Лекція 5а. Ще раз про: включення VS закрите наслідування; множинне наслідування

1. *Приклад класу зі змістовними статичними методами*
2. *Побудова об'єкта за допомогою включення.*
3. *Побудова об'єкта за допомогою закритого наслідування, порівняння синтаксису.*
4. *Множинне наслідування. Віртуальні батьківські класи.*

У попередній лекції ми вже говорили про закрите та про множинне наслідування. А зараз продемонструємо їхнє застосування ще на декількох прикладах. Почнемо з класу *Date*, що моделює календарну дату. Він стане основою для побудови різними способами класу «Особа», але і сам по собі він досить цікавий. Це перший приклад класу, в якому є досить багато не надуманих статичних членів. Як перевірити, чи високосним є певний рік? У кого довідатися кількість днів у місяці? Чи яка сьогодні дата? Логічно було б інкапсулювати такі дії в класі *Date*. Для запиту про кількість днів у певному місяці не потрібно створювати якийсь об'єкт-дату – достатньо запитати про це в самого класу. Тому відповіді на поставлені вище запитання дають статичні методи класу *Date*. Нагадаємо, що до статичних членів звертаються через клас. Наприклад, нинішню дату можна отримати так: *Date today = Date::DateToday();* Повне оголошення класу *Date* наведено нижче.

Файл Date.h

```
#pragma once
#include <iostream>

class Date
{
private:
	// дата зберігає рік, місяць, день
	int year, month, day;
	// кількість днів у місяцях невисокосного року
	static int daysInMonth[12];
public:
	Date() :year(1), month(1), day(1) {}
	// Дату можна створити за кількістю днів, що минула від початку тисячоліття
	explicit Date(int);
	Date(unsigned d, unsigned m, unsigned y);

	// Клас може надавати послуги без створення екземплярів:
	// - перевірка, чи є вказаний рік високосним
	static bool isLeapYear(int year)
	{
		return year % 400 == 0 || year % 4 == 0 && year % 100 != 0;
	}
	// - повідомляє кількість днів у році
	static int DaysInYear(int year)
	{
		return 365 + isLeapYear(year);
	}
	// - повідомляє кількість днів у вказаному місяці вказаного року
	static int DaysInMonth(int month, int year = 2019)
	{
		if (month == 2 && isLeapYear(year)) return 29;
		else return daysInMonth[month - 1];
	}
	// - створює екземпляр, що зображає поточну дату
	static Date DateToday();
```

```
	void printOn(std::ostream& os) const
	{
		os.width(2);
		os << day << (month < 10 ? ".0" : ".") << month << '.' << year;
	}
	// дати календаря можна порівнювати
	bool operator>(const Date& date) const
	{
		return this->year > date.year ||
			this->year == date.year && (this->month > date.month ||
				this->month == date.month && this->day > date.day);
	}
	// "відстань" між датами, обчислена в днях
	int operator-(const Date& date) const
	{
		return this->DaysFromStart() - date.DaysFromStart();
	}
	// дату можна змінити на вказану кількість днів
	Date operator+(int days) const
	{
		return Date(this->DaysFromStart() + days);
	}
	// сервісні методи для читання полів екземпляра
	int Year() const { return year; }
	int Month() const { return month; }
	int Day() const { return day; }
	// скільки днів минуло від початку тисячоліття
	int DaysFromStart() const;
};

std::ostream& operator<<(std::ostream& os, const Date& date);
```

Кожна особа має ім'я, має день народження. Тому між сутностями «Особа» і «Дата» існує відношення *has-a*. «Особа містить Дату». З попередньої лекції ми знаємо, що таке відношення можна змоделювати в програмі двома способами. Випробуємо обидва.

Включення. Для оголошення полів складного об'єкта можна використати раніше визначені класи: стандартні або власні. У результаті отримаємо композит, що містить вкладені об'єкти та використовує їхню реалізацію. Про наслідування інтерфейсу вкладених об'єктів мова не йде взагалі. Включення одного об'єкта до складу іншого очевидним чином моделює відношення *has-a*. Використаємо цей підхід для оголошення класу *Human* (людина, особа).

