**Лекція 13 Класи та підкласи (Відношення клас-підклас.)**

**Статичні елементи класу**

За допомогою модифікатору **static** можна описати статичні поля і методи класу. Їх можна розглядати як глобальні змінні або функції, доступні тільки в межах області класу.

Статичні поля застосовуються для зберігання даних, загальних для всіх об'єктів класу, наприклад, кількості об'єктів або посилання на той, що розділяється всіма об'єктами ресурс. Ці поля існують для всіх об'єктів класу в єдиному екземплярі, тобто не дублюються. Нижче перераховані особливості статичних полів:

1. Пам'ять під статичне поле виділяється один раз при його ініціалізації незалежно від числа створених об'єктів (і навіть при їх відсутності) і ініціалізується за допомогою операції доступу до зони дії, а не операції вибору (визначення має бути записане поза функціями):

**class А**

**{**

**public:** static int count; // Оголошення у класі

**};**

…

int A::count; // Визначення у глобальній області

// За умовчанням ініціалізується нулем

//Приклад ініціалізації довільним значенням

// int A::count = 10;

2. Статичні поля доступні як через ім'я класу, так і через ім'я об'єкту:

А \*а, b;

cout << A::count << a->count << b.count;

// Буде виведено одне й теж

3. На статичні поля розповсюджується дія специфікаторів доступу, тому статичні поля, описані як **private**, не можна змінити за допомогою операції доступу до зони дії, як описано вище. Це можна зробити тільки за допомогою статичних методів(див. далі).

4. Пам'ять, яку займає статичне поле, не враховується при визначенні розміру об'єкту за допомогою операції **sizeof**.

5. Конструктор та деструктор в С++ не можуть бути статичними!

## Статичні методи

Метод - це серія тверджень, яка виконується для виконання конкретного завдання. Методи можуть приймати входи та отримувати результати. Статичні та нестатичні методи - це два типи методів, присутні в об'єктно-орієнтованих мовах програмування. Статичний метод - це метод, який асоціюється з класом. Метод, який асоціюється з об'єктом, називається нестатичним (екземпляром) методом. В об'єктно-орієнтованих мовах методи використовуються як механізм роботи над даними, які зберігаються в об'єктах.

## В об'єктно-орієнтованому програмуванні статичний метод - це метод, який асоціюється з класом. Тому статичні методи не мають можливості оперувати певним екземпляром класу. Статичні методи можна викликати без використання об’єкта класу, який містить статичний метод.

Статичні методи призначені для звернення до статичних полів класу. Вони можуть звертатися безпосередньо тільки до статичних полів і викликати тільки інші статичні методи класу, тому що їм не передається прихована вказівка **this**. Звернення до статичних методів проводиться так само, як до статичних полів – або через ім'я класу, або, коли хоча б один об'єкт класу вже створений, через ім'я об'єкту.

**class А**

**{**

static int count; // Поле count – приховане

**public:**

static void inc\_count(){ count++: }

**};**

A::int count; // Визначення в глобальній області

## Void f()

**{**

А а;

// a.count++ – не можна, поле count приховане

// Зміна поля за допомогою статичного методу:

a.inc\_count(); // або А::inc\_\_count();

**}**

Статичні методи не можуть бути константними (**const**) та віртуальними (**virtual**).

Якщо статичні змінні-члени відкриті, то ми отримуємо доступ напряму через ім’я класу та оператор розрішення області видимості. Якщо статичні змінні-члени закриті, розглянемо код:

|  |  |
| --- | --- |
|  | **class Anything**  **{**  **private:**  **static int s\_value;**  **public:**  // статичний метод  **static int getValue() { return s\_value; }**  **};**  // визначення статичного члену, не зважаючи, що він є private  **int Anything::s\_value = 3;**  **int main()**  **{**  **std::cout << Anything::getValue() << '\n';**  **}** |

Як отримати к **Anything::s\_value** в **main**, якщо **s\_value** є private, оскільки ми не можемо напряму отримати доступ до **Anything::s\_value** з функції **main()**, оскільки цей член є **private**.

Як і статичні змінні-члени, **статичні методи** не прив’язані к до будь-якого одного об’єкту класу, то їх можна викликати напряму з класу, через об’єкти класу викликати не рекомендується.

**Статичні методи не мають покажчика \*this,** оскільки вони не прив’язані до об’єкту.

Окрім того **статичні методи можуть напряму звертатися до інших статичним членам і методам класу,** але не можуть напряму звертатися до нестатичних членів, оскільки останні належать конкретному об’єкту класу, а статичні методи не належать.

