ООП в C++

Класс – тип данных, содержащий поля (переменные) и методы (функции) для их обработки.

Объект – переменная типа класс.

Классы – это лишь описание того, какими свойствами и поведением будут обладать объекты.

Объекты являются экземплярами класса с собственным состоянием свойств.

Пример:

- Треугольник описывает класс объектов, обладающих тремя сторонами и тремя углами.
- Равносторонний треугольник с длиной стороны в 5 см является экземпляром с конкретными свойствами.

Описание класса

```
class <имя> {
    [private:]
     <описание скрытых элементов>
   public:
     <описание доступных
  элементов>
};
```

Элементы класса – поля и методы.

Основные принципы ООП:

- Абстракция данных – выделение наиболее значимых характеристик объектов и объединение их в класс;

- Инкапсуляция – сокрытие данных и отдельных методов, необходимых лишь внутреннему состоянию описываемого класса объектов;

- Наследование – частичное или полное заимствование функциональности уже существующего класса при описании нового;

-Полиморфизм – использование объектов с одинаковым интерфейсом без информации о конкретном типе и структуре объектов.

Абстракция

Пример.

```
class Cat
{
public:
    int32_t age;
    double weight;
    float length;
};

int main()
{
    Cat chonkCat;
    chonkCat.age = 10;
    chonkCat.weight = 8.0;
    chonkCat.length = 52.3f;
}
```

Абстракция

Класс Cat объединяет в себе основные свойства типичной кошки или же кота: возраст, вес и длину.

В функции main() создается экземпляр класса Cat, описывающий коренастого кота с определенными свойствами.

Все это позволяет нам создавать объекты, свойства которых будут различны, но при этом выполнять какие-либо универсальные операции над ними в будущем. Таким образом, вводится абстракция над данными, объединяя их в обобщенный класс.

Пример является ***не самым лучшим***, потому что здесь не используется инкапсуляция данных.

В данной ситуации можно было использовать и простую структуру. Однако это не отменяло бы наличие абстрагирования данных.

Инкапсуляция

Применим к имеющемуся коду инкапсуляцию.

```
class Cat
{
public:
    Cat(int age, double weight, float length)
      : age(age), weight(weight), length(length) { }
    int getHumanAge() { return age * 7; }
protected:
    int age;
    double weight;
    float length;
};
int main()
{
    Cat chonkCat(10, 8.0, 52.3f);
    std::cout << "chonkCat human age: "
      << chonkCat.getHumanAge() << std::endl;
}
```

Инкапсуляция

В данном примере в определенной степени исправлены недостатки предыдущей абстракции. Теперь здесь выполняется сокрытие данных.

В функции main() вновь создается экземпляр класса Cat, описывающий коренастого кота, однако теперь используется конструктор класса, который позволяет проинициализировать объект.

Основным проявлением инкапсуляции здесь является использование функции getHumanAge(). Это функция использует данные, сокрытые в объекте, для вычисления человеческого аналога возраста кота, который их возвращает. Таким образом, взаимодействия напрямую с полем класса age нет, но при этом остается возможным получить необходимую нам информацию.

Данный наш конструктор просто копирует значения необходимых полей внутрь объекта. Однако в реальной жизни конструкторы чаще всего не просто копируют данные, а используют входные параметры для осуществления внутренней инициализации и сохранения только нужной информации, инкапсулированной внутри класса.

Наследование

```
class Shape
{
public:
    Shape(int v, int s)
      : vertices(v), sides(s) { }
    int getNumberOfVertices() { return vertices; }
    int getNumberOfSides() { return sides; }
private:
    int vertices;
    int sides;
};
class Triangle : public Shape
{
public:
    Triangle(std::array<float, 3> angles)
      : Shape(3, 3), angles(angles) {}
private:
    std::array<float, 3> angles;

};
```

Наследование

```
class Circle : public Shape
{
public:
    Circle(float radius)
      : Shape(0, 0), radius(radius) {}
private:
    float radius;
};
```

Класс Shape описывает геометрические фигуры.
Для простоты он содержит только количество вершин

и количество сторон геометрической фигуры.

