Виртуальные функции

динамический полиморфизм подтипов

- Полиморфизм — одна из трех основных парадигм ООП.
- Способность объекта вести себя по разному в зависимости от контекста.

«*один интерфейс (как перечень объявлений) —*
много реализаций (определений, связываемых с этими объявлениями)»

Бьярн Страуструп о полиморфизме

- Виртуальной функцией называется функция-член класса (aka метод), которая может быть переопределена в классах-наследниках.
- Конкретная реализация будет определена в рантайме.

```
struct A {
      virtual void foo() { cout << "A::foo()" << endl; }
};

struct B : public A {
      virtual void foo() { cout << "B::foo()" << endl; }
};

B object_B;
A *point_to_Object = &object_B;
point_to_Object->foo();
```

Что на экране?

godbolt.org

- Более наглядный пример – реализация оружия в игре.

```
struct Weapon {
      virtual void reload();
      virtual void shoot();
};
```

- Более наглядный пример – реализация оружия в игре.

```
struct Weapon {
      virtual void reload();
      virtual void shoot();
};

struct AR15 : public Weapon {
      virtual void reload();
      virtual void shoot();
};

struct AK101 : public Weapon {
      virtual void reload();
      virtual void shoot();
};
```

- Более наглядный пример – реализация оружия в игре

```
Weapon* Arms[4]= {
                        new AR15(),
                        new USP(),
                        new Knife(),
                        new AK101()
                    };

void event(Event e) {
      int CurrentWeapon = 0;
      if(e.isLeftClick())
            Arms[CurrentWeapon]->shoot();
}
```

- Ключевое слово `virtual` позволяет переопределять метод в классах-наследниках, таким образом добиваясь полиморфного поведения.
- Любой класс с виртуальным методом имеет таблицу виртуальных функций, по которой определяется реализация в рантайме.
- Это накладывает оверхед в виде лишнего перехода по указателю к искомой реализации.
- При замещении виртуальных функций требуется полное совпадение сигнатуры.

```
struct Weapon {
      virtual void shoot();
};

struct AR15 : public Weapon {
      virtual int shoot();
};
```

- Ключевое слово `virtual` позволяет переопределять метод в классах-наследниках, таким образом добиваясь полиморфного поведения.
- Любой класс с виртуальным методом имеет таблицу виртуальных функций, по которой определяется реализация в рантайме.
- Это накладывает оверхед в виде лишнего перехода по указателю к искомой реализации.
- При замещении виртуальных функций требуется полное совпадение сигнатуры.

```
struct Weapon {
      virtual void shoot();
};

struct AR15 : public Weapon {
      virtual int shoot();  // error
};
```

Абстрактный класс

- Базовый класс, не предполагающий непосредственного инстанцирования.
- Напротив, абстрактный класс служит «рецептом» для наследования и построения более конкретных сущностей.
- Абстрактным классом является любой класс с хотя бы одной чисто-виртуальной функцией.

```
struct Weapon {
      virtual void shoot() = 0;
};
```

- Как по другому назвать такой класс? Такие языки как Java, C#, Go и т.д. имеют специальное слово для таких классов…

Он же интерфейс

```
struct Weapon {
      virtual void shoot() = 0;
};
```

- Мы не хотим создавать инстансы Weapon, поскольку это слишком размытое определение оружия.
- Из этого интерфейса мы понимаем, что из оружия можно стрелять, и нам этого достаточно для производства конкретных реальных прототипов вооружения нашей игры.

```
struct DeathFinger : public Weapon {
      virtual void shoot() { cout << "piu-piu" << endl; }
};
```

Зоопарк

- Создадим базовый класс животного, он нужен как рецепт для создания производных классов.
- Но мы не хотели бы создавать инстанс общего класса животного, а вместо этого хотели бы создать какой-то конкретный.

```
struct Animal {
      string name;
      Animal(string n) : name(n) { }
      virtual void speak() = 0;
};

struct Cat : public Animal {
      Cat(string n) : Animal(n) { }
      void speak() { cout << name <<" say meow-meow" << endl; }
};
```

