаются «снизу-вверх» - от базовых к производным
- Порядок вызовов конструкторов:
  1. Конструкторы виртуальных базовых классов
  2. Конструкторы прямых базовых классов
  3. Конструкторы полей
  4. Конструктор класса

- Деструкторы вызываются в обратном порядке

Переопределение функций

- Переопределение функций – создание функции в дочернем классе с сигнатурой, совпадающей с функцией в родительском классе

- Для определения вызываемой функции компилятор сначала ищет ее в дочернем классе, если не находит, то ищет в базовых классах по цепочке наследования

- Переопределение функций позволяет изменять поведение базового класса. Является статическим полиморфизмом.

Пример переопределения

```
class Base{
public:
    void print(){
        std::cout <<  "Base\n";}
};

class Deriv: public Base{
public:
    void print(){
        std::cout << "Derived\n";}
};

int main(){
    Base b;
    b.print(); // "Base"
    Deriv d;
    d.print(); // "Deriv"
    Base& b_ref = d;
    b_ref.print(); // "Base"
}
```

Вызов конструкторов базового класса (1)

- В данном случае ошибка, так как конструктор не наследуется, и нельзя вызвать конструктор производного класса с параметром

```
class Base{
public:
    Base(int a){}
};

class Deriv: public Base{
public:
    Deriv(){}
};

int main(){
    Deriv obj(4);
}
```

Вызов конструкторов базового класса (2)

- Необходимо получать аргументы и явно вызывать базовый конструктор

```
class Base{
public:
    Base(int a){}
};

class Deriv: public Base{
public:
    Deriv(int a):Base(a){}
};

int main(){
    Deriv obj(4);
}
```

Важность деструкторов при наследовании

- Так как деструкторы не наследуются, возможна утечка памяти

```
class Base{
    int* data;
public:
    Base():data(new int[10]){}
    ~Base(){delete [] data;}
};

class Deriv: public Base{
public:
    Deriv():Base(){}
};

int main(){
    Deriv obj();
}
```

Проблема переопределения функций

```
class Base{
public:
    void print(char a){std::cout << a <<'\n';}
    void print(int a){std::cout << a << '\n';}
};

class Deriv: public Base{
public:
    void print(std::string a){std::cout << "Deriv - " << a << '\n';}
};

int main(){
    Deriv obj;
    obj.print("a");
    obj.print('a'); //Ошибка
    obj.print(47);  //Ошибка
}
```

Ключевое слово using

- Позволяет не переопределять каждую перегруженную функцию базового класса
- Позволяет задать использование базового конструктора в качестве дочернего
- Позволяет изменять спецификатор доступа функций

using с конструкторами

- Позволяет использовать все базовые конструкторы

```
class Base{
public:
    Base(int a){}
};
class Deriv: public Base{
public:
    using Base::Base;
};
int main(){
  Deriv obj(4);
}
```

using с методами

```
class Base{
public:
    void print(char a){std::cout << a <<'\n';}
    void print(int a){std::cout << a << '\n';}
};
class Deriv: public Base{
public:
    using Base::print;
    void print(std::string a){std::cout << "Deriv - " << a << '\n';}
};
int main(){
    Deriv obj;
    obj.print("a");
    obj.print('a');
    obj.print(47);
}
```

Изменение видимости доступа (1)

- Запрещаем использовать public методы из базового класса

```
class Base{
public:
    void print(char a){std::cout << a <<'\n';}
    void print(int a){std::cout << a << '\n';}
};
class Deriv: public Base{
private:
    using Base::print;
};
int main(){
    Deriv obj;
    obj.print('a'); //Ошибка
    obj.print(47);  //Ошибка
}
```

Изменение видимости доступа (2)

- Разрешаем использовать protected методы из базового класса

```
class Base{
protected:
    void print(char a){std::cout << a <<'\n';}
    void print(int a){std::cout << a << '\n';}
};
class Deriv: public Base{
public:
    using Base::print;
};
int main(){
    Deriv obj;
    obj.print('a');
    obj.print(47);
}
```

Виртуальные функции

Виртуальные функции

- Функция-член класса, которую предполагается переопределить в производных классах
- Вызывает функцию производного класса даже через ссылку или указатель на базовый класс
- Модификатор **virtual** располагается перед типом возвращаемого значения
- Должна быть определена в месте первого объявления
- Может быть переопределена в дочерних классах

