Лекція 5. Моделювання відношення «has-a»: включення, закрите наслідування. Множинне наслідування. Бібліотека RTTI

1. *Включення об'єкта до складу іншого об'єкта.*
2. *Закрите наслідування, особливості синтаксису.*
3. *Захищене наслідування – різновид закритого наслідування.*
4. *Множинне наслідування. Віртуальні батьківські класи.*
5. *Бібліотека RTTI.*

У попередніх лекціях ми побудували ієрархію класів *Shape* – [*Rectangle*, *Square*, *Circle*]. Абстрактний базовий клас моделює спільні властивості геометричних фігур на площині, задає спільний протокол взаємодії; підкласи конкретизують поведінку відповідних фігур.

Підкласи утворено за допомогою відкритого наслідування, наприклад:– *class Square : public Rectangle ...* . Таке наслідування моделює відношення «is-a» (відношення «є») між класами: *Квадрат є Прямокутник*, *Круг є Фігура* тощо.

Засобами ООП можна моделювати не тільки відношення «is-a». Ще одним поширеним відношенням між класами є «has-a» (відношення «має» або «містить»). Ці відношення повсякчас можна спостерігати в реальному світі: Sens *є* Автомобіль, Sens *має* Двигун; Конус *є* Об'ємна Фігура, Конус *містить* Круг (основу) – такі класи ми вже моделювали на попередніх заняттях; Студент *є* Особа, Студент *має* Ім'я, Студент *має* Оцінки.

Спорідненість надклас-підклас відображає спільність поведінки та формує ієрархію класів за рівнем абстрактності (конкретності) цієї поведінки. Складні об'єкти (Автомобіль, Конус, Студент) мають ще ієрархію за структурою: конус містить висоту і основу, основа містить радіус тощо. Відомо, що об'єкт підкласу *структурно* містить в собі об'єкт базового класу, тому може виникнути спокуса виводити складніший об'єкт нащадком простішого. Проте головною рисою підкласу є наслідування *поведінки* батьківського класу. Очевидно, було б помилкою робити клас *Студент* нащадком класу *Рядок*, бо що б тоді мала означати операція конкатенації в породженому класі?

Отож, ієрархію за структурою об'єктів моделюють засобами мови C++ за допомогою *включення* одного об'єкта до складу іншого (включення ще називають *компонуванням* чи *композицією*). Композит отримує реалізацію включених об'єктів.

У мові C++ відкрите наслідування – не єдиний спосіб оголошення підкласів. Використовують ще *закрите* і *захищене* наслідування.

> За умов *закритого наслідування* всі *protected* і *public* елементи базового класу стають *private* елементами породженого класу. Наприклад, розглянемо такий фрагмент коду

```
class Base {
private: int a;
protected:  int getA() { return a; }
            void setA(int k) { a = k; }
public: Base(int k = 0) : a(k) {}
            void show() { cout << a; }
            int increment() { return a++; }
};
class First : private Base {
private: double b;
public: First(int k = 1, double x = 0.) : Base(k), b(x) {}
            double product(){ return b*getA(); }
            void show() { Base::show(); cout << '*' << b; }
};
```

Клас *First* утворено за допомогою закритого наслідування від *Base*. Це означає, що кожен екземпляр класу *First* міститиме неіменований підоб'єкт базового класу (поле *a*), в тілі методів класу *First* можна викликати *Base::getA()*, *Base::setA()*, *Base::show()* та *Base::increment()*

для маніпулювання недоступним полем *a* та для його виведення, проте ці методи не входять до інтерфейсу класу *First*. Користувачам доступні лише *First::product()* та *First::show()* (тут *First::show()* імітує наслідування з батьківського класу). Таким чином, клас *First* отримав у використання реалізацію класу *Base*, але не унаслідував його інтерфейс – це характерні ознаки відношення «has-a» між класами. *Тобто, закрите наслідування моделює відношення «має»!*

