21.04.2025

ООП в Си

Филиппов Михаил Витальевич

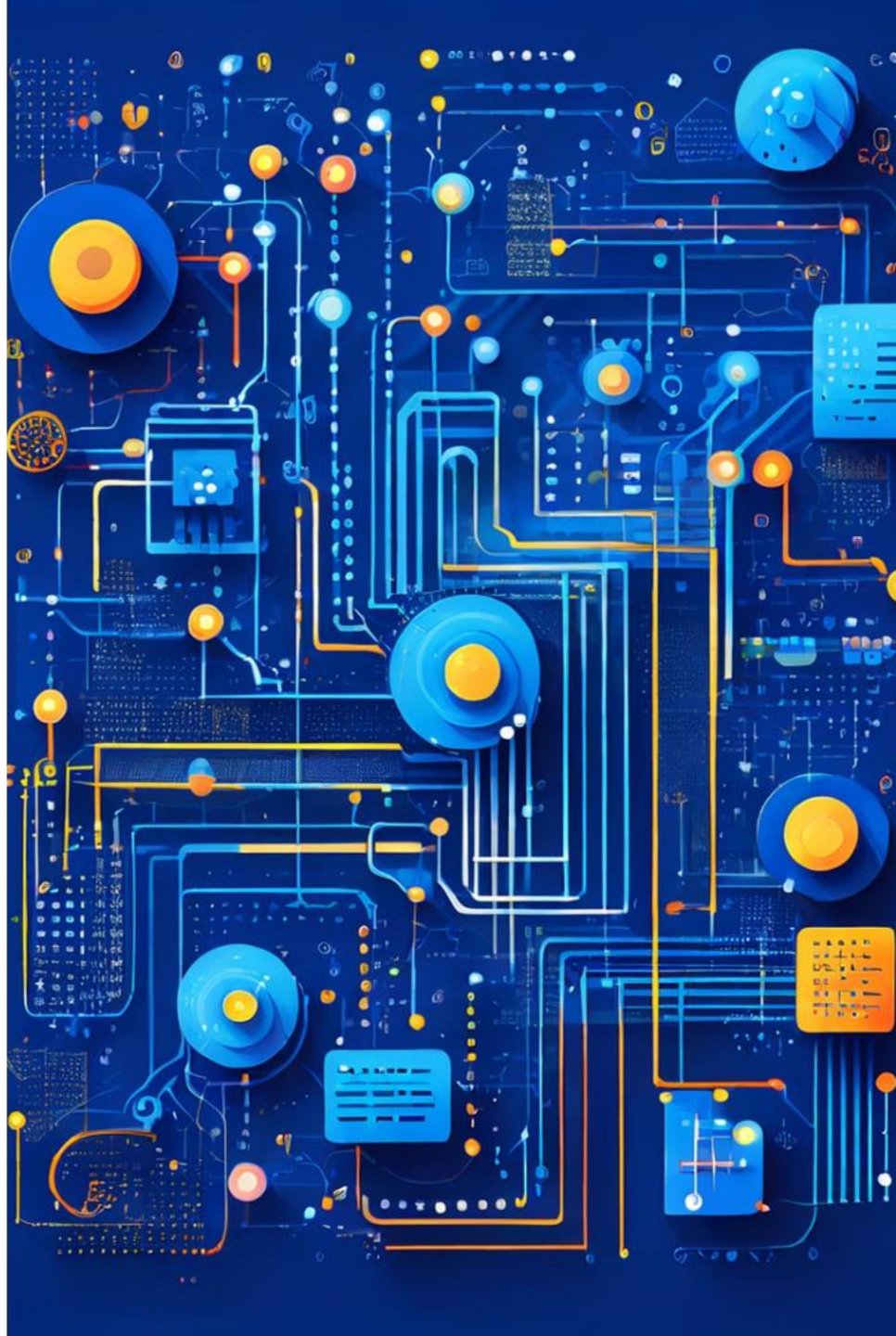
m.filippov@g.nsu.ru

89232283872

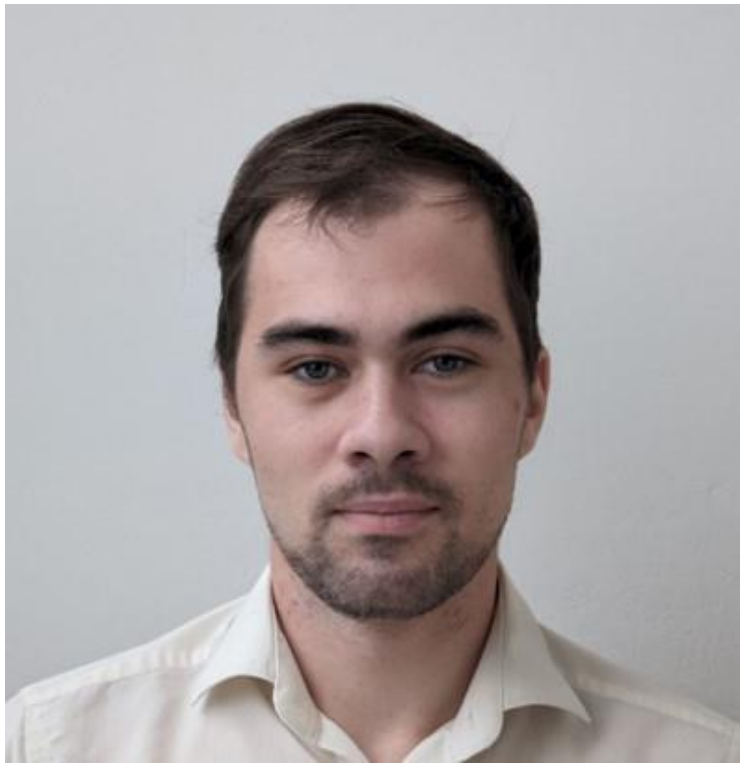
Императивное программирование, 2024-2025

Новосибирский государственный университет

*НАСТОЯЩАЯ НАУКА



Давайте познакомимся



Филиппов Михаил Витальевич

- Окончил магистратуру ФФ НГУ
- Окончил аспирантуру ИТ СО РАН
- Являюсь м.н.с. ИТ СО РАН
- 7+ лет опыт в программировании C/C++

Адженда

ООП и инкапсуляция	Композиция и агрегация	Наследование и полиморфизм	Абстракция данных и ООП в C++
30 минут	20 минут	25 минут	10 минут

Адженда

ООП и инкапсуляция	Композиция и агрегация	Наследование и полиморфизм	Абстракция данных и ООП в C++
30 минут	20 минут	25 минут	15 минут

Введение

Объектно-ориентированному программированию (ООП) посвящено множество отличных книг и статей. Но мне кажется, немногие из них рассматривают эту тему в контексте такого процедурного языка, как C! Действительно, разве это возможно? Можно ли писать объектно-ориентированные программы на языке, который на это не рассчитан? В частности, возможно ли написать на C программу в стиле ООП?

Если коротко, то да.

Вы также можете задаться вопросом, какой смысл обсуждать и изучать ООП, если вы собираетесь задействовать C в качестве основного языка программирования? Почти все зрелые проекты на C, такие как открытые ядра, реализации сервисов наподобие HTTPD, Postfix, nfsd и ftpd, а также многочисленные библиотеки, такие как OpenSSL и OpenCV, написаны в объектно-ориентированном стиле. Это не означает, что C — объектно-ориентированный язык; просто подход, который использовался при организации внутренней структуры указанных проектов, основан на объектно-ориентированном образе мышления.

В синтаксисе C нет таких объектно-ориентированных концепций, как классы, наследование и виртуальные функции. Но их поддержку можно реализовать самостоятельно. На самом деле почти все компьютерные языки, которые когда-либо существовали, имели все необходимое для поддержки ООП — задолго до Smalltalk, C++ и Java. Синтаксис C не может и не должен поддерживать объектно-ориентированные возможности; и не потому, что старый, а по нескольким очень весомым причинам. Проще говоря, на C можно писать в стиле ООП, просто это немного сложнее и требует чуть больше усилий.

Объектно-ориентированное мышление

Объектно-ориентированное мышление — способ анализа и разбиения на части окружающего нас мира. Когда вы смотрите на вазу, стоящую на столе, вы понимаете, что это два отдельных объекта, и для этого вам не нужно долго думать.

Вы интуитивно осознаете, что между этими предметами есть граница. Вы знаете, что изменение цвета вазы не повлияет на цвет стола.

Эти наблюдения говорят о том, что мы воспринимаем окружающий мир в объектноориентированном ключе. Иными словами, мы просто создаем у себя в голове отражение объектно-ориентированной реальности. Это часто можно видеть в компьютерных играх, программах для 3D-моделирования и инженерном ПО, где мы имеем дело со множеством взаимодействующих объектов.

Суть ООП состоит в применении объектно-ориентированного мышления к проектированию и разработке программного обеспечения.

Как мы мыслим

Концепция — мысленный или абстрактный образ, который существует в нашем воображении в виде мысли или идеи. Он может быть сформирован нашим восприятием материального объекта, но может быть и полностью воображаемым. Когда вы смотрите на дерево или думаете об автомобиле, вам на ум приходят соответствующие образы, которые представляют собой две разные концепции.

