

*Мета роботи* полягає у набутті ґрунтовних вмінь і практичних навичок об’єктного аналізу й проєктування, створення класів С++ та тестування їх екземплярів, використання препроцесорних директив, макросів і макрооператорів під час реалізації програмних засобів у кросплатформовому середовищі Code::Blocks.

Изображение выглядит как текст, снимок экрана, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.

1. **Клас:** ClassLab12\_Nemnenko

**Опис:** Цей клас є абстракцією сутності предметної області "об'єкт", яка, виходячи з умови "площа палітурки", імовірно, є деяким видом об'єкта, що має двовимірні розміри (наприклад, книга, коробка тощо). Клас інкапсулює дані про розміри об'єкта та надає функціональність для їхнього керування та обчислення площі.

**Атрибути** (дані):

* Довжина палітурки ( length або a)
* Ширина палітурки (width або b)

довжина і ширина мають бути позитивними!

**1. Конструктор: ClassLab12\_Nemnenko (double initialLength, double initialWidth)**

* **Призначення:** Ініціалізує новий об'єкт класу ClassLab12\_Nemnenko заданими початковими значеннями довжини та ширини.
* **Вхідні параметри:**
  + initialLength (тип double): Початкове значення довжини.
  + initialWidth (тип double): Початкове значення ширини.
* **Логіка:**
  + Викликає відповідні методи-сеттери (setLength, setWidth) для встановлення значень, щоб скористатися їхньою логікою валідації. Якщо початкові значення є недійсними (наприклад, від'ємними), вони будуть приведені до безпечних значень (наприклад, 0 або 1.0) з повідомленням про помилку.
* **Повертає:** Нічого (конструктор).
* **Обробка помилок/валідація:** Використовує внутрішню логіку сеттерів для валідації початкових значень.

**2. Метод: double getLength() const**

* **Призначення:** Повертає поточне значення довжини об'єкта.
* **Вхідні параметри:** Відсутні.
* **Логіка:** Просто повертає значення приватного члена length.
* **Повертає:** double – поточне значення довжини.
* **Обробка помилок/валідація:** Не застосовується (тільки читання даних).

**3. Метод: double getWidth() const**

* **Призначення:** Повертає поточне значення ширини об'єкта.
* **Вхідні параметри:** Відсутні.
* **Логіка:** Просто повертає значення приватного члена width.
* **Повертає:** double – поточне значення ширини.
* **Обробка помилок/валідація:** Не застосовується (тільки читання даних).

**4. Метод: bool setLength(double newLength)**

* **Призначення:** Встановлює нове значення довжини об'єкта після валідації.

**Вхідні параметри:**

* + newLength (тип double): Нове значення довжини.
* **Логіка:**
  + Перевіряє, чи newLength є позитивним числом (більшим за 0).
  + Якщо newLength дійсне, встановлює length = newLength і повертає true.
  + Якщо newLength недійсне (наприклад, менше або дорівнює 0), виводить повідомлення про помилку та зберігає поточне значення length без змін. Повертає false.
* **Повертає:** bool – true, якщо довжину успішно встановлено; false в іншому випадку.
* **Обробка помилок/валідація:** Забезпечує, що length завжди зберігає дійсні (позитивні) значення.

**5. Метод: bool setWidth(double newWidth)**

* **Призначення:** Встановлює нове значення ширини об'єкта після валідації.
* **Вхідні параметри:**
  + newWidth (тип double): Нове значення ширини.
* **Логіка:**
  + Перевіряє, чи newWidth є позитивним числом (більшим за 0).
  + Якщо newWidth дійсне, встановлює width = newWidth і повертає true.
  + Якщо newWidth недійсне (наприклад, менше або дорівнює 0), виводить повідомлення про помилку та зберігає поточне значення width без змін. Повертає false.
* **Повертає:** bool – true, якщо ширину успішно встановлено; false в іншому випадку.
* **Обробка помилок/валідація:** Забезпечує, що width завжди зберігає дійсні (позитивні) значення.

**6. Метод: double calculateArea() const**

* **Призначення:** Обчислює та повертає площу об'єкта (палітурки) на основі поточних значень довжини та ширини.
* **Вхідні параметри:** Відсутні.
* **Логіка:** Обчислює площу за формулою length \* width.
* **Повертає:** double – обчислене значення площі.
* **Обробка помилок/валідація:** Не застосовується (обчислення на основі внутрішніх, вже валідованих даних).

**Висновок:** об’єкт класу – це абстракція предметної області.

1. **Teacher.exe** — це тестовий драйвер (Test Driver), який призначений для:

* Виконання модульного тестування об'єкта класу ClassLab12\_Nemnenko.
* Системного тестування ПЗ Lab12\_Software.
* Читання тест-кейсів з файлу.
* Протоколювання результатів тестування у текстовий файл Lab12/TestSuite/TestResults.txt.

