Объектно-ориентированное программирование в C++

Наследование и полиморфизм

Виртуальные методы

Сокрытие элементов

Закрытые конструкторы и деструкторы

Наследование функциональности

- Функции базового класса наследуются также как и поля: открытые и защищенные функции доступны в производном классе, закрытые функции не доступны
- В производном классе можно вводить функции полностью совпадающие по сигнатуре; в этом случае реализуется либо *сокрытие*, либо *переопределение*

```
class Animal {
public:
..
    void SayHello();
};
```

```
class Dog : public Animal
{
public:
     void SayHello();
};
```

```
class Cat : public Animal
{
public:
     void SayHello();
};
```

Сокрытие функций

- При *сокрытии* выбор функции осуществляется на основе типа объекта (статически, на стадии компиляции программы)

```
Dog dog("Bobik");
Cat cat("Murzik");
dog.SayHello(); //Dog::SayHello();
cat.SayHello(); //Cat::SayHello();
// Неявное преобразования к Animal
// void Show(Animal& a) {a.SayHello(); }
Show(cat);   //Animal::SayHello();
Show(dog);   //Animal::SayHello();
// Вызов через указатель на Animal
Animal* p = &dog;
p->SayHello(); //Animal::SayHello();
```

```
Dog Bobik says: Rrrr..
Cat Murzik says: Miaw..

Show(Animal& a)
Animal Murzik says: aefhjkuiep
Animal Bobik says: aefhjkuiep
Animal Bobik says: aefhjkuiep
```

Динамический полиморфизм и переопределение

```
void Show(Animal& a) {
        a.SayHello();
}
```

- ООП поддерживает возможность выбора метода в момент выполнения программы на основе *фактического* типа объекта (*динамический полиморфизм*)
- Для реализации динамического полиморфизма в C++ необходимо в базовом классе пометить метод как *виртуальный*

```
class Animal {
        public: virtual void SayHello();
};
```

- Целесообразность в виртуальных методах возникает, когда в коде возможна работа с объектом *через ссылку или указатель базового типа* и ожидается *полиморфное* поведение объектов

Таблица виртуальных методов

```
Animal* p = &dog;
cout << p->getName();
p->SayHello();
```

- При обработке обычных (невиртуальных) методов компилятор определяет адрес функции и подставляет его в точку вызова (*на стадии компиляции*)
- Если в классе определяется хотя бы одна виртуальная функция, то компилятор вводит скрытое-поле vmtp (указатель на таблицу виртуальных методов). В таблице хранятся адреса описанных в классе виртуальных функций.
- При обработке вызовов виртуальных функций компилятор вставляет в объектный код блок инструкций:
  - извлечение адреса таблицы виртуальных методов для конкретного объекта (vmtp);
  - извлечение адреса виртуальной функции из таблицы
  - вызов функции по адресу

Виртуальные функции

- В C++ можно объявить виртуальным любой метод, **кроме конструкторов** и **статических методов**
- Для введения виртуальных методов необходимо использовать ключевое слово **virtual** в базовом классе; в производном классе использовать не обязательно
- Рекомендуется не использовать (=не вызывать) виртуальные функции в конструкторе и деструкторе
- В случае классов с виртуальными функциями рекомендуется вводить *виртуальный деструктор*

```
Animal* p = new Dog("Rex");
// Выбор деструктора зависит от объекта,
// на который указывает p
delete p;
```

Абстрактные классы и функции

- В базовом классе можно вводить функции без реализации; классы-наследники предоставят свои собственные реализации
- Для этого в базовом классе вводятся *чисто виртуальные функции*
- Класс, в котором есть хотя бы одна виртуальная функция, называется *абстрактным классом*
- Абстрактный класс предназначен исключительно для наследования (и полиморфизма); создавать экземпляры абстрактного класса нельзя

```
class Animal {
public:
        virtual void SayHello() const = 0;
        virtual void Move() = 0;
};
```

Абстрактные классы

- Абстрактные классы наряду с чисто виртуальными функциями могут содержать и обычные элементы (функции с реализацией, поля)
- Если производный класс не предоставляет реализацию для всех чисто виртуальных функций, то он тоже является абстрактным

```
class Figure {
    Point* points;
    int nPoints;
public:
    virtual double Volume() = 0;
    virtual double Length() = 0;
    virtual void setPoints(Point* points, int size);
    virtual Point* getPoints();
};
```

Закрытые деструкторы

- Конструкторы и деструкторы можно объявлять как *закрытые* и *защищенные*.
- *Закрытый конструктор* запрещает явное создание экземпляров; обеспечить возможность получения (создания) объектов можно через открытый метод класса (как правило, статический)
- *Закрытый деструктор* исключает возможность введения переменных-объектов, размещаемых в стэке (локальные переменные) или в статической памяти (глобальные переменные).
- Объекты такого класса можно размещать исключительно в динамической памяти, так как это позволяет «обойтись» без обращения к деструктору
- Освобождение памяти в этом случае можно выполнять в отдельном методе, который явно вызывается
- *Закрытый деструктор* исключает возможность наследования
- *Защищенный деструктор* разрешает наследование, но как и закрытый деструктор, разрешает хранение объектов только в динамической памяти.

Сокрытие .. и раскрытие

- В производном классе можно вводить *одноименные* элементы по отношению к элементам базового класса;
- Может различаться и тип элемента, и уровень доступа к элементу
- В этом случае происходит *сокрытие* элементов базового класса

```
Child ch;
ch.x = 3.14f;
// Не доступно
ch.Do();
```

- В C++ поддерживается возможность обратиться к скрытому элементу через *область видимости* (имя базового класса)

```
// "раскрытие"
ch.Parent::Do();
```

```
ch.f(13); // не доступно
ch.f(1, 3);
ch.Parent::f(13);
```

```
class Parent
{
private:
    int x;
public:
    void Do() {};
    void f(int x) {};
};
class Child : public Parent
{
    void Do() {};
public:
    double x;
    int f(int x, int y);
};
```