Лекция 11. Методология объектно-ориентированного программирования

Объектно-ориентированное программирование

Определение классов

Кроме использования встроенных типов, таких как int, double и т.д., мы можем определять свои собственные типы или **классы**. Класс представляет составной тип, который может использовать другие типы.

Класс предназначен для описания некоторого типа объектов. То есть по сути класс является планом объекта. А объект представляет конкретное воплощение класса, его реализацию. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. То есть некоторый шаблон - этот шаблон можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек будет представлять **объект** или **экземпляр** этого класса.

Для определения класса применяется ключевое слово **class**, после которого идет имя класса:

```
class имя_класса
{
    // компоненты класса
};
```

После названия класса в фигурных скобках располагаются компоненты класса. Причем после закрывающей фигурной скобки идет точка с запятой.

Например, определим простейший класс:

```
class Person { };

int main()
{

}
```

В данном случае класс называется Person. Как правило, названия классов начинаются с большой буквы. Допустим, данные класс представляет человека. Данный класс пуст, не содержит никаких компонентов, тем не менее он уже представляет новый тип. И после определения класса мы можем определять его переменные или константы:

```
class Person
{

};
int main()
```

```
{
    Person person;
}
```

Но данный класс мало что делает. Класс может определять переменные и константы для хранения состояния объекта и функции для определения поведения объекта. Поэтому добавим в класс Person некоторое состояние и поведение:

```
#include <iostream>

class Person
{
public:
    std::string name;
    unsigned age;
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
};
int main()
{
    Person person;
    // устанавливаем значения полей класса
    person.name = "Tom";
    person.age = 38;
    // вызываем функцию класса
    person.print();
}
```

Теперь класс Person имеет две переменных name и age, которые предназначены для хранения имени и возраста человека соответственно. Переменные класса еще называют **полями** класса. Также класс определяет функцию print, которая выводит значения переменных класса на консоль. Также стоит обратить внимание на модификатор доступа **public:**, который указывает, что идущие после него переменные и функции будут доступны извне, из внешнего кода.

Затем в функции main создается один объект класса Person. Через точку мы можем обратиться к его переменным и функциям.

```
объект.компонент
```

Например, мы можем установить значения полей класса

```
person.name = "Tom";
person.age = 38;
```

Ну и также мы можем вызывать функции у объекта:

```
person.print();
```

Консольный вывод данной программы:

```
Name: Tom    Age: 38
```

Подобным образом можно получать значения переменных объектов

```
#include <iostream>

class Person
{
public:
    std::string name;
    unsigned age;
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
};
int main()
{
    Person person;
    // устанавливаем значения полей класса
    person.name = "Bob";
    person.age = 42;
    // получаем значения полей
    std::string username = person.name;
    unsigned userage = person.age;
    // выводим полученные данные на консоль
    std::cout << "Name: " << username << "\tAge: " << userage << std::endl;
}
```

Также можно поля класса, как и обычные переменные, инициализировать некоторыми начальными значениями:

```
#include <iostream>

class Person
{
public:
    std::string name{"Undefined"};
    unsigned age{18};
    void print()
```

```
9         {
10            std::cout << "Name: " << name << "\tAge: " << age << std::endl;
11        }
12    };
13    int main()
14    {
15        Person person;
16        person.print(); // Name: Undefined  Age: 18
17    }
```

Указатели на объекты классов

На объекты классов, как и на объекты других типов, можно определять указатели. Затем через указатель можно обращаться к членам класса - переменным и методам. Однако если при обращении через обычную переменную используется символ точка, то для обращения к членам класса через указатель применяется стрелка (->):

```
1     #include <iostream>
2
3     class Person
4     {
5     public:
6         std::string name;
7         unsigned age;
8         void print()
9         {
10            std::cout << "Name: " << name << "\tAge: " << age << std::endl;
11        }
12    };
13    int main()
14    {
15        Person person;
16        Person *ptr = &person;
17        // обращаемся к полям и функции объекта через указатель
18        ptr->name = "Tom";
19        ptr->age = 22;
20        ptr->print();
21        // обращаемся к полям объекта
22        std::cout << "Name: " << person.name << "\tAge: " << person.age << std::endl;
23    }
```

Изменения по указателю ptr в данном случае приведут к изменениям объекта person.

