Как мы увидим далее, для создания хорошей архитектуры необходимо понимать и уметь применять принципы объектно-ориентированного программирования (ОО). Но что такое ОО?

Один из возможных ответов на этот вопрос: «комбинация данных и функций». Однако, несмотря на частое цитирование, этот ответ нельзя признать точным, потому что он предполагает, что `o.f()` — это нечто отличное от `f(o)`. Это абсурд. Программисты передавали структуры в функции задолго до 1966 года, когда Даль и Нюгор перенесли кадр стека функции в динамическую память и изобрели ОО.

Другой распространенный ответ: «способ моделирования реального мира». Это слишком уклончивый ответ. Что в действительности означает «моделирование реального мира» и почему нам может понадобиться такое моделирование? Возможно, эта фраза подразумевает, что ОО делает программное обеспечение проще для понимания, потому что оно становится ближе к реальному миру, но и такое объяснение слишком размыто и уклончиво. Оно не отвечает на вопрос, что же такое ОО.

Некоторые, чтобы объяснить природу ОО, прибегают к трем волшебным словам: *инкапсуляция*, *наследование* и *полиморфизм*. Они подразумевают, что ОО является комплексом из этих трех понятий или, по крайней мере, что объектно-ориентированный язык должен их поддерживать.

Давайте исследуем эти понятия по очереди.

# Инкапсуляция?

Инкапсуляция упоминается как часть определения ОО потому, что языки ОО поддерживают простой и эффективный способ инкапсуляции данных и функций. Как результат, есть возможность очертить круг связанных данных и функций. За пределами круга эти данные невидимы и доступны только некоторые функции. Воплощение этого понятия можно наблюдать в виде приватных членов данных и общедоступных членов-функций класса.

Эта идея определенно не уникальная для ОО. Например, в языке C имеется превосходная поддержка инкапсуляции. Рассмотрим простую программу на C:

**point.h**

```
struct Point;
struct Point* makePoint(double x, double y);
double distance (struct Point *p1, struct Point *p2);
```

**point.c**

```
#include "point.h"
#include <stdlib.h>
#include <math.h>

struct Point {
  double x,y;
};

struct Point* makepoint(double x, double y) {
  struct Point* p = malloc(sizeof(struct Point));
  p->x = x;
  p->y = y;
  return p;
}

double distance(struct Point* p1, struct Point* p2) {
  double dx = p1->x - p2->x;
  double dy = p1->y - p2->y;
  return sqrt(dx*dx+dy*dy);
}
```

Пользователи `point.h` не имеют доступа к членам структуры `Point`. Они могут вызывать функции `makePoint()` и `distance()`, но не имеют никакого представления о реализации структуры `Point` и функций для работы с ней.

Это отличный пример поддержки инкапсуляции не в объектно-ориентированном языке. Программисты на C постоянно использовали подобные приемы. Мы можем объявить структуры данных и функции в заголовочных файлах и реализовать их в файлах реализации. И наши пользователи никогда не получат доступа к элементам в этих файлах реализации.

Но затем пришел объектно-ориентированный C++ и превосходная инкапсуляция в C оказалась разрушенной.

По техническим причинам[1] компилятор C++ требует определять переменные-члены класса в заголовочном файле. В результате объектно-ориентированная версия предыдущей программы `Point` приобретает такой вид:

**point.h**

```
class Point {
public:
  Point(double x, double y);
  double distance(const Point& p) const;
```

[1] Чтобы иметь возможность определить размер экземпляра каждого класса.

```
private:
  double x;
  double y;
};
```

**point.cc**

```
#include "point.h"
#include <math.h>

Point::Point(double x, double y)
: x(x), y(y)
{}

double Point::distance(const Point& p) const {
  double dx = x-p.x;
  double dy = y-p.y;
  return sqrt(dx*dx + dy*dy);
}
```

Теперь пользователи заголовочного файла `point.h` знают о переменных-членах `x` и `y`! Компилятор не позволит обратиться к ним непосредственно, но клиент все равно знает об их существовании. Например, если имена этих членов изменятся, файл `point.cc` придется скомпилировать заново! Инкапсуляция оказалась разрушенной.