Цікаво було б використати цю можливість для альтернативного оголошення класу якоїсь об'ємної фігури, наприклад, чотирикутної піраміди. Піраміда містить прямокутник – основу. Змоделюємо це відношення за допомогою закритого наслідування.

```
class RectangularPyramid : private Rectangle
{
private:
	double h;
public:
	// основу ініціалізують конструктори надкласу, викликані, як при наслідуванні
	RectangularPyramid() : Rectangle(), h(1.) {}
	RectangularPyramid(double a, double b, double c) : Rectangle(a, b), h(c) {}
	virtual void printOn(ostream& os) const
	{
		os << "RectangularPyramid(" << h << " x ";
		Rectangle::printOn(os);
		os << ')';
	}
	virtual void storeOn(ofstream& fout) const
	{
		fout << "RectangularPyramid " << h << ' ';
		Rectangle::storeOn(fout);
	}
	// Площу основи піраміди може обчислити успадкований метод
	virtual double baseSquare() const { return this->square(); }
	// Інші обчислення дещо складніші. Вони також використовують успадкований функціонал
	virtual double volume() const { return square() * h / 3.; }
	// Площа поверхні залежить від площ частин.
	virtual double surfaceSquare() const
	{
		return this->baseSquare() + this->sideSquare();
	}
	// Піраміда має безпосередній доступ до полів базового класу - прямокутника
	virtual double sideSquare() const
	{
		return a*sqrt(h*h + b*b*0.25) + b*sqrt(h*h + a*a*0.25);
	}
};
```

Ви помітили? Жодних труднощів з обчисленням площі бічної поверхні! Раніше ми вимушено оголошували підклас *RectAB*, щоб дістатися до сторін прямокутника, а тепер піраміда сама має права підкласу і вільний доступ до захищених полів базового класу *Rectangle*.

Щоб побачити, що ще змінилося, порівняймо реалізацію конструкторів і методів у *RectangularPyramid* і оголошеному раніше композиті *Parallelepiped*.

```
class Parallelepiped
{
private:
	Rectangle base;  // тут міститимуться сторони основи паралелепіпеда
	double h;        // тут - сторона, перпендикулярна до основи
public:
	// синтаксис ініціалізатора основи в конструкторах паралелепіпеда
	// подібний до оголошення об'єкта-прямокутника, при цьому працюють
	// конструктори класу Rectangle
	Parallelepiped() :base(), h(1.) {}
	Parallelepiped(double a, double b, double c) :base(a, b), h(c) {}
```

```
	virtual void printOn(ostream& os) const
	{
		os << "Parallelepiped(" << h << " x " << base << ')';
	}
	virtual void storeOn(ofstream& fout) const
	{
		fout << "Parallelepiped " << h << ' ';
		this->base.storeOn(fout);
	}
	// Про площу основи паралелепіпед питає свою основу.
	virtual double baseSquare() const { return base.square(); }
	// Інші обчислення дещо складніші. Вони також використовують функціонал основи
	virtual double volume() const { return base.square()*h; }
	virtual double sideSquare() const { return base.perim()*h; }
	// Площа поверхні залежить від площ частин.
	virtual double surfaceSquare() const
	{
		return this->baseSquare()*2. + this->sideSquare();
	}
};
```

Паралелепіпед містить іменований вкладений об'єкт *base*, а піраміда – безіменний успадкований екземпляр базового класу. Конструктори паралелепіпеда використовують ініціалізатори поля *base*, а піраміди – виклики конструкторів базового класу. Методи паралелепіпеда надсилають повідомлення вкладеному об'єктові *base* з вимогою щось обчислити чи надрукувати себе, а методи піраміди викликають потрібні методи базового класу *Rectangle*.

