

# Лекція 16.

## Поліморфізм та віртуальні функції.

# План на сьогодні

1

Види поліморфізму: статичний та динамічний

2

Віртуальні функції та таблиці віртуальних функцій

3

Раннє та пізнє зв'язування

4

Модифікатори `override` та `final`

5

Віртуальні деструктори

6

Чисті віртуальні функції та абстрактні класи

7

Інтерфейси та їх переваги



# Види поліморфізму

# Поліморфізм

Слово «**поліморфізм**» означає наявність багатьох форм. Зазвичай поліморфізм виникає, коли існує ієрархія класів, пов'язаних через успадкування.

Поліморфізм у C++ означає, що виклик функції-члена спричинить виконання різної функції залежно від типу об'єкта, який її викликає.

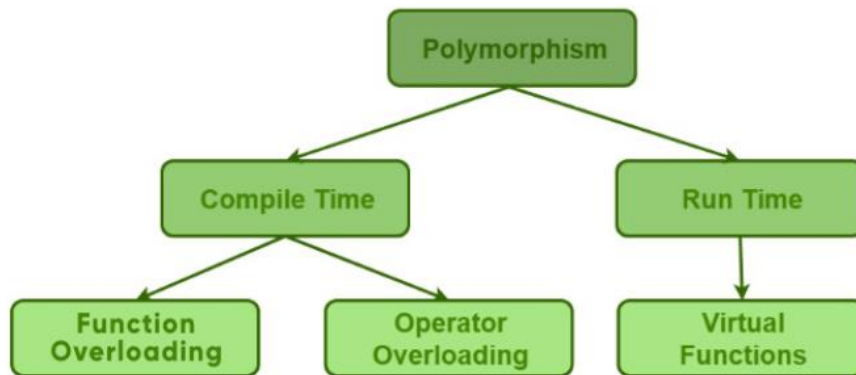
Реальним прикладом поліморфізму є людина, яка може мати **різні ролі одночасно**. Наприклад, чоловік може бути батьком, чоловіком і працівником водночас. Тобто одна й та сама людина демонструє різну поведінку залежно від ситуації. Це і є **поліморфізм**.

*Поліморфізм вважається однією з ключових характеристик об'єктно-орієнтованого програмування.*

# Види поліморфізму

- Статичний (під час компіляції)

- Динамічний (під час виконання)



# Поліморфна змінна

- ❑ Поліморфна змінна, це змінна, яка може набувати значень різних типів.
- ❑ Змінній базового типу можна присвоювати об'єкти похідних типів.

```
class Base{};
class Derived:public Base{};
void main()
{
    Derived d;
    Base b = d; //поліморфна змінна, статичне зв'язування

    int choice = 1;
    Base* b; //поліморфна змінна, динамічне зв'язування
    if (choice==1)
        b = new Base();
    else
        b = new Derived();
}
```

# Поліморфізм під час компіляції (compile-time)



# Поліморфізм під час компіляції

Цей тип поліморфізму досягається за допомогою перевантаження функцій або операторів.



# Перевантаження функції (під час компіляції)

- Коли існує кілька функцій з однаковою назвою, але різними параметрами, такі функції вважаються перевантаженими. Це називається **перевантаженням функцій (Function Overloading)**.
- Функції можна перевантажувати шляхом **зміни кількості аргументів** або/та **зміни їхнього типу**.
- Простими словами, це можливість об'єктно-орієнтованого програмування надавати **кілька функцій з однаковою назвою, але з різними параметрами**, коли під одним ім'ям функції виконуються **різні завдання**.

# Приклад

```
#include <bits/stdc++.h>

using namespace std;
class Test {
public:
    // Function with 1 int parameter
    void func(int x)
    {
        cout << "value of x is " << x << endl;
    }

    // Function with same name but
    // 1 double parameter
    void func(double x)
    {
        cout << "value of x is " << x << endl;
    }

    // Function with same name and
    // 2 int parameters
    void func(int x, int y)
    {
        cout << "value of x and y is " << x << ", " << y << endl;
    }
};
```

```
// Driver code
int main()
{
    Test obj1;

    // Function being called depends
    // on the parameters passed
    // func() is called with int value
    obj1.func(7);

    // func() is called with double value
    obj1.func(9.132);

    // func() is called with 2 int values
    obj1.func(85, 64);
    return 0;
}
```

```
value of x is 7
value of x is 9.132
value of x and y is 85, 64
```

