**Лекція 25**. **Наслідування/Спадкування. Базові поняття та основні принципи**

**Директива using**

На попередній лекції ми розглядали  **Простір імен - namespace**

1. Простір імен – це спосіб логічного групування

namespace N1

{ int g();

char h();}

namespace N2

{ int f1();

int f2();}

int N2::f1()

{ return f2() + N1::g();}

1. Операція розв‘язання видимості :: використовується для звертання до функцій з namespace за іменем поза простором
2. *Кожен клас – це власний простір імен*
3. *Директива using може зробити доступними функцію або всі функції з іншого простору імен (класу).*

namespace N2

{

...

using N1::g;

}

namespace N2

{

...

using namespace N1;

}

#include <iosream>

using namespace std;

**Inline функція**

Специфікатор inline “просить” компілятор розмістити код цієї функції безпосередньо у місце виклику функції, що дозволяє компілятору перетворювати виклик вбудованої функції безпосередньо в код в точці виклику, що, в свою чергу, скорочує витрати на звернення до функції і повернення керування. Inline-функція повинна:

* + бути малою (три оператора);

**Приклад:**

**inline int max(int a, int b)  
{  
 return (a>b)?a:b;  
}**

* + не містити оператора циклу
  + бути реалізована до першого виклику

Абстрагування та інкапсуляція часто породжують велику кількість функцій невеликого розміру, які виконують незначну кількість роботи. Inline-функція дозволяє зберегти вигоди інкапсуляції і уникнути витрат при виклику функції. З іншого боку, частий виклик такої функції буде породжувати велику кількість її копій.

**Функція-член класу може бути описана всередині оголошення класу. У цьому випадку вона вважається вбудовуваною (inline) функцією.**

**Спадкування** ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0" \o "Англійська мова) *inheritance*, „наслідування“ це помилкова назва, калькована з рос. наследование) — один з принципів [об'єктно-орієнтовного програмування](https://uk.wikipedia.org/wiki/%D0%9E%D0%B1%27%D1%94%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D1%96%D1%94%D0%BD%D1%82%D0%BE%D0%B2%D0%B0%D0%BD%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)[[](https://uk.wikipedia.org/wiki/%D0%9D%D0%B0%D1%81%D0%BB%D1%96%D0%B4%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)#cite_note-1), який дає [класу](https://uk.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)) можливість використовувати програмний код іншого (базового) класу, доповнюючи його своїми власними деталями реалізації. Іншими словами, під час успадкування відбувається отримання нового (похідного) класу, який містить програмний код базового класу з зазначенням власних особливостей використання. Успадкування належить до типу [is-a](https://uk.wikipedia.org/wiki/Is-a) відношень між класами. При успадкуванні створюється спеціалізована версія вже наявного класу. Позначення наслідування на [UML](https://uk.wikipedia.org/wiki/Unified_Modeling_Language)-[діаграмі класів](https://uk.wikipedia.org/wiki/%D0%94%D1%96%D0%B0%D0%B3%D1%80%D0%B0%D0%BC%D0%B0_%D0%BA%D0%BB%D0%B0%D1%81%D1%96%D0%B2)

## Переваги використання успадкування

Правильне використання механізму успадкування дає наступні взаємозв'язані переваги:

* **ефективна побудова складних** [**ієрархій**](https://uk.wikipedia.org/wiki/%D0%86%D1%94%D1%80%D0%B0%D1%80%D1%85%D1%96%D1%8F) **класів** з можливістю їх модифікації. Роботу класів в ієрархії можна змінювати шляхом додавання нових успадкованих класів в потрібному місці ієрархії;
* **повторне використання раніше написаного коду** з подальшою його модифікацією під поставлену задачу. Своєю чергою, новостворений код також може використовуватися на ієрархіях нижчих класів;
* **зручність в супроводі** (доповнені) програмного коду шляхом введення нових класів з новими можливостями;
* **зменшення кількості логічних помилок** при розробці складних програмних систем. Повторно використовуваний код частіше тестується, а, отже, менша ймовірність наявності в ньому помилок;
* **легкість в узгодженні різних частин програмного коду шляхом використання** [**інтерфейсів**](https://uk.wikipedia.org/wiki/%D0%86%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_(%D0%BE%D0%B1%27%D1%94%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D1%96%D1%94%D0%BD%D1%82%D0%BE%D0%B2%D0%B0%D0%BD%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)). Якщо два класи успадковані від загального нащадка, поведінка цих класів буде однакова у всіх випадках. Це твердження виходить з вимоги, що схожі об'єкти повинні мати схожу поведінку. Саме використання інтерфейсів зумовлює схожість поведінки об'єктів;
* **створення** [**бібліотек коду**](https://uk.wikipedia.org/wiki/%D0%91%D1%96%D0%B1%D0%BB%D1%96%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_%D0%BF%D1%96%D0%B4%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC), які можна використовувати й доповнювати власними розробками;
* **можливість реалізовувати відомі** [**шаблони проєктування**](https://uk.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D0%B8_%D0%BF%D1%80%D0%BE%D1%94%D0%BA%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F) для побудови гнучкого коду, який не змінює попередніх розробок;
* **використання переваг** [**поліморфізму**](https://uk.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D1%96%D0%BC%D0%BE%D1%80%D1%84%D1%96%D0%B7%D0%BC_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)) неможливо без успадкування. Завдяки поліморфізму забезпечується принцип: один інтерфейс — декілька реалізацій;
* **забезпечення дослідницького програмування (швидкого макетування)**. Таке програмування використовується у випадках, коли цілі та потреби до програмної системи на початку нечіткі. Спочатку створюється макет структури, потім цей макет поетапно вдосконалюється шляхом успадкування попереднього. Процес триває до отримання потрібного результату;
* **ліпше розуміння структури програмної системи** програмістом завдяки природному представленню механізму успадкування. Якщо при побудові складних ієрархій намагатись використовувати інші принципи, то це може значно ускладнити розуміння усієї задачі та призведе до збільшення кількості помилок.

**Недоліки використання наслідування**

При використанні наслідування в програмах були помічені наступні недоліки:

* **неможливо змінити успадковану реалізацію** під час виконання;
* **низька швидкість виконання**. Швидкість виконання програмного коду загального призначення нижча, ніж у випадку використання спеціалізованого коду, який написаний конкретно для цієї задачі. Однак, цей недолік можна виправити завдяки оптимізації коду;
* **великий розмір програм** через використання бібліотек загального призначення. Якщо для деякої задачі розробляти вузькоспеціалізований програмний код, то цей код буде займати менше пам'яти ніж код загального призначення;
* **збільшення складності програми** у випадку неправильного або невмілого використання успадкування. Програміст зобов'язаний вміти коректно використовувати успадкування при побудові ієрархій класів. В іншому випадку це призведе до великого ускладненню програмного коду, і, як результат, збільшенню кількості помилок;
* **складність засвоєння початковими програмістами основ побудови програм**, які використовують успадкування. Однак, цей недолік умовний, бо залежить від досвіду програміста.

**Основні підходи при роботі з класами**

**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. 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**

**class Point { int x, y;**

**public:**

**const Point& f(const Point&) const;**

**};**

1. **const Point& перед функцією означає, що повертається посилання на константний об’єкт**
2. **f(const Point&) означає, що у функцію передається посилання на констатний об’єкт**
3. **const; у кінці оголошення функції означає, що ця функція – константна не змінює об’єкта до якого застосовується**

**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.

**class X**

**{ int n;**

**public:**

**int f();**

**int f() const;**

**};**

**int X::f() //Вірно**

**{ return n++;}**

**int X::f() const**

**//Помилка**

**{ return n++;}**

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

**class X**

**{ int n;**

**public:**

**int f();**

**int f() const;**

**};**

**int X::f() //Вірно**

**{ return n++;}**

**int X::f() const**

**//Помилка**

**{ return n++;}**

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.