**С++ не підтримує статичний конструктор.**

**Правило: Всюди, де можливо використовуйте const**

**Константні функції-члени класу**

В оголошенні функції після списку параметрів можна додати модифікатор const. Це буде означати, що функція не змінює стан об'єкта, до якого вона застосовується.

**Функції, що відрізняються const – можуть бути перевантаженні.**

Константну функцію-член класу можна викликати як для константного, так і для не константного об'єкта, у той час як не константну функцію можна викликати тільки для об'єкта, який не є константою.

Коли константна функція-член описується зовні, а не всередині класу, то потрібно додати суфікс const:

inline int Date::year() const //правильно

{ return y;}

Нехай у класі Х існують функції, які надають і змінюють значення об’єкта типу Х. Але, на жаль, не існує способу для перевірки значення об’єкта Х. Проте цю проблему можна легко вирішити, описавши ці функції як ***константні функції-члени***, тобто функції, які не змінюють стан Х:

class Birthday {

int d,m,y;

const string congratulations;

public:

int day() const {return d;}

int month() const {return m;}

int year() const {return y;}

const string getCon() { return congratulations; }

void print (const int d, const int m, const int y)

{cout<<“My birthday is ”<< d<<“.”<<m<<“.”<<y<<“.”;}

};

**Mutable**

**mutable** — антипод const, визначає член, який не буде const ні за яких умов (навіть для const об‘єкта). Інколи виникає потреба в зміні деякого об’єкту в середині класу, гарантуючи недоторканість інших елементів. Недоторканість можна гарантувати за допомогою const, але const забороняє зміну всього.

class Exm

{

int a;

int b;

public:

int getA() const

{

return a; // все правильно

}

int setA(int i) const

{

a = i;// помилка доступу

}

}

Допомогти в цьому випадку може визначення змінної *а* з ключовим словом mutable. Внесемо зміни до наданого вище прикладу:

class Exm

{

mutable int a; //додали в оголошення ключове слово mutable

// яке дозволяє ігнорувати модифікатор const

// по відношенню до даної змінної

int b;

public:

int getA() const //

{

return a; // все правильно

}

int setA(int i) const

{

a = i;// теперь все правильно. Можемо змінювати змінну а

b = i;//Помилка!Змінна b залишається не доступною для змін

}

}

**Перевантаження функцій (попередні відомості)**

1. Перевантажені функції повинні відрізнятися:
   * + *кількістю аргументів*
     + *типом чи порядком слідування хоча б двох аргументів*
   1. Перевантажені функції не відрізняються:
      * *типом, що повертає функція*
      * *специфікатором перед типом змінної*
      * *посиланням перед змінною*

**Вкладені класи**

Клас, оголошений усередині іншого класу, називається вкладеним. Він є членом охоплюючого класу, і його визначення може бути відкрите (**public**) або закрите (**private**). Рівень вкладеності не обмежується.

Ім'я вкладеного класу має бути унікальне в охоплюючому класі, але може збігатися з іншими іменами поза класом.

Доступу за умовчанням до приватних компонентів охоплюючого класу вкладений клас не має, як і охоплюючий клас – до приватних компонентів вкладеного. Обійти заборону допомагає механізм дружніх стосунків (наступна лекція).

Ні вкладений клас, ні охоплюючий не можуть звертатися до методів один одного безпосередньо. Як і в звичайних класах, необхідно оголосити об'єкт, який викликає потрібний метод. Об'єкт охоплюючого класу може передаватися методу вкладеного класу як аргумент.

**Void External::Inner::MethodInner(const External &t)**

**{**

...

// виклик методу охоплюючого класу

memInner = t.MethodExternal();

...

**}**

Метод вкладеного класу **MethodInner()** отримує посилання на об'єкт зовнішнього класу і звичайним способом викликає метод **MethodExternal()**.

Якщо вкладений клас оголошений як **public**, то його можна використовувати як тип за всією програмою. Його ім'я слід писати з префіксом – ім'ям охоплюючого класу:

External::Inner \*pointer;

Усередині методів вкладеного класу ключове слово **this** є вказівкою на поточний об'єкт вкладеного класу.

Методи вкладеного класу можна реалізувати безпосередньо усередині класу. Якщо ж методи вкладеного класу визначаються поза класом, визначення необхідно ставити поза самим зовнішнім з охоплюючих класів – в області глобальної видимості. Ім'я методу у такому разі повинне мати префікси; кількість префіксів дорівнює рівню вкладеності класів.

В області глобальної видимості поза охоплюючим класом можна визначити і сам вкладений клас. C++ вирішує це робити, якщо в охоплюючому класі задати оголошення класу, наприклад:

**class A**

**{**

//...