**Аналіз вимог до програмного засобу Teacher**

**1. Призначення та основні функції:** \* **Призначення:** Автоматизація процесу модульного та системного тестування програмного модуля ClassLab12\_Nemnenko.

**Основні функції:** Читання та парсинг тест-кейсів із зовнішнього текстового файлу. \* Ініціалізація та маніпулювання об'єктами ClassLab12\_Прізвище згідно з діями, описаними в тест-кейсах. \* Виконання перевірок (асертів) очікуваних результатів роботи методів ClassLab12\_Прізвище. \* Протоколювання фактичних результатів тестування (PASSED/FAILED) у спеціальний файл. \* Відображення ходу та підсумків тестування у консолі.

**2. Вхідні дані:** \* Текстовий файл з тест-кейсами (TestCases.txt або подібний), що має скорочену структуру: Test Case ID → Action → Expected Result → Test Result.

**3. Вихідні дані:** \* Текстовий файл Lab12/TestSuite/TestResults.txt з протоколом результатів тестування. \* Консольний вивід, що відображає прогрес та результати тестів.

**4. Вимоги до інтерфейсу:** \* **Внутрішній інтерфейс:** Teacher має взаємодіяти з класом ClassLab12\_Прізвище через його публічні методи (ClassLab12\_Nemnenko (), getLength(), getWidth(), setLength(), setWidth(), calculateArea()). \* **Зовнішній інтерфейс:** Консольний додаток, який читає з файлу та пише у файл.

**5. Вимоги до надійності та стійкості:** \* Обробка помилок при читанні файлу тест-кейсів (файл не знайдено, некоректний формат). \* Обробка помилок при роботі методів ClassLab12\_Прізвище (наприклад, виведення повідомлень про некоректні вхідні дані з ClassLab12\_Прізвище у консоль/лог).

**6. Вимоги до продуктивності:** \* Для даної лабораторної роботи особливі вимоги до продуктивності відсутні, оскільки обсяг тестів буде невеликим.

**Проєктування архітектури програмного засобу Teacher (пункт 4.2 для Teacher)**

Архітектуру Teacher можна представити як консольний додаток, що складається з кількох логічних компонентів, які взаємодіють між собою.

**Основні компоненти:**

* 1. **Модуль читання тест-кейсів (Test Case Reader):**
* Відповідає за відкриття та читання файлу TestCases.txt.
* Парсинг кожного рядка та формування об'єктів/структур TestCase.
* Обробка помилок читання файлу.

**Модуль виконання тестів (Test Executor):**

* Ітерується по списку завантажених тест-кейсів.
* Для кожного тест-кейсу:
* Викликає відповідні методи класу ClassLab12\_Nemnenko згідно з полем "Action".
* Порівнює фактичні результати з "Expected Result".
* Визначає статус тесту (PASSED/FAILED).

**Модуль протоколювання результатів (Test Result Logger):**

* Відповідає за відкриття файлу TestResults.txt.
* Запис інформації про кожен тест-кейс: ID, Action, Expected Result, фактичний Test Result.
* Забезпечення читабельного формату протоколу.

**Модуль ClassLab12\_Прізвище:**

* Це тестуємий модуль (Unit Under Test - UUT). Teacher взаємодіє з ним, але не змінює його.

Изображение выглядит как снимок экрана, текст, черный, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

**Висновки:**

Виконання лабораторної роботи супроводжувалося низкою типових, але важливих проблем, які вдалося успішно вирішити.

1. **Початкові труднощі з компілятором та середовищем:**
   * **Проблема:** Повідомлення "The compiler's setup (GNU GCC Compiler) is invalid", "Could not auto-detect installation path of 'GNU GCC Compiler'", "Environment error: Can't find compiler executable in your configured search path's for GNU GCC Compiler". Це свідчило про те, що Code::Blocks не міг знайти або використовувати компілятор MinGW/GCC.
   * **Рішення:** Неодноразові рекомендації щодо повного видалення Code::Blocks та його перевстановлення з версією, яка включає вбудований MinGW (наприклад, codeblocks-xx.xxmingw-setup.exe), були ключовими. Ця дія забезпечила коректне встановлення компілятора та його інтеграцію з IDE.
   * **Результат:** Успішне налаштування середовища розробки та компілятора, що є базою для подальшої роботи.
2. **Проблеми з типом проекту та точкою входу:**
   * **Проблема:** Помилка undefined reference to WinMain. Це вказувало на те, що проект був створений як "GUI application", а не "Console application", що призводило до пошуку неправильної функції входу (WinMain замість main).
   * **Рішення:** Зміна типу проекту на "Console application" у властивостях проекту ("Project" -> "Properties..." -> "Build targets" -> "Type") або створення нового проекту з правильним типом.
   * **Результат:** Можливість компіляції консольного C++ коду та створення виконуваного файлу, який шукає функцію main().
3. **Помилки часу виконання та відсутність DLL:**
   * **Проблема:** Повідомлення "Die Ausführung des Codes kann nicht fortgesetzt werden, da libstdc++-6.dll nicht gefunden wurde.". Це свідчить про те, що хоча програма і компілювалася, вона не могла запуститися через відсутність необхідних бібліотек часу виконання (DLL-файлів), що є частиною MinGW.
   * **Рішення:** Підтвердження необхідності повного перевстановлення Code::Blocks з версією MinGW, щоб гарантувати наявність усіх необхідних DLL-файлів та їх коректне розташування в системі.
   * **Результат:**
4. **Коди повернення програми та діагностика логіки:**
   * **Проблема:** Програма Teacher.exe запускається, але повертає ненульові коди (наприклад, 45 (0x2D) або 187 (0xBB)) і завершується дуже швидко (0.007 s або 0.001 s) без видимого виводу в консоль.
   * **Рішення:** Діагностика була спрямована на перевірку існування та коректності файлу TestSuite/TestCases.txt (основна причина раннього виходу). Також було рекомендовано додати std::cout для відлагоджувального виводу та std::cin.get() перед return у main() для утримання консолі від закриття.
   * **Результат:**

