Программирование в задачах радиолокации

КМБО-02-19

24 июня 2020 г.

1 ООП, принципы, отношения объектов

Замечание 1 *Этот раздел - адаптация первой главы опросника по МиСП.*

1.1 ООП

Определение 1 ***Объектно-ориентированный подход*** *- подход, при котором предметная область представлена совокупностью объектов, взаимодействующих между собой с помощью сообщений.*

Определение 2 ***Предметная область*** *- множество предметов и условий, в рамках которых происходит работа и выполнение задачи.*

Определение 3 ***Объект*** *- описание сущности из предметной области.*

Определение 4 ***Объектно-ориентированное программирование*** *- методология программирования, основанная на представлении программы в виде совокупности объектов.*

1.2 Свойства объекта

- **Определение 5** ***Состояние*** *- каждая уникальная комбинация свойств объекта(атрибутов) и связей с другими объектами. Меняется со временем.*
- **Определение 6** ***Поведение*** *- определяется методами - определяет действия объекта относительно внешних связей и манипуляций с собственными свойствами.*
- **Определение 7** ***Идентичность*** *- свойство объекта, отличающее его ото всех других объектов. В C++ это адрес.*

1.3 Принципы объектной модели

Основные принципы:

- **Определение 8** ***Абстракция*** *- выделение наиболее существенных характеристик некоторого объекта, отличающих его от всех других объектов, важных с точки зрения дальнейшего рассмотртения.*
- **Определение 9** ***Инкапсуляция*** *- отделение друг от друга элементов объекта, определяющих его устройство и поведение. Служит для изоляции абстракции от реализации.*

- **Определение 10** ***Модульность*** *- возможность спроектировать взаимодействия объектов так, чтобы объекты между собой взаимодействовали неинтенсивно, но внутри самих объектов происходила интенсивная работа. Используется для переиспользования объектов.*

- **Определение 11** ***Иерархия*** *- упорядочивание абстракций по уровням.*

Дополнительные принципы:

- **Определение 12** ***Типизация*** *- защита от неправильного использования объекта, т.е. от ситуации, когда объект одного класса используется вместо другого.*

- **Определение 13** ***Устойчивость*** *- возможность объекта переживать породивший его процесс.*

- **Определение 14** ***Параллелизм*** *- наличие в системе нескольких потоков управления одновременно.*

1.4 Принципы объектно-ориентированного программирования

1.4.1 Сами принципы

1. **Абстракция** - см. выше.

2. **Инкапсуляция** - см. выше.

3. **Определение 15** ***Наследование*** *- механизм создания новых объектов на основе уже существующих путём сохранения свойств и поведения с возможностью расширения функциональности и переопределения.*

4. **Определение 16** ***Полиморфизм*** *- возможность создавать объекты с одинаковым интерфейсом и различной реализацией.*

1.4.2 Примеры и особенности реализации принципов инкапсуляции, наследования и полиморфизма в C++

Инкапсуляция в C++ реализуется за счёт разделения атрибутов и методов класса(объекта) на публичные, защищённые и скрытые области видимости, реализующиеся с помощью спецификаторов public, protected и private соответственно.

В листинге ниже представлен класс *Contact*, публичные переменные и методы доступны из основной программы (*main*). Приватные переменные и методы могут прочитаны, вызваны или изменены только самим классом.Попытка напечатать или изменить приватную переменную *mobile_number* из основной программы (*main*) вызовет ошибку при компиляции потому как доступ к приватным данным в классе ограничен.

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

class Contact
{
    private:
        int mobile_number;              // private variable
        int home_number;                // private variable
    public:
        Contact()                       // constructor
        {
            mobile_number = 12345678;
            home_number = 87654321;
        }
        void print_numbers()
        {
            cout << "Mobile number: " << mobile_number;
            cout << ", home number: " << home_number << endl;
        }
};

int main()
{
    Contact Tony;
    Tony.print_numbers();
    // cout << Tony.mobile_number << endl;
    // will cause compile time error
    return 0;
}
```

Наследование в C++ подразделяется на публичное(*public*), защищённое(*protected*) и приватное(*private*).

Разница в следующем:

- **Публичное наследование** - публичные и защищённые данные наследуются без изменения доступа к ним. Т.е. если в исходном классе было поле в защищённой области, то и для наследника это поле останется в защищённой области. Если у предка был публичный метод - он останется публичным и у наследника.
- **Защищённое наследование** - все поля из public и protected родителя становятся protected-полями потомка.
- **Приватное наследование** - поля public и protected предка становятся полями private потомка.

Замечание 2 ***Private поля предка ни в каком случае не наследуются!***

Пример приватного наследования:

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

class Device {
    public:
        int serial_number = 12345678;

        void turn_on() {
            cout << "Device is on" << endl;
        }
};

class Computer: private Device {
    public:
        void say_hello() {
            turn_on();
            cout << "Welcome to Windows 95!" << endl;
        }
};

int main() {
    Device Device_instance;
    Computer Computer_instance;

    cout << "\t Device" << endl;
    cout << "Serial number is: "<< Device_instance.serial_number << endl;
    Device_instance.turn_on();

    // cout << "Serial number is: " << Computer_instance.serial_number << endl;
    // Computer_instance.turn_on();
    // will cause compile time error

    cout << "\t Computer" << endl;
    Computer_instance.say_hello();
    return 0;
}
```

Класс *Computer* теперь использует метод *turn_on*() как и любой приватный метод: *turn_on*() может быть вызван изнутри класса, но попытка вызвать его напрямую из *main* приведет к ошибке во время компиляции. Для базового класса *Device*, метод *turn_on*() остался публичным, и может быть вызван из *main*.

