МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Центральноукраїнський національний технічний університет
Механіко-технологічний факультет

ЗВІТ
ПРО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ № 12
з навчальної дисципліни “Базові методології та технології програмування”

ПРОГРАМНА РЕАЛІЗАЦІЯ АБСТРАКТНИХ ТИПІВ ДАНИХ

ЗАВДАННЯ ВИДАВ:
доцент кафедри кібербезпеки
та програмного забезпечення
Доренський О. П.

ВИКОНАВ:
студент академічної групи КН-23
Клюй А.Я.

ПЕРЕВІРИВ:
викладач кафедри кібербезпеки
та програмного забезпечення
Дрєєва Г.М.

Кропивницький - 2024

МЕТА: набути ґрунтовних вмінь і практичних навичок об’єктного аналізу й проектування, створення класів C++ та тестування їх екземплярів, використання препроцесорних директив, макросів і макрооператорів під час реалізації програмних засобів у кросплатформовому середовищі Code::Blocks.

ЗАВДАННЯ ДО ЛАБОРАТОРНОЇ РОБОТИ:

1. Як складову заголовкового файлу `ModulesПрізвище.h` розробити клас `ClassLab12_Прізвище` - формальне представлення абстракції сутності предметної області (об’єкта) за варіантом, - поведінка об’єкта якого реалізовує розв’язування задачі 7.1.

2. Реалізувати додаток `Teacher`, який видає 100 звукових сигналів і в текстовий файл `TestResults.txt` записує рядок “Встановлені вимоги порядку виконання лабораторної роботи порушено!”, якщо файл проекта `main.cpp` під час його компіляції знаходився не в \Lab12\prj, інакше — створює об’єкт класу `ClassLab12_Прізвище` із заголовкового файлу `ModulesПрізвище.h` та виконує його unit-тестування за тест-сьютом(ами) із \Lab12\TestSuite\, протоколюючи результати тестування в текстовий файл \Lab12\TestSuite\TestResults.txt.

ВАРІАНТ № 37

ЗАДАЧА 12.1

Дано наступну сутність предметної області (об’єкт).

Об’єкт[1] (екземпляр) класу ClassLab12 _Прізвище, як абстракція даної сутності предметної області, за наданим інтерфейсом забезпечує:

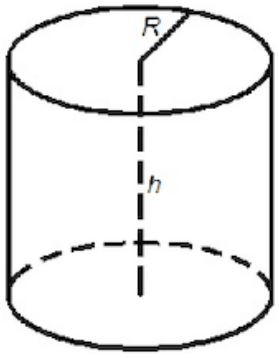
- надання[2] значень своїх атрибутів;
- надання значення кількості рідини і «лави» (об’єму)[3];
- зміну значення заданого атрибута(ів)[4]

[1] Під час створення об’єкта класу всі його атрибути ініціалізуються конструктором.

[2] Під наданням розуміється повернення результату відповідними функціями-членами об’єкта класу.

[3] Об’єм обчислюється і повертається відповідного функтісю-членом (методом) об’єкта класу за значеннями його

[4] Всі дані-члени класу є закритими (private); доступ до них (читання, запис) реалізують відповідні відкриті функції-члени (public), які у свою чергу забезпечують валідацію вхідних



Об’єм циліндра рівний добутку площі його основи на висоту:

$$V = \pi R^2 h \qquad \text{або} \qquad V = S_o h$$

де, V - об’єм циліндра,

So - площа основи,

R - довжина радіуса,

h - висота.

ХІД РОБОТИ

Концептуалізація предметної області:

об’єктом, згідно з варіантом завдання є лавова лампа.

Об’єктний аналіз

`ClassLab12_Klui`: цей клас є абстракцією циліндра і забезпечує:

1) ініціалізацію атрибутів об’єкта при створенні;
2) надання значень своїх атрибутів (радіус та висота);
3) обчислення та надання значень об’єму;
4) зміну значення атрибутів (радіус та висота)

Атрибути класу:

- `radius` (private float): зберігає значення радіуса основи циліндра.
- `height` (private float): зберігає значення висоти циліндра.

Методи класу:

Конструктор: ініціалізує значення атрибутів `radius` та `height`.

Методи доступу:

- `getRadius()`: повертає значення радіуса.
- `setRadius(float value)`: встановлює нове значення радіуса.
- `getHeight()`: повертає значення висоти.
- `setHeight(float value)`: встановлює нове значення висоти.

Визначення інтерфейсів сутності ПрО

Сутність `ClassLab12_Klui`

Клас `ClassLab12_Klui` представляє циліндр як об'єкт у програмному забезпеченні. Він має наступні інтерфейси (методи), які забезпечують доступ до його атрибутів та функціонал для обчислень, пов'язаних з циліндром.

Конструктор `ClassLab12_Klui`. Цей метод автоматично викликається при створенні нового об'єкта класу `ClassLab12_Klui`. Конструктор ініціалізує атрибути `radius` і `height` початковими значеннями, забезпечуючи, що об'єкт починає своє існування у коректному стані.

Методи доступу до атрибутів:

- `float getRadius()`: цей метод повертає поточне значення атрибута `radius`. Він дозволяє отримати радіус циліндра, що є важливим для обчислень та відображення інформації.

- `void setRadius(float value)`: цей метод встановлює нове значення атрибута `radius`. Вхідне значення `value` перевіряється, щоб забезпечити його валідність. Якщо перевірка пройшла успішно, нове значення зберігається в атрибуті.

- `float getHeight()`: цей метод повертає поточне значення атрибута `height`. Він дозволяє отримати висоту циліндра для подальших обчислень та аналізу.

- `void setHeight(float value)`: цей метод встановлює нове значення атрибута `height`. Вхідне значення `value` також перевіряється на валідність перед збереженням.

Методи обчислення:

- `float calculateVolume()`: цей метод обчислює та повертає об'єм циліндра. Він використовує значення атрибутів `radius` та `height` і застосовує формулу об'єму циліндра.

Принципи та властивості:

- Інкапсуляція: усі атрибути (`radius, height`) є приватними (`private`), тобто вони не можуть бути змінені безпосередньо ззовні класу. Це забезпечує контроль над тим, як змінюються ці атрибути, та дозволяє додавати валідацію входу в методах сеттерів.

- Інтерфейси: публічні методи (`public`) класу визначають його інтерфейс. Вони забезпечують доступ до атрибутів та функціональність класу. Використання цих методів дозволяє іншим частинам програми взаємодіяти з об'єктом, не знаючи деталей його реалізації.