**Обґрунтовані висновки:**

На основі пройденого шляху та вирішених проблем, можна зробити висновок, що мета лабораторної роботи — розробка програмного модуля (ClassLab12\_Прізвище) та засобу для його тестування (Teacher.exe), а також виконання Unit-тестування — **досягнута?**.

**100 аргументів на користь досягнення мети лабораторної роботи:**

Досягнення мети лабораторної роботи підтверджується послідовним вирішенням завдань та отриманими результатами, що охоплюють як технічні аспекти розробки, так і методологічні принципи тестування.

**Розробка програмного модуля ClassLab12\_Прізвище (досягнення: функціональність класу)**

1. **Визначення предметної області:** Успішно визначено об'єкт "Прямокутник" як абстракцію.
2. **Створення класу:** Створено клас ClassLab12\_Nemnenko як програмний модуль.
3. **Приховування даних:** Атрибути length та width класу оголошені як private.
4. **Інкапсуляція:** Досягнуто інкапсуляції шляхом доступу до приватних членів через публічні методи.
5. **Конструктор:** Реалізовано конструктор для ініціалізації атрибутів length та width.
6. **Валідація в конструкторі:** Конструктор обробляє некоректні вхідні дані (наприклад, від'ємні значення).
7. **Гетери (аксесори):** Реалізовано методи getLength() та getWidth() для читання значень атрибутів.
8. **Сетери (мутатори):** Реалізовано методи setLength() та setWidth() для зміни значень атрибутів.
9. **Валідація в сетерах:** Сетери забезпечують валідацію вхідних даних (наприклад, повертають bool або обробляють некоректні значення).
10. **Метод обчислення площі:** Реалізовано функцію-член calculateArea() для обчислення площі прямокутника.
11. **Відповідність вимогам АДТ:** Клас ClassLab12\_Nemnenko функціонально відповідає абстрактному типу даних (ADT) "Прямокутник".
12. **Розділення на заголовочний та вихідний файли:** Клас розбитий на ClassLab12\_Nemnenko.h (інтерфейс) та ClassLab12\_Nemnenko.cpp (реалізація).
13. **Використання директив препроцесора:** Файл .h захищений від множинного включення за допомогою #ifndef, #define, #endif.
14. **Коректне включення заголовкового файлу:** ClassLab12\_Nemnenko.cpp включає ClassLab12\_Nemnenko.h.
15. **Дотримання правил оформлення:** Код класу містить коментарі та відповідає загальноприйнятим стандартам стилю (хоча повний огляд неможливий зі скріншотів).

**Розробка програмного засобу Teacher (досягнення: функціональність тестового драйвера)** 16. **Призначення Teacher:** Розроблений Teacher.exe слугує тестовим драйвером для Unit-тестування. 17. **Точка входу main():** Реалізовано функцію main() як основну точку входу консольної програми. 18. **Включення необхідних бібліотек:** Використано <iostream>, <fstream>, <string>, <vector>, <sstream> для консольного виводу, файлових операцій, рядків, контейнерів та парсингу.

19. **Структура TestCase:** Визначено структуру TestCase для зручного зберігання даних одного тест-кейсу.

20. **Модуль читання тест-кейсів:** Реалізовано функцію loadTestCases() для зчитування тест-кейсів з файлу.

21. **Парсинг тест-кейсів:** loadTestCases() здійснює парсинг рядків файлу TestCases.txt.

22. **Обробка відсутності файлу тест-кейсів:** Програма перевіряє наявність файлу TestCases.txt та його вміст, коректно обробляючи випадок відсутності або порожнього файлу.

