Поняття класу та об'єкта в об'єктно-орієнтованому програмуванні

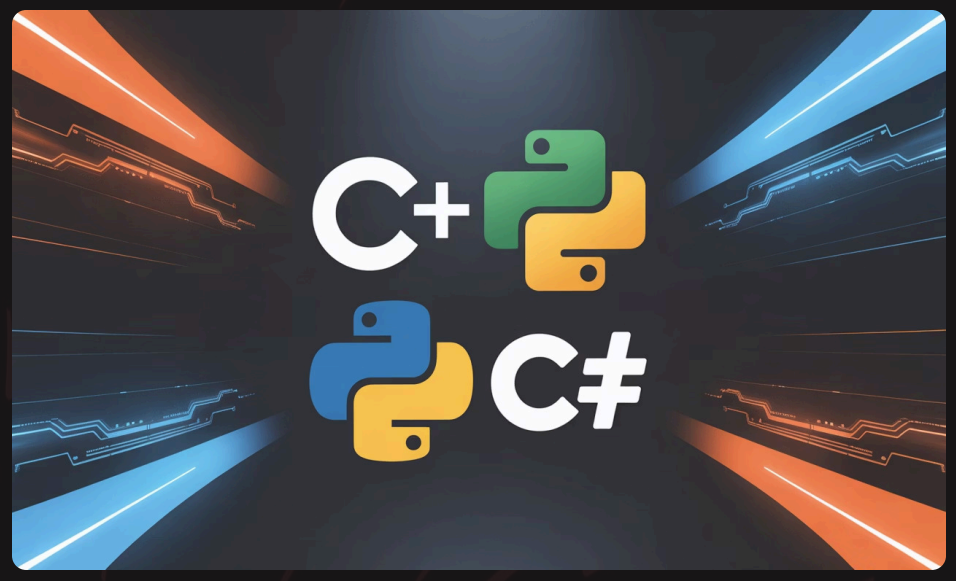
Цей опорний конспект присвячений фундаментальним поняттям об'єктно-орієнтованого програмування: класам, об'єктам, конструкторам, деструкторам, інтерфейсам та реалізації. Документ допоможе вам зрозуміти основи ООП для успішної підготовки до єдиного фахового вступного випробування з інформаційних технологій.

Вступ: Роль ООП у мовах програмування

Об'єктно-орієнтоване програмування (ООП) — це одна з основних парадигм програмування, яка розглядає програму як сукупність об'єктів, що взаємодіють між собою. На сьогодні ООП займає центральне місце серед інших парадигм, таких як процедурне, функціональне та логічне програмування. Воно виникло як відповідь на зростаючу складність програмних систем та необхідність більш природного моделювання реального світу в коді.

ООП базується на чотирьох фундаментальних принципах: інкапсуляція, наслідування, поліморфізм та абстракція. Ці принципи допомагають створювати більш модульний, підтримуваний та розширюваний код, що особливо важливо для великих проєктів з багатьма розробниками.

Популярність ООП підтверджується широким використанням об'єктно-орієнтованих мов програмування в сучасній індустрії розробки програмного забезпечення.
Найпоширенішими ООП-мовами є:



ООП стало домінуючою парадигмою в багатьох сферах розробки програмного забезпечення, від створення мобільних додатків до великих корпоративних систем. Це дозволяє розробникам моделювати складні системи на основі реальних об'єктів та їх взаємодії, що робить код більш інтуїтивно зрозумілим та придатним для повторного використання.

- **C++** — потужна мова з підтримкою різних парадигм, включаючи ООП, що забезпечує високу продуктивність та контроль над ресурсами.
- **Java** — чисто об'єктно-орієнтована мова з автоматичним керуванням пам'яттю та кросплатформеністю.
- **Python** — інтерпретована мова з динамічною типізацією та простим синтаксисом, що підтримує різні стилі програмування, включаючи ООП.
- **C#** — сучасна ООП мова від Microsoft, розроблена для .NET платформи з багатими можливостями та елегантним синтаксисом.

Клас: Визначення, структура, приклади

Клас є основним будівельним блоком в об'єктно-орієнтованому програмуванні та визначає шаблон або схему для створення об'єктів. Він виступає як креслення, яке описує структуру та поведінку сутностей певного типу. Клас можна порівняти з кресленням будинку, де об'єкти – це конкретні будинки, побудовані за цим кресленням.

