Міністерство освіти і науки

Національний університет “Львівська політехніка”

**Кафедра ЕОМ**



**Звіт**

з лабораторної роботи № 7

# з дисципліни: “ Програмування, частина 2 (Об'єктно-орієнтоване програмування”

на тему: “ МНОЖИННЕ СПАДКУВАННЯ. ПОЛІМОРФІЗМ”

Виконав: ст. гр. КІ-15 Гвоздь Ю. А.

Прийняв: Козак Н. Б.

Львів – 2020

Мета: познайомитися із множинним спадкуванням класів та поліморфізмом.

**ТЕОРЕТИЧНІ ВІДОМОСТІ**

Множинне спадкування

Якщо спадкування здійснюється від декількох батьківських класів одночасно, тоді воно називається множинним спадкуванням. Визначальним для похідного класу породженого множинним спадкуванням є те, що він явно чи неявно повинен успадковувати характеристики декількох базових класів. Основні принципи одинарного спадкування, зокрема спадкування членів, модифікаторів доступу до членів базових класів, розширення та обмеження характеристик, без жодних доповнень можуть бути перенесені на множинне спадкування. Неявним множинним спадкуванням можуть бути випадки змішаного спадкування. Результатом цих спадкуваннь є ієрархія, в якій похідний клас неявно (через один проміжний) успадкував характеристики двох базових класів. Якщо похідний клас породжується від декількох базових, то в декларації класу треба вказати усі базові класи, розділяючи