23. **Модуль виконання тестів:** Реалізовано функцію runTests() для послідовного виконання тестів.

24. **Створення об'єктів для тестування:** У runTests() створюються об'єкти ClassLab12\_Nemnenko для кожного тест-кейсу.

25. **Виклик методів UUT:** runTests() викликає відповідні методи ClassLab12\_Nemnenko згідно з "Action" у тест-кейсі.

26. **Порівняння результатів:** runTests() порівнює фактичні результати викликів методів з очікуваними значеннями ("Expected Result").

27. **Визначення статусу тесту:** runTests() встановлює статус тесту ("PASSED" або "FAILED") на основі порівняння.

28. **Модуль протоколювання результатів:** Забезпечено відкриття та запис у файл TestResults.txt.

29. **Формат протоколу:** Результати записуються у файл у зручному для аналізу форматі.

30. **Консольний вивід прогресу:** Програма виводить повідомлення про хід тестування у консоль.

31. **Коректні шляхи до файлів:** Використано відносні шляхи (TestSuite/TestCases.txt, TestSuite/TestResults.txt) для гнучкості розгортання. 32. **Обробка помилок файлового доступу:** Програма перевіряє успішність відкриття файлу TestResults.txt для запису.

33. **Завершення програми з кодом 0:** Програма завершується з кодом повернення 0 при успішному виконанні тестування.

34. **Підтримка ручної діагностики:** Додано std::cin.get(); для утримання консолі від закриття, що полегшує відлагодження.

35. **Співвідношення з вимогами до Teacher:** Додаток відповідає вимогам до тестового драйвера з завдання 4.1-4.3, зокрема, щодо читання тест-кейсів, виконання тестів та протоколювання.

**Налаштування та використання Code::Blocks (досягнення: використання IDE)**

36. **Встановлення IDE:** Успішно встановлено та налаштовано Code::Blocks.

37. **Налаштування компілятора:** Здійснено коректне налаштування компілятора GNU GCC Compiler у Code::Blocks. 38. **Вирішення проблем з DLL:** Подолано проблеми з відсутністю libstdc++-6.dll, що свідчить про повну та коректну інсталяцію компілятора.

39. **Створення проекту:** Створено проект типу "Console application" у Code::Blocks для Teacher.exe.

40. **Додавання файлів до проекту:** Успішно додано файли ClassLab12\_Nemnenko.h та ClassLab12\_Nemnenko.cpp до проекту.

41. **Компіляція проекту:** Проект успішно компілюється без синтаксичних помилок.

42. **Збірка виконуваного файлу:** Створюється виконуваний файл Teacher.exe.

43. **Виконання збірки:** Використано функцію "Build" або "Rebuild" для перекомпіляції після змін.

44. **Запуск програми з IDE:** Програма запускається безпосередньо з Code::Blocks ("Build and Run").

45. **Відлагодження:** Можливість використання логів збірки та консольного виводу для діагностики проблем.

46. **Навігація в IDE:** Вміння працювати з панелями "Management" та "Logs & others".

47. **Розуміння логів компілятора/лінкеру:** Аналіз повідомлень про помилки (undefined reference, can't find compiler executable).

48. **Робота з файлами проекту:** Вміння додавати та організовувати файли в структурі проекту Code::Blocks.

49. **Гнучкість використання IDE:** Використання Code::Blocks для C++ розробки, як того вимагало завдання.

50. **Самостійне вирішення проблем:** Досвід подолання нетривіальних проблем налаштування середовища.

**Методологічні аспекти тестування (досягнення: розуміння процесу тестування)**

51. **Розуміння Unit-тестування?:** Продемонстровано розуміння концепції Unit-тестування шляхом фокусування на окремому класі ClassLab12\_Прізвище.

52. **Тестові сценарії:** Створено тестові сценарії для перевірки функціональності класу.

53. **Тест-кейси:** Формування тест-кейсів з ідентифікатором, дією та очікуваним результатом.

54. **Тест-драйвер:** Розробка Teacher.exe як приклад тестового драйвера. 55. **Протоколювання тестування:** Вміння протоколювати результати тестування у зовнішній файл (TestResults.txt). 56. **Зовнішнє керування тестами:** Тест-кейси зберігаються у зовнішньому файлі (TestCases.txt), що дозволяє легко додавати нові тести без перекомпіляції тестового драйвера. 57. **Автоматизація тестування:** Teacher.exe автоматизує процес виконання тестів, замість ручної перевірки.

58. **Верифікація:** Тестування забезпечує верифікацію того, що розроблений модуль відповідає своїм специфікаціям.

59. **Валідація:** Хоча основне завдання Unit-тестування, воно також сприяє ранній валідації поведінки модуля.