Конструкторы представляют специальную функцию, которая имеет то же имя, что и класс, которая не возвращает никакого значения и которая позволяют инициалилизировать объект класса во время го создания и таким образом

гарантировать, что поля класса будут иметь определенные значения. При каждом создании нового объекта класса вызывается конструктор класса.

В прошлой теме был разработан следующий класс:

```
#include <iostream>

class Person
{
public:
    std::string name;
    unsigned age;
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
};
int main()
{
    Person person;  // вызов конструктора
    person.name = "Tom";
    person.age = 22;
    person.print();
}
```

Здесь при создании объекта класса Person, который называется person

```
Person person;
```

вызывается **конструктор по умолчанию**. Если мы не определяем в классе явным образом конструктор, как в случае выше, то компилятор автоматически компилирует конструктор по умолчанию. Подобный конструктор не принимает никаких параметров и по сути ничего не делает.

Теперь определим свой конструктор. Например, в примере выше мы устанавливаем значения для полей класса Person. Но, допустим, мы хотим, чтобы при создании объекта эти поля уже имели некоторые значения по умолчанию. Для этой цели определим конструктор:

```
#include <iostream>

class Person
{
public:
    std::string name;
    unsigned age;
    void print()
    {
```

```
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    Person(std::string p_name, unsigned p_age)
    {
        name = p_name;
        age = p_age;
        std::cout << "Person has been created" << std::endl;
    }
};
int main()
{
    Person tom("Tom", 38);  // создаем объект - вызываем конструктор
    tom.print();
}
```

Теперь в классе Person определен конструктор:

```
Person(std::string p_name, unsigned p_age)
{
    name = p_name;
    age = p_age;
    std::cout << "Person has been created" << std::endl;
}
```

По сути конструктор представляет функцию, которая может принимать параметры и которая должна называться по имени класса. В данном случае конструктор принимает два параметра и передает их значения полям name и age, а затем выводит сообщение о создании объекта.

Если мы определяем свой конструктор, то компилятор больше не создает конструктор по умолчанию. И при создании объекта нам надо обязательно вызвать определенный нами конструктор.

Вызов конструктора получает значения для параметров и возвращает объект класса:

```
Person tom("Tom", 38);
```

После этого вызова у объекта person для поля name будет определено значение "Tom", а для поля age - значение 38. Вполедствии мы также сможем обращаться к этим полям и переустанавливать их значения.

В качестве альтернативы для создания объекта можно использовать инициализатор в фигурных скобках:

```
Person tom{"Tom", 38};
```

Тажке можно присвоить объекту результат вызова конструктора:

```
Person tom = Person("Tom", 38);
```

По сути она будет эквивалетна предыдущей.

Консольный вывод определенной выше программы:

```
Person has been created
Name: Tom       Age: 38
```

Конструкторы облегчают нам создание нескольких объектов, которые должны иметь разные значения:

```
#include <iostream>

class Person
{
public:
    std::string name;
    unsigned age;
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    Person(std::string p_name, unsigned p_age)
    {
        name = p_name;
        age = p_age;
        std::cout << "Person has been created" << std::endl;
    }
};
int main()
{
    Person tom{"Tom", 38};
    Person bob{"Bob", 42};
    Person sam{"Sam", 25};
    tom.print();
    bob.print();
    sam.print();
}
```

Здесь создаем три разных объекта класса Person (условно трех разных людей), и соответственно в данном случае консольный вывод будет следующим:

```
Person has been created
Person has been created
```

```
Person has been created
Name: Tom       Age: 38
Name: Bob       Age: 42
Name: Sam       Age: 25
```

Определение нескольких конструкторов

Подобным образом мы можем определить несколько конструкторов и затем их использовать:

```
#include <iostream>

class Person
{
    std::string name{};
    unsigned age{};
public:
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    Person(std::string p_name, unsigned p_age)
    {
        name = p_name;
        age = p_age;
    }
    Person(std::string p_name)
    {
        name = p_name;
        age = 18;
    }
    Person()
    {
        name = "Undefined";
        age = 18;
    }
};
int main()
{
    Person tom{"Tom", 38};  // вызываем конструктор Person(std::string p_name, u
    Person bob{"Bob"};      // вызываем конструктор Person(std::string p_name)
    Person sam;             // вызываем конструктор Person()
    tom.print();
    bob.print();
    sam.print();
```

```
}
```

В классе Person определено три конструктора, и в функции все эти конструкторы используются для создания объектов:

```
Name: Tom       Age: 38
Name: Bob       Age: 18
Name: Undefined Age: 18
```

