Лекция 4.
Классы

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.

Классы

- 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-ориетированного 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:
  - Файлы, каталоги…
  - Пространства имен, пакеты…

Класс: от требований к реализации

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

Модель данных

```
class Rational
{
    // Числитель
    int numer;
    // Знаменатель (>=1)
    int denom;

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 предоставит 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;
}

...


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

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 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.

Открытый(public) доступ

```
class Human
{
public:
    int Age;
    string Name;
};

int main()
{
    Human human1;
    human1.Age = 18;
    human1.Name = "Tom";

    cout << "Name: " << human1.Name << endl;
    cout << "Age: " << human1.Age << endl;

    return 0;
}
```

Защищенный(protected) доступ

```
class Human
{
protected:
    int Age;
    string Name;

public:
    void setAge(int age);
    int getAge();
    void setName(string Name);
    string getName();
};

void Human::setAge(int age){ Age = age; }
int Human::getAge() { return Age; }

void Human::setName(string name){ Name = name; }
string Human::getName() { return Name; }
```

Защищенный(protected) доступ

```
int main()
{
    Human human1;
    human1.setAge(18);
    human1.setName("Tom");

    cout << "Name: " << human1.getName() << endl;
    cout << "Age: " << human1.getAge() << endl;

    return 0;
}
```

Закрытый(private) доступ

```
class Human
{
private:
    int Age;
    string Name;

public:
    void setAge(int age);
    int getAge();
    void setName(string Name);
    string getName();
};

void Human::setAge(int age){ Age = age; }
int Human::getAge() { return Age; }

void Human::setName(string name){ Name = name; }
string Human::getName() { return Name; }
```

Конструктор копирования

```
class Human
{
public:
    int Age;
    char* Name;

    Human(int age, const char* name);
    ~Human();
};

Human::Human(int age, const char* name)
{
    cout << "constructor" << endl;
    Age = age;
    if (name != NULL)
    {
        Name = new char [strlen(name) + 1];
        strcpy(Name, name);
    }
    else
        Name = NULL;

}

Human::~Human()
{
    cout << "~destructor" << endl;
    if (Name != NULL)
        delete [] Name;
 }

void showFunc(Human object)
{
  cout << "Function takes a parameter" << endl;
}
```

Конструктор копирования

```
int main()
{
    Human human1(18, "Tom");
    showFunc(human1);

    return 0;
}
```

Конструктор копирования

```
// Конструктор копий
Human::Human(const Human& CopySource)
{
    cout << "Copy constructor: copying from MyString" << endl;
    if(CopySource.Name != NULL)
    {
        Name = new char [strlen(CopySource.Name) + 1];
        strcpy(Name, CopySource.Name);
    }
    else
        Name = NULL;
}
```

“public”-наследование

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

“public”-наследование

```
class A
{
  private: int x;
  public: int y;
  protected: int z;
};

class B: public A
{
    public:
       void update()
          {
            x=100; //Будет ошибка
            y=200; //y из класса B стало равным 200
            z=300; //z из класса B стало равным 300
          }
       void show()
       {
            cout << "y: " << y << endl;
            cout << "z: " << z;
       }

};
```

“private”-наследование

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

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

“private”-наследование

```
class A
{
  private: int x;
  public: int y;
  protected: int z;
};

class B: private A
{
    public:
       void update()
          {
            //x=100; //Будет ошибка
            y=200; //y из класса B стало равным 100
            z=300; //z из класса B стало равным 100
          }
};

class C: public B
{
    public:
        void update()
          {
            //Нельзя напрямую обратиться к private
            //элементу унаследованному от родителя
            //x=300;
            //y=300;
            //z=300;
          }
};
```

“protected”-наследование

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

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

“protected”-наследование

```
class A
{
  private: int x;
  public: int y;
  protected: int z;
};

class B: protected A
{
    public:
       void update()
          {
            //x=100; //Будет ошибка
            y=200; //y из класса B стало равным 100
            z=300; //z из класса B стало равным 100
          }
};

class C: public B
{
    public:
        void update()
          {
            //Нельзя напрямую обратиться к private
            //элементу унаследованному от родителя
            //x=300;

