**Лабораторна робота №14. Розробка програм із застосуванням поліморфізму**

**Мета:** Розробка програм із застосуванням поліморфізму

**Порядок виконання роботи**

1. Ознайомитися з теоретичними основами розробки віртуальних функцій та застосуванням поліморфізму.
2. Розробити клас Point для завдання координати точки на площині. Вибираючи цей клас в якості базового, розробити похідні класи Circle, що визначає кола різного радіусу, Line, яка визначає довжину лінії.. Визначити в класі Circle функції повертають довжину кола і площа круга. У всіх класах оголосити віртуальну функцію Length, яка повертає довжину відповідного об'єкта. У головній програмі оголосити масив покажчиків на об'єкти точок і кіл. Вивести середнє арифметичне довжин кіл і середнє арифметичне їх площ. Оформити однією програмою, до якої включити опис класу.
3. Продовжити працювати над програмою лабораторної роботи №11, закінчити попередні завдання з урахуванням наданих зауважень (для тих, хто не здав). Результати надсилати разом зі сформованими файлами.
4. В першому рядку програми та заголовкового файлу повинні бути записаними в коментарі номер групи та прізвище, а також номер ЛР (через кому до попередньої).
5. Результати надсилати на електронну адресу викладача

[**t.i.lumpova@gmail.com**](mailto:t.i.lumpova@gmail.com)у вигляді заголовкового файлу та cpp-файлу з іменем у форматі

Для заголовкового файлу

**<Прізвище англійською>.h**

Наприклад, Ivanov.h

Для cpp-файлу:

**<Номер групи> <Номер лабораторної><Прізвище англійською>.cpp**

Наприклад, 31-01Ivanov.cpp.

Тему в заголовку листа записати

**ООП<Номер групи>-ЛР <Номер лабораторної>-<Прізвищеанглійською>**

**Строк відсилки ЛР ІПЗ-31 10.10.2022**

Всі запитання, що виникнуть, надсилайте на електронну адресу викладача, тему в заголовку листа записати