Структура класу

Структура класу складається з двох основних компонентів:

- **Поля (атрибути)** – змінні, що зберігають стан об'єкта. Наприклад, для класу Student це можуть бути name, age, gradePoint.
- **Методи** – функції, що визначають поведінку об'єкта та операції, які він може виконувати. Наприклад, для класу Student це можуть бути методи study(), takeExam(), calculateAverageGrade().

Модифікатори доступу

Важливими елементами класу є модифікатори доступу, які контролюють видимість членів класу:

- **public**: доступні звідусіль
- **private**: доступні лише всередині класу
- **protected**: доступні всередині класу та його нащадків

Статичні члени класу

Належать класу, а не окремим об'єктам:

- **Статичні поля**: спільні дані для всіх об'єктів
- **Статичні методи**: викликаються через клас, не через об'єкт

Константні члени

Елементи, які не можуть змінюватися:

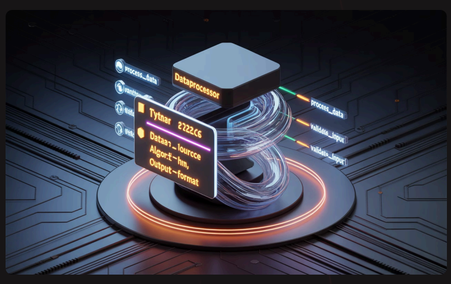
- **const поля**: незмінні значення
- **const методи**: не модифікують стан об'єкта

Синтаксис створення класу відрізняється залежно від мови програмування. Ось приклади оголошення простого класу Student в C++ та Java:

```
// Приклад класу в C++
class Student {
private:
    string name;
    int age;
    double gradePoint;

public:
    Student(string n, int a) { // конструктор
        name = n;
        age = a;
        gradePoint = 0;
    }

    void study() {
        // реалізація методу
    }

    double getGradePoint() {
        return gradePoint;
    }
};
```

Класи є потужним інструментом для створення абстракцій, що моделюють реальні об'єкти та концепції в коді. Правильне проєктування класів є важливим аспектом розробки об'єктно-орієнтованих систем.

Об'єкт: Сутність, створення, використання



Об'єкт є фундаментальною концепцією об'єктно-орієнтованого програмування та являє собою конкретний екземпляр класу. Якщо клас — це шаблон або креслення, то об'єкт — це фактичний "екземпляр", створений на основі цього креслення. Об'єкти мають два основні аспекти: стан (дані) та поведінку (методи).

Ключові характеристики об'єктів

- **Стан** — визначається значеннями атрибутів (полів) об'єкта. Наприклад, для об'єкта класу Student це можуть бути конкретні значення імені, віку та середнього балу.
- **Поведінка** — визначається методами, які об'єкт може виконувати. Наприклад, студент може навчатися, складати іспити, отримувати оцінки.
- **Ідентичність** — кожен об'єкт є унікальним, навіть якщо має точно такий самий стан як інший об'єкт.

Створення об'єктів

Процес створення об'єкта називається інстанціюванням класу. У більшості об'єктно-орієнтованих мов програмування для створення об'єкта використовується оператор **new**. Ось як це виглядає в різних мовах:

```
// C++
Student* ivan = new Student("Іван Петренко", 19);

// Java
Student ivan = new Student("Іван Петренко", 19);

// Python
ivan = Student("Іван Петренко", 19)

// C#
Student ivan = new Student("Іван Петренко", 19);
```

Використання об'єктів

Після створення об'єкта, ми можемо взаємодіяти з ним через його інтерфейс — публічні методи та властивості. Цей процес відбувається за допомогою операторів доступу до членів класу:

```
// C++
ivan->study();               // Виклик методу
double gpa = ivan->getGradePoint(); // Отримання значення

// Java
ivan.study();              // Виклик методу
double gpa = ivan.getGradePoint();  // Отримання значення
```

Створення об'єкта

Спочатку ми створюємо новий екземпляр класу за допомогою оператора new, який викликає конструктор класу.

Ініціалізація об'єкта

Конструктор ініціалізує стан об'єкта, встановлюючи початкові значення полів.

Використання об'єкта

Ми використовуємо об'єкт, викликаючи його методи та отримуючи доступ до властивостей.

Знищення об'єкта

Коли об'єкт більше не потрібен, він знищується системою (автоматично або вручну).

