Лекция 3

Основы ООП

Объектно-ориентированное программирование

- **Объектно-ориентированное** программирование (ООП) — парадигма программирования, в которой основными концепциями являются понятия объектов и классов.
- ООП возникло в результате развития методологии процедурного программирования, где данные и подпрограммы (процедуры, функции) их обработки формально не связаны.

Объектно-ориентированное программирование

- **Класс** – разновидность абстрактного типа данных в объектно-ориентированном программировании, характеризуемый способом своего построения.

Класс может включать в себя:

1. **Свойство/Поле** – это данные, хранимые внутри объекта.
2. **Метод** – это функция или процедура, являющиеся членом класса.
3. **Специальный метод** – конструкторы, деструктор, конструктор копирования, оператор присваивания копированием и т.д..

Класс/Свойство/Метод

```
// Пример human.hpp
#ifndef HUMAN_HPP
#define HUMAN_HPP
#include <string>

// Начало определения класса
struct human {
  // ниже открытые свойства(данные)
  std::string Name;         // имя
  std::string MiddleName;   // отчество
  size_t Age;               // возраст

public: // <- модификатор доступа
  // ниже открытые методы
  void IntroduceSelf();           // представится
  void Talk(std::string say);     // сказать
  human(std::string, std::string, size_t);// конструктор
  human() {} // конструктор по умолчанию
  ~human();  // деструктор

};
// Конец определения класса
#endif
```

Класс/Свойство/Метод

```cpp
// Пример human.cpp
#include "human.hpp"
#include <iostream>

// Определение метода IntroduceSelf
void human::IntroduceSelf() {
  std::cout << "Меня зовут " << this->Name << " "
  << MiddleName << " и мне " << Age << std::endl;
}

// Определение метода Talk
void human::Talk(std::string say) {
  std::cout << this->Name << " " << this->MiddleName
      << " : " << say << std::endl;
}
```

Специальные методы

- **Конструктор** – это специальный метод, вызываемый при создания объекта; Конструктор носит имя класса и не имеет возвращаемого значения; конструктор можно перегружать;
- **Конструктор по умолчанию** – это конструктор, который не принимает параметры;
- **Деструктор** – это специальный метод, но вызываемый при уничтожении объекта; у деструктора перед именем класса стоит символ '~' и он не принимает параметров.

Конструктор/Деструктор

```
// продолжение файла human.cpp
// способ 1
human::human(std::string _Name, std::string _mName, size_t _Age)
  : Name(_Name), MiddleName(_mName), Age(_Age) {
  // тело конструктора
  std::cout << "Конструктор human" << std::endl;
}
// способ 2
human::human(std::string Name, std::string mName, size_t Age) {
  this->Name = Name;
  MiddleName = mName;
  this->Age = Age;
  std::cout << "Конструктор human" << std::endl;
}
human::~human() {
  std::cout << "Деструктор ~human" << std::endl;
}
```

Объект

- **Объект** в программировании – некоторая сущность, которая объединяет в себе как описывающие его данные (свойства), так и средства обработки этих данных (методы).

Объект является представлением (*экземпляром)* конкретного класса. Над классом действий производить нельзя, а над объектном можно.

Объект

```
// Пример main_obj.cpp
#include "human.hpp"

int main() {
   // hman – объект класса human
   human hman;
   hman.Name = "Артём";
   hman.MiddleName = "Анатольевич";
   hman.Age = 22;
   hman.IntroduceSelf();
   hman.Talk("Привет");
   return 0;
}
```

Объект

```
// Пример main_ptr.cpp
#include "human.hpp"

int main() {
   // hman – указатель на объект класса human
   human* hman = new human();
   hman->Name = "Артём";
   hman->MiddleName = "Анатольевич";
   (*hman).Age = 22;
   hman->IntroduceSelf();
   hman->Talk("Привет");
   delete hman;
   return 0;
}
```

Принципы ООП

- Абстракция
- Инкапсуляция
- Наследование
- Полиморфизм

Абстракция

- **Абстракция** – в ООП это придание объекту характеристик, которые отличают его от всех других объектов, четко определяя его концептуальные границы.

Инкапсуляция

- **Инкапсуляция** – свойство языка программирования, позволяющее пользователю не задумываться о сложности определения используемого программного компонента, а взаимодействовать с ним посредством предоставляемого интерфейса, а также объединить и защитить жизненно важные для компонента данные. При этом пользователю предоставляется только спецификация (интерфейс) объекта.

Наследование

- **Наследование** – один из четырёх важнейших механизмов объектно-ориентированного программирования, позволяющий описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса заимствуются новым классом (дочерним).