**ООП<Номер групи>-Запитання-<Прізвище англійською>**.

**Теоретичні відомості.**

Розглянемо приклад ієрархії класів

**#include <iostream>**

**#include <Windows.h>**

**class Animal**

**{**

**public:**

**Animal() { std::cout << "Animal::Animal" << std::endl; }**

**virtual ~Animal() { std::cout << "Animal::~Animal" << std::endl; }**

**virtual void DoSound() {}**

**};**

**class Cat : public Animal**

**{**

**public:**

**Cat() { std::cout << "Cat::Cat" << std::endl; }**

**~Cat() { std::cout << "Cat::~Cat" << std::endl; }**

**void DoSound() { std::cout << "Mew!" << std::endl; }**

**};**

**class Dog : public Animal**

**{**

**public:**

**Dog() { std::cout << "Dog::Dog" << std::endl; }**

**~Dog() { std::cout << "Dog::~Dog" << std::endl; }**

**void DoSound() { std::cout << "Wow" << std::endl; }**

**};**

**class Frog : public Animal**

**{**

**public:**

**Frog() { std::cout << "Frog::Frog" << std::endl; }**

**~Frog() { std::cout << "Frog::~Frog" << std::endl; }**

**void DoSound() { std::cout << "Croc!" << std::endl; }**

**};**

**int main()**

**{ system("color F0");**

**Animal\* animal1 = new Cat();**

**Animal\* animal2 = new Dog();**

**Animal\* animal3 = new Frog();**

**animal1->DoSound();**

**animal2->DoSound();**

**animal3->DoSound();**

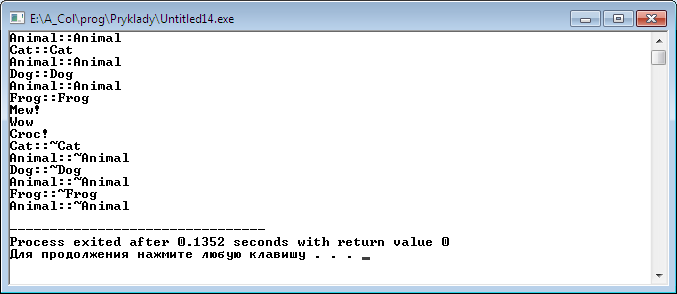
**delete animal1;**

**delete animal2;**

**delete animal3;**

**return 0;**

**}**



**Як це працює?**

Компілятор генерує код який дозволяє програмі на стадії виконання обрати необхідну віртуальну функцію в залежності від класу об'єкту. Для цього він додає деяку службову інформацію на стадії компіляції. Один з найрозповсюдженіших методів вирішення цієї задачі це - використання таблиці віртуальних методів.

Для того щоб зрозуміти це, перш за все, потрібно зрозуміти, що таке класи і об'єкти з точки зору компілятора.

**Oб'єкти з точки зору компілятору**

Розмір об'єктів які не містять віртуальні методу у пам'яті дорівнює сумі його членів даних. Звичайні методи не потребують додаткового місця у пам'яті об'єкту.

class A

{

int x;

void m1() {}

void m2() {}

};

A obj; // sizeof(obj) == 4 (1 машинне слово для 32-бітної платформи)

obj.x = 0x12345678;

Тепер розглянемо приклад для класу з віртуальними методами:

class B

{

int x;

void m1() {}

void m2() {}

virtual void vm1() {}

virtual void vm2() {}

};

B obj; // sizeof(obj) == 8 (2 машинних слова для 32-бітної платформи)

obj.x = 0x12345678;

Додаткове місце за зміщенням 0 займає вказівник на таблицю віртуальних методів (службова інформація додана компілятором).

**Таблиця віртуальних методів**

Якщо у класі присутні віртуальні методи то компілятор створює спеціальну структуру яка називається таблицею віртуальних методів. Така таблиця існує у одному екземплярі для кожного класу і містить вказівники на віртуальні методи класу.

Таблиця віртуальних методів класу B:

|  |  |
| --- | --- |
| Зміщення | Метод |
| +0 | B::vm1 |
| +4 | B::vm2 |

**Таблиця віртуальних методів**

Після цього компілятор додає вказівник на таблицю віртуальних методів у кожний об'єкт класу. Розглянемо на прикладі об'єкту класу B:

При виклику віртуального методу програма знаходить вказівник на віртуальну таблицю, а потім викликає потрібний метод з таблиці.

|  |  |
| --- | --- |
| Зміщення | Дані |
| +0 | Вказівник на віртуал. таблицю класу B |
| +4 | int x |

|  |  |
| --- | --- |
| Зміщення | Метод |
| +0 | B::vm1 |
| +4 | B::vm2 |

Вказівник на таблицю віртуальних методів ініціалізується компілятором при конструюванні об'єкту.

Повернемося до прикладу:

Animal\* animal1 = new Cat();

У цьому випадку спочатку конструюється Animal і вказівник на таблицю віртуальних методів ініціалізується таблицею Animal. Після цього конструєються Cat і вказівник на таблицю віртуальних методів переписується таблицею Cat.

Таким чином при виклику:

animal1->DoSound();

викликається метод саме з класу Cat.

Це називається пізнім зв'язуванням.

У стандарті C++11 було введено два додаткових ключових слова:

* **override** - дозволяє створення віртуального методу у класі нащадку лише якщо метод з такою ж сигнатурою є у батьківському класі
* **final** - забороняє перевизначати віртуальний метод у класах нащадках

**Динамічна ідентифікація типів (run-time type identification –RTTI)**

В такій поліморфній мові, як C++,можливі ситуації, в яких тип об'єкта невідомий у період компілювання, оскільки точна природа цього об'єкта не буде визначена доти, доки програма на почне виконуватися. Як уже зазначалося вище, мова програмування C++ реалізує поліморфізм за допомогою використання ієрархії класів, віртуальних функцій і покажчиків на об'єкти базових класів. Покажчик на базовий клас можна використовувати для посилання на члени як цього базового класу, так і на члени будь-якого об'єкта, виведеного з нього. Отже, не завжди наперед відомо, на об'єкт якого типу посилатиметься покажчик на базовий клас у довільний момент часу. Це з'ясується тільки у процесі виконання програми – при використанні одного із засобів динамічної ідентифікації типів.

