Лекция 9

Классы и объекты. Синтаксис описания класса.

Классы и структуры в C++

- Обеспечивают механизм создания собственных типов и определения различных действий над ними.
- Обычно используются для описания различных понятий, фигурирующих в решаемой задаче.
- Акцентируют внимание разработчика на моделировании данных, а не действий.

Класс и объект

Класс — это сущность, которая задает общие свойства и общее поведение для объектов (общий шаблон для создания объектов). По сути класс является типом данных.

Объект — это сущность в адресном пространстве вычислительной системы, которая появляется при создании экземпляра класса и обладает определенным состоянием, уникальностью и поведением.

Основные элементы объектной модели

Концептуальной базой объектно-ориентированного стиля программирования является объектная модель, основывающаяся на 4-х главных принципах:

- Абстракция
- Инкапсуляция
- Модульность
- Иерархия

Абстракция

Выделяет существенные характеристики некоторого объекта, отличающие его от других видов объектов.

Определяет концептуальные границы объекта с точки зрения наблюдателя.

Инкапсуляция

Отделяет друг от друга элементы объекта, определяющие его устройство и поведения.

Изолирует внешний интерфейс (то, что нам нужно знать для работы с объектом) от внутренней реализации.

Иерархия

Позволяет упорядочить абстракции, сформировать уровни абстрагирования, определить способы взаимодействия абстракций, их отношения.

Модульность

Позволяет описать систему как набор компонентов с сильными внутренними связями и более слабыми внешними связями.

Уменьшает сложность системы.

Уровни модульности:

1. Файлы, каталоги…
2. Пространства имен, пакеты…

Класс: от требований к реализации

1. Определить свойства рассматриваемой сущности, важные для данной задачи
2. Определить основные действия
3. Определить, какой набор данных достаточен для описания этих свойств
4. Определить список функций (методов), соответствующих требуемым действиям

Модель данных

```
class Rational
{    int numer;      // Числитель
     int denom;    // Знаменатель (>=1)
public:
   Rational();
   Rational(int n, int d);
   int getNumer();
   int getDenom();      };
```

Конструкторы

Конструктор — это метод класса который всегда вызывается при создании экземпляра класса (объекта). Конструкторы предназначены для создания объектов класса, и для инициализации атрибутов объекта.

- Имя конструктора всегда совпадает с именем класса.
- Конструктор может принимать параметры, но никогда не возвращает значения.
- Класс может содержать несколько конструкторов (перегрузка конструкторов).
- Если в классе не объявлен конструктор, то компилятор предоставляет конструктор по умолчанию (стандартный конструктор).
- Стандартный конструктор не принимает параметров и не выполняет никаких действий.

Конструкторы

```
Rational::Rational()
{  std::cout << "Call the default constructor" << std::endl;
   numer = 0;
   denom = 1;    }
Rational::Rational(int n, int d)
{   std::cout << "Call constructor with parameters" << std::endl;
   numer = n;
   denom = d;   }
```

Деструкторы

Деструктор — это метод класса который предназначен для уничтожения экземпляров класса, а также для освобождения ресурсов используемых в объектах класса (например освобождение памяти).

- Деструктор не принимает параметров и не может возвращать значение.
- Класс может иметь только один деструктор.
- Имя деструктора начинается символом «~»

Деструкторы

- Деструкторы не могут перегружаться.
- Деструкторы невозможно вызвать, они вызываются автоматически.
- Если в классе не объявлен деструктор, то компилятор предоставит деструктор по умолчанию (стандартный деструктор). Стандартный деструктор не выполняет никаких действий.
- Деструктор всегда вызывается при выходе объекта за пределы области видимости

Деструкторы

```
Rational::~Rational()
{   std::cout << "Call destructor" << std::endl;   }


int main()
{   Rational r;
    Rational r2(2, 3);     }
```

Уровни доступа к членам класса

Для поддержки принципа инкапсуляции, существуют три основных уровня доступа к членам класса. Приведем их в порядке открытости для внешних абстракций:

- **Открытый(public) доступ** – члены с этим уровнем доступа видимы всем клиентам класса.
- **Защищенный(protected) доступ** – члены этого уровня видимы самому классу, его подклассам, и абстракциям.
- **Закрытый(private) доступ** – члены этого уровня видимы только изнутри самого класса.