            // protected
            y=300;
            z=300;
          }
};
```

Public наследование	Private наследование	Protected наследование

Class A
{
private: int x;
public: int y;
protected: int z;
};
	Class A
	{
	private: int x;
	public: int y;
	protected: int z;
	};
		Class A
		{
		private: int x;
		public: int y;
		protected: int z;
		};

Class B
{
~~private: int x;~~
public: int y;
protected: int z;
};
	Class B
	{
	~~private: int x;~~
	private: int y;
	private: int z;
	};
		Class B
		{
		~~private: int x;~~
		protected: int y;
		protected: int z;
		};

Class C
{
~~private: int x;~~
public: int y;
protected: int z;
};
	Class C
	{
	~~private: int x;~~
	~~private: int y;~~
	~~private: int z;~~
	};
		Class C
		{
		~~private: int x;~~
		protected: int y;
		protected: int z;
		};

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.

Абстрактный класс

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

Абстрактный класс

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.

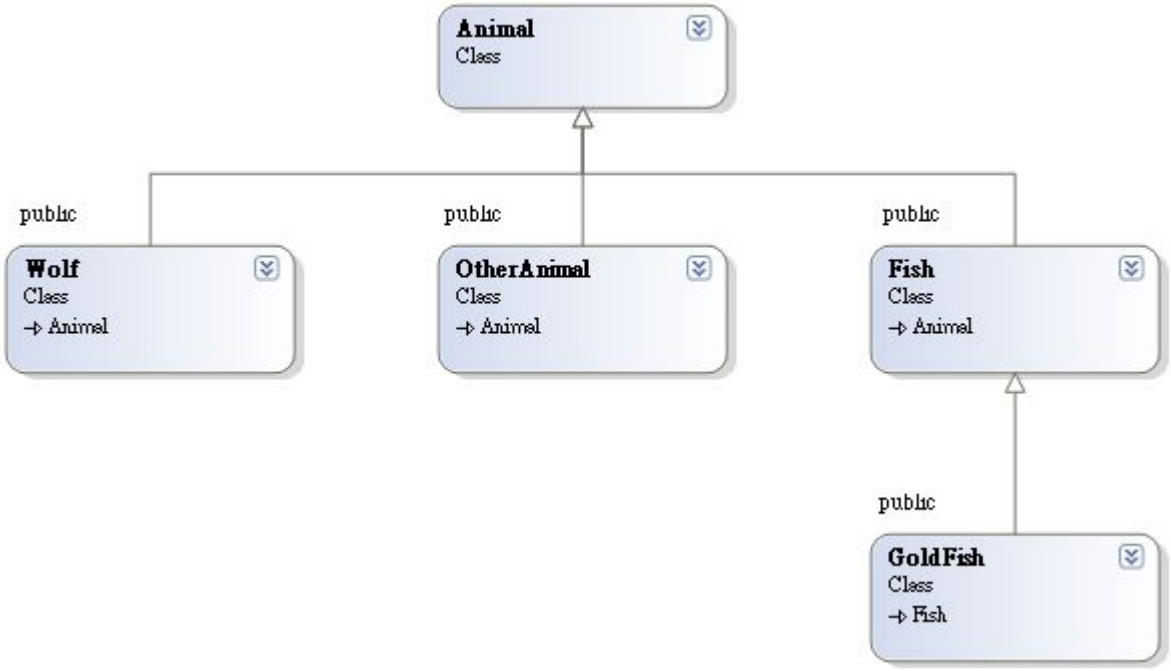
Виртуальные методы

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.

Виртуальные методы



Виртуальные методы

```
class Animal {
public:
    void move() {
        std::cout << "This animal moves in some way" << std::endl;
    }
    virtual void eat() {
        std::cout << "Animal ate something!" << std::endl;
    }
    virtual ~Animal(){} // деструктор
};

class Wolf : public Animal {
public:
    void move() {
        std::cout << "Wolf walks" << std::endl;
    }
    void eat(void) { // метод eat переопределён и тоже является виртуальным
        std::cout << "Wolf eats!" << std::endl;
    }
};

class Fish : public Animal {
public:
    void move() {
        std::cout << "Fish walks" << std::endl;
    }
    void eat(void) { // метод eat переопределён и тоже является виртуальным
        std::cout << "Fish swims!" << std::endl;
    }
};
```

Виртуальные методы

```
int main(int argc, char *argv[])
{
    Animal* zoo[] = {new Wolf(), new Animal(), new Fish()};
    for (int index = 0; index < 3; index++)
    {
        zoo[index]->move();
        zoo[index]->eat();
    }
    return 0;
}
```

```
This animal moves in some way
Wolf eats!
This animal moves in some way
Animal ate something!
This animal moves in some way
Fish eats!
```

Виртуальные методы

```
class Animal {
public:
    void virtual move() {
        std::cout << "This animal moves in some way" << std::endl;
    }
    virtual void eat() {
        std::cout << "Animal ate something!" << std::endl;
    }
    virtual ~Animal(){} // деструктор
};
```

```
Wolf walks
Wolf eats!
This animal moves in some way
Animal ate something!
Fish swims
Fish eats!
```

Виртуальный деструктор

```
class Animal {
public:
    void virtual move() {
        std::cout << "This animal moves in some way" << std::endl;
    }
    virtual void eat() {
        std::cout << "Animal ate something!" << std::endl;
    }
    virtual ~Animal(){ std::cout << "Animal destructor" << std::endl; } // деструктор
};

class Wolf : public Animal {
public:
    void move() {
        std::cout << "Wolf walks" << std::endl;
    }
    void eat(void) { // метод eat переопределён и тоже является виртуальным
        std::cout << "Wolf eats!" << std::endl;
    }

    ~Wolf(){ std::cout << "Wolf destructor" << std::endl; }
};

class Fish : public Animal {
public:
    void move() {
        std::cout << "Fish swims" << std::endl;
    }
    void eat(void) { // метод eat переопределён и тоже является виртуальным
        std::cout << "Fish eats!" << std::endl;
    }
    ~Fish(){ std::cout << "Fish destructor" << std::endl; }
};
```

Виртуальный деструктор

```
int main(int argc, char *argv[])
{
    Animal* zoo[] = {new Wolf(), new Animal(), new Fish()};
    for (int index = 0; index < 3; index++)
    {
        zoo[index]->move();
        zoo[index]->eat();
        delete zoo[index];
    }
    return 0;
}
```

```
Wolf walks
Wolf eats!
Wolf destructor
Animal destructor
This animal moves in some way
Animal ate something!
Animal destructor
Fish swims
Fish eats!
Fish destructor
Animal destructor
```

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
const_cast

```
int main()
{
    const int i = 5;
    const int * pi = &i;
    // *pi имеет тип const int,
    // но pi указывает на int, который константным не является
    int* j = const_cast<int *> (pi);
    cout << "j = " << *j << endl;
    *j = 10;
    cout << "j = " << *j << endl;
    return 0;
}
```

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);