Использование виртуальных функций

- Определение конкретного типа объекта не требуется

- При вызове будет использован полиморфизм и реализация будет выбрана в зависимости от типа объекта

```
void printList(const vector<Employee*>& employees) {
    for (Employee* current : employees) {
        current->print();
    }
}
```

Полиморфный класс

- Любой класс, содержащий по крайней мере одну виртуальную функцию, является полиморфным
- Каждый объект такого класса содержит таблицу виртуальных функций **(vtable)**
- При использовании ссылки / указателя разрешение методов происходит динамически в момент вызова

Таблица виртуальных функций

- Координирующая таблица (vtable)
- Указатель на vtable хранится в каждом объекте
- Содержит адреса динамически связанных методов объекта
- Выбор реализации метода при вызове осуществляется определением адреса требуемого из таблицы виртуальных методов

Таблица виртуальных функций в памяти

```
struct Person {
    virtual ~Person() {}
    string name() const;
    virtual string position() const = 0;
};

struct Student : Person {
    string position() const;
    virtual int group();
};
```

Person		
0	~Person	0xAB22
1	position	0x0000

Student		
0	~Student	0xAB46
1	position	0xAB68
2	group	0xAB8A

Таблица виртуальных функций в памяти

```
struct Person {
    virtual ~Person() {}
    virtual string position() const = 0;
};

struct Teacher : Person {
    string position() const;
    virtual string course();
};
struct Professor : Teacher {
    string position() const;
    virtual string thesis();
};
```

Person		
0	~Person	0xAB20
1	position	0x0000

Teacher		
0	~Teacher	0xAB48
1	position	0xAB60
2	course	0xAB84

Professor		
0	~Professor	0xABA8
1	position	0xABB4
2	course	0xAB84
3	thesis	0xABC8

Виртуальные ф-ии и полиморфизм

- Виртуальные функции являются основным механизмом динамического полиморфизма

```
class Base{
public:
    virtual void  print() const{std::cout << "Base\n";}
};
class Deriv: public Base{
public:
    void print() const {std::cout << "Derived\n";}
};
void f(const Base& obj){obj.print();}
int main(){
    f(Base());
    f(Deriv());
}
```

Ключевое слово final

- Для виртуальных функций может обозначать, что это конечное переопределение
- Не может применяться к не виртуальным функциям
- Для класса определяет, что от него нельзя дальше наследоваться
- Для определения финального класса может использовать `std::is_final`

Ключевое слово final

```
struct Base{
    virtual void foo();
};

struct A : Base{
    void foo() final; // Base::foo переопределен и A::foo последнее переопределение
    void bar() final; // Ошибка: bar не может быть финальным
};

struct B final : A{ // struct B финальный класс
    void foo() override; // Ошибка: foo не может быть переопределена
};

struct C : B{ // Ошибка: B финальный класс
};
```

Определение типа объекта

Определение типа объекта

- Для определения типа объекта, располагающегося в указателе на базовый класс, можно:
  1. Убедиться, что указатель может ссылаться только на объект базового класса (спецификатор final)
  2. Использовать специальное поле для хранения информации о типе объекта
  3. Использовать механизм виртуальных функций
  4. Использовать dynamic_cast
  5. Использование оператора typeid из модуля typeinfo

- dynamic_cast и typeid реализуют механизм RTTI

Использование поля для хранения типа

```
class Employee {
public:
    enum EmployeeType {MANAGER, EMPLOYEE};
    Employee() : type(EMPLOYEE) {}
    EmployeeType getType() const {
        return type;
    }
protected:
    Employee(EmployeeType type) : type(type) {}
private:
    EmployeeType type;
};

class Manager : public Employee {
public:
    Manager() : Employee(MANAGER), level(0) {}
    int getLevel() const {
        return level;
    }
private:
    int level;
};
```

Определение типа через поле

```
void printEmployee (const Employee *employee) {
    switch (employee->getType()) {
        case Employee::MANAGER:
            const Manager *manager = (const Manager*)employee;
            cout << manager->getLevel() << endl;
        case Employee::EMPLOYEE:
            cout << employee->getName() << endl;
    }
}

void printList(const vector<Employee*>& employees) {
    for (Employee* current : employees) {
        printEmployee(current);
    }
}
```

