Лекція 6. Класи в C++.

Версія 20 березня 2023 р.

Анотація

Ця лекція знайомить з одною з найважливіших концепцій мови C++, а саме, абстрагування об'єктів та процедур в логічні блоки за допомогою класів.

Зміст

1 Власні типи

При розробці програм, таких, що є зрозумілими для читача, а також таких, що зменшують вірогідність неприємностей завдяки правильному дизайну, особливо якщо використовується строго типізована мова, дуже важливо грамотно працювати з типами даних. Це означає, (i) використання тих типів, які найкраще підходять під задачу, (ii) уникнення надмірного перевикористання загальних типів (таких як `std::string`, `int`), які не говорять користувачу нічого про суть переданого значення, і які можуть дати помилку, коли ці дані переплутаються,[1] і навпаки (iii) використання 'доречних'

[1]Навіть гірше – коли ніякої помилки немає, натомість ми маємо нестандартну поведінку, яку дуже важко пояснити.

типів, а саме таких, що дають уявлення про природу даних, відокремлюють їх від інших, та не несуть з собою більше інформації, ніж потрібно.

Наприклад, бібліотека `std::chrono` задає велику кількість типів для запису різної протяжності часу. Таким чином, сігнатура функції, яка приймає проміжок часу, одразу скаже нам в яких одиницях він вимірюється, а також не дозволить передати значення в інших одиницях.[2]

```
void waitFor(std::chrono::seconds timeToWait) {
   // implement waiting mechanism
}
```

Звичайно, можна було б для цих цілей використовувати звичайний `int`, але не будемо повторювати недоліки цього підходу.

Є три основних підходи для створення своїх власних типів в C++:

1. псевдоніми для існуючих типів,
2. `enum` для набору кінцевої кількості цілих значень,
3. класи (структури) для повного управління всіма властивостями об'єктів.

1.1 Псевдоніми

В C++ існує ключове слово `typedef`, а також, починаючи з C++11, using, що дозволяє задати свою назву для існуючого типу. Це може бути корисно з трьох причин:

1. Нова назва часто банально більш коротка, її легше запам'ятати та писати,
2. Вона краще описує роль та суть інформації, що зберігається всередині,
3. Якщо базовий тип змінюється, то його достатньо змінити в одному місці – в дефініції псевдо.

Наприклад, ви зберігаєте профілі користувачів в структурі `std::map`, де ключ – це їх ім'я, а значення – ще одна `std::map`, що тримає інформацію по конкретній людині. Параметр профілю задається рядком, а значення допускається в форматі `std::string`, або числовому `int`. В такому випадку, одна з двох наступних декларації значно покращить якість вашого коду:

```
typedef std::map<std::string, std::map<std::string,
    std::variant<std::string, int>>> UsersTable;
using UsersTable = std::map<std::string,
    std::map<std::string, std::variant<std::string, int>>>;
```

[2] Або ж автоматично трансформує значення, якщо така операція задана.

1.2 enum

Перечислювачі (enumerators/enumerations) також призвані полегшити там життя та зробити нашу програму більш зрозумілішою для людини. Вони задають свій тип, але такий, що може приймати тільки одне з дозволенних значень. Вони задаються рядком, як назви замінної, але мають під собою ціле значення – за замовченням `int`, або `unsigned int`, або таке, що в них конвертується[3]. Наприклад, подивимося на функцію

```
void DescribeColor(int code) {
   std::cout << "This color is: ";
   switch(code) {
      case 0:
         std::cout << "red.\n";
         break;
      case 1:
         std::cout << "green.\n";
         break;
      case 2:
         std::cout << "blue.\n";
         break;
   }
}
```

Ніщо не заважає нам передати до функції `DescribeColor` число 100, або -1, в такому випадку вона наче коректно відпрацює, але на екрані буде неповне повідомлення. Простим запобіжником було б додати випадок `defult`, який би повернув помилку, що говорить про те, що отриманий код не відповідає кольору, але тільки після того, як частина повідомлення була виведена на екран. Набагато зрозуміліше та безпечніше використовувати для цього `enum`:

```
enum RGB {
   RED,
   GREEN,
   BLUE
}
void DescribeColor(RGB color) {
   std::cout << "This color is: ";
   switch(color) {
      case REF:
         std::cout << "red.\n";
```

[3]Відлік починається з нуля.

```
            break;
        case GREEN:
            std::cout << "green.\n";
            break;
        case BLUE:
            std::cout << "blue.\n";
            break;
    }
}
```

Тепер з сігнатури функції там зрозуміло, що треба передати не будь-яке число, а об'єкт типу `RGB`, або таке число, що може бути перетворене на `RGB`. У такому разі, ми ще не заходячи в функцію автоматично отримаємо помилку, про то, що передане значення не відповідає жодному зі значень `enum`. Наостанок зазначимо, що є гарною практикою використовувати `enum class`, або ж `enum struct`, щоб назва такого перерахування створювала свій простір імен. Таким чином, якщо змінити наш попередній приклад і написати

```
enum class RGB { RED, GREEN, BLUE }
```

то звертатися до значень необхідно за допомогою оператора `::`, а саме

```
RGB redColor = RGB::RED;
auto blueColor = RGB::BLUE;
```

1.3 Класи

Класи (`class`), або ж структури (`struct`), дозволяють нам створювати свої власні об'єкти, зі своїми структурами даних[4], та методами, що задани для них. Детальніше ми поговоримо про них далі.