Тепер варто, напевне, об'єднати оголошені класи в одну ієрархію. Допоможе, як і раніше, абстрактний базовий клас

```
class Shape3D
{
public:
	virtual ~Shape3D() {} // не можна забувати!!!
	virtual double baseSquare() const = 0;
	virtual double sideSquare() const = 0;
	virtual double surfaceSquare() const = 0;
	virtual double volume() const = 0;
	// виведення в потік зовнішнього вигляду об'єкта
	virtual void printOn(ostream&) const = 0;
	// зберігання об'єкта до файла у вигляді, придатному для відтворення
	virtual void storeOn(ofstream&) const = 0;
};

ostream& operator<<(ostream& os, const Shape3D& s);
bool operator>(const Shape3D& a, const Shape3D& b);
```

Клас *Shape3D* не містить ніякої реалізації – лише абстрактні методи. Такий клас у C++ відіграє роль чистого інтерфейсу. (У C# для цього використовують спеціальні сутності *interface*.) Оголошення піраміди і паралелепіпеда зазнає незначних змін:

```
class Parallelepiped : public Shape3D
{  . . .  };
class RectangularPyramid : private Rectangle, public Shape3D
{  . . .  };
```

Як бачите, піраміда тепер наслідує від *двох* базових класів. У C++ дозволене множинне наслідування, тому таке оголошення синтаксично правильне і майже не викликає проблем у застосуванні. Поекспериментуйте з екземпляром *RectangularPyramid*, і переконайтеся, що компілятор відмовляється виводити його в потік перевантаженим оператором. Спробуйте пояснити, чому.

Продовжимо наші вправи з різними способами реалізації відношення “has-a”. На початку цієї лекції ми сказали «Студент *має* Ім’я, Студент *має* Оцінки». Давайте оголосимо клас *Student* за допомогою компонування і за допомогою закритого наслідування.

За допомогою компонування оголосити клас Студент можна так:

```
class Student {
private:
	NewStr name; //	прізвище
	ArrayDb scores;	//	оцінки
public:	...
};
```

Тепер до складу кожного екземпляра класу *Student* входитимуть як поля даних екземпляр класу *NewStr* і екземпляр класу *ArrayDb*. Об’єкт *name* є частиною об’єкта *aStudent*, але клас *NewStr* не є предком класу *Student*, тому *Student* наслідує реалізацію і не наслідує інтерфейс (поведінку): всі доступні методи класу *NewStr* використовують всередині *Student* для маніпулювання рядком, але ці методи невидимі зовні, тому для зовнішнього світу *Student* має інший інтерфейс.

Зробимо невеликий відступ і пояснимо, що за класи використано для оголошення *Student*. Клас *NewStr* моделює рядок літер (спрощена версія, порівняно зі стандартним класом *string*).

```
#include <iostream>
using std::ostream; using std::istream;

class NewStr
{
private:
	char * str;
	int len;
	static const int CINLIM = 80;
public:
	NewStr(const char* s);
	NewStr();
	NewStr(const NewStr&);
	~NewStr();
	int length() const { return len; }
	NewStr& operator=(const NewStr&);
	NewStr& operator=(const char*);
	char& operator[](int i);
	const char& operator[](int i) const;
	friend bool operator<(const NewStr&, const NewStr&);
	friend bool operator>(const NewStr&, const NewStr&);
	friend bool operator==(const NewStr&, const NewStr&);
	friend ostream& operator<<(ostream&, const NewStr&);
	friend istream& operator>>(istream&, NewStr&);
};
```

Клас *ArrayDb* є втіленням масиву дійсних чисел, що надає більше можливостей порівняно з вбудованим масивом: дає змогу присвоювати один масив іншому та виконує перевірку індекса при звертаннях до елементів масиву. Цей клас, як і *NewStr*, оперує з динамічним масивом (тільки не літер, а чисел), тому багато в чому вони схожі:

```
#include <iostream>
using std::ostream;

class ArrayDb
{
private:
	unsigned int size;
	double* arr;
public:
```

```
        ArrayDb(): arr(0), size(0) {}
        explicit ArrayDb(unsigned int n, double val = 0.0);
        ArrayDb(const double* pn, unsigned int n);
        ArrayDb(const ArrayDb& a);
        virtual ~ArrayDb();
        unsigned int arSize() const { return size; }
        double average() const;
        double& operator[](int i);
        const double& operator[](int i) const;
        ArrayDb& operator=(const ArrayDb& a);
        friend ostream& operator<<(ostream& os, const ArrayDb& a);
};
```

Екземпляри класу потребують «глибокого» копіювання, тому визначено конструктор копіювання, перевизначено конструктор присвоєння. Перевірку коректності індекса елемента виконано в тілі перевизначених операторів «квадратні дужки». У всіх доцільних випадках використано модифікатор *const*. Зверніть увагу на перший конструктор створення. Його сигнатура вказує на те, що його можна було б використати для автоматичного перетворення значення типу *unsigned* до типу *ArrayDb*. У нашому проекті така можливість небажана, тому ми відімкнули її за допомогою ключового слова *explicit*.