Поліморфізм Поліморфізм – це здатність коду при постійному інтерфейсі змінювати свою поведінку в залежності від ситуації, яка виникає на момент виконання. Іншими словами один і той же метод може бути визначений для об‘єктів різних класів, що є між собою в ієрархії спадкування, при цьому метод якого класу викликати вирішується під час виконання програми. Поліморфізм широко застосовується при створені складних бібліотек класів та програмних систем. Один продуманий інтерфейс може бути з успіхом використаний у різних ситуаціях, що зменшує складність програмної структури завдяки уніфікації інтерфейсу. Спрощення створення програмного інтерфейсу дає можливість розробнику "тримати" в пам'яті меншу кількість інформації. Загалом розрізняють два види поліморфізму: статичний (static або compilation-time polymorphism) і динамічний (run-time polymorphism). Статичний поліморфізм реалізовується за допомогою так званого раннього зв’язування через механізм перевантаження функцій, методів та операторів і віртуальні класи. Він притаманний класичним структурним мовам програмування. При використанні статичного поліморфізму вибір функції чи методу, що буде викликатися здійснюється компілятором при компіляції програми (раннє зв‘язування). Вибір функції чи методу в даному випадку залежить від типу вказівника чи посилання і не залежить від типу реального об‘єкту на який вказує вказівник чи посилається посилання. Тобто при звертанні до методу об'єкту похідного класу, використовуючи вказівник чи посилання на базовий клас, викликається буде метод базового класу. Раннє зв‘язування реалізується наступним чином. Компілятор на основі вихідного коду використовує фіксовані ідентифікатори функцій і методів. На етапі компонування ці ідентифікатори замінюються фізичними адресами відповідних функцій і методів. Раннє зв'язування характеризується високою швидкістю виконання програми, оскільки єдиними витратами в період виконання є: передавання аргументів, виконання виклику функції або методу та очищення стеку. Основним недоліком раннього зв'язування є втрати гнучкості виконуваного коду. Динамічний поліморфізм реалізовується за допомогою так званого пізнього зв’язування через механізм віртуальних функцій. Він притаманний об‘єктно-орієнтованим мовам програмування і може бути застосований лише до методів класів. При використанні динамічного поліморфізму вибір методу, що буде викликатися здійснюється в процесі виконання програми (пізнє зв‘язування). Вибір методу в даному випадку залежить від типу реального об‘єкту на який вказує вказівник чи посилається посилання. Тобто при звертанні до методу об'єкту похідного класу, використовуючи вказівник чи посилання на базовий клас, викликатися буде метод похідного класу. Динамічний поліморфізм дозволяє значно підвищити ефективність і гнучкість поведінки програми, але ціною за це є сповільнення виконання програми, оскільки в процесі виконання програмі додатково необхідно приймати рішення який метод у якій ситуації викликати. Тому використання механізму пізнього зв‘язування має сенс лише за умови існування ієрархії класів. В іншому випадку єдине що ми отримаємо – це сповільнення виконання програми. Пізнє зв‘язування при одинарному спадкуванні реалізується за допомогою таблиці віртуальних функцій (ТВФ) та вказівника vptr. Для кожного класу, що містить віртуальні функції в процесі роботи програми створюється тільки одна ТВФ. Вказівник vptr в компіляторах компанії Microsoft завжди міститься за нульовим зміщенням відносно початку об‘єкта в усіх об‘єктах, що мають віртуальні функції незалежно від того чи то об‘єкти базового чи похідного класу і часто називається \_\_vfptr, а сама ТВФ називається vftable. Таблиця віртуальних функцій – це масив, кожен елемент якого містить вказівник на метод. Отримати доступ до вказівника vptr засобами мови С++ неможливо. ТВФ створюється за допомогою конструктора класу. Якщо класи знаходяться в одинарній ієрархії, то спочатку конструктор створює ТВФ для базового класу і в неї включаються всі віртуальні функції базового класу, а вказівнику vptr присвоюється адреса цієї таблиці. Віртуальні методи у ТВФ вносяться в порядку слідування, тому компілятор завжди може однозначно визначити де знаходиться той чи інший метод не зберігаючи його назви. Далі викликається конструктор для похідного класу і створюється копія ТВФ базового класу в якій при необхідності змінюються адреси перевизначених віртуальних методів та додаються нові знову ж таки в порядку слідування починаючи з індекса на якому закінчилася індексація віртуальних методів базового класу, а vptr присвоюється адреса цієї ТВФ. Процес продовжується доти, поки не створиться останній об‘єкт похідного класу. При виклику віртуального методу компілятор генерує код звернення за вказівником vptr до ТВФ, за допомогою черговості оголошення методу визначає індекс методу в ТВФ і в необхідне місце програми вставляє код непрямого виклику методу за адресою, що міститься в ТВФ за відповідним зміщенням (у відповідній комірці масиву, який реалізує ТВФ). При знищені об‘єкту процес відбувається у зворотньому порядку відносно створення об‘єкту – спочатку знищується об‘єкт похідного класу, а vptr присвоюється адреса об‘єкту на рівень вище в ієрархії класів і т.д. поки не знищиться найвищий в ієрархії об‘єкт класу. У випадку якщо використовується множинне спадкування, то для кожного класу, що містить множинне спадкування додаються стільки вказівників vptr, скільки класів спадкує похідний класу. Це робиться для того, щоб кожен з базових класів можна було використати в якості вказівника чи посилання. Мова С++ є гібридною мовою програмування в плані підтримки поліморфізму, оскільки в ній реалізована підтримка як статичного так і динамічного поліморфізму. Тож відповідно до типу зв'язування в С++ розрізняють звичайні функції і методи – функції і методи раннього зв'язування і поліморфні або віртуальні методи (virtual methods) - методи пізнього зв'язування. Для того, щоб оголосити віртуальний метод треба перед оголошенням методу поставити ключове слово virtual. Для того, щоб метод став віртуальним достатньо вжити ключове слово virtual лише раз у всій ієрархії спадкування. Зазвичай це робиться у базовому класі, а в похідних класах цей метод перевизначається як звичайний метод класу. Віртуальна функція є членом класу, а тому може викликатись як звичайний метод класу.