Файл Human.h

```
#pragma once
#include <string>
#include "Date.h"

class Human
{
private:
	std::string name;
	Date birthday;
public:
	Human();
	Human(const char* n, unsigned d, unsigned m, unsigned y)
		:name(n), birthday(Date(d, m, y)) {}
	virtual ~Human();

	// методи для читання даних екземпляра
	Date Birthday() const { return birthday; }
	std::string Name() const { return name; }
```

```
	// виведення в потік
	virtual void printOn(std::ostream& os) const
	{
		os << name << " (";
		birthday.printOn(os);
		os << ')';
	}
};

std::ostream& operator<<(std::ostream& os, const Human& human);
```

Конструктор ініціалізує кожне з полів; методи читання повертають поля; метод виведення в потік використовує здатність рядка і екземпляра *Date* робити таке виведення.

Закрите наслідування. Нагадаємо, що при закритому наслідуванні всі члени базового класу стають приватними членами похідного класу. Таким чином підклас отримує від надкласу реалізацію і не наслідує інтерфейс. Отже, закрите наслідування, як і включення, моделює відношення *has-a*. Запропонуємо альтернативне оголошення класу «Особа» з використанням закритого наслідування. Щоб не плутатися з іменами, назвемо його *Person* (на відміну від *Human*).

Файл Person.h

```
#pragma once
#include <string>
#include "Date.h"

class Person :
	private Date
{
private:
	std::string name;
public:
	Person();
	Person(const char* n, unsigned d, unsigned m, unsigned y)
		:Date(d, m, y), name(n) {}
	// методи читання даних екземпляра
	// - день нарождення отримуємо в результаті
	//   неявного приведення до базового типу
	Date Birthday() const { return *this; }
	std::string Name() const { return name; }
	// Для виведення в потік потрібен також метод базового класу
	virtual void printOn(std::ostream& os) const
	{
		os << name << " (";
		Date::printOn(os);
		os << ')';
	}
	virtual ~Person();
};

std::ostream& operator<<(std::ostream& os, const Person& person);
```

Порівняно з попередньою реалізацією, у класі поменшало полів: тепер об'єкт-дата неявно входить до складу екземпляра підкласу. Конструктор викликає в списку ініціалізації конструктора базового класу. Метод читання дати народження повертає сам об'єкт-особу, але за рахунок автоматичного приведення до базового типу метод поверне копію успадкованого об'єкта-дати. У методі виведення явно викликається метод виведення базового класу.

У використанні класи *Human* і *Person* не відрізняються. У фрагменті нижче продемонстровано використання конструктора за промовчанням, конструктора з параметрами, методів доступу до частин об'єкта і оператора виведення в потік.

Фраґмент файла MainProg.cpp

```
#include <iostream>
#include "Human.h"
#include "Person.h"

int main()
{
	Human baby;
	Human Ernest("Ernest Miller Hemingway", 21, 7, 1899);
	std::cout << baby << '\n';
	std::cout << Ernest << '\n';
	std::cout <<"Human "<<Ernest.Name()<<" was born on "<<Ernest.Birthday()<<"\n\n";
	Person pups;
	Person JackLondon("John Griffith Chaney", 12, 1, 1876);
	std::cout << pups << '\n';
	std::cout << JackLondon << '\n';
	std::cout << "Person " << JackLondon.Name()
		<< " was born on " << JackLondon.Birthday() << '\n';
	system("pause");
```

Отримано в консолі

```
Unknown (30.03.2019)
Ernest Miller Hemingway (21.07.1899)
Human Ernest Miller Hemingway was born on 21.07.1899

Unknown ( 1.01.1)
John Griffith Chaney (12.01.1876)
Person John Griffith Chaney was born on 12.01.1876
```