# Пояснення

У наведеному прикладі одна функція з назвою `func()` поводить себе по-різному в трьох різних ситуаціях, що є властивістю поліморфізму.

```
// Driver code
int main()
{
    Test obj1;

    // Function being called depends
    // on the parameters passed
    // func() is called with int value
    obj1.func(7);

    // func() is called with double value
    obj1.func(9.132);

    // func() is called with 2 int values
    obj1.func(85, 64);
    return 0;
}
```

# Перевантаження операторів(під час компіляції)

- С++ має можливість надавати операторам спеціальне значення для певного типу даних. Ця можливість називається **перевантаженням операторів (Operator Overloading)**.
- Наприклад, оператор додавання **(+)** можна використати для класу рядків **(string)**, щоб об'єднувати два рядки. Ми знаємо, що цей оператор зазвичай виконує додавання двох операндів. Однак один і той самий оператор **(+)** **додає числа**, коли використовується з **цілочисловими операндами**, і **об'єднує рядки**, коли застосовується до **рядкових операндів**.

# Приклад

```
#include <iostream>
using namespace std;

class Complex {
private:
    int real, imag;

public:
    Complex(int r = 0, int i = 0)
    {
        real = r;
        imag = i;
    }

    Complex operator+(Complex const& obj)
    {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }

    void print() { cout << real << " + i" << imag << endl; }
};
```

```
// Driver code
int main()
{
    Complex c1(10, 5), c2(2, 4);

    // An example call to "operator+"
    Complex c3 = c1 + c2;
    c3.print();
}
```



12 + i9

# Пояснення

У наведеному прикладі **оператор + перевантажено**. Зазвичай цей оператор використовується для **додавання двох чисел** (цілих або з плаваючою комою), але тут він використовується **для додавання двох уявних або комплексних чисел**.

```
#include <iostream>
using namespace std;

class Complex {
private:
    int real, imag;

public:
    Complex(int r = 0, int i = 0)
    {
        real = r;
        imag = i;
    }

    Complex operator+(Complex const& obj)
    {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }

    void print() { cout << real << " + i" << imag << endl; }
};
```

# Поліморфізм під час виконання (runtime)

# Поліморфізм під час виконання

- Поліморфізм часу виконання також називають **пізнім зв'язуванням (Late Binding)** або **динамічним поліморфізмом (Dynamic Polymorphism)**.
- У поліморфізмі часу виконання виклик функції визначається під час **виконання програми**. На відміну від поліморфізму часу компіляції, де компілятор заздалегідь визначає, яку функцію викликати, у динамічному поліморфізмі це рішення приймається безпосередньо **під час роботи програми**.
- Цей тип поліморфізму досягається за допомогою **віртуальних функцій**.



# Перевизначення функцій (Function Overriding)

Перевизначення функцій (**Function Overriding**) відбувається, коли похідний клас має власне визначення для однієї з функцій-членів базового класу. У такому випадку кажуть, що базова функція була перевизначена.

# Приклад перевизначення функції

Динамічний поліморфізм в даному випадку не працює (статичне зв'язування)

```
class BaseClass {
public:
    void display() {
        cout << "Function of Parent Class"<< endl;
    }
};

class DerivedClass : public BaseClass {
public:
    void display() {
        cout << "Function of Child Class" << endl;
    }
};

int main() {
    DerivedClass d_obj;
    d_obj.display();

    BaseClass& b_obj = d_obj;
    b_obj.display();

    BaseClass* b_obj1 = new DerivedClass();
    b_obj1->display();

    delete b_obj1;

    return 0;
}
```

```
Function of Child Class
Function of Parent Class
Function of Parent Class
```

# Віртуальні функції



FACULTY OF APPLIED  
MATHEMATICS AND  
INFORMATICS  
LVIV UNIVERSITY

# Віртуальні функції

- **Віртуальна функція** — це функція-член, яка оголошена в базовому класі за допомогою ключового слова **virtual** і **перевизначена (override)** у похідному класі.
- Віртуальні функції є **динамічними** за своєю природою.
- Вони визначаються за допомогою ключового слова **virtual** в базовому класі і завжди оголошуються в базовому класі та перевизначаються в похідному класі.
- Віртуальна функція викликається під час **часу виконання** (Runtime) через вказівник або референс на базовий клас, спрацює реалізація в залежності від типу об'єкта