Наследование

На основе этого класса, благодаря наследованию, можно создать еще несколько: классы Triangle и Circle, описывающие треугольники круг соответственно. Они не имеют специальных методов, хотя могли бы, но расширяют класс Shape, заимствуя представленные поля и методы, а также добавляя свои индивидуальные.

Например, класс Triangle содержит дополнительно информацию о трех углах.

Класс Circle содержит информацию о радиусе круга.

При этом оба класса наследника будут содержать методы getNumberOfVertices() и getNumberOfSides(), возвращающие количество вершин и сторон

соответственно.

Спецификаторы доступа

ключевые слова как public, protected и private.
Эти ключевые слова являются спецификаторами доступа, которые виляют на наследование и инкапсуляцию.

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

В примере выше есть базовый класс A, содержащий три поля с разными спецификаторами доступа. Если создать объект класса A и попытаться получить доступ к полям, то доступно будет лишь поле x, к другим же полям y и z нельзя обратиться напрямую через экземпляр класса.

Спецификаторы доступа

Public поля и методы являются доступными как через экземпляры класса, так и классам наследникам;

Protected не доступны через экземпляры класса, однако доступны классам наследникам;

Private являются недоступными ни через экземпляры, ни наследникам.

Также спецификаторы доступа могут быть использованы для изменения спецификаторов наследуемого класса.

Пусть у нас будет три класса, унаследованных от класса A, описанного выше:

```
class B : public A { };
class C : protected A { };
class D : private A { };
```

Спецификаторы доступа

Все эти классы имеют внутренний доступ к полю y базового класса.

При public наследовании (в случае класса B), спецификаторы доступа наследуемого класса не изменяются.

При protected наследовании (в случае класса C), спецификаторы доступа полей и методов public наследуемого класса изменяются и становятся protected. Таким образом, через экземпляр класса C нельзя будет получить доступ даже к полю x.

При private наследовании (в случае класса D), спецификаторы доступа всех полей и методов наследуемого класса изменяются и становятся private.

Если для описания класса используется ключевое слово struct, то все поля и методы по умолчанию являются public. Если же используется ключевое слово class и не указаны спецификаторы доступа в явном виде, то по умолчанию будет подставлен спецификатор private.

Полиморфизм

Полиморфизм предполагает, что объекты наследуют и реализуют какой-то обобщенный интерфейс, что позволяет работать с ними исключительно через него, при этом не имея информации о том, чем конкретно является объект.

В языке C++ нет такого понятия как интерфейс в чистом виде, в то время как в таких языках как C# или же Java интерфейсы являются встроенным механизмом языка.

Для реализации интерфейсов в C++ используется механизм виртуальных методов класса. Если функция объявлена со спецификатором virtual в начале сигнатуры, то такая функция может быть переопределена в классе наследнике и при этом будет правильно вызываться, даже если работа производится с экземпляром класса наследника лишь через указатель на базовый класс.

```
class A
{
public:
    virtual void print() { std::cout << "A::print()" << std::endl; }
};
class B : public A
{
public:
    void print() override { std::cout << "B::print()" << std::endl; }
};
int main()
{
    A* b = new B();
    b->print();
    delete b;
    return 0;
}
```

Полиморфизм

В данном случае при вызове функции print() через указатель на тип A, будет вызвана функция, реализованная в классе B.

Соответственно будет выведено “B::print()”. Если бы слова virtual не было, то была бы вызвана функция, реализованная в классе A.

Однако в данном случае A не является интерфейсом в чистом виде. Для полноценной реализации интерфейса необходимо использовать абстрактные классы.

Override *используется в классе-потомке, чтобы указать, что функция должна переопределять виртуальную функцию, объявленную в базовом классе.*

Будет рассмотрено позже:

Абстрактные классы

Виртуальный деструктор

Множественное наследование

Шаблоны классов