Концепции играют важную роль в объектно-ориентированном мышлении, ведь если вы не можете держать в уме понимание объектов, то не сможете извлечь информацию о том, что они представляют и к чему относятся, как и понять их взаимоотношения.

Таким образом, объектно-ориентированное мышление основано на концепциях и связях между ними.

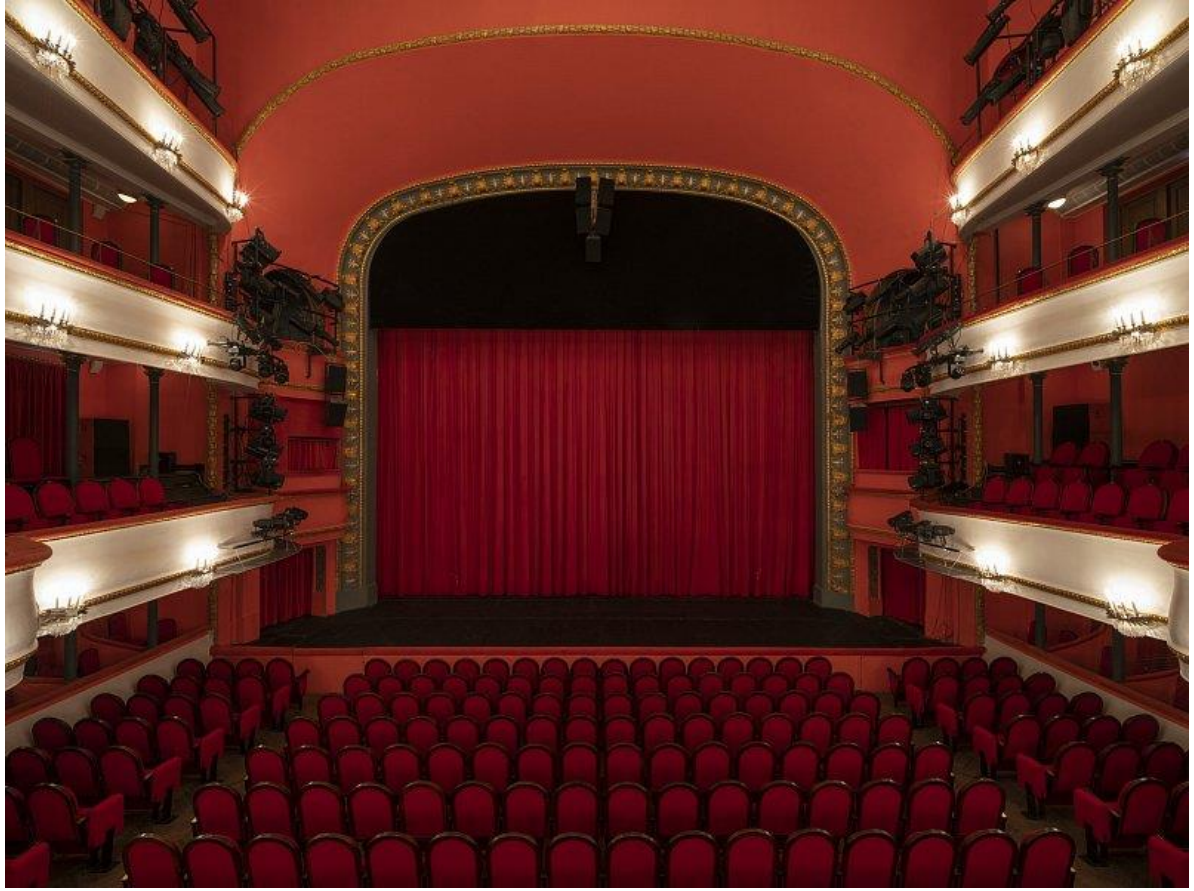
Четыре массива, названные по одному и тому же принципу с использованием префикса student_ и предназначенные для хранения информации о десяти студентах

```
char *student_first_names[10];
char *student_surnames[10];
int student_ages[10];
double student_marks[10];
```

Четыре массива с произвольными именами, предназначенные для хранения информации о десяти студентах

```
char *aaa[10];
char *bbb[10];
int ccc[10];
double ddd[10];
```

Диаграммы связей и объектные модели



Чтобы лучше понять все то, о чем мы говорили выше, рассмотрим практический пример. Допустим, вы создали описание сцены. Цель подобного рода описания — донести до аудитории соответствующие концепции.

Теперь остановитесь на минуту и подумайте о том, чем отличается ваш подход к извлечению концепций и их отношений. Каждый делает это по-своему. То же самое происходит при решении конкретной проблемы: вначале вам необходимо создать в уме диаграмму связей. Мы будем называть данный этап пониманием.

В объектно-ориентированной программе концепции представлены объектами, а вместо диаграммы связей, которую мы держим в уме, она использует объектную модель, хранящуюся в памяти.

Но зачем нам компьютеры для имитации наших диаграмм связей? Все просто: они незаменимы с точки зрения скорости и точности. Это классический ответ на подобные вопросы, и в нашем случае он подходит. Создание и хранение большой диаграммы связей и соответствующей объектной модели — сложная задача, с которой компьютеры справляются очень хорошо. К тому же объектную модель, созданную программой, можно сохранить на диск и использовать в будущем.

В коде нет никаких объектов

- Объекты могут существовать только в запущенной программе — точно так же, как концепции могут существовать лишь в разуме живого человека. Это значит, что за пределами активной программы никаких объектов нет.
- В вашем объектно-ориентированном коде нет никаких объектов. Они создаются, когда вы собираете и запускаете свою программу.
- На самом деле ООП не сводится к созданию объектов. Данная парадигма заключается в генерации набора инструкций, которые при запуске программы превращаются в полностью динамическую объектную модель. Поэтому объектно-ориентированный код, скомпилированный и запущенный, должен уметь создавать, изменять, связывать и даже удалять разные объекты.

Таким образом, написание ***объектно-ориентированного кода — непростая задача.*** Вам необходимо представлять себе объекты и отношения между ними в то время, когда они еще не существуют.

В самом коде мы лишь планируем создание объектов. Где и как они будут создаваться — определяют инструкции, которые мы генерируем. Конечно, создание — еще не все. Всевозможные операции с объектами можно описывать с помощью языка программирования. Объектно-ориентированный язык предоставляет набор инструкций (и грамматических правил), позволяющих писать и планировать различные операции, связанные с объектами. Каждый объект имеет отдельный жизненный цикл.

Атрибуты объектов

Любая концепция, которую мы себе представляем, имеет некие атрибуты. Если вернуться к нашему описанию классной комнаты, то у нас был коричневый стул с именем chair1. То есть каждый стул имел такой атрибут, как цвет, и в случае с chair1 этот цвет был коричневым. Мы знаем, что в классе было еще четыре стула, и каждый из них тоже имел некий цвет (возможно, уникальный). В нашем описании все они были коричневыми, но теоретически один или два из них могли быть желтыми.

Объект может иметь целый набор атрибутов. Совокупность значений, которые присвоены этим атрибутам, называется состоянием объекта. Состояние можно представить в виде списка значений, каждое из которых принадлежит определенному атрибуту, относящемуся к объекту. Объект можно изменять на протяжении его времени жизни. В таком случае его называют изменяемым (mutable). Это просто означает, что его состояние может меняться со временем. Объекты также могут не иметь состояния (или каких-либо атрибутов).

Объект может быть и неизменяемым — точно так же, как концепция числа 2, которую нельзя изменить. Неизменяемость означает, что состояние определяется в момент создания и после этого его нельзя модифицировать.

Объект без состояния можно считать неизменяемым объектом, поскольку его состояние не меняется на протяжении его существования. Состояние нельзя изменить, если его нет.

Предметная область

У любой программы, написанной для решения конкретной задачи, даже самой незначительной, есть четко определенная ***предметная область (или домен).*** Это еще один важный термин, который широко используется в литературе, посвященной разработке ПО. ***Предметная область*** определяет рамки, в которых проявляются возможности программы, и требования, коим эта программа должна удовлетворять.

Для достижения этой цели предметная область использует определенную и заранее подготовленную терминологию (словарь терминов), что помогает разработчикам оставаться в ее рамках. Все участники проекта должны иметь представление о предметной области, в которой он находится.

Например, банковское программное обеспечение обычно создается для строго определенной предметной области. Оно содержит набор общеизвестных терминов, таких как «счет», «кредитные средства», «баланс», «перевод», «заем», «процентная ставка» и т. д.

Определение предметной области можно прояснить с помощью терминов, находящихся в ее словаре; например, в банковской предметной области вы не найдете упоминания таких понятий, как «пациент», «лекарства» и «дозировка».

Если язык программирования не предоставляет средств для работы с концепциями предметной области, написать для нее ПО будет хоть и выполнимой, но, несомненно, сложной задачей. Более того, по мере увеличения количества кода его будет все сложнее развивать и сопровождать.

Отношения между объектами

Объекты могут быть взаимосвязанными; для обозначения этих связей они могут ссылаться друг на друга. Например, если вернуться к нашему примеру с классной комнатой, то объект student4 (четвертый студент) может иметь отношение к объекту chair3 (третий стул) в том смысле, что сидит на нем. Иными словами, student4 сидит на chair3. Таким образом, ***все объекты в системе ссылаются друг на друга и формируют сеть, известную как объектная модель***.

Если между двумя объектами есть связь, то изменение состояния одного из них может сказаться на состоянии другого. **Пример:**

Объект p1 содержит такие атрибуты: **{x: 53, y: 345, red: 120, green: 45, blue: 178}.**

Атрибуты объекта p2 следующие: **{x: 53, y: 346, red: 79, green: 162, blue: 23}.**

Синтаксис, который мы здесь использовали, очень похож (хоть и не до конца) на формат JSON. Чтобы их связать, нужно использовать дополнительный атрибут, который будет обозначать отношение между ними.

Таким образом, новое состояние
объекта p1 **{x: 53, y: 345, red: 120, green: 45, blue: 178, adjacent_down_ pixel: p2},**
а объекта p2 — на **{x: 53, y: 346, red: 79, green: 162, blue: 23, adjacent_up_pixel: p1}.**

Итак, мы видим, что для установления отношений между объектами их состояние (списки значений, соответствующие их атрибутам) нужно поменять. Для этого объекты расширяются за счет новых атрибутов, и их отношения становятся частью их состояния.

Объектно-ориентированные операции

Объектно-ориентированный язык программирования позволяет планировать создание и уничтожение объекта, а также изменение его состояния еще до запуска программы. Для начала посмотрим, как создается объект.
Более точный термин — «построение», но в технической литературе эту операцию принято называть созданием. Запланировать создание объекта можно двумя способами:

- Первый заключается в создании либо полностью пустого объекта (без каких-либо атрибутов в его состоянии), либо объекта с минимальным набором атрибутов.

  Остальные атрибуты будут определены и добавлены во время выполнения кода. Таким образом, у одного и того же объекта могут быть разные атрибуты в двух разных сценариях использования программы, в зависимости от изменений, обнаруженных в окружающей среде. Данный подход применяется в таких языках программирования, как JavaScript, Ruby, Python, Perl и PHP.
- Второй подход состоит в создании объектов, атрибуты которых определены заранее и не меняются по ходу выполнения. Новые атрибуты добавлять нельзя, и объект сохранит свою изначальную структуру. Меняться могут только значения атрибутов и лишь в том случае, если объект изменяемый.

  Чтобы использовать эту методику, программист должен заранее подготовить шаблон или класс объекта с описанием всех атрибутов, которые данный объект должен иметь на этапе выполнения. Затем этот шаблон нужно скомпилировать и передать объектно-ориентированному языку во время работы программы.
  Во многих языках программирования, таких как Java, C++ и Python, этот шаблон называется классом, а сам подход известен как классовое ООП. Стоит отметить, что, помимо классового ООП, Python поддерживает и прототипное.

Объекты имеют поведение

Любой объект вместе со всеми своими атрибутами имеет определенный список операций, которые может выполнять. Например, автомобиль может разгоняться, замедляться, поворачивать и т. д. В ООП эти операции всегда соответствуют требованиям предметной области. Например, в банковской объектной модели клиент может заказать открытие нового счета, но не может пообедать. Конечно, клиент, как и любой другой живой человек, может есть, но если потребление еды не относится к банковской сфере, то для соответствующего объекта данная возможность считается необязательной.

Любая операция может изменить состояние объекта, модифицируя значения его атрибутов. Приведу простой пример. Автомобиль может разгоняться. Разгон — это его возможность. В момент разгона меняется скорость автомобиля (один из его атрибутов).

Таким образом, **объект** — просто набор атрибутов и операций.

Почему язык С не является объектно-ориентированным

Язык С не объектно-ориентированный, но это не связано с его возрастом, иначе к текущему моменту мы бы уже нашли способ добавить в него поддержку ООП.
С другой стороны, у нас есть С++ — попытка создать на основе С язык с поддержкой ООП. Если бы С было суждено уступить место объектно-ориентированному языку, то на него не было бы спроса в наши дни, в основном из-за существования С++. Но текущая востребованность программистов на С показывает, что это не так.
Человек мыслит объектно-ориентированным образом, но машинные инструкции, которые выполняет центральный процессор, являются процедурными. Они выполняются одна за другой, хотя иногда процессору приходится переходить по другому адресу и выполнять инструкции, которые там находятся. Это очень напоминает вызовы функций в программе, написанной на процедурном языке, таком как С.
Язык С не может быть объектно-ориентированным, поскольку находится на границе между ООП и процедурным программированием. Объектно-ориентированность — человеческое понимание задачи, но процессор решает ее в процедурном стиле. Поэтому у нас должно быть нечто на стыке этих двух миров. В противном случае высокоуровневые программы, которые обычно написаны с применением ООП, нельзя было бы напрямую транслировать в процедурные инструкции, потребляемые процессором.

Инкапсуляция

Каждый объект имеет набор атрибутов и операций. Здесь же мы поговорим о том, как объединить эти атрибуты и операции в сущность под названием «объект». Для этого воспользуемся процессом, известным под названием «инкапсуляция».
Инкапсуляция означает объединение отдельных вещей в некую капсулу, которая представляет объект. Вначале мы это делаем в нашем воображении и затем воплощаем в коде. Каждый раз, когда вам кажется, что у объекта должны быть какие-то атрибуты и операции, вы производите мысленную инкапсуляцию; данный процесс необходимо реализовать на уровне программы.
Инкапсуляция — неотъемлемая часть языка программирования, без которой объединение связанных между собой переменных превратилось бы в непосильную задачу. Объект состоит из набора атрибутов и операций. Все они должны быть инкапсулированы. Сначала поговорим об инкапсуляции атрибутов.

Инкапсуляция атрибутов

Инкапсуляция по именам переменных существует во всех языках программирования (поскольку у переменных всегда есть имена), даже в ассемблере. В C инкапсуляция происходит за счет структур.

```c
int pixel_p1_x = 56;
int pixel_p1_y = 34;
int pixel_p1_red = 123;
int pixel_p1_green = 37;
int pixel_p1_blue = 127;

int pixel_p2_x = 212;
int pixel_p2_y = 994;
int pixel_p2_red = 127;
int pixel_p2_green = 127;
int pixel_p2_blue = 0;
```



```c
typedef struct {
    int x, y;
    int red, green, blue;
} pixel_t;
pixel_t p1, p2;

p1.x = 56;
p1.y = 34;
p1.red = 123;
p1.green = 37;
p1.blue = 127;

p2.x = 212;
p2.y = 994;
p2.red = 127;
p2.green = 127;
p2.blue = 0;
```

Инкапсуляция поведения

Объект — некая капсула с атрибутами и методами.
Метод — еще один стандартный термин, который обычно используется для обозначения части логики или функциональности, хранящейся в объекте. Его можно считать обычной функцией языка C, у которой есть имя, список аргументов и возвращаемый тип. Атрибуты представляют собой значения, а методы — поведение. Таким образом, объект имеет список значений и может демонстрировать определенное поведение в системе.

В таких классовых объектно-ориентированных языках, как C++, атрибуты и методы легко объединить в класс. В прототипных языках наподобие JavaScript объект, как правило, создается пустым (лат. ex nihilo — «из ничего») или клонируется из другого объекта. Чтобы объект имел поведение, в него нужно добавить методы.

Создание объекта clientObj в JavaScript

```
// создаем пустой объект
var clientObj = {};
// устанавливаем атрибуты
clientObj.name = "John";
clientObj.surname = "Doe";
// добавляем метод для открытия
банковского счета
clientObj.orderBankAccount = function () {
...
}
...
// вызываем метод
clientObj.orderBankAccount();
```

Инкапсуляция поведения

Создание объекта clientObj в C++

```
class Client
{
public:
    void orderBankAccount(){
        ...
    }
    std::string name;
    std::string surname;
};
...
Client clientObj;
clientObj.name = "John";
clientObj.surname = "Doe";
...
clientObj.orderBankAccount();
```

Если взглянуть на способы инкапсуляции разных элементов, которые применяются в открытых и широко известных проектах на языке C, то можно заметить нечто общее. Назовем инкапсуляцию в C **«неявная инкапсуляция»,** так как поведение в ней инкапсулируется так, что язык C не знает об этом. Исходя из того, что нам доступно в стандарте ANSI C, нет никакой возможности сделать так, чтобы языку C было известно о существовании классов. Поэтому любые попытки реализации объектной ориентированности в C неизбежно оказываются неявными.

Многие знаменитые проекты с открытым исходным кодом используют этот подход для написания частично объектно-ориентированного кода. Один из них — библиотека libcurl. Если взглянуть на ее исходный код, то можно увидеть много структур и функций, имена которых начинаются с curl_.

Инкапсуляция

Неявная инкапсуляция предполагает следующее.

- Использование структур языка C для атрибутов объекта (явная инкапсуляция атрибутов). Мы будем называть их структурами атрибутов.
- Для инкапсуляции поведения в C применяются функции. Это так называемые поведенческие функции. Вероятно, вы знаете, что язык C не позволяет размещать функции в структурах. Поэтому подобные функции должны существовать за пределами структуры атрибутов (неявная инкапсуляция поведения).
- В качестве одного из аргументов (обычно первого или последнего) поведенческие функции должны принимать указатель на структуру. Данный указатель ссылается на структуру атрибутов объекта. Дело в том, что поведенческой функции может понадобиться доступ (на чтение или запись) к атрибутам — это довольно частое явление.
- Поведенческие функции должны иметь подходящие имена, которые указывают на их принадлежность к одному и тому же классу объектов. Использование суффикса _t в именах структур атрибутов. Но, конечно, вас никто не заставляет их задействовать; вы можете выработать собственные соглашения об именовании.
- Объявления поведенческих функций обычно размещаются в одном заголовочном файле с объявлением структуры атрибутов. Этот файл называется заголовком объявления.
- Определения поведенческих функций обычно находятся в одном или нескольких отдельных исходных файлах, которые подключают заголовок объявления.

Обратите внимание: при использовании неявной инкапсуляции у нас нет никаких классов, но их существование подразумевается самим программистом.

Инкапсуляция поведения

Объявления структуры атрибутов и поведенческих функций класса Car

```
#ifndef CAR_H
#define CAR_H
// эта структура содержит все атрибуты,
относящиеся к объекту car
typedef struct
{
    char name[32];
    double speed;
    double fuel;
} car_t;
// это объявления функций, которые
составляют поведение объекта car
void car_construct(car_t *, const char *);
void car_destruct(car_t *);
void car_accelerate(car_t *);
void car_brake(car_t *);
void car_refuel(car_t *, double);
#endif
```

Инкапсуляция поведения

Определения поведенческих функций, входящих в класс Car

```c
#include <string.h>
#include "car.h"
// определения подключенных выше функций
void car_construct(car_t *car, const char *name) {
    strcpy(car->name, name);
    car->speed = 0.0;
    car->fuel = 0.0;
}
void car_destruct(car_t *car) { /*Здесь ничего не происходит!*/ }
void car_accelerate(car_t *car) {
    car->speed += 0.05;
    car->fuel -= 1.0;
    if (car->fuel < 0.0)
        car->fuel = 0.0;
}
void car_brake(car_t *car) {
    car->speed -= 0.07;
    if (car->speed < 0.0)
        car->speed = 0.0;
    car->fuel -= 2.0;
    if (car->fuel < 0.0)
        car->fuel = 0.0;
}
void car_refuel(car_t *car, double amount) { car->fuel = amount; }
```

Инкапсуляция поведения

Главная функция в примере

```c
#include <stdio.h>
#include "car.h"
int main(int argc, char **argv) // главная функция {
    car_t car;                          // создаем переменную объекта
    car_construct(&car, "Renault"); // создаем объект
    car_refuel(&car, 100.0);            // основной алгоритм
    printf("Car is refueled, the correct fuel level is %f\n", car.fuel);
    while (car.fuel > 0) {
        printf("Car fuel level: %f\n", car.fuel);
        if (car.speed < 80) {
            car_accelerate(&car);
            printf("Car has been accelerated to the speed: %f\n", car.speed);
        }
        else {
            car_brake(&car);
            printf("Car has been slowed down to the speed: %f\n", car.speed);
        }
    }
    printf("Car ran out of the fuel! Slowing down ...\n");
    while (car.speed > 0) {
        car_brake(&car);
        printf("Car has been slowed down to the speed: %f\n", car.speed);
    }
    car_destruct(&car); // уничтожаем объект
    return 0;
}
```

Принцип сокрытия информации

У инкапсуляции есть еще одно важное назначение или следствие — сокрытие информации. Это принцип, защищающий (или скрывающий) те или иные атрибуты и аспекты поведения, которые не должны быть видны извне. Под «извне» имеется в виду любой код, не относящийся к поведению объекта. Согласно данному определению, к приватному атрибуту или поведению объекта, которые не являются частью публичного интерфейса класса, не могут обращаться никакие другие функции.

Обратите внимание: поведенческие функции двух объектов одного типа, таких как car1 и car2, полученных из класса Car, имеют доступ к атрибутам любого объекта того же типа. Это обусловлено тем фактом, что поведенческие функции создаются в одном экземпляре сразу для всех объектов класса.

Например, роль атрибута может играть сила электрического тока, поданного на стартер, но она должна быть изолирована внутри объекта. Это относится и к определенным аспектам поведения объекта. Скажем, впрыск топлива в камеру сгорания — внутреннее поведение, которое не должно быть доступно пользователям, иначе они могут повлиять на него и нарушить работу двигателя.

Обычно код, который использует объект, становится зависимым от публичных атрибутов и поведения данного объекта. Это серьезная проблема. Если атрибуты, являющиеся частью публичного интерфейса, сделать приватными, то это может фактически нарушить сборку зависимого кода. После такого изменения все остальные участки кода, которые используют данные атрибуты публично, должны перестать компилироваться. Это означает нарушение обратной совместимости.

Принцип сокрытия информации

Пример 28.1

list.c

```
// #include "list.h" - не подключаем!!!!!!
…
typedef struct
{
    size_t size;
    int *items;
} list_t;
…
```

list.h

```
…
// структура атрибутов без
публично доступных полей
struct list_t;
…
```

main.c

```
…
#include "list.h"
…
```

```
$ gcc -c list.c -o list.o
$ gcc -c main.c -o main.o
$ gcc main.o list.o -o a.out
$ ./a.out
[4 6 1 5 ]
[5 1 6 4 ]
```

На самом деле компоновщик формирует программу из сочетания приватных определений и публичных объявлений.
Приватные поведенческие функции можно обозначать и по-другому. Мы можем использовать в их именах префикс __. Например, функция __check_index — приватная. ***Обратите внимание:*** у приватных функций нет соответствующих объявлений в заголовочном файле.

Используя публичный API, можно написать программу, которая компилируется, но не работает, если не предоставить и не скомпоновать соответствующие объектные файлы с приватным кодом.

Адженда

ООП и инкапсуляция	Композиция и агрегация	Наследование и полиморфизм	Абстракция данных и ООП в C++
30 минут	20 минут	25 минут	15 минут

Отношения между классами

Объектная модель — набор взаимосвязанных объектов. Количество связей в модели может быть огромным, но отношения между двумя объектами (или их соответствующими классами) могут быть организованы лишь несколькими способами. Существует две основные категории отношений: «иметь» и «быть».

Объекты и классы

```
// Структура атрибутов
// Person на языке C
typedef struct
{
    char name[32];
    char surname[32];
    unsigned int age;
} person_t;
// Класс Person на языке C++
class Person
{
public:
    std::string name;
    std::string family;
    uint32_t age;
};
```

- класс — это схема, которая используется в качестве «чертежа» для создания объектов;
- из одного класса можно создать много объектов;
- класс определяет, какие атрибуты должны присутствовать в каждом объекте, создаваемом на его основе. Но он не имеет никакого отношения к значениям, которые могут быть присвоены этим атрибутам;
- сам класс не потребляет никакой памяти (по крайней мере не в C или C++) и существует лишь в виде исходного кода и только на этапе компиляции. Но объекты существуют во время выполнения и потребляют память;
- создание объекта начинается с выделения памяти. Освобождение памяти — последняя операция, которую выполняет объект;
- объект должен создаваться сразу после выделения памяти и уничтожаться непосредственно перед ее освобождением;
- объект может владеть частью ресурсов наподобие потоков, буферов, массивов и пр., которые должны быть освобождены перед его уничтожением.

Композиция

Отношение **«композиция»** между двумя объектами означает, что один из них обладает другим. Иными словами, один объект состоит из другого.

Например, у автомобиля есть двигатель; то есть объект «автомобиль» содержит объект «двигатель». Таким образом, эти два объекта имеют отношение типа «композиция». При его использовании должно выполняться одно важное условие: ***время жизни внутреннего объекта привязано к времени жизни внешнего объекта (контейнера)***.

Если существует контейнер, то должен существовать и внутренний объект. Вместе с тем последний необходимо уничтожить непосредственно перед уничтожением контейнера. Это условие часто подразумевает, что внутренний объект является приватным для контейнера.

Некоторые элементы контейнера могут быть доступны через публичный интерфейс (или поведенческие функции) его класса, но время жизни внутренних объектов должно определяться самим контейнером. Если какой-то участок кода может уничтожить внутренний объект, не уничтожая внешний, то отношение между объектами больше нельзя считать композицией.

Следует отметить, что в некоторых областях двигатель может находиться за пределами автомобиля; например, в системе автоматизированного проектирования для машиностроения. Поэтому типы отношений между различными объектами определяются проблемной областью.

Композиция

Пример 28.2 – 5 файлов engine.c, engine.h, car.c, car.h, main.c

```
$ gcc -c engine.c -o engine.o
$ gcc -c car.c -o car.o
$ gcc -c main.c -o main.o
$ gcc engine.o car.o main.o -o a.out
$ ./a.out
Engine temperature before starting the car: 15.000000
Engine temperature after starting the car: 75.000000
Engine temperature after stopping the car: 15.000000
```

Предварительное объявление – объявление структуры без ее полей.

В большинстве случаев два объекта разных типов не должны знать о деталях реализации друг друга. Это продиктовано принципом сокрытия.

Агрегация

В агрегации тоже применяется контейнер, который содержит другой объект. Основное отличие в том, что время жизни контейнера никак не связано с временем жизни содержащегося в нем объекта.
При использовании агрегации внутренний объект можно создать даже раньше контейнера. Для сравнения, композиция подразумевает, что время жизни контейнера должно быть не короче времени жизни внутреннего объекта.

Агрегация

Пример 28.3 – 5 файлов player.c, player.h, gun.c, gun.h, main.c

```
$ gcc -c player.c -o player.o
$ gcc -c gun.c -o gun.o
$ gcc -c main.c -o main.o
$ gcc player.o gun.o main.o -o a.out
$ ./a.out
$
```

Если указатель агрегации необходимо установить в процессе создания, то адрес соответствующего объекта следует передать конструктору в качестве аргумента. Такую агрегацию называют ***принудительной***.

Если во время создания объекта указатель можно оставить обнуленным, как в приведенном примере, то это называется ***необязательной*** агрегацией. В таких случаях обнуление необходимо выполнять в конструкторе.

В отличие от композиции, контейнер здесь не является владельцем внутреннего объекта, поэтому не может контролировать его жизненный цикл.

При использовании необязательной агрегации контейнерный объект может оставаться без определенного значения. Поэтому с указателем агрегации нужно обращаться осторожно, поскольку любое обращение к неустановленному или нулевому указателю приводит к ошибке сегментации.

Композиция и агрегация

В объектной модели реального проекта агрегация обычно применяется чаще, чем композиция. Кроме того, агрегация лучше видна снаружи, поскольку ее работа требует наличия отдельных поведенческих функций (по крайней мере в публичном интерфейсе контейнера) для установки и сброса внутреннего объекта.

Агрегация, в отличие от композиции, является временным отношением между двумя объектами. Таким образом, композицию можно считать усиленной разновидностью владения (отношения типа «иметь»), а агрегацию — ослабленной.

Теперь встает вопрос: если агрегация носит временный характер для двух объектов, то является ли она временной для их соответствующих классов? Ответ отрицательный. Если существует малейшая вероятность того, что в будущем два объекта сформируют связь на основе агрегации, то их типы должны поддерживать это отношение на постоянной основе. То же самое относится и к композиции.

Даже небольшая вероятность возникновения отношения типа «агрегация» — повод для объявления некоторых указателей в структуре атрибутов контейнера, что означает внесение постоянного изменения.

Композиция и агрегация описывают владение некими объектами. То есть, они представляют отношение типа «иметь»; игрок имеет пистолет, автомобиль имеет двигатель. Если вам кажется, что один объект владеет другим, то между ними (и их соответствующими классами) следует установить отношение композиции или агрегации.

Адженда

ООП и инкапсуляция	Композиция и агрегация	Наследование и полиморфизм	Абстракция данных и ООП в C++
30 минут	20 минут	25 минут	15 минут

Наследование

Наследование также называют расширением, поскольку оно только добавляет новые атрибуты и операции в существующий объект или класс. Бывают случаи, когда один объект должен иметь те же атрибуты, которые есть в другом. Иными словами, новый объект — расширение существующего.

Наследование (или расширение) — это отношение типа «быть».

```
typedef struct
{
        char first_name[32];
        char last_name[32];
        unsigned int birth_year;
} person_t;
typedef struct
{
        char first_name[32];
        char last_name[32];
        unsigned int birth_year;
        char student_number[16];      // дополнительный атрибут
        unsigned int passed_credits; // дополнительный атрибут
} student_t;
```

Природа наследования

Если копнуть глубже и попытаться понять, что же на самом деле представляет собой наследование, то можно прийти к выводу: по своей природе это композиция.

Структуры атрибутов для классов Person и Student, но на этот раз вложенные

```c
typedef struct
{
        char first_name[32];
        char last_name[32];
        unsigned int birth_year;
} person_t;
typedef struct
{
        person_t person;
        char student_number[16];     // Дополнительный атрибут
        unsigned int passed_credits; // Дополнительный атрибут
} student_t;
```

На самом деле вложение структур с помощью структурных переменных (а не указателей) имеет большой потенциал. Это позволяет поместить в новую структуру переменную сложного типа так, чтобы данная структура являлась его расширением. Это так называемое ***восходящее приведение*** (или преобразование, upcasting) — то есть приведение структуры атрибутов потомка к типу структуры атрибутов родителя.

Природа наследования

Восходящее преобразование указателей на объекты Student и Person

```c
#include <stdio.h>
typedef struct {
        char first_name[32];
        char last_name[32];
        unsigned int birth_year;
} person_t;
typedef struct {
        person_t person;
        char student_number[16];     // дополнительный атрибут
        unsigned int passed_credits; // дополнительный атрибут
} student_t;
int main(int argc, char **argv) {
        student_t s;
        student_t *s_ptr = &s;
        person_t *p_ptr = (person_t *)&s;
        printf("Student pointer points to %p\n", (void *)s_ptr);
        printf("Person pointer points to %p\n", (void *)p_ptr);
        return 0;
}
```

```
$ gcc main.c -o a.out
$ ./a.out
Student pointer points to 0x7ffec46af4a0
Person pointer points to 0x7ffec46af4a0
```

Поведенческие функции класса Person можно вызывать с помощью указателя на объект student. Иными словами, поведенческие функции класса Person могут использоваться объектами student.

Природа наследования

Отношение наследования, которое не компилируется!

```
struct person_t;
typedef struct
{
        struct person_t person;       // генерирует ошибку!
        char student_number[16];      // дополнительный атрибут
        unsigned int passed_credits; // дополнительный атрибут
} student_t;
```

Строчка, в которой объявляется поле person, генерирует ошибку, поскольку мы не можем создать переменную из неполного типа. Его могут иметь только указатели, но не переменные. Память для неполного типа нельзя выделить даже в куче.

Так что же это означает? При реализации наследования с помощью вложенных структурных переменных структура student_t должна иметь доступ к определению person_t, которое согласно тому, что мы знаем об инкапсуляции, должно быть приватным и недоступным для любых других классов.

В связи с этим наследование можно реализовать двумя путями:

- сделать так, чтобы дочерний класс имел доступ к приватной реализации (определению) базового класса;
- сделать так, чтобы дочерний класс мог обращаться только к публичному интерфейсу базового класса.

I и II-ой подход к наследованию в C

Пример 28.4,5. 6 файлов main.c, person_p.c, person.c, person.h, student.c, student.hx

```
$ gcc -c person.c -o person.o
$ gcc -c student.c -o student.o
$ gcc -c main.c -o main.o
$ gcc person.o student.o main.o -o a.out
$ ./a.out
First name: John
Last name: Doe
Birth year: 1987
Student number: TA5667
Passed credits: 134
```

Здесь будем использовать указатель на структурную переменную родителя. Это позволит сделать дочерний класс независимым от реализации родительского, что является положительным фактором с точки зрения принципа сокрытия информации.

Его основное отличие в том, что класс Student зависит только от публичного интерфейса класса Person и не требует доступа к его приватному определению. Это важно, поскольку так мы можем изолировать классы друг от друга и легко изменять реализацию родителя, не меняя реализацию потомка.

В предыдущем примере класс Student, строго говоря, не нарушал принцип сокрытия, но имел такую возможность, поскольку имел доступ к определению структуры person_t и к ее полям. Как результат, он мог читать или изменять ее поля в обход поведенческих функций класса Person.

Класс Student должен повторять все поведенческие функции, объявленные внутри Person. Дело в том, что мы больше не можем привести указатель student_t к типу person_t. Иными словами, для указателей Student и Person больше не работает восходящее преобразование.

Приватное определение person_t теперь находится в исходном файле, и мы больше не используем приватный заголовок. Это значит, мы не станем делать его доступным ни для каких других классов, включая Student. Мы хотим добиться полноценной инкапсуляции класса Person и скрыть все детали его реализации.

Сравнение двух подходов

Ниже перечислены их основные сходства и различия.

- Оба подхода, по сути, демонстрируют отношение типа «композиция».
- В первом подходе структурная переменная находится в структуре атрибутов потомка. Здесь требуется доступ к приватной реализации родительского класса. Но во втором подходе используется структурный указатель неполного типа, принадлежащего структуре атрибутов родителя, поэтому теряется зависимость от приватной реализации родительского класса.
- В первом подходе родительский и дочерний типы тесно связаны. Во втором классы независимы друг от друга и все, что находится в родительской реализации, скрыто от потомка.
- В первом подходе может быть только один родитель. То есть это реализация одиночного наследования в C. Во втором же подходе родителей может быть сколько угодно; так выглядит концепция множественного наследования.
- В первом подходе структурная переменная родителя должна быть первым полем структуры атрибутов дочернего класса. Во втором указатели на родительские объекты могут находиться в любом месте структуры.
- В первом подходе у нас не было двух отдельных объектов. Родительский объект был частью дочернего, и указатель на дочерний объект являлся также указателем на родительский.
- В первом подходе мы могли использовать поведенческие функции родительского класса. Во втором нам пришлось делать для них обертки в дочернем классе.

Один из важнейших способов применения наследования —
реализация полиморфизма в объектной модели.

Что такое полиморфизм

```
// Создание трех объектов с типами Animal, Cat и Duck
struct animal_t *animal = animal_malloc();
animal_ctor(animal);
struct cat_t *cat = cat_malloc();
cat_ctor(cat);
struct duck_t *duck = duck_malloc();
duck_ctor(duck);
//Вызов операции sound из созданных ранее объектов
// в отсутствие полиморфизма
animal_sound(animal);
cat_sound(cat);
duck_sound(duck);
//Вызов одной и той же операции sound из всех трех
объектов
// это полиморфизм
animal_sound(animal);
animal_sound((struct animal_t*)cat);
animal_sound((struct animal_t*)duck);
```

На самом деле полиморфизм — не отношение между двумя классами. Прежде всего это метод использования одного и того же кода для реализации разного поведения. С его помощью можно расширять код и добавлять новые возможности, не прибегая к перекомпиляции всей кодовой базы.

Полиморфизм — просто предоставление разного поведения с помощью одного и того же публичного интерфейса (или набора поведенческих функций).

Представьте, что у вас есть два класса, Cat и Duck, каждый из которых имеет поведенческую функцию sound для вывода издаваемых ими звуков.

Как видите, мы использовали одну и ту же функцию, animal_sound, но с разными указателями, поэтому внутри выполнялись разные операции.

Вывод в результате вызова функции

```
Animal: Beeeep
Cat: Meow
Duck: Quake
```

Что такое полиморфизм

```c
typedef struct {
        ...
} animal_t;
typedef struct {
        animal_t animal;
        ...
} cat_t;
typedef struct {
        animal_t animal;
        ...
} duck_t;
//Функция animal_sound еще не полиморфная!
void animal_sound(animal_t* ptr) {
        printf("Animal: Beeeep");
}
// Эти вызовы могли бы быть полиморфными,
но таковыми НЕ являются!
animal_sound(animal);
animal_sound((struct animal_t*)cat);
animal_sound((struct animal_t*)duck);
```

```
Animal: Beeeep
Animal: Beeeep
Animal: Beeeep
```

В этом полиморфном коде подразумевается наличие отношения наследования между классом Animal и двумя другими классами, Cat и Duck, поскольку мы должны иметь возможность приводить указатели duck_t и cat_t к типу animal_t. Еще одна особенность данного кода состоит в том, что для использования преимуществ этой разновидности полиморфизма нам следует применять первый подход к наследованию в C.

Представим, что функция animal_sound определена так. Независимо от того, какой указатель она получает в качестве аргумента, ее поведение остается неизменным, поэтому без отдельного внутреннего механизма ее вызовы не будут полиморфными

Зачем нужен полиморфизм

Полиморфизм прежде всего нужен потому, что мы хотим использовать один и тот же код при работе с разными подтипами базового типа.

Нам не хочется модифицировать нашу логику при добавлении в систему новых подтипов или когда один из подтипов меняет свое поведение. Конечно, при появлении новой возможности нельзя совсем обойтись без обновления кода — какие-то изменения обязательно потребуются. Но благодаря полиморфизму мы можем существенно уменьшить количество необходимых изменений.
Еще один повод для использования полиморфизма связан с понятием абстракции. Абстрактные типы (или классы) обычно содержат какие-то неопределенные или нереализованные поведенческие функции, которые необходимо переопределять в дочерних классах, и полиморфизм играет в этом ключевую роль.
Поскольку мы хотим писать свою логику с помощью абстрактных типов, нам нужно как-то вызывать подходящую реализацию при работе с ними. Здесь тоже помогает полиморфизм. Полиморфное поведение необходимо в любом языке, иначе стоимость сопровождения крупных проектов будет быстро расти — например, при добавлении в код нового подтипа.

Полиморфное поведение в языке C

Пример 28.6. 8 файлов main.c, animal.c, animal.h, animal_p.h, cat.h, cat.c, duck.h, duck.c

```
$ gcc -c animal.c -o animal.o
$ gcc -c cat.c -o cat.o
$ gcc -c duck.c -o duck.o
$ gcc -c main.c -o main.o
$ gcc animal.o cat.o duck.o main.o -o a.out
$ ./a.out
Animal: Beeeep
Cat: Meow
Duck: Quacks
```

Чтобы получить полиморфизм в C, необходимо использовать первый подход к реализации наследования, который мы исследовали. Вдобавок можно применять указатели на функции. Однако на сей раз их следует оформить в виде полей структуры атрибутов. У нас есть три класса: Animal, Cat и Duck. Последние два — подклассы первого. У каждого из них есть по одному заголовку и исходнику. У класса Animal есть также дополнительный приватный заголовочный файл с определением его структуры. Приватный заголовок будет использоваться классами Cat и Duck.

У класса Animal есть две поведенческие функции. Первая, animal_sound, должна быть полиморфной и доступной для переопределения со стороны дочерних классов; вторая, animal_get_name, будет обычной функцией, которую дочерние классы не смогут переопределять.

При использовании полиморфизма каждый дочерний класс может предоставить собственную версию функции animal_sound. Иными словами, каждый подкласс может переопределить функцию, унаследованную от родительского. Поэтому любой потомок, который хочет выполнить переопределение, должен иметь свою разновидность функции. В случае с функцией animal_sound это означает, что мы будем вызывать ту ее версию, которую переопределил потомок.Вот почему мы используем указатели на функции. В каждом экземпляре animal_t будет указатель, предназначенный специально для операции animal_sound; он станет ссылаться на определение соответствующей полиморфной функции внутри класса.

Полиморфное поведение в языке C

Для каждой полиморфной поведенческой функции следует предусмотреть отдельный указатель. С помощью такого указателя нужно вызывать подходящую функцию в каждом подклассе.
Само полиморфное поведение находится в функции animal_sound.
Внутри конструктора animal_ctor мы сохраняем в поле sound_func объекта animal адрес __animal_sound. В такой конфигурации этот указатель на определение по умолчанию, __animal_sound, наследуется каждым потомком.
Далее внутри поведенческой функции animal_sound вызывается операция, на которую указывает поле sound_func. Это поле — указатель на фактическое определение операции, роль которого в данном случае играет __animal_sound.
Обратите внимание: animal_sound, в сущности, перенаправляет вызов к настоящей поведенческой функции.
Таким образом, если поле sound_func указывает на другую функцию, именно она будет выполняться при вызове animal_sound. Мы будем использовать этот прием, чтобы переопределить стандартную операцию sound в классах Cat и Duck.
Данная методика используется для переопределения стандартной операции sound. В классовом языке программирования поведенческие функции, которые должны быть полиморфными, требуют особого внимания и обращения. В противном случае поведенческая функция, не имеющая представленного выше внутреннего механизма, не может быть полиморфной. Вот почему эти функции называются особым образом и в языках вроде C++ обозначаются специальными ключевыми словами. Это виртуальные функции — операции, которые могут быть переопределены дочерними классами. Они должны отслеживаться компилятором, и при их переопределении в соответствующих объектах должны использоваться указатели на сами операции. Во время выполнения с помощью этих указателей вызывается подходящая версия функции.

Адженда

ООП и инкапсуляция	Композиция и агрегация	Наследование и полиморфизм	Абстракция данных и ООП в C++
30 минут	20 минут	25 минут	15 минут

Абстракция данных

Понятие абстракции данных может иметь разные значения в разных сферах науки и техники. Но в программировании, и особенно в ООП, оно, в сущности, относится к абстрактным типам данных. В классовой объектной ориентированности это то же самое, что и абстрактные классы. Абстрактными называют специальные классы, из которых нельзя создавать объекты; они не завершены и не готовы для использования в создании объектов.

Так зачем же они нужны? Дело в том, что, работая с абстрактными и обычными типами данных, мы можем избежать возникновения жестких зависимостей между различными частями кода.

Например, между классами Человек и Яблоко могут быть следующие отношения:

Объект класса Человек ест объект класса Яблоко.

Объект класса Человек ест объект класса Апельсин.

Если к Яблоку и Апельсину нужно добавить другие классы, которые может есть объект класса Человек, то у последнего появятся дополнительные связи. Вместо этого мы можем создать абстрактный класс Фрукт, который будет родителем Яблока и Апельсина. Это позволит ограничиться лишь одним отношением — между Человеком и Фруктом. Таким образом, мы можем свести два предыдущих утверждения к одному:

Объект класса Человек ест объект одного из подтипов класса Фрукт.

Класс Фрукт — абстрактный, поскольку не несет в себе информацию о форме, вкусе, запахе, цвете и многих других атрибутах, свойственных фруктам. Значения этих атрибутов становятся известными, только когда мы получаем яблоко или апельсин. Классы Яблоко и Апельсин называют конкретными типами.

Абстракция данных

Мы можем повысить уровень абстракции. Класс Человек может также есть Салат и Шоколад. Поэтому допустимо сказать следующее:

Объект типа Человек может есть объект одного из подтипов класса Пища.

Как видите, Пища находится на еще более высоком уровне абстракции, чем Фрукт. Это отличный подход к проектированию объектных моделей с минимальной зависимостью от конкретных типов и возможностью легкого расширения, если в будущем в систему понадобится добавить другие конкретные типы.

Возвращаясь к предыдущему примеру, мы могли бы абстрагироваться еще сильнее, используя тот факт, что Человеку свойственно потреблять пищу. Таким образом, можно получить еще более абстрактное утверждение:

Объект одного из подтипов класса Едок ест объект одного из подтипов класса Пища.

Мы можем продолжить абстрагирование объектной модели и подобрать типы данных, уровень абстракции которых выше того, который необходим для решения нашей задачи. Это так называемое чрезмерное абстрагирование. Оно случается, когда вы пытаетесь создать абстрактные типы, не имеющие реального применения в контексте ваших текущих или будущих потребностей. Этого следует избегать любой ценой, поскольку абстракция, несмотря на все свои преимущества, может создать проблемы.

При определении того, насколько сильно нужно абстрагировать объектную модель, можно руководствоваться принципом абстракции.

Абстракция данных

На посвященной ему странице в «Википедии», утверждается следующее.

Каждая существенная часть функциональности программы должна быть реализована только на одном участке кода. В целом, если разные участки кода имеют похожие функции, то их имеет смысл объединить путем абстрагирования того, чем они отличаются.

Данный принцип лежит в основе рассмотренных нами отношений между объектами. Поэтому на практике, если вы не ожидаете, что конкретная логика будет варьироваться в будущем, на данном этапе не нужно вводить абстракцию.

Для создания абстракций в языках программирования используются две возможности: наследование и полиморфизм. Абстрактный класс, такой как Пища, — супертип по отношению к своим конкретным классам, таким как Яблоко. И это достигается за счет наследования.

Полиморфизм тоже играет важную роль. Некое поведение абстрактного типа не может иметь реализации по умолчанию на этом абстрактном уровне. Например, у атрибута вкус, реализованного в классе Пища в виде поведенческой функции eatable_get_taste, не может быть определенного значения. Иными словами, мы не можем создать объект напрямую из класса Пища, не зная, как определить поведенческую функцию eatable_get_taste.

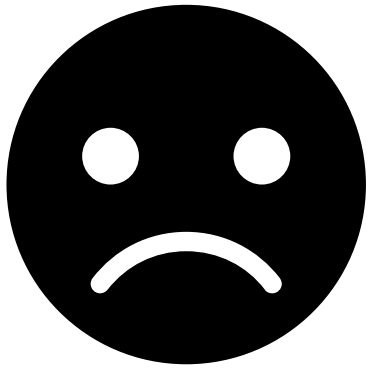
Эта функция может иметь определение, только когда дочерний класс является достаточно конкретным. Например, мы знаем, что у Яблок атрибут вкус должен быть сладким (предположим, что кислых яблок не существует). Вот где на помощь приходит полиморфизм. Он позволяет дочернему классу переопределить поведение родителя и, к примеру, вернуть подходящий вкус.

Абстракция данных

Поведенческие функции, которые могут переопределяться дочерними классами, называются виртуальными. Имейте в виду, что у виртуальной функции может вообще не быть никакого определения. И это, конечно, делает абстрактным класс, к которому она принадлежит.

При достижении определенного уровня абстракции у нас получаются классы без каких-либо атрибутов или определений по умолчанию, а только с виртуальными функциями. Такие классы называются интерфейсами. Иными словами, они предоставляют возможности, но не предлагают никакой реализации; обычно с их помощью в программных проектах создаются зависимости между различными компонентами. В наших предыдущих примерах роль интерфейсов играли классы Едок и Пища.

Обратите внимание: как и в случае с абстрактными классами, из интерфейсов нельзя создавать объекты. В представленном коде показано, почему эту концепцию нельзя воплотить в языке C.



Абстракция данных

```
typedef enum {SWEET, SOUR} taste_t;
// тип указателя на функцию
typedef taste_t (*get_taste_func_t)(void*);
typedef struct {
// указатель на определение виртуальной функции
get_taste_func_t get_taste_func;
} eatable_t;
eatable_t* eatable_new() { ... }
void eatable_ctor(eatable_t* eatable) {
// у этой виртуальной функции нет определения по умолчанию
eatable->get_taste_func = NULL;
}
// виртуальная поведенческая функция
taste_t eatable_get_taste(eatable_t* eatable) {
return eatable->get_taste_func(eatable);
}
eatable_t *eatable = eatable_new();
eatable_ctor(eatable);
// Ошибка сегментации!
taste_t taste = eatable_get_taste(eatable);
free(eatable);
```

Внутри конструктора мы присвоили функции get_taste_func значение NULL. Поэтому вызов виртуальной функции eatable_get_taste повлечет ошибку сегментации. Это практический аспект того, почему из интерфейса Eatable нельзя создавать объекты; другие причины обусловлены самим понятием «интерфейс» и правилами проектирования.

Чтобы избежать создания объекта абстрактного типа, из публичного интерфейса класса можно убрать функцию-аллокатор. В результате удаления аллокатора создание объектов из структуры атрибутов родителя становится доступным только дочерним классам.

Внешний код больше не может этого делать.

Если подытожить, то для получения абстрактного класса в C необходимо обнулить указатели на виртуальные функции, у которых не должно быть определения по умолчанию на абстрактном уровне. На высочайшем уровне абстракции у нас получится интерфейс, у которого обнулены все указатели на функции. Чтобы внешний код не мог создавать объекты абстрактных типов, следует удалить функцию-аллокатор из публичного интерфейса.

Доп. материал ООП C vs C++

Инкапсуляция

main.c

```c
#include <stdio.h>
typedef struct
{
	int width;
	int length;
} rect_t;
int rect_area(rect_t *rect)
{
	return rect->width * rect->length;
}
int main(int argc, char **argv)
{
	rect_t r;
	r.width = 10;
	r.length = 25;
	int area = rect_area(&r);
	printf("Area: %d\n", area);
	return 0;
}
```

main.cpp

```cpp
#include <iostream>
class Rect
{
public:
    int Area()
    {
        return width * length;
    }
    int width;
    int length;
};
int main(int argc, char **argv)
{
    Rect r;
    r.width = 10;
    r.length = 25;
    int area = r.Area();
    std::cout << "Area: " << area << std::endl;
    return 0;
}
```

Инкапсуляция

```
$ gcc -S main.c -o main_c.s
$ g++ -S main.cpp -o main_cpp.s
```

main_c.s

```
…
rect_area:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movq    %rdi, -8(%rbp)
        movq    -8(%rbp), %rax
        movl    (%rax), %edx
        movq    -8(%rbp), %rax
        movl    4(%rax), %eax
        imull   %edx, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
…
```

Невероятно, но они абсолютно одинаковы!

Для функций на C и C++ был сгенерирован ассемблерный код с высокой степенью схожести.

Можно сделать вывод: компилятор C++ использует подход, аналогичный тому, с которым мы познакомились при обсуждении неявной инкапсуляции.

Обратите внимание на ассемблерные инструкции, выделенные в обоих терминалах жирным шрифтом. Переменные width и length считываются путем прибавления к адресу памяти, переданному в первом аргументе. Первый аргумент-указатель находится в регистре %rdi, что соответствует спецификации System V ABI. Из этого можно сделать вывод: компилятор C++ изменил функцию Area так, чтобы ее первым аргументом был указатель на сам объект.

main_cpp.s

```
…
_ZN4Rect4AreaEv:
.LFB1731:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movq    %rdi, -8(%rbp)
        movq    -8(%rbp), %rax
        movl    (%rax), %edx
        movq    -8(%rbp), %rax
        movl    4(%rax), %eax
        imull   %edx, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
…
```

Наследование

main.c

```c
#include <string.h>
typedef struct
{
	char c;
	char d;
} a_t;
typedef struct
{
	a_t parent;
	char str[5];
} b_t;
int main(int argc, char **argv)
{
	b_t b;
	b.parent.c = 'A';
	b.parent.d = 'B';
	strcpy(b.str, "1234");
	// чтобы исследовать
структуру памяти, здесь нужно
создать точку останова
	return 0;
}
```

Наследование легче поддается анализу, чем инкапсуляция. В C++ указатели дочерних типов можно присваивать указателю родительского типа. Кроме того, у дочернего класса должен быть доступ к приватному определению родителя.

Оба факта говорят о том, что в C++ используется первый из двух подходов к наследованию.

Но не все так просто. C++ поддерживает множественное наследование, которое исключено в первом подходе.

main.cpp

```cpp
#include <string.h>
class A
{
public:
    char c;
    char d;
};
class B : public A
{
public:
    char str[5];
};
int main(int argc, char **argv)
{
    B b;
    b.c = 'A';
    b.d = 'B';
    strcpy(b.str, "1234");
    // чтобы исследовать
структуру памяти, здесь нужно
создать точку останова
    return 0;
}
```

Наследование

main.c

```
$ gcc -g main.c -o a_c.out
$ gdb ./a_c.out
...
(gdb) b main.c:19
Breakpoint 1 at 0x117e: file main.c, line 19.
(gdb) r
Starting program: /mnt/.../a_c.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0x7fffffffdb18) at main.c:19
19              return 0;
(gdb) x/7c &b
0x7fffffffd9f1: 65 'A'  66 'B'  49 '1'  50 '2'  51 '3'  52 '4'  0 '\000'
(gdb) c
Continuing.
[Inferior 1 (process 1194) exited normally]
(gdb) q
```

Наследование

main.cpp

```
$ g++ -g main.cpp -o a_cpp.out
$ gdb ./a_cpp.out
...
(gdb) b main.cpp:20
Breakpoint 1 at 0x117e: file main.cpp, line 20.
(gdb) r
Starting program: /mnt/c/.../a_cpp.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0x7fffffffdb18) at main.cpp:20
20          return 0;
(gdb) x/7c &b
0x7fffffffd9f1: 65 'A'  66 'B'  49 '1'  50 '2'  51 '3'  52 '4'  0 '\000'
(gdb) c
Continuing.
[Inferior 1 (process 2355) exited normally]
(gdb) q
```

Структура памяти в этих двух терминалах и значения, хранящиеся в атрибутах, ничем не отличаются. Не стоит удивляться тому, что в версии на C++ поведенческие функции и атрибуты находятся внутри класса; при выполнении они интерпретируются отдельно. В C++ атрибуты класса всегда собираются в одном блоке памяти соответствующего объекта, вне зависимости от того, где вы их укажете, а функции всегда отделены от атрибутов.

Наследование

Здесь видно, что класс c_t хочет быть наследником сразу двух классов: a_t и b_t. После объявления всех типов мы создаем объект c_obj. Дальше создаются разные указатели.
Необходимо отметить, что все эти указатели должны содержать один и тот же адрес. Указатели a_ptr и c_ptr можно безопасно использовать с любыми поведенческими функциями из классов a_t и c_t, чего нельзя сказать об указателе b_ptr; он ссылается на поле в классе c_t, которое является объектом a_t. Попытка обратиться к этому полю внутри b_t с помощью указателя b_ptr приведет к непредсказуемому поведению

main.c

```
typedef struct { ... } a_t;
typedef struct { ... } b_t;
typedef struct {
    a_t a;
    b_t b;
    ...
} c_t;
c_t c_obj;
a_t* a_ptr = (a_ptr*)&c_obj;
b_t* b_ptr = (b_ptr*)&c_obj;
c_t* c_ptr = &c_obj;

c_t c_obj;
a_t* a_ptr = (a_ptr*)&c_obj;
b_t* b_ptr = (b_ptr*)(&c_obj + sizeof(a_t));
c_t* c_ptr = &c_obj;
```

В предпоследней строке можно видеть, что мы прибавили размер объекта a_t к адресу c_obj; благодаря этому указатель теперь ссылается на поле b внутри c_t.
Имейте в виду: приведение типов в C не делает ничего магического; оно не влияет на передаваемые значения (в данном случае на адрес в памяти). После присваивания адрес, указанный справа, в конечном счете будет скопирован в левую часть выражения.

Наследование

```
#include <string.h>
class A
{
public:
    char a;
    char b[4];
};
class B
{
public:
    char c;
    char d;
};
class C
{
public:
    char e;
    char f;
};
class D : public A, public B, public C
{
public:
    char str[5];
};
```

main.cpp

```
int main(int argc, char **argv)
{
    D d;
    d.a = 'A';
    strcpy(d.b, "BBB");
    d.c = 'C';
    d.d = 'D';
    d.e = 'E';
    d.f = 'F';
    strcpy(d.str, "1234");
    A *ap = &d;
    B *bp = &d;
    C *cp = &d;
    D *dp = &d;
    // в этой строке нужно создать точку останова
    return 0;
}
```

Представьте, что у нас есть класс D, который является наследником трех разных классов: A, B и C.

Наследование

main.cpp

```
$ gcc -g main.c -o a_c.out
$ gdb ./a_c.out
...
(gdb) b main.cpp:40
Breakpoint 1 at 0x11b9: file main.cpp, line 40.
(gdb) r
Starting program: /mnt/c/.../a_cpp.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0x7fffffffdb18) at main.cpp:40
40          return 0;
(gdb) x/14c &d
0x7fffffffd9ea: 65 'A'  66 'B'  66 'B'  66 'B'  0 '\000'        67 'C'  68 'D'  69 'E'
0x7fffffffd9f2: 70 'F'  49 '1'  50 '2'  51 '3'  52 '4'  0 '\000'
```

Как видите, атрибуты находятся по соседству. Это говорит о том, что в памяти объекта d находится сразу несколько объектов родительских классов.

Наследование

Но что насчет указателей ap, bp, cp и dp? В C++ приведение типов в ходе присваивания дочернего указателя родительскому (восходящее приведение) может происходить неявно.

main.cpp

```
(gdb) print ap
$1 = (A *) 0x7fffffffd9ea
(gdb) print bp
$2 = (B *) 0x7fffffffd9ef
(gdb) print cp
$3 = (C *) 0x7fffffffd9f1
(gdb) print dp
$4 = (D *) 0x7fffffffd9ea
(gdb) q
A debugging session is active.

        Inferior 1 [process 587] will be killed.

Quit anyway? (y or n) y
```

Как видите, начальный адрес объекта d ($4) совпадает с адресом, на который указывает ap ($1). Это явное свидетельство того, что компилятор C++ размещает объект типа A в первом поле соответствующей структуры атрибутов класса D. Учитывая адреса в указателях и результат выполнения команды x, можно утверждать, что объект типа B и затем объект типа C были размещены на одном участке памяти, принадлежащем объекту d.

Кроме того, эти адреса показывают, что приведение типов в C++ не является пассивной операцией; когда один тип приводится к другому, мы можем выполнять арифметические действия с передаваемым адресом. Это было сделано для того, чтобы преодолеть проблему, с которой мы столкнулись при реализации множественного наследования в C. Теперь эти указатели можно легко использовать во всех поведенческих функциях, и арифметические операции притом не нужно выполнять вручную.

Полиморфизм

Пример 28_9

Сравнение внутренних механизмов, стоящих за полиморфизмом в C и C++, — нелегкая задача. Ранее был продемонстрирован простой способ реализации полиморфизма в C, но в C++ используется куда более замысловатый подход, хотя основной принцип все тот же.

В C родительский класс должен хранить в своей структуре атрибутов список указателей на функции. Эти указатели (в родительском классе) либо ссылаются на определения виртуальных функций по умолчанию, либо равны нулю. У псевдокласса, представленного в данном листинге, есть m невиртуальных поведенческих функций и n виртуальных.

Обратите внимание на то, что все поведенческие функции полиморфны. Их называют виртуальными. В некоторых языках, таких как Java, они называются виртуальными методами.

Невиртуальные функции не полиморфны, и при их вызове поведение никогда не будет меняться. Иными словами, это обычная функция, которая просто выполняет логику внутри определения и не перенаправляет вызов другой функции. Для сравнения, виртуальные функции должны перенаправлять вызовы подходящим определениям, установленным в конструкторе родителя или потомка. Если дочерний класс хочет переопределить какие-либо унаследованные виртуальные функции, то должен обновить их указатели.

Дочернему классу достаточно обновить лишь несколько указателей в структуре атрибутов родителя. В C++ применяется похожий подход. Когда вы объявляете поведенческую функцию как виртуальную (с помощью ключевого слова virtual), C++ создает массив указателей на функции, похожий на тот, который мы только что видели.

Мы также добавили по одному атрибуту с указателем для каждой виртуальной функции, но в C++ это делается более элегантно. Для этого используется массив, который называется виртуальной таблицей (virtual table, vtable). Она создается непосредственно перед созданием самого объекта и заполняется во время вызова сначала конструктора базового класса, а затем конструктора.

Поскольку виртуальная таблица заполняется только внутри конструкторов, вызова полиморфных методов из конструктора родительского или дочернего класса следует избегать, поскольку их указатели могут еще не подвергнуться обновлению и ссылаться не на то определение.

Абстракция

```
enum class Taste
{
    Sweet,
    Sour
};
// это интерфейс
class Eatable
{
public:
    virtual Taste GetTaste() = 0;
};
class Apple : public Eatable
{
public:
    Taste GetTaste() override
    {
        return Taste::Sweet;
    }
};
```

Абстракция в C++ возможна благодаря чистым виртуальным функциям. Если сделать метод класса виртуальным и присвоить ему ноль, то получится чистая виртуальная функция.
Внутри класса Eatable находится виртуальная функция GetTaste с нулевым значением. Это чистая виртуальная функция, которая делает весь класс абстрактным. Вы больше не можете создавать объекты типа Eatable — язык C++ этого не позволяет. Кроме того, Eatable является интерфейсом, поскольку все его функции-члены чисто виртуальные. GetTaste можно переопределить в дочернем классе.
Чистые виртуальные функции очень похожи на обычные виртуальные. Адреса их определений точно так же хранятся в виртуальной таблице, но с одним отличием. Указатель на чистую виртуальную функцию изначально равен NULL; для сравнения, во время выполнения конструктора указатель на обычную виртуальную функцию должен ссылаться на определение по умолчанию.
В отличие от компилятора языка C, которому ничего не известно об абстрактных типах, компилятор C++ о них знает и при попытке создать из них объект генерирует ошибку компиляции.