# Приклад 1

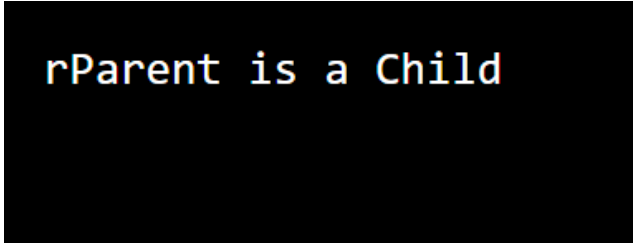
```
#include <iostream>

class Parent
{
public:
    virtual const char* getName() { return "Parent"; }
    // додали ключове слово virtual
};

class Child: public Parent
{
public:
    const char* getName() override { return "Child"; }
};
```

```
int main()
{
    Child child;
    Parent &rParent = child;
    std::cout << "rParent is a "
    << rParent.getName() << '\n';

    return 0;
}
```



rParent is a Child

Оскільки **rParent** є посиланням на батьківську частину об'єкту **child**, то, звичайно, при обробці **rParent.getName()** викликався б **Parent::getName()**. Проте, оскільки **Parent::getName()** є віртуальною функцією, то компілятор автоматично створює vtable і vptr для класів ієрархії, за допомогою яких викликає **Child::getName()**!

# Приклад 2

```
#include <iostream>
```

```
class A
{
public:
    virtual const char* getName() { return "A"; }
};
```

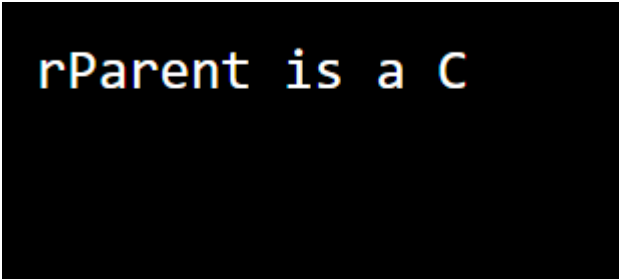
```
class B: public A
{
public:
    const char* getName() override { return "B"; }
};
```

```
class C: public B
{
public:
    const char* getName() override { return "C"; }
};
```

```
class D: public C
{
public:
    const char* getName() override { return "D"; }
};
```

```
int main()
{
    C c;
    A &rParent = c;
    std::cout << "rParent is a " <<
rParent.getName() << '\n';

    return 0;
}
```



rParent is a C

# Приклад

```
class Parent
{
public:
    virtual int getValue() { return 7; }
};

class Child: public Parent
{
public:
    virtual double getValue() { return 9.68; }
};
```

**Типи повернення  
віртуальної функції і  
її перевизначень  
повинні збігатися**

В цьому випадку **Child::getValue()** не рахується відповідним перевизначенням для **Parent::getValue()**, тому що типи повернень різні (метод *Child::getValue()* вважається повністю окремою функцією).

# Приклад 3

- Віртуальні функції дозволяють викликати метод похідного класу через вказівник або посилання на базовий клас.
- Динамічний поліморфізм працює через `vtable` і `vptr`, що автоматично створюються компілятором.
- Без `virtual` виклик працює статично, і похідний метод не буде викликаний.
- При обробці **`animal.speak()`**, компілятор бачить, що метод **`Animal::speak()`** є віртуальною функцією. Коли `animal` посилається на частину **`Animal`** об'єкту **`Cat`**, то компілятор використовує **`vptr`** об'єкту **`Cat`** щоб викликати відповідний метод **`Cat::speak()`**.