Полиморфизм

Полиморфизм – возможность объектов с одинаковой спецификацией иметь различную реализацию. Язык программирования поддерживает полиморфизм, если классы с одинаковой спецификацией могут иметь различную реализацию — например, реализация класса может быть изменена в процессе наследования.

Абстракция

- **Абстракция** – в ООП это придание объекту характеристик, которые отличают его от всех других объектов, четко определяя его концептуальные границы.

Абстракция

```
// Пример human.hpp
#ifndef HUMAN_HPP
#define HUMAN_HPP
#include <string>

// Начало определения класса
struct human {
  // ниже открытые свойства(данные)
  std::string Name;         // имя
  std::string MiddleName;   // отчество
  size_t Age;               // возраст
  …
public:
  // ниже открытые методы
  void IntroduceSelf();           // представится
  void Talk(std::string say);     // сказать
  …
};
// Конец определения класса
#endif
```

Открытые(публичные) свойства

Достоинством публичных свойств является меньшее количество строк кода при создания определения класса.

К **недостаткам** же можно отнести:

- Нет контроля за изменением данных (свойств);
- Возрастает вероятность нарушить целостность данных;

Инкапсуляция

- **Инкапсуляция** – свойство языка программирования, позволяющее пользователю не задумываться о сложности реализации используемого программного компонента, а взаимодействовать с ним посредством предоставляемого интерфейса, а также объединить и защитить жизненно важные для компонента данные. При этом пользователю предоставляется только спецификация (интерфейс) объекта.

Модификаторы доступа

Модификаторы доступа позволяют разработчику класса решать, что в классе может быть доступно извне его.

- **public** (открытый) – допускается доступ к элементам класса извне объекта класса;
- **protected** (защищенный) – допускается доступ к элементам класса только производному классу и друзьям класса;
- **private** (закрытый) – допускается доступ только методам данного класса и друзьям класса;

Ключевое слово **struct** по умолчанию задает всем элементам модификатор доступа **public,** ключевое слово **class** задает для всех элементам модификатор доступа **private**.

Модификаторы доступа

```
// Пример human.hpp
#ifndef HUMAN_HPP
#define HUMAN_HPP
#include <string>

// Начало определения класса
class human {
private : // <- модификатор доступа
  // ниже закрытые свойства(данные)
  std::string Name;            // имя
  std::string MiddleName;      // отчество
  size_t Age;                  // возраст

public : // <- модификатор доступа
  // ниже открытые методы
  void IntroduceSelf();                  // представится
  void Talk(std::string say);            // сказать
  human(std::string, std::string, size_t);// конструктор
  human() {} // конструктор по умолчанию
  ~human();  // деструктор
};
// Конец определения класса
#endif
```

Модификаторы доступа

```
// Пример main.cpp
#include "human.hpp"

int main() {
   human hman("Артём","Анатольевич",22);
   //std::cout << hman.Name; // ошибка
   hman.IntroduceSelf();
   hman.Talk("Привет");
   return 0;
}
```

Сеттеры/геттеры

- **Сеттер** – метод позволяющий изменять свойство объекта;
- **Геттер** – метод позволяющий получить свойство объекта;

Сеттеры/геттеры

```
// Пример human.hpp
#ifndef HUMAN_HPP
#define HUMAN_HPP
#include <string>

class human {
private:  // ниже закрытые данные
  std::string Name;             // имя
  std::string MiddleName;       // отчество
  size_t Age;                   // возраст
  // ...
public: // ниже методы
  // поведение, конструктор и деструктор
  // сеттеры
  void setName(std::string);
  void setMiddleName(std::string);
  void setAge(size_t);
  // геттеры
  std::string getName() const { return Name; }
  inline std::string getMiddleName() const { return MiddleName; }
  inline size_t getAge() const { return Age; }
  // ...
};
#endif
```

Сеттеры/геттеры

```
// Пример human.cpp
#include "human.hpp"

void human::setName(std::string Name) {
  this->Name = Name;
}

void human::setMiddleName(std::string mName) {
  MiddleName = mName;
}

void human::setAge(size_t age) {
  Age = age;
}
```

Сеттеры/геттеры

```
// Пример main.cpp
#include "human.hpp"

int main() {
   human hman("Артём","Анатольевич",22);
   std::cout << hman.getName() << std::endl;
   hman.IntroduceSelf();
   hman.Talk("Привет");
   return 0;
}
```

Закрытые свойства с использованием сеттеров/геттеров

Достоинства:

- Уменьшение количества случайных ошибок при написание кода, за счет увеличения инкапсуляции;
- Можно осуществлять какие-нибудь дополнительные операции (например: проверка диапазона);

Недостатки:

- Появляются небольшие накладные расходы, в следствие использования методов;

Дружественная функция/процедура

Дружественная функция — это функция, которая не является членом класса, но имеет доступ к членам класса, объявленным с помощью модификатора доступа **private** или **protected**.