Загалом обидва наведені класи схожі на контейнер з попередньої лекції.

Тепер дамо повне оголошення класу *Student*:

```
#include <iostream>
#include "arraydb.h"
#include "str.h"
using std::ostream;
using std::istream;
class Student
{
private:
        NewStr name;
        ArrayDb scores;
public:
        Student(): name("Nobody"), scores() {}
        Student(const NewStr& s): name(s), scores() {}
        Student(int n): name("Anybody"), scores(n) {}
        Student(const NewStr& s, int n): name(s), scores(n) {}
        Student(const NewStr& s, const ArrayDb& a): name(s), scores(a) {}
        Student(const char* str, const double* pd, int n): name(str), scores(pd,n) {}
        ~Student() {}
        double& operator[](int i);
        const double& operator[](int i) const;
        double average() const;
        friend ostream& operator<<(ostream& os, const Student& stu);
        friend istream& operator>>(istream& is, Student& stu);
};
```

Тут визначено досить багато конструкторів для створення. Всі вони використовують списки ініціалізації. Наприклад, *name("Nobody"), scores()* – це означає, що для створення об'єкта *name* треба використати рядок *"Nobody"*, створює об'єкт відповідний конструктор класу *NewStr*. Відповідно, об'єкт *scores* створює конструктор за замовчуванням класу *ArrayDb*. Тобто, за допомогою списків ініціалізації викликають конструктори вкладених об'єктів.

У конструкторах агрегованих об'єктів велику частину роботи (у нашому прикладі – всю роботу) виконують конструктори включених об'єктів.

Поля екземпляра класу *Student* містять динамічні масиви: масив літер і масив чисел. Чи потрібно в такому випадку визначити в класі власний конструктор копіювання? Якщо цього не зробити, то конструктор копіювання автоматично згенерує компілятор. Такий «автоматичний» конструктор просто копіюватиме значення полів. А значить, для включених об'єктів *name* і *scores* працюватимуть конструктори копіювання їхніх класів – це саме те, що треба. Справді, сам клас *Student* динамічну пам'ять не розподіляє (це роблять екземпляри

включених класів), тому й не потребує власного конструктора копіювання – цілком вистачить таких конструкторів класів *NewStr* і *ArrayDb*. Такі ж міркування стосуються і деструкторів: усю роботу щодо звільнення динамічної пам'яті виконують деструктори включених класів, причому автоматично, без спеціального втручання програміста.

Ще одне зауваження: клас *Student* «імітує» наслідування інтерфейсу включених класів. Справді, середнє арифметичне оцінок студента є середнім арифметичним значень масиву, тому в класі оголошено метод *Student::average()*, що повертає результат виклику *scores.average()* – доступ до реалізації включеного класу через ім'я об'єкта. Метод *operator[]()* визначено в кожному з класів *NewStr* та *ArrayDb*, і виникла би проблема неоднозначності, якби клас *Student* успадковував їхні інтерфейси. Оскільки клас *Student* отримує тільки реалізацію, неоднозначності не виникає: кожен з операторів доступний через імена об'єктів: *scores[]* та *name[]*. Мабуть, мати доступ до окремих оцінок важливіше, ніж до окремих літер прізвища, тому *Student::operator[]()*, повертає *scores[]*.