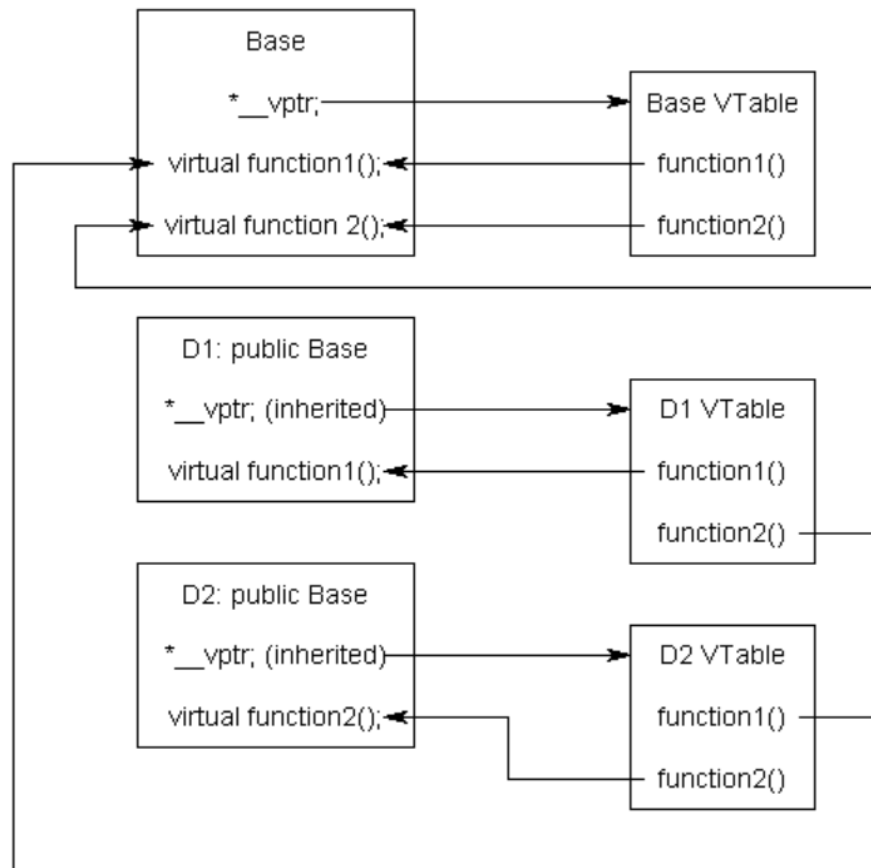


# Деталі поліморфізму

```
class Base
{
public:
    virtual void function1() {};
    virtual void function2() {};
};

class D1: public Base
{
public:
    virtual void function1() {};
};

class D2: public Base
{
public:
    virtual void function2() {};
};
```



# Деталі поліморфізму

- Для кожного класу з віртуальними методами компілятор автоматично надає VTABLE – масив адрес віртуальних методів
- Кожен об'єкт має неявний VPTR, проініціалізований адресою VTABLE класу до виконання 1-ї інструкції конструктора
- Конструктори похідних типів змінюють значення VPTR на адресу VTABLE свого класу
- Для неперевизначеного віртуального методу задається адреса методу з базовго класу
- Виклик віртуального методу у конструкторі чи деструкторі – використовується локальна версія методу (поліморфізм не працює)

# Деталі перевизначення віртуальних функцій

- Не викликай віртуальні функції в конструкторі чи деструкторі – поліморфізм не спрацює! Завжди буде викликатися базова версія методу
- Причина: базова частина об'єкту створюється швидше ніж похідна частина

```
class A {  
public:  
    A() { f(); }  
    virtual void f() {cout << "A::f";}  
};  
class B : public A {  
public:  
    void f() {cout << "B::f"; }  
};  
void main()  
{  
    A * a = new B();  
    delete a;  
}
```

# Недолік віртуальних функцій

## Неефективно:

- ❑ Виклик віртуальної функції займає більше часу, ніж виклик звичайної.
- ❑ Компілятор також повинен виділити додатковий вказівник для кожного об'єкта класу, що має одну або кілька віртуальних функцій.

# Ігнорування віртуалізації

```
class Base{  
public:  
    virtual string getName() { return "Base"; }  
};  
  
class Derived: public Base  
{  
public:  
    virtual string getName() { return "Derived"; }  
};
```

Якщо ми хочемо, щоб викликався метод з базового класу, то вказуємо це через специфікатор доступу:

```
int main()  
{  
    Derived derived;  
    Base &base = derived;  
    // Викликай Base::GetName() замість віртуалізації Derived::GetName()  
    cout << base.Base::getName();  
}
```

# Блокування поліморфізму

- ❑ Виклик віртуального методу не через посилання( вказівник) – статичне зв'язування
- ❑ Виклик з кваліфікатором (назвою) конкретного класу
- ❑ Перевизначення одного з перевантажених віртуальних методів приховує інші віртуальні з тою ж назвою
- ❑ Зрізка при копіюванні: конструктор копіювання базового класу перевизначає вказівник VPTR на таблицю VTABLE базового класу.