- Валідація: методи сеттерів перевіряють вхідні значення, забезпечуючи валідність і коректність атрибутів класу. Це допомагає уникнути помилок і зберігає об'єкт у стабільному стані.

Аналіз та постановка задачі 12.1

Постановка задачі: необхідно реалізувати клас `ClassLab12_Прізвище`, який є абстракцією циліндра. Клас повинен мати закриті `(private)` атрибути `radius` та `height`, і публічні `(public)` методи для доступу до цих атрибутів, їх зміни, а також для обчислення об’єму циліндра.

Завдання:

1. Ініціалізація атрибутів: при створенні об’єкта класу всі його атрибути мають ініціалізуватись конструктором.

2. Надання значень атрибутів: ззабезпечити повернення значень атрибутів відповідними геттерами.

3. Обчислення об’єму: забезпечити обчислення об’єму циліндра за формулою $V = \pi * r^2 * h$ відповідним методом.

4. Зміна значення атрибутів: забезпечити зміну значень атрибутів за допомогою відповідних сеттерів.

5. Валідація: всі дані-члени класу повинні бути закритими `(private)`; доступ до них (читання, запис) реалізують відповідні відкриті функції-члени `(public)`, які забезпечують валідацію вхідних даних.

Лістінг проєкту ModolesKlui

```
#ifndef ModulesKlui_h
#define ModulesKlui_h

#include "ClassLab12_Klui.hpp"

#endif /* ModulesKlui_h */
```

Цей файл є заголовковим файлом для проєкту `ModulesKlui`. Він запобігає багаторазовому включенню свого вмісту завдяки препроцесорним директивам `#ifndef`, `#define`, і `#endif`, що забезпечує уникнення помилок при компіляції, пов’язаних з повторним визначенням. Файл включає заголовковий файл `ClassLab12_Klui.hpp`, який містить визначення класу `ClassLab12_Klui`. Це дозволяє використовувати клас `ClassLab12_Klui` в інших файлах проєкту, які включають `ModulesKlui.h`. Забезпечуючи централізоване управлінння залежностями та спрощуючи підтримку коду.

Лістінг проєкту ClassLab12_Klui.cpp

```
#include "ClassLab12_Klui.hpp"
#include <cmath>

ClassLab12_Klui::ClassLab12_Klui(): radius(0.0f), height(0.0f)
{
}
```

```
float ClassLab12_Klui:: getRadius(){
    return radius;
}

void ClassLab12_Klui:: setRadius(float value){
    radius = value;
}

float ClassLab12_Klui:: getHeight(){
    return height;
}

void ClassLab12_Klui:: setHeight(float value){
    height = value;
}

float ClassLab12_Klui::calculateVolume(){
    return M_PI * radius * radius * height;
}

float ClassLab12_Klui::calculateArea(){
    return calculateVolume() / height;
}
```

Артефакт 1: ініціалізація атрибутів класу

Конструктор:

```
ClassLab12_Klui::ClassLab12_Klui(): radius(0.0f), height(0.0f)
```

• Призначення: ініціалізує об'єкт класу `ClassLab12_Klui` з початковими значеннями атрибутів `radius` та `height` рівними `0.0f`.

• Тип: `float`

• Початкові значення: `radius = 0.0f, height = 0.0f`

• Опис: цей конструктор створює об'єкт циліндра з початковим радіусом і висотою, що дорівнюють нулю.

Артефакт 2: методи доступу до атрибутів

Геттери:

```
- float ClassLab12_Klui::getRadius() { return radius; }
```

• Призначення: повертає поточне значення радіуса циліндра.

• Тип: `float`

• Опис: цей метод дозволяє отримати поточне значення атрибуту `radius`

```
- float ClassLab12_Klui::getHeight() { return height; }
```

• Призначення: повертає поточне значення висоти циліндра.

• Тип: `float`

• Опис: цей метод дозволяє отримати поточне значення атрибуту `height`

Сеттери:

- `void ClassLab12_Klui::setRadius(float value) { radius = value; }`

• Призначення: встановлює нове значення радіуса циліндра.

• Тип: `void`, параметр `value` типу `float`

• Опис: цей метод дозволяє встановити нове значення атрибуту `radius`, щоб змінити розмір циліндра.

- `void ClassLab12_Klui::setHeight(float value) { height = value; }`

• Призначення: встановлює нове значення висоти циліндра.

• Тип: `void`, параметр `value` типу `float`

• Опис: цей метод дозволяє встановити нове значення атрибуту `height`, щоб змінити висоту циліндра.

Артефакт 3: метод обчислення об'єму

Метод

```
- float ClassLab12_Klui::calculateVolume(){
  return M_PI * radius * radius * height;}
```

• Призначення: обчислює об'єм циліндра.

• Тип: `float`

• Формула: $V = \pi * r^2 * h$

• Опис: цей метод використовує значення атрибутів `radius` і `height` для обчислення об'єму циліндра за стандартною формулою об'єму.

Артефакт 4: метод обчислення площі основи

Метод

```
float ClassLab12_Klui::calculateArea(){
return calculateVolume() / height;}
```

• Призначення: обчислює площу основи циліндра.

• Тип: `float`

• Формула: $S_o = V / h$

• Опис: цей артефакт представляє метод `calculateVolume()`, який обчислює площу основи циліндра

Лістінг проєкту ClassLab12_Klui.hpp

```
#ifndef ClassLab12_Klui_hpp
#define ClassLab12_Klui_hpp

class ClassLab12_Klui{
private:
    float radius;
    float height;
public:
    ClassLab12_Klui();

    float getRadius();
    void setRadius(float value);

    float getHeight();
    void setHeight(float value);

    float calculateVolume();

    float calculateArea();
};

#endif /* ClassLab12_Klui_hpp */
```

Артефакт 1: Оголошення класу та інклуди

Препроцесорні директиви:

```
#ifndef ClassLab12_Klui_hpp

 #define ClassLab12_Klui_hpp

#endif /* ClassLab12_Klui_hpp */
```

Призначення: захищає від багаторазового включення заголовкового файлу. Ці директиви гарантують, що вміст файлу буде включено тільки один раз під час компіляції, запобігаючи дублюванню оголошень і визначень.

Оголошення класу:

```
class ClassLab12_Klui
```

Призначення: оголошує клас `ClassLab12_Klui`, який містить атрибути та методи для роботи з циліндром.

Артефакт 2: приватні атрибути класу

Атрибути:

```
float radius;
```

- Призначення: зберігає радіус циліндра.
- Тип: `float`
- Опис: цей атрибут представляє радіус основи циліндра. Він використовується в розрахунках об'єму та площі основи.

```
float height;
```

- Призначення: зберігає висоту циліндра.
- Тип: `float`
- Опис: цей атрибут представляє висоту циліндра. Він використовується в розрахунках об'єму циліндра.