Дружественная функция

```
// файл human.hpp
#ifndef HUMAN_HPP
#define HUMAN_HPP
#include <string>

class human {
private:
  // ниже закрытые данные
  std::string Name;         // имя
  std::string MiddleName;   // отчество
  size_t Age;               // возраст
public :
  // ниже открытые методы
  // методы, конструктор/деструктор, сеттеры/геттеры
  // объявление дружественной процедуры
  friend void showData(const human & h);
};
#endif
```

Дружественная функция

```cpp
// Пример human.cpp
#include "human.hpp"

// определение дружественной процедура
void showData(const human & h) {
  std::cout << "Имя: " << h.Name << std::endl;
  std::cout << "Отчество: " << h.MiddleName << std::endl;
  std::cout << "Возраст: " << h.Age << std::endl;
}

// Пример main.cpp
#include "human.hpp"

int main() {
  human hman("Артём","Анатольевич",22);
  hman.IntroduceSelf();
  hman.Talk("Привет");
  // вызов дружественной процедуры
  showData(hman);
  return 0;
}
```

Дружественная функция

Достоинства:

- Позволяет получить доступ к protected и private данным класса;
- Иногда можно уменьшить количество строк кода;

Недостатки:

- Нарушает инкапсуляцию, из-за чего является плохим тоном;

Особенность работы компилятора с классами

Компилятор по умолчанию генерирует следующие специальные методы для класса:

- Конструктор по умолчанию (если других конструкторов нет);
- Конструктор копирования;
- Оператор присваивания копированием;
- Деструктор;
- Конструктор перемещения (C++11);
- Оператор присваивания перемещением (C++11);

Класс Pen

```
// Пример pen.h
#ifndef PEN_H
#define PEN_H

#include <iostream>
#include <string>

class Pen {
private:
  std::string color_;
  size_t size_;
public:
  void setData(const std::string & color, size_t size)
  { color_ = color; size_ = size; }
  void show() const
  { std::cout << color_ << ' ' << size_ << std::endl; }
  // ...
};
#endif
```

main

```
// Пример main.cpp
#include "pen.h"

int main()
{
  Pen p1 = Pen();    // Конструктор по умолчанию
  p1.setData("red", 5);
  Pen p2{}, p3 = {};// Конструктор по умолчанию C++11
  p2.setData("yellow", 10);
  Pen copyPen = p1; // Конструктор копирования
  copyPen.show();    // red 5
  copyPen = p2;      // Оператор присваивания копированием
  copyPen.show();    // yellow 10
  return 0;
}
```

Реализация компилятором по умолчанию

```
class Pen {
  // ...
public:
  // Конструктор по умолчанию
  Pen()
    : color_(std::string()), size_(size_t()) { }
  // Конструктор копирования
  Pen(const Pen & rhs)
    : color_(rhs.color_), size_(rhs.size_) { }
  // Оператор присваивания копированием
  Pen & operator=(const Pen & rhs)
  {
    if (this != &rhs)
    {
      color_ = rhs.color_;
      size_ = rhs.size_;
    }
    return *this;
  }
  // Деструктор
  ~Pen() { }
};
```

Правило трёх (C++98)

Правило трёх (также известное как «Закон Большой Тройки» или «Большая Тройка») — правило в C++, гласящее, что если класс или структура определяет один из следующих методов, то они должны явным образом определить все три метода:

- Деструктор;
- Конструктор копирования;
- Оператор копирующего присваивания;

Правило пяти (C++11)

Теперь при реализации конструктора необходимо реализовать:

- Деструктор;
- Конструктор копирования;
- Оператор копирующего присваивания;
- Конструктор перемещения;
- Оператор перемещающего присваивания;

Правило нуля (C++11)

Мартин Фернандес предложил также правило нуля. По этому правилу не стоит определять ни одну из пяти функций самому; надо поручить их создание компилятору (присвоить им значение = default;). Для владения ресурсами вместо простых указателей стоит использовать специальные классы-обёртки, такие как std::unique_ptr и std::shared_ptr.

Список литературы

1. Основные принципы ООП
2. SOLID
3. Встроенные функции
4. Дружественные функции
5. Дружественные классы
6. Б.Страуструп. Язык программирования C++