Об'єкти є основними елементами взаємодії в ООП-програмах. Вони інкапсулюють дані та функціональність, забезпечуючи модульність та повторне використання коду. Розуміння того, як створювати та використовувати об'єкти, є ключовим для успішного об'єктно-орієнтованого програмування.

Основні принципи ООП: Інкапсуляція, Наслідування, Поліморфізм

Об'єктно-орієнтоване програмування базується на чотирьох фундаментальних принципах, які є його опорними стовпами. Розуміння цих принципів критично важливе для ефективного застосування ООП у розробці програмного забезпечення.

Інкапсуляція

Інкапсуляція — це механізм, який поєднує дані (атрибути) та методи, що працюють з цими даними, в єдину одиницю — клас, при цьому обмежуючи прямий доступ до деяких компонентів об'єкта.

- Приховує внутрішню реалізацію від зовнішнього світу
- Забезпечує контрольований доступ до даних через методи
- Підвищує безпеку та стабільність програми

Абстракція

Абстракція — це процес виділення істотних характеристик об'єкта, які відрізняють його від інших об'єктів, та ігнорування неважливих деталей.

- Зосереджується на "що робить", а не "як робить"
- Спрощує складні системи, приховуючи непотрібні деталі
- Реалізується через інтерфейси та абстрактні класи

Наслідування

Наслідування — це механізм, який дозволяє новому класу отримувати (успадковувати) властивості та методи існуючого класу, розширюючи або змінюючи їх поведінку.

- Створює ієрархії класів
- Сприяє повторному використанню коду
- Підтримує "відношення є" (наприклад, "Студент є Людиною")

Поліморфізм

Поліморфізм дозволяє об'єктам різних класів реагувати по-різному на однакові методи, забезпечуючи гнучкість у роботі з об'єктами різних типів.

- Дозволяє використовувати єдиний інтерфейс для різних реалізацій
- Включає перевантаження методів і операторів
- Забезпечує гнучкість коду та розширюваність

Ці принципи не є окремими концепціями, а тісно взаємопов'язаними механізмами, які разом формують потужну парадигму програмування. Інкапсуляція захищає дані та забезпечує контрольований доступ до них. Наслідування дозволяє розширювати функціональність існуючих класів без модифікації їхнього коду. Поліморфізм забезпечує гнучкість у роботі з об'єктами різних типів через спільний інтерфейс. Абстракція допомагає зосередитися на важливих аспектах об'єкта, ігноруючи несуттєві деталі.

Застосування цих принципів у розробці програмного забезпечення сприяє створенню більш модульного, підтримуваного та розширюваного коду, що є особливо цінним для великих та складних систем.

Конструктори: Визначення та призначення

Конструктор – це спеціальний метод класу, який автоматично викликається при створенні об'єкта. Головне призначення конструктора – виконати початкову ініціалізацію об'єкта, тобто встановити початкові значення полів та підготувати об'єкт до використання. Конструктор гарантує, що об'єкт завжди знаходиться у валідному стані після створення.

Призначення конструкторів

Ініціалізація полів об'єкта

Виділення ресурсів

Пам'ять, файли, мережеві з'єднання

Встановлення зв'язків

З іншими об'єктами системи

Види конструкторів

У більшості об'єктно-орієнтованих мов існує декілька типів конструкторів:

Конструктор за замовчуванням

Конструктор без параметрів, який створюється компілятором автоматично, якщо в класі не визначено жодного конструктора. Ініціалізує поля значеннями за замовчуванням (нулі для числових типів, null для посилань).

Конструктор з параметрами

Приймає один або більше параметрів, які використовуються для ініціалізації полів об'єкта. Дозволяє встановити початковий стан об'єкта під час його створення.

Конструктор копіювання

Створює новий об'єкт як копію існуючого об'єкта того ж класу. Особливо важливий у C++ для правильного управління ресурсами при копіюванні об'єктів.

