Объектно-ориентированная парадигма

Основные элементы объектной модели

Объект – физическая или умозрительная сущность, облегчающая понимание реального мира и имеющее чётко определённое функциональное назначение в данной предметной области. Объект моделирует часть окружающей действительности и таким образом существует во времени и пространстве.

Класс – описание структуры объекта и его интерфейса.

Инкапсуляция – разделение интерфейса объекта и его реализации.

Наследование – это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других (множественное наследование) классов.

Полиморфизм – возможность одним и тем же способом взаимодействовать с объектами различных типов. При этом достигаемый результат будет зависеть от типа объекта, к которому производится обращение.

Классы в языке C++

Управление доступом к членам классов в языке C++

public – доступны везде, где виден объект.

protected – доступны только методам класса и методам классов-наследников.

private – доступны только методам класса.

Объявление и определение

```
class <имя> {
   [ private: ]
     <описание скрытых элементов>
   public:
     <описание доступных элементов>
};
// Описание заканчивается точкой
// с запятой
```

Объявление и определение

```
class Monster {
  int hitPoints, armorClass;
public:
  Monster(int hp = 100, int ac = 10) {
    hitPoints = hp;
    armorClass = ac;
  }
  void Draw(int x, int y,
            int scale, int position);
  int GetHitPoints() {
    return hitPoints;
  }
  int GetArmorClass() {
    return armorClass;
  }
};
```

Объявление и определение

```
void Monster::Draw(int x, int y,
                   int scale, int position) {
  /* тело метода */
}
```

Объявление и определение

```
Monster owlbear;

Monster demilich(80, 20);

vector<Monster> horde;
```

Объявление и определение

```
class Monster {

  ...

  int GetSpeed() const {
    return speed;
  }
};

...

const Monster gelatinousCube(84, 6, 15);
cout << gelatinousCube.GetSpeed() << endl;
```

Ссылка на себя. Ключевое слово this

```
class Monster {
   ...
   Monster& ChooseTheHealthiest(
                          Monster& anotherMonster) {
     if (hitPoints > anotherMonster.hitPoints) {
       return *this;
     }
     return anotherMonster;
   }
};

...
Monster theHealthiest =
        demilich.ChooseTheHealthiest(owlbear);
```

Создание и удаление объектов. Конструкторы и деструкторы

```
// Список параметров не должен быть пустым
имя_класса имя_объекта [(список параметров)];

// Создается объект без имени (список может быть пустым)
имя_класса (список параметров);

// Создается объект без имени и копируется
имя_класса имя_объекта = выражение;
```

Создание и удаление объектов. Конструкторы и деструкторы

```
// Список параметров не должен быть пустым
имя_класса имя_объекта [(список параметров)];

// Создается объект без имени (список может быть пустым)
имя_класса (список параметров);

// Создается объект без имени и копируется
имя_класса имя_объекта = выражение;

Monster owlbear;

Monster demilich = Monster(80, 20);

Monster gelatinousCube = 84;
```

Создание и удаление объектов. Конструкторы и деструкторы

```
enum Type { undead, monstrosity, ooze };

class Monster {
  int hitPoints, armorClass;
  Type monsterType;
  string monsterName;

public:
  Monster(Type type);
  Monster(string name);
  ...
  Monster(int hp = 100, int ac = 10) :
        hitPoints(hp), armorClass(ac),
        monsterType(monstrosity), monsterName("none") {}
 ...
};
```

Конструктор копирования

```
T::T(const T&) { ... / * Тело конструктора ... }

class Monster {
   ...
   Monster(const Monster& anotherMonster) {
     cout << "Copy Constructor" << endl;
   }
   ...
};
```

Конструктор копирования

```
T::T(const T&) { ... / * Тело конструктора ... }

class Monster {
   ...
   Monster(const Monster& anotherMonster) {
     cout << "Copy Constructor" << endl;
   }
   ...
};

Monster shadowDragon = dragon;
Monster blackDragon(dragon);
```

Статические члены класса

```
class Monster {
  static int counter;
public:
  Monster() {
    ++counter;
  }
  ~Monster() {
    --counter;
  }
  static int CountMonsters() {
    return counter;
  }
};
```

Статические члены класса

```
class Monster {
  static int counter;
public:
  Monster() {
    ++counter;
  }
  ~Monster() {
    --counter;
  }
  static int CountMonsters() {
    return counter;
  }
};

int Monster::counter = 0;

int main() {
      cout << Monster::CountMonsters() << endl;
}
```

Дружественные функции и классы

```
class Monster;

class Hero {
public:
  void Attack(Monster& monster);
};

class Monster {
  ...
  friend void Hero::Attack(Monster&);
  ...
};

void Hero::Attack(Monster& monster) {
  monster.hitPoints = 0;
}
```

Перегрузка операций

```
тип operator операция ( список параметров)
   { тело функции }
```

Перегрузка операций

```
тип operator операция ( список параметров)
   { тело функции }

class Monster {
   ...
   Monster& operator ++() {
     ++hitPoints;
     return *this;
   }
   ...
};
```

Перегрузка операций

```
тип operator операция ( список параметров)
   { тело функции }

class Monster {
   ...
   friend Monster& operator ++(Monster& monster);
   ...
};

Monster& operator ++(Monster& monster) {
      ++monster.hitPoints;
      return monster;
}
```

Перегрузка операций

```
тип operator операция ( список параметров)
   { тело функции }

class Monster {
   ...
   bool operator >(const Monster& monster){
     return (hitPoints > monster.hitPoints);
   }
   ...
};
```

Перегрузка операций

```
тип operator операция ( список параметров)
  { тело функции }

bool operator >(const Monster& firstMonster,
                const Monster& secondMonster){
  return (firstMonster.GetHitPoints() >
          secondMonster.GetHitPoints());
}
```