**Лекція 24.** Класи та спадкування в С++

**Відношення між класами**

***Відношення залежності***

Залежністю – називають відношення використання, згідно з яким зміна в специфікації одного елемента може вплинути на поведінку іншого елементу, що його використовує, причому зворотне не обов'язково. Найчастіше залежності застосовуються при роботі з класами, щоб відобразити в сигнатурі операції той факт, що один клас використовує інший як аргумент.

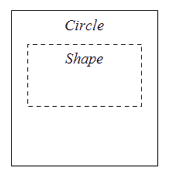
***Відношення агрегації***

Агрегацією – називають відношення включення, коли клас А включає в себе об’єкти (покажчики на об’єкти) класу В. Його ще називають відношенням частина/ціле або “has a”. Клас-агрегат вміщує колекцію покажчиків на екземпляри класів-частин.

**Композиція** – частинний і більш сильний випадок агрегації, коли зі знищенням цілого знищуються частини. Клас-композит вміщує колекцію екземплярів агрегованого класу.

***Відношення узагальнення***

Узагальненням називається відношення класифікації між загальною сутністю, суперкласом (батьківським, базовим) і більш спеціалізованим різновидом цієї сутності, що називають підкласом чи нащадком або похідним. Узагальнення називають зв’язком “is a”, від анг. is a kind of. Троянда is a kind of (це є вид) квітки.

Узагальнення і наслідування / успадкування (спеціалізація) – це протилежні напрямки одного відношення. Спадкування – відношення обернене до узагальнення. Клас-нащадок повторює структуру і поведінку іншого класу (одиночне спадкування) або других (множинне спадкування) [Буч, гл. 2]. Клас-нащадок наслідує (вміщує) всі поля та методи батьківського класу, хоча може мати і власні.

Наявність механізму спадкування відрізняє об’єктно-орієнтовані мови від просто об’єктних.

**Основні принципи наслідування**

Мета ООП полягає в повторному використовуванні створених [клас](http://moodle.ipo.kpi.ua/moodle/mod/glossary/showentry.php?courseid=508&concept=%D0%9A%D0%BB%D0%B0%D1%81)ів, що економить час і сили. Якщо існує деякий клас, то можливі ситуації, що новому класу потрібні деякі або навіть всі особливості вже існуючого класу, і необхідно додати один або декілька елементів даних або функцій. В таких випадках C++ дозволяє будувати новий об'єкт, використовуючи характеристики вже існуючого об'єкту. Іншими словами, новий об'єкт буде успадковувати елементи існуючого класу (базового класу). Коли будується новий клас з існуючого, цей новий клас часто називається похідним класом. Цей процес називається **наслідуванням** або **успадкуванням**. Наслідування є фундаментальною концепцією об'єктно-орієнтованого програмування.

**Англійською:** похідні класи - Derived classes;

Наслідування - Inheritance

***Б.Страуструп:*** *«*Похідні типи дають *простий, гнучкий і ефективний* апарат для:

* завдання класові альтернативного інтерфейсу (повторне використання коду);
* визначення класу шляхом додавання нових можливостей до базового класу без перепрограмування чи перекомпілювання».

Наслідування поділяють на просте і множинне, відкрите і закрите.

***Основні концепції наслідування***

* Якщо програми використовують успадкування, то для породження нового класу необхідний базовий клас, тобто новий клас успадковує елементи уже існуючого базового класу.
* Для ініціалізації елементів похідного класу програма повинна викликати [конструктор](http://moodle.ipo.kpi.ua/moodle/mod/glossary/showentry.php?courseid=508&concept=%D0%9A%D0%BE%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80)и базового і похідного класів.
* Використовуючи оператор "крапку", програми можуть легко звертатися до елементів базового і похідного класів.
* На додаток до загальних ([public](http://moodle.ipo.kpi.ua/moodle/mod/glossary/showentry.php?courseid=508&concept=public)) (доступним всім) і приватних ([private](http://moodle.ipo.kpi.ua/moodle/mod/glossary/showentry.php?courseid=508&concept=private)) (доступним методам класу) елементів C++ надає захищені (protected) елементи, які доступні базовому і похідному класам.
* Для вирішення конфлікту імен між елементами базового і похідного класів програма може використовувати оператор глобального дозволу, указуючи перед ним ім'я базового або похідного класу.

При успадкуванні, за необхідності, можна перевизначити деякі функції батьківського класу в класі-нащадку. Звичайно у базовому класі описуються найбільш загальні властивості, які придатні для всіх похідних від нього класів. Похідний клас успадковує ці загальні властивості і додає нові, характерні тільки для нього.

**Принцип підстановки**

Відкрите наслідування встановлює між класами відношення “є” (“is a”): клас нащадок є різновидом базового класу. Всюди, де використовується об‘єкт базового класу, дозволяється використовувати об‘єкт похідного класу. Дане положення називається принципом підстановки.

**Форми наслідування**

Бадд [15,розділ 7.3-7.4] таким чином визначає форми наслідування.

**Спеціалізація**. Клас-нащадок є більш конкретним або спеціалізованим випадком батьківського класу. Тобто, клас-нащадок є підтипом батьківського класу.

**Специфікація**. Батьківській клас описує поведінку, яка реалізується в класі-нащадку, але залишено нереалізованим в батьківському.

**Конструювання**. Клас-нащадок використовує методи, надані батьківським класом, але не є підтипом батьківського класу (реалізація методів порушує принцип підстановки).