**Отримання типу об'єкта у процесі виконання програми**

Для цього необхідно приєднати до програми заголовок <**typeinfо**>. Найпоширеніший формат використання оператора **typeid** такий:

**typeid**(*object*)

У цьому записі елемент *object* означає об'єкт, тип якого потрібно отримати. Можна робити запити не тільки про вбудований тип, але і про тип класу, створеного програмістом. Оператор **typeid** повертає посилання на об'єкт типу **type\_info**, який описує тип об'єкта *object*.

Якщо оператор **typeid** застосовується до покажчика на поліморфний базовий клас (пригадайте: поліморфний клас – це клас, який містить хоч би одну віртуальну функцію), він автоматично повертає тип реального об'єкта, на який той вказує: будь то об'єкт базового класу або об'єкт класу, виведеного з базового.

Отже, оператор **typeid** можна використовувати для динамічного визначення типу об'єкта, який адресується покажчиком на базовий клас. Застосування цієї можливості продемонстровано в такому коді програми.

**Демонстрація механізму застосування оператора typeid до**

**ієрархії поліморфних класів**

**#include <iostream>** // Для потокового введення-виведення

**#include** <**typeinfo**> // Для динамічної ідентифікації типів

**#include <Windows.h>**

**using namespace std;** // Використання стандартного простору імен

// Оголошення базового класу

**class** Base {

**virtual void** Fun() {}; // Робимо клас Base поліморфним

//. . .

};

**class** DerivedA: **public** Base {

//. . .

};

**class** DerivedB: **public** Base {

//. . .

};

**int main**()

{ system("color F0");

Base \*p, baseob;

DerivedA ObjA; // Створення об'єкта класу

DerivedB ObjB; // Створення об'єкта класу

p = &baseob;

**cout** << " Variable p show object ";

**cout** << **typeid**(\*p).**name**() << **endl**;

p = &ObjA;

**cout** << " Variable p show object ";

**cout** << **typeid**(\*p).**name**() << **endl**;

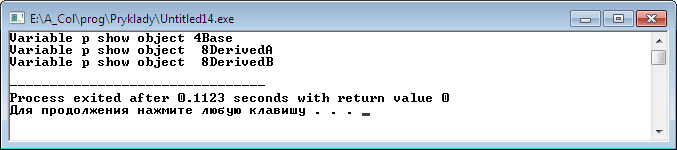
p = &ObjB;

**cout** << " Variable p show object ";

**cout** << **typeid**(\*p).**name**() << **endl**;

**getch**(); **return** 0;

}



Якщо оператор **typeid** застосовується до покажчика на базовий клас поліморфного типу, то тип об'єкта, який реально адресується, як підтверджують ці результати, буде визначений у процесі виконання програми.

У всіх випадках застосування оператора **typeid** до покажчика на неполіморфну ієрархію класів буде отримано покажчик на базовий тип, тобто те, на що цей покажчик реально вказує. Як експеримент спробуйте перетворити на коментар віртуальну функцію Fun() у класі Base і подивіться на результат. Ви побачите, що тип кожного об'єкта після внесення у програму цієї зміни буде визначений як Base, оскільки саме цей тип має покажчик р.

Оскільки оператор **typeid** зазвичай застосовується до перейменованого покажчика (тобто до покажчика, до якого вже застосовано оператор "\*"), то для оброблення ситуації, коли цей перейменований покажчик виявиться нульовим, створено спеціальний виняток. У цьому випадку оператор **typeid** генерує виняток типу

**bad**\_**typeid**.

Посилання на об'єкти ієрархії поліморфних класів працюють подібно до покажчиків. Якщо оператор **typeid** застосовується до посилання на поліморфний клас, то він повертає тип об'єкта, на який вона реально посилається, і це може бути об'єкт не базового, а похідного типу. Описаний засіб найчастіше використовують при передачі об'єктів функціям за посиланням. Наприклад, у наведеному нижче коді програми функція WhatType() оголошує посилальний параметр на об'єкти типу Base. Це означає, що функції WhatType() можна передавати посилання на об'єкти типу Base або посилання на об'єкти будь-яких класів, похідних від Base. Оператор **typeid**, що застосовується до такого параметра, поверне реальний тип об'єкта, який передається функції.

