Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра математики

Курсова робота

освітній ступінь – бакалавр

на тему: «**ДОСЛІДЖЕННЯ МОЖЛИВОСТЕЙ СТАТИЧНОГО ПОЛІМОРФІЗМУ**»

Виконав: студент 3-го року навчання,

Освітньої програми «Інженерія програмного забезпечення», 121

Санченко Георгій Олександрович

Керівник Бублик В.В.

кандидат наук, доцент

Київ – 2023

Календарний план виконання роботи:

№ п/п	Назва етапу кваліфікаційної роботи	Термін виконання етапу	Примітка
1.	Отримання теми курсової роботи	27.02.2023	
2.	Опрацювання літератури за темою роботи	Березень 2023	
3.	Розробка проекту	Квітень 2023	
4.	Написання текстової частини роботи	Травень 2023	
5.	Здача роботи на перевірку на плагіат	18.05.2023	
6.	Захист курсової роботи	25.05.2023	

Науковий керівник Бублик Володимир Васильович (ПІБ)

Виконавець курсової роботи Санченко Георгій Олександрович (ПІБ)

ЗМІСТ

ВСТУП

У період швидкого технологічного розвитку C++ залишається однією з найбільш широко вживаних та актуальних серед сучасних мов програмування. Такий успіх C++ можна пояснити підтримкою багатьох парадигм програмування, таких як процедурне, об’єктно-орієнтоване, функціональне та узагальнене програмування. Таке різноманіття дозволяє розробникам писати високопродуктивний код в багатьох сферах, як от - системне програмування, ігрові двигуни та системи високопродуктивних обчислень.

Мультипарадигменність C++ є можливою внаслідок підтримки широкого спектра концепцій програмування. Однією з них є поліморфізм – одна з основних концепцій об’єктно-орієнтованого програмування, яка дозволяє створювати гнучкий та масштабований код завдяки єдиному інтерфейсу, що може представляти багато різних типів. У C++ поліморфізм можна досягнути як динамічними, так і статичними засобами.

Динамічний поліморфізм досягається за допомогою успадкування та віртуальних функцій, що дозволяє виконувати динамічну диспетчеризацію викликів функцій у залежності від реального типу об’єкта під час виконання програми. Цей механізм вживають для створення якісного коду, який є легким для розширення та задовольняє принципам SOLID. Проте, динамічний поліморфізм має певні недоліки, як-от додаткове використання пам’яті для віртуальних таблиць та погіршена ефективність виконання.

На противагу недолікам динамічного поліморфізму, серед науковців та розробників зростає зацікавленість у дослідженні альтернативних засобів досягнення поліморфізму з мінімальними затратами ресурсів. Прикладом такого засобу є статичний поліморфізм, або поліморфізм на етапі компіляції.

Метою курсової роботи є дослідження способів реалізації статичного поліморфізму мовою програмування C++, визначення їх переваг та недоліків, та порівняння із динамічним поліморфізмом. Основними завданнями дослідження є:

- вивчення поліморфізму, його видів та ролі у програмуванні мовою C++
- вивчення поняття статичного поліморфізму та аналіз способів його реалізації засобами мови C++
- створення прикладів реалізації та застосування статичного поліморфізму в реальних умовах
- дослідження застосування статичного поліморфізму в проєктах з відкритими джерелами.

Робота складається з трьох розділів.

У першому розділі розглянуто поняття поліморфізму та його різновиди, визначення статичного поліморфізму та патерни його реалізації, зокрема так званий патерн «дивно рекурсивного шаблону».

Другий розділ присвячений порівнянню динамічного та статичного поліморфізму на основі прикладів реальних програм. Розглянуто як порівняння за якістю коду, так і за ефективністю його виконання.

Третій розділ присвячений дослідженню використання патернів реалізації статичного поліморфізму у проєктах з відкритими джерелами.

РОЗДІЛ 1. СТАТИЧНИЙ ПОЛІМОРФІЗМ У С++

1.1. Поліморфізм у С++

Поліморфізм (з грецької «багато форм») – фундаментальне поняття в об’єктно-орієнтованому програмуванні. Поліморфізм дозволяє працювати з об’єктами різних типів як з об’єктами спільного супертипу, що надає єдиний інтерфейс, водночас зберігаючи різницю в деталях реалізації. Це надає розробникам створювати більш гнучкий та легкий для повторного використання код.

