Лекція 11  **Розподіл поведінки та реалізації**

**Директива using**

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

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

**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);
  + 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.**

**Повторення.**

Для доступу до членів класу (після оголошення деякої змінної цього класу або покажчика на об'єкт даного класу) використовується наступний синтаксис:

**<змінна>.<ім'я члена класу>**

**<покажчик> -> <ім'я члена класу>**

**Управління доступом до членів класу**

**Інкапсуляція (обмеження доступу) –** це процес захисту окремих елементів об’єкта, що не порушує суттєвих характеристик об’єкта як цілого

**Правила доступу** ― це поділ класу на відкриту (public) і закриту (private) частини. Закрита частина доступна лише з середини класу, відкрита ― звідусіль.

Формально клас відрізняється від структури лише правилом замовчування прав доступу:

1. Все, що явно не відкрите в класі, вважається закритим
2. Все, що явно не закрите в структурі, вважається відкритим

Структури звичайно вживають як сукупність даних, класи ― як сукупність даних (атрибутів) і функцій-членів класу (методів)

**Функції-члени класу: методи**

Функція, оголошена в класі без спеціфікатора friend, називається функцією-членом класу або методом. Її виклик має відповідний синтаксис.

**<змінна>.<ім'я методу класу>()**

**<покажчик> -> <ім'я методу класу>()**

Будь-яка функція-член класу може безпосередньо використовувати імена членів свого класу.

**Розділення класу на інтерфейс і реалізацію**

Створюється два файли клас.h клас.сpp

* клас.h – файл для інтерфейсу класу – оголошення класу його методів і даних
* клас.сpp – файл для реалізації класу – реалізація методів

**Запитання:** *Навіщо потрібні функції доступу*?Для того щоб контролювати всі випадки використання атрибутів у кожному об'єкті.

*Яка різниця між автоматичними і динамічними змінними*? Пам‘ять для автоматичної змінної виділяється/створюється при вході в процедуру або блок, що керує цими змінними. При виході ця пам‘ять автоматично звільняється. В момент створення автоматичної змінної її ім‘я зв‘язується з відповідною ділянкою виділеної пам‘яті і цей зв‘язок не може бути зміненим доки існує змінна. Пам‘ять для автоматичних змінних (заснованих на механізмах стека) виділяється без будь-якої явної вказівки з боку користувача. Для динамічної змінної пам‘ять виділяється з "кучі" за спеціальним запитом користувача і, відповідно, за запитом повинна звільнятися. "Куча" – це просто сховище для змінних.

**Скот Майерс. Эффективное использование С++.**

**"Правило 18. Стремитесь к таким интерфейсам класса, которые будут полными и минимальными"**

Повний інтерфейс надає можливість користувачу в деяких межах робити все, що він хоче, що означає, що для будь-якої розумної задачі, яку користувач хоче вирішити, повинен існувати реальний спосіб досягнення поставленої мети, хоч він може бути і не достатньо зручним, як хотілось. З іншого боку, мінімальний інтерфейс – це інтерфейс з мінімально можливою кількістю функцій, такою, що ніякі дві функції не мають функціональності, що перекривається. Тобто правило вимагає, що при повному мінімальному інтерфейсі користувачі мали змогу робити все, що їм потрібно, але інтерфейс класу не повинен бути більш складним ніж це в принципі необхідно.

У інтерфейсу класу перевантаженого функціями є технічні недоліки:

1. Чим більше функцій в інтерфейсі, тим важче потенційному клієнту його зрозуміти, що не викличе в нього ентузіазму щодо вивчення такого інтерфейсу. Існує різниця в сприйнятті класу з 10 функціями та класу зі 100 функціями. Перевантажений інтерфейс може привести до плутанини.
2. Труднощі в експлуатації. Клас з багатьма функціями важче підтримувати та удосконалювати, ніж клас, що має декілька функцій. Складніше уникати дублювання коду (та супутнього дублювання помилок). Такий клас важче документувати.
3. Перевантажене визначення класу приводить до використання занадто довгих файлів заголовків.

Таким чином, непродумане додавання функцій до інтерфейсу є дорогим задоволенням, якщо функції додаються лише задля зручності.