Приклади конструкторів у різних мовах програмування:

```
// C++ - різні типи конструкторів
class Student {
private:
    string name;
    int age;
    float gpa;

public:
    // Конструктор за замовчуванням
    Student() {
    name = "Невідомо";
    age = 0;
    gpa = 0.0;
}

// Конструктор з параметрами
Student(string n, int a) {
    name = n;
    age = a;
    gpa = 0.0;
}

// Конструктор копіювання
Student(const Student &other) {
    name = other.name;
    age = other.age;
    gpa = other.gpa;
    }
};

// Використання конструкторів
Student s1;                  // Викликає конструктор за замовчуванням
Student s2("Іван", 20);      // Викликає конструктор з параметрами
Student s3 = s2;             // Викликає конструктор копіювання
```

Конструктори є важливим інструментом для забезпечення коректної ініціалізації об'єктів. Вони допомагають дотримуватися принципу інкапсуляції, дозволяючи об'єкту контролювати свій внутрішній стан з моменту створення. Правильно спроєктовані конструктори сприяють створенню надійних та передбачуваних об'єктів, які завжди починають своє "життя" у валідному стані.

Деструктори: Поняття і використання

Деструктор — це спеціальний метод класу, який автоматично викликається, коли об'єкт знищується. Головна мета деструктора — правильно звільнити ресурси, які були виділені об'єкту під час його існування, такі як динамічна пам'ять, файлові дескриптори, мережеві з'єднання тощо.

Основні характеристики деструкторів

- Деструктор має те саме ім'я, що й клас, але з префіксом ~ (у C++)
- В класі може бути лише один деструктор
- Деструктор не має параметрів і не повертає значення
- Не може бути перевантажений або успадкований
- Викликається автоматично при знищенні об'єкта

Відмінності від конструктора

Характеристика	Конструктор	Деструктор
Призначення	Ініціалізація об'єкта	Очищення ресурсів об'єкта
Час виклику	При створенні об'єкта	При знищенні об'єкта
Кількість у класі	Може бути кілька (перевантаження)	Тільки один
Параметри	Може приймати параметри	Не приймає параметрів
Повернення значення	Не повертає значення	Не повертає значення

Важливість деструкторів

Деструктори є особливо важливими в мовах програмування, які не мають автоматичного керування пам'яттю (збирача сміття). У C++, наприклад, програміст відповідає за звільнення динамічно виділеної пам'яті, і деструктори є ключовим інструментом для запобігання витоку пам'яті.

В мовах із збирачем сміття, таких як Java, C# або Python, необхідність у деструкторах менша, оскільки пам'ять звільняється автоматично. Однак, навіть у цих мовах існують аналоги деструкторів для обробки інших ресурсів, які не керуються збирачем сміття (наприклад, метод finalize() в Java, методи з аннотацією @PreDestroy у Spring, або методи __del__ у Python).

```
// Приклад деструктора в C++
class ResourceManager {
private:
    int* dataArray;
    FILE* logFile;

public:
    // Конструктор виділяє ресурси
    ResourceManager() {
        dataArray = new int[1000]; // Виділення динамічної пам'яті
        logFile = fopen("log.txt", "w"); // Відкриття файлу
    }

    // Деструктор звільняє ресурси
    ~ResourceManager() {
        delete[] dataArray; // Звільнення динамічної пам'яті
        fclose(logFile); // Закриття файлу
        cout << "Ресурси звільнено" << endl;
    }
};

// Використання
void someFunction() {
    ResourceManager rm; // Створення об'єкта
    // Використання об'єкта...
} // Об'єкт rm виходить з області видимості, деструктор викликається автоматично
```

Правильна реалізація деструкторів є критично важливою для запобігання витоку ресурсів, особливо в довготривалих програмах або тих, що працюють з обмеженими ресурсами. В C++ це є частиною концепції RAII (Resource Acquisition Is Initialization — Отримання ресурсу є ініціалізацією), яка гарантує, що ресурси звільняються автоматично, коли об'єкт виходить з області видимості.

Інтерфейс класу: Значення та призначення

Інтерфейс класу – це набір публічно доступних методів та властивостей, через які зовнішній код може взаємодіяти з об'єктами класу. Інтерфейс визначає, що об'єкт може робити, але не розкриває, як саме це реалізовано. Він служить "контрактом" між класом та його користувачами, обіцяючи певну функціональність.

Функції інтерфейсу

Інтерфейс класу виконує роль чітко визначеного "контракту" між класом та його користувачами. Він встановлює чіткі рамки взаємодії, дозволяючи користувачам класу знати, які операції можна виконувати, не вникаючи в деталі реалізації.