**Узагальнення**. Клас-нащадок модифікує або перевизначає деякі методи батьківського класу з метою отримання об’єкту більш загальної категорії.

**Розширення**. Клас-нащадок додає нові функціональні можливості до батьківського класу, але не змінює успадковану поведінку.

**Обмеження**. Клас-нащадок обмежує використання деяких методів батьківського класу.

**Варіювання**. Клас-нащадок та батьківський клас є варіаціями на одну тему, та зв’язок «клас—підклас» довільний.

**Комбінування**. Клас-нащадок успадковує риси більш ніж одного батьківського класу. Це — множинне успадкування; воно буде розглянуто в наступних лекціях окремо.

Переваги спадкування з єдиним предком/батьком полягає в тому, що функціональні можливості останнього класу успадковуються всіма об’єктами. Таким чином гарантується, що кожний об’єкт володіє загальним мінімальним рівнем функціональності. Мінус в тому, що єдина ієрархія "зщеплює" всі класи один з одним.

**Синтаксис наслідування**

**class Base**

**{**

**//оголошення базового класу**

**};**

**class Derived : специфікатор\_доступу Base [, специфікатор\_доступу Base2, … ]**

**{**

**//оголошення класу нащадка**

**};**

Усі члени базового класу автоматично стають членами похідного. Керуючись оголошенням похідного класу, компілятор ніби збирає його з різних частин — спочатку він бере усі властивості базового класу, а потім додає до них нові функціональні можливості похідного. Хоча всі члени базового класу автоматично стають членами похідного класу, однак доступ до цих членів визначається видом успадкування. У залежності від специфікатора доступу, зазначеного при оголошенні похідного класу, розрізняють відкрите, закрите і захищене успадкування. За замовчуванням використовується закрите успадкування (специфікатор private).

При описі класу в заголовку перечислюються усі класи, які є для нього базовими / батьківськими. Можливість звернення до елементів цих класів регулюється за допомогою специфікаторів доступу ***private, protected*** та ***public:***

**class имя : [private | protected | public] имя\_базового\_класу**

**{ /\* тіло класу \*/ };**

Якщо базових класів декілька, вони перелічуються через крапку. Ключ доступу може стояти перед кожним класом, наприклад:

**class А { ... };**

**class В { ... };**

**class С { ... };**

**class D: А, protected В, public С { ... }:**

**Специфікатори доступу**

За замовчуванням для класів використовується специфікатор доступу ***private,*** для структур ***-***  ***public***.

Для будь-якого члену класу може також застосовуватися специфікатор доступу ***protected,*** який для одиночних класів, що не входять до ієрархії, аналогічний до ***private*** . Різниця виявляється при спадкуванні, що демонструє наступна таблиця.

|  |  |  |
| --- | --- | --- |
| Специфікатор доступу | | |
| Базовий клас | В базовому класі | Доступ в класі-нащадку |
| ***private*** | ***private*** | відсутній |
| ***protected*** | ***private*** |
| ***public*** | ***private*** |
| ***protected*** | ***private*** | відсутній |
| ***protected*** | ***protected*** |
| ***public*** | ***protected*** |
| ***public*** | ***private*** | відсутній |
| ***protected*** | ***protected*** |
| ***public*** | ***public*** |

Ми бачимо, що ***private*** – члени базового класу в класі-нащадку недоступні незалежно від специфікатору доступу. Звернення до них може здійснюватися через методи базового класу.

Члени базового класу ***protected*** при спадкуванні зі специфікатором доступу ***private*** стають в класі-нащадку ***private***, в решті випадків права доступу до них не змінюються.

Доступ до ***public***– членів базового класу при спадкуванні стає відповідним специфікатору доступу

Якщо базовий клас спадкується зі специфікатором доступу ***private,*** можна вибірково зробити деякі його члени доступними в класі-нащадку, оголосивши їх в секції ***public*** в класі-нащадку за допомогою операції доступу до області видимості:

**class Base{**

**public: void f();**

**}**

**class Derived : private Base{**

**public: Base::void f();**

**}**

**Просте спадкування**

Простим називається спадкування, при якому похідний клас або клас-нащадок має один базовий клас або одного батька. Для різних методів класу існують різні правила спадкування — наприклад, конструктори, деструктори та операція присвоювання в похідному класі не спадкуються.

***Поля і методи при спадкуванні***

* Клас-нащадок успадковує всі поля та методи батьківського класу.
* Якщо у батьківському класі поле чи метод приватний, то нащадок не має до нього доступу.
* Допускається не тільки успадкування методів базового класу, але також додавання нових і перевизначення існуючих методів.
* Якщо ім’я поля (методу) у похідному і базовому класі співпадають, говорять про перевизначення або перекриття. Для звернення до змінної базового класу використовують операцію доступу до області видимості "**::".**

***Спеціальні методи при спадкуванні***

* Клас-нащадок успадковує всі поля і методи батьківського класу крім:
  + - Конструкторів
    - Деструктора
    - Операції присвоєння
* Основне правило: у конструкторі нащадку потрібно ініціалізувати власні змінні, а для наслідуваних даних - викликати конструктор базового класу.

При створенні об'єктів похідного класу необхідно спочатку створити проміжний об'єкт базового класу, оскільки об'єкт похідного класу являє собою модифікований об'єкт базового класу. Отже, спочатку викликається конструктор базового класу, а потім — конструктор похідного класу. Якщо базових класів декілька, вони викликаються в порядку їх перерахування в списку наслідуваних класів.