Відмінності в коді альтернативних класів систематизуємо у вигляді таблиці

	Включення	Закрите наслідування

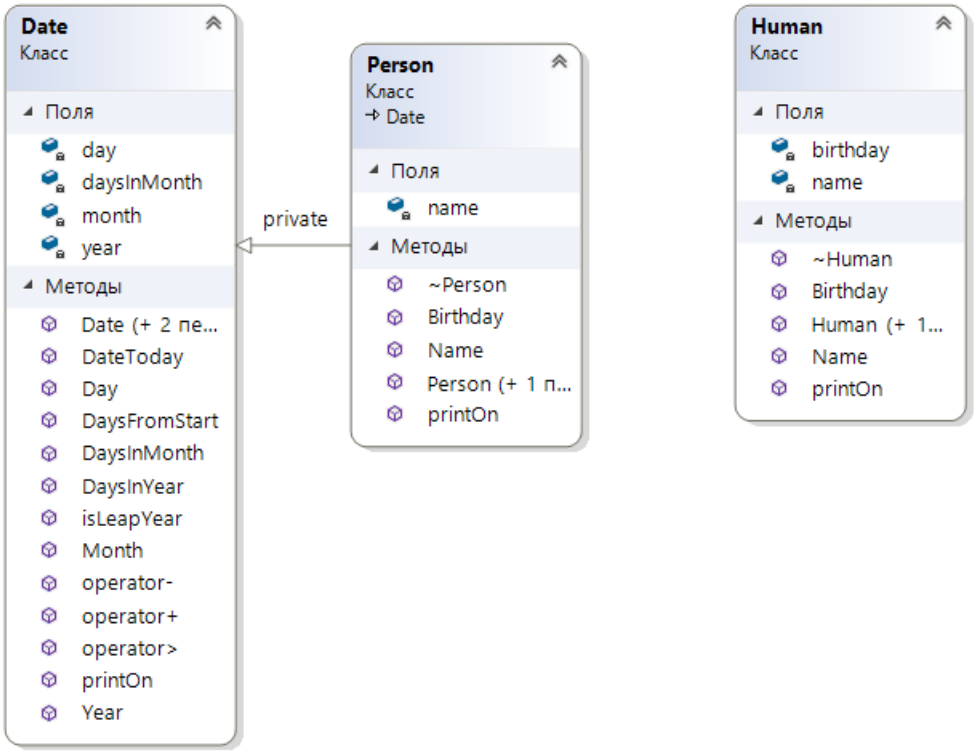
Оголошення класу	`class Human`
	`{`
	`private:`
	`  std::string name;`
	`  Date birthday;`
	`// поле типу Date`
		`class Person :`
		`  private Date`
		`{`
		`private:`
		`  std::string name;`
		`// успадкований об’єкт Date`

Конструктор	`Human(const char* n, unsigned d,`
	`  unsigned m, unsigned y)`
	`:name(n), birthday(Date(d, m, y))`
	`{}`
	`// ініціалізатори полів`
		`Person(const char* n, unsigned d,`
		`  unsigned m, unsigned y)`
		`  :Date(d, m, y), name(n) {}`
		`// конструктор базового класу`

Доступ до частини	`Date Birthday() const {`
	`  return birthday; }`
	`// повертає поле`
		`Date Birthday() const {`
		`  return *this; }`
		`// повертає успадкований об’єкт`

Використання реалізації	`virtual void printOn(`
	`  std::ostream& os) const`
	`{`
	`  os << name << " (";`
	`  birthday.printOn(os);`
	`  os << ')';`
	`}`
	`// звертання до вкладеного об’єкта`
		`virtual void printOn(`
		`  std::ostream& os) const`
		`{`
		`  os << name << " (";`
		`  Date::printOn(os);`
		`  os << ')';`
		`}`
		`// виклик базової реалізації методу`

На завершення обговорення різних способів реалізації відношення *has-a* наведемо діаграми створених класів:



Повернемось до обговорення *множинного* наслідування: в оголошенні класу вказано два (або більше) батьківські класи. У цьому випадку новий клас наслідує з кожного батьківського класу. Більшість мов програмування не підтримують такого складного інструменту. Програмісти ж на C++ мають у своєму розпорядженні потужний засіб швидкого створення класів зі складною поведінкою. Розглянемо його використання на прикладах.