# Приклад 4. Приховування віртуальних методів

```
class Base {
public:
    virtual void show() { std::cout << "Base::show()\n"; }
    virtual void show(int x) { std::cout << "Base::show(int) with x = " << x << "\n"; }
};

class Derived : public Base {
public:
    void show() override { std::cout << "Derived::show()\n"; } // Перевизначає тільки show(), але приховує show(int)
};

int main() {
    Derived d;
    d.show();           // Викличе Derived::show()

    //d.show(10); // ✗ ПОМИЛКА! Base::show(int) приховано

    Base* b = &d;
    b->show(10);        // ✓ Викличе Base::show(int), бо доступ через вказівник на Base

    return 0;
}
```

# Приклад 4. Приховування віртуальних методів

## Проблема

- ❑ У **Base** є дві перевантажені версії `show()`:
  - `show()`
  - `show(int)`
- ❑ У **Derived** перевизначається **лише** `show()`, але це **приховує** всі версії `show()` з **Base**, навіть якщо вони не перевизначені.

## Рішення

Щоб зробити приховані методи базового класу доступними в похідному класі, потрібно явно вказати `using`

```
class Derived : public Base {  
public:  
    using Base::show; // Додає всі версії show() з Base в область видимості Derived  
    void show() override { std::cout << "Derived::show()\n"; }  
};
```



# Приклад 5. Зрізання (slicing) при копіюванні

## Проблема

Об'єкт **Derived** **d1** створюється правильно (спочатку **Base**, потім **Derived**).

Оголошення **Base** **b = d1;**

- Викликає конструктор копіювання **Base(const Base&)**.
- Копіюється тільки частина **Base**, а похідна частина (**Derived**) ігнорується.
- Вказівник **VPTR** тепер вказує на **VTABLE** базового класу, тому при виклику **b.show()** викликається **Base::show()**, а не **Derived::show()**.

## Рішення

Щоб забезпечити правильне копіювання об'єктів із урахуванням поліморфізму, потрібно використовувати віртуальну функцію **clone()**

# Приклад 6. Віртуальна функція clone( )

## Рішення

Щоб забезпечити **правильне копіювання** об'єктів із урахуванням поліморфізму, потрібно використовувати **віртуальну функцію `clone()`**

## Висновки

- ❑ **конструктор копіювання базового класу не знає про похідний клас**, тому копіює лише частину **Base**, змінюючи VPTR.
- ❑ **При створенні копій поліморфних об'єктів потрібно використовувати `clone()`**, щоб зберегти правильну таблицю VTABLE.
- ❑ **Уникайте `Base b = d1;` у випадках поліморфізму**, бо це призводить до зрізання (slicing).

# override та final



# override

Ключове слово **override** виконує дві основні функції:

1. **Чіткість коду** – показує, що цей метод є **віртуальним** і перевизначає віртуальний метод базового класу.
2. **Перевірка компілятором** – допомагає уникнути помилок, перевіряючи, що метод дійсно **перевизначає** існуючий метод, а не створює новий.

# Приклад 7

```
class Base
{
    public:
        virtual int foo(float x) = 0;
};

class Derived: public Base
{
    public:
        int foo(float x) override { ... } // OK
};

class Derived2: public Base
{
    public:
        int foo(int x) override { ... } // ERROR
};
```

# final

## Використання специфікатора **final** у C++11

1. Іноді потрібно заборонити похідному класу перевизначати віртуальну функцію базового класу. У C++11 з'явилася вбудована можливість запобігти перевизначенню віртуальної функції за допомогою специфікатора **final**.

# Приклад 8

```
#include <iostream>
using namespace std;

class Base
{
public:
    virtual void myfun() final
    {
        cout << "myfun() in Base";
    }
};

class Derived : public Base
{
    void myfun()
    {
        cout << "myfun() in Derived\n";
    }
};
```

```
int main()
{
    Derived d;
    Base &b = d;
    b.myfun();
    return 0;
}
```

```
prog.cpp:14:10: error: virtual function 'virtual void Derived::myfun()' overriding final function
   14 |         void myfun()
      |         ~~~~~
prog.cpp:7:18: note: overridden function is 'virtual void Base::myfun()'
    7 |         virtual void myfun() final
      |         ~~~~~
```

# final

## Специфікатор **final** у C++11 для заборони успадкування

Специфікатор **final** у C++11 можна використовувати не лише для заборони перевизначення віртуальних функцій, а й для **запобігання успадкуванню класу або структури**.

Якщо клас або структура позначені як **final**, вони стають **неспадковуваними** і не можуть використовуватися як базові.



