X

C++: что дальше

В данной лекции совершим обзор возможностей, которые вас ждут далее в C++ в следующем семестре. Упор будет идти на общие идеи и концепции, а не на синтаксис и перечисление всех правил самого языка программирования.

C++: ООП, идея I

И первое, с чего начнём — это 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 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.

Ещё раз конкретизируем две главные концепции объктов:

- возможность *скрыть* внутреннее состояние объекта;
- возможность определить набор «сообщений», на которые объект в состоянии «отвечать».

C++: ООП, идея IV

На практике, в большинстве языков программирования, предлагающих синтаксические конструкции для реализации ООП, обе эти концепции реализованы. В том числе, и в C++. Не везде употребляется именно такая терминология, но суть остаётся той же. Конечно, особенности языков программирования накладывают определённые ограничения. Например, в C++ нет ограничения на исключительное общение объектов между собой посредством сообщений: посылая некое сообщение объекту в ответ возвращается вполне себе конкретное значение какого-нибудь типа.
И основной момент, почему ООП важно: большая часть стандартной библиотеки C++ написана в этом стиле.

Рассмотрим простой пример взятия модуля от целого числа:

```
1 int val = -7;
2 int abs_val = abs(val);
3 printf("|%d| = %d\n", val, abs_val);
```

Процедурный код, вся работа делается через функцию **abs**. А теперь представим, что в языке программирования есть тип **Integer**, который позволяет создавать «объект» (в терминах ООП) целого числа. И изменять его значение можем только через посылку сообщений. Это можно выразить следующим псевдокодом[1]:

```
1 Integer i_obj;
2 i_obj <- value, -17;
3 i_obj <- value;
4 i_obj <- abs;
5 i_obj <- negative;
```

[1] **псевдокод** в том смысле, что рассматриваемых в примере конструкций не существует в C++

C++: ООП, идея сообщений I

Разберём пример построчно.

1. Для начала, просто создаём рассматриваемый объект придуманного типа.

```
1 Integer i_obj;
```

i_obj — переменная, связанная с созданным объектом.

2. В данной и следующих строках демонстрируется идея посылки сообщений. Например, для установки конкрентного значения объекта посылаем сообщение **value** объекту **i_obj** и передаём аргументом число **-17**:

```
2 i_obj <- value, -17;
```

3. Для того чтобы получить значение, хранящиеся в объекте, просто посылаем сообщение **value** без аргументов:

```
3 i_obj <- value;
```

C++: ООП, идея сообщений II

4. И, в стиле ООП, получаем модуль хранимого значения через сообщение **abs**

```
4 i_obj <- abs;
```

5. Просто для закрепления идеи, ещё одно сообщение **negative**, возвращающее противоположное по знаку значение (по отношению к сохранённому):

```
5 i_obj <- negative;
```

Сами по себе «сообщения» можно рассматривать как некоторые функции, в которых содержатся конкретные инструкции для выполнения ожидаемых действий (получить значение, установить значение, получить модуль целого числа и т.п.). Принципиальное отличие их от тех функций, что успели рассмотреть в C++ — это то, что они вызываются для конкретных переменных некоторого типа.
Выше в примере конструкция со стрелкой, обращённой влево, была выбрана только для демонстрации идеи посылки сообщений: есть название сообщения и есть объект, который их принимает. В C++ и множестве других языков программирования (в том числе, широко используемых в настоящее время: Java, C#, Python, Javascript и прочие) для работы с объектами был выбран синтаксис[2], аналогичный получению полей структурного объекта.

[2]Чтобы увидеть альтернативный синтаксис, можно написать в поисковике «Objective C» и посмотреть примеры кода на этом языке.

А именно,

```
1 Integer i_obj;
2 i_obj.value(-17);
3 i_obj.value();
4 i_obj.abs();
5 i_obj.negative();
```

Теперь это валидный код на C++ с точностью до определения самого типа **Integer**. Как можно убедиться, общая идея «посылки сообщений» в языке реализована через механизм вызова функций, привязанных к переменной конкретного типа. Вторая и третья строчки демонстрируют, забегая далеко вперёд, что перегрузка работает и в данном случае.

По итогам развития концепции объектно-ориентированного программирования в области компьютерных наук, на практике ведущими реализациями ООП в языках программирования стали: ООП, основанное на **классах** и ООП, основанное на **прототипах**. В C++ с точки зрения поддержки средствами языка присутствует только классовое ООП.
Разница по смыслу между двумя подходами в том, что в случае с классами создаётся спецификация нового типа, а затем через этот тип создаются конкретные объекты в коде. В прототипном ООП новый объект создаётся с использованием *уже существующего* объекта (за подробностями этого подхода — в языки типа Javascript или Lua).

Формальное определение **класса**

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

И сразу ещё одно определение, касающееся терминологии.

Определение **методов**

Функции в C++, которые применяются **только** к объектам некоторого типа (см. пример на 11 слайде), получили название **методов**.

Термин введён для смыслового (семантического) различия между обычными функциями (иногда применяется термин — *свободные функции*) и функциями, которые для обращения к ним требуют создание объектов.

Проименованные типизированные элементы выше в определении — это те же самые поля, которые определяли при рассмотрении структурных типов.
Ограничение доступа говорит в C++ о том, что при использовании в коде объектов некоторого класса, невозможно будет получить напрямую значение группы полей или вызвать набор методов, если доступ к ним запрещён на уровне спецификации (определения) класса.
Для определения класса исползуется ключевое слово **class**.

Для рабочего примера на C++, приведём определение класса **Integer** для того, чтобы код со слайда 11 стал рабочим.

```
1  class Integer
2  {
3  public:
4    int abs() const { return abs(val); }
5    int negative() const { return -val; }
6    int value() const { return val; }
7    void value(int new_val) { val = new_val;   }
8
9  private:
10   int val = 0;
11 };
```

В определении класса **Integer**:

- метки **public:** (доступ разрешён) и **private:** (доступ запрещён) отвечают за управление доступа к полям/методам. На практике это означает, что следующий код не компилируется — нет прямого доступа к полю **val**:

```
1 Integer my_int;
2 my_int.val = 101; // Ошибка компиляции
```

- полю **val** предоставлено *значение по умолчанию*. Каждый раз, когда будет создаваться переменная представленного типа, в её поле **val** будет сохранено значение **нуль**;
- зачем в определении методов присутствует **const** — узнаете или самостоятельно, или в следующем семестре.

На этом закончим обсуждение конкретных деталей реализации классов в C++. Всё это будет у вас в следующем семестре. Целью данной части лекции было дать введение в концепцию ООП: так или иначе, это очень значительная часть языка C++, в том числе обеспечившая ему значительную популярность в качестве инструмента для разработки ПО.

C++: ООП, в следующем семестре I

- Вы детально разберётесь в том, что означают термины **инкапсуляция**, **сокрытие данных/методов**, **наследование** и **полиморфизм**. Для затравки, пример со слайда 11 полностью демонстрирует идею инкапсуляции применительно к C++.
- Узнаете про **конструкторы** и **деструктор** классов. Если кратко, то это специальные методы, которые можно определять в создаваемых пользовательских составных типах. Целью *конструкторов* является определение действий, которые будут происходить в *момент создания объекта* некоторого типа. *Деструктор* нужен для того, чтобы определять действия, которые будут происходить в тот момент, когда объект удаляется (напомним, это либо выход из области видимости, либо явное удаление динамического объекта).

- Рассмотрите такую особенность C++ как **перегрузку операторов**. Для пользовательских составных типов данных с помощью определения специальных функций (или методов) можно определить подавляющее большинство операторов языка (==, !=, >, <, арифметические операторы, обращение по индексу и прочие). Для примера без подробных объяснений, для класса **Integer** можно написать пару функций:

```
1 bool operator==(const Integer left,
2                 const Integer right)
3 { return left.value() == right.value(); }
4 bool operator!=(const Integer left,
5                 const Integer right)
6 { return !(left == right); }
```

и для объектов класса будет доступна проверка на равенство или неравенство:

```
1 Integer obj1, obj2;
2 if (obj1 == obj2) {
3   printf("Yes, objects have the same value");
4 }
```

- Изучите концепцию и реализацию *функциональных объектов* (сокращённый термин — *функторов*). Пример будет приведён далее.
- Познакомитесь со **статическими** полями и методами классов.

В стандартной библиотеке C++ множество вспомогательного функционала для написания прикладных программ реализовано с использованием концепции ООП. Это и класс для работы со строками, и динамический массив (с автоматическим выделением памяти под новые элементы), и более алгоритмически провдвинутые генераторы псевдослучайных чисел (по сравнению с **rand**), и набор классов для осуществления операций ввода/вывода (в дополнение к уже рассмотренным возможностям заголовочного файла <**cstdio**>).
Для практического применения каждого класса просматривается документация/справка по стандартной библиотеке, определяется набор нужных в конкретной задаче методов/полей и пишется программный код с использованием подходящих объектов.

Для примера, более удобная работа со строками в C++ может быть организована с помощью класса **std::string** из заголовочного файла **<string>**.

```
1  string s1 = "Example one", s2;
2  s2 = "Example two";
3  printf("First: %s\nSecond: %s\n",
4         s1.c_str(), s2.c_str());
5
6  string s3 = "Another string example";
7  printf("%c - %c - %c\n", s3[0], s3[2], s3[4]);
8
9  print("String s3 by symbol:\n");
10 for (char sym : s3) { printf("%c ", sym); }
11 printf("\n");
12
13 string s4 = s1 + ";" + s2;
14 printf("s4: <<%s>> has %zu symbols\n",
15        s4.c_str(), s4.length());
```

В C++ реализован и ввод/вывод через классы: консольный, файловый, строчный. Применяется та же концепция потока, только каждый поток представляет уже не указатель на структуру плюс набор функций, а объект типа из стандартной библиотеки и набор методов для получения результата. Так, за консольный ввод/вывод отвечает заголовочный файл <**iostream**>. При его подключении, в программе становятся доступны три объекта:

- **std::cout** — отвечает за вывод информации в стандартный поток вывода (консольный вывод). Аналог **stdout** из <**cstdio**>;
- **std::cin** — стандартный поток ввода, аналог **stdin**;
- **std::cerr** — стандартный поток вывода для сообщения об ошибках, аналог **stderr**.

Базовая работа с потоками из **<iostream>** реализована с помощью перегрузки операторов сдвига «>>» и «<<». В данном случае, первый из них означает ввод значения из потока в переменную, второй — вывод значения переменной в поток вывода.

```
1 int i_num;
2 std::cout << "Enter integer number: ";
3 std::cin >> i_num;
4
5 std::cout << "You entered: " << i_num << "\n";
```

На первый взгляд, может показаться упрощением операций ввода/вывода по сравнению с функциями **printf/scanf** из **<cstdio>**.

Но не всегда. Для примера, вывести число с плавающей точкой с точностью ровно в два знака после запятой в поле вывода из 10 символов с выравниванием по левому краю.

```
1 double real = 1234.87634;
2
3 printf("{{%-10.2f}}\n", real);
4 // vs
5 std::cout << "{{" << std::fixed
6           << std::setprecision(2)
7           << std::left << std::setw(10)
8           << real << "}}\n";
```

А **printf** и куча операторов сдвига применительно к **cout** делают одно и то же[3]. Потоковый ввод/вывод в файлы через объекты определён в <**fstream**>. А ещё определены строковые потоки — <**sstream**>.

[3] кому интересно, справка по консольному вводу/выводу через потоковые объекты: https://github.com/posgen/OmsuMaterials/wiki/Format-output

И третий пример использования ООП в стандартной библиотеке — генераторы псевдослучайных чисел. В дополнении к функции **rand** из <**cstdlib**>, определён заголовочный файл <**random**> который определяет дополнительные ГПСЧ (и ещё некоторые сущности, касательно генераторов). Необходимость в них может возникнуть, поскольку в некоторых задачах недостаточно периода генератора псевдослучайных чисел, который скрывается за функцией **rand** (например, некоторый класс задач криптографии).

Сам пример (предполагается, что файл <**random**> подключён соответствующей директивой):

```
1 std::mt19337 gnr{time(nullptr)};
2
3 printf("Max number: %zu\n", gnr.max());
4 printf("Min number: %zu\n", gnr.min());
5
6 printf("Rand number #1: %zu\n", gnr() );
7 printf("Rand number #2: %zu\n", gnr() );
8 printf("Rand number #3: %zu\n", gnr() );
```

Строка **(1)** демонстрирует вызов *конструктора* для создаваемого объекта. **gnr** представляет собой умопянутый ранее *функциональный объект*. Это демонстрируется строками с **шестой** по **восьмую**: для получения случайного числа применяем оператор «круглые скобки» к переменной **gnr**. Это похоже на вызов функции, отсюда и название вида объектов.

C++: шаблонное программирование

C++ предоставляет механизм **шаблонов** (templates): специального синтаксиса для задания функций или составных типов без явной конкретизации используемых типов. Типы в данном механизме выступают в качестве *параметров*. Реализацию конкретных функций/типов делает уже компилятор.

Например, обмен значениями между двумя переменными:

```
1 template <typename Type>
2 void my_swap(Type& left, Type& right)
3 {
4   Type tmp = left;
5   left = right;
6   right = tmp;
7 }
```

Здесь **Type** выступает как *параметр типа*. Для того чтобы компилятор **инстанцировал** (создал реализацию) данной функции, необходимо сделать её вызов в программе.

Использование шаблонной функции:

```
1 int i1 = 10, i2 = -5;
2 my_swap(i1, i2);
3
4 std::string s1 = "uuu", s2 = "eee";
5 my_swap(s1, s2);
6
7 int *p1 = &i1, *p2 = &i2;
8 my_swap<int*>(p1, p2);
```

Объяснения и синтаксиса, и правил вывода типов получите в следующем семестре.

Можно создавать и шаблонные типы:

```
1 template <typename T1, typename T2>
2 struct MyPair
3 {
4   T1 first;
5   T2 second;
6 }
```

Шаблон структуры, которая может хранить значения двух произвольных (в том числе и различных) типов.

```
1 MyPair <int, double> p1 = {111, 3.14};
2 printf("pair: {%d, %f}\n", p1.first, p1.second);
```

Как и в примере с шаблонной функцией, компилятор сам определит конкретную структуру, первое поле которой будет иметь тип **int**, а второе — **double**.

Шаблоны также интенсивно используются в стандартной библиотеке C++. В частности, в **<algorithm>**. Приведём пример использования заголовочного файла **<limits>** — в нём через класс **numeric_limits** определён способ получения граничных значений фундаментальных типов.

```
1  printf("[double] min: %f, max: %f\n",
2         std::numeric_limits<double>::min(),
3         std::numeric_limits<double>::max());
4
5  printf("[int] min: %d, max: %d\n",
6         std::numeric_limits<int>::min(),
7         std::numeric_limits<int>::max());
8
9  printf("[size_t] min: %zu, max: %zu\n",
10        std::numeric_limits<size_t>::min(),
11        std::numeric_limits<size_t>::max());
```

Пример, в дополнение, демонстрирует использование *статических* методов класса **numeric_limits**.

C++: обработка ошибок

Рекомендую вам в следующем семестре на данной дисциплине попросить преподавателя как минимум одну тему посвятить такому вопросу, как обработка ошибок в программах. Учитывая возможности C++, его стандратная библиотека содержит различные стратегии для обработки. Например, одну стратугию с вами неявно видели в теме о файловом вводе/выводе: часть функций возвращают специальное значение, если произошла ошибка (касательно файлов, это значение задавалось макросом **EOF**).
Сам язык предоставляет ещё один механизм для сообщения об ошибках и их обработке — это механим **исключений**.

C++: обработка ошибок

Образно говоря, **исключение** — это объект произвольного типа, который «бежит» из места своего возникновения до функции **main**. По пути этот объект может быть пойман специальным обработчиком. И тогда говорят, что исключение было обработано. Если же исключение не встретило ни одного обработчика (в том числе и внутри **main**), то оно немедленно прекращает работу программы с сообщением об ошибке. Синтаксис и примеры оставляем на следующий семестр.