Виртуальные функции и их переопределение

При вызове функции программа должна определять, с какой именно реализацией функции соотносить этот вызов, то есть связать вызов функции с самой функцией. В C++ есть два типа связывания - статическое и динамическое.

```cpp
#include <iostream>

class Person
{
public:
    Person(std::string name): name{name}
    { }
    void print() const
    {
        std::cout << "Name: " << name << std::endl;
    }
private:
    std::string name;       //  имя
};
class Employee: public Person
{
public:
    Employee(std::string name, std::string company): Person{name}, company{company}
    { }
    void print() const
    {
        Person::print();
        std::cout << "Works in " << company << std::endl;
    }
private:
    std::string company;    // компания
};

```

Когда вызовы функций фиксируются до выполнения программы на этапе компиляции, это называется статическим связыванием (static binding), либо ранним связыванием (early binding).

При этом вызов функции через указатель определяется исключительно типом указателя, а не объектом, на который он указывает.

```cpp
int main()
{
    Person tom {"Tom"};
    Person* person {&tom};
    person->print();     // Name: Tom

    Employee bob {"Bob", "Microsoft"};
    person = &bob;
    person->print();    // Name: Bob
}
```

класс Employee наследуется от класса Person, но оба этих класса определяют функцию `print()`, которая выводит данные об об объекте. В функции main создаем два объекта и поочередно присваиваем их указателю на тип **Person** и вызываем через этот указатель функцию print. Однако даже если этому указателю присваивается адрес объекта Employee, то все равно вызывает реализация функции из класса Person:

```
Name: Tom
Name: Bob
```

Динамическое связывание и виртуальные функции

Другой тип связывания представляет динамическое связывание (dynamic binding), еще называют поздним связыванием (late binding), которое позволяет на этапе выполнения решать, функцию какого типа вызвать. Для этого в языке C++ применяют **виртуальные функции**. Для определения виртуальной функции в базовом классе функция определяется с ключевым словом **virtual**. Причем данное ключевое слово можно применить к функции, если она определена внутри класса. А производный класс может **переопределить** ее поведение.

```cpp
#include <iostream>

class Person
{
public:
    Person(std::string name): name{name}
    { }
    virtual void print() const  // виртуальная функция
    {
        std::cout << "Name: " << name << std::endl;
    }
private:
    std::string name;
};
class Employee: public Person
{
public:
    Employee(std::string name, std::string company): Person{name}, company{company}
    { }
    void print() const
    {
        Person::print();
        std::cout << "Works in " << company << std::endl;
    }
private:
    std::string company;
};
```

```cpp

int main()
{
    Person tom {"Tom"};
    Person* person {&tom};
    person->print();     // Name: Tom
    Employee bob {"Bob", "Microsoft"};
    person = &bob;
    person->print();    // Name: Bob
                        // Works in Microsoft
}
```

базовый класс Person определяет виртуальную функцию print, а а производный класс Employee **переопределяет** ее. В примере, где функция `print` не была виртуальной, класс Employee не не переопределял, а скрывал ее. Теперь при вызове функции print для объекта Employee через указатель Person* будет вызываться реализация функции именно класса Employee. В этом и состоит отличие переопределения виртуальных функций от скрытия.

Вывод на экран

```
Name: Tom
Name: Bob
Works in Microsoft
```

Класс, который определяет или наследует виртуальную функцию, еще назвается **полиморфным** (polymorphic class). То есть в данном случае Person и Employee являются полиморфными классами.

вызов виртуальной функции через имя объекта всегда разрешается статически.

```
Employee bob {"Bob", "Microsoft"};
Person p = bob;
p.print();  // Name: Bob - статическое связывание
```

Динамическое связывание возможно только через указатель или ссылку.

```
Employee bob {"Bob", "Microsoft"};
Person &p {bob};    // присвоение ссылке
p.print();  // динамическое связывание

Person *ptr {&bob};    // присвоение адреса указателю
ptr->print();  // динамическое связывание
```

При определении вирутальных функций есть ряд ограничений. Чтобы функция попадала под динамическое связывание, в производном классе она должна иметь тот же самый набор параметров и возвращаемый тип, что и в базовом классе. Например, если в базовом классе виртуальная функция определена как константная, то в производном классе она тоже должна быть константной. Если же функция имеет разный набор параметров или несоответствие по константности, то мы будем иметь дело со скрытием функций, а не переопределением. И тогда будет применяться статическое связывание.

Ключевое слово override

Чтобы явным образом указать, что мы хотим переопредлить функцию, а не скрыть ее, в производном классе после списка параметров функции указывается слово **override**

```
#include <iostream>

class Person
{
public:
    Person(std::string name): name{name}
    { }
    virtual void print() const  // виртуальная функция
    {
        std::cout << "Name: " << name << std::endl;
    }
private:
    std::string name;
};
class Employee: public Person
{
public:
    Employee(std::string name, std::string company): Person{name}, company{company}
    { }
    void print() const override // явным образом указываем, что функция переопределена
    {
        Person::print();
        std::cout << "Works in " << company << std::endl;
    }
private:
    std::string company;
};
```

```
int main()
{
    Person tom {"Tom"};
    Person* person {&tom};
    person->print();     // Name: Tom
    Employee bob {"Bob", "Microsoft"};
    person = &bob;
    person->print();     // Name: Bob
                         // Works in Microsoft
}
```

здесь выражение `void print() const override` указывает, что мы явным образом хотим переопределить функцию print. **override** явным образом указывает компилятору, что это переопределяемая функция. И если она не соответствует виртуальной функции в базовом классе по списку параметров, возвращаемому типу, константности, или в базовом классе вообще нет функции с таким именем, то компилятор при при компиляции сгенерирует ошибку. И по ошибке мы увидим, что с нашей переопределенной функцией что-то не так. Если же **override** не указать, то компилятор будет считать, что речь идет о скрытии функции, и никаких ошибок не будет генерировать, компиляция пройдет успешно.