Хотя пример выше прекрасно работает, однако мы можем заметить, что все три конструктора выполняют фактически одни и те же действия - устанавливают значения переменных name и age. И в C++ можем сократить их определения, вызова из одного конструктора другой и тем самым уменьшить объем кода:

```
#include <iostream>

class Person
{
    std::string name{};
    unsigned age{};
public:
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    Person(std::string p_name, unsigned p_age)
    {
        name = p_name;
        age = p_age;
        std::cout << "First constructor" << std::endl;
    }
    Person(std::string p_name): Person(p_name, 18) // вызов первого конструктора
    {
        std::cout << "Second constructor" << std::endl;
    }
    Person(): Person(std::string("Undefined")) // вызов второго конструктора
    {
        std::cout << "Third constructor" << std::endl;
    }
};
int main()
{
    Person sam;     // вызываем конструктор Person()
    sam.print();
}
```

Запись `Person(string p_name): Person(p_name, 18)` представляет вызов конструктора, которому передается значение параметра p_name и число 18. То есть второй конструктор делегирует действия по инициализации переменных первому конструктору. При этом второй конструктор может дополнительно определять какие-то свои действия.

Таким образом, следующее создание объекта

```
Person sam;
```

будет использовать третий конструктор, который в свою очередь вызывает второй конструктор, а тот обращается к первому конструктору.

Данная техника еще называется **делегированием конструктора**, поскольку мы делегируем инициализацию другому конструктору.

Параметры по умолчанию

Как и другие функции, конструкторы могут иметь параметры по умолчанию:

```
#include <iostream>

class Person
{
    std::string name;
    unsigned age;
public:
    // передаем значения по умолчанию
    Person(std::string p_name = "Undefined", unsigned p_age = 18)
    {
        name = p_name;
        age = p_age;
    }
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
};
int main()
{
    Person tom{"Tom", 38};
    Person bob{"Bob"};
    Person sam;
    tom.print();    // Name: Tom   Age: 38
    bob.print();    // Name: Bob   Age: 18
    sam.print();    // Name: Undefined   Age: 18
}
```

Инициализация констант и списки инициализации

В теле конструктора мы можем передать значения переменным класса. Однако константы требуют особого отношения. Например, вначале определим следующий класс:

```
class Person
{
    const std::string name;
    unsigned age{};
public:
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    Person(std::string p_name, unsigned p_age)
    {
        name = p_name;
        age = p_age;
    }
};
```

Этот класс не будет компилироваться из-за отсутствия инициализации константы name. Хотя ее значение устанавливается в конструкторе, но к моменту, когда инструкции из тела конструктора начнут выполняться, константы уже должны быть инициализированы. И для этого необходимо использовать **списки инициализации**:

```
#include <iostream>

class Person
{
    const std::string name;
    unsigned age{};
public:
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    Person(std::string p_name, unsigned p_age) : name{p_name}
    {
        age = p_age;
    }
};
int main()
{
    Person tom{"Tom", 38};
```

```
20      tom.print();    // Name: Tom    Age: 38
21  }
```

Списки инициализации представляют перечисления инициализаторов для каждой из переменных и констант через двоеточие после списка параметров конструктора:

```
1   Person(std::string p_name, unsigned p_age) : name{p_name}
```

Здесь выражение `name{p_name}` позволяет инициализировать константу значением параметра p_name. Здесь значение помещается в фигурные скобки, но также можно использовать кругулые:

```
1   Person(std::string p_name, unsigned p_age) : name(p_name)
```

Списки инициализации пободным образом можно использовать и для присвоения значений переменным:

```
1   class Person
2   {
3       const std::string name;
4       unsigned age;
5   public:
6       void print()
7       {
8           std::cout << "Name: " << name << "\tAge: " << age << std::endl;
9       }
10      Person(std::string p_name, unsigned p_age) : name(p_name), age(p_age)
11      { }
12  };
```

При использовании списков инициализации важно учитывать, что передача значений должна идти в том порядке, в котором константы и переменные определены в классе. То есть в данном случае в классе сначала определена константа name, а потом переменная age. Соответственно в таком же порядке идет передача им значений. Поэтому при добавлении дополнительных полей или изменения порядка существующих придется следить, чтобы все инициализировалось в надлежащем порядке.