У контексті С++ про поліморфізм найчастіше говорять, маючи на увазі динамічний поліморфізм. Динамічний поліморфізм – це різновид поліморфізму, що дозволяє визначати метод, який необхідно викликати, на етапі виконання програми. У С++ такий різновид поліморфізму досягається за допомогою успадкування та віртуальних функцій. Коли віртуальна функція викликається, тип об’єкта, на якому робиться виклик, використовується для визначення версії функції, до якої необхідно перейти.

Така поведінка стає можливою завдяки механізму віртуальної таблиці. Кожен клас, який має принаймні одну віртуальну функцію, має власну віртуальну таблицю. Віртуальна таблиця – це масив вказівників на віртуальні функції, визначені у класі. При створенні екземпляра класу, який містить віртуальні функції, до цього об’єкта додається «віртуальний вказівник» (vpointer), який відсилається до віртуальної таблиці. При виклику віртуальної функції програма не робить прямого переходу до конкретної реалізації. Натомість програма йде за віртуальним вказівником до віртуальної таблиці. Наступним етапом є знаходження правильного вказівника у масиві віртуальної таблиці. Це робиться за допомогою «відступу», який залежить від порядку визначення методів у класі та стає відомим на етапі компіляції.

Отже, віртуальні функції надають розробникам збільшену зручність та гнучкість під час роботи над кодом. Проте вони мають один суттєвий недолік – реалізація поліморфічної поведінки потребує додаткових операцій на етапі виконання коду, що може мати негативний вплив на ефективність програми.

1.2. Статичний поліморфізм

Термін «статичний поліморфізм» в науковій літературі про C++ інтерпретується по різному. У широкому сенсі, статичний поліморфізм – це здатність однієї сутності (функції або оператора) працювати з об’єктами різних типів на етапі компіляції, що надає більшу ефективність та більш строгу типізацію. Основні категорії концепцій C++, які вважають прикладами статичного поліморфізму:

- Перевантаження функцій
- Перевантаження операторів
- Шаблони – одним із найбільш потужних проявів статичного поліморфізму в C++ є використання шаблонів, які дозволяють функціям та класам оперувати узагальненими типами. Це дозволяє створювати легкі для повторного використання компоненти, водночас зберігаючи строгу типізацію та ефективність.

Окрім традиційного визначення, статичний поліморфізм може бути інтерпретований у ширшому значенні як засіб для імітації динамічного поліморфізму на етапі компіляції. Це ширше значення стосується не стільки визначення варіанту функції, який необхідно виконати, на основі типів аргументів, скільки виклику коректного методу для кожного дочірнього типу, подібно до поведінки, яку забезпечує динамічний поліморфізм. У науковій літературі запропоновано декілька можливих підходів та патернів для реалізації статичного поліморфізму, зокрема приклад, наведений Дж. Коплієном [1].

1.3. Патерн «дивно рекурсивного шаблону»

Патерн «дивно рекурсивного шаблону» (англ. Curiously Recurring Template Pattern) - потужний та унікальний прийом у C++, що поєднує у собі успадкування та шаблони. Цей прийом полягає у створенні дочірніх класів, які успадковують параметризований батьківський клас і передають самі себе в якості параметра типу. Термін «Curiously Recurring Template Pattern» був запроваджений Дж. Коплієном у його статті 1995 року [1].

Розглянемо базовий приклад патерну.

```
template <typename Derived>
class Base {
public:
  void someFunction() {
    static_cast<Derived*>(this)->someFunctionImpl();
  }
};

class Derived1 : public Base<Derived1> {
public:
  void someFunctionImpl() {
    // Деталі реалізації
  }
};
```

У цьому прикладі клас Base є параметризованим і приймає тип Derived як аргумент шаблону. Клас Derived1 успадковується від класу Base та одночасно передає себе як параметр шаблону у клас Base. Це створює нібито кругову залежність, де батьківський клас залежить від дочірнього, і навпаки.