Безпека та стабільність

Добре спроєктований інтерфейс обмежує доступ до внутрішнього стану об'єкта, захищаючи його від неправильного використання. Це підвищує безпеку та надійність програми, оскільки внутрішні дані можна модифікувати лише контрольованим способом.

Гнучкість розробки

Відокремлення інтерфейсу від реалізації дозволяє змінювати внутрішню структуру класу без впливу на код, який його використовує. Це сприяє більш гнучкій розробці та полегшує підтримку програмного забезпечення протягом його життєвого циклу.

До складу інтерфейсу класу входять:

- **Публічні методи**: функції, які можуть викликати користувачі класу
- **Публічні властивості**: атрибути, до яких можна отримати доступ ззовні
- **Перевантажені оператори**: визначають поведінку операторів для об'єктів класу
- **Публічні константи та типи**: константи та типи даних, визначені в класі

Приклад інтерфейсу класу в C++:

```
class BankAccount {
private:
    // Ці поля є частиною реалізації, а не інтерфейсу
    string accountNumber;
    double balance;
    string owner;

public:
    // Ці методи є частиною інтерфейсу
    BankAccount(string accountNumber, string owner);
    double getBalance() const;
    void deposit(double amount);
    bool withdraw(double amount);
    void transferTo(BankAccount &recipient, double amount);
    string getOwner() const;
    string getAccountNumber() const;
};
```

У цьому прикладі інтерфейс класу BankAccount складається з конструктора та шести публічних методів. Користувачі класу можуть створювати банківські рахунки, перевіряти баланс, вносити та знімати кошти, а також переказувати гроші між рахунками. Однак, вони не мають прямого доступу до внутрішніх полів, таких як balance, що захищає ці дані від неправильного використання.

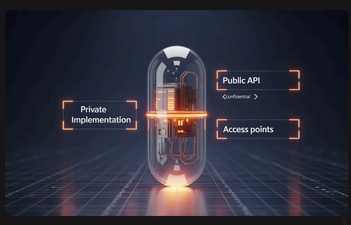
Важливо відрізняти поняття "інтерфейс класу" від "інтерфейсу" як окремого конструкту в деяких мовах програмування (наприклад, interface в Java, C# або TypeScript). Ці інтерфейси є спеціальними структурами, що визначають набір методів, які класи повинні реалізувати, але не містять самої реалізації.

Добре спроєктований інтерфейс класу має бути інтуїтивно зрозумілим, послідовним та простим, надаючи лише ті методи, які дійсно потрібні користувачам класу. Це полегшує використання класу та зменшує ймовірність помилок.

Реалізація класу: Приватність і інкапсуляція

Реалізація класу – це внутрішня частина класу, яка визначає, як саме виконуються операції, оголошені в інтерфейсі. Реалізація включає приватні поля, приватні методи, а також код, що реалізує публічні методи. На відміну від інтерфейсу, реалізація прихована від зовнішнього коду і може змінюватися без впливу на користувачів класу.

Приватна частина класу



Приватні елементи класу доступні лише всередині методів цього класу. Вони становлять "закриту" частину класу і не є видимими для користувачів класу.

- **Приватні поля (атрибути)**: зберігають внутрішній стан об'єкта
- **Приватні методи**: допоміжні функції, що використовуються для реалізації публічних методів
- **Приватні константи та типи**: використовуються для внутрішньої роботи класу

Інкапсуляція та приховування даних

Інкапсуляція – один з фундаментальних принципів ООП – напряму пов'язана з концепцією реалізації класу. Вона включає два ключові аспекти:

Поєднання даних і методів

Об'єднання в одній сутності (класі) даних та функцій, що оперують цими даними, дозволяє створити логічно завершений компонент з чітко визначеною відповідальністю.

Приховування внутрішніх деталей

Обмеження доступу до внутрішнього стану об'єкта, дозволяючи взаємодіяти з ним лише через чітко визначений інтерфейс. Це захищає дані від помилкових змін та дозволяє змінювати реалізацію без впливу на зовнішній код.

Контрольована взаємодія

Надання доступу до даних лише через методи, які можуть перевіряти вхідні дані, підтримувати інваріанти та забезпечувати узгодженість стану об'єкта.