Додавання функцій є виправданим у випадках, коли

1. часто виконувана задача може бути реалізована значно ефективніше, якщо використовувати функцію-член класу;
2. додавання функції-члену робить клас значно більш зручним у використанні;
3. додавання функції може убезпечити користувача від помилок.

Якщо ви реалізуєте доступ до даних-члена, ви можете пізніше замінити член класу обчисленням і ніхто з користувачів цього не помітить.

**"Правило 21. Везде, где только можно, используйте const "**

Використання **const** гарантує, що значення відповідної змінної буде залишатися інваріантним і це забезпечить компілятор.

Коли ви робите значення, що повертає функція, константним, ви зменшуєте вірогідність помилки користувача, не знижуючи безпеку та ефективність. Мета функцій-членів з **const** можна викликати для константних об’єктів. Функції, які різняться лише оголошенням **const** можуть бути перевантажені.

**"Правило 28. Расчленяйте глобальное пространство имен "**

Глобальна область видимості лише одна і це може стати проблемою при розробленні великих програмних продуктів, над якими працює багато людей, кожен з яких створює свої імена в одній і той же області видимості, що може привести до конфлікту імен. Використання власного простору імен дозволяє уникнути цього конфлікту.

**Селектори і модифікатори**

**Селектори (Get)** — методи класу, що повертають значення приватного атрибута (даних);

**Модифікатори (Set)** — методи класу, що встановлюють коректне значення приватного атрибута(даних).

Як добратися до атрибутів, якщо вони закриті? ― За допомогою методів доступу: селекторів і модифікаторів

Селектор

double getX() { return **\_x**; };

Модифікатор

void setX (double x) { \_x = x;}

Окремий модифікатор дозволяє контролювати кожну спробу зміни значення атрибуту, а селектор ― кожне використання його значення.

Але кожна мова програмування пропонує оператор індексування [ ] ― по суті селектор-модифікатор. Модифікатор-селектор

1. не відрізняє зміну значення від читання;
2. порушує інкапсуляцію

#include <iostream>

using namespace std;

class Point

{// Атрибути -private закрили прямий доступ до атрибутів

double \_x, \_y;

int \*ptr; //\* Покажчик на ділянку пам‘яті

// відкрили доступ до методів

public: // Конструктор

Point (double x=0, double y=0**): \_x(x),\_y(y) { }**;

Point (const Point &obj) // Конструктор копіювання

{ cout << "\nCopy Constr\n"; }

~Point(){cout << "\nDestruct\n";}; // Деструктор

// Функції доступу до атрибутів - селектори

double& x() {return \_x;}

/\* ім'я \_х в тексті функції позначає поле \_х того екземпляру, до якого застосовано функцію \*/

double& y() {return \_y;}

void set\_x(double a){\_x=a;}

void set\_y(double a){\_y=a;}

};

void funcShow(Point object)

{ cout << "\nFunction get object as param\n"; }

Point funcReturnObject()

{ Point object;

cout << "\nFunction return object\n";

return object;

}

int main(int argc, char\* argv[])

{

Point a1(1,2);

cout<< a1.x() <<";"<< a1.y() <<endl;

Point b1(1);

cout<< b1.x() <<";"<< b1.y() <<endl;

Point c1;

cout<< c1.x() <<";"<< c1.y()<<endl;

cout << "1 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";

Point obj1; // створюємо об‘єкт класу

cout << "2 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";

funcShow(obj1); // передаємо об‘єкт до функції

cout << "3 - 4 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";

funcReturnObject(); // функція повертає об‘єкт

cout << "5 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";

Point obj2 = obj1; // ініціалізація об‘єкту при створенні

Point \*ptr;

ptr = new Point (22,33);

cout<< ptr->x() <<";"<< ptr->y() <<endl;

delete ptr;

Point \*ptrM = new Point[3];

for (int j=0; j<3; j++)

{ptrM[j].set\_x(j); ptrM[j].set\_y(j);