Така кругова залежність можлива внаслідок того, як компілятори C++ заповнюють шаблони. По-перше, шаблони в C++ заповнюються ліниво – компілятор заповнює тільки ті частини шаблону, які дійсно використовуються, замість того, щоб інстанціювати весь шаблонний клас. Таким чином, компілятор може уникнути нескінченної рекурсії, за умови, що батьківський клас не залежить від розміру та структури класу, переданого в шаблоні. По-друге, C++ припускає неповні типи, тобто такі типи, які було оголошено, але не визначено. Такі типи дозволені у деяких контекстах, як от

в оголошенні вказівника або відсилки. У патерні «дивно рекурсивного шаблону» аргумент шаблону, який представляє дочірній клас, є неповним типом, оскільки він є повністю визначеним на момент оголошення шаблону. Проте, батьківський клас має потребу лише в інформації про тип, але не у повному визначенні.

Синтаксис *static_cast* є важливою частиною патерну «дивно рекурсивного шаблону», бо дозволяє батьківському класу викликати методи дочірніх без використання віртуальних функцій та віртуальної таблиці. *static_cast* також дозволяє пересвідчитись, що дочірній клас реалізує метод, що викликається в батьківському, оскільки відсутність або неправильна сигнатура цього методу призведе до помилки компіляції.

Зразок «дивно рекурсивного шаблону» був використаний В.В. Бубликом у статті 2021 року [2] для реалізації подвійної диспетчеризації для двох типів представлення комплексних чисел: класи AComplex та TComplex визначені, як нащадки узагальненого класу Base, який приймає тип дочірнього класу як параметр шаблона. Застосування статичного поліморфізму дозволило уникнути порушення принципів SOLID, яке виникало при спробах реалізації ієрархії комплексних чисел з використанням віртуальних методів.

РОЗДІЛ 2. ПОРІВНЯННЯ ДИНАМІЧНОГО ТА СТАТИЧНОГО ПОЛІМОРФІЗМУ

2.1. Міксіни

Одним із найбільш потужних застосувань патерну «дивно рекурсивного шаблону» є реалізація «міксінів» (англ. “MixIn”) [3]. «Міксін» - це клас, який містить в собі певну функціональність, яка призначена для використання у класах-нащадках. Метою такого дизайну є визначення багатьох ізольованих класів, кожен з яких відповідає за єдину задачу, для подальшого «збирання» функціональності у класі-клієнті.

Традиційно в об’єктно-орієнтованому програмуванні на C++ поліморфізм досягається з допомогою успадкування та віртуальних функцій. Розглянемо приклад сутностей в ігровій розробці. Поставимо за мету визначення певної базової сутності, інтерфейс якої потребує реалізації логіки рендерингу та фізики:

```
class Entity {
public:
  virtual void render() = 0;
  virtual void updatePhysics() = 0;
};
```

Успадковуючи даний клас, можна створювати різні реалізації інтерфейсу «сутність»:

```
class Player : public Entity {
public:
  void render() override {
  }

  void updatePhysics() override {
  }
};

class Enemy : public Entity {
public:
  void render() override {
  }

  void updatePhysics() override {
  }
};
```

Цей підхід має переваги на початковому етапі розробки, коли інтерфейс сутності має невелику кількість методів. Проте він має суттєвий недолік – клас Entity має «знати» про всі можливі функції, яких будь-яка сутність у грі буде потребувати у майбутньому. Це може призвести до роздутого інтерфейсу Entity, що порушує принцип єдиної відповідальності (Single Responsibility Principle) [4].

Патерн «дивно рекурсивного шаблону» надає змогу досягнути поліморфічної поведінки без використання віртуальних методів, а також без недоліків роздутого базового класу. Використання цього патерну дозволяє створювати «міксіни», або зручні для повторного використання блоки функціональності, які можна додавати до будь-яких класів.

```
template <typename T>
class Renderable {
public:
  void render() {
    static_cast<T*>(this)->renderImplementation();
  }
};

template <typename T>
class Physical {
public:
  void updatePhysics() {
    static_cast<T*>(this)->physicsImplementation();
  }
};
```

У даному прикладі інтерфейс сутності розбито на логічно відокремлені частини – клас Renderable, який відповідає за зображення сутності на екрані, та клас Physical, який відповідає за фізику сутності у грі. Ці класи можна повторно використовувати у багатьох різних конкретних класах, при цьому кожен конкретний клас може обирати тільки ті «міксіни», які є актуальними для нього.

```
class Player : public Renderable<Player>, public Physical<Player> {
public:
  void renderImplementation() {
  }
```

```
  void physicsImplementation() {
  }
};

class Enemy : public Renderable<Enemy>, public Physical<Enemy> {
public:
  void renderImplementation() {
  }

  void physicsImplementation() {
  }
};
```

Класи Player та Enemy успадковують визначені раніше міксіни за допомогою патерну «дивно рекурсивного шаблону». Конкретні реалізації мають бути визначені у методах renderImplementation та physicsImplementation.

