Лекция 4.
Дружественные функции и классы. Виртуальные функции.

2 семестр

Лектор: ст.пр. Бельченко Ф.М.

«Дружественность» при наследовании

Скрытие данных - фундаментальная концепция объектно-ориентированного программирования. Оно ограничивает доступ закрытых членов извне класса.

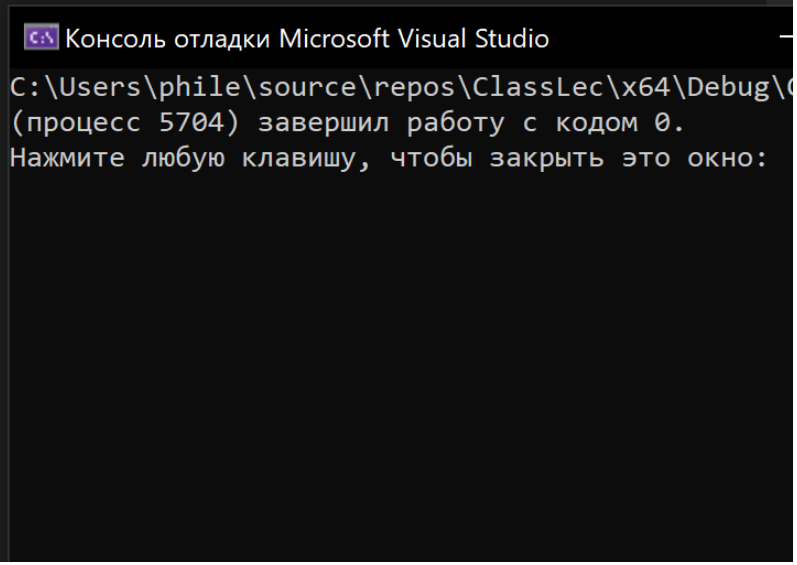
Однако в C ++ есть функции, называемые **дружественными функциями**, которые «Нарушают» это правило и позволяют нам получать доступ к функциям-членам извне класса.

Дружественная функция может обращаться к **частным** и **защищенным** данным класса. Мы объявляем дружественную функцию, используя ключевое слово friend внутри тела класса.

```
class className {
    ... .. ...
    friend returnType functionName(arguments);
    ... .. ...
}
```

Пример дружественной функции

```cpp
#include <iostream>
using namespace std;

class Distance {
private:
    int meter;

    // дружественная функция
    friend int addFive(Distance);

public:
    Distance() : meter(0) {}

};

// Описание дружественной функции
int addFive(Distance d) {
    d.meter += 5;
    return d.meter;
}

int main() {
    Distance D;
    cout << "Distance: " << addFive(D);
    return 0;
}
```



Пример дружественной функции для двух классов

```cpp
#include <iostream>
using namespace std;

// forward declaration
class ClassB;

class ClassA {

public:
    // конструктор инициализирует numA со значением 12
    ClassA() : numA(12) {}

private:
    int numA;

    // Объявление дружественной функции
    friend int add(ClassA, ClassB);
};

class ClassB {

public:
    // конструктор инициализирует numB со значением 1
    ClassB() : numB(1) {}

private:
    int numB;

    // Объявление дружественной функции
    friend int add(ClassA, ClassB);
};

// доступ к членам обоих классов
int add(ClassA objectA, ClassB objectB) {
    return (objectA.numA + objectB.numB);
}

int main() {
    ClassA objectA;
    ClassB objectB;
    cout << "Sum: " << add(objectA, objectB);
    return 0;
}
```

Консоль отладки Microsoft Visual Studio

```
C:\Users\phile\source\repos\ClassLec\x64\Debug\
(процесс 2136) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно:
```

Понятие дружественного класса

Дружественные классы в C++ означают, что все методы одного класса, который объявлен другом, автоматически становятся дружественными методам другого класса, который инициирует дружбу. Это означает, что такие методы получают доступ ко всем членам другого класса.

Поскольку ClassB это дружественный класс, мы можем получить доступ ко всем членам ClassA изнутри ClassB.

Однако мы не можем получить доступ к членам ClassB изнутри ClassA. Это потому, что дружеское отношение в C ++ только предоставляется, а не принимается.

Пример дружественного класса

```cpp
#include <iostream>
using namespace std;

// forward declaration
class ClassB;

class ClassA {
private:
    int numA;

    // объявление дружественного класса
    friend class ClassB;

public:
    // конструктор инициализирующий numA как 12
    ClassA() : numA(12) {}
};

class ClassB {
private:
    int numB;

public:
    // конструктор инициализирующий numB как 1
    ClassB() : numB(1) {}

    // Обращаемся к дружественному классу
    int add() {
        ClassA objectA;
        return objectA.numA + numB;
    }
};

int main() {
    ClassB objectB;
    cout << "Sum: " << objectB.add();
    return 0;
}
```

```
Консоль отладки Microsoft Visual Studio
C:\Users\phile\source\repos\ClassLec\x64\Deb
(процесс 18508) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окн
```

Виртуальные функции

Виртуальная функция — это функция-член, которую предполагается переопределить в производных классах. При ссылке на объект производного класса с помощью указателя или ссылки на базовый класс можно вызвать виртуальную функцию для этого объекта и выполнить версию функции производного класса.

Виртуальные функции обеспечивают вызов соответствующей функции для объекта независимо от выражения, используемого для вызова функции.