Еще парочку

```
struct Dog : public Animal {
      Dog(string n) : Animal(n) { }
      void speak() { cout << name <<" say gaw-gaw" << endl; }
};

struct Cow : public Animal {
      Cow(string n) : Animal(n) { }
      void speak() { cout << name <<" say moow" << endl; }
};

struct Lion : public Animal {
      Lion(string n) : Animal(n) { }
      void speak() { cout << name <<" say rrr-rrr" << endl; }
};
```

```
Animal* animals[4]= {
                        new Dog("Bob"),
                        new Cat("Murka"),
                        new Cow("Murawushka"),
                        new Lion("King")
                    };
for(int k=0; k < 4; ++k)
      animals[k]->speak();
```

- Что на экране?

godbolt.org

```
Animal* animals[4]= {
                        new Dog("Bob"),
                        new Cat("Murka"),
                        new Cow("Murawushka"),
                        new Lion("King")
                    };
for(int k=0; k < 4; ++k)
      animals[k]->speak();
```

- Что на экране?

```
Bob say gaw-gaw
Murka say meow-meow
Murawushka say moow
King say rrr-rrr
```

godbolt.org

Абстрактный класс

- Как и всякий класс, абстрактный класс может иметь явно определенный конструктор. Из конструктора можно вызывать методы класса.
- Но обращение из конструктора к чистым виртуальным функциям приведут к ошибкам во время выполнения программы.
- Абстрактный класс нельзя применять для задания типа параметра функции, или в качестве типа возвращаемого значения.
- Его нельзя использовать при явном приведении типов. Зато можно определять ссылки и указатели на абстрактные классы.

```
void foo(Animal animal) {
      animal.sound();              // error, but
}
```

Абстрактный класс

- Как и всякий класс, абстрактный класс может иметь явно определенный конструктор. Из конструктора можно вызывать методы класса.
- Но обращение из конструктора к чистым виртуальным функциям приведут к ошибкам во время выполнения программы.
- Абстрактный класс нельзя применять для задания типа параметра функции, или в качестве типа возвращаемого значения.
- Его нельзя использовать при явном приведении типов. Зато можно определять ссылки и указатели на абстрактные классы.

```
void foo(Animal* animal) {
      animal->sound();                  // ok
}
```

```
struct User {
	Documents* docs;
	User(string path) : docs(new Documents(path)) { }
	~User() { delete docs; }
	virtual void show_docs();
};

struct Customer : public User {
	Customer(string path) : User(path) { }
	virtual void show_docs() override;
};
```

В чем проблема?

```
struct User {
      Documents* docs;
      User(string path) : docs(new Documents(path)) { }
      ~User() { delete docs; }
      virtual void show_docs();
};

struct Customer : public User {
      Customer(string path) : User(path) { }
      virtual void show_docs() override;
};
```

В чем проблема?

```
void foo() {
      Customer c(path);
      c.show_docs();
} // leak
```

- При уничтожении объекта `Customer` не происходит освобождение памяти, которая была выделена в базовом классе `User` при создании `Customer`.

- Для корректного освобождения памяти в базовых классах, деструктор базового класса должен быть помечен как `virtual`.

```
struct User {
	Documents* docs;
	User(string path) : docs(new Documents(path)) { }
	virtual ~User() { delete docs; }
	virtual void show_docs();
};

struct Customer : public User {
	Customer(string path) : User(path) { }
	virtual void show_docs() override;
};
```

```
struct User {
      Documents* docs;
      User(string path) : docs(new Documents(path)) { }
      virtual ~User() { delete docs; }
      virtual void show_docs();
};

struct Customer : public User {
      Customer(string path) : User(path) { }
      virtual void show_docs() override;
};


 void foo() {
       Customer c(path);
       c.show_docs();
 } // ok
```

- Порядок вызова деструкторов при наследовании с определенным виртуальным деструктором – от производных к базовым классам.

```
struct A {
	virtual void print() { cout << "I'm B object" << endl; }
	virtual ~A() { cout << "A::dtor" << endl; }
};

struct B : public A {
	void print() override { cout << "I'm B object" << endl; }
	virtual ~B() { cout << "B::dtor" << endl; }
};

void foo() {
	B b;
	b.print();
}
```

- Что на экране?

godbolt.org

Итог

- Если у класса имеются виртуальные функции, имеет смысл создать для него виртуальный деструктор.
- Если класс не содержит виртуальных функций, то скорее всего наследоваться от него не лучшая идея.
- Могут ли быть виртуальные конструкторы?
- Динамический полиморфизм накладывает оверхед в виде индерекшена по таблице виртуальных функций.
- Однако только с помощью полиморфизма можно создавать гибкие расширяемые решения.