**Демонстрація механізму застосування оператора typeid до посилального параметра**

**#include <iostream>** // Для потокового введення-виведення

**#include** <**typeinfo**> // Для динамічної ідентифікації типів

**using namespace std;** // Використання стандартного простору імен

// Оголошення базового класу

**class** Base {

**virtual void** Fun() {; // робимо клас Base поліморфним

//. . .

};

**class** DerivedA: **public** Base {

//. . .

};

**class** DerivedB: **public** Base {

//. . .

};

// Демонструємо застосування оператора **typeid** до посилального параметра.

**void** WhatType(Base &obj)

{

**cout** << "Параметр obj посилається на об'єкт типу ";

**cout** << **typeid**(obj).**name**() << **endl**;

}

**int main**()

{

**int** c;

Base baseob;

DerivedA ObjA; // Створення об'єкта класу

DerivedB ObjB; // Створення об'єкта класу

WhatType(baseob);

WhatType(ObjA);

WhatType(ObjB);

**getch**(); **return** 0;

}

Внаслідок виконання ця програма відображає на екрані такі результати:

Параметр obj посилається на об'єкт типу Base

Параметр obj посилається на об'єкт типу DerivedA

Параметр obj посилається на об'єкт типу DerivedB

Існує ще одна версія застосування оператора **typeid**, яка як аргумент приймає ім'я типу. Формат його є таким:

**typeid**(*ім'я\_типа*)

Наприклад, наступна настанова абсолютно допускається:

**cout** << **typeid**(**int**).**name**();

Призначення цієї версії оператора **typeid** – отримати об'єкт типу **type\_info** (який описує заданий тип даних), щоб його можна було використовувати в настанові порівняння типів.

Інтерфейс для класу **type\_info**

class type\_info {

public:

type\_info(const type\_info& rhs) = delete; // cannot be copied

virtual ~type\_info();

size\_t hash\_code() const;

\_CRTIMP\_PURE **bool operator==(**const type\_info& rhs) const;

type\_info& operator=(const type\_info& rhs) = delete; // cannot be copied

\_CRTIMP\_PURE **bool operator!=(**const type\_info& rhs) const;

\_CRTIMP\_PURE **int before**(const type\_info& rhs) const;

size\_t hash\_code() const noexcept;

\_CRTIMP\_PURE **const char\* name()** const;

\_CRTIMP\_PURE const char\* raw\_name() const;

};

Оператори == и != можна використовувати для порівняння на рівність та нерівність з іншими об’єктами типу **type\_info**.

Функцію type\_info::before можна використовувати для визначення порядку сортування типів, вона не пов’язана з ієрархією класів .

Функція-член type\_info::name повертає const char\* в рядкову змінну (в кінці нуль-символ) ім’я типу.

Приклад.

#include <iostream>

#include <typeinfo>

using namespace std;

class myclass {

// . . .

};

int main()

{

int i, j;

float f;

myclass ob;

cout << "Тип переменной i: " << typeid(i).name();

cout << endl;

cout << "Тип переменной f: " << typeid(f).name();

cout << endl;

cout << "Тип переменной ob: " << typeid(ob).name();

cout << "\n\n";

if(typeid(i) == typeid(j))

cout << "Типы переменных i и j одинаковы.\n";

if(typeid(i) != typeid(f))

cout << "Типы переменных i и f неодинаковы.\n";

return 0;

}

При виконанні цієї програми отримаємо результати.

Тип переменной i: int

Тип переменной f: float

Тип переменной ob: class myclass

Типы переменных i и j одинаковы.

Типы переменных i и f неодинаковы.