

## Поліморфізм

Слово поліморфізм означає "мати багато форм".

Зазвичай поліморфізм виникає тоді, коли існує ієрархія класів, і вони пов'язані наслідуванням. Поліморфізм

С ++ означає, що виклик функції-члена викликає різну реалізацію в залежності від типу об'єкта, який викликає функцію.

Поліморфізм можна продемонструвати більш наочно на прикладі: припустимо, ви хочете зробити просту гру, яка включає різних ворогів: монстрів, ніндзя тощо. У всіх ворогів є одна спільна функція: функція атака. Однак вони атакують по-різному. У цій ситуації поліморфізм дозволяє викликати одну і ту ж функцію нападу на різні об'єкти, але призводить до різної поведінки.

Перший крок - створення класу Enemy .

```
class Enemy {  
protected:  
    int attackPower;  
public:  
    void setAttackPower(int a){  
        attackPower = a;  
    }  
};
```

У нашому класі Enemy є відкритий метод під назвою **setAttackPower** , який встановлює захищену змінну учасника **attackPower** .

Другий наш крок - створити класи для двох різних типів ворогів, ніндзя та монстрів. Обидва ці нові класи успадковують від класу Enemy, тому кожен має силу атаки. При цьому кожен має специфічну функцію атаки.

```
class Ninja: public Enemy {  
public:  
    void attack() {  
        cout << "Ninja! - " << attackPower << endl;  
    }  
};  
  
class Monster: public Enemy {  
public:  
    void attack() {  
        cout << "Monster! - " << attackPower << endl;  
    }  
};
```

Як бачимо, їхні індивідуальні функції атаки відрізняються. Тепер ми можемо створювати наші об'єкти Ninja та Monster в main.

```
int main() {  
    Ninja n;  
    Monster m;  
}
```

Ninja та Monster успадковуються від Enemy, тому всі об'єкти Ninja та Monster є об'єктами Enemy. Це дозволяє нам зробити наступне:

```
Enemy *e1 = &n;  
Enemy *e2 = &m;
```

Зараз ми створили два покажчики типу Enemy, вказуючи їх на об'єкти Ninja та Monster.

Тепер ми можемо викликати відповідні функції:

```
int main() {  
    Ninja n;  
    Monster m;  
    Enemy *e1 = &n;  
    Enemy *e2 = &m;  
  
    e1->setAttackPower(20);  
    e2->setAttackPower(80);  
  
    n.attack();  
    m.attack();  
}  
  
/* Output:  
Ninja! - 20  
Monster! - 80  
*/
```

## Virtual Functions

Попередній приклад демонструє використання покажчиків базового класу до похідних класів. Чому це корисно? Продовжуючи наш приклад гри, ми хочемо, щоб кожен ворог мав функцію **attack()**.

Щоб мати можливість викликати відповідну функцію **attack()** для кожного з похідних класів за допомогою покажчиків Enemy, нам потрібно оголосити функцію базового класу як **virtual**.

Визначення **віртуальної функції** в базовому класі з відповідною версією у похідному класі дозволяє поліморфізму використовувати вказівники Enemy для виклику функцій похідних класів.

Кожен похідний клас замінить функцію **attack()** і матиме окрему реалізацію:

```

class Enemy {
public:
    virtual void attack() {
    }
};

class Ninja: public Enemy {
public:
    void attack() {
        cout << "Ninja!"<<endl;
    }
};

class Monster: public Enemy {
public:
    void attack() {
        cout << "Monster!"<<endl;
    }
};

```

Віртуальна функція є базовою функцією класу, яка оголошується з допомогою ключового слова **virtual**.

Зараз ми можемо використовувати покажчики **Enemy** для виклику функції `attack ()` .

```

int main() {
    Ninja n;
    Monster m;
    Enemy *e1 = &n;
    Enemy *e2 = &m;

    e1->attack();
    e2->attack();
}

/* Output:
Ninja!
Monster!
*/

```

Оскільки функція `attack()` оголошена віртуальною, вона працює як шаблон, повідомляючи про те, що похідний клас може мати власну функцію `attack ()` .