Приклад відокремлення інтерфейсу від реалізації:

```
// Файл StringList.h - інтерфейс класу
class StringList {
public:
    // Публічний інтерфейс
    StringList(); // Конструктор
    ~StringList(); // Деструктор
    void add(const string &str); // Додати рядок
    bool remove(const string &str); // Видалити рядок
    bool contains(const string &str) const; // Перевірити наявність
    size_t size() const; // Отримати розмір

private:
    // Приватна реалізація (деталі приховані)
    struct Node {
        string data;
        Node* next;
    };

    Node* head;
    size_t count;

    // Приватні допоміжні методи
    Node* findNode(const string &str) const;
    void clearList();
};

// Файл StringList.cpp - реалізація класу
StringList::StringList() : head(nullptr), count(0) {}

StringList::~StringList() {
    clearList();
}

void StringList::add(const string &str) {
    Node* newNode = new Node{str, head};
    head = newNode;
    count++;
}

bool StringList::remove(const string &str) {
    // Реалізація методу видалення...
}

// Інші методи...
```

У цьому прикладі публічний інтерфейс класу `StringList` надає методи для роботи зі списком рядків, але приховує деталі того, як список зберігається та як саме виконуються операції. Це дозволяє змінити внутрішню реалізацію (наприклад, використовувати масив замість зв'язаного списку) без впливу на код, який використовує цей клас.

Відокремлення інтерфейсу від реалізації має багато переваг:

- Підвищення безпеки даних та надійності коду
- Полегшення підтримки та модифікації класу
- Зменшення зв'язності між компонентами системи
- Спрощення розуміння та використання класу

Ця концепція є основою для створення стабільних, розширюваних та підтримуваних об'єктно-орієнтованих систем.

Практичні приклади: Конструктори, деструктори, інтерфейс

Розглянемо повний приклад класу, який демонструє використання конструкторів, деструкторів, інтерфейсу та реалізації. Ми створимо клас DynamicArray, який представляє динамічний масив цілих чисел – спрощену версію класу std::vector з C++.


```
// Клас DynamicArray - динамічний масив цілих чисел
class DynamicArray {
private:
    // Приватні члени (реалізація)
    int* data;       // Вказівник на динамічний масив
    size_t size;     // Поточний розмір масиву
    size_t capacity; // Ємність масиву (виділена пам'ять)

    // Приватний метод для збільшення ємності
    void resize(size_t newCapacity) {
        // Створюємо новий масив
        int* newData = new int[newCapacity];

        // Копіюємо дані зі старого масиву
        for (size_t i = 0; i < size; i++) {
            newData[i] = data[i];
        }

        // Звільняємо стару пам'ять
        delete[] data;

        // Оновлюємо вказівник та ємність
        data = newData;
        capacity = newCapacity;
    }

public:
    // Конструктор за замовчуванням
    DynamicArray() : data(nullptr), size(0), capacity(0) {
        // Виділяємо початкову пам'ять
        data = new int[4];
        capacity = 4;
    }

    // Конструктор з параметром (початкова ємність)
    DynamicArray(size_t initialCapacity)
        : data(nullptr), size(0), capacity(0) {
        // Виділяємо пам'ять заданого розміру
        data = new int[initialCapacity];
        capacity = initialCapacity;
    }

    // Конструктор копіювання
    DynamicArray(const DynamicArray& other)
        : data(nullptr), size(other.size), capacity(other.capacity) {
        // Виділяємо пам'ять та копіюємо дані
        data = new int[capacity];
        for (size_t i = 0; i < size; i++) {
            data[i] = other.data[i];
        }
    }

    // Деструктор
    ~DynamicArray() {
        // Звільняємо виділену пам'ять
        delete[] data;
    }

    // Публічні методи (інтерфейс)

    // Додати елемент в кінець масиву
    void push_back(int value) {
        // Якщо недостатньо місця, збільшуємо ємність
        if (size == capacity) {
            resize(capacity * 2);
        }

        // Додаємо новий елемент
        data[size] = value;
        size++;
    }

    // Отримати елемент за індексом
    int get(size_t index) const {
        if (index >= size) {
            throw std::out_of_range("Індекс поза межами масиву");
        }
        return data[index];
    }

    // Встановити значення елемента за індексом
    void set(size_t index, int value) {
        if (index >= size) {
            throw std::out_of_range("Індекс поза межами масиву");
        }
        data[index] = value;
    }

    // Отримати поточний розмір масиву
    size_t getSize() const {
        return size;
    }

    // Отримати поточну ємність масиву
    size_t getCapacity() const {
        return capacity;
    }

    // Очистити масив
    void clear() {
        size = 0;
    }
};
```