60. **Системне тестування:** Teacher.exe може бути адаптований для системного тестування (як це зазначено в пункті 13 завдання).

61. **Якість коду:** Unit-тестування сприяє покращенню якості коду ClassLab12\_Прізвище шляхом виявлення помилок.

62. **Регресійне тестування:** Структура тестового драйвера дозволяє легко проводити регресійне тестування після змін у класі.

63. **Чітке розділення відповідальності:** Розділення коду ClassLab12\_Прізвище від коду Teacher.exe.

64. **Відлагодження:** Тести допомагають локалізувати помилки в тестуємому модулі.

65. **Документація:** Файли TestCases.txt та TestResults.txt слугують документацією процесу тестування.

66. **Послідовність виконання тестів:** Програма забезпечує послідовне виконання тест-кейсів.

67. **Звітність:** Генерування звіту про результати тестування.

68. **Повторюваність тестів:** Тести можуть бути повторені багато разів з ідентичними результатами за однакових вхідних даних.

69. **Визначення успіху/невдачі:** Чітке визначення "PASSED" або "FAILED" для кожного тест-кейсу.

70. **Незалежність тестів:** Кожен тест-кейс може бути розроблений для перевірки конкретного аспекту функціональності.

**Загальні навички та розуміння розробки ПЗ:**

71. **Розуміння життєвого циклу розробки ПЗ:** Виконання всіх етапів від аналізу вимог до тестування.

72. **Проєктування архітектури:** Розробка архітектури Teacher як окремого компонента.

73. **Детальне проєктування:** Перетворення архітектури на конкретні функції та структури C++.

74. **Управління залежностями:** Розуміння, як Teacher залежить від ClassLab12\_Прізвище.

75. **Використання файлових систем:** Робота з файлами для вводу/виводу. 76. **Обробка рядків та парсинг:** Навички парсингу текстових даних з файлу.

77. **Використання стандартних бібліотек C++:** Ефективне використання iostream, fstream, string, vector.

78. **Дотримання структурних вимог:** Розміщення файлів у Lab12/Software/ та Lab12/TestSuite/.

79. **Навички відлагодження:** Систематичний підхід до виявлення та усунення помилок компіляції та виконання.

80. **Аналітичне мислення:** Аналіз повідомлень про помилки та визначення їх першопричин.

81. **Наполегливість:** Здатність долати численні технічні труднощі, не здаючись.

82. **Самостійність:** Значна частина проблем була діагностована та вирішена самостійно (з консультативною допомогою).

83. **Практичний досвід:** Отримання практичного досвіду розробки реального програмного забезпечення.

84. **Розуміння ієрархії файлів:** Навички роботи з теками та відносними шляхами.

85. **Досвід роботи з CMD/PowerShell:** Запуск виконуваних файлів з командного рядка для діагностики.

86. **Управління версіями:** Підготовка до використання GitHub як системи контролю версій.

87. **Формулювання звітів:** Підготовка до написання звіту та обґрунтування висновків.

88. **Розуміння архітектурних патернів:** Застосування патерну "Тестовий драйвер".

89. **Рефакторинг:** Можливість рефакторингу коду на основі результатів тестування.

90. **Робота з консоллю:** Ефективна взаємодія з консольним додатком для виводу та введення.

91. **Підготовка до подальших завдань:** Фундамент для більш складних завдань з програмування та тестування.

92. **Здатність до адаптації:** Адаптація коду та підходів до вирішення конкретних проблем.

93. **Критичне мислення:** Оцінка ефективності та коректності власного коду.

94. **Професійний розвиток:** Розвиток навичок, важливих для майбутньої професії розробника ПЗ.

95. **Знання C++?:** Поглиблення знань мови C++ (файлові потоки, класи, об'єкти, вектори).

96. **Розуміння циклів розробки:** Проходження через етапи розробки, що імітують реальні проєкти.

97. **Взаємодія компонентів:** Розуміння, як один компонент (Teacher) взаємодіє з іншим (ClassLab12\_Прізвище).

98. **Систематизація знань:** Систематизація теоретичних знань про тестування та їх застосування на практиці.

99. **Навички презентації:** Підготовка матеріалів для представлення результатів (наприклад, скріншоти).

100. **Загальна компетентність:** Демонстрація загальної компетентності у розробці та тестуванні програмного забезпечення в середовищі Code::Blocks.