Використання «міксінів» у поєднанні зі статичним поліморфізмом має так переваги:

- Гнучкість – розробник може легко модифікувати функціональність конкретного класу, додаючи або прибираючи необхідні «міксіни» зі списку успадкованих класів.
- Продуктивність – патерн «дивно рекурсивного шаблону» допомагає реалізувати поліморфізм на етапі компіляції та уникнути додаткових операцій, до яких призводить використання віртуальних функцій.
- Роз’єднання (англ. decoupling) – кожен «міксін» спеціалізується на окремому функціоналі. Як наслідок, зберігається один із принцип SOLID – принцип єдиної відповідальності. Це робить код легшим для розуміння та підтримки.

2.2. «Ланцюжок» викликів

Розглянемо приклад застосування патерну Builder «банди чотирьох» [5]. У класичному варіанті, патерн Builder передбачає створення класу,

методи якого повертають екземпляр цього ж класу. Це дозволяє створювати «ланцюг» викликів методів цього класу. Спробуємо визначити примітивний приклад реалізації класу SelectQueryBuilder.

```
class DynamicSelectQueryBuilder {
public:
  DynamicSelectQueryBuilder &Select(const std::string &field) {
    m_fields.push_back(field);
    return *this;
  }

  DynamicSelectQueryBuilder &From(const std::string &table) {
    m_table = table;
    return *this;
  }

  DynamicSelectQueryBuilder &Where(const std::string &condition) {
    m_conditions.push_back(condition);
    return *this;
  }

  virtual std::string Build() const = 0;

protected:
  std::vector<std::string> m_fields;
  std::string m_table;
  std::vector<std::string> m_conditions;
};
```

Цей клас є абстрактним завдяки чисто віртуального методу Build, отже неможливо створити екземпляр даного класу. Аби мати змогу використати SelectQueryBuilder, необхідно створити дочірній клас, наприклад, MysqlSelectQueryBuilder.

```
class DynamicMySQLSelectQueryBuilder : public DynamicSelectQueryBuilder {
public:
  std::string Build() const override {
    std::ostringstream oss;
    oss << "SELECT ";
    for (size_t i = 0; i < m_fields.size(); ++i) {
      if (i != 0) oss << ", ";
      oss << m_fields[i];
    }
    oss << " FROM " << m_table;
    for (const auto &condition: m_conditions) {
      oss << " WHERE " << condition;
    }
    oss << ";"; // MySQL requires a semicolon at the end of the query
    return oss.str();
  }
};
```

Тепер ми можемо створювати запити за допомогою MySQLSelectQueryBuilder:

```
DynamicMySQLSelectQueryBuilder().Select("name").From("users").Where("age > 20").Build()
```

Тепер уявімо, що нам необхідно додати метод у дочірній клас.

```
DynamicMySQLSelectQueryBuilder &SomeChildSpecificMethod() {
    return *this;
}
```

Виникає проблема при наслідуванні базового класу – втрачається можливість створювати «ланцюг» із методами, визначеними в дочірніх класах, адже за сигнатурами методи базового класу повертають тип саме цього класу, а не дочірнього [6].