Невелика програма демонструє використання спроектованого класу *Student*:

```
#include <iostream>
using std::cout;  using std::cin;
#include "studIncl.h"

void set(Student& s, int n);
const int pupils = 3;
const int quizzes = 5;

int main() {
	Student ada[pupils]={quizzes,quizzes,quizzes};
	int i;
	for (i=0; i<pupils; i++) set(ada[i],quizzes);
	for (i=0; i<pupils; i++)
		cout<<'\n'<<ada[i]<<" average = "<<ada[i].average()<<'\n';
	cin.get(); return 0;
}
void set(Student& s, int n) {
	cout<<"Please enter the student's name: ";
	cin >> s;
	cout<<"Please enter "<<n<<" quiz scores: ";
	for (int i=0; i<n; i++) cin>>s[i];
	while (cin.get() !='\n') continue;
}
```

Створення об'єктів виконає конструктор *Student(int n)*, введення – процедура *set(Student& s, int n)*, обчислення – метод *average()*, а виведення – оператор виведення.

Використаємо нову можливість для ще одного оголошення класу *Student*. Якщо виконати закрите наслідування класу *Student* з класу *NewStr*, то до складу об'єкта ввійде неіменований підоб'єкт-рядок (для зберігання імені). Потрібен ще масив для оцінок. Мова C++ дозволяє наслідувати з декількох класів (це згадана раніше нова можливість – множинне наслідування!), тому виконаємо закрите наслідування також і з класу *ArrayDb*:

```
#include <iostream>
#include "arraydb.h"
#include "str.h"
using std::ostream;
using std::istream;

class Student: private NewStr, private ArrayDb
{
public:
	Student(): NewStr("Nobody"), ArrayDb() {}
	Student(const NewStr& s): NewStr(s), ArrayDb() {}
	Student(int n): NewStr("Anybody"), ArrayDb(n) {}
	Student(const NewStr& s, int n): NewStr(s), ArrayDb(n) {}
	Student(const NewStr& s, const ArrayDb& a): NewStr(s), ArrayDb(a) {}
```

```
        Student(const char* str, const double* pd, int n): NewStr(str), ArrayDb(pd,n) {}
        ~Student() {}
        double& operator[](int i);
        const double& operator[](int i) const;
        double average() const;
        friend ostream& operator<<(ostream& os, const Student& stu);
        friend istream& operator>>(istream& is, Student& stu);
};
```

Нове оголошення майже не відрізняється від попереднього: той самий набір конструкторів, методів і дружніх операторів, тільки відсутні іменовані поля – тепер потрібні структури входять до складу екземпляра класу як неіменовані підоб'єкти батьківських класів. Зміни зачепили також визначення конструкторів і методів. У списках ініціалізації конструкторів тепер замість імен полів використовують конструктори батьківських класів, як це зазвичай роблять у відкритому одиничному наслідуванні. В тілі методів тепер замість імен об'єктів використовують кваліфікатори імен (замість імені поля використовують ім'я класу для вказання приналежності методу). Порівняйте:

```
//                              КОМПОЗИЦІЯ (ВКЛЮЧЕННЯ)
double Student::average() const {
        return scores.average();
}
double& Student::operator[](int i) {
        return scores[i];
}
const double& Student::operator[](int i) const {
        return scores[i];
}
ostream& operator<<(ostream& os, const Student& stu) {
        os << "Student's " << stu.name << " scores are:\n" << stu.scores;
        return os;
}

//                              ЗАКРИТЕ НАСЛІДУВАННЯ
double Student::average() const {
        return ArrayDb::average();
}
double& Student::operator[](int i) {
        return ArrayDb::operator[](i);
}
const double& Student::operator[](int i) const {
        return ArrayDb::operator[](i);
}
ostream& operator<<(ostream& os, const Student& stu) {
        os << "Student's " << (const NewStr&) stu
                << " scores are:\n" << (const ArrayDb&) stu;
        return os;
}
```

Окремої уваги заслуговує оператор виведення нового класу: тепер, щоб скористатися операторами виведення батьківських класів, доводиться явно перетворювати посилання на *Student* у посилання на *NewStr* чи *ArrayDb*.

Більше нічого модифікувати у проекті не потрібно! Ми можемо використати головну програму і визначення класів *NewStr* та *ArrayDb* з попереднього проекту без жодних змін, і отримаємо такі ж результати, як і раніше.