Завдання: Спроектувати і реалізувати ієрархію класів, що описують предметну область згідно варіанту, яка реалізується класом 1. Клас 1 в свою чергу утворюється шляхом множинного спадкування класів 2 і 3 кожен з яких в свою чергу успадковує клас 4. Додаткові вимоги:

1. Базовий клас містить мінімум один віртуальний метод, один невіртуальний метод і одну динамічно створювану властивість.
2. Забезпечити механізми коректної роботи конструкторів і деструкторів.
3. Перевантажити оператор присвоєня з метою його коректної роботи.
4. Кожен з класів має містити мінімум одину властивість і 4 методи.
5. Написати main() функцію де створити об‘єкт класу 1 і продемонструвати різницю між статичним і динамічним поліморфізмом.



Виконання завдання:

Код програми:

***Source.cpp***

#include "AssaultPlane.h"

using namespace std;

int main()

{

cout << "----- Creating objects -----" << endl;

CFighter f1("MIG-1", "Minigun");

f1.SetAirage(20.1);

f1.ShowFullInformation();

CBomber b1("WG-5", "FN-12");

b1.SetAirage(10.0);

b1.ShowFullInformation();

CAssaultPlane a1("IL-2", "Minigun-2", "FN-14");

a1.SetAirage(0.9);

a1.ShowFullInformation();

CAssaultPlane a2 = a1;

a2.ShowFullInformation();

CAssaultPlane a3(a2);

a3.ShowFullInformation();

f1 = a3;

f1.ShowFullInformation();

{

CAssaultPlane a4(a2);

a4.ShowFullInformation();

}

//------------------------------------------------------------------

cout << "----- Creating reference -----" << endl;

cout << "----- CPlane &ref1 = f1; -----" << endl;

CPlane& ref1 = f1;

ref1.ShowFullInformation();

cout << "----- ref1 = b1; -----" << endl;

ref1 = b1;

ref1.ShowFullInformation();

cout << "----- CPlane &ref2 = b1; -----" << endl;

CPlane& ref2 = b1;

ref2.ShowFullInformation();

cout << "----- CPlane &ref3 = a1; -----" << endl;

CPlane& ref3 = a1;

ref3.ShowFullInformation();

cout << "----- Creating dynamic objects -----" << endl;

cout << "----- CPlane\* ptr1 = new CFighter(\"F-2\", \"F-Gun\") -----" << endl;

CPlane\* ptr1 = new CFighter("F-2", "F-Gun");

ptr1->SetAirage(1.2);

ptr1->ShowArmament();

ptr1->ShowFunctionality();

ptr1->ShowFullInformation();

cout << "----- CPlane\* ptr2 = new CBomber(\"B-2\", \"Bomb-2\"); -----" << endl;

CPlane\* ptr2 = new CBomber("B-2", "Bomb-2");

ptr2->SetAirage(0.2);

ptr2->ShowArmament();

ptr2->ShowFunctionality();

ptr2->ShowFullInformation();

cout << "----- CPlane\* ptr3 = new CAssaultPlane(\"A-2\", \"F-Gun\", \"Bomb-2\"); -----" << endl;

CPlane\* ptr3 = new CAssaultPlane("A-2", "F-Gun", "Bomb-2");

ptr3->SetAirage(4.2);

ptr3->ShowArmament();

ptr3->ShowFunctionality();

ptr3->ShowFullInformation();

//------------------------------------------------------------------

cout << "delete ptr1 " << endl;

delete ptr1;

cout << "-------------------------" << endl << "delete ptr2 " << endl;

delete ptr2;

cout << "-------------------------" << endl << "delete ptr3 " << endl;

delete ptr3;

cout << "-------------------------" << endl;

return 0;

}

***AssaultPlane.cpp***

#include "AssaultPlane.h"

CAssaultPlane::CAssaultPlane() :

CPlane(), CFighter(), CBomber()

{

cout << "CAssaultPlane Default Constructor" << endl;

}

CAssaultPlane::CAssaultPlane(const char\* model, const char\* gunModel, const char\* bombModel) :

CPlane(model), CFighter(model, gunModel), CBomber(model, bombModel)

{

cout << "CAssaultPlane Constructor" << endl;

}

CAssaultPlane::CAssaultPlane(const CAssaultPlane& obj) :