Функции приведения типов

- static_cast – производится на этапе компиляции
  - TYPE static_cast<TYPE> (object);
- dynamic_cast – производится во время работы программы
  - TYPE& dynamic_cast<TYPE&> (object);
  - TYPE* dynamic_cast<TYPE*> (object);
- dynamic_cast – работает только с полиморфными классами

Приведение типов

- Использование ссылки производного класса допустимо везде, где предполагается ссылка базового класса

```
Manager manager(“Name”, “Surname”, “Sales”);
Employee &ref = manager; // Manager& -> Employee&
Employee *ptr = &manager; // Manager* -> Employee*
```

- Допустимо присвоение переменной базового класса объекта производного
- При этом используется конструктор копирования родительского класса

```
Manager manager(“Name”, “Surname”, “Sales”);
Employee employee = manager; // Employee(“Name”, “Surname”);
```

Приведение типов с модификаторами доступа

- При использовании **public** наследования: использование ссылки на базовый класс допустимо везде

```
class Class {};
class PublicChild : public Class{};
class ProtectedChild : protected Class{};
class PrivateChild : private Class{};
```

- При наследовании с модификатором **protected**: о том что *Class* является базовым для *ProtectedChild* знают сам класс и его наследники
- При использовании модификатора **private**: приведение ссылки к базовому классу допустимо только внутри *PrivateChild*

Интерфейсы

Чистая виртуальная функция

- Функция, которая объявляется в базовом классе, но не имеет в нем определения

- Всякий производный класс обязан иметь свою собственную версию

- Для объявления чистой виртуальной функции следует:
  1. Использовать ключевое слово `virtual`, расположив его перед типом возвращаемого значения
  2. Указать `= 0;` после списка аргументов
  3. Исключить тело функции - оставить её без реализации

Абстрактный класс

- Любой класс, содержащий по крайней мере одну чистую виртуальную функцию, является абстрактным
- Предназначен для хранения общей реализации и поведения некоторого множества дочерних классов
- Объекты абстрактного класса создать нельзя
- Рекомендуется добавлять чисто виртуальный деструктор

Использование чистых виртуальных функций

- В C++ отсутствует специальная синтаксическая конструкция для определения интерфейса
- Интерфейсом является класс, содержащий только **public** секцию и только чистые виртуальные методы
- Интерфейс не должен содержать поля
- Каждый интерфейс является абстрактным классом, но не каждый абстрактный класс интерфейс
- При использовании интерфейс реализуют, абстрактный класс – наследуют

Использование чистых виртуальных функций

```
class Base{
public:
    virtual void  print() = 0;
};
class Deriv: public Base{
public:
    void print() const {std::cout << "Derived\n";}
};
void f(const Base* obj){obj->print();}
int main(){
    f(new Base()); //Ошибка
    f(new Deriv());
}
```