Варто згадати і про інші можливості. Ви можете легко робити батьківські методи частиною інтерфейсу породженого класу навіть за умов закритого наслідування за допомогою *using* оголошення, як у такому фрагменті:

```
class Student: private NewStr, private ArrayDb {
public:
        Student(): NewStr("Nobody"), ArrayDb() {}
        . . . . . .
        ~Student() {}
```

```
        using ArrayDb::operator[];
        using NewStr::average();
        . . . . . .
};
```

В оголошенні вказують тільки ім'я методу. Одне оголошення робить видимими всі перевантажені методи батьківського класу (*ArrayDb::operator[]*).

Для оголошення класу *Student* можна було б використати і змішаний підхід:

```
class Student: private NewStr {
private: ArrayDb name;
public:
        Student(): NewStr("Nobody"), name() {}
        Student(int n): NewStr("Anybody"), name(n) {}
        . . . . . .
};
```

Компонування та закрите наслідування дуже схожі між собою, проте є відмінності в синтаксисі та способах використання:

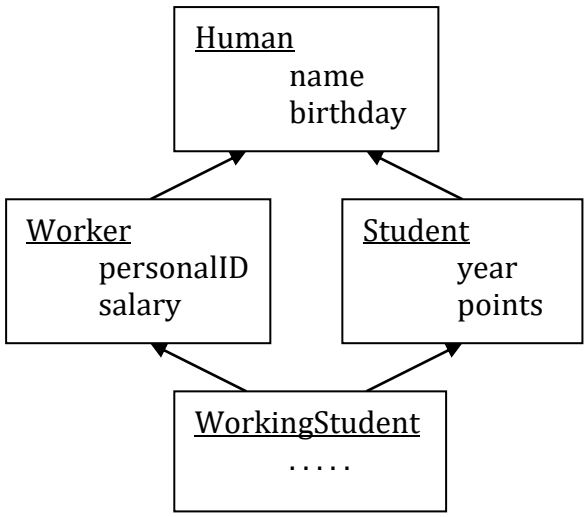
- компонувати можна довільну кількість однотипних об'єктів, а успадкувати тільки один;
- якщо в різних батьківських класів є однойменні члени, то у випадку множинного наслідування можливі колізії, а у випадку компонування – жодних проблем;
- перевага наслідування виявляється тоді, коли батьківські класи мають *protected* методи: для нащадка вони доступні, а для композита – ні;
- наслідування дає змогу робити методи батьківського класу доступними за допомогою *using* оголошення.

Зауважимо, що компонування природніше і використовується частіше.

Захищене наслідування є різновидом закритого. Його задають директивою *protected*:

```
class Second : protected Base { . . . };
```

За умов захищеного наслідування всі *public* елементи базового класу стають *protected* елементами породженого класу. Ця відмінність виявляється тільки в наступних поколіннях класів: вони матимуть доступ до захищених членів, а за умов закритого наслідування (до приватних) – ні.



Повернемось до обговорення *множинного* наслідування: в оголошенні класу вказано два (або більше) батьківські класи. У цьому випадку новий клас наслідує з кожного батьківського класу. Якщо у цих батьків є спільний предок, то виникає низка проблем, що потребують вирішення. Розглянемо приклад: клас *Human* моделює дані особи, що має ім'я та дату народження; Працівник (*Worker*) є Особою, має ідентифікаційний номер і прибуток; Студент (*Student*) є Особою, має колекцію оцінок і певний рік навчання. Як визначити Працюючого Студента? Новий клас мав би містити і оцінки, і прибутки, тому спробуємо зробити його нащадком класів *Worker* і *Student*:

```
class Human { protected: String name; Date birthday; ... };
class Worker: public Human {...};   class Student: public Human {...};
class WorkingStudent: public Worker, public Student {...};
```

На які дані тепер вказує *name* (чи *birthday*) в класі *WorkingStudent*? Екземпляр цього класу успадковує поля даних від кожного з предків, по кожній лінії спорідненості: ((name, birthday), personalID, salary) від *Worker* та ((name, birthday), year, points) від *Student*. Виникла неоднозначність, бо невідомо, де зберігається ім'я працюючого студента, як його модифікувати. Що станеться, якщо ми задамо *Student::name*, а на друк виведемо *Worker::name*? Але найголовніше запитання – навіщо зберігати два значення імені? Безперечно, така ситуація є помилковою. Для правильного оголошення класу *WorkingStudent* треба застосувати нові синтаксичні правила:

- батьківський клас *Human* оголосити віртуальним в *Worker* і *Student* (це означає, що компілятор згенерує «обережний» код успадкування полів батьківського класу: спочатку перевірити, чи такі поля вже є, і лише потім додавати, якщо їх нема; всі неявні виклики конструкторів віртуального батьківського класу буде скасовано – в підкласах тепер їх треба викликати явно);
- деструктор класу *Human* оголосити віртуальним абстрактним;
- явно вказати виклик конструктора *Human* в конструкторах *WorkingStudent*.

```
class Human {
private: String name; Date birthday;
protected:
      virtual void part() const { cout<<name<<' '<<birthday<<'\n'; }
public:
      Human(): name("Billy"), birthday(256) {}
      Human(const NewStr& s): name(s), birthday(256) {}
      virtual ~Human() = 0;
      virtual void show() const;
};
class Worker: virtual public Human {
private: long personalID; double salary;
protected:
      virtual void part() const { cout<<personalID<<' '<<salary<<'\n'; }
public:
      Worker(): Human(),personalID(0),salary(0) {}
      Worker(const NewStr& s, long No): Human(s),personalID(No),salary(100){}
      void show() const;
};
class Student: virtual public Human {
private: int year; ArrayDb points;
protected:
      virtual void part() const { cout<<year<<' '<<points<<'\n'; }
public:
      Student(): Human(),year(1),points() {}
      Student(const NewStr& s, int y): Human(s),year(y),points() {}
      void show() const;
};
class WorkingStudent: public Worker, public Student {
public:
      WorkingStudent(): Worker(),Student() {} //працює Human() за замовчуванням
      WorkingStudent(const NewStr& s, long No, int y)
                              : Human(s),Worker(s,No),Student(s,y) {}
      void show() const;
};
```

Тепер клас *WorkingStudent* отримає один набір правильно ініціалізованих полів класу *Human*. Залишається вияснити, як правильно оголосити методи цього класу, наприклад, метод *show()*. В умовах простого наслідування зазвичай використовують інкрементний підхід: дочірній клас викликає батьківський метод для відображення успадкованих даних, а тоді відображає власні. Тобто, з розростанням дерева підкласів додається код відображення даних, але кожен з підкласів відображає батьківські дані. Для класу *WorkingStudent* це знову не те, що треба: навіщо двічі, через кожен з надкласів, друкувати ім'я і дату народження? Тут

застосуємо інший, модульний підхід: методи *part()* кожного з класів відповідальні тільки за свою частину даних, методи *show()* кожного з класів викликають потрібні частини:

```
void Human::show() const { part(); }
void Worker::show() const { Human::part(); part(); }
void Student::show() const { Human::part(); part(); }
void WorkingStudent::show() const
{ Human::part(); Student::part(); Worker::part(); }
```

На завершення поговоримо про з'ясування типу об'єкта під час виконання програми. Коли може виникнути така потреба? По-перше, якщо в програмі є поліморфна структура даних чи хоча б один поліморфний вказівник (або посилання), по-друге, якщо не вся функціональність описана в інтерфейсі кореневого класу: додаткові методи оголошено на нижчих рівнях ієрархії.

Відомо, що вказівникові на екземпляр батьківського класу можна присвоювати адреси екземплярів породжених класів, масив таких вказівників є поліморфною колекцією об'єктів (її елементами можуть бути екземпляри будь-яких класів ієрархії). У більшості випадків нам не потрібно знати конкретний тип таких об'єктів, адже спеціальну поведінку класу можна описати віртуальними методами, і механізм пізнього зв'язування забезпечить автоматичний вибір правильного методу. Але якщо потрібний метод з'являється в інтерфейсі підкласу, а ми працюємо з вказівником на надклас, то звідки знати, чи зможе об'єкт опрацювати повідомлення? Можуть виникати і інші причини для з'ясування типу об'єкта: наприклад, перехід до наступного етапу виконання програми може залежати від типу результату, отриманого на попередньому етапі.