Деструктори викликаються в зворотному порядку.

Оголошення конструктора похідного класу.

ім*'*я*\_*похідного*\_*класу:

**ім*'*я*\_*базового класу*1*(параметри), ..., ім*'*я*\_*базового класу(параметри)**

**{**

**тіло конструктора**

**}**

***Конструктори при спадкуванні***

* Якщо в конструкторі похідного класу явно не викликається конструктор базового класу, то компілятор сам викликає конструктор за замовчуванням базового класу.
* Якщо необхідно викликати конструктор базового класу такого ж виду, то конструктор вказується в рядку його ініціалізації.
* Тіло конструктора базового класу завжди виконується раніше тіла конструктора похідного класу

***Деструктори при спадкуванні***

* Деструктор похідного класу не вимагає явно викликати деструктор базового класу. У деструкторі похідного класу компілятор автоматично генерує виклики базових деструкторів
* Тіло деструктора похідного класу завжди виконується раніше тіла деструктора базового класу

Знищення виконується в оберненому порядку до створення.

***Приклад.*** Розробити програму, що реалізує приклад взаємодії базового (class Base) і похідного (class Derived) класів. Розглянути роботу конструкторів.

**#include <iostream >**

**#include <conio.h>**

**#include <Windows.h>**

**using namespace std;**

**class Base**

**{ int i;**

**public: int a, b; //поля базового класу**

**Base (int a=0, int b=0){this->a = a; this->b = b;**

**cout << "constructor class Base" << endl;}**

**~Base() {cout << "destructor class Base" << endl;};**

**void set\_i(int n) { i = n; }**

**int get\_i(){return i;};**

**};**

**class Derived : public Base**

**{ public: int c, j; //власні поля нащадка**

**Derived (int a=0, int b=0, int c =0):**

**Base (a, b) //явний виклик конструктора базового класу**

**{this->c = c; //присвоєння власних змінних**

**cout << "constructor class Derived" << endl;};**

**~Derived ()**

**{cout << "destructor class Derived" << endl;};**

**void set\_j(int n) { j = n; }**

**int mult() { return j\*get\_i();}**

**};**

**int main()**

**{ /\*Отримання дескриптора\*/**

**system("color F0"); /\*колір фону - білий, тексту - чорний\*/**

**Base b1, b2; //змінні базового класу**

**Derived d1,d2; //змінні похідного класу**

**b1 = b2; //операція = базового класу**

**cout<<"b1 = b2 = " << b1.b <<endl;**

**d1 = d2; //операція = похідного класу**

**cout<<"d1 = d2 = "<< d1.b <<endl;**

**b2 = d1; //базовий = похідний:**

**cout<<"b2 = d1 = "<< b2.b <<endl;**

**d1.Base::b = 2; //b - перевизначена тому звернення через ::**

**d1.c = 3; d2.c = 4;**

**cout<<"b2, d1 = "<< b2.b << "," << d1.b <<endl; //принцип підстановки**

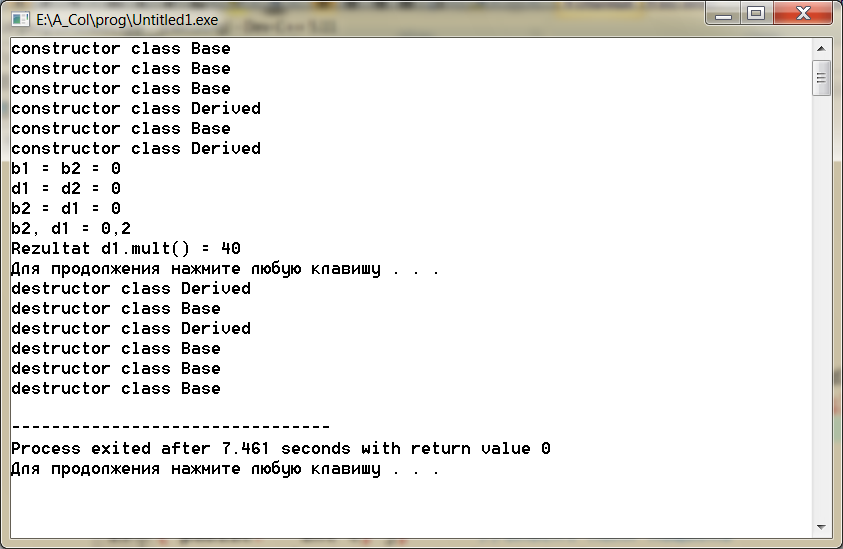
**d1.set\_i(10);**

**d1.set\_j(4);**

**cout<<"Rezultat d1.mult() = "<<d1.mult()<<endl;**

**system ("pause");**

**return 0;}**



Результат роботи програми:

**Rezultat= 40**

У програмі подано спочатку опис базового класу **Base**, а потім похідного класу **Derived**. У похідному класі оголошено, що базовий клас є відкритим для похідного, тобто всі захищені і відкриті дані і функції (але не закриті) класу **Base** доступні в класі **Derived** . За таких умов у функції **mult()** використовується відкрита функція **get\_i()** класу **Base** , однак змінна **і**, яка оголошена в закритому розділі класу **Base**, безпосередньо не може застосовуватися у класі **Derived**.