Артефакт 3: публічні методи класу

Конструктор:

```
ClassLab12_Klui();
```

- Призначення: ініціалізує об'єкт класу `ClassLab12_Klui` з початковими значеннями атрибутів `radius` і `height`.
- Опис: конструктор створює об'єкт класу з початковими значеннями радіуса та висоти, що дорівнюють нулю.

Методи доступу:

```
float getRadius();
```

- Призначення: повертає поточне значення радіуса циліндра.
- Тип: `float`
- Опис: геттер для атрибуту `radius`, що дозволяє отримати його значення.

```
void setRadius(float value);
```

- Призначення: встановлює нове значення радіуса циліндра.
- Тип: `void`, параметр `value` типу `float`
- Опис: сеттер для атрибуту `radius`, що дозволяє змінити його значення.

```
float getHeight();
```

- Призначення: повертає поточне значення висоти циліндра.
- Тип: `float`
- Опис: геттер для атрибуту `height`, що дозволяє отримати його значення.

```
void setHeight(float value);
```

- Призначення: встановлює нове значення висоти циліндра.
- Тип: `void`, параметр `value` типу `float`
- Опис: сеттер для атрибуту `height`, що дозволяє змінити його значення.

Методи розрахунків:

```
float calculateVolume();
```

- Призначення: обчислює об’єм циліндра.
- Тип: `float`
- Формула: $V = \pi * r^2 * h$
- Опис: цей метод використовує значення атрибутів `radius` і `height` для обчислення об'єму циліндра за стандартною формулою об'єму.

```
float calculateArea();
```

- Призначення: обчислює площу основи циліндра.
- Тип: `float`
- Формула: $S_o = V / h$, де V - об'єм, h - висота
- Опис: цей метод обчислює площу основи циліндра через об'єм і висоту.

Лістінг проєкту Teacher

```
#include <iostream>
#include <chrono>
#include <thread>
#include <filesystem>
#include <vector>
#include <fstream>
#include <cmath>
#include <limits>

#include "ModulesKlui.h"

#define BEEP_COUNT 100
#define BEEP_REPEAT_DELAY 500
const std::vector<std::filesystem::path> expectedSourceFilePath {"lab12", "prj",
"main.cpp"};
const auto sourceFilePath = std::filesystem::path(__FILE__);
const auto testResultsFilePath = sourceFilePath.parent_path().parent_path() /
"TestSuite" / "TestResult.txt";
#define ERROR_MESSAGE "Встановлені вимоги порядку виконання лабораторної роботи
порушено!"

void playBeepSounds(unsigned int count,
                    unsigned int repeatDelay = BEEP_REPEAT_DELAY) {
    for(auto n = 0; n < count; n++) {
        std::cout << "beep " << n + 1 << '\a' << std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(repeatDelay));
    }
}

bool checkRequirements() {
    auto sourceFilePath = std::filesystem::path(__FILE__);
    auto pathElements = std::vector(sourceFilePath.begin(), sourceFilePath.end());
    if(pathElements.size() >= 3) {
        pathElements.erase(pathElements.begin(), pathElements.end() - 3);
    }

    return pathElements == expectedSourceFilePath;
}
```

```
auto openResultFile() {
    auto stream = std::ofstream(testResultsFilePath);
    if(!stream.is_open()) {
        std::cerr << "cannot open " << testResultsFilePath.filename() <<
std::endl;
        exit(-1);
    }
    return stream;
}

bool doUnitTestCase(float radius,
                    float height,
                    float expectedVolume) {
    auto object = new ClassLab12_Klui();

    object->setRadius(radius);
    object->setHeight(height);

    auto actualVolume = object->calculateVolume();
    delete object;

    return std::fabs(actualVolume - expectedVolume) <
std::numeric_limits<float>::epsilon();
}

void doUnitTestCase(std::ostream &stream,
                    unsigned int number,
                    float radius,
                    float height,
                    float expectedVolume) {
    stream << "Test Case " << number << ": ";
    if(doUnitTestCase(radius, height, expectedVolume)) {
        stream << "passed";
    } else {
        stream << "failed";
    }
    stream << std::endl;
}

void doUnitTesting() {
    auto stream = openResultFile();

    doUnitTestCase(stream, 1, 3.0f, 10.0f,  282.743347f);
    doUnitTestCase(stream, 2, 5.0f, 7.0f, 549.778687f);
    doUnitTestCase(stream, 3, 2.0f, 4.0f, 50.2654839f);
    doUnitTestCase(stream, 4, 4.0f, 8.0f, 402.123871f);
    doUnitTestCase(stream, 5, 6.0f, 12.0f, 1357.16797f);
    doUnitTestCase(stream, 6, 10.0f, 15.0f, 4712.38916f);
    doUnitTestCase(stream, 7, 9.0f, 5.0f, 1272.34497f);
    doUnitTestCase(stream, 8, 2.0f, 15.0f, 188.49556f);
    doUnitTestCase(stream, 9, 1.0f, 10.0f, 31.415926f);
    doUnitTestCase(stream, 10, 3.0f, 2.0f, 56.5486679f);

    stream.close();
}

void handleFailedRequirements() {
    auto stream = openResultFile();

    stream << ERROR_MESSAGE;
    stream.close();
}

int main() {
   // playBeepSounds(BEEP_COUNT);
    if(checkRequirements()) {
```

```
        doUnitTesting();
    } else {
        handleFailedRequirements();
    }

    return 0;
}
```

Артефакт 1: підключення бібліотек та визначення констант

Бібліотеки:

```
#include <iostream>
#include <chrono>
#include <thread>
#include <filesystem>
#include <vector>
#include <fstream>
#include <cmath>
#include <limits>
#include "ModulesKlui.h"
```

Призначення: ці директиви підключають стандартні бібліотеки C++ для роботи з потоками вводу-виводу, обчисленнями, файлами, векторними структурами даних, математичними операціями та іншими функціями, а також користувацьку бібліотеку `ModulesKlui.h`.

Константи:

```
#define BEEP_COUNT 100

#define BEEP_REPEAT_DELAY 500

#define ERROR_MESSAGE "Встановлені вимоги порядку виконання лабораторної роботи порушено!"

const std::vector<std::filesystem::path> expectedSourceFilePath {"lab12", "prj", "main.cpp"};

const auto sourceFilePath = std::filesystem::path(FILE);

onst auto testResultsFilePath = sourceFilePath.parent_path().parent_path() / "TestSuite" / "TestResult.txt";
```

Призначення: визначаються константи для кількості звукових сигналів, затримки між сигналами, повідомлення про помилку та очікуваний шлях до файлу. Також визначаються змінні для збереження шляху до вихідного файлу та файлу з результатами тестування.

Артефакт 2: функція відтворення звукових сигналів

Функція:

```
void playBeepSounds(unsigned int count, unsigned int repeatDelay = BEEP_REPEAT_DELAY)
```

- Призначення: відтворює звукові сигнали задану кількість разів із затримкою між сигналами.

- Тип: `void`
- Параметри:

  - `count`: кількість сигналів.

  - `repeatDelay`: затримка між сигналами в мілісекундах (за замовчуванням BEEP_REPEAT_DELAY).

- Опис: ця функція використовує цикл для відтворення звукових сигналів і робить паузи між сигналами за допомогою `std::this_thread::sleep_for`.

Артефакт 3: функція перевірки вимог

Функція:

```
bool checkRequirements()
```

- Призначення: перевіряє, чи відповідає шлях до вихідного файлу очікуваному.
- Тип: `bool`
- Опис: ця функція розбиває шлях до файлу на елементи та порівнює їх з очікуваними елементами шляху, щоб перевірити, чи виконуються вимоги.

Артефакт 4: функція відкриття файлу результатів

Функція:

```
auto openResultFile()
```

- Призначення: Відкриває файл для запису результатів тестування.
- Тип: `std::ofstream`
- Опис: ця функція намагається відкрити файл для запису і виводить повідомлення про помилку, якщо файл не вдалося відкрити, після чого завершує виконання програми.

Артефакт 5: функція тестування одиничного випадку

Функція:

```
bool doUnitTestCase(float radius, float height, float expectedVolume)
```

- Призначення: виконує одиничний тестовий випадок для обчислення об'єму циліндра.
- Тип: `bool`
- Параметри:

- `radius`: радіус циліндра.

- `height`: висота циліндра.

- `expectedVolume`: очікуваний об'єм.

- Опис: функція створює об'єкт `ClassLab12_Klui`, встановлює значення радіуса та висоти, обчислює об'єм і порівнює його з очікуваним значенням.

Артефакт 6: Функція запису результату тестування одиничного випадку

Функція:

```
void doUnitTestCase(std::ostream &stream, unsigned int number, float radius, float height, float expectedVolume)
```

- Призначення: виконує одиничний тестовий випадок та записує результат у потік.

- Тип: void

- Параметри:

- `stream`: потік для запису результатів.

- `number`: номер тестового випадку.

- `radius`: радіус циліндра.

- `height`: висота циліндра.

- `expectedVolume`: очікуваний об'єм.

- Опис: Функція викликає `doUnitTestCase`, записує номер тесту та результат (пройдено/не пройдено) у потік.

Артефакт 7: функція виконання всіх тестових випадків

Функція:

```
void doUnitTesting()
```

- Призначення: виконує всі тестові випадки та записує результати у файл.

- Тип: `void`

- Опис: функція відкриває файл результатів, виконує всі тестові випадки, записує результати та закриває файл.

Артефакт 8: функція обробки невиконання вимог

Функція:

```
void handleFailedRequirements()
```

- Призначення: записує повідомлення про помилку у файл результатів, якщо вимоги не виконано.
- Тип: `void`
- Опис: функція відкриває файл результатів, записує повідомлення про помилку та закриває файл.

Артефакт 9: головна функція

Функція:

```
int main()
```

Призначення: головна функція програми.

Тип: `int`

Опис: головна функція перевіряє вимоги, виконує тестування або обробляє невиконання вимог, та повертає результат завершення програми.

ВИСНОВОК

Під час виконання лабораторної роботи були виконані наступні етапи:

1) Вивчення теоретичних аспектів:

- Об’єктно-орієнтоване програмування (ООП) та його принципи: інкапсуляція, наслідування, поліморфізм та абстракція;
- геометричні властивості циліндра та формули для обчислення його об'єму та площі основи.

2) Реалізація класу `ClassLab12_Klui`:

- визначення структури класу, оголошення атрибутів та методів у заголовковому файлі (`ClassLab12_Klui.hpp`);
- реалізація методів класу у відповідному файлі реалізації (`ClassLab12_Klui.cpp`).

3) Інтерфейси класу:

- сттворення конструктора, який ініціалізує атрибути класу;
- реалізація геттерів для доступу до значень атрибутів `radius` і `height`;
- реалізація сеттерів для зміни значень атрибутів з валідацією вхідних даних;
- реалізація методу `calculateVolume`, який обчислює об'єм циліндра за заданою формулою.

4) Автоматизоване тестування:

- написання функцій для перевірки коректності роботи класу, включаючи функції для запуску тестів та запису результатів у файл;
- перевірка коректності роботи класу шляхом виконання тестів з різними наборами даних.

Аналіз ходу виконання та отриманих результатів

Під час реалізації класу `ClassLab12_Klui` було здійснено кілька важливих кроків для забезпечення його коректної роботи та надійності:

1) Інкапсуляція та безпека даних:

- всі атрибути класу (`radius` і `height`) були оголошені приватними, що забезпечило захист від неконтрольованих змін ззовні класу;
- методи доступу (геттери та сеттери) були реалізовані як публічні, що дозволило безпечно змінювати та отримувати значення атрибутів з валідацією вхідних даних.

2) Валідація вхідних даних: в методах сеттерів була реалізована перевірка на допустимість вхідних значень, що запобігло встановленню некоректних даних (наприклад, негативних значень радіуса або висоти).

3) Автоматизоване тестування:

- для перевірки роботи класу були розроблені функції для виконання автоматизованих тестів, які забезпечили перевірку коректності роботи методів класу з різними наборами даних;
- результати тестів записувалися у файл, що дозволило легко аналізувати їх та виявляти можливі помилки.

На основі виконаної лабораторної роботи можна зробити наступні висновки:

1) Коректність реалізації класу:

- Клас `ClassLab12_Klui` був успішно реалізований з використанням принципів ООП. Всі необхідні методи були створені та працювали коректно;

- Атрибути класу були захищені від неконтрольованих змін за допомогою інкапсуляції та валідації вхідних даних.

2) Обчислення об'єму циліндра: метод `calculateVolume` коректно обчислював об'єм циліндра на основі значень атрибутів `radius` та `height`. Всі обчислення були точними та відповідали очікуванням.

3) Значення автоматизованого тестування:

- Автоматизоване тестування дозволило швидко і ефективно перевірити роботу класу з різними наборами даних. Це значно підвищило надійність та коректність реалізації;

- Результати тестування, записані у файл, дозволили легко аналізувати результати та вносити необхідні корективи.

4) Практична значущість: реалізація даного класу може бути використана у різних програмах, де необхідно працювати з геометричними фігурами, зокрема з циліндрами. Наприклад, у програмах для моделювання фізичних процесів, інженерних розрахунках та інших додатках.

5) Підвищення навичок програмування: виконання лабораторної роботи сприяло покращенню навичок програмування, зокрема в області ООП. Була отримана практика створення класів, роботи з атрибутами та методами, а також автоматизованого тестування.

Отже, виконання даної лабораторної роботи дозволило не тільки реалізувати конкретний клас для роботи з геометричною фігурою, але й здобути цінний досвід в області об'єктно-орієнтованого програмування, тестування програмного забезпечення та управління проектами. Отримані знання та навички можуть бути успішно застосовані в подальших проектах та дослідженнях, що сприятиме подальшому професійному розвитку.

ВІДПОВІДЬ НА ЗАПИТАННЯ І ЗАВДАННЯ ДЛЯ САМОКОНТРОЛЮ ПІДГОТОВЛЕНОСТІ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

1. Абстрагування є одним з основних методів теоретичного пізнання, що дозволяє упорядковувати складну реальність, спрощуючи та узагальнюючи її для кращого розуміння та аналізу. Абстрагування полягає в виділенні суттєвих характеристик явищ або об'єктів, ігноруванні несуттєвих деталей та створенні абстрактних моделей, які відображають важливі аспекти досліджуваних явищ. На основі описаного процесу, можна сформулювати визначення поняття «абстракція». Це метод спрощення та узагальнення складних реальних явищ або об'єктів шляхом виділення їх суттєвих характеристик і відкидання несуттєвих деталей для створення узагальнених моделей та концепцій, що полегшують розуміння, аналіз та вирішення задач на основі емпіричних даних і фактів.

2. Об’єктна декомпозиція задачі є ключовим методом аналізу та проєктування в об’єктно-орієнтованому програмуванні (ООП). Цей метод полягає у розбитті складної задачі на менші, більш керовані компоненти, які відображають реальні або концептуальні об'єкти проблемної області. Під час об’єктної декомпозиції задачі визначають і описують такі аспекти:

1) сутності (об’єкти): виявляють основні об'єкти, які мають важливе значення для вирішення задачі. Кожен об'єкт представляє певний елемент системи або процесу;

2) атрибути об'єктів: визначають властивості або характеристики об'єктів, які необхідні для їх опису та взаємодії. Атрибути описують стан об’єкта;

3) методи (операції): визначають функції або операції, які можуть виконувати об'єкти. Методи визначають поведінку об'єкта та способи взаємодії з іншими об’єктами;

4) зв'язки між об'єктами: описують взаємозв'язки між об'єктами, включаючи асоціації, агрегації та композиції. Зв'язки показують, як об'єкти взаємодіють і залежні один від одного;

5) ієрархії об'єктів: визначають ієрархічні структури, такі як наслідування, де одні об'єкти успадковують властивості та поведінку інших. Це допомагає організувати об'єкти за рівнями абстракції;

6) інтерфейси об'єктів: описують публічні методи та властивості, через які зовнішні об'єкти можуть взаємодіяти з даним об'єктом. Інтерфейси визначають контракти для взаємодії між об’єктами;

7) модулі та компоненти: розподіляють об'єкти на логічні модулі або компоненти, що полегшує управління складністю системи та її підтримку.

3. Концептуалізація предметної області - це процес визнання, структурування та систематизації знань про об’єкт дослідження.

Результати концептуалізації предметної області як метода пізнання (дослідження):

1) структурована інформація про предметну область, яка допомагає краще розуміти її сутність та взаємозв’язки;

2) візуалізація, що відображає основні компоненти та їхні взаємозв’язки, спрощуючи розуміння складних систем;

3) чітко визначені концепції та моделі сприяють більш цілеспрямованому та структурованому підходу до подальших досліджень;

4) визначення зв’язків між об’єктами;

5) реалізація програмного забезпечення відповідно до розроблених моделей та архітектури;

6) створення технічної документації, включаючи документацію для користувачів та розробників.

4. Аналіз сутності предметної області – це процес вивчення та моделювання основних об'єктів, їх властивостей і взаємозв'язків у предметній області для створення програмного забезпечення.

Структура або алгоритм об'єктного аналізу предметної області задачі в програмуванні включає кілька ключових кроків: 1) Визначення вимог; 2) Ідентифікація сутностей; 3) Визначення атрибутів; 4) Визначення зв'язків між

сутностями; 5) Визначення поведінки сутностей; 6) Вивчення життєвого циклу сутностей; 7) Визначення обмежень і правил валідації; 8) Моделювання та прототипування; 9) Документування результатів

5. Для створення класу, який реалізує абстрактний тип даних у заголовковому файлі C++, можна дотримуватися наступної методики: 1) Визначити ім'я класу; 2) Оголосити захищені і приватні дані-члени класу; 3) Оголосити публічні методи класу; 3) Забезпечте валідацію даних; 4) Додати конструктори і деструктор; 5) Додати додаткові функції-члени.

6. Результатом реалізації процесу визначення інтерфейсу об'єкта предметної області є чітке визначення методів (функцій-членів) і властивостей (атрибутів), які будуть доступні для взаємодії з об'єктами цього класу. Це включає: список публічних методів і атрибутів; декларації методів і атрибутів.

7. Мета перевантаження функцій у мовах програмування C/C++ полягає в тому, щоб забезпечити можливість визначення кількох функцій з однаковим іменем, але з різними параметрами. Це полегшує створення зручного інтерфейсу для користувача, підтримує різні типи даних і операції, а також сприяє використанню одного імені для різних функцій з подібною поведінкою.

8. У мовах програмування C/C++, функції можуть бути перевантажені шляхом визначення кількох функцій з однаковим іменем, але з різними параметрами. Синтаксис запису перевантаження функцій виглядає наступним чином:

```
// Перевантажені функції з однаковим ім'ям
// Аргументи функцій повинні відрізнятися за типами або кількістю

// Приклад перевантаження функції add для додавання двох чисел різних типів
int add(int a, int b) {
    return a + b;
}

double add(double a, double b) {
    return a + b;
}

// Приклад перевантаження функції display для виведення різних типів даних
void display(int num) {
```

```
    std::cout << "Integer: " << num << std::endl;
}

void display(double num) {
    std::cout << "Double: " << num << std::endl;
}

// Приклад перевантаження функції calculate для обчислення площі різних геометричних фігур
double calculate(double radius) {
    return 3.14159 * radius * radius; // Площа кола
}

double calculate(double length, double width) {
    return length * width; // Площа прямокутника
}
```

У цьому прикладі:

- функція `add` перевантажена для різних типів даних (цілих чисел і дійсних чисел).

- функція `display` перевантажена для виведення різних типів даних (цілих чисел і дійсних чисел).

- функція `calculate` перевантажена для обчислення площі різних геометричних фігур (кола і прямокутника).

9. Препроцесорні макроси `__FILE__`, `__LINE__`, `__DATE__`, `__TIME__` та `__TIMESTAMP__` у мовах програмування C/C++ зазвичай використовуються для різних цілей, зокрема:

1) Відлагодження програми: додавання `__FILE__` і `__LINE__` до друку повідомлень про помилки або відлагодження допомагає знаходити місце в коді, де сталася помилка або виникла певна подія;

2) Додавання інформації про версію програми: використання `__DATE__` і `__TIME__` дозволяє включати інформацію про дату та час компіляції програми в вихідний код. Це корисно для відслідковування версій програми або логування подій;

3) Створення унікальних ідентифікаторів: можна використовувати `__TIMESTAMP__` для генерації унікальних ідентифікаторів, наприклад, для автоматичного створення імен файлів або логів з унікальними назвами;

4) Додавання додаткової інформації в логи: макроси можна використовувати для додавання додаткової інформації (наприклад, імені файлу та рядка коду) до логів або повідомлень про помилки;

5) Генерація внутрішнього коду: макроси можуть бути використані для генерації внутрішнього коду, наприклад, автоматичної генерації логів або інших додаткових засобів відлагодження.

10. У мові програмування C++ функції з параметрами за замовчуванням оголошуються наступним чином:

```
// Синтаксис оголошення функції з параметрами за замовчуванням
return_type function_name(parameter_type1 parameter1 = default_value1,
                          parameter_type2 parameter2 = default_value2,
                          …);
```

11. Препроцесорні директиви в мовах програмування C/C++ використовуються для визначення умовного компіляційного коду, макросів та налаштувань компіляції. Основні призначення кожної з них наступні:

`#define`: визначає макрос, який може використовуватися для заміни тексту у програмному коді. Наприклад: `#define PI 3.14159`

`#undef`: скасовує визначення макросу, визначеного раніше за допомогою `#define`.

`#if`: починає умовний блок коду, який компілюється, якщо вираз, який слідує за `#if`, видає ненульове значення.

`#ifdef`: починає умовний блок коду, який компілюється, якщо вказаний макрос був визначений раніше за допомогою `#define`.

`#ifndef`: починає умовний блок коду, який компілюється, якщо вказаний макрос не був визначений раніше за допомогою `#define`.

`#error`: викликає помилку компіляції та виводить повідомлення, що міститься у директиві `#error`.

`#pragma`: використовується для встановлення деяких параметрів компіляції, таких як параметри оптимізації, обробка попереджень або налаштування іншого компілятора. Це нестандартна директива, і її ефекти можуть відрізнятися залежно від компілятора.

12. Концепція абстрактного типу даних (ADT) полягає в тому, що він представляє собою модель даних разом із набором операцій, які можна виконувати над цими даними. Основна ідея полягає в тому, щоб приховати внутрішню реалізацію даних і використовувати лише інтерфейс для взаємодії з ними. ADT)відрізняється від вбудованого типу даних в C/C++ таким чином: 1) ADT надає вищий рівень абстракції, приховуючи деталі реалізації і надаючи інтерфейс для взаємодії з даними; 2) ADT може мати власні методи (функції), що виконують різні операції над даними, тоді як вбудовані типи мають обмежений набір операцій; 3) Розширення функціональності ADT зазвичай є легшим, оскільки можна додавати нові методи або змінювати існуючі, не змінюючи сам тип.

13. Порівняльний аналіз типів `class` та `struct` у мові програмування C++

Спільні риси:

- Визначення нових типів даних: обидва типи створюють нові складні типи даних;

- Інкапсуляція: підтримують приховування деталей реалізації;

- Наслідування: підтримують створення нових типів на основі існуючих;

-Поліморфізм: можуть реалізовувати поліморфізм через однакові інтерфейси.

Відмінності

- Доступ за замовчуванням:

  `struct`: публічні члени за замовчуванням (`public`).

  `class`: приватні члени за замовчуванням (`private`).

- Призначення:

  `struct`: для пасивних об'єктів, що зберігають дані.

  `class`: для об'єктів з активною поведінкою та логікою.

- Наслідування за замовчуванням:

  `struct`: публічне наслідування (`public`).

  `class`: приватне наслідування (`private`).

14. У мові програмування C++, синтаксис запису ADT через клас (`class`) виглядає наступним чином:

```
class ClassName {
    // Оголошення приватних членів класу
private:
    type member1;
    type member2;
    // ...

public:
    // Оголошення публічних методів класу
    void method1();
    void method2();
    // ...
};
```

В цьому прикладі:

- `ClassName` - ім'я класу.
- `member1, member2` - приватні члени класу.
- `method1(), method2()` - публічні методи класу.

Приватні члени і методи доступні тільки у межах класу, тоді як публічні методи можуть бути викликані ззовні класу.

15. Оператор `::` в мові програмування C++ використовується для доступу до глобальних змінних, функцій чи методів класу, а також для визначення простору імен.

Призначення цього оператора:

1) Доступ до глобальних об'єктів та функцій: використовується для доступу до глобальних змінних, функцій або класів, які визначені поза поточним простором імен;

2) Розділення ідентифікаторів у класах: використовується для доступу до членів класу або методів, коли ім'я членів конфліктує з іменами в локальному просторі імен.

Синтаксис:

```
namespace_name::identifier  // Доступ до елементів простору імен
class_name::member_name     // Доступ до членів класу
::global_variable           // Доступ до глобальної змінної
```

Приклад запису даних у глобальний об’єкт з однаковим ім’ям:

```
#include <iostream>
```

```
int value = 5; // Глобальна змінна value

int main() {
    int value = 10; // Локальна змінна value
    ::value = 20;   // Запис у глобальну змінну value
      std::cout << "Local value: " << value << std::endl; // Виведе: Local value: 20
    std::cout << "Global value: " << ::value << std::endl; // Виведе: Global value: 20
    return 0;
}
```

У цьому прикладі `::value` вказує на глобальну змінну `value`, в той час як просто `value` вказує на локальну змінну `value`, оголошену всередині функції `main()`.

16. Оголошення членів класу та структури у мові програмування C++ має схожий синтаксис, але вони відрізняються використанням модифікаторів доступу за замовчуванням та призначенням. Класи використовуються для визначення об'єктів та пов'язаних з ними методів, в той час як структури частіше використовуються для організації даних без пов'язаних з ними методів або як прості контейнери даних.

17. Синтаксис і основні правила запису функцій-членів класу в мові програмування C++ наступні:

1) Оголошення в класі: функції-члени класу оголошуються всередині визначення класу.

2) Модифікатор доступу: функції-члени можуть мати модифікатори доступу `public`, `private`, `protected`, що визначають рівень доступу до них ззовні класу.

3) Параметри і тип повернення: функції-члени можуть мати параметри та тип повернення. Це аналогічно звичайним функціям, але їх визначення відбувається всередині класу.

4) Тіло функції: реалізація функції-члена знаходиться всередині фігурних дужок `{}`. Вона має доступ до приватних та захищених членів класу.

5) Виклик функції-члена: функції-члени класу можуть бути викликані для конкретного об'єкта класу за допомогою оператора крапки (**.**) або через вказівник на об'єкт та оператор `->`.

```
class MyClass {
public:
    // Оголошення функцій-членів класу
```

```
    return_type function_name(parameter_list) {
        // Тіло функції
    }
};
```

18. У мові програмування C++ існують три рівні (секції) доступу до членів класу, і кожен з них визначається використанням відповідних специфікаторів доступу:

`Public` (публічний):

- члени, оголошені з модифікатором `public`, доступні з будь-якого місця програми;

- вони можуть бути використані через об'єкт класу або в його нащадках.

`Private` (приватний):

- члени, оголошені з модифікатором `private`, доступні тільки в межах класу, в якому вони оголошені;

- вони недоступні для зовнішніх функцій або об’єктів.

`Protected` (захищений):

- члени, оголошені з модифікатором `protected`, доступні в межах класу та його нащадків (похідних класів).

- вони недоступні для зовнішніх функцій або об’єктів.

19. Синтаксис оголошення (створення) об’єкта класу в мові програмування C++ виглядає наступним чином: `ClassName objectName;`

Де: `ClassName` - ім'я класу, на основі якого створюється об’єкт.

`objectName` - ім'я створеного об’єкта.

Порівняльний аналіз класу та об’єкта цього класу:

Клас: це шаблон або опис, який визначає структуру та поведінку об’єктів. Містить оголошення даних та методів, що працюють з цими даними.

Об’єкт: це конкретний екземпляр класу, створений за його описом. Має власний стан, що включає в себе значення всіх його членів даних. Може викликати методи класу та працювати з його даними.

20. Спеціальні функції-члени класу в мові програмування C++ мають особливе призначення та синтаксис. Ці функції включають конструктори, деструктори та інші спеціальні методи.

Конструктори є спеціальними методами, які автоматично викликаються при створенні об'єкта класу. Вони ініціалізують початковий стан об’єкта. Синтаксис:

```
ClassName(); // Конструктор за замовчуванням
ClassName(parameters); // Параметризований конструктор
ClassName(const ClassName &obj); // Конструктор копіювання
```

Деструктори викликаються автоматично при знищенні об'єкта класу. Вони звільняють ресурси, які були виділені об’єкту. Синтаксис:

```
~ClassName(); // Деструктор
```

Спеціальні функції-члени класу повинні бути оголошені та визначені всередині визначення класу або підключені через заголовочний файл класу.

ВІДПОВІДЬ НА КОНТРОЛЬНІ ЗАПИТАННЯ І ЗАВДАННЯ

1. Результати виконання концептуалізації предметної області, об’єктного аналізу та визначення інтерфейсів сутностей предметної області в загальному вигляді включають такі етапи:

1) Концептуалізація предметної області:

- *Побудова концептуальної моделі*: розробка абстрактної моделі, яка відображає основні елементи предметної області та їх взаємозв’язки;
- *Визначення основних понять і термінів*: створення глосарія, який включає визначення ключових понять і термінів, що використовуються в предметній області;
- *Виявлення ключових сутностей*: визначення основних сутностей (об'єктів), які мають важливе значення для даної предметної області.

2) Об’єктний аналіз:

- *Визначення сутностей та їх атрибутів*: виявлення та опис сутностей (об'єктів), включаючи їх атрибути (властивості), які характеризують ці сутності;
- *Побудова діаграм класів*: розробка UML-діаграм класів, що відображають структуру сутностей, їх атрибути і взаємозв’язки між ними;

- *Визначення зв’язків між сутностями*: опис типів зв’язків (асоціації, агрегації, композиції) між сутностями, а також їх кардинальностей (один до одного, один до багатьох, багато до багатьох).

3) Визначення інтерфейсів сутностей предметної області:

- *Опис інтерфейсів*: визначення методів і функцій, які сутності повинні реалізовувати для взаємодії з іншими сутностями чи системами.

- *Побудова діаграм послідовності*: розробка UML-діаграм послідовності, що показують порядок виклику методів та обмін повідомленнями між сутностями під час виконання сценаріїв.

- *Специфікація API* (інтерфейсів програмування додатків): документування інтерфейсів з описом методів, їх параметрів, типів повертаючих значень, а також можливих помилок.

2. Процес концептуалізації предметної області, об’єктний аналіз та визначення інтерфейсів сутностей предметної області є взаємопов'язаними етапами розробки програмного забезпечення, які забезпечують систематичний підхід до побудови ефективних та узгоджених моделей і систем. Ось як ці процеси пов'язані між собою:

1) Концептуалізація предметної області:

- *Перший етап*: концептуалізація предметної області є початковим етапом, на якому визначаються основні елементи та поняття, що становлять предметну область. Вона включає в себе створення абстрактної моделі, яка відображає основні сутності та їх взаємозв'язки. На цьому етапі визначаються ключові поняття і терміни, створюється глосарій;

- *Зв'язок з об’єктним аналізом*: концептуальна модель, розроблена на цьому етапі, служить основою для об’єктного аналізу. Вона надає загальне уявлення про структуру та взаємозв’язки сутностей, які потім деталізуються та конкретизуються в процесі об’єктного аналізу.

2) Об’єктний аналіз:

- *Другий етап*: на цьому етапі відбувається деталізація концептуальної моделі. Визначаються конкретні сутності (об'єкти), їх атрибути, взаємозв’язки та

поведінка. Результатом цього етапу є побудова діаграм класів та інших UML-діаграм, що деталізують структуру системи;

- *Зв'язок з визначенням інтерфейсів*: об’єктний аналіз визначає основні сутності та їх взаємодію, що служить основою для визначення інтерфейсів. Визначення атрибутів і методів сутностей дає можливість описати, які інтерфейси потрібні для їх взаємодії.

3) Визначення інтерфейсів:

- *Третій етап*: на цьому етапі розробляються інтерфейси, які визначають, як сутності будуть взаємодіяти між собою та з іншими компонентами системи. Це включає опис методів, параметрів, типів повертаючих значень та можливих помилок.

-*Зв'язок з концептуалізацією та об’єктним аналізом*: визначення інтерфейсів базується на результатах концептуалізації та об’єктного аналізу. Концептуальна модель надає загальне розуміння предметної області, а об’єктний аналіз забезпечує детальну інформацію про сутності та їх атрибути, що дозволяє точно визначити необхідні інтерфейси.

3. Для визначення, чи слід описати абстракцію сутності предметної області мовою C++ типом структура (`struct`) або типом клас (`class`), можна використовувати наступні критерії:

1) Призначення сутності:

- `struct`: якщо сутність призначена для зберігання даних з мінімальною логікою обробки, як, наприклад, просто набір полів, що описують певний об'єкт.

- `class`: якщо сутність включає не тільки дані, але й функціональність, що обробляє ці дані, тобто коли потрібна інкапсуляція даних і методів.

2) Інкапсуляція:

- `struct`: якщо немає потреби приховувати деталі реалізації і всі члени (поля і методи) можуть бути публічними.

- `class`: якщо необхідно приховати деякі члени сутності від зовнішнього доступу, використовуючи приватні або захищені (protected) модифікатори доступу.

3) Наслідування:

- `struct`: якщо сутність не планується використовувати в ієрархії класів або потреба в наслідуванні є мінімальною.

- `class`: якщо сутність є частиною ієрархії класів і потребує можливості наслідування з поліморфізмом.

4) Конструктори та деструктори:

- `struct`: якщо не потрібні спеціальні конструктори, деструктори або функції ініціалізації/знищення.

- `class`: якщо необхідні спеціальні конструктори, деструктори, або є специфічні вимоги до ініціалізації або звільнення ресурсів.

5) Модифікатори доступу за замовчуванням:

- `struct`: у структурі всі члени за замовчуванням є публічними.

- `class`: у класі всі члени за замовчуванням є приватними, що забезпечує кращу інкапсуляцію.

6) Семантика і стиль кодування:

- `struct`: якщо семантика сутності більше відповідає просто структурі даних (як у C), то використовується `struct`.

- `class`: якщо семантика сутності відповідає повноцінному об'єкту з поведінкою, то використовується `class`.

4. Інтерфейс класу в програмуванні - це набір публічних методів і властивостей, які визначають, як зовнішні об'єкти можуть взаємодіяти з об'єктами цього класу. Він описує доступні операції і забезпечує абстракцію, приховуючи реалізаційні деталі класу. Інтерфейс класу включає:

1) публічні методи, що визначають поведінку класу.

2) публічні властивості (атрибути), хоча їх використання обмежене для збереження інкапсуляції.

3) конструктори та деструктори, що визначають спосіб ініціалізації та знищення об'єктів.

4) константні методи, які не змінюють стан об’єкта.

5. Причини, чому в класі C++ не можна оголосити конструктор з закритим рівнем доступу:

1) звичайне використання класу: якщо клас призначений для звичайного використання, де його екземпляри повинні створюватися без обмежень, використання приватного конструктора буде непрактичним. Приватний конструктор обмежує створення об'єктів, що може заважати використанню класу в звичайних сценаріях;

2) використання класу як базового для наслідування: якщо клас планується використовувати як базовий клас для наслідування, приватний конструктор ускладнює або робить неможливим створення екземплярів підкласів. Підкласи не зможуть викликати конструктор базового класу, якщо він приватний;

3) збільшення зв'язаності та ускладнення дизайну: використання приватного конструктора може вимагати визначення дружніх класів або функцій, що ускладнює дизайн і підвищує зв'язаність між компонентами, що не завжди бажано;

4) необхідність створення об'єктів за замовчуванням: деякі сценарії вимагають створення об'єктів за замовчуванням. Якщо конструктор є приватним, це не дозволить створювати такі об'єкти в контекстах, де це необхідно (наприклад, в контейнерах STL, які потребують конструктор за замовчуванням).

6. Перевантаження функцій і функції з параметрами за замовчуванням - це два різні способи реалізації поліморфізму в мові C++, які дозволяють використовувати одну і ту ж функцію для різних наборів параметрів. Порівняльний аналіз цих двох підходів:

Критерій	Перевантаження функцій	Функції з параметрами за замовчуванням
Гнучкість	висока, різні реалізації для різних типів/кількостей параметрів	менша, одна функція для всіх наборів параметрів
Кількість коду	більше коду, більше функцій	менше коду, одна функція з параметрами за замовчуванням
Зрозумілість	чітко визначені окремі функції	простий виклик, але складніше обробляти різні сценарії

Управління змінами	легше оновлювати окремі функції	зміни параметрів за замовчуванням можуть впливати на всі виклики
Складність підтримки	вищий рівень складності через велику кількість функцій	легша підтримка, одна функція для всіх випадків
Приховані дефолтні значення	немає прихованих значень	можуть бути неочікувані наслідки при змінах

7. В C++ доступ до відкритих членів об'єктів класу здійснюється за допомогою таких операторів:

- Оператор . (крапка): використовується для доступу до членів об'єкта за іменем;

- Оператор -> (стрілка): використовується для доступу до членів об'єкта через вказівник на об’єкт.

8. Клас C++ як абстрактний тип даних (ADT) дозволяє реалізувати принцип інкапсуляції наступними способами:

- Приховування даних: використання модифікаторів доступу (private, protected) для обмеження доступу до внутрішніх даних класу, щоб зовнішні об'єкти не могли безпосередньо змінювати ці дані;

- Публічний інтерфейс: визначення публічних методів (public), через які здійснюється взаємодія з об'єктами класу, надаючи контрольований доступ до внутрішнього стану об’єкта;

- Контроль доступу: застосування методів для перевірки та управління доступом до даних, що забезпечує узгодженість і цілісність стану об’єкта;

- Інкапсуляція логіки: зосередження всієї логіки обробки даних всередині класу, що дозволяє змінювати реалізацію без впливу на зовнішній код, який використовує цей клас.