Наслідування **від різних** класів здебільшого не викликає проблем. У цій лекції ми вже розглядали клас *Human*: людина має ім'я, дату народження, вміє говорити, повідомляти свої дані, друкувати себе в потік. Розглянемо ще один клас – *Robot*. Робот має модель, серійний номер, джерело живлення, вміє виконувати роботу, друкувати себе в потік. Класи не споріднені. На їх основі можна швидко створити новий клас, що матиме ознаки і можливості обох.

Файл Robotics.h

```
class Robot
{
private:
	std::string model;
	std::string powerType;
	int serialNo;
	static int SerialNumber;
public:
	Robot();
	Robot(const char* mod, const char* pow);
	virtual ~Robot() {}
	virtual void printOn(std::ostream& os) const;
	Robot& doWork(std::string aWork)
	{
		std::cout << model << " #" << serialNo << " did work [" << aWork << "]\n";
		return *this;
	}
};

std::ostream& operator<<(std::ostream& os, const Robot& robot);
```

```
// множинне наслідування від різних класів

class Robocop :
	public Human,
	public Robot
{
private:
	Date releaseDate;
public:
	Robocop() :Human(), Robot(), releaseDate() {}
	Robocop(const char* n, unsigned d, unsigned m, unsigned y)
		:Human(n, d, m, y), Robot("Police", "Nuclear"), releaseDate(d, m, y + 30) {}
	virtual void printOn(std::ostream& os) const
	{
		os << (Human)*this << " & " << (Robot)*this
			<< " manufactured on " << releaseDate;
	}
	virtual ~Robocop() {}
};
// перевизначення оператора виведення - обов'язкове
std::ostream& operator<<(std::ostream& os, const Robocop& robocop);
```

Тепер *Robocop* містить дані обох базових класів: ім'я, модель, серійний номер, дату народження (людської частини), джерело безперебійного живлення – та вміє робити те ж, що й базові класи: говорити, виконувати роботу, виводити себе в потік. Клас може додавати власні поля та поведінку, як і при звичайному наслідуванні. Так *Robocop* містить додаткове поле – дату виготовлення – того ж типу, що й успадковане поле *birthday*. Випробуємо його в програмі.

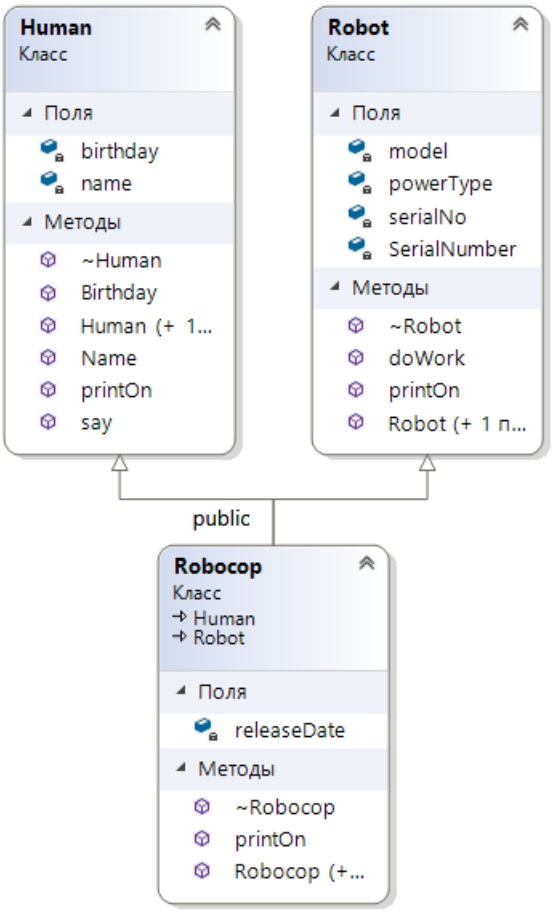
Фраґмент файла MainProg.cpp

```
#include <iostream>

#include "Robotics.h"

int main()
{
	. . .
	Robot Roby;
	std::cout << Roby << '\n';
	Roby.doWork("Make a cup of coffee").doWork("Make a cake");
	std::cout << '\n';
	Robocop John("Judge Dredd", 11, 7, 1982);
	std::cout << John << '\n';
	John.doWork("Push the wall").doWork("Grab the criminal");
	John.say("Asta La Vista, BABY!");
	system("pause");
```