CPlane(obj), CFighter(obj), CBomber(obj)

{

cout << "CAssaultPlane Copy Constructor" << endl;

}

CAssaultPlane::~CAssaultPlane()

{

cout << "CAssaultPlane Destructor" << endl;

}

void CAssaultPlane::ShowArmament()

{

CFighter::ShowArmament();

CBomber::ShowArmament();

}

void CAssaultPlane::ShowFunctionality()

{

CFighter::ShowFunctionality();

CBomber::ShowFunctionality();

}

void CAssaultPlane::ShowFullInformation()

{

cout << "Model: " << m\_pszModel << endl

<< "Airage: " << m\_dblAirage << " hours" << endl;

cout << "Armament: " << endl;

ShowArmament();

cout << "Functionality: " << endl;

ShowFunctionality();

cout << "-------------------------" << endl;

}

CAssaultPlane& CAssaultPlane::operator=(CAssaultPlane& obj)

{

if (this != &obj)

{

cout << "CAssaultPlane operator=" << endl;

CFighter::operator=(obj);

CBomber::operator=(obj);

}

return \*this;

}

***Bomber.cpp***

#include "Bomber.h"

CBomber::CBomber() :

CPlane(), m\_pszBombModel(new char[8])

{

cout << "CBomber Default Constructor" << endl;

strcpy(m\_pszBombModel, "No bomb");

}

CBomber::CBomber(const char\* model, const char\* bombModel) :

CPlane(model), m\_pszBombModel(new char[strlen(bombModel) + 1])

{

cout << "CBomber Constructor" << endl;

strcpy(m\_pszBombModel, bombModel);

}

CBomber::CBomber(const CBomber& obj) :

CPlane(obj), m\_pszBombModel(new char[strlen(obj.m\_pszBombModel) + 1])

{

cout << "CBomber Copy Constructor" << endl;

strcpy(m\_pszBombModel, obj.m\_pszBombModel);

}

CBomber::~CBomber()

{

cout << "CBomber Destructor" << endl;

delete[] m\_pszBombModel;

}

void CBomber::ShowArmament()

{

cout << "Bomb model: " << m\_pszBombModel << endl;

}

void CBomber::ShowFunctionality()

{

cout << "Bombarding" << endl;

}

void CBomber::ShowFullInformation()

{

cout << "Model: " << m\_pszModel << endl

<< "Airage: " << m\_dblAirage << " hours" << endl;

cout << "Armament: " << endl;

ShowArmament();

cout << "Functionality: " << endl;

ShowFunctionality();

cout << "-------------------------" << endl;

}

CBomber& CBomber::operator=(CBomber& obj)

{

if (this != &obj)

{

cout << "CBomber operator=" << endl;

delete[] m\_pszBombModel;

m\_pszBombModel = new char[strlen(obj.m\_pszBombModel) + 1];

strcpy(m\_pszBombModel, obj.m\_pszBombModel);

CPlane::operator=(obj);

}

return \*this;

}

***Fighter.cpp***

#include "Fighter.h"

CFighter::CFighter() :

CPlane(), m\_pszGunModel(new char[7])

{

cout << "CFighter Default Constructor" << endl;

strcpy(m\_pszGunModel, "No gun");

}

CFighter::CFighter(const char\* model, const char\* gunModel) :

CPlane(model), m\_pszGunModel(new char[strlen(gunModel) + 1])

{

cout << "CFighter Constructor" << endl;

strcpy(m\_pszGunModel, gunModel);

}

CFighter::CFighter(const CFighter& obj) :

CPlane(obj), m\_pszGunModel(new char[strlen(obj.m\_pszGunModel) + 1])

{

cout << "CFighter Copy Constructor" << endl;

strcpy(m\_pszGunModel, obj.m\_pszGunModel);

}

CFighter::~CFighter()

{

cout << "CFighter Destructor" << endl;

delete[] m\_pszGunModel;

}

void CFighter::ShowArmament()

{

cout << "Gun model: " << m\_pszGunModel << endl;