**class В;** // оголошення вкладеного класу

//...

**};**

**class A::B** // зовнішнє визначення вкладеного класу

**{**

//...

**};**

Доступність визначеного таким чином класу залежить від того, в якій частині охоплюючого класу знаходиться оголошення, – якщо воно приватне, то і визначення є приватним в охоплюючому класі.

**Делегування.**

При створенні нового об’єкта **класу**, компілятор мови C++ неявно викликає **конструктор** цього об’єкту. Можна зустріти клас з декількома конструкторами, які частково виконують одне і те ж, наприклад:

class Boo

{

public:

    Boo()

    {

        // Частина коду X

    }

    Boo(int value)

    {

        // Частина коду X

        // Частина коду Y

    }

};

Тут є 2 конструктори: конструктор за замовчуванням і конструктор, який приймає цілочисельне значення. Оскільки Частина коду X потрібна обом конструкторам, то вона дублюється в кожному з них. Дублювання коду — це те, чого слід уникати, з чим повинен боротися програміст. Починаючи з версії С++11 конструкторам дозволяється викликати інші конструктори. До C++11 явний виклик одного конструктора з іншого призводить до створення тимчасового об’єкта, який потім ініціалізується за допомогою конструктора цього об’єкту і ігнорується, залишаючи вихідний об’єкт незмінним.

З урахуванням цієї можливості попередній код може виглядати так:

class Boo

{

public:

    Boo()

    {

        // Частина коду X

    }

    Boo(int value)

    {

        Boo(); // використовуємо вищенаведений конструктор для виконання частини коду X

        // Частина коду Y

    }

};

Конструкторам дозволено викликати інші методи класу, які не є конструкторами. Кращим рішенням буде створення окремого методу (не конструктора), який виконуватиме загальну ініціалізацію, і обидва конструктори викликатимуть цей метод, що зведе дублювання коду до мінімуму.. Наприклад:

class Boo

{

private:

    void DoX()

    {

        // Частина коду X

    }

public:

    Boo()

    {

        DoX();

    }

    Boo(int nValue)

    {

        DoX();

        // Частина коду Y

    }

};

Крім того, коли потрібно написати метод для повторної ініціалізації класу назад до значень за замовчуванням, може виникнути спокуса спробувати викликати цей конструктор з вашого методу. Однак це призведе до несподіваних результатів. Якщо зкопіювати код з конструктора в функцію ініціалізації — це призведе також до дублювання коду. Кращим рішенням буде перемістити код з конструктора в нову функцію і змусити конструктор викликати нову функцію для виконання ініціалізації:

class Boo

{

public:

    Boo()

    {

        Init();

    }

    Boo(int value)

    {

        Init();

        // Робимо що-небудь з value

    }

    void Init()

    {

        // Код ініціалізації Boo

    }

};

Тут підключено функцію Init() для ініціалізації змінних-членів назад значеннями за замовчуванням, а потім кожен конструктор викликає функцію Init() перед своїм фактичним виконанням. Це зменшує дублювання коду до мінімуму і дозволяє явно викликати Init() з будь-якого місця в програмі.

Починаючи з C++11, конструкторам дозволено викликати інші конструктори. Цей процес називається **делегуванням конструкторів** (або **“ланцюжком конструкторів”**). Щоб один конструктор викликав інший, потрібно просто зробити виклик цього конструктора в **списку ініціалізації членів**. Наприклад:

#include <iostream>

#include <string>

class Employee

{

private:

    int m\_id;

    std::string m\_name;

public:

    Employee(int id=0, const std::string &name=""):

        m\_id(id), m\_name(name)

    {

        std::cout << "Employee " << m\_name << " created.\n";

    }

// Використовуємо делегуючі конструктори для зменшення дубльованого коду

    Employee(const std::string &name) : Employee(0, name) { }

};

int main()

{

    Employee a;

    Employee b("Ivan");

    return 0;

}

Цей клас має 2 конструктори (один з яких викликає іншого). Таким чином, кількість дубльованого коду зменшено (нам потрібно записати тільки одне визначення конструктора замість двох).

## Правила щодо делегуючих конструкторів

1) Конструктору, який викликає інший конструктор, забороняється виконувати будь-яку ініціалізацію членів класу. Тому конструктори можуть або викликати інші конструктори, або виконувати ініціалізацію, але не все одночасно.