# Приклад 9

```
#include <iostream>

class Base final
{
};

class Derived : public Base
{
};

int main()
{
    Derived d;
    return 0;
}
```

```
prog.cpp:6:7: error: cannot derive from 'final' base 'Base' in derived type 'Derived'
  6 | class Derived : public Base
    |               ^~~~~~
```

# final

У C++11 **final** не є зарезервованим ключовим словом, а має спеціальне значення лише в певних контекстах:

1. **Заборона перевизначення віртуальної функції**
2. **Заборона успадкування класу або структури**

```
#include <iostream>
using namespace std;

int main()
{
    int final = 20;
    cout << final;
    return 0;
}
```



20

# Віртуальні деструктори



FACULTY OF APPLIED  
MATHEMATICS AND  
INFORMATICS  
LVIV UNIVERSITY

# Віртуальні деструктори

## Видалення об'єкта похідного класу через вказівник на базовий клас

Якщо в базовому класі **деструктор не є віртуальним**, видалення об'єкта похідного класу через вказівник на базовий клас може призвести до **memory leak** у випадку, якщо похідний клас містить динамічно виділені поля.

# Приклад 10. Memory leak.

```
#include <iostream>

class Base {
public:
    ~Base() { // Деструктор НЕ віртуальний
        std::cout << "Base Destructor\n";
    }
};

class Derived : public Base {
private:
    int* m_array;
public:
    Derived(int length){m_array = new int[length];}
    ~Derived() { // Деструктор похідного класу
        std::cout << "Derived Destructor\n";
        delete[] m_array;
    }
};

int main() {
    Base* obj = new Derived(7); // Створюємо об'єкт похідного класу
    delete obj; // Видаляємо через вказівник на базовий клас
    return 0;
}
```

Base Destructor

# Віртуальні деструктори

## **Рішення: Використання віртуального деструктора**

Щоб правильно викликати деструктор похідного класу, потрібно оголосити **віртуальний деструктор** у базовому класі.

# Приклад

```
#include <iostream>
```

```
class Base {  
public:  
    virtual ~Base() { // Віртуальний деструктор  
        std::cout << "Base Destructor\n";  
    }  
};
```

```
class Derived : public Base {  
private:  
    int* m_array;  
Public:  
    Derived(int length){m_array = new int[length];}  
    ~Derived() {  
        delete[] m_array;  
        std::cout << "Derived Destructor\n";  
    }  
};
```

```
int main() {  
    Base* obj = new Derived(7);  
    delete obj; // Тепер викликається  
                //правильний деструктор  
    return 0;  
}
```

Derived Destructor  
Base Destructor

## Рекомендація щодо віртуального деструктора

Якщо клас містить хоча б одну віртуальну функцію, завжди слід додавати віртуальний деструктор, навіть якщо він не виконує жодних дій.

# Чисті віртуальні функції



FACULTY OF APPLIED  
MATHEMATICS AND  
INFORMATICS  
LVIV UNIVERSITY



# Чисті віртуальні функції

До цього моменту ми записували визначення всіх наших віртуальних функцій. Однак мова C++ дозволяє створювати особливий вид віртуальних функцій, так званих **чистих віртуальних функцій** (або **«абстрактних функцій»**), які взагалі не мають визначення! Перевизначають їх дочірні класи.

При створенні чистої віртуальної функції, замість визначення (написання тіла) віртуальної функції, ми просто присвоюємо їй значення 0.

будь-який клас з однією і більше чистими віртуальними функціями стає **абстрактним** класом, об'єкти якого створювати заборонено

Кожен похідний клас повинен визначити тіло чистих методів, інакше він теж буде абстрактним

# Приклад

```
class Parent
{
public:
    const char* sayHi() { return "Hi"; } // звичайна невіртуальна функція

    virtual const char* getName() { return "Parent"; } // звичайна віртуальна
функція

    virtual int getValue() = 0; // чиста віртуальна функція

    int doSomething() = 0; // помилка компіляції: заборонено присвоювати
невіртуальним функціям значення 0
};
```

Таким чином, ми повідомляємо компілятору: **«Реалізацією цієї функції займуться дочірні класи».**

# Приклад

```
#include <iostream>
#include <string>


class Animal // цей Animal є абстрактним батьківським класом
{
protected:
    std::string m_name;

public:
    Animal(std::string name)
        : m_name(name)
    {
    }

    std::string getName() { return m_name; }
    virtual const char* speak() =0; // зверніть увагу, speak() є чистою
    віртуальною функцією
};
```

ПОМИЛКА

```
int main() {
    Animal* animal = new
    Animal("Woof");
    std::cout << animal->speak();
    return 0;
}
```



**speak()** є чистою віртуальною функцією. Це означає, що Animal -  
**абстрактний** батьківський клас. Його об'єкт створювати не можна.