Розглянемо, як можна застосувати патерн «дивно рекурсивного шаблону» у цьому випадку. Нехай базовий клас SelectQueryBuilder матиме шаблон, що приймає параметром шаблона дочірній клас. Тепер можемо визначити методи, як такі, що повертають об’єкт типу дочірнього класу.

```
template<typename Derived>
class StaticSelectQueryBuilder {
public:
  Derived &Select(const std::string &field) {
    m_fields.push_back(field);
    return static_cast<Derived &>(*this);
  }

  Derived &From(const std::string &table) {
    m_table = table;
    return static_cast<Derived &>(*this);
  }

  Derived &Where(const std::string &condition) {
    m_conditions.push_back(condition);
    return static_cast<Derived &>(*this);
  }

  std::string Build() const {
    return static_cast<const Derived &>(*this).BuildImpl();
  }

protected:
  std::vector<std::string> m_fields;
  std::string m_table;
  std::vector<std::string> m_conditions;
};
```

Тепер дочірній клас можна визначити таким чином:

```
class StaticMySQLSelectQueryBuilder : public
StaticSelectQueryBuilder<StaticMySQLSelectQueryBuilder> {
public:
  std::string BuildImpl() const {
    std::ostringstream oss;
    oss << "SELECT ";
    for (size_t i = 0; i < m_fields.size(); ++i) {
      if (i != 0) oss << ", ";
      oss << m_fields[i];
    }
    oss << " FROM " << m_table;
    for (const auto &condition: m_conditions) {
      oss << " WHERE " << condition;
    }
    oss << ";";
    return oss.str();
  }
};
```

Для використання об’єктів різних дочірніх класів визначено узагальнену функцію, яка може працювати з будь-якими нащадками SelectQueryBuilder:

```
template<typename T>
std::string BuildSampleQuery(StaticSelectQueryBuilder<T> &builder) {
  return builder.Select("name")
          .From("users")
          .Where("age > 20")
          .Build();
}
```

Як наслідок, показано, що використання патерну «дивно рекурсивного шаблону» одночасно надає змогу створювати легкі для наслідування та розширення класи, водночас уникаючи використання віртуальних функцій.

2.3. Порівняння ефективності виконання

Основною перевагою статичного поліморфізму над динамічним є швидкість виконання. Заради порівняння ефективності виконання визначено приклад з використанням бібліотеки benchmark та сайту https://quick-bench.com/:

```cpp
#include <benchmark/benchmark.h>

class DynamicBase {
public:
  virtual void increment() = 0;
};

class DynamicDerived : public DynamicBase {
private:
  int value = 0;
public:
  void increment() override {
    value += 1;
  }
};

template<typename Derived>
class StaticBase {
public:
  void increment() {
    static_cast<Derived *>(this)->incrementImpl();
  }
};

class StaticDerived : public StaticBase<StaticDerived> {
private:
  int value = 0;
public:
  void incrementImpl() {
    value += 1;
  }
};

int N = 40000;

static void BM_Dynamic(benchmark::State& state) {
  DynamicBase* obj = new DynamicDerived();
  unsigned i = 0;

  for (auto _: state) {
    for (unsigned j = 0; j < i; j++) {
      obj->increment();
    }
    i++;
  }
}

BENCHMARK(BM_Dynamic)->Iterations(N);


static void BM_Static(benchmark::State& state) {
  StaticBase<StaticDerived>* obj = new StaticDerived();

  unsigned i = 0;

  for (auto _: state) {
    for (unsigned j = 0; j < i; j++) {
      obj->increment();
    }
    i++;
  }
}
```

```
BENCHMARK(BM_Static)->Iterations(N);
```

При опції компілятора –O0 (відсутня оптимізація) варіант з динамічним поліморфізмом виявляється швидшим:

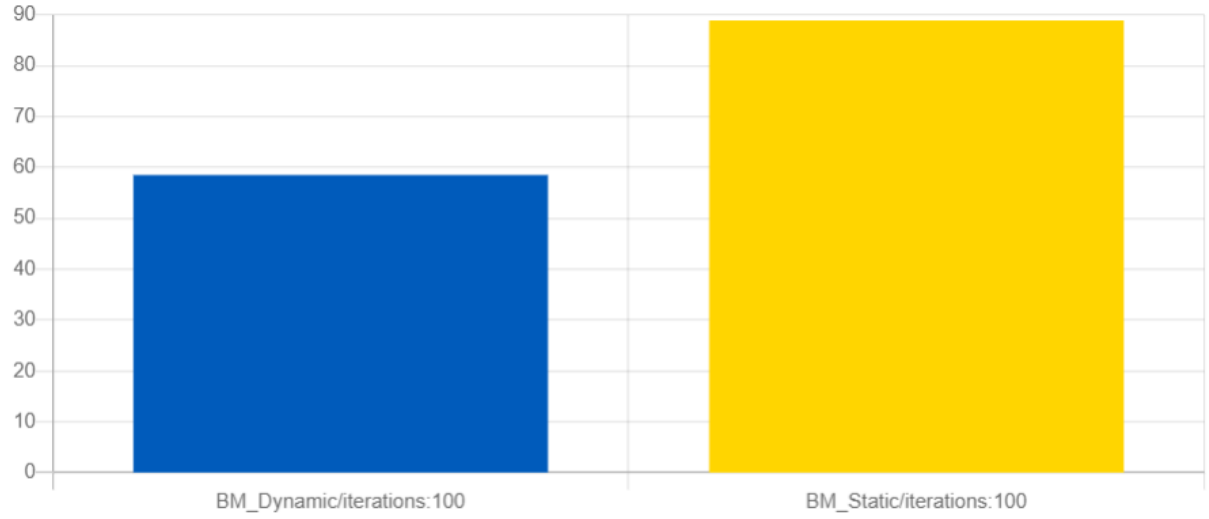

рис 1. Порівняння швидкості виконання прикладів коду із застосуванням динамічного та статичного поліморфізму за відсутності оптимізацій

Проте при рівні оптимізації -O3 варіант зі статичним поліморфізмом виконується швидше у 6 разів:

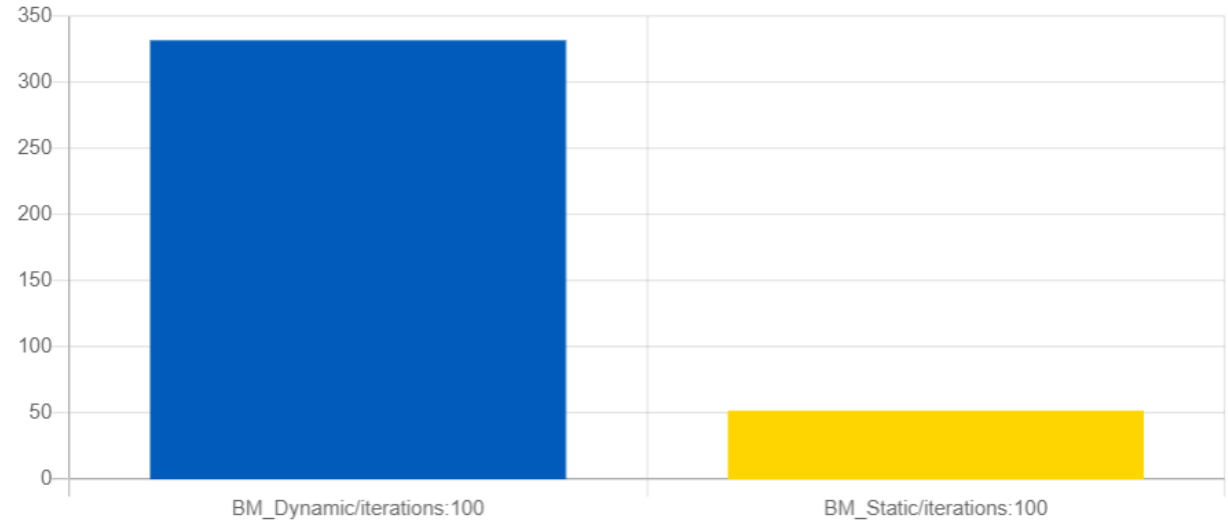

рис 2. . Порівняння швидкості виконання прикладів коду із застосуванням динамічного та статичного поліморфізму з рівнем оптимізації -O3

Такі результати пов’язані із тим, що за відсутності оптимізації у версії зі статичним поліморфізмом відбувається більше викликів методів (спочатку

методу базового класу, а вже потім дочірнього). У випадку застосування оптимізації у статичній реалізації компілятор може застосовувати inlining [7], що і призводить до швидших результатів.

В результаті експерименту встановлено, що при застосуванні оптимізацій компілятора статична реалізація є швидшою за часом виконання ніж відповідна динамічна реалізація.