Поэтому, при переопределении виртуальной функции в производном классе лучше указывать слово **override**

C++: Чистые виртуальные функции и абстрактные классы

Абстрактные классы - это классы, которые содержат или наследуют без переопределения хотя бы одну чистую виртуальную функцию. Абстрактный класс определяет интерфейс для переопределения производными классами.

Иногда возникает необходимость определить класс, который не предполагает создания конкретных объектов. Например, класс фигуры. В реальности есть конкретные фигуры: квадрат, прямоугольник, треугольник, круг и так далее. Однако абстрактной фигуры самой по себе не существует. В то же время может потребоваться определить для всех фигур какой-то общий класс, который будет содержать общую для всех функциональность. И для описания подобных сущностей используются абстрактные классы.

Что такое **чистые виртуальные функции** (pure virtual functions)? Это функции, которые не имеют определения. Цель подобных функций - просто определить функционал без реализации, а реализацию определят производные классы. Чтобы определить виртуальную функцию как чистую, ее объявление завершается значением "=0".

абстрактный класс, который представляет геометрическую фигуру:

```
class Shape
{
public:
    virtual double getSquare() const = 0;     // площадь фигуры
    virtual double getPerimeter() const = 0;  // периметр фигуры
};
```

Класс Shape является абстрактным, потому что содержит виртуальную функцию (две штуки).

ни одна из функций не имеет никакой реализации.

производный класс от Shape должен будет предоставить для этих функций свою реализацию.

мы не можем создать объект абстрактного класса:

```
Shape shape{};
```

два класса-наследника от абстрактного класса Shape - Rectangle (прямоугольник) и Circle (круг).

При создании классов-наследников все они должны либо определить для чистых виртуальных функций конкретную реализацию, либо повторить объявление чистой виртуальной функции. Во втором случае производные классы также будут абстрактными.

Circle, и Rectangle являются конкретными классами и реализуют все виртуальные функции.

```cpp
#include <iostream>

class Shape
{
public:
    virtual double getSquare() const = 0;     // площадь фигуры
    virtual double getPerimeter() const = 0;  // периметр фигуры
};
class Rectangle : public Shape  // класс прямоугольника
{
public:
    Rectangle(double w, double h) : width(w), height(h)
    { }
    double getSquare() const override
    {
        return width * height;
    }
    double getPerimeter() const override
    {
        return width * 2 + height * 2;
    }
private:
    double width;   // ширина
    double height;  // высота
};
```

```cpp
class Circle : public Shape     // круг
{
public:
    Circle(double r) : radius(r)
    { }
    double getSquare() const override
    {
        return radius * radius * 3.14;
    }
    double getPerimeter() const override
    {
        return 2 * 3.14 * radius;
    }
private:
    double radius;  // радиус круга
};

int main()
{
    Rectangle rect{30, 50};
    Circle circle{30};

    std::cout << "Rectangle square: " << rect.getSquare() << std::endl;
    std::cout << "Rectangle perimeter: " << rect.getPerimeter() << std::endl;
    std::cout << "Circle square: " << circle.getSquare() << std::endl;
    std::cout << "Circle perimeter: " << circle.getPerimeter() << std::endl;
}
```

Вывод на экран

```
Rectangle square: 1500
Rectangle perimeter: 160
Circle square: 2826
Circle perimeter: 188.4
```

Стоит отметить, что абстрактный класс может определять и обычные функции и переменные, может иметь несколько конструкторов, но при этом нельзя создавать объекты этого абстрактного класса.

```cpp
class Shape
{
public:
    Shape(int x, int y): x{x}, y{y}
    {}
    virtual double getSquare() const = 0;      // площадь фигуры
    virtual double getPerimeter() const = 0;   // периметр фигуры
    void printCoords() const
    {
        std::cout << "X: " << x << "\tY: " << y << std::endl;
    }
private:
    int x;
    int y;
};
class Rectangle : public Shape  // класс прямоугольника
{
public:
    Rectangle(int x, int y, double w, double h) : Shape{x, y}, width(w), height(h)
    { }
    double getSquare() const override
    {
        return width * height;
    }
    double getPerimeter() const override
    {
        return width * 2 + height * 2;
    }
private:
    double width;   // ширина
    double height;  // высота
};
```

```cpp
class Circle : public Shape     // круг
{
public:
    Circle(int x, int y, double r) : Shape{x, y}, radius(r)
    { }
    double getSquare() const override
    {
        return radius * radius * 3.14;
    }
    double getPerimeter() const override
    {
        return 2 * 3.14 * radius;
    }
private:
    double radius;  // радиус круга
};

int main()
{
    Rectangle rect{0, 0, 30, 50};
    rect.printCoords();     // X: 0    Y: 0

    Circle circle{10, 20, 30};
    circle.printCoords();   // X: 10   Y: 20
}
```

случае класс Shape также имеет две переменных, конструктор, который устанавливает их значения, и невиртуальную функцию, которая выводит их значения. В производных классах также необходимо вызвать этот конструктор.

объект абстрактного класса с помощью собственного конструктора создать нельзя