Деструктор выполняет освобождение использованных объектом ресурсов и удаление нестатических переменных объекта. Деструктор автоматически вызывается, когда удаляется объект. Удаление объекта происходит в следующих случаях:

- когда завершается выполнение области видимости, внутри которой определены объекты
- когда удаляется контейнер (например, массив), который содержит объекты
- когда удаляется объект, в котором определены переменные, представляющие другие объекты
- динамически созданные объекты удаляются при применении к указателю на объект оператора **delete**

По сути деструктор - это функция, которая называется по имени класса (как и конструктор) и перед которой стоит тильда (~):

```
~имя_класса()
{
    // код деструктора
}
```

Деструктор не имеет возвращаемого значения и не принимает параметров. Каждый класс может иметь только один деструктор.

Обычно деструктор не так часто требуется и в основном используется для освобождения связанных ресурсов. Например, объект класса использует некоторый файл, и в деструкторе можно определить код закрытия файла. Или если в классе выделяется память с помощью оператора `new`, то в деструкторе можно освободить подобную память.

Сначала рассмотрим простейшее определение деструктора:

```
#include <iostream>

class Person
{
public:
    Person(std::string p_name)
    {
        name = p_name;
        std::cout << "Person " << name << " created" << std::endl;
    }
    ~Person()
    {
        std::cout << "Person " << name << " deleted" << std::endl;
    }
private:
    std::string name;
};

```

```
19  int main()
20  {
21      {
22          Person tom{"Tom"};
23          Person bob{"Bob"};
24      }   // объекты Tom и Bob уничтожаются
25
26      Person sam{"Sam"};
27  }   // объект Sam уничтожается
```

В классе Person определен деструктор, который просто уведомляет об уничтожении объекта:

```
1  ~Person()
2  {
3      std::cout << "Person " << name << " deleted" << std::endl;
4  }
```

В функции main создаются три объекта Person. Причем два из них создается во вложенном блоке кода.:

```
1  {
2      Person tom{"Tom"};
3      Person bob{"Bob"};
4  }
```

Этот блок кода задается границы области видимости, в которой существуют эти объекты. И когда выполнение блока завершается, то уничтожаются обе переменных, и для обоих объектов вызываются деструкторы.

После этого создается третий объект - sam

```
1  int main()
2  {
3      {
4          Person tom{"Tom"};
```

```
5           Person bob{"Bob"};
6       }   // объекты Tom и Bob уничтожаются
7
8       Person sam{"Sam"};
9   }   // объект Sam уничтожается
```

Поскольку он определен в контексте функции main, то и уничтожается при завершении этой функции. В итоге мы получим следующий консольный вывод:

```
Person Tom created
Person Bob created
Person Bob deleted
Person Tom deleted
Person Sam created
Person Sam deleted
```

Чуть более практический пример. Допустим, у нас есть счетчик объектов Person в виде статической переменной. И если в конструкторе при создании каждого нового объекта счетчик увеличивается, то в деструкторе мы можем уменьшать счетчик:

```
1   #include <iostream>
2
3   class Person
4   {
5   public:
6       Person(std::string p_name)
7       {
8           name = p_name;
9           ++count;
10          std::cout << "Person " << name << " created. Count: " << count << std::end
11      }
12      ~Person()
13      {
14          --count;
```

```
15            std::cout << "Person " << name << " deleted. Count: " << count << std::endl;
16        }
17    private:
18        std::string name;
19        static inline unsigned count{}; // счетчик объектов
20    };
21
22    int main()
23    {
24        {
25            Person tom{"Tom"};
26            Person bob{"Bob"};
27        }   // объекты Tom и Bob уничтожаются
28        Person sam{"Sam"};
29    }   // объект Sam уничтожается
```

Консольный вывод программы:

```
Person Tom created. Count: 1
Person Bob created. Count: 2
Person Bob deleted. Count: 1
Person Tom deleted. Count: 0
Person Sam created. Count: 1
Person Sam deleted. Count: 0
```

При этом выполнение самого деструктора еще не удаляет сам объект. Непосредственно удаление объекта производится в ходе явной фазы удаления, которая следует после выполнения деструктора.

Стоит также отметить, что для любого класса, который не определяет собственный деструктор, компилятор сам создает деструктор. Например, если бы класс Person не имел бы явно определенного деструктора, то для него автоматически создавался бы следующий деструктор:

```
1    ~Person(){}
```