Статическое связывание

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.

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.

Пример программы со статическим связыванием

```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
};

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

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

```
Консоль отладки Microsoft Visual Studio
Name: Tom
Name: Bob

C:\Users\phile\source\repos\ClassLec\x64\Debug\(
Нажмите любую клавишу, чтобы закрыть это окно:
```

В данном случае класс Employee наследуется от класса Person, но оба этих класса определяют функцию print(), которая выводит данные 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

Динамическое связывание

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.

Как работают виртуальные функции

Возможность вызова методов производного класса через ссылку или указатель на базовый класс осуществляется с помощью механизма виртуальных функций. Чтобы при вызове **p.name()** вызвался метод класса **Student**, реализуем классы следующим образом:

```
struct Person
{
        virtual string name() const;
};

struct Student: Person
{
        string name() const;
};
```

Перед методом **name** класса **Person** мы указали ключевое слово **virtual**, которое указывает, что метод является виртуальным. Теперь при вызове **p.name()** произойдет вызов метода класса **Student**, несмотря на то, что мы его вызываем через ссылку на базовый класс **Person**.

Виртуальные функции и указатели

Аналогичная ситуация и с указателями:

```
Student s;
Person *p = &s ;
p->name(); //вызовется Student::name();
Person n;
p = &n;
p->name(); //вызовется Person::name()
```

Если с некоторого класса в иерархии наследования метод стал виртуальным, то во всех производных от него классах он будет виртуальным, вне зависимости от того, указано ли ключевое слово **virtual** в классах наследниках.

Механизм виртуальных функций реализует полиморфизм времени выполнения: какой виртуальный метод вызовется будет известно только во время выполнения программы.

Пример с виртуальной функцией

```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;
};

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
}
```

```
Выбрать Консоль отладки Microsoft Visual Studio
Name: Tom
Name: Bob
Works in Microsoft

C:\Users\phile\source\repos\ClassLec\x64\Debug\Cl
2360) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно:
```

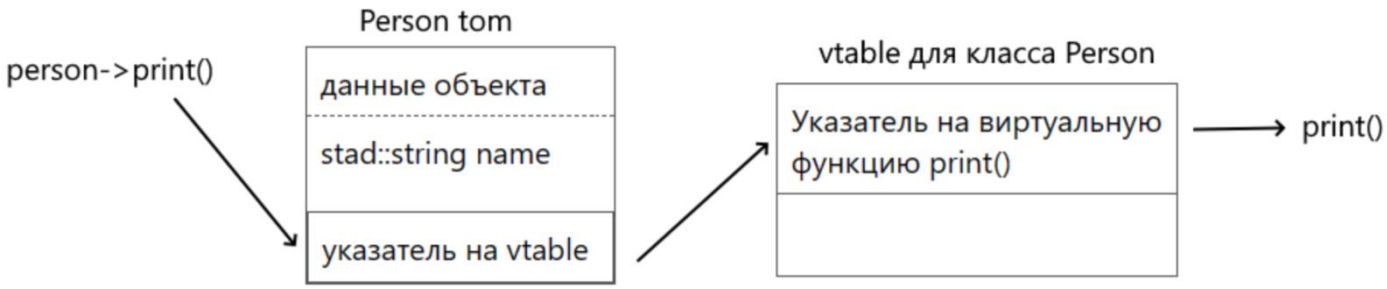
Что нужно учитывать при работе с виртуальными функциями?

Стоит отметить, что виртальные функции имеют свою цену - объекты классов с виртуальными функциями требуют немного больше памяти и немного больше времени для выполнения. Поскольку при создании объекта полиморфного класса (который имеет виртуальные функции) в объекте создается специальный указатель. Этот указатель используется для вызова любой виртуальной функции в объекте.

Специальный указатель указывает на таблицу указателей функций, которая создается для класса. Эта таблица, называемая виртуальной таблицей или vtable, содержит по одной записи для каждой виртуальной функции в классе.

Принцип выполнения виртуальных функций

Когда функция вызывается через указатель на объект базового класса, происходит следующая последовательность событий:



1. Указатель на vtable в объекте используется для поиска адреса vtable для класса.

2. Затем в таблице идет поиск указателя на вызываемую виртуальную функцию.

3. Через найденный указатель функции в vtable вызывается сама функция. В итоге вызов виртуальной функции происходит немного медленнее, чем прямой вызов невиртуальной функции, поэтому каждое объявление и вызов виртуальной функции несет некоторые накладные расходы.

Запет переопределения

С помощью спецификатора **final** мы можем запретить определение в производных классах функций, которые имеют то же самое имя, возвращаемый тип и список параметров, что и виртуальная функция в базовом классе.

```
1  class Person
2  {
3  public:
4      virtual void print() const final
5      {
6
7      }
8  };
9  class Employee : public Person
10 {
11 public:
12     void print() const override     // Ошибка!!!
13     {
14
15     }
16 };
```

Запет переопределения

Также можно переопределить функцию базового класса, но запретить ее переопределение в дальнейших производных классах:

```
1  class Person
2  {
3  public:
4      virtual void print() const // переопределение разрешено
5      {
6
7      }
8  };
9  class Employee : public Person
10 {
11 public:
12     void print() const override final   // в классах, производных от Employee переопределение запрещено
13     {
14
15     }
16 };
```