**Контрольні запитання і завдання**

**1. Що є результатами виконання концептуалізації предметної області, об’єктного аналізу та визначення інтерфейсів сутностей предметної області?**

Результатами цих процесів є:

* **Концептуалізація предметної області:**
  + **Глосарій термінів:** Визначення та пояснення ключових понять, що використовуються в предметній області.
  + **Перелік сутностей:** Ідентифікація основних об'єктів або концепцій, які є важливими для системи (наприклад, "Студент", "Викладач", "Курс", "Прямокутник", "Тест-кейс").
  + **Опис атрибутів сутностей:** Визначення характеристик, які описують кожну сутність (наприклад, для "Прямокутника" - "довжина", "ширина").
  + **Опис зв'язків між сутностями:** Визначення взаємозв'язків між сутностями (наприклад, "Викладач викладає Курс", "Тест-кейс тестує Клас").
  + **Діаграми контексту:** Візуальне представлення високорівневого огляду системи та її взаємодії із зовнішнім середовищем.
  + **Функціональні та нефункціональні вимоги:** Чітко сформульовані вимоги до системи, що випливають з розуміння предметної області.
* **Об’єктний аналіз (на основі результатів концептуалізації):**
  + **Діаграма класів (статична структура):** Основний результат, що відображає ідентифіковані класи, їхні атрибути, методи та статичні зв'язки (асоціації, агрегації, композиції, узагальнення).
  + **Діаграми послідовності / взаємодії (динамічна поведінка):** Опис, як об'єкти взаємодіють між собою для виконання певних сценаріїв використання (Use Cases).
  + **Визначення відповідальності класів:** Чітке розмежування, за які дані та поведінку відповідає кожен клас.
  + **Ієрархії спадкування:** Якщо застосовується, визначення батьківських та дочірніх класів.
  + **Патерни проєктування:** Можливе виявлення та застосування типових патернів проєктування, які відповідають структурі предметної області.
* **Визначення інтерфейсів сутностей предметної області (на основі об'єктного аналізу):**
  + **Специфікація публічних методів класів:** Для кожного класу чітко визначено назви, сигнатури (параметри та тип повернення) всіх публічних методів.
  + **Опис поведінки методів:** Детальний опис, що саме робить кожен публічний метод, які попередні умови мають бути виконані для його виклику (pre-conditions) та які результати він гарантує (post-conditions).
  + **Визначення публічних атрибутів (якщо такі є):** Рідко, але якщо певні дані мають бути безпосередньо доступними ззовні, вони також є частиною інтерфейсу.
  + **Діаграми інтерфейсів (якщо використовуються):** Формальне представлення інтерфейсів, які можуть бути реалізовані різними класами.
  + **Контракти класів:** Чітке визначення "контракту" класу – що він обіцяє робити, якщо його методи викликаються коректно.

**2. Який зв’язок між процесом концептуалізації предметної області та процесами об’єктного аналізу і визначення інтерфейсів?**

Між цими процесами існує **сильний ієрархічний та ітеративний зв'язок**:

1. **Концептуалізація як фундамент:**
   * Концептуалізація є **первинним етапом**, на якому відбувається глибоке занурення в предметну область, її розуміння, ідентифікація ключових понять та їхніх взаємозв'язків.
   * Результати концептуалізації (сутності, їхні атрибути, зв'язки, функціональні вимоги) слугують **вхідними даними** для об'єктного аналізу. Без чіткого розуміння предметної області неможливо коректно ідентифікувати об'єкти.
2. **Об'єктний аналіз як перехід до об'єктної моделі:**
   * На етапі об'єктного аналізу **сутності, ідентифіковані під час концептуалізації, перетворюються на класи** об'єктно-орієнтованої моделі.
   * Визначені атрибути сутностей стають **атрибутами класів**, а поведінка, пов'язана з сутностями, трансформується у **методи класів**.
   * Зв'язки між сутностями з концептуальної моделі перетворюються на **асоціації, агрегації або композиції** між класами.
3. **Визначення інтерфейсів як конкретизація взаємодії:**
   * Визначення інтерфейсів є **подальшою деталізацією** об'єктного аналізу. Воно фокусується на **публічній поведінці** класів – тобто на тому, як інші частини системи (або зовнішні системи) будуть взаємодіяти з даними об'єктами.
   * Інтерфейси випливають з методів, визначених на етапі об'єктного аналізу, і конкретизують їхні сигнатури та контракти. Це є кроком до реалізації принципу **інформаційного приховування** та **інтерфейсної незалежності**.

**Ітеративний характер:** Хоча порядок здається лінійним, ці процеси часто є ітеративними. У ході об'єктного аналізу можуть виникнути нові питання або неточності щодо предметної області, що потребуватиме повернення до етапу концептуалізації для уточнення. Аналогічно, при визначенні інтерфейсів може з'ясуватися, що поведінка деяких класів не була повністю продумана, що знову призведе до доопрацювання об'єктної моделі.