**class Point**

**{**

**int x, y;**

**public:**

**void f1();**

**void f2() const;**

**};**

**Point A;**

**const Point B;**

**A.f1();**

**A.f2();**

**B.f1();//помилка виклик не константної функції до констатного об'єкта**

**B.f2();**

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)**

Перевантажені функції повинні відрізнятися:

* + - *кількістю аргументів*
    - *типом чи порядком слідування хоча б двох аргументів*

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 скопіювати 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.

**Показчик this**

Ключове слово **this** позначає вказівник на об'єкт, для якого викликана ця функція, тобто всередині функції класу член того ж класу з ім'ям **х** можна позначати як **x,** і як **this->x.** Щоб забезпечити роботу методу з полями того об’єкта, для якого він був викликаний, в метод автоматично передається прихований параметр this, в якому зберігається посилання на об’єкт, що викликав функцію.

**Метод Main:**

Monster a = new…

Monster b = new…

a.Passport()

b.Passport()

**Метод Passport:**

Виведення полів

name

health

ammo

ДАНІ (heap):

**Об’єкт a:**

name

health

ammo

**Об’єкт b:**

name

health

ammo

this=a

this=b

this.name

this.health this.ammo

**Використання явного this**

У явному вигляді параметр this застосовується:

1) щоб повернути з метода посилання на об’єкт, що його викликав:

class Demo

{ double y;

public Demo T() { return this; }

2) для ідентифікації поля, якщо його ім’я співпадає з

іменем параметра метода:

public void Set\_y( double y ) { this.y = y; }

Покажчик на об'єкт, для якого викликана функція, є неявним параметром цієї функції.

**class X**

**{int a;**

**public:**

**// Члену класу a присвоюється значення параметра a**

**void f(int a) { this->a = a; }**

**};**

**Особливості використання явного this**

* покажчик this у функції-члена класу Х має тип X\*.
* покажчик this у константного методу має тип const X\*.
* this не має адреси &this
* this нічого не можна присвоїти \*this=.
* значення this можна запам’ятати у іншій змінній цього ж типу =\*this;

Метод класу може повертати за допомогою покажчика **this** посилання на об'єкт класу відповідного об'єкта return **\*this**. Покажчик **\*this** це прихований константний покажчик адреси об‘єкту, який викликає метод класу.

Покажчик **\*this** завжди вказує на поточний об‘єкт.

**↓ Покажчик this – повернення з функції**

Розглянемо ще приклад, де ми не передаємо параметри в методи ні при визначенні, ні при виклику методів, тобто не вказуємо з якими параметрами працюватимуть методи класу. При виклику методів, які належать класу, їм неявно передається той самий покажчик **this.** Це відбувається автоматично і непомітно для нас, оскільки він є прихованим першим параметром будь-якого методу-елемента класу. Покажчик this вказує на адресу створеного об'єкта класу – в нашому випадку він отримує адреса об'єкта object і вказує методу setData() в які елементи об'єкта треба внести змінені дані, а методу getData() – елементи якого об'єкта треба відобразити на екрані. Можна використовувати покажчик this явно, ці способи надаються в за коментованих рядках тексту функції getData():

**class X**

**{**

**...**

**public:**

**X& f();**

**};**

**X& X::f()**

**{**

**...**

**return \*this;**

**}**

#include <iostream>

using namespace std;

class SetGetData // Визначаємо клас

{

char str[128];

int number;

public:

void setData() // Метод класу, що приймає дані

{

cout << "Input line: ";

cin.getline(str, 128);

cout << "Input integer: ";

cin >> number;

}

void getData() // Метод класу, що надає дані

{

cout << "\n 1 " << str << number << endl;

cout << "\n 2 " << this->str << this->number << endl ;

cout << "\n 3 " << (\*this).str << (\*this).number << endl;

//Використання покажчика (\*this) при виведенні даних

}

};

int main()