Отримано в консолі

```
ROBOT <model: 'Base XL' No: 10001001 on Solar energy>
Base XL #10001001 did work [Make a cup of coffee]
Base XL #10001001 did work [Make a cake]

Judge Dredd (11.07.1982) & ROBOT <model: 'Police' No: 10001002 on Nuclear energy>
manufactured on 11.07.2012
Police #10001002 did work [Push the wall]
Police #10001002 did work [Grab the criminal]
Judge Dredd: "Asta La Vista, BABY!"
```

Бачимо, що *Robocop* проявляє себе і як *Robot*, і як *Human*. Діаграму класів подано нижче.



Побудувати новий клас вдалося майже без проблем: базові класи мають різні набори полів і методів, тому не виникає суперечностей. Хіба що одна. Обидва базові класи мають метод *printOn* і пов'язаний з ним оператор виведення в потік. З попереднього досвіду ми знаємо, що для виведення в потік об'єктів деякої ієрархії достатньо один раз визначити оператор виведення, який викликає метод виведення конкретного класу. Для цього такі методи оголошують віртуальними. Ми так і зробили, й у всіх трьох класах: *Human*, *Robot*, *Robocop* – визначили віртуальний *printOn*. Проте підкласові у спадок дістається **два** оператори виведення: по одному від кожного з базових класів. У такій ситуації компілятор не може вирішити, яким з них користуватися, і сигналізує про помилку. Щоб виправити неоднозначність, доводиться визначати оператор виведення утретє, вже для нового підкласу.

Ще одна невелика проблема пов'язана з роботою системи інтелектуальних підказок Visual Studio: вона вперто не хоче признавати за об'єктом *Robocop* можливості виконувати *say()* і позначає його виклик як помилку. Добре, що з компіляцією все гаразд!

Інакше виглядає наслідування **від споріднених** класів. Якщо у базових класів 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 дату вступу до університету. Як визначити Працюючого Студента? Новий клас мав би містити і оцінки, і прибутки, тому спробуємо зробити його нащадком класів *Worker* і *Student*:

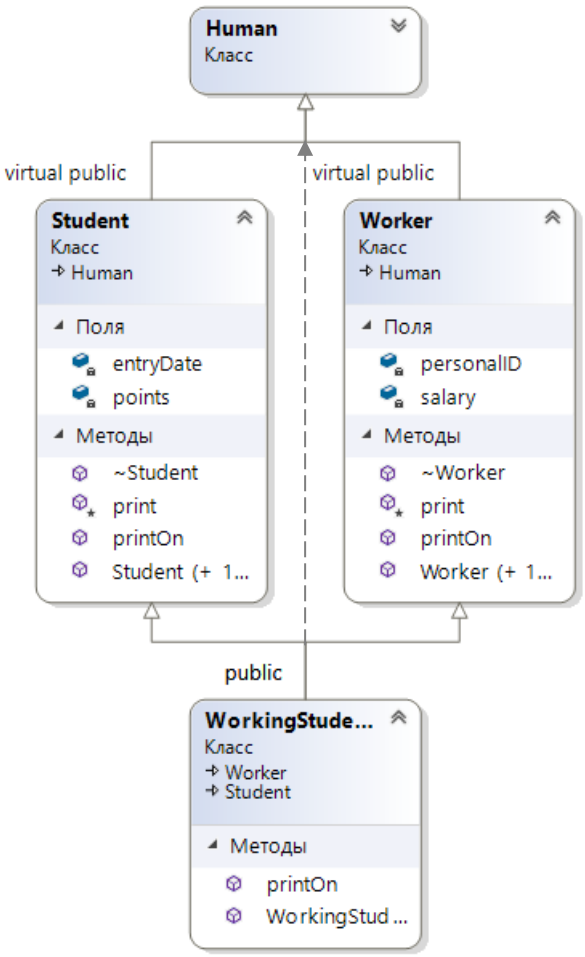
```
class Human { protected: string name; Date birthday; ... };
class Worker: public Human {...};   class Student: public Human {...};
class BadWorkingStudent: public Worker, public Student {...};
```