“public”-наследование

Если класс объявлен как базовый для другого класса со спецификатором доступа «public»:

- «public»-члены базового класса — доступны как «public»-члены производного класса;
- «protected»-члены базового класса — доступны как «protected»-члены производного класса;

```
class A {}; // Базовый класс
class B : public A {}; // Public-наследование
```

“private”-наследование

«public»- и «protected»- члены базового класса – доступны как «private»-члены производного класса.

```
class Z : private A {}; // Private-наследование
```

“protected”-наследование

«public» и «protected» - члены базового класса - доступны как «protected»-члены производного класса;

```
class C : protected A {}; // Protected-наследование
```

Абстрактный класс

Абстрактный класс в ООП — базовый класс, который не предполагает создания экземпляров. Абстрактный класс может содержать абстрактные методы и свойства. Абстрактный метод не реализуется для класса, в котором описан, однако должен быть реализован для его неабстрактных потомков. Абстрактные классы представляют собой наиболее общие абстракции, то есть имеющие наибольший объём и наименьшее содержание.

```
class Account {
public:
   Account( double d );   // Constructor.
   virtual double GetBalance();   // Obtain balance.
   virtual void PrintBalance() = 0;   // Pure virtual function.
private:
    double _balance;
};
```

Ограничения на использование абстрактных классов

Абстрактные классы невозможно использовать для следующих элементов:

- переменных и данных членов;
- типов аргументов;
- типов возвращаемых функциями значений;
- типов явных преобразований.

Виртуальные методы

Виртуальный метод (виртуальная функция) — в ООП метод (функция) класса, который может быть переопределён в классах-наследниках так, что конкретная реализация метода для вызова будет определяться во время исполнения.

Таким образом, программисту необязательно знать точный тип объекта для работы с ним через виртуальные методы: достаточно лишь знать, что объект принадлежит классу или наследнику класса, в котором метод объявлен.

Виртуальные методы

Виртуальные методы позволяют создавать общий код, который может работать как с объектами базового класса, так и с объектами любого его класса-наследника. При этом базовый класс определяет способ работы с объектами и любые его наследники могут предоставлять конкретную реализацию этого способа.

Виртуальный деструктор

Если деструктор объявлен как виртуальный, то при вызове его через указатель на объект базового класса (через **delete**) будет вызван вначале деструктор производного класса, а затем деструктор базового класса.

Приведение типов: const_cast

Снимает cv qualifiers — const и volatile, то есть константность и отказ от оптимизации компилятором переменной. Это преобразование проверяется на уровне компиляции и в случае ошибки приведения типов будет выдано сообщение.

Синтаксис:

TYPE const_cast<TYPE> (object);

Приведение типов: static_cast

Преобразует выражения одного статического типа в объекты и значения другого статического типа. Поддерживается преобразование численных типов, указателей и ссылок по иерархии наследования как вверх, так и вниз. Проверка производится на уровне компиляции, так что в случае ошибки сообщение будет получено в момент сборки приложения или библиотеки.

Синтаксис:

```
TYPE static_cast<TYPE> (object);
```

Приведение типов: dynamic_cast

Используется для динамического приведения типов во время выполнения. В случае неправильного приведения типов для ссылок вызывается исключительная ситуация std::bad_cast, а для указателей будет возвращен 0. Использует систему RTTI (Runtime Type Information). Безопасное приведение типов по иерархии наследования, в том числе для виртуального наследования.

Синтаксис:

TYPE& dynamic_cast<TYPE&> (object);

TYPE* dynamic_cast<TYPE*> (object);

Приведение типов: reinterpret_cast

Приведение типов без проверки. reinterpret_cast — непосредственное указание компилятору. Применяется только в случае полной уверенности программиста в собственных действиях. Не снимает константность и volatile. Применяется для приведения указателя к указателю, указателя к целому и наоборот.

Синтаксис:

TYPE reinterpret_cast<TYPE> (object);