Пояснення коду крок за кроком

1. **Реалізація (приватна частина)**: Клас містить три приватні поля для зберігання даних масиву, його розміру та ємності, а також приватний метод resize для внутрішнього використання.
2. **Конструктори**: Клас має три конструктори – за замовчуванням, з параметром та копіювання. Кожен конструктор ініціалізує об'єкт, виділяючи пам'ять і встановлюючи початкові значення полів.
3. **Деструктор**: Деструктор звільняє динамічно виділену пам'ять, запобігаючи витоку ресурсів.
4. **Інтерфейс (публічна частина)**: Клас надає ряд публічних методів для взаємодії з масивом, такі як push_back, get, set та інші.

Використання класу

```
// Створення та використання об'єкта DynamicArray
int main() {
    // Створюємо масив за допомогою конструктора за замовчуванням
    DynamicArray arr;

    // Додаємо елементи
    arr.push_back(10);
    arr.push_back(20);
    arr.push_back(30);

    // Виводимо елементи
    for (size_t i = 0; i < arr.getSize(); i++) {
        cout << arr.get(i) << " ";
    }
    cout << endl;

    // Змінюємо значення
    arr.set(1, 25);

    // Виводимо елементи після зміни
    for (size_t i = 0; i < arr.getSize(); i++) {
        cout << arr.get(i) << " ";
    }
    cout << endl;

    // Інформація про масив
    cout << "Розмір: " << arr.getSize() << endl;
    cout << "Ємність: " << arr.getCapacity() << endl;

    // Створюємо копію масиву
    DynamicArray arrCopy = arr;

    // Змінюємо копію
    arrCopy.push_back(40);

    // Порівнюємо оригінал і копію
    cout << "Оригінал: ";
    for (size_t i = 0; i < arr.getSize(); i++) {
        cout << arr.get(i) << " ";
    }
    cout << endl;

    cout << "Копія: ";
    for (size_t i = 0; i < arrCopy.getSize(); i++) {
        cout << arrCopy.get(i) << " ";
    }
    cout << endl;

    return 0;
} // Деструктори викликаються автоматично
```

Цей приклад демонструє основні принципи ООП:

Інкапсуляція

Дані масиву (вказівник data, розмір size та ємність capacity) є приватними і доступні лише через методи класу. Користувачі класу не мають прямого доступу до внутрішніх деталей реалізації.

Конструктори та деструктор

Клас має кілька конструкторів для різних сценаріїв створення об'єктів, а також деструктор для звільнення ресурсів. Вони забезпечують правильне створення та знищення об'єктів.

Інтерфейс та реалізація

Клас чітко розділяє інтерфейс (публічні методи) та реалізацію (приватні поля та методи). Внутрішня реалізація може бути змінена без впливу на код, який використовує клас.

Цей приклад показує, як основні концепції ООП – класи, об'єкти, конструктори, деструктори, інтерфейс та реалізація – працюють разом, утворюючи потужний інструмент для моделювання та структурування коду. Розуміння цих концепцій є ключовим для успішного застосування об'єктно-орієнтованого програмування у розробці програмного забезпечення.

Приклади тестових завдань по даній темі для підготовки

1. Що таке клас в об'єктно-орієнтованому програмуванні?

- А) Екземпляр об'єкта в пам'яті
- Б) Функція, що виконує певне завдання
- В) Шаблон для створення об'єктів, що визначає їх структуру та поведінку
- Г) Змінна, яка зберігає значення

Правильна відповідь: В

Пояснення: Клас - це шаблон або креслення, що визначає структуру (поля, атрибути) та поведінку (методи) об'єктів. Він служить основою для створення об'єктів з однаковими характеристиками.

2. Що таке об'єкт в ООП?

- А) Екземпляр класу, що створений у пам'яті
- Б) Фрагмент коду, що виконується послідовно
- В) Функція, яка повертає значення
- Г) Тип даних для зберігання тексту

Правильна відповідь: А

Пояснення: Об'єкт - це конкретний екземпляр класу, що створюється в пам'яті під час виконання програми. Він має власний стан (значення полів) і поведінку, визначену методами класу.

3. Що таке конструктор класу?