2) Один конструктор може викликати інший конструктор, в коді якого може знаходитися виклик першого конструктора. Це створить нескінченний цикл і призведе до того, що пам’ять **стеку** закінчиться і відбудеться збій. Можна цього уникнути, переконавшись, що в конструкторі, який викликається, немає виклику першого (і взагалі будь-якого іншого) конструктора. Вкладені виклики конструкторів використовувати не рекомендується.

**Відношення між класами**

Класи не існують ізольовано.

Основні типи відносин між класами:

1. "узагальнення/спеціалізація" (загальне–частинне) “**is a**”

напр., троянди є частинним випадком квітів, тобто підкласом більш загального класу «квіти»

1. "ціле–частина" — “**part of**”

напр., пелюстки є частиною квітів.

3. асоціація — семантичний, смисловий зв‘язок.

ОО мови програмування підтримують різні комбінації наступних типів відносин:

* асоціація — найбільш загальне та невизначене відношення
* успадкування — "загальне–частинне"
* агрегація — "ціле–частина"
* використання — наявність зв‘язку між екземплярами класів
* інстанціонування — специфічний різновид узагальнення
* метаклас — клас класів (класи як об‘єкти).

Альтернативою успадкуванню вважають делегування, при цьому об'єкти розглядаються як прототипи, які делегують свою поведінку спорідненим об'єктам. Отже, класи стають не потрібними.

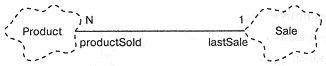
Із шести перерахованих видів відношень найзагальнішою є асоціація. Зазвичай аналітик констатує наявність асоціації й, поступово уточнюючи проект, перетворює її в якийсь спеціалізованіший зв'язок.

Успадкування, ймовірно, варто вважати семантично найцікавішим. Воно виражає відношення загального й частини. Однак лише успадкування недостатньо, щоб виразити все різноманіття явищ і відношень життя. Корисна також аґреґація, яка задає відношення цілого й частини між екземплярами класів. Не зайво додати відношення використання, що означає наявність зв'язку між екземплярами класів.

**Асоціація**

**Приклад** — товари та продаж.

Бажаючи автоматизувати роздрібну торговельну мережу, ми виявляємо дві абстракції – товари й продажі. На рис. показана асоціація, яку ми при цьому вбачаємо. Клас Product – це те, що ми продали за деякою угодою, а клас Sale – сама угода, за якою продано кілька товарів. Асоціація працює в обидва боки: знаючи товар, можна вийти на угоду, за якою він був проданий, а маючи угоду, знайти, що було згідно з нею продано.

****

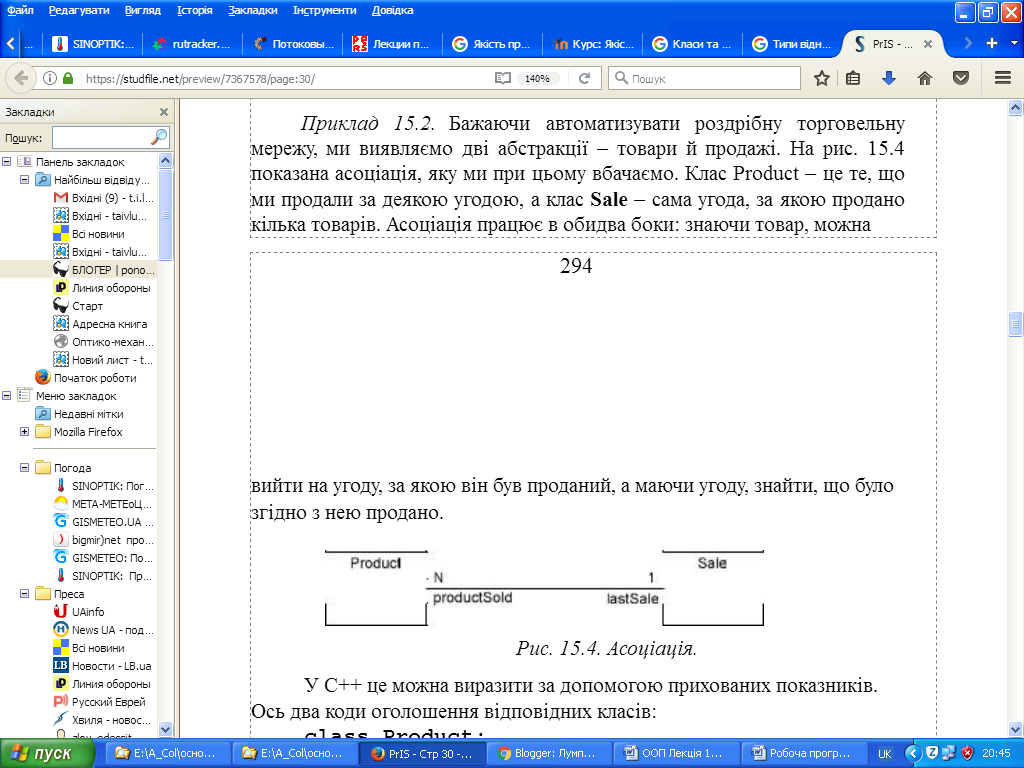


Рис.. Асоціація.

