Лекція 6. Класи в C++.

Версія 26 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, (і) використання

тих типів, які найкраще підходять під задачу, (ii) уникнення надмірного перевикористання загальних типів (таких як `int`, `std::string` тощо), які не повідомляють користувачу нічого про суть переданого значення, і які можуть дати помилку, коли ці дані переплутаються,[1] і навпаки (iii) використання 'доречних' 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

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

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

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>>>;
```

1.2 enum

Перечислювачі (enumerators/enumerations) також призвані полегшити там життя та зробити нашу програму більш зрозумілою для людини. Вони задають свій тип, але такий, що може приймати тільки одне з дозволенних значень, які 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 {
```

[3]Відлік починається з нуля.

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

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

2 Синтаксис класів в C++

2.1 Загальні відомості

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 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 сприйняття, розуміння 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`, сам доступ здійснюється за допомогою крапки:

```
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`.[6][7] 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:

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

[6] В теорії достатньо мати тільки окремий блок `public`, щоб все працювало як потрібно, але в реальних проектах покладатись на налаштування за замовченням вважається дурним тоном, тим паче це б означало, що блок `private` повинен буде зверху, а багато хто віддає перевагу ставити його в кінець. Більше того, часто ви можете побачити блоки `public` або `private`, що йдуть під ряд. З точки зору синтаксису C++ це не має жодної цілі, але для читача ми поділяємо API класу на логічні підблоки, які починаються одним з цих специфікаторів, так що його не треба шукати.

[7] Існує третій варіант доступу – `protected`, який веде себе як `private`, але такий, що дозволяє використання атрибутів під своїм захистом похідними класами від поточного.

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

   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[8] (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:
```

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

```
    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) {
    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, що методи, які не змінюють дані в полях класу варто[9] позначати як `const` (див. тут),[10].

[9] Сприймайте це як "обов'язково".

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

2.2 Конструктори

Конструктори – це спеціальні методи, яки викликаються при створенні об'єкту, як явно (explicitly), так і неявно (implicitly). Наприклад,

```
class Student {
public:
   Student();
   Student(const std::string& name);

private:
   // data members here
}

int main() {
   Student s1; // calls Student::Student()
   Student s2{"Bjarne"}; // calls Student::Student(const
       std::string& name)

   auto s3 = Student(); // calls Student::Student()
   std::vector<Student> s4 = {{"Bjarne"}}; // calls
       Student::Student(const std::string& name)
}
```

Якщо робота конструктора зводиться до копіювання чи запису даних всередину себе, то такі конструктори часто пишуться з використанням member initializer lists, що запобігає створенню тимчасових об'єктів та має більш стислий вигляд. Наприклад,

```
class Student {
public:
   explicit Student(const std::string& name);

private:
   std::string name_;
}

Student::Student(const std::string& name) : name_(name) { }
```

Зауважте, в цьому прикладі було використано ключове слово `explicit`, що не дозволяє робити автоматичний виклик конструктора, фактично, конвертацію типів, у цьому випадку з `std::string` у `Student`. Це важливо для запобігання потенційної небажаної поведінки, особливо, коли мова іде про

такі загальні універсальні типи як `std::string`.[11] Таким чином, перший з наступних виразів дасть помилку, а другий скомпілюється правильно:

```
Student s1 = "Bjarne"; // implicit
auto s2 = Student("Bjarne"); // explicit
```

2.3 Деструктор

Якщо конструктори створюють об'єкт, їх може бути декілька, та вони можуть мати скільки завгодно вхідних параметрів, то деструктор дозволяє нам втрутитись та додати свою логіку у процес знищення (вивільнення пам'яті) об'єкта, він може бути тільки один та не мусить мати вхідних аргументів, бо викликається автоматично. Як і конструктор, він не має вихідних параметрів, та задається за допомогою спеціального символа ∼:

```
#include <iostream>

class C {
public:
   explicit C(int);
   ∼C();
private:
   int num_;
};

C::C(int num) : num_(num) { }
C::∼C() {
   std::cout << "Object holding " << num_ << " is being
       destructed.\n";
}

int main() {
   C c1(1), c2(2);

   {
      C c3(3);
   }

   C c4(4);
```

[11] Тому що у `std::string` ви можете передати практично будь яку інформацію. Ми б були спокійніше, якщо б використовували свій тип, на кшталт `StudentName`, але не завжди створювання великої кількості своїх мікротипів є доцільним.

```
}
```

При виконанні програми, ми побачимо наступний вивід

```
Object holding 3 is being destructed.
Object holding 4 is being destructed.
Object holding 2 is being destructed.
Object holding 1 is being destructed.
```

Найчастіше, деструктори використовуються для вивільнення динамічної пам'яті, [12] закриття відкритих каналів з'єднань та файлів.

2.4 Методи класу

Методи класу – функції, котрі викликаються за допомогою об'єкту класу та оператора `.`, або, якщо це статичний метод, також і за допомогою назви класу на оператора `::`. Статичні методи – це функції, задані в просторі імені класу, які не мають прямого відношення до жодного конкретного об'єкту. Звичайним методам, в свою чергу, передається вказівник на об'єкт, який його викликав, який доступний за допомогою ключового слова `this`. Таким чином, ми можемо зчитувати стан об'єкта, або модифікувати його. Ба більше, це настільки поширена дія, що в C++ не обов'язково навіть писали слово `this` – компілятор сам розпізнає назви атрибутів цього класу. Отже, написання `data_` в тілі метода трактується як `this->data_`, так само виклик методів `doSomething()` – `this->doSomething()`.

Один з підвидів методів це так звані гетери (getters) та сетери (setters). Вони задають спосіб взаємодії з даними класу.[13] Вони можуть бути простими:

Лістинг 3: "Getters and setters"

```
class Student {
public:
   explicit Student(const std::string& name);
   ~Student() = default;

public:
   const std::string& name() const { return name_; }
   void setYear(unsigned year) { year_ = year; }
```

[12] Тобто тої, яка була отримана за допомогою ключового слова `new`.

[13] Якщо це взагалі потрібно – іноді користувачу не треба знати нічого про то як і які самі дані зберігає клас для того, щоб використовувати його для конкретних операцій.

```
private:
   std::string name_;
   unsigned year_;
};
```

або складнішими[14]

```
class Student {
// code omitted
   void setYear(unsigned year) {
      if (year > 6) {
         std::cerr << "Invalid year! Student to have up to
            six years of education!\n";
         return;
      }
      year_ = year;
   }

// code omitted
};
```

Майте на увазі, що давати можливість напряму модифікувати дані не є гарною практикою на практиці. Задля передбачуваності, варто ініціалізувати все в конструкторі і передати управління даними самому класу, якщо це можливо.

Важливою властивістю методів, яка є частиною їх сігнатури, це ключове слово `const`, яке ставиться в її кінці (див. `name()` в лістингу 3). Все, що вона робить, це говорить компілятору та користувачеві, що цей метод не може змінювати об'єкт, який його викликає. Тобто, різні виклики з однаковими вхідними параметрами повинні давати той самий результат.[15] Також, компілятор, як не дивно, не дозволить вам звертатись до інших, не `const` методів в тілі такої функції. Такий синтаксис допомагає на моменті компіляції відловити потенційні помилки, що могли би трапитись при виконанні програми. А саме, якщо ми отримали об'єкт як `const`[16], ми не можемо його змінювати, але ми можемо викликати його методи. І якщо при виконанні коду дійде до необхідності внести зміну в сам об'єкт, програма повинна завершитись помилкою. Натомість, ще на моменті розробки програми, ком-

[14]Зауважте, використання типу `unsigned` замість `int` говорить користувачу, що допускаються тільки невід'ємні числа. Це вже покращує дизайн, але `int` може конвертуватись в `unsigned`, таким чином, що -1 буде максимальним додатним числом, що вміщується в `unsigned`, а також туди входить нуль. Чи допоміг би нам тут якось `enum`?

[15]Якщо вся ця передбачуваність не руйнується глобальними змінними.

[16]Найчастіше це константне посилання.

пілятор дозволяє нам користуватись тільки тими методами, що помічені `const`, якщо сам об'єкт має ту саму властивість.

2.5 Члени класу

Членами (або атрибутами) класу можуть бути об'єкти будь якого типу, включаючи посилання, вказівники, константні, або ні. Коли це можливо, їх варто ініціалізувати прямо в декларації класу. Це зручно для читача, а також звільняє від необхідності створювати конструктор за замовченням, який тільки і робить, що присвоює ці початкові значення. Наприклад

```
class Student {
   public:
      // code omitted
   private:
      unsigned year_ = 1u;
      unsigned coursesCompleted_ = 0u;
}
```

У той час, для `struct` так робити не варто. Як було зазначено, вони найчастіше використовуються для зберігання інформації, тому значення має сенс ініціалізувати при створенні об'єкту за допомогою initializer list:

```
struct Student {
   unsigned year;
   unsigned coursesCompleted;
}

Student student{1, 5};
// student completes 5 more courses and graduates to year 2
student.coursesCompleted += 5;
student.year++:
```

3 Успадкування ти поліморфізм

Ви можете впізнати успадкування та поліморфізм як один з трьох стовпів об'єктно-орієнтованого програмування. Третій – це інкапсуляція, про яку вже багато було сказано в попередній секції, коли мова йшла про принципи зберігання, структурування та взаємодії з даними всередині об'єктів класу. Ця секція, в свою чергу, буде стосуватися успадкування та інкапсуляції – різних, але пов'язаних концепцій. Звичайно, це дуже широка тема, як і будь яка тема наших лекцій, тому цей матеріал є просто коротким вступом.

Успадкування дозволяє нам перевикористовувати структуру і функціонал в різних класах, встановлюючи логічні зв'язки між абстракціями. Похідний клас може користуватись всіма не приватними атрибутами базового класу, розширюючи його, або змінюючи їх поведінку. Наслідування буває трьох типів: `public` – таке, що зберігає всі специфікатори доступу, `protected` – перетворює всі публічні атрибути на `protected`, та `private` – який перетворює всі публічні і захищені члени на приватні. Останні два способи також накладають обмеження на поліморфне використання таких класів. Оскільки `protected` та `private` успадкування не входить в програму цього курсу, детальніше про них можна самостійно ознайомитись тут.

Синтаксис створювання класу `Derived`, який наслідує `Base`, виглядає наступним чином

```
class Derived : public Base {
   // Contents of the derived class
};
```

`Derived` буде мати доступ до всіх публічних та захищених методів `Base`, у той час маючи свої конструктори, деструктор та оператор присвоєння. Наприклад

```
class Book {
public:
   explicit Book(unsigned numPages) : numPages_(numPages) { }
   virtual ~Book() = default;

private:
   unsigned numPages_;
};

class ChildrenBook : public Book {
public:
   ChildrenBook(unsigned numPages) : Book(numPages) { }
};
```

3.1 Перевизначення функцій

Уявімо, що як і в попередньому прикладі, ми хочемо працювати з різними типами книг по-різному. Якби не існувало успадковування, одним з варіантів реалізації такого функціоналу біло б зберігання різновиду книги всередині одного з членів класу, наприклад

```
#include <iostream>

class Book {
public:
   enum Genre {
      GENERIC,
      TEXTBOOK,
      CHILDREN,
      NOVEL
   };

public:
   explicit Book(unsigned numPages) : genre_(GENERIC),
       numPages_(numPages) { }
   Book(Genre genre, unsigned numPages) : genre_(genre),
       numPages_(numPages) { }
   virtual ~Book() = default;

public:
   void describe() const;

private:
   Genre genre_;
   unsigned numPages_;
};

void Book::describe() const {
   std::cout << "This is a ";

   switch (genre_) {
      case GENERIC:
         std::cout << "generic book";
         break;
      case CHILDREN:
         std::cout << "children's book";
         break;
      case TEXTBOOK:
         std::cout << "textbook";
         break;
      case NOVEL:
         std::cout << "novel";
         break;
```

```
    }

    std::cout << " with " << numPages_ << " pages." <<
        std::endl;
}

int main() {
    Book myBook{Book::TEXTBOOK, 495};
    myBook.describe();

    return 0;
}
```

Натомість, ми можемо задати якусь функцію для базового класу, перезначивши її для похідних, таким чином роблячи наш код зрозумілішим, коротшим, а інтерфейс – зручнішим для використання:

Лістинг 4: "Book Base Class"

```
#include <iostream>

class Book {
public:
    explicit Book(unsigned numPages) : numPages_(numPages) { }
    virtual ~Book() = default;

public:
    virtual void describe() const;

protected:
    unsigned numPages_;
};

void Book::describe() const {
    std::cout << "This is a generic book with " << numPages_ <<
        " pages." << std::endl;
}

class TextBook : public Book {
public:
    explicit TextBook(unsigned numPages) : Book(numPages) { }
    void describe() const override;
};
```

```
void TextBook::describe() const {
   std::cout << "This is a textbook with " << numPages_ << "
       pages." << std::endl;
}

// similar for other kinds of Book

int main() {
   TextBook myBook{495};
   myBook.describe();

   return 0;
}
```

В цьому прикладі ви могли помітити цікаву особливість, яка пов'язана поліморфізмом, а саме, використання ключових слів `virtual` та `override`. Насправді, у цьому прикладі ми могли би їх опустити, і результат від цього б не змінився. Вони необхідні, якщо ми хочемо використовувати ці класи *поліморфно*, а саме, передавати посилання чи вказівник на похідний клас, там, де очікується базовий. Наприклад, замінимо `main()` на

```
int main() {
   Book* myBook = new TextBook{495};
   myBook->describe();

   delete myBook;
   return 0;
}
```

Оператор `new` повертає об'єкт типу `TextBook*`, який присвоюється змінній типу `Book*`, але це дозволено і працює так, як ми б очікували.

Без `virtual` та `override` при виконанні `myBook->describe()` викликалась би функція `describe()` базового класу `Book` (бо `myBook` це вказівник на `Book`), а на екрані з'явилося б

```
   This is a generic book with 495 pages.
```

Натомість, `virtual` у зв'язці з `override`[17] зробить так, що `myBook` викличе саме `TextBook::describe()` і ми побачмо правильне повідомлення. Така

[17]Насправді, достатньо використовувати лише `virtual`, але таке додаткове маркування є гарною практикою та роблять ваші наміри явними, також може запобігти помилкам у майбутньому.

поведінка зветься поліморфізмом часу виконання (runtime polymorphism). Рантайм C++ створює таблицю віртуальних функцій, яки можуть викликатись[18] через посилання та вказівники на базові класи.

Найчастіше цей принцип використовується при створенні інтерфейсів, про які ми поговоримо далі.

3.2 Абстрактні класи та інтерфейси

Повернемось ще раз до прикладу з лістингу 4, деякі методи, з який складається API класу, можуть не мати реалізації у базовому класі.[19] В такому випадку, ми можемо позначити метод, `describe()` в цьому випадку, як чистий абстрактний (pure abstract method) наступним чином

```
virtual void describe() const = 0;
```

Це означає, що базовий клас `Book` не задає реалізацію цього методу, а вимагає класи, що успадковуються від нього, надати свою реалізацію. Також, тому що `Book` тепер "неповний", ми не можемо створювати об'єкти цього класу напряму. Такий клас називається абстрактним. Якщо абстрактний клас складається тільки з абстрактних методів і немає своїх даних, він також зветься *інтерфейсом*. Чисті інтерфейси дуже зручні і корисні на практиці – вони ставлять умови на API класів, які пов'язані з ним, і дозволяє користуватись ними поліморфно.

Література

- Stroustrup, Bjarne. The C++ programming language. Pearson Education, 2013 (Глава 16).
- CPPReference, Classes.
- Core C++ Guidelines, Classes.
- Google Style Guide, Inheritance.

[18] За умови присутності `virtual` та повного співпадіння сігнатури.

[19] Наприклад, що таке тут generic book? Виводити таке повідомлення користувачу може не мати сенсу, як і мати представників такого класу взагалі.