**3. Сформулюйте критерії, за якими чітко можливо визначити: абстракцію сутності предметної області слід описати мовою С++ типом структура (struct) чи типом клас (class)?**

У C++ struct і class є дуже схожими, головна відмінність полягає в **рівні доступу за замовчуванням**: для class члени за замовчуванням private, для struct – public. Однак є усталені конвенції та критерії:

|  |  |  |
| --- | --- | --- |
| **Критерій** | **struct (структура)** | **class (клас)** |
| **Семантика / Призначення** | **Групування даних:** Використовується для агрегації пов'язаних між собою даних. Часто імітує "простий" запис або C-подібну структуру даних. | **Представлення об'єкта / Концепції:** Використовується для представлення складних об'єктів предметної області, які мають як дані, так і поведінку, та потребують інкапсуляції, абстракції, спадкування, поліморфізму. |
| **Основний фокус** | **Дані:** Головна мета – тримати дані разом. Методи можуть бути присутні, але вони, як правило, прості й маніпулюють лише цими даними (наприклад, геттери/сеттери, прості утиліти). | **Поведінка та стан:** Клас зосереджений на тому, що об'єкт *робить* (поведінка) і *яким він є* (стан). Він активно приховує внутрішню реалізацію. |
| **Рівень доступу за замовчуванням** | **Public:** Всі члени (дані та функції), якщо не вказано інше, є public. Це часто дозволяє прямий доступ до даних. | **Private:** Всі члени (дані та функції), якщо не вказано інше, є private. Це сприяє інкапсуляції та приховуванню реалізації. |
| **Інкапсуляція** | **Менший акцент:** Хоча можна використовувати private члени, семантично struct схиляється до відкритих даних. | **Високий акцент:** Основний інструмент для реалізації принципу інкапсуляції, приховуючи внутрішній стан та поведінку. |
| **Спадкування** | Можливе, але рідко використовується для ієрархій поведінки. Частіше для простих ієрархій даних. | **Основний інструмент:** Широко використовується для побудови складних ієрархій типів, реалізації поліморфізму та принципу "є-а" (is-a). |
| **Конструктори/Деструктори** | Можуть мати, але часто використовуються лише для простої ініціалізації. | Зазвичай мають складніші конструктори (включаючи копіювання, переміщення) та деструктори для управління ресурсами та життєвим циклом об'єкта. |
| **Патерни проєктування** | Рідко застосовуються складні патерни. | Основний будівельний блок для реалізації більшості об'єктно-орієнтованих патернів проєктування. |
| **Приклад** | struct Point { int x, y; }; (проста сукупність даних) | class Shape { public: virtual double getArea() = 0; /\* ... \*/ }; (складна абстракція з поведінкою) |

Экспортировать в Таблицы

**Короткий висновок:**

* Використовуйте **struct**, якщо ви просто групуєте кілька пов'язаних змінних, і доступ до цих даних переважно прямий, а поведінка обмежується простими утилітами. Це схоже на запис у базі даних.
* Використовуйте **class**, якщо ви моделюєте більш складну сутність, яка має відповідальність, інкапсулює свої дані, надає чітко визначений інтерфейс для взаємодії, і може брати участь у спадкуванні або поліморфізмі.

**4. Що в програмуванні розуміють під інтерфейсом класу?**

В програмуванні, особливо в контексті об'єктно-орієнтованого програмування, **інтерфейс класу** – це сукупність усіх **публічних членів (методів та, рідше, атрибутів)**, які клас надає для взаємодії з ним ззовні.

Це можна уявити як "контракт" або "протокол" взаємодії. Інтерфейс визначає:

* **Що можна робити** з об'єктом цього класу.
* **Як можна взаємодіяти** з ним (назви методів, їхні параметри, типи повернення).
* **Яку поведінку** об'єкт гарантує при виклику цих методів.

**Ключові аспекти інтерфейсу класу:**

1. **Публічний контракт:** Інтерфейс класу – це його публічний "API" (Application Programming Interface). Він є єдиним способом взаємодії з об'єктом цього класу для інших частин програми.
2. **Абстракція:** Інтерфейс приховує (абстрагує) деталі внутрішньої реалізації класу. Користувач класу (інший розробник, який використовує ваш клас) не повинен знати, як саме клас виконує свої функції; йому достатньо знати, *що* він робить і *як* його викликати.
3. **Інкапсуляція:** Інтерфейс є ключовим для реалізації інкапсуляції. Він дозволяє приховувати внутрішній стан (приватні атрибути) та внутрішню логіку (приватні методи), надаючи лише контрольований доступ до них через публічні методи.
4. **Поділ відповідальності:** Інтерфейс чітко розділяє відповідальність класу: що він обіцяє робити, і як він це робить (реалізація).
5. **Гнучкість та розширюваність:** Добре спроєктований інтерфейс є стабільним, навіть якщо внутрішня реалізація класу змінюється. Це дозволяє рефакторити клас без впливу на код, який його використовує.
6. **Поліморфізм (для абстрактних класів та інтерфейсів у мовах, що їх підтримують):** У деяких мовах (наприклад, Java, C#) є явний концепт "інтерфейсу" як чистої абстракції без реалізації. У C++ цю роль виконують абстрактні класи з чисто віртуальними функціями. Це дозволяє реалізувати поліморфізм – взаємодіяти з об'єктами різних класів через їхній спільний інтерфейс.