На які дані тепер вказує *name* (чи *birthday*) в класі *BadWorkingStudent*? 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);
- явно вказати виклик конструктора *Human* в конструкторах *WorkingStudent*; якщо цього не зробити, компілятор автоматично викличе конструктор за замовчуванням *Human*.

```
#pragma once
#include "Human.h"

class Worker :
	virtual public Human
{
private:
	long personalID;
	double salary;
protected:
	// метод "часткового" виведення в потік – друкує частину працівника
	virtual void print(std::ostream& os) const
	{
		os << " #" << personalID << " $" << salary;
	}
public:
	Worker() : Human(), personalID(0), salary(0) {}
	// Звичайна ініціалізація
	Worker(const char* n, unsigned d, unsigned m, unsigned y, long No)
		: Human(n, d, m, y), personalID(No), salary(300) {}
	virtual ~Worker() {}
	virtual void printOn(std::ostream& os) const
	{
		Human::printOn(os);
		os << " with ";
		this->print(os);
	}
};

class Student :
	virtual public Human
{
private:
	Date entryDate;
	int points[50];
protected:
	// метод "часткового" виведення в потік – друкує частину студента
	virtual void print(std::ostream& os) const
	{
		os << " entered at " << entryDate;
	}
public:
	Student() : Human(), entryDate(Date()) {}
	// Звичайна ініціалізація
	Student(const char* n, unsigned d, unsigned m, unsigned y)
		: Human(n, d, m, y), entryDate(Date(1,9,2016)) {}
	virtual ~Student() {}
	virtual void printOn(std::ostream& os) const
	{
		Human::printOn(os);
		this->print(os);
	}
};

class WorkingStudent :
	public Student,
	public Worker
{
public:
	WorkingStudent() : Student(), Worker() {}
	// Незвичайна ініціалізація: усіх трьох складових
	WorkingStudent(const char* n, unsigned d, unsigned m, unsigned y, long No) :
		Human(n, d, m, y),
		Student(n, d, m, y),
		Worker(n, d, m, y, No) {}
	// Виведення в потік "збирає частини"
```

```
        virtual void printOn(std::ostream& os) const
        {
                Human::printOn(os);
                Student::print(os);
                Worker::print(os);
        }
};
```



Діаграму класів такого успадкування зображено ліворуч. Зверніть увагу на підписи біля стрілок успадкування. Деякі автори вважають, що на такій діаграмі варто зображати ще одну стрілку (на рисунку вона зображена пунктиром), яка б зображала особливий спосіб успадкування полів класу *Human* підкласом *WorkingStudent*.

Тепер клас *WorkingStudent* отримає один набір правильно ініціалізованих полів класу *Human*. Залишається вияснити, як правильно оголосити методи цього класу, наприклад, метод *printOn()*. 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. Так було, наприклад, у класі *Person*: *Date*, або у *Robocop*: *Human*, *Robot*. 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 *print()* кожного з класів відповідальні тільки за свою частину даних, методи *printOn()* кожного з класів викликають потрібні частини:

```
virtual void Worker::printOn(std::ostream& os) const
{       Human::printOn(os); this->print(os);    }

virtual void Student::printOn(std::ostream& os) const
{       Human::printOn(os); this->print(os);    }

virtual void WorkingStudent::printOn(std::ostream& os) const
{       Human::printOn(os); Student::print(os); Worker::print(os);    }
```