Наш ігровий приклад слугує для демонстрації концепції поліморфізму; ми використовуємо вказівники **Enemy** для виклику однієї і тієї ж функції `attack ()` та отримання різних результатів.

```

e1->attack();
e2->attack();

```

Якщо функція в базовому класі є **virtual**, реалізація функції у похідному

класі викликається відповідно до фактичного типу об'єкта, на який посилається, незалежно від заявленого типу вказівника.

Клас, який оголошує або успадковує віртуальну функцію, називається **поліморфним** класом.

Віртуальні функції також можуть мати свою реалізацію в базовому класі:

```
class Enemy {
public:
    virtual void attack() {
        cout << "Enemy!"<<endl;
    }
};

class Ninja: public Enemy {
public:
    void attack() {
        cout << "Ninja!"<<endl;
    }
};

class Monster: public Enemy {
public:
    void attack() {
        cout << "Monster!"<<endl;
    }
};
```

Тепер, коли ви створюєте вказівник Enemy, та викликаєте функцію attack () компілятор викличе функцію, що відповідає типу об'єкта, на яку вказує вказівник:

```
int main() {
    Ninja n;
    Monster m;
    Enemy e;

    Enemy *e1 = &n;
    Enemy *e2 = &m;
    Enemy *e3 = &e;

    e1->attack();
    // Outputs "Ninja!"

    e2->attack();
    // Outputs "Monster!"

    e3->attack();
    // Outputs "Enemy!"
}
```

Ось як використовується поліморфізм . У вас є різні класи з функцією з

тим же ім'ям і навіть однаковими параметрами, але з різними реалізаціями.

У деяких ситуаціях ви хочете включити віртуальну функцію в базовий клас, щоб її можна було перевизначити у похідному класі, щоб вона підходила об'єктам цього класу, але щоб не було змістовного визначення, яке ви могли б дати для функції в базовому класі.

Віртуальні функції без визначення відомі як **чисто віртуальні функції**. Вони вказують, що похідні класи визначають цю функцію як власну.

Синтаксис полягає у заміні їх визначення на `= 0`:

```
class Enemy {  
public:  
    virtual void attack () = 0 ;  
};
```

`= 0` повідомляє компілятору, що у функції немає тіла.

Чисто віртуальна функція визначає, що похідні класи будуть визначати цю функцію, як власну.

Кожен похідний клас, що успадковується від класу з чисто віртуальною функцією, **повинен** перезаписати цю функцію.

Якщо чиста віртуальна функція не перезаписана у похідному класі, код не зможе скомпільоватись і призводить до помилки при спробі інстанціювати об'єкт похідного класу.

Чиста віртуальна функція в класі **Enemy** повинна бути перезаписана у похідних класах.

```
class Enemy {  
public:  
    virtual void attack() = 0;  
};  
  
class Ninja: public Enemy {  
public:  
    void attack() {  
        cout << "Ninja!"<<endl;  
    }  
};  
  
class Monster: public Enemy {  
public:  
    void attack() {  
        cout << "Monster!"<<endl;  
    }  
};
```

## Абстрактні класи

Ви **не можете** створити об'єкти базового класу з чисто віртуальною функцією. Запуск наступного коду поверне помилку:

```
Enemy e; // Error
```

Ці класи називають **абстрактними**. Це класи, які можуть бути використані лише як базові класи, і тому їм дозволяється мати чисті віртуальні функції.

Ви можете подумати, що абстрактний базовий клас марний, але це не так. З його допомогою можна створити покажчики та скористатись усіма своїми поліморфними здібностями.

Наприклад, ви можете написати:

```
Ninja n;  
Monster m;  
Enemy *e1 = &n;  
Enemy *e2 = &m;  
  
e1->attack();  
e2->attack();
```

У цьому прикладі об'єкти різних, але споріднених типів посилаються на використання унікального типу вказівника (Enemy \*), і належна функція члена викликається кожен раз, лише тому, що вони є віртуальними.