- А) Метод, який виконується при знищенні об'єкта
- Б) Метод для зміни значень полів об'єкта
- В) Спеціальний метод, що викликається при створенні об'єкта
- Г) Функція для виведення даних на екран

Правильна відповідь: В

Пояснення: Конструктор - це спеціальний метод класу, який автоматично викликається під час створення нового об'єкта. Він використовується для ініціалізації полів об'єкта та виконання інших дій, необхідних для правильного створення об'єкта.

4. Що таке деструктор?

- А) Метод, який видаляє поля класу
- Б) Спеціальний метод, що викликається при знищенні об'єкта
- В) Метод для створення нових об'єктів
- Г) Функція, що змінює значення змінних

Правильна відповідь: Б

Пояснення: Деструктор - це спеціальний метод класу, який автоматично викликається перед видаленням об'єкта з пам'яті. Він використовується для звільнення ресурсів, якими користувався об'єкт (наприклад, закриття файлів, звільнення динамічної пам'яті).

5. Що таке інкапсуляція в ООП?

- А) Механізм успадкування властивостей від батьківського класу
- Б) Здатність об'єктів різних класів відповідати на однакові повідомлення
- В) Принцип приховування внутрішніх даних класу та надання доступу через публічний інтерфейс
- Г) Процес створення нових об'єктів

Правильна відповідь: В

Пояснення: Інкапсуляція - один з фундаментальних принципів ООП, який передбачає приховування внутрішньої реалізації класу (даних та методів) і надання доступу до них лише через чітко визначений публічний інтерфейс. Це дозволяє захистити дані від неконтрольованої зміни та спрощує використання класу.

6. Що означає модифікатор доступу "private" в класі?

- А) Поля та методи доступні для всіх
- Б) Поля та методи доступні тільки всередині класу
- В) Поля та методи доступні для нащадків класу
- Г) Поля та методи доступні всередині пакету

Правильна відповідь: Б

Пояснення: Модифікатор "private" обмежує доступ до членів класу, дозволяючи використовувати їх лише всередині самого класу. Це важливий механізм інкапсуляції, який допомагає приховувати внутрішню реалізацію класу.

7. Що таке наслідування в ООП?

- А) Можливість об'єднувати дані та методи в одній структурі
- Б) Здатність об'єктів різних класів виконувати однойменні методи по-різному
- В) Механізм, який дозволяє класу отримувати властивості та поведінку іншого класу
- Г) Процес створення об'єктів класу

Правильна відповідь: В

Пояснення: Наслідування - один з основних принципів ООП, який дозволяє створювати нові класи на основі існуючих. Клас-нащадок успадковує поля та методи батьківського класу, може додавати нові та перевизначати існуючі. Це сприяє повторному використанню коду та створенню ієрархій класів.

8. Що таке поліморфізм в ООП?

- А) Принцип приховування даних і надання інтерфейсу
- Б) Здатність об'єктів з однаковим інтерфейсом мати різні реалізації
- В) Механізм створення нових класів на основі існуючих
- Г) Процес структурування даних у класи

Правильна відповідь: Б

Пояснення: Поліморфізм - це принцип ООП, який дозволяє використовувати об'єкти з однаковим інтерфейсом без інформації про конкретний тип цих об'єктів. Це дозволяє писати більш гнучкий код, який може працювати з різними типами об'єктів, що підтримують спільний інтерфейс.

9. Що таке інтерфейс класу?

- А) Набір приватних полів класу
- Б) Зовнішній вигляд графічного додатку
- В) Набір публічних методів, через які можна взаємодіяти з об'єктом
- Г) Спосіб реалізації наслідування

Правильна відповідь: В

Пояснення: Інтерфейс класу - це сукупність публічних методів та властивостей, які доступні для використання іншими класами. Він визначає, як можна взаємодіяти з об'єктами даного класу, не розкриваючи деталей внутрішньої реалізації.

10. Яке твердження щодо об'єктів та класів є правильним?

- А) Клас - це екземпляр об'єкта
- Б) Від одного класу можна створити лише один об'єкт
- В) Об'єкт - це екземпляр класу
- Г) Класи та об'єкти - це одне й те саме поняття

Правильна відповідь: В

Пояснення: Об'єкт є екземпляром класу. Клас виступає як шаблон або опис, а об'єкт - це конкретна реалізація цього шаблону. Від одного класу можна створити багато різних об'єктів, кожен з яких матиме свій стан, але поділятиме ту саму структуру та поведінку.