# Чисті віртуальні функції з визначенням

- При визначенні чистої віртуальної функції, її тіло (визначення) повинно бути записане окремо (не вбудовано).
- Це корисно, коли ви хочете, щоб дочірні класи мали можливість перевизначати віртуальну функцію або залишити її реалізацію за замовчуванням (яку надає батьківський клас). У випадку, якщо дочірній клас задоволений реалізацією за замовчуванням, він може просто викликати її напряму.
- обов'язкова реалізація для чистого деструктора базового класу, якщо він єдиний в класі віртуальний метод (заборона утворення об'єкта); у похідному реалізує компілятор

# Чисті віртуальні функції з визначенням

```
#include <iostream>
#include <string>

class Animal // це абстрактний батьківський клас
{
protected:
    std::string m_name;

public:
    Animal(std::string name)
        : m_name(name)
    {
    }

    std::string getName() { return m_name; }
    virtual const char* speak() = 0; // присвоювання значення "= 0" повідомляє про те, що
    ця функція є чистою віртуальною функцією
};

const char* Animal::speak() // незважаючи на те, що ось тут знаходиться її визначення
{
    return "buzz";
```

Виявляється, ми можемо визначити чисті віртуальні функції:

# Приклад 11. Абстрактний клас

```
// Abstract base class
class Shape {
public:
    virtual double area() const = 0;    // Pure virtual function for area
    virtual double perimeter() const = 0; // Pure virtual function for perimeter
    virtual void display() const = 0;    // Virtual function for displaying shape info

    // Pure virtual destructor (must be defined outside the class)
    virtual ~Shape() = 0;
};

// Definition of pure virtual destructor
Shape::~Shape() {
    std::cout << "Shape destructor called\n";
}
```

- ❑ Абстрактний клас **Shape** забезпечує поліморфізм, дозволяючи працювати з **Circle** і **Triangle** через один інтерфейс
- ❑ Дозволяють розширювати функціонал без зміни базового коду (Принцип відкритості/закритості **SOLID**).
- ❑ Дотримується принцип підстановки Лісков (**LSP from SOLID**) – об'єкти підкласів можна використовувати замість базового класу.

```
int main() {
    const size_t size = 4;
    Shape** arr = new Shape*[size];
    arr[0] = new Triangle(3.0, 4.0, 5.0);
    arr[1] = new Circle(7.0);
    arr[2] = new Triangle(7.0, 7.0, 7.0);
    arr[3] = new Circle(5.0);

    for (size_t i = 0; i < size; ++i)
    {
        arr[i]->display();
        delete arr[i];
    }
    delete[] arr;



    return 0;
}
```

# Абстрактний клас

- ❑ Конструктор абстрактного класу може бути використано (явно або неявно) лише в конструкторі похідного типу
- ❑ Абстрактний клас не може використовуватись як тип параметру функції, як тип функції або як тип явного приведення.
- ❑ Можна визначати вказівники і посилання на абстрактний клас

```
Shape x;    // error: object of abstract class
Shape* p;   // OK
Shape f();  // error
void g(Shape); // error
Shape& h(Shape&); // OK
```

# Переваги та недоліки

Переваги 	Недоліки 
Поліморфізм – базовий клас дозволяє працювати з об'єктами похідних класів через один інтерфейс	Неможливо створювати об'єкти напряму
Інкапсуляція спільного коду	Множинне наслідування може спричиняти конфлікти
Дотримання принципів SOLID (OCP, LSP)	Віртуальні виклики споживають більше пам'яті (vptr, vtable)
Можливість часткової реалізації	Ускладнення коду при надмірному використанні

## Коли використовувати абстрактний клас:

- ☐ Якщо потрібно частково реалізувати функціональність (на відміну від чистого інтерфейсу).
- ☐ Якщо буде спільна логіка для всіх підкласів.
- ☐ Якщо об'єкти будуть використовуватися через базовий клас (`Base* ptr = new Derived;`).
- ☐ Якщо потрібно підтримувати розширюваність без зміни існуючого коду.



# Інтерфейси



FACULTY OF APPLIED  
MATHEMATICS AND  
INFORMATICS  
LVIV UNIVERSITY

# Інтерфейс

**Інтерфейс** — це клас, який не має змінних-членів і **всі методи** якого є чистими віртуальними функціями! Інтерфейси ще називають «**класами-інтерфейсами**» або «**інтерфейсними класами**».

# Інтерфейс

Інтерфейсні класи прийнято називати з **I** на початку, наприклад:

```
class IErrorLog
{
public:
    virtual bool openLog(const char *filename) = 0;
    virtual bool closeLog() = 0;

    virtual bool writeError(const char *errorMessage) = 0;

    virtual ~IErrorLog() {}; // створюємо віртуальний деструктор, щоб викликався
    відповідний деструктор дочірнього класу у випадку, якщо видалимо вказівник на IErrorLog
};
```

# Інтерфейс

- Будь-який клас, який наслідує **IErrorLog**, повинен надати свою реалізацію всіх 3 методів класу **IErrorLog**.
- Ви можете створити дочірній клас з ім'ям **FileErrorLog**, де **openLog()** відкриває файл на диску, **closeLog()** — закриває файл, а **writeError()** — записує повідомлення в файл. Ви можете створити ще один дочірній клас з ім'ям **ScreenErrorLog**, де **openLog()** і **closeLog()** нічого не роблять, а **writeError()** виводить повідомлення у спливаючому вікні.

# Переваги інтерфейсів

- Інтерфейси надзвичайно популярні, тому що вони прості у використанні, зручні в підтримці, і їх функціонал легко розширювати.
- Деякі мови, такі як Java і C#, навіть додали в свій синтаксис ключове слово `interface`, яке дозволяє програмістам напяму визначати інтерфейсний клас, не вказуючи явно, що всі методи є абстрактними.

# Тест!



FACULTY OF APPLIED  
MATHEMATICS AND  
INFORMATICS  
LVIV UNIVERSITY

```
#include <iostream>

class A
{
public:
    virtual const char* getName() { return "A"; }
};

class B: public A
{
public:
    virtual const char* getName() { return "B"; }
};

class C: public B
{
public:
    // Примітка: Тут немає методу getName()
};

class D: public C
{
public:
    virtual const char* getName() { return "D"; }
};
```

```
int main()
{
    C c;
    A &rParent = c;
    std::cout <<
rParent.getName() << '\n';

    return 0;
}
```

Який результат виконання програми?

```
#include <iostream>
```

```
class A
{
public:
    virtual const char* getName() { return "A"; }
};
```

```
class B: public A
{
public:
    virtual const char* getName() { return "B"; }
};
```

```
class C: public B
{
public:
    virtual const char* getName() { return "C"; }
};
```

```
class D: public C
{
public:
    virtual const char* getName() { return "D"; }
};
```

```
int main()
{
    C c;
    B &rParent = c; // примітка: rParent
на цей раз класу B
    std::cout << rParent.getName() <<
'\n';

    return 0;
}
```

Який результат виконання програми?



```
#include <iostream>
```

```
class A
```

```
{  
public:  
    const char* getName() { return "A"; } // примітка:  
virtual  
};
```

```
class B: public A
```

```
{  
public:  
    virtual const char* getName() { return "B"; }  
};
```

```
class C: public B
```

```
{  
public:  
    virtual const char* getName() { return "C"; }  
};
```

```
class D: public C
```

```
{  
public:  
    virtual const char* getName() { return "D"; }  
};
```

```
int main()
```

```
{  
    C c;  
    A &rParent = c;  
    std::cout << rParent.getName() << '\n';  
  
    return 0;  
}
```

Який результат виконання програми?

```
#include <iostream>
```

```
class A
```

```
{
```

```
public:
```

```
    virtual const char* getName() { return "A"; }
```

```
};
```

```
class B: public A
```

```
{
```

```
public:
```

```
    const char* getName() { return "B"; } // примітка: Немає ключового слова
```

```
virtual
```

```
};
```

```
class C: public B
```

```
{
```

```
public:
```

```
    const char* getName() { return "C"; }
```

```
};
```

```
class D: public C
```

```
{
```

```
public:
```

```
    const char* getName() { return "D"; }
```

```
int main()
```

```
{
```

```
    C c;
```

```
    B &rParent = c; // примітка: rParent на цей  
раз класу B
```

```
    std::cout << rParent.getName() << '\n';
```

```
    return 0;
```

```
}
```

Який результат виконання програми?

# Дякую!



FACULTY OF APPLIED  
MATHEMATICS AND  
INFORMATICS  
LVIV UNIVERSITY