У головній програмі оголошено об’єкт **d1** класу **Derived**, однак для його обробки реалізується як функція **d1.set\_i (10)** класу **Base**, так і функція **d1.set\_j(4)** класу **Derived**. Тому при виконанні функції **d1.mult()** буде виведено 40. Якби клас А успадковувався у закритому (**private**)режимі, то усі його елементи було б заборонено використовувати в класі-нащадку.

**Приклади використання конструкторів різного типу**

**Конструктори і деструктори.**

Серед функцій-членів класу можуть бути такі, що визначають процеси створення, ініціювання, копіювання та знищення об’єктів свого класу. До цих функцій належать конструктори і деструктори, які, якщо явно не оголошуються, то автоматично викликаються системою за замовчуванням.

Чим і як може ініціалізуватися клас?

* Деякими замовчуваними значеннями
* Списком значень, що відповідає типам даних
* Об’єктом цього ж класу визначеним раніше
* Деякими даними сумісними за типом з типом класу

У мові С++ для ініціалізації об'єктів призначений механізм, називаний ***конструктором***. Це — функція-член, ім'я якої збігається з ім'ям класу. Вона може мати будь-які параметри, що необхідні для ініціалізації полів об'єкта. При цьому конструктор не має ніякого значення, що повертається. Його виклик являє собою визначення об'єкта. Інакше кажучи, визначаючи об'єкт, ви викликаєте конструктор.

**Види конструкторів:**

* За замовчуванням
* Ініціалізації (з параметрами)
* Копіювання
* Додаткові конструктори

Головною метою **конструкторів** є ініціювання змінних-об’єктів класу та розподілення пам’яті для їх зберігання. Конструктор викликається кожного разу при створенні об’єкта даного класу або явно, або автоматично.

**Дії, які виконують конструктори.**

* Виділяють динамічну пам’ять
* Ініціалізують дані класу
* Якщо програміст не вказав жодного конструктора або якісь поля не були проініціалізовані, то полям значимих типів присвоюється нуль, полям показникових типів — значення null;
* Не можуть нічого повернути
* Забороняють або дозволяють і виконують приведення до типу класу
* Можуть викликатися явно та неявно

**Основні правила роботи з конструкторами**:

* ім’я конструктора повинне співпадати з ім’ям свого класу;
* для конструктора не вказується тип значення, яке повертає функція;
* клас може мати декілька конструкторів або не мати жодного;
* конструктор за замовчуванням — це конструктор, який не має параметрів, або всі його параметри мають значення за замовчуванням;
* конструктор копіювання безпосередньо призначений для створення об’єкта класу шляхом копіювання даних з існуючого об’єкта.

**Конструктор за замовчуванням** – це конструктор, який дозволяє створювати екземпляри класів з неявною ініціалізацією даних**.** Такий конструктор автоматично створюється компілятором для класів, в яких не визначений власний конструктор. Проте, як тільки в класі визначається хоч один конструктор, такий автоматичний конструктор перестає діяти. В разі необхідності створення екземплярів без ініціалізації, варто визначити в класі конструктор за умовчанням. Найпростіший спосіб зробити це – перевантажити конструктор, або визначити умовчання для всіх його параметрів.

**Конструктор ініціалізації** містить окремі значення, що використовуються для ініціалізації стану полів екземпляру класу. У списку параметрів може бути зазначений нуль, один чи більше параметрів будь-якого типу. В прикладі минулої лекції це

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

Це скорочена форма запису, конструктор може мати вигляд

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

Приклад минулої лекції

#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[])

{ system("color F0");

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);

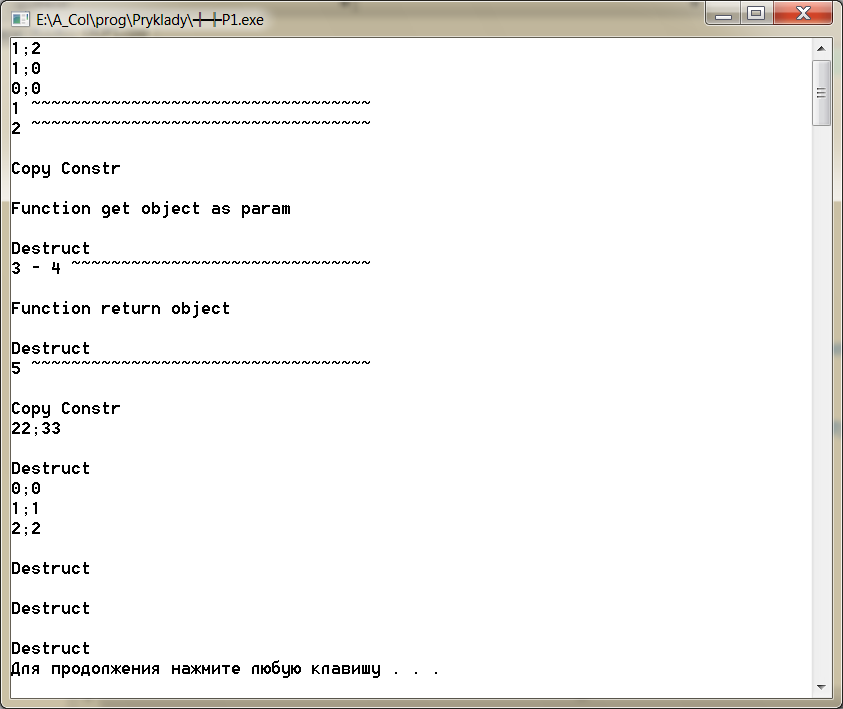
cout<< ptrM[j].x() <<";"<< ptrM[j].y() <<endl; }

delete [] ptrM ;

system("pause");

return 0;

}



Використання списку ініціалізаторів членів **переважніше**, ніж встановлення значень в тілі конструктора, оскільки він безпосередньо ініціалізує елемент. В наданому прикладі показано список ініціалізаторів членів, який складається зі всіх виразів ***ідентифікаторів (аргументів)*** після двокрапки (":"). Ідентифікатор повинен посилатися на член класу; він ініціалізується значенням аргументу. Аргумент може бути одним з параметрів конструктора, викликом функції.

*Список ініціалізації — єдиний спосіб задати значення константних змінних-членів.* Крім того, така ініціалізація буває більш ефективною. Правда, якщо клас містить велику кількість членів, список може стати занадто довгим, що знизить наочність програми.

Можна описати будь-яку кількість конструкторів класу. Вони повинні відрізнятися кількістю або типами параметрів (властивість поліморфізму).

**class** Country

{

**private**:

**char** name[40];

**double** area;

**int** population;

**public**:

Country(**double** area) { **this**->area = area; }

Country(**const** **char**\* name);

Country(**const** **char**\* name, **double** area);

Country(**const** **char**\* name, **double** area, **int** population);

. . .

};

Country::Country(**const** **char** \* name)

{

strcpy(**this**->name, name);

area = 1; // істотно, щоб територія не мала значення 0

// і не виникало помилки у функції density()

}

Country::Country(**const** **char** \* name, **double** area)

{

strcpy(**this**->name, name);

**this**->area = area;

}

Country::Country(**const** **char** \* name, **double** area, **int** population)

{

strcpy(**this**->name, name);

**this**->area = area;

**this**->population = population;

}

**Конструктор копіювання** ініціалізує стан класу значення іншого екземпляру цього класу. В списку параметрів вказується єдиний параметр, що має тип “посилання на екземпляр класу”. Для явного визначення конструктора копіювання необхідно дотримуватись особливого синтаксису:

**<ім’я\_класу> (const <ім’я\_класу> & );**

Параметром конструктора копіювання є стала змінна-посилання на екземпляр класу. Його призначення – коректне створення копії екземпляру. Особливо важливо це у випадку, коли членами класу є вказівники, пам’ять під які виділяється оператором **new**. Адже тоді поелементне копіювання копіює вказівник (поверхневе копіювання), а не об’єкт, на який він посилається, – для цього необхідне глибоке копіювання.

*Конструктор копіювання потрібен нам для того, щоб створювати «реальні» копії об'єктів класу, а не побітову копію об'єкта*. Іноді це принципово важливо. Таку «реальну» копію об'єкта треба створювати в декількох випадках:

* коли ми передаємо об'єкт в яку-небудь функцію у вигляді параметра;
* коли яка-небудь функція повинна повернути об'єкт класу в результаті своєї роботи;
* коли ми в головній функції один об'єкт класу ініціалізуємо іншим об'єктом класу.

Наприклад, ми передаємо об'єкт в функцію у вигляді параметра. Функція буде працювати не з самим переданим об'єктом, а з його побітової копією. Припустимо в конструкторі класу, при створенні об'єкта, виділяється певний об'єм пам'яті, а деструкція класу цю пам'ять звільняє. Покажчик побітової копії об'єкта буде зберігати ту ж адресу пам'яті, що й оригінальний об'єкт. І, коли при завершенні роботи функції і знищенні побітової копії об'єкта, спрацює деструктор, він обов'язково звільнить пам'ять, яка була зайнята об'єктом-оригіналом. На додачу, ще й при завершенні роботи програми, деструктор спрацює повторно і спробує ще раз звільнити цей обсяг пам'яті, що неминуче призведе до помилок програми. Та ж доля спіткає і пам'ять, виділену для покажчика об'єкта, якщо буде віддалятися побітова копія повертається функцією об'єкта, і побітова копія при ініціалізації об'єкту класу іншим об'єктом.

Щоб уникнути цих проблем і помилок існує конструктор копіювання. Його робота полягає в тому, щоб створити реальну копію об'єкта зі своєю особистою виділеної динамічною пам'яттю.

Розглянемо конструктор копіювання.

#include <iostream>

**#include <cassert>**

class Drob

{

private:

    int m\_numerator;

    int m\_denominator;

public:

    // Конструктор з параметрами

    Drob(int numerator=0, int denominator=1) :

        m\_numerator(numerator), m\_denominator(denominator)

    {

**assert**(denominator != 0);

    }

    // Конструктор копіювання

    Drob(const Drob &drob) :

        m\_numerator(drob.m\_numerator), m\_denominator(drob.m\_denominator)

/\* Примітка: Ми маємо прямий доступ до членів об‘єкту drob, оскільки ми зараз знаходимося всередині класу Drob \*/

    {

/\*Нема необхідності виконувати перевірку denominator тут, оскільки ця перевірка вже здійснюється в конструкторі класу Drob\*/

        std::cout << "Copy constructor worked here!\n"; /\* просто, щоб показати, що це працює\*/

    }

*friend std::ostream& operator<<(std::ostream& out, const Drob &d1);*

};

*std::ostream& operator<<(std::ostream& out, const Drob &d1)*

{

out << d1.m\_numerator << "/" << d1.m\_denominator;

return out;

}

int main()

{

Drob sixSeven(6, 7); /\* пряма ініціалізація об‘єкту класу Drob, викликається конструктор Drob(int, int)\*/

Drob dCopy(sixSeven); /\* пряма ініціалізація, викликається конструктор копіювання класу Drob\*/

std::cout << dCopy << '\n';

}

Результат виконання програми:

**Конструктори і деструктори.**

Серед функцій-членів класу можуть бути такі, що визначають процеси створення, ініціювання, копіювання та знищення об’єктів свого класу. До цих функцій належать конструктори і деструктори, які, якщо явно не оголошуються, то автоматично викликаються системою за замовчуванням.

Чим і як може ініціалізуватися клас?

* Деякими замовчуваними значеннями
* Списком значень, що відповідає типам даних
* Об’єктом цього ж класу визначеним раніше
* Деякими даними сумісними за типом з типом класу

У мові С++ для ініціалізації об'єктів призначений механізм, називаний ***конструктором***. Це — функція-член, ім'я якої збігається з ім'ям класу. Вона може мати будь-які параметри, що необхідні для ініціалізації полів об'єкта. При цьому конструктор не має ніякого значення, що повертається. Його виклик являє собою визначення об'єкта. Інакше кажучи, визначаючи об'єкт, ви викликаєте конструктор.

**Види конструкторів:**

* За замовчуванням
* Ініціалізації (з параметрами)
* Копіювання
* Додаткові конструктори

Головною метою **конструкторів** є ініціювання змінних-об’єктів класу та розподілення пам’яті для їх зберігання. Конструктор викликається кожного разу при створенні об’єкта даного класу або явно, або автоматично.

**Дії, які виконують конструктори.**

* Виділяють динамічну пам’ять
* Ініціалізують дані класу
* Якщо програміст не вказав жодного конструктора або якісь поля не були проініціалізовані, то полям значимих типів присвоюється нуль, полям показникових типів — значення null;
* Не можуть нічого повернути
* Забороняють або дозволяють і виконують приведення до типу класу
* Можуть викликатися явно та неявно

**Основні правила роботи з конструкторами**:

* ім’я конструктора повинне співпадати з ім’ям свого класу;
* для конструктора не вказується тип значення, яке повертає функція;
* клас може мати декілька конструкторів або не мати жодного;
* конструктор за замовчуванням — це конструктор, який не має параметрів, або всі його параметри мають значення за замовчуванням;
* конструктор копіювання безпосередньо призначений для створення об’єкта класу шляхом копіювання даних з існуючого об’єкта.

**Конструктор за замовчуванням** – це конструктор, який дозволяє створювати екземпляри класів з неявною ініціалізацією даних**.** Такий конструктор автоматично створюється компілятором для класів, в яких не визначений власний конструктор. Проте, як тільки в класі визначається хоч один конструктор, такий автоматичний конструктор перестає діяти. В разі необхідності створення екземплярів без ініціалізації, варто визначити в класі конструктор за умовчанням. Найпростіший спосіб зробити це – перевантажити конструктор, або визначити умовчання для всіх його параметрів.

**Конструктор ініціалізації** містить окремі значення, що використовуються для ініціалізації стану полів екземпляру класу. У списку параметрів може бути зазначений нуль, один чи більше параметрів будь-якого типу. В прикладі минулої лекції це

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

Це скорочена форма запису, конструктор може мати вигляд

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

Приклад минулої лекції

#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[])

{ system("color F0");

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);

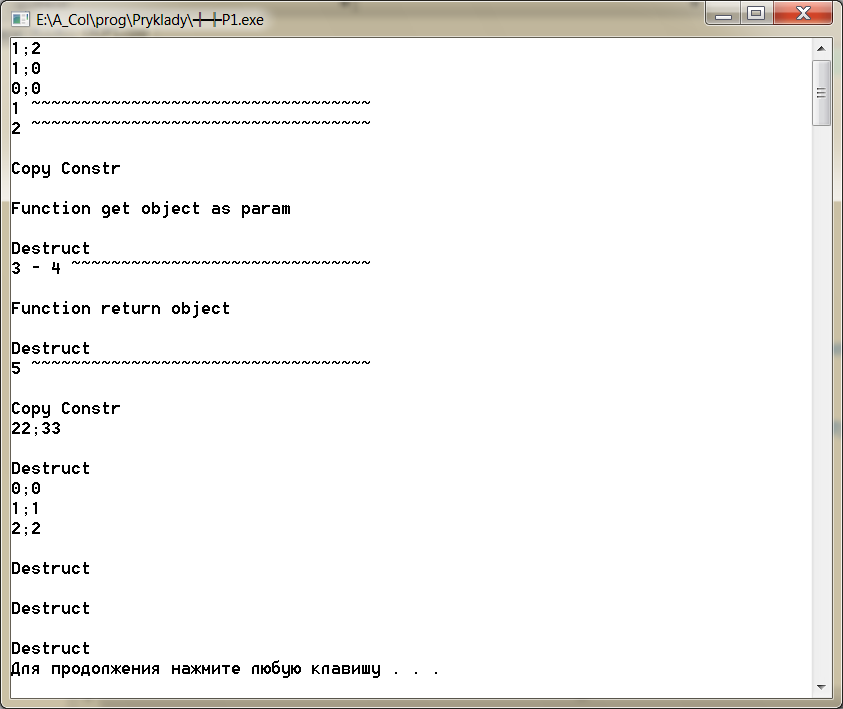
cout<< ptrM[j].x() <<";"<< ptrM[j].y() <<endl; }

delete [] ptrM ;

system("pause");

return 0;

}



Використання списку ініціалізаторів членів **переважніше**, ніж встановлення значень в тілі конструктора, оскільки він безпосередньо ініціалізує елемент. В наданому прикладі показано список ініціалізаторів членів, який складається зі всіх виразів ***ідентифікаторів (аргументів)*** після двокрапки (":"). Ідентифікатор повинен посилатися на член класу; він ініціалізується значенням аргументу. Аргумент може бути одним з параметрів конструктора, викликом функції.

*Список ініціалізації — єдиний спосіб задати значення константних змінних-членів.* Крім того, така ініціалізація буває більш ефективною. Правда, якщо клас містить велику кількість членів, список може стати занадто довгим, що знизить наочність програми.

Можна описати будь-яку кількість конструкторів класу. Вони повинні відрізнятися кількістю або типами параметрів (властивість поліморфізму).

**class** Country

{

**private**:

**char** name[40];

**double** area;

**int** population;

**public**:

Country(**double** area) { **this**->area = area; }

Country(**const** **char**\* name);

Country(**const** **char**\* name, **double** area);

Country(**const** **char**\* name, **double** area, **int** population);

. . .

};

Country::Country(**const** **char** \* name)

{

strcpy(**this**->name, name);

area = 1; // істотно, щоб територія не мала значення 0

// і не виникало помилки у функції density()

}

Country::Country(**const** **char** \* name, **double** area)

{

strcpy(**this**->name, name);

**this**->area = area;

}

Country::Country(**const** **char** \* name, **double** area, **int** population)

{

strcpy(**this**->name, name);

**this**->area = area;

**this**->population = population;

}

**Конструктор копіювання** ініціалізує стан класу значення іншого екземпляру цього класу. В списку параметрів вказується єдиний параметр, що має тип “посилання на екземпляр класу”. Для явного визначення конструктора копіювання необхідно дотримуватись особливого синтаксису:

**<ім’я\_класу> (const <ім’я\_класу> & );**

Параметром конструктора копіювання є стала змінна-посилання на екземпляр класу. Його призначення – коректне створення копії екземпляру. Особливо важливо це у випадку, коли членами класу є вказівники, пам’ять під які виділяється оператором **new**. Адже тоді поелементне копіювання копіює вказівник (поверхневе копіювання), а не об’єкт, на який він посилається, – для цього необхідне глибоке копіювання.

*Конструктор копіювання потрібен нам для того, щоб створювати «реальні» копії об'єктів класу, а не побітову копію об'єкта*. Іноді це принципово важливо. Таку «реальну» копію об'єкта треба створювати в декількох випадках:

* коли ми передаємо об'єкт в яку-небудь функцію у вигляді параметра;
* коли яка-небудь функція повинна повернути об'єкт класу в результаті своєї роботи;
* коли ми в головній функції один об'єкт класу ініціалізуємо іншим об'єктом класу.

Наприклад, ми передаємо об'єкт в функцію у вигляді параметра. Функція буде працювати не з самим переданим об'єктом, а з його побітової копією. Припустимо в конструкторі класу, при створенні об'єкта, виділяється певний об'єм пам'яті, а деструкція класу цю пам'ять звільняє. Покажчик побітової копії об'єкта буде зберігати ту ж адресу пам'яті, що й оригінальний об'єкт. І, коли при завершенні роботи функції і знищенні побітової копії об'єкта, спрацює деструктор, він обов'язково звільнить пам'ять, яка була зайнята об'єктом-оригіналом. На додачу, ще й при завершенні роботи програми, деструктор спрацює повторно і спробує ще раз звільнити цей обсяг пам'яті, що неминуче призведе до помилок програми. Та ж доля спіткає і пам'ять, виділену для покажчика об'єкта, якщо буде віддалятися побітова копія повертається функцією об'єкта, і побітова копія при ініціалізації об'єкту класу іншим об'єктом.

Щоб уникнути цих проблем і помилок існує конструктор копіювання. Його робота полягає в тому, щоб створити реальну копію об'єкта зі своєю особистою виділеної динамічною пам'яттю.

Розглянемо конструктор копіювання.

#include <iostream>

**#include <cassert>**

class Drob

{

private:

    int m\_numerator;

    int m\_denominator;

public:

    // Конструктор з параметрами

    Drob(int numerator=0, int denominator=1) :

        m\_numerator(numerator), m\_denominator(denominator)

    {

**assert**(denominator != 0);

    }

    // Конструктор копіювання

    Drob(const Drob &drob) :

        m\_numerator(drob.m\_numerator), m\_denominator(drob.m\_denominator)

/\* Примітка: Ми маємо прямий доступ до членів об‘єкту drob, оскільки ми зараз знаходимося всередині класу Drob \*/

    {

/\*Нема необхідності виконувати перевірку denominator тут, оскільки ця перевірка вже здійснюється в конструкторі класу Drob\*/

        std::cout << "Copy constructor worked here!\n"; /\* просто, щоб показати, що це працює\*/

    }

*friend std::ostream& operator<<(std::ostream& out, const Drob &d1);*

};

*std::ostream& operator<<(std::ostream& out, const Drob &d1)*

{

out << d1.m\_numerator << "/" << d1.m\_denominator;

return out;

}

int main()

{

Drob sixSeven(6, 7); /\* пряма ініціалізація об‘єкту класу Drob, викликається конструктор Drob(int, int)\*/

Drob dCopy(sixSeven); /\* пряма ініціалізація, викликається конструктор копіювання класу Drob\*/

std::cout << dCopy << '\n';

}

Результат виконання програми:

***Застосування конструкторів з параметрами***

#include <stdio.h>

#include <Windows.h>

using namespace std;

class TInner

{

public:

int i;

TInner(int n):i(n){printf("Ctor Inner\n");}

TInner(TInner& x){ \*this = x; printf("Copy ctor TInner\n");}

~TInner(){printf("Dtor TInner\n");}

};

class TBase1

{

public:

double a;

**TInner b;**

TBase1(int x, double y):b(x),a(y){printf("Ctor TBase1\n");}

TBase1(TBase1& x):b(10){ \*this = x; printf("Copy ctor TBase1");}

~TBase1(){printf("Dtor TBase1\n");}

void printBase() {printf("TBase1::TInner::i = %d a = %lf\n",b.i, a);}

};

class TBase2

{

public:

char c;

TBase2(char x):c(x){printf("Ctor TBase2\n");}

TBase2(TBase2& x){\*this = x; printf("Copy ctor TBase2");}

~TBase2(){printf("Dtor TBase2\n");}

void printBase2() {printf("TBase2::c = c = %c\n",c);}

};

class TDerived: public TBase1, public TBase2

{

public:

float f;

**TDerived(float x):TBase1(20.0, 30),TBase2('X')**

{ f = x;printf("Ctor TDerived\n"); }

~TDerived(){printf("Dtor TDerived\n");}

void printDerived2() {printf("TDerived2::%lf\n",f);}

};

int main()

{system("color F0");

TDerived obj(15.0);

printf("TBase1::a = %lf \n",obj.a);

printf("TBase1::b = %d \n",**obj.b.i**);

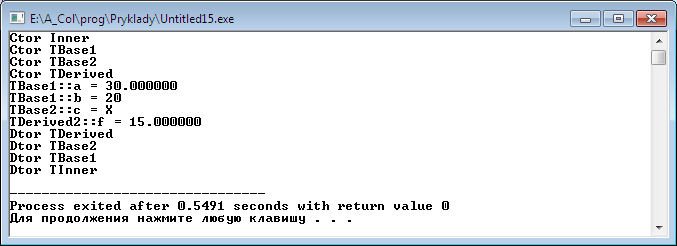
printf("TBase2::c = %c \n",obj.c);

printf("TDerived2::f = %lf \n",obj.f);

return 0;

}

Результат роботи



**Передача параметрів конструкторам базових класів**

#include <stdio.h>

#include <Windows.h>

using namespace std;

class TInner

{

public:

int i;

TInner(int n):i(n){printf("Ctor Inner\n");}

TInner(TInner& x){ \*this = x; printf("Copy ctor TInner\n");}

~TInner(){printf("Dtor TInner\n");}

};

class TBase1

{

public:

double a;

***TInner b;***

TBase1(int x, double y):b(x),a(y){printf("Ctor TBase1\n");}

TBase1(TBase1& x):b(10){ \*this = x; printf("Copy ctor TBase1");}

~TBase1(){printf("Dtor TBase1\n");}

void printBase() {printf("TBase1::TInner::i = %d a = %lf\n",b.i, a);}

};

class TBase2

{

public:

char c;

TBase2(char x):c(x){printf("Ctor TBase2\n");}

TBase2(TBase2& x){\*this = x; printf("Copy ctor TBase2");}

~TBase2(){printf("Dtor TBase2\n");}

void printBase2() {printf("TBase2::c = c = %c\n",c);}

};

class TDerived: public TBase1, public TBase2

{

public:

float f;

**TDerived(int x, double y, char z, float w):TBase1(x, y),TBase2('z')**

{ f = w; printf("Ctor TDerived\n"); }

~TDerived(){printf("Dtor TDerived\n");}

void printDerived2() {printf("TDerived2::%lf\n",f);}

};

int main()

{ system("color F0");

**TDerived obj(1.0, 2, 'Y', 3.0);**

printf("TBase1::a = %lf \n",obj.a);

printf("TBase1::b = %d \n",obj.b.i);

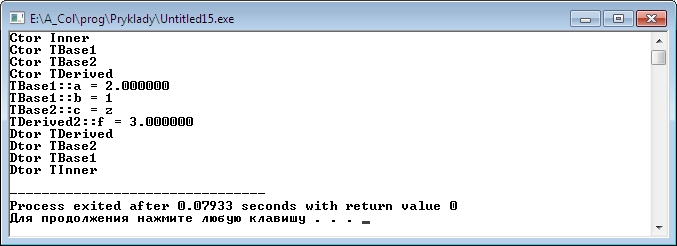
printf("TBase2::c = %c \n",obj.c);

printf("TDerived2::f = %lf \n",obj.f);

return 0;

}

Результат роботи



***Контрольні запитання*.**

1. Які типи відношень між класами Ви знаєте?
2. Як пов’язані відношення агрегації і композиції?
3. Наведіть приклад відношення узагальнення.
4. Визначте основні концепції наслідування.
5. В чому полягає принцип підстановки?
6. Які форми наслідування Ви можете визначити?
7. В чому полягає просте спадкування?
8. Як визначається поведінка конструкторів та деструкторів при спадкуванні?

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

**Література**

1. Страуструп Б. Язык программирования С++. — СПб.; М.: Невский диалект — ЗАО “Изд-во “Бином”, 1999.
2. 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. 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.*