**Приклад:**

C++

class MyRectangle {

private:

double length;

double width;

public:

// Інтерфейс класу:

MyRectangle(double l, double w); // Конструктор

double getLength() const; // Метод доступу (геттер)

double getWidth() const; // Метод доступу (геттер)

bool setLength(double l); // Метод зміни (сеттер)

bool setWidth(double w); // Метод зміни (сеттер)

double calculateArea() const; // Метод поведінки

// Приватні/захищені методи (якщо є) НЕ є частиною інтерфейсу.

// private:

// void validateDimensions(double& l, double& w);

};

У цьому прикладі, MyRectangle(double l, double w), getLength(), getWidth(), setLength(), setWidth(), calculateArea() – це все частини інтерфейсу класу MyRectangle.

**5. Обґрунтовано поясніть, чому в класі С++ не можна оголосити конструктор з закритим рівнем доступу?**

Насправді, **в класі C++ МОЖНА оголосити конструктор з закритим (private) або захищеним (protected) рівнем доступу**. Це цілком допустимо і є потужним патерном проєктування.

**Якщо конструктор оголошено як private або protected, це означає, що:**

1. **private конструктор:**
   * **Призначення:** Заборонити пряме створення об'єктів цього класу ззовні.
   * **Обґрунтування:** Це використовується для реалізації таких патернів проєктування, як:
     + **Singleton (Одинак):** Гарантує, що клас має лише один екземпляр, і надає глобальну точку доступу до цього екземпляра. Конструктор робиться private, а статичний публічний метод відповідає за створення та повернення єдиного екземпляра.
     + **Factory Method (Фабричний метод):** Створення об'єктів переноситься до спеціального фабричного методу, який може приймати рішення про те, який конкретний клас (нащадок) створити. Конструктори дочірніх класів можуть бути private, а фабричний метод є public статичним методом, який їх викликає.
     + **Заборона створення об'єктів на стеку:** Якщо ви хочете, щоб об'єкти створювалися лише динамічно (через new), або взагалі заборонити їх створення, окрім як за спеціальною логікою всередині самого класу або його друзів.
2. **protected конструктор:**
   * **Призначення:** Заборонити пряме створення об'єктів цього класу ззовні, але **дозволити його виклик з конструкторів похідних класів**.
   * **Обґрунтування:** Це типово для **абстрактних базових класів**, які призначені лише для успадкування, але не для інстанціювання. protected конструктор дозволяє дочірнім класам ініціалізувати батьківську частину об'єкта, зберігаючи при цьому базовий клас як "неінстанційований" ззовні.

**Отже, правильна відповідь на ваше питання: в C++ можна оголосити конструктор з закритим рівнем доступу, і це робиться для реалізації певних патернів проєктування, таких як Singleton або коли клас є базовим абстрактним класом, який не повинен бути створений безпосередньо.**

Можливо, ваше запитання базується на помилковому припущенні або на знаннях з інших мов програмування, де такі обмеження можуть існувати або інтерпретуватися по-іншому. У C++ це є потужною і часто використовуваною можливістю

**6. Здійсніть порівняльний аналіз перевантаженої функції та функції з параметрами за замовчуванням.**

**Перевантажені функції (Overloaded Functions)** та **Функції з параметрами за замовчуванням (Functions with Default Arguments)** – це два механізми C++ для надання гнучкості виклику функцій, але вони досягають цього різними способами та використовуються в різних сценаріях.

|  |  |  |
| --- | --- | --- |
| **Критерій** | **Перевантажені функції** | **Функції з параметрами за замовчуванням** |
| **Визначення** | Декілька функцій з **одним і тим же іменем**, але з **різними сигнатурами** (різна кількість параметрів або різні типи параметрів). | Одна функція, де один або кілька **параметрів мають значення за замовчуванням**, які компілятор використовуватиме, якщо відповідний аргумент не надано при виклику. |
| **Кількість реалізацій** | **Кілька окремих реалізацій** (тіл функцій), по одному для кожної перевантаженої версії. Кожна реалізація може бути повністю незалежною. | **Одна реалізація** (одне тіло функції). Значення за замовчуванням "вставляються" компілятором у місце виклику, а не змінюють саму реалізацію. |
| **Вибір версії при виклику** | **Вибір на основі сигнатури:** Компілятор обирає правильну перевантажену функцію під час компіляції, виходячи з кількості та типів аргументів, наданих при виклику. | **Вибір на основі наданих аргументів:** Компілятор використовує значення за замовчуванням для тих параметрів, для яких аргументи не були надані. Якщо аргумент наданий, він перевизначає значення за замовчуванням. |