У мові C++ для ідентифікації типу об'єктів на етапі виконання використовують бібліотеку RTTI (runtime type information). Вона надає програмістові такі засоби:

- оператор *dynamic_cast<type*>* генерує, якщо це можливо, вказівник на породжений тип, за вказівником на батьківський тип; у протилежному випадку оператор повертає 0;
- оператор *typeid* повертає величину, що ідентифікує тип об'єкта;
- структура *type_info* містить інформацію про визначений тип.

Ці засоби можна застосовувати тільки до ієрархії класів з віртуальними методами. Розглянемо їхнє використання на прикладі:

```
#include <iostream>
using namespace std;
#include <cstdlib>        // потрібні для
#include <ctime>          // використання генератора випадкових чисел
#include <typeinfo>       // оголошення класу type_info

class Grand {
private:    int hold;
public:
      Grand(int h=0): hold(h) {}
      virtual ~Grand() {}
// всі класи ієрархії по-своєму відповідають на повідомлення speak()
      virtual void speak() const { cout<<"I am a grand class!\n"; }
      virtual int value() const { return hold; }
};
class Superb : public Grand {
public:
      Superb(int h=0): Grand(h) {}
      virtual void speak() const { cout<<"I am a superb class!!\n"; }
// тут з'являється нова функціональність
      virtual void say() const {cout<<"I hold the grand value of "<<value()<<" !\n";}
};
class Magnificent : public Superb {
private:    char ch;
public:
      Magnificent(int h=0, char c='A'): Superb(h), ch(c) {}
      virtual void speak() const { cout<<"I am a magnificent class!!!\n"; }
```

```
	virtual void say() const { cout<<"I hold the caracter '"<<ch
		<<"' and the integer "<<value()<<" !\n"; }
};

Grand* GetOne() { // випадковим чином генерує екземпляри ієрархії Grand
	Grand* p;
	switch (rand() % 3) {
		case 0: p = new Grand(rand() % 100); break;
		case 1: p = new Superb(rand() % 100); break;
		case 2: p = new Magnificent(rand() % 100, 'A' + rand() % 26); break;
	} return p;
}
int main() {
	srand(time(0)); // запуск генератора випадкових чисел
	Grand* G;
	Superb* S;
	for (int i=0; i<7; i++) {
		G = GetOne();
// Довідуємся тип згенерованого об'єкта: оператор typeid повертає інформаційну
// структуру, яка у відповідь на повідомлення name() повертає рядок – ім'я класу
		cout<<"\n--- MAIN : now processing type ["<<typeid(*G).name()<<"].\n";
		G->speak();
// Чи можна вказівник G перетворити у вказівник на Superb? Якщо так, то результат
// виконання оператора відмінний від нуля (заодновиконали і присвоєння), можна
// надсилати повідомлення say()
		if (S=dynamic_cast<Superb*>(G)) S->say();
// А тут перевірено тип об'єкта на рівність
		if (typeid(Magnificent)==typeid(*G))
			cout<<"--- MAIN : Yes, you're really magnificent.\n";
		delete G;
	}
	return 0;
}
/*   ОТРИМАНІ РЕЗУЛЬТАТИ
--- MAIN : now processing type [class Grand].
I am a grand class!

--- MAIN : now processing type [class Superb].
I am a superb class!!
I hold the grand value of 88 !

--- MAIN : now processing type [class Magnificent].
I am a magnificent class!!!
I hold the caracter 'K' and the integer 44 !
--- MAIN : Yes, you're really magnificent.

--- MAIN : now processing type [class Grand].
I am a grand class!

--- MAIN : now processing type [class Superb].
I am a superb class!!
I hold the grand value of 28 !

--- MAIN : now processing type [class Grand].
I am a grand class!

--- MAIN : now processing type [class Magnificent].
I am a magnificent class!!!
I hold the caracter 'P' and the integer 62 !
--- MAIN : Yes, you're really magnificent.
*/
```

Поекспериментуйте з цим кодом і висновки зробіть самі.