У C++ це можна виразити за допомогою прихованих показників. Ось два коди оголошення відповідних класів:

Class Product; //те, що продали  
Class Sale; //угода, в якій продано декілька товарів  
Class Product {  
 public:  
 . . .  
 protected:  
 Sale\* lastSale;  
};  
Class Sale {  
 public:  
 . . .  
 protected:  
 Product\*\* productSold;  
};

Це асоціація вигляду "один-до-багатьох": кожний екземпляр товару відноситься тільки до однієї останньої продажі, в той час як кожний екземпляр Sale може вказувати на сукупність проданих товарів.

**Семантичні залежності.**

*Асоціація* — смисловий зв‘язок, як правило, не має напрямку та не пояснює, як класи спілкуються один з одним. Це також семантичний, змістовний зв’язок. Напр., комахи асоціюються з квітами. За замовчуванням, вона не має напрямків (якщо не зазначено інше, асоціація, як у цьому прикладі, розуміє двосторонній зв'язок) і не пояснює, як класи спілкуються один з одним (ми можемо тільки відзначити семантичну залежність, вказавши, які ролі відіграють класи). Однак саме це нам потрібно на ранній стадії аналізу. Отже, ми фіксуємо учасників, їх ролі і (як буде сказано далі) потужність відношень.

*Потужність* — кількість учасників цього смислового зв`язку

* + один до одного;
  + один до багатьох;
  + багато до багатьох.

Відношення "один-до-одного" позначає дуже вузьку асоціацію. Наприклад, у роздрібній системі продажів прикладом міг би бути зв'язок між класом Sale і класом CreditCardTransaction: кожний продаж відповідає рівно одному зняттю грошей з цієї кредитної картки. Відношення "багато-до-багатьох" теж рідкісні. Наприклад, кожний об'єкт класу Customer (покупець) може ініціювати транзакцію з декількома об'єктами класу Saleperson (торговельний аґент), і кожний торговельний аґент може взаємодіяти з кількома покупцями. Всі три види потужності мають різного роду варіації.

***Контрольні запитання*.**

1. Як можна використати статичні члени?
2. Для чого призначені статичні методи класу?
3. В чому різниця між статичними та нестатичними методами?
4. Які особливості мають статичні методи?
5. Коли доцільно використовувати модифікатор **mutable?**
6. Що таке вкладений клас і які його особливості?
7. Які залежності існують між класами?

***Для самостійного вивчення***: Поглибити матеріал лекції за наданою літературою. Вивчення лекційного матеріалу та додаткових джерел. Розгляд запитань і виконання завдань для самостійної роботи.

**Рекомендована література**

1. Павловская Т.А. С/С++. Программирование на языке высокого уровня. СПб.: Питер, 2003. – 461 с. URL: <http://www.ph4s.ru/bookprogramir_1.html>
2. Страуструп Б. Язык программирования С++. — СПб.; М.: Невский диалект — ЗАО “Изд-во “Бином”, 1999.
3. Джейс Либерти Освой самостоятельно С++ за 21 день: 3-е изд. пер. с англ.: Уч. пос. – М.: Издательский дом „Вильямс”, 2001. – 816 с.: ил..

4. Липпман С. Б., Лажойе Ж. Язык программирования С++: Вводный курс. — М.: ДМК, 2001. URL: <http://www.insycom.ru/html/metodmat/inf/Lipman.pdf>

5. Дейтел Х., Дейтел П. Основы программирования на С++. – М.: Бином, 1999. – 1024 с. URL: <http://ijevanlib.ysu.am/wp-content/uploads/2018/03/deytel.pdf>

6. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на С++ . [2-е изд.] / Буч Г. - СПб.: Невский Диалект, 1998. - 560 с.

7. Бадд Т. Объектно-ориентированное программирование в действии. [2-е изд.] – СПб.: Изд-во "Питер". 1997. URL: <http://khizha.dp.ua/library/Timothy_Budd_-_Introduction_to_OOP_(ru).pdf>

8. *Скотт Мейерс.* Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов. *"ДМК", 2000; 240 с.*