Замечание 3 *Более подробно о наследовании в C++, порядке вызовов конструкторов и деструкторов, виртуальном наследовании и прочем подробно написано в соответствующем разделе в материалах по методам и стандартам программирвания. В этом же разделе приводятся лишь некоторые общетеоретические моменты.*

Полиморфизм в C++ зачастую реализуется с помощью механизма наследования абстрактного класса классами с конкретной реализацией методов. Иными словами, существует **абстрактный класс - класс, в котором есть хотя бы одна чисто виртуальная функция** - от которого так или иначе наследуются другие классы, в которых как раз и создаётся реализация методов, заявленных в абстрактном классе. Также необходимо сказать о таких явлениях, как полиморфные функции: это некоторые функции, способные обрабатывать различные типы входных данных; наверняка Вы помните такие по первому семестру. Конечно, есть и другие проявления полиморфизма.

Замечание 4 *В данном контексте мы говорим о так называемом* ***полиморфизме подтипов****(он же* ***Ad-hoc - полиморфизм****). Он заключается в том, что разным типам входных данных соответствует различное поведение.*

Также различают так называемый ***параметрический полиморфизм****, его суть в том, что для различных типов данных объект обеспечивает одинаковое поведение. Реализацию этого вида полиморфизма в C++ предлагает механизм шаблонов.*

Для примера предлагаю рассмотреть нашу с вами лабораторную работу по написанию стека(её ведь все сделали, я надеюсь?). У нас есть абстрактный класс *StackImplementation*, содержащий интерфейс, который должна поддерживать структура, на основе которой работает стек. Уже от этого класса наследуются $VectorStack, ListStack$ - данные классы поддерживают уже реализации на конкретных структурах данных: векторе и списке соответственно. На следующей странице приведены примеры объявления класса *StackImplementation* и класса $VectorStack$.

```
template <class ValueType>
class StackImplementation {
public:
        // добавление в хвост
        virtual void push(const ValueType& value) = 0;
        // удаление с хвоста
        virtual void pop() = 0;
        // посмотреть элемент в хвосте
        virtual const ValueType& top() const = 0;
        // проверка на пустоту
        virtual bool isEmpty() const = 0;
        // размер
        virtual size_t size() const = 0;
        // виртуальный деструктор
        virtual ~StackImplementation() {};
};

template <class ValueType>
class VectorStack : public MyVector<ValueType>,
virtual public StackImplementation<ValueType>
{
public:
    //конструктор
    VectorStack();
    //конструктор копированием
    VectorStack(const VectorStack& copyVec);
    VectorStack& operator=(const VectorStack& copyVec);
    // Конструктор копирования присваиванием
    VectorStack(VectorStack&& moveVec) noexcept;
    VectorStack& operator=(VectorStack&& moveVec) noexcept;
    // добавление в конец
    void push(const ValueType& value) override;
    // удаление с хвоста
    void pop() override;
    // посмотреть элемент в хвосте
    const ValueType& top() const override;
    // проверка на пустоту
    bool isEmpty() const override;
    // размер
    size_t size() const override;
    // деструктор
    ~VectorStack();

};
```

1.5 Типы отношений объектов

Определение 17 *Отношения - способ организации взаимодействия.*

1.6 Классификация:

- По виду: одно- и дву- направленные;
- По характеру: содержит (**has-a**), является (**is-a**), использует (**uses-a**);
- По кратности: один на один, один объект на много объектов, много на много...

1.6.1 Типы отношений:

Зависимость	Она же может считаться конкретизацией поведения. Рассматривая на примере: один объект обращается к функции другого. При этом объекты **не являются полями друг друга** - отношение ($friend$). Является однонаправленным отношением.
Часть-целое	**Определение 18** ***Агрегация*** *- целое не управляет жизненным циклом частей.* Пример: работник переживает кампанию. Иными словами, если используем агрегацию, то в деструкторе агрегированный объект не удаляем. *has-a*
	Определение 19 ***Композиция*** *- часть принадлжеит только одному объекту, который за неё 'отвечает'.* Пример: часть удаляется в деструкторе объекта-хозяина, а также находится в приватной части. *has-a*
Ассоциация	Объекты используют друг друга для своих нужд. Отношения двунаправленные, никто никому не принадлежит. Прекрасный пример: отношения врача и пациента - это явно не отношения часть-целое! У врача спокойно может быть огромное количество пациентов в день, а у пациента своя жизнь с блэкджеком и программированием в задачах радиолокации. *uses-a*
Наследование	Комментарии тут излишни, всё сказали раньше. *is-a*

Замечание 5 *Примеры на C++ для приведённых выше отношений объектов Вы без проблем можете придумать и сами. Поясним только следующее: пример* ***зависимости*** *мы можем найти, вспомнив класс $MyString$, где нам приходилось делать friend-ом класс 'ostream', чтобы работал корректный вывод;* ***часть-целое*** *- односвязный список - в идеале это композиция, ибо в деструкторе мы разрушаем все node в списке; наследование - очевидный пример со стеком;* ***агрегация*** *- возъмём какой-нибудь класс, который имеет полем std :: string, получим, что класс не несёт ответственности за жизнь этого поля после себя, при этом это поле может быть передано и другим объектам;* ***ассоциация*** *- один класс запрашивает возвращаемые значения методов другого, чтобы конкретизировать свою работу - на пальцах сказать сложно, но если Вам попадётся код с такими отношениями, Вы сразу поймёте.*

Замечание 6 *Если у Вас всё хорошо с английским и Вы не поняли того, что написано выше, предлагаю прочитать статью по ссылке:*

`https://www.learncpp.com/cpp-tutorial/10-1-object-relationships/`