Введением в язык ключевых слов `public`, `private` и `protected` инкапсуляция была частично восстановлена. Однако это был лишь *грубый прием* (хак), обусловленный технической необходимостью компилятора видеть все переменные-члены в заголовочном файле.

Языки Java и C# полностью отменили деление на заголовок/реализацию, ослабив инкапсуляцию еще больше. В этих языках невозможно разделить объявление и определение класса.

По описанным причинам трудно согласиться, что ОО зависит от строгой инкапсуляции. В действительности многие языки ОО практически не имеют принудительной инкапсуляции[1].

ОО безусловно полагается на поведение программистов — что они не станут использовать обходные приемы для работы с инкапсулированными данными. То есть языки, заявляющие о поддержке ОО, фактически ослабили превосходную инкапсуляцию, некогда существовавшую в C.

[1] Например, Smalltalk, Python, JavaScript, Lua и Ruby.

# Наследование?

Языки ОО не улучшили инкапсуляцию, зато они дали нам наследование.

Точнее — ее разновидность. По сути, наследование — это всего лишь повторное объявление группы переменных и функций в ограниченной области видимости. Нечто похожее программисты на C проделывали вручную задолго до появления языков ОО[1].

Взгляните на дополнение к нашей исходной программе `point.h` на языке C:

**namedPoint.h**

```
struct NamedPoint;

struct NamedPoint* makeNamedPoint(double x, double y, char* name);
void setName(struct NamedPoint* np, char* name);
char* getName(struct NamedPoint* np);
```

**namedPoint.c**

```
#include "namedPoint.h"
#include <stdlib.h>

struct NamedPoint {
  double x,y;
  char* name;
};

struct NamedPoint* makeNamedPoint(double x, double y, char* name) {
  struct NamedPoint* p = malloc(sizeof(struct NamedPoint));
  p->x = x;
  p->y = y;
  p->name = name;
  return p;
}

void setName(struct NamedPoint* np, char* name) {
  np->name = name;
}

char* getName(struct NamedPoint* np) {
  return np->name;
}
```

[1] И не только программисты на C: большинство языков той эпохи позволяли маскировать одни структуры данных под другие.

**main.c**

```
#include "point.h"
#include "namedPoint.h"
#include <stdio.h>

int main(int ac, char** av) {
  struct NamedPoint* origin = makeNamedPoint(0.0, 0.0, "origin");
  struct NamedPoint* upperRight = makeNamedPoint
    (1.0, 1.0, "upperRight");
  printf("distance=%f\n",
    distance(
      (struct Point*) origin,
      (struct Point*) upperRight));
}
```

Внимательно рассмотрев основной код в файле `main.c`, можно заметить, что структура данных `NamedPoint` используется, как если бы она была производной от структуры `Point`. Такое оказалось возможным потому, что первые два поля в `NamedPoint` совпадают с полями в `Point`. Проще говоря, `NamedPoint` может маскироваться под `Point`, потому что `NamedPoint` фактически является надмножеством `Point` и имеет члены, соответствующие структуре `Point`, следующие в том же порядке.

Этот прием широко применялся[1] программистами до появления ОО. Фактически именно так C++ реализует единственное наследование.

То есть можно сказать, что некоторая разновидность наследования у нас имелась задолго до появления языков ОО. Впрочем, это утверждение не совсем истинно. У нас имелся трюк, хитрость, не настолько удобный, как настоящее наследование. Кроме того, с помощью описанного приема очень сложно получить что-то похожее на множественное наследование.

Обратите также внимание, как в `main.c` мне пришлось приводить аргументы `NamedPoint` к типу `Point`. В настоящем языке ОО такое приведение к родительскому типу производится неявно.

Справедливости ради следует отметить, что языки ОО действительно сделали маскировку структур данных более удобной, хотя это и не совсем новая особенность.

Итак, мы не можем дать идее ОО ни одного очка за инкапсуляцию и можем дать лишь пол-очка за наследование. Пока что общий счет не впечатляет.

Но у нас есть еще одно понятие.

---

[1] И продолжает применяться.