}

void CFighter::ShowFunctionality()

{

cout << "Shooting" << endl;

}

void CFighter::ShowFullInformation()

{

cout << "Model: " << m\_pszModel << endl

<< "Airage: " << m\_dblAirage << " hours" << endl;

cout << "Armament: " << endl;

ShowArmament();

cout << "Functionality: " << endl;

ShowFunctionality();

cout << "-------------------------" << endl;

}

CFighter& CFighter::operator=(CFighter& obj)

{

if (this != &obj)

{

cout << "CFighter operator=" << endl;

delete[] m\_pszGunModel;

m\_pszGunModel = new char[strlen(obj.m\_pszGunModel) + 1];

strcpy(m\_pszGunModel, obj.m\_pszGunModel);

CPlane::operator=(obj);

}

return \*this;

}

***Plane.cpp***

#include "Plane.h"

CPlane::CPlane() :

m\_pszModel(new char[9]), m\_dblAirage(0.0)

{

cout << "CPlane Default Constructor" << endl;

strcpy(m\_pszModel, "No model");

}

CPlane::CPlane(const char\* model) :

m\_pszModel(new char[strlen(model) + 1]), m\_dblAirage(0.0)

{

cout << "CPlane Constructor" << endl;

strcpy(this->m\_pszModel, model);

}

CPlane::CPlane(const CPlane& obj) :

m\_dblAirage(obj.m\_dblAirage), m\_pszModel(new char[strlen(obj.m\_pszModel) + 1])

{

cout << "CPlane Copy Constructor" << endl;

strcpy(m\_pszModel, obj.m\_pszModel);

}

CPlane::~CPlane()

{

cout << "CPlane Destructor" << endl;

delete[] m\_pszModel;

}

void CPlane::SetAirage(double airage)

{

m\_dblAirage = airage;

}

double CPlane::GetAirage()

{

return m\_dblAirage;

}

void CPlane::ShowArmament()

{

cout << "Armament: nothing" << endl;

cout << "-------------------------" << endl;

}

CPlane& CPlane::operator=(const CPlane& obj)

{

if (this != &obj)

{

cout << "CPlane operator=" << endl;

delete[] m\_pszModel;

m\_pszModel = new char[strlen(obj.m\_pszModel) + 1];

strcpy(m\_pszModel, obj.m\_pszModel);

m\_dblAirage = obj.m\_dblAirage;

}

return \*this;

}

***AssaultPlane.h***

#include "Fighter.h"

#include "Bomber.h"

#ifndef AssaultPlane

#define AssaultPlane

class CAssaultPlane :

public CFighter, public CBomber

{

public:

CAssaultPlane();

CAssaultPlane(const char\* model, const char\* gunModel, const char\* bombModel);

CAssaultPlane(const CAssaultPlane& obj);

virtual ~CAssaultPlane();

void ShowArmament();

virtual void ShowFunctionality();

virtual void ShowFullInformation();

CAssaultPlane& operator=(CAssaultPlane&);

};

#endif

***Bomber.h***

#include "Plane.h"

#ifndef Bomber

#define Bomber

class CBomber :

virtual public CPlane

{

protected:

char\* m\_pszBombModel;

public:

CBomber();

CBomber(const char\* model, const char\* bombModel);

CBomber(const CBomber& obj);

virtual ~CBomber();

void ShowArmament();

virtual void ShowFunctionality();

virtual void ShowFullInformation();

CBomber& operator=(CBomber&);

};

#endif

***Fighter.h***

#include "Plane.h"

#ifndef Fighter

#define Fighter

class CFighter :

virtual public CPlane

{

protected:

char\* m\_pszGunModel;

public:

CFighter();

CFighter(const char\* model, const char\* gunModel);

CFighter(const CFighter& obj);

virtual ~CFighter();

void ShowArmament();

virtual void ShowFunctionality();

virtual void ShowFullInformation();

CFighter& operator=(CFighter&);

};

#endif

***Plane.h***

#include <iostream>

#include <cstring>

using namespace std;