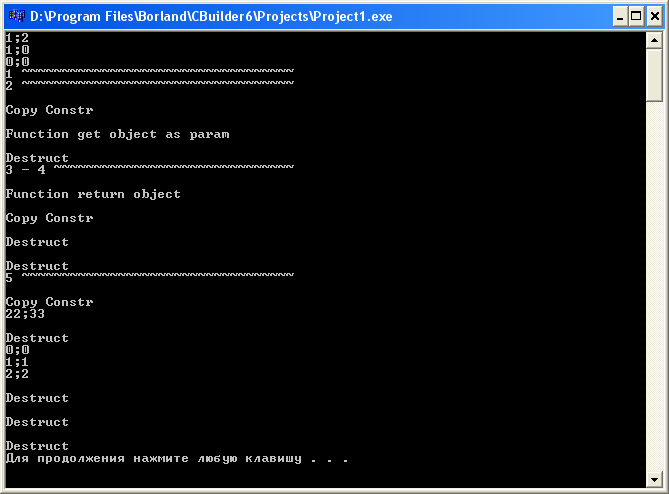
cout<< ptrM[j].x() <<";"<< ptrM[j].y() <<endl; }

delete [] ptrM ;

system("pause");

return 0;

}



**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.

**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.

**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():

#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

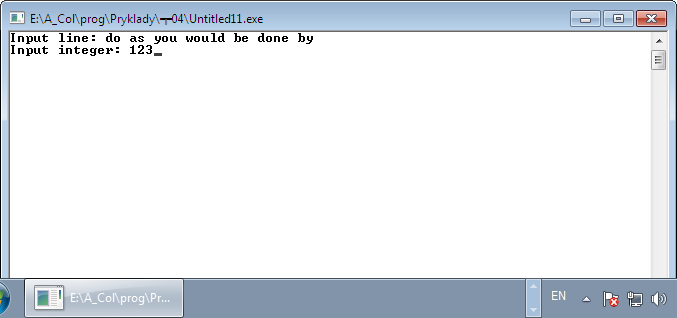
object.getData(); // Дані відображуємо на екрані

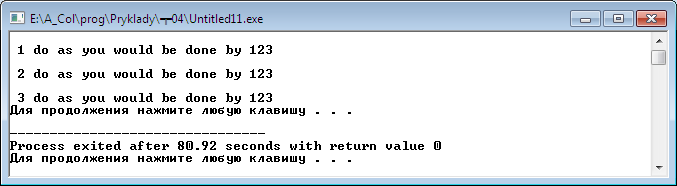
system("pause");

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.

***Контрольні запитання*.**

1. Як співвідносяться структури і класи в С++?
2. Що означають вбудовані методи класу? Коли вони застосовуються? Чи є вони обов’язковими?
3. Як пов’язані селектори та модифікатори з гетерами та сеттерами?
4. В чому сутність ключового слова **this**?
5. Чи може бути масив членом класу? Якого типу елементи може мати такий масив?
6. Як оголосити масив структур? Як здійснюється доступ до полів структури в масиві структур?
7. Чи можуть бути об’єкти елементами масиву? Як одержати конкретний об’єкт з такого масиву?

***Для самостійного вивчення***: Розібрати приклад з комплексними числами в главі 4 та задачу в главі 5 [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]. Поглибити матеріал лекції за наданою літературою.

**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.*

**Скот Мейерс. Эффективное использование С++.**

* Глава 1 Приучайтесь к C++
* [Правило 1: Относитесь к C++ как к конгломерату языков](http://cpp.com.ru/meyers_1/ch1.html)
* Правило 2: Предпочитайте const, enum и inline использованию #define
* Правило 3: Везде, где только можно используйте const
* Константные функции-члены
* Правило 4: Прежде чем использовать объекты, убедитесь, что они инициализированы
* [Глава 2 Конструкторы, деструкторы и операторы присваивания](http://cpp.com.ru/meyers_1/ch2.html)
* Правило 5: Какие функции C++ создает и вызывает молча
* Правило 6: Явно запрещайте компилятору генерировать функции, которые вам не нужны
* [Глава 4 Проектирование программ и объявления](http://cpp.com.ru/meyers_1/ch4.html)
* Правило 18: Проектируйте интерфейсы так, что их легко было использовать правильно и трудно – неправильно
* Правило 19: Рассматривайте проектирование класса как проектирование типа
* Правило 20: Предпочитайте передачу по ссылке на const передаче по значению
* Правило 21: Не пытайтесь вернуть ссылку, когда должны вернуть объект
* Правило 22: Объявляйте данные-члены закрытыми
* Правило 23: Предпочитайте функциям-членам функции, не являющиеся ни членами, ни друзьями класса