2 Синтаксис класів в C++

Цитуючи, перекладаючи, з Stroustrup 2013, класи в C++ є інструментом для створення нових типів, які можна використовувати з такою самою зручністю, що і вбудовані типи. Крім того, похідні класи та шаблони дозволяють програмісту виражати взаємозв'язки між класами та використовувати

[4]Що звісно складаються з базових типів, структур і контейнерів стандартної бібліотеки, або інших користувацьких типів.

переваги таких взаємозв'язків.[5] Тип – це конкретне представлення концепції (ідеї, поняття тощо). Програма, яка задає та використовує типи, що тісно пов'язані з концепціями реалізації, як правило, легша для розуміння, міркувань і модифікації, ніж така, яка цього не робить.

Дефініція, або ж декларація, класу починається к ключового слова `class`, або `struct`. Результат вони мають однаковий, а відрізняються лише режимом доступу до атрибутів за замовченням – `private` для `class`, та `public` для `struct`:

```
class MyClass {
   void DoAction();
   int state;
}

void clientFunction(MyClass arg) {
   arg.DoAction(); // error, not part of public interface
   std::cout << "Current state: " << arg.state; // same error
}
```

у той час, як ці операції були б доступні, якщо б ми задекларували `MyClass` як `struct`. Звичайно, клас, який не має публічних атрибутів (полів) чи функцій (методів) не дуже корисний на практиці, тому тіло найчастіше поділяється на блоки `private` та `public`. За такої умови, якщо не покладатися на поведінку за замовченням, то взагалі немає різниці між ключовими словами `class` та `struct`. На практиці, зазвичай, дотримується наступна конвенція: `struct` використовується для (зазвичай невеликих) простих типів, які зберігають набір даних з відкритим доступом, і не мають якоїсь специфічної логіки. Наприклад,

```
struct Student {
   std::string name;
   std::string group;
   int graduationYear;
   std::map<std::string, double> marks;
}

void printStudentInfo(Student student) {
   std::cout << "Student " << student.name
          << " (grad. " << student.graduationYear << "), "
          << "Group " << student.group <<
          << ", has the following marks:\n";
```

[5]Похідні класи описуються далі, шаблони – в наступних лекціях.

```
    for (const auto& [course, mark] : student.marks) {
        std::cout << course << ": " << mark << std::endl;
    }
}
```

Як вже було зазначено, за допомогою `private` та `public` можна відділяти приватний (що є частиною внутрішньої імплементації) інтерфейс від публічного (як нашим класом можна користуватись ззовні). Наприклад, наступний клас задає прототип інтерфейсу[6] (без конкретної імплементації) для класу, що зчитує якісь дані з диску при створенні об'єкту, але замість того, щоб видати користувачу їх у вигляді, наприклад, контейнера, він зберігає їх всередині, даючи можливість отримати доступ до них по одному через публічні методи:

Лістинг 1: "Reader.h"

```
class Reader {
public:
    Reader(const std::string& filename); // creates object
        and loads data
    std::string getNextLine();
    std::string getPrevLine();

private:
    std::vector<std::string> data_;
    std::vector<std::string>::const_iterator currLine_;
}
```

Зауважте, часто, для імен приватних атрибутів використовується окремий формат – наприклад, ніжне підкреслення в кінці імені, або приставка `m_`.

Код з лістінгу 1 мусить знаходитись в файлі заголовку `Reader.h`, у той час, як його імплементація – у `Reader.cpp`, в якому, по перше, треба зробити `#include` відповідного заголовку, а по-друге, доступ до методів здійснюється через оператор `::` оскільки класи створюють свій простір імен:

Лістинг 2: "Reader.cpp"

```
#include "Reader.h"

Reader::Reader(const std::string& filename) {
```

[6]Для його позначення часто використовується абревіатура API (Application Programming Interface, промовляється Ей-Пі-Ай)

```
    data_ = LoadData(filename); // LoadData to be defined by
        author
    currLine_ = data_.cbegin();
}

std::string Reader::getNextLine() {
    if (currLine_ == data_.cend())
        return {};

    return *currLine_++;
}

std::string Reader::getPrevLine() {
    if (currLine_ == data_.cbegin())
        return {};

    return *currLine_--;
}
```

На кінець цієї секції, варто зазначити, (i) що методи, які не змінюють дані в полях класу варто позначати як `const` (див. тут),[7] а також, (ii) що конструктори часто пишуться з використанням member initializer lists, що запобігає створенню тимчасових об'єктів.

Література

- Stroustrup, Bjarne. The C++ programming language. Pearson Education, 2013 (Глава 16).
- CPPReference, Classes.

[7] В класі `Reader` з лістингу 1 та 2 методи `getNextLine` та `getPrevLine` не можуть бути `const`, оскільки змінюють ітератор поточного рядка.