{ system("color F0");

SetGetData object; // Створити об‘єкт

object.setData(); // Вносимо дані

system("cls"); // Очищуємо екран перед виведенням

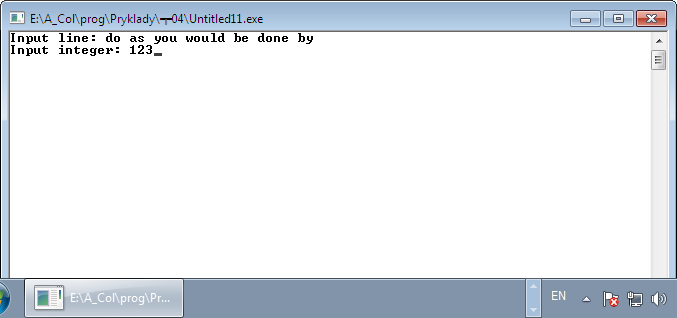
object.getData(); // Дані відображуємо на екрані

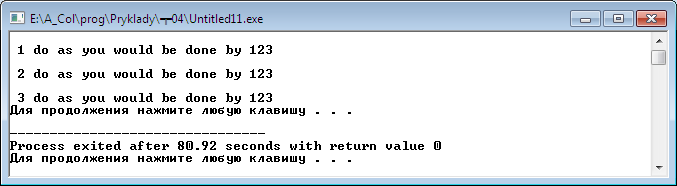
system("pause");

return 0;

}

**Результати виконання**





**Аксессори**

Accessor - Функція-член, яка використовується для отримання даних захищених елементів. До аксесору **get** поміщають код, що повертає значення певного закритого поля класу, з яким пов’язана ця властивість. У аксесорі **set** визначаються дії, що забезпечують присвоєння тому самому полю певного значення. Важливо розуміти, що коли властивість використовується у лівій частині оператора присвоєння, то ***автоматично*** викликається аксесор **set**, який приймає змінну з наперед визначеним ідентифікатором **value**. В іншому випадку так само автоматично викликається аксесор **get**. З технічної точки зору при компіляції властивості генеруються два методи, подібні до тих, що ми використали у попередньому прикладі. Слід розуміти, що властивість не має доступу до пам’яті, отже, її використання без деякого поля класу не має сенсу. Властивість лише керує доступом до цього поля. Прийнятий стиль оформлення властивості полягає у використанні для ідентифікатора властивості поля імені, яке збігається із назвою самого поля з точністю до першого символу – у поля це маленька літера, а у властивості цього поля – велика.

Мутатори – Функція-член, яка використовується для редагування даних захищених членів.

**Статичні змінні — члени класу**

Статичні змінні-члени мають особливі властивості. Вони доступні всім об'єктам класу, не належачи нікому з них окремо. Якби клас не був абстракцією, можна було б сказати, що статична змінна належить класу, а не об'єктам. У реальності це означає, що статична змінна-член класу існує в єдиному екземплярі незалежно від кількості об'єктів. Особливо важно, що статична змінна ініціалізується ще до створення першого об'єкта класу.

Оголошення статичної змінної-члена класу має декілька тонкощів. По-перше, замало оголосити статичну змінна-член — необхідно виділити для неї пам'ять. Для цього її оголошення слід повторити поза тілом класу, в області визначення глобальних змінних. Визначення статичної змінної-члена можна сполучити з її ініціалізацією. По-друге, повторювати ключове слово static при визначенні статичної функції не можна!

Воно повинно бути присутнім тільки в тілі класу! Розглянемо приклад, у якому змінні Re і Im оголошені статичними членами класу TComplex.