РОЗДІЛ 3. ЗАСТОСУВАННЯ СТАТИЧНОГО ПОЛІМОРФІЗМУ В ПРОЄКТАХ З ВІДКРИТИМИ ДЖЕРЕЛАМИ

3.1. Бібліотека Protocol Buffers

Бібліотека Protocol Buffers, створена компанією Google, є прикладом високоефективного та гнучкого інструменту для серіалізації структурованих даних. У коді бібліотеки використовується патерн «дивно рекурсивного шаблону» для досягнення поліморфізму на етапі компіляції.

Спершу визначено базовий клас MapEntryImpl [8]:

```
template <typename Derived, typename Base, typename Key, typename Value,
          WireFormatLite::FieldType kKeyFieldType,
          WireFormatLite::FieldType kValueFieldType>
class MapEntryImpl : public Base { /*...*/ };
```

Цей клас використовується для реалізації серіалізації та десеріалізації записів структури даних «мапа». Використовуючи параметр шаблону Derived, клас MapEntryImpl має можливість визначати правильні виклики методів дочірнього класу на етапі компіляції:

```
if (!Derived::ValidateKey(key)) return nullptr;
```

Це має особливе значення в контексті бібліотеки, адже бібліотека Protocol Buffers заточена під високу продуктивність, і додаткові інструкції для визначення правильної реалізації методу, яку необхідно викликати, могли б спричинити затримки.

Слід зазначити, що цей приклад є нестандартним, адже базовий клас MapEntryImpl викликає статичний метод дочірнього класу. Попри це, таке використання можна вважати прикладом статичного поліморфізму, оскільки поведінка методу батьківського класу залежить від специфічної реалізації статичного методу у дочірньому класі.

3.2. Бібліотека Folly

Бібліотека Folly (Facebook Open-source Library) – колекція утиліт для мови C++ розроблена у Facebook. Бібліотека широко використовується завдяки її заточеним під високу ефективність структурам даних.

У коді бібліотеки є цінний приклад використання патерну «дивно рекурсивного шаблону» для досягнення статичного поліморфізму. Замість традиційного патерну, у бібліотеці спочатку визначена допоміжна структура FBounded [9], подібно до прикладу наведеного у статті Дж. Боккари [10]:

```
template <class Self>
struct FBounded {
  using SelfType = Self;
  const Self& self() const { return *static_cast<const Self*>(this); }

  Self& self() { return *static_cast<Self*>(this); }
};
```

Метод self() використовує static_cast для того, аби конвертувати this в тип класу-нащадка.

Далі бібліотека використовує цю допоміжну структуру для визначення класів, які відповідають базовому класу у патерні «дивно рекурсивного шаблону».

```
template <class Self>
class Operator : public FBounded<Self> { /*...*/ };
```

У класі задекларований метод compose:

```
template <class Source, class Value, class ResultGen = void>
ResultGen compose(const GenImpl<Value, Source>& source) const;
```

Цей метод має бути реалізований у дочірніх класах.

Пізніше клас Operator успадковують різні конкретні класи, що відповідають певним операціям.

```
template <class Predicate>
class Map : public Operator<Map<Predicate>> { /*...*/ };
```

Кожен із дочірніх класів Operator реалізує метод compose подібним чином:

```
template <
    class Source,
```

```
    class Gen =
        Generator<typename Source::ValueType, typename Source::SelfType>>
Gen compose(Source source) const {
  return Gen(std::move(source.self()), pred_);
}
```

У якості параметра шаблону передається будь-який інший клас, який успадковує FBounded, і в реалізації методу маємо можливість отримати доступ до екземпляра власне дочірнього класу за допомогою метода self().

ВИСНОВКИ

У курсовій роботі розглянуто різні інтерпретації поняття статичного поліморфізму та можливості їх реалізації мовою C++. У результаті дослідження було порівняно динамічний та статичний поліморфізм на прикладах коду. Показано недоліки динамічного поліморфізму та застосування патернів реалізації статичного поліморфізму, зокрема патерну «дивно рекурсивного шаблону», для покращення якості та ефективності коду. Також проаналізовано застосування статичного поліморфізму у бібліотеках з відкритими джерелами. На основі дослідження можна зробити висновок, що статичний поліморфізм, за умови доречного використання, може нести із собою численні переваги, такі як: безпечна типізація, підвищена продуктивність та гнучкість коду.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Coplien, James O. (1995). Curiously Recurring Template Patterns. C++ Report
2. Бублик В. В. До питання створення статичного патерну проєктