II

ООП

C++: ООП, идея I

Концепция **объектно-ориентированного программирования** (сокращённо — ООП). ООП представляет собой ещё один стиль (в дополнение к процедурному) организации программного кода.
Появление ООП обусловлено стремлением исследователей в области компьютерных наук найти более простой способ разбиения сложных программ на отдельные компоненты. В первом приближении, схожую задачу решала и функциональная декомпозиция исходного кода. Но с ростом сложности возникавших в программировании задач продолжались попытки найти подход, с одной стороны — упрощающий написание кода, с другой — упрощающий его сопровождение.
Идеи ООП относят к началу 60-х годов прошлого века. Появившись в разных научных группах, все они включали в себя понятие **объекта**. Концептуально, это некоторая

сущность, которая выполняет некоторый набор различных задач. Причём, действия, которые использует объект для выполнения каждой из задач, скрыты от того, кто даёт ему команду на выполнение конкретной задачи.
Наглядные аналогии для термина **объект** приводил выдающийся математик и специалист в области компьютерных наук — **Алан Кэй** (англ. Alan Kay). Кроме всего прочего, он является одним из создателей полностью объектно-ориентированного языка программирования **Smalltalk**. Кэй для объяснения понятия «объект» использовал два примера. Первый заключался в сравнении объектов с биологическими клетками, которые общаются между собой посредством некоторой системы сигналов. Второй пример — это набор отдельных компьютеров, объединённых в сеть и общающихся между собой **только с помощью** посылки друг другу сообщений. В обоих примерах именно передача

C++: ООП, идея III

сообщений/сигналов является ключевой идеей: каждый объект не показывает свою внутренную структуру, но принимает сообщения и может (но, не обязан) на них отреагировать. Например, послать ответное сообщение. При этом, как в аналогиях с клетками и отдельными компьютерами, внутреннее устройство объектов может быть сложным. Однако, это не волнует того, кто взаимодействует с конкретным объектом.
Основной момент, почему ООП важно: большая часть стандартной библиотеки C++ написана в этом стиле.

C++: ООП, идея

Как итог концепции объектов:

- у объекта есть своё внутреннее состояние: набор полей, в которых хранятся данные;
- у объекта есть набор операций, которые могут быть запрошены для него;
- доступ к полям/операциям объекта может быть ограничен;
- объекты одних типов могут создаваться от других типов: то есть, включать в себя поля/операции сторонней группы объектов.

ООП: как реализовать?

- **Классовое** ООП: язык программирования предоставляет средства для определения типа, который описывает поля для данных и набор операций. Затем объекты такого типа создаются для использования в программе. Именно такая реализация ООП в **C++**.
- **Прототипное** ООП: объект нового типа создаётся путём расширения объекта уже известного типа. Пример языков: **Javascript, Lua**. Не поддерживается в **C++**.

C++ предоставляет возможности для реализации следующих аспектов объектно-ориентированной парадигмы:

- **инкапсуляция** — объединение в одном типе полей для данных и операций над ними;
- **сокрытие данных** — возможность ограничить прямой доступ с помощью «дот»-нотации (dot notation) к полям и/или операциям;
- **наследование** — возможность создавать новый тип на основе уже существующего. Такой тип называют *производным*, а тот, на основе которого он был создан, — *базовым*;
- **полиморфизм** — возможность использовать объекты *производного* типа там, где ожидаются объекты *базового*.

Формальное определение **класса**

Термин

Под **классом** понимают составной тип данных, объединяющий множество проименнованных типизированных элементов (полей) и множество функций для совершения действий над ними. Причём, как к элементам, так и функциям прямой доступ может быть ограничен.

Термин... нужно больше терминов

Функции в C++, которые применяются *только* к объектам конкретного типа, получили название **методов**.

Последний введён для смыслового (семантического) различия между обычными функциями (также известны как *свободные функции*) и функциями, которые для обращения к ним требуют создание объектов.

Для знакомства с возможностям классов в C++ обернём сделаем тип для работы с целым числом как с объектом.
Что попадёт в создаваемый тип:

1. Поле типа **int** – 1 штука.
2. Операции для работы с объектом:
   - получить значение объекта;
   - изменить значение объекта;
   - получить модуль значения объекта;
   - получить противоположное по знаку значение объекта;

Создаём класс **Integer**

```
1 class Integer
2 {
3   int val;
4 };
```

По умолчанию, все поля и методы класса — закрыты, доступ к ним из кода не получить:

```
1 Integer iobj;
2 // Не пройдёт компиляцию:
3 iobj.val = 101;
```

И это **главное** отличие ключевого слова **class** от **struct**.

Для контроля доступности используются *ключевые слова* **public** (открытое поле/метод) и **private** (закрытое поле/метод). Предыдущее определение класса **Integer** эквивалентно следующему:

```
1 class Integer
2 {
3 private:
4   int val;
5 };
```

Добавляем методы для предоставления задуманных операций. Для того, чтобы ими пользоваться они должны попасть в *открытую часть* определения класса:

```
1  class Integer
2  {
3  public:
4    void value(int new_val); // установить значение
5    int value();     // получить значение объекта
6    int abs();       // модуль текущего значения
7    int negative(); // противоположное значение
8
9  private:
10   int val;
11 };
```

Осталось определить объявленные методы. Для этого C++ предоставляет два способа:

1. тело метода внутри определения класса;

```
1 class Integer
2 {
3 public:
4   void value(int new_val)
5   { val = new_val; }
6   ...
7 };
```

2. определение метода отдельно от класса:

```
1 void Integer::value(int new_val)
2 {
3   val = new_val;
4 }
```

Плюс первая особенность методов — они имеют прямой доступ к любым полям класса.

Полное определение всех методов:

```
12 void Integer::value(int new_val)
13 {
14   val = new_val;
15 }
16
17 int Integer::value()
18 {
19   return val;
20 }
21
22 int Integer::abs()
23 {
24   return abs(val);
25 }
26
27 int Integer::negative()
28 { return -val; }
```

Поскольку три из четырёх методов не меняют состояние объекта, к ним можно добавить *квалификатор* **const**

```
1  class Integer
2  {
3  public:
4    void value(int new_val);
5    int value() const;
6    int abs() const;
7    int negative() const;
8
9  private:
10   int val;
11 };
```

Теперь компилятор будет проверять, не меняется ли поле класса в таких методах.

И определение всех методов чуть модифицируется:

```
12 void Integer::value(int new_val)
13 {
14   val = new_val;
15 }
16
17 int Integer::value() const
18 {
19   return val;
20 }
21
22 int Integer::abs() const
23 {
24   return abs(val);
25 }
26
27 int Integer::negative() const
28 { return -val; }
```

Пример использования:

```
1 Integer num;
2 num.value(12);
3 printf("first value is %d\n", num.value());
4
5 Integer inum;
6 inum.value(-155);
7 printf("second value is %d\n", inum.value());
8 printf("          abs is %d\n", inum.abs());
9 printf("    opposite is %d\n", inum.negative());
```

Немного вспомним про *присвоение начальных значений*.

```
1 Integer num1;        // (1)
2 Integer num2{};      // (2)
3 Integer num3{55};    // (3), compile error
4 Integer num4 = -3;   // (4), compile error
```

Случаи **(3)-(4)** — просто не компилируются сейчас, работоспособен только первый способ.

До того, как заставить работать всё, ещё одна возможность для *составных типов* в C++ — задание для полей **значений по умолчанию**:

```
1  class Integer
2  {
3  public:
4    void value(int new_val);
5    int value() const;
6    int abs() const;
7    int negative() const;
8
9  private:
10   int val = 10;
11 };
```

И тогда:

```
1  Integer num1;      // num1.value() == 10
2  Integer num2{};    // num2.value() == 10
```

Концепция **конструктора** для класса: специальный метод, который позволяет устанавливать состояние объекта в момент его создания (то есть, при создании переменной конкретного класса появляется возможность присваивать конкретные значения закрытым полям).

В C++ **конструкторами** являются *методы класса*, имя которых **совпадает с названием класса** и которые **не возвращают никакого значения**.

Для любого объекта конструктор может быть вызван только **единожды**.

Для любого составного типа (**class** или **struct**) C++ предоставляет **неявный конструктор по умолчанию**, который инициализирует объект в соответствии с общими правилами.

Пусть конструктор выйдет на свет:

```
1  class Integer
2  {
3  public:
4    Integer(int other) { val = other; }
5
6    // other methods...
7
8  private:
9    int val;
10 };
```

Пусть конструктор выйдет на свет:

```
1  class Integer
2  {
3  public:
4    Integer(int other) { val = other; }
5
6    // other methods...
7
8  private:
9    int val;
10 };
```

И тогда,

```
1 Integer num1;       // (1), compile error
2 Integer num2{};     // (2), compile error
3 Integer num3{55};   // (3)
4 Integer num4 = -3; // (4)
```

Два способа – заработали, два – сломались. Not bad.

Почему два способа создания объектов класса перестали работать?

- Пока не было определено ни одного конструктора, классу C++ добавлял **неявный конструктор по умолчанию**
- Если определён вручную хотя бы один конструктор, неявный конструктор по умолчанию не добавляется в класс
- Для возможности определять переменные класса без конкретных значений нужно определить перегруженный метод конструктора без аргументов (если логика позволяет)

Возращение конструктора без параметров:

```
1  class Integer
2  {
3  public:
4    Integer(): val{0} {}
5    Integer(int other): val{other} {}
6
7    // other methods...
8
9  private:
10   int val;
11 };
```

И тогда,

```
1 Integer num1;       // (1) num1.value() == 0
2 Integer num2{};     // (2) num1.value() == 0
3 Integer num3{55};   // (3) num1.value() == 55
4 Integer num4 = -3; // (4) num1.value() == -3
```

И всё бы хорошо, обернули целое число в собственный тип, работаем в объектном стиле. Да вот только простейшие математические операции не поддерживаются:

```
1 Integer num1 = 155, num2 = 304;
2 num1 = num2;    // Ok
3 num2 = 155;     // Ok
4
5 num1 < num2;    // compile error;
6 num1 == num2;   // compile error;
7 num1 + num2;    // compile error;
8 num1 * num2;    // compile error;
```

А ведь хочется, для некоторых типов.
И для выполнения подобных трюков C++ предоставляет механизм **перегрузки операторов** для составных типов.

Перегрузка операторов — возможность использовать собственные типы в качестве операндов конкретных операторов. Реализуется с помощью определения специальных *методов* или *свободных функций*, имя которых состоит из ключевого слова **operator** и символа оператора.

Для класса **Integer** определим нужные операторы:

```
1  Integer operator+(Integer lhs, Integer rhs)
2  { return Integer{lhs.value() + rhs.value()}; }
3
4  Integer operator-(Integer lhs, Integer rhs)
5  { return Integer{lhs.value() - rhs.value()}; }
6
7  Integer operator*(Integer lhs, Integer rhs)
8  { return Integer{lhs.value() * rhs.value()}; }
9
10 Integer operator/(Integer lhs, Integer rhs)
11 { return Integer{lhs.value() / rhs.value()}; }
12
13 bool operator==(Integer lhs, Integer rhs)
14 { return lhs.value() == rhs.value(); }
15
16 bool operator<(Integer lhs, Integer rhs)
17 { return lhs.value() < rhs.value(); }
```

И тогда всё заработает:

```
1 Integer num1 = 155, num2 = 304;
2 num1 = num2;    // Ok
3 num2 = 155;     // Ok
4
5 num1 < num2;    // Ok
6 num1 == num2;   // Ok
7 num1 + num2;    // Ok
8 num1 * num2;    // Ok
```

Конечно, для всех случаев нужно перегружать *ещё больше* операторов.

Поля или методы класса могут быть **статическими**. Для примера,

```
1  #include <climits>
2
3  class Integer
4  {
5  public:
6    static const int MAX = INT_MAX;
7    static const int MIN = INT_MIN;
8    // other methods and fields ...
9  };
10
11 printf("Integer objects can hold values "
12        "in range [%d; %d]",
13        Integer::MIN, Integer::MAX);
```

Статические поля создаются один раз и не пренадлежат никакому объекту класса.

Связь структур и классов: структуры и классы в C++ — это одинаковые сущности. Единственное отличие приведено на **11** слайде.

Общий совет

Используйте **структуры** исключительно в смысле языка C – как группировку значений других типов данных в открытых полях. Допустимо добавление конструкторов (или использование значений по умолчанию для полей). Если объектам создаваемого типа нужны методы — стоит предпочесть класс вместо структуры.

Прежде, чем идти дальше

Начиная со стандарта **C++11** существует специальная форма цикла **for** для перебора всех элементов составных объектов, предоставляющих последовательный доступ к ним[элементам]. Именуется как *for-range* или диапазонный цикл **for**).
Его общая форма:

```
for (<имя_переменной> : <объект>) {
  // работа с переменной
}
```

Работает следующим образом: **каждый** элемент составного объекта, начиная с первого, **присваивается** указанной переменной и выполняется итерация цикла. Типы переменной и элементов **должны совпадать**. Тип переменной различается в зависимомти от сценариев использования.

Прежде, чем идти дальше

Для примера, печать элементов массива.

```
1 const size_t SZ = 4;
2 double rates[SZ] = { 1.1, 2.2, 5.2, 6.5 };
3
4 for (double r : rates) {
5   printf("%.3f ", r);
6 }
7 printf("\n");
```

Прежде, чем идти дальше

Если нужно изменение элементов – используется **ссылка на lvalue**. Например, возведение всех значений массива в квадрат.

```
1  const size_t SZ = 4;
2  int scores[SZ] = { 3, 5, 8, 12 };
3
4  for (int& score : scores) {
5    score *= 2;
6  }
7
8  for (int sc : scores) {
9    printf("%d ", sc);
10 }
11 printf("\n");
```

В этом случае копирования элементов массива не происходит.

Прежде, чем идти дальше

Если задача состоит в том, чтобы избежать копирования при переборе элементов массива циклом *for-range* – можно использовать **константную ссылку**.

```
1  for (const char& symb : "Sweet dreams happens ↩
       soon!") {
2    if (symb != ' ') {
3      printf("[%c]", symb);
4    } else {
5      printf("%c", symb);
6    }
7
8    if (symb == 'e') {
9      symb = 'E';  // compile error
10   }
11 }
12 printf("\n");
```

Прежде, чем идти дальше

Циклом *for-range* можно пользоваться и при работе с многомерными массивами.

```
1  using Matrix4x4 = int[4][4];
2
3  Matrix4x4 matr1 = { {1, 2, 3, 4}, {9, 8, 7, 6},
4      {5, 1, 5, 9}, {2, 3, 3, 2} };
5
6  for (auto row : matr1) {
7    for (int elem : row) {
8      printf("%d ", elem);
9    }
10   printf("\n");
11 }
```

Прежде, чем идти дальше

Ограничение цикла *for-range* для стандартных статических массивов состоит в том, что для его работы нужно, чтобы компилятор знал размер используемого массива. А информация об размере пропадает при передаче статического массива в функции. Но и эту ситуацию C++ позволяет немного сгладить с помощью использования ссылок в качестве параметров функций.

Прежде, чем идти дальше

Как только массив передаётся в некоторую функцию по ссылке, информация о его размере доступна компилятору и внутри этой функции.

```
1  void init_by_zeroes(int (&arr)[10])
2  {
3    for (int& elem : arr) {
4      elem = 0;
5    }
6  }
7
8  int arr1[10] = {1, 2, 3, 4, 5, 6};
9  int arr2[] = {5, 4, 3, 2, 1};
10
11 init_by_zeroes(arr1);
12 // init_by_zeroes(arr2); // compile error
```

Прежде, чем идти дальше

И пример с двумерным массивом

```
1  void sequence_init(Matrix4x4 matr,
2                     int start_value)
3  {
4    int add = 0;
5    for (auto& row : matr) {
6      for (int& elem : row) {
7        elem = start_value + add;
8        add++;
9      }
10   }
11 }
12
13 Matrix4x4 mtr;
14 sequence_init(mtr);
15
16 for (auto row : mtr) {
17   for (int elem : row) {
18     printf("%d ", elem);
19   }
20   printf("\n");
21 }
```