**Оголошення статичних членів класу**

**#include <stdio.h>**

**#include <malloc.h>**

**class TComplex**

**{**

**static double Re;**

**static double Im;**

**public:**

**void print();**

**void init();**

**};**

**double TComplex::Re=0.0;**

**double TComplex::Im=0.0;**

**int main()**

**{**

**TComplex Z;**

**printf("sizeof = %d \n",sizeof(Z));**

**Z.init(); // Виклик функції-члена init()**

**Z.print(); // Виклик функції-члена print()**

**return 0;**

**}**

**void TComplex::print() {printf("Z = %lf + i\*%lf \n",Re,Im);}**

**void TComplex::init() {Re = 1; Im = 2;}**

Цікаво, що розмір класу TComplex змінився. Тепер він дорівнює лише одному байту (як у зовсім порожнього класу).

Статичні змінні зручно використовувати як лічильник об'єктів, а також як індикатори на зразок «Зайнято» — «Вільно». Об'єкти можуть по черзі змінювати значення статичної змінної, повідомляючи своїм спадкоємцям, що вони виконали свою місію.

Розглянемо програму, у якій створюються і виводяться на екран 25 комплексних чисел. Зверніть увагу на синтаксичну конструкцію, прийняту для звертання до статичної змінної-члена: **TComplex::counter**. Це підкреслює, що статична змінна-член не належить жодному об'єкту. До речі, це не єдина форма звертання до неї. Можна також використовувати вираз **TComplex.counter**. Для того щоб підкреслити, що кожен об'єкт має доступ до статичного змінна-члену, можна оператор **TComplex.counter++** замінити оператором Z.counter++.

**Використання статичних членів класу**

**#include <stdio.h>**

**#include <malloc.h>**

**class TComplex**

**{**

**double Re;**

**double Im;**

**public:**

**static int counter;**

**void print();**

**void init(double x, double y);**

**};**

**int TComplex::counter=0;**

**int main()**

**{**

**for (int i = 1; i <= 5; i++)**

**for (int j = 1; j <= 5; j++)**

**{**

**TComplex Z;**

**Z.init((double)i,(double)j); // Виклик функції-члена init()**

**Z.print(); // Виклик функції-члена print()**

**TComplex::counter++; // Інкрементація статичної змінної-члена**

**}**

**printf("Counter = %d\n",TComplex::counter);**

**return 0;**

**}**

**void TComplex::print() {printf("Z = %lf + i\*%lf \n",Re,Im);}**

**void TComplex::init(double x, double y) {Re = x; Im = y;}**

**Статичні функції-члени класу**

Статичними можуть бути не тільки змінні-члени, але і функції-члени класу. У цьому випадку вони втрачають доступ до нестатичних членів класу і можуть звертатися тільки до статичних змінних-членів і функцій. Крім того, вони не можуть бути віртуальними, одержувати вказівник this, а також не повинні мати кваліфікаторів const і volatile.

Розглянемо програму, у якій для висновку значення статичного лічильника застосовується статична функція **TComplex::HowMuch().**

**Використання статичних функцій-членів класу**

**#include <stdio.h>**

**#include <malloc.h>**

**class TComplex**

**{**

**double Re;**

**double Im;**

**public:**

**static int counter;**

**void print();**

**void init(double x, double y);**

**static void HowMuch();**

**};**

**int TComplex::counter=0;**

**int main()**

**{**

**for(int i = 1; i <= 5 ; i++)**

**for(int j = 1; j <= 5 ; j++)**

**{**

**TComplex Z1;**

**Z1.init((double)i,(double)j); // Виклик функції-члена init()**

**Z1.print(); // Виклик функції-члена print()**

**TComplex::counter++; // Інкрементація статичного лічильника**

**}**

**TComplex::HowMuch(); // Виклик статичної функції**

**return 0;**

**}**

**void TComplex::print() {printf("Z = %lf + i\*%lf \n",Re,Im);}**

**void TComplex::init(double x, double y) {Re = x; Im = y;}**

**void TComplex::HowMuch(){printf("Counter = %d\n",TComplex::counter);}**

Застосування статичних функцій обмежено ініціалізацією статичних змінних, котру необхідно виконати до створення реальних об'єктів.

**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.

1. 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.

***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?