**Лабораторна робота №11. Використання специфікаторів доступу при спадкуванні.**

**Мета:** Набуття навичок в розробці програм, де використовуються похідні класи та дружні функції, а також розв’язання області видимості.

**Завдання**

1. Для заданого класу

class Parent

{

public:

int m\_public;

Parent(int a=1, int b=2, int c=3)

{this->m\_public = a; this->m\_private = b;

this->m\_protected = c;

cout << "constructor class Parent " << endl;}

~Parent() {cout << "destructor class Parent" << endl;};

friend void Pprnt(Parent &);

private:

int m\_private;

protected:

int m\_protected;

};

* створити захищений похідний клас class Child: protected Parent, в якому ввести дві змінні одну захищену, другу – закриту (private) та надати доступ до m\_public класу Parent;
* створити відповідний конструктор ініціалізації з урахуванням наявності конструктору для Parent (значення m\_public береться з класу Parent);
* для обох класів записати дружні функції друку всіх членів класу;
* створити закритий похідний клас class Child2: private Parent, в якому ввести дві змінні одну захищену, другу – закриту (private) та надати доступ до m\_public класу Parent;
* в головній програмі викликати функції друку для демонстрації звернень до змінних.

1. Закінчити роботу з класами *Person* та *Student* за завданням попередньої лабораторної роботи.
2. Результати надсилати на електронну адресу викладача

[**t.i.lumpova@gmail.com**](mailto:t.i.lumpova@gmail.com)у вигляді заголовкового файлу та cpp-файлу з іменем у форматі

Для заголовкового файлу

**<Прізвище англійською>.h**

Наприклад, Ivanov.h

Для cpp-файлу:

**<Номер групи> <Номер лабораторної><Прізвище англійською>.cpp**

Наприклад, 31-01Ivanov.cpp.

Тему в заголовку листа записати

**ООП<Номер групи>-ЛР <Номер лабораторної>-<Прізвищеанглійською>**

**Строк відсилки ЛР ІПЗ-31 08.11.2020**

Всі запитання, що виникнуть, надсилайте на електронну адресу викладача, тему в заголовку листа записати

**ООП<Номер групи>-Запитання-<Прізвище англійською>**.

**ТЕОРЕТИЧНІ ВІДОМОСТІ**

**Контейнерні класи**

**Контейнерний клас** (або ***“клас-контейнер”***) в мові C++ — це клас, призначений для зберігання і організації декількох об’єктів певного типу даних (користувацьких чи фундаментальних). Існує багато різних контейнерних класів, кожен з яких має свої переваги, недоліки або обмеження у використанні. Найбільш використовуваним контейнером в програмуванні є масив. Хоча в мові C++ є звичайні стандартні масиви, більшість програмістів використовують контейнерні класи-масиви: std::array або std::vector через переваги, які вони надають. На відміну від стандартних масивів, контейнерні класи-масиви мають можливість динамічної зміни свого розміру, коли елементи додаються або видаляються. Це не тільки робить їх більш зручними, ніж звичайні масиви, а й безпечнішими для використання.

Зазвичай, **функціонал класів-контейнерів** мови C++ наступний:

   Створення пустого контейнера (через конструктор).

   Додання нового об’єкта в контейнер.

   Видалення об’єкта з контейнеру.

   Перегляд кількості об’єктів, які знаходяться на даний момент в контейнері.

   Очистка контейнера від всіх об’єктів.

   Доступ до збережених об’єктів.

   Сортування об’єктів/елементів (не завжди).

Іноді функціонал контейнерних класів може бути не настільки великим, як це зазначено вище. Наприклад, контейнерні класи-масиви часто не мають функціоналу додавання/видалення об’єктів, тому що вони і так повільні, і розробник просто не хоче збільшувати навантаження.

**Типом відносин** в класах-контейнерах є ***«член чогось»*** (не в сенсі члена класу C++). Наприклад, елементи масиву «є членами» масиву (належать йому).

**Типи контейнерних класів**

**Контейнерні класи зазвичай бувають двох типів:**

**Контейнери значення** — це композиції, які зберігають копії об’єктів (і відповідальні за створення/знищення цих копій).

**Контейнери посилання** — це агрегації, які зберігають вказівники або посилання на інші об’єкти (і не відповідальні за створення/знищення цих об’єктів).

В мові C++ контейнери зазвичай містять тільки один тип даних. Наприклад, якщо у вас цілочисельний масив, то він може містити тільки цілочисельні значення. C++ не дозволяє змішувати різні типи даних всередині одного контейнера. Якщо вам потрібні контейнери для зберігання значень типів int і double, то вам доведеться написати два окремих контейнери (або використовувати шаблони). Незважаючи на обмеження їх використання, контейнери надзвичайно корисні, так як роблять програмування простішим, безпечнішим і швидшим.

Розглянемо приклад класу-масиву типу контейнера значення ArrayInt, в якому зберігатимуться копії елементів, а не самі елементи. Щоб наш масив міг змінювати свою довжину, то нам потрібно використовувати **динамічне виділення пам’яті**, що означає, що ми будемо використовувати вказівник для зберігання даних. Потрібно додати два конструктори, щоб мати можливість створювати об’єкти класу ArrayInt: перший створюватиме порожній масив, другий — масив заданого розміру: Потрібні функції, які виконуватимуть очистку ArrayInt. По-перше, додамо [**деструктор**](https://acode.com.ua/urok-128-destruktory/), який просто звільнятиме будь-яку динамічно виділену пам’ять. По-друге, напишемо функцію erase(), яка виконуватиме очистку масиву і скидатиме його довжину на 0:

#ifndef ARRAYINT\_H

#define ARRAYINT\_H

#include <cassert> // для assert()

class ArrayInt

{

private:

    int m\_length;

    int \*m\_data;

public:

    ArrayInt():

        m\_length(0), m\_data(nullptr) { }

    ArrayInt(int length): m\_length(length)

    {

        assert(length >= 0);

        if (length > 0)

            m\_data = new int[length];

        else

            m\_data = nullptr;

    }

    ~ArrayInt()

    {

        delete[] m\_data;

    }

    void erase()

    {

        delete[] m\_data;

        /\* Вказуємо m\_data значення nullptr, щоб на виході не було висячого вказівника \*/

        m\_data = nullptr;

        m\_length = 0;

    }

    int& operator[](int index)

    {

        assert(index >= 0 && index < m\_length);

        return m\_data[index];

    }

    int getLength() { return m\_length; }

};

#endif