#ifndef Plane

#define Plane

class CPlane

{

protected:

char\* m\_pszModel;

double m\_dblAirage;

public:

CPlane();

CPlane(const char\* model);

CPlane(const CPlane& obj);

virtual ~CPlane();

void SetAirage(double);

double GetAirage();

void ShowArmament();

virtual void ShowFunctionality() = 0;

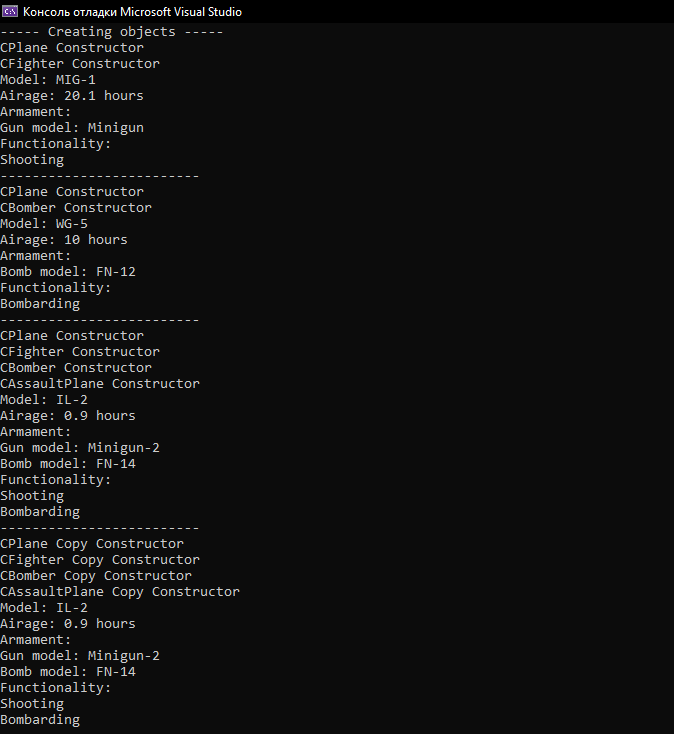
virtual void ShowFullInformation() = 0;

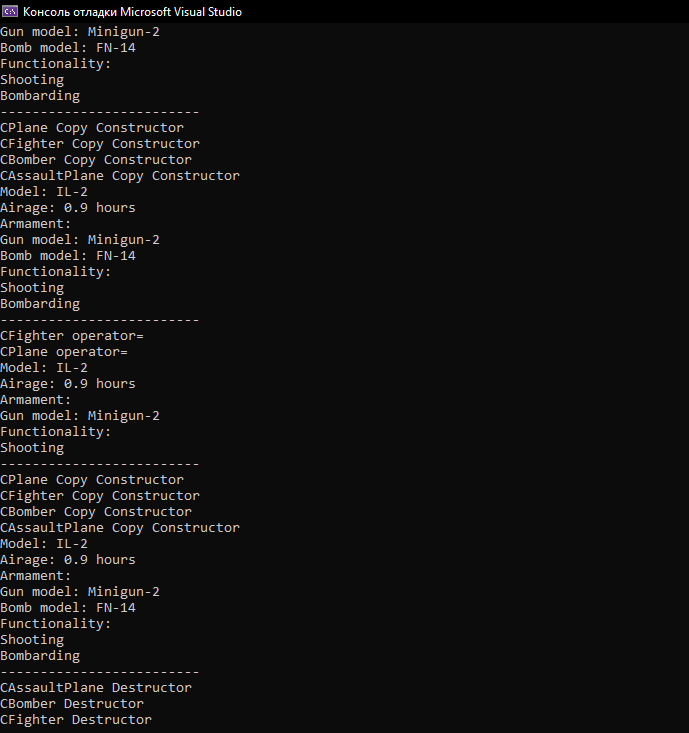
CPlane& operator=(const CPlane&);

};

#endif

Результат роботи програми:





Висновок: на цій лабораторній роботі я познайомився із множинним спадкуванням класів та поліморфізмом.