Виртуальное наследование

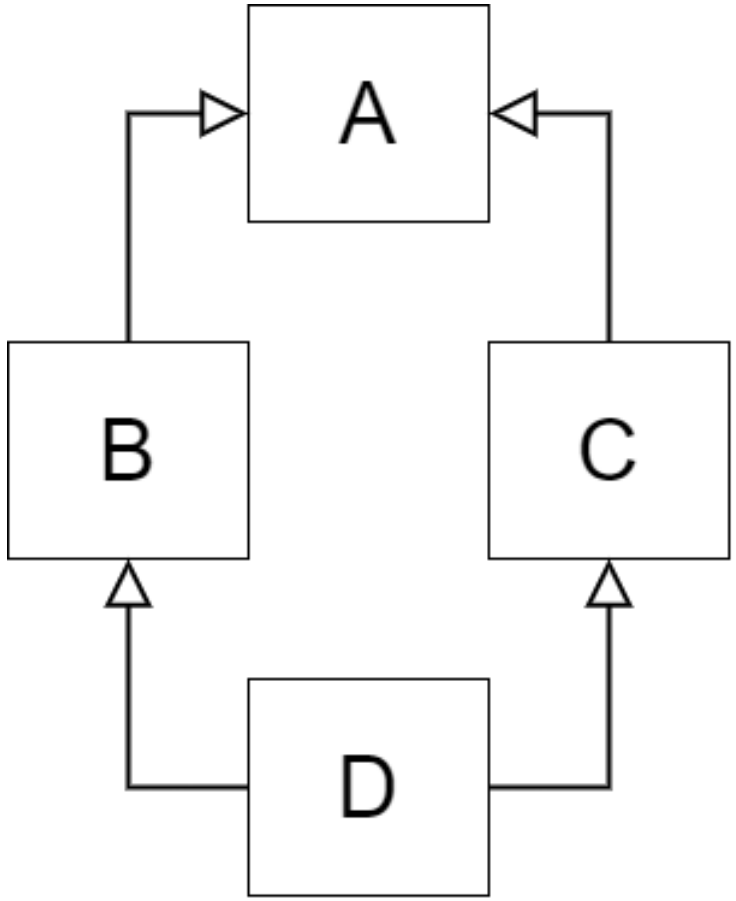
Проблема «ромбовидного» наследования

```
class A{
public:
    void foo();
};

class B: public A{
};

class C: public A{
};

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



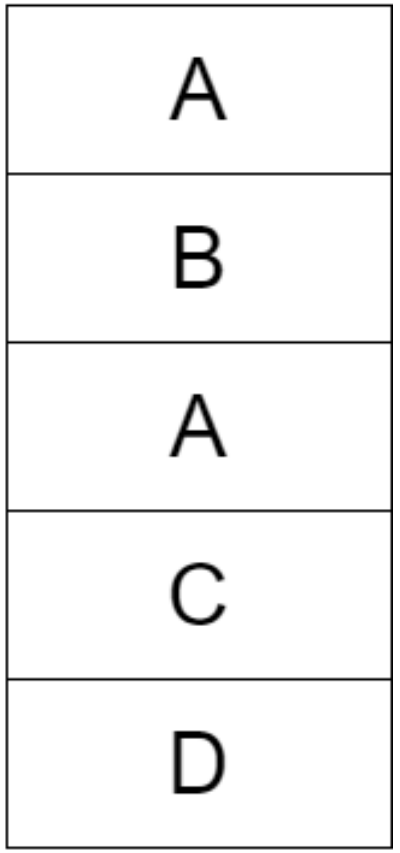
Проблема «ромбовидного» наследования

```
class A{
public:
    void foo();
};

class B: public A{
};

class C: public A{
};

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



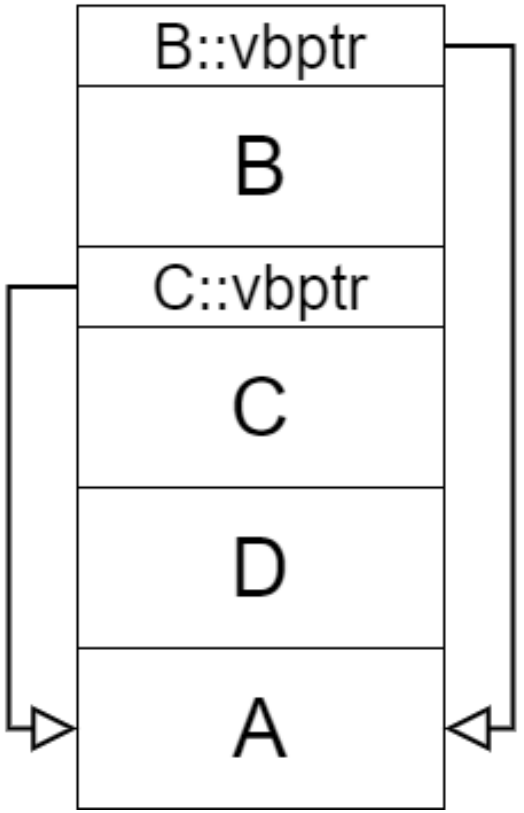
Проблема «ромбовидного» наследования

```
class A{
public:
    void foo();
};

class B: virtual public A{
};

class C: virtual public A{
};

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



Виртуальное наследование

- Необходимо явно вызывать конструкторы всех виртуально унаследованных классов
- Необходимо стараться избегать множественного наследования не от интерфейсов
- Опасность вызова функции из класса «брата»

Дружественность

Дружественные функции

- Дружественные функции:
  - Имеют доступ к **private** и **protected** полям класса
  - Не являются методом класса
  - Дружественность объявляется внутри класса спецификатором **friend**

- Дружественность не наследуется

Дружественные классы

- Дружественный класс имеет доступ к private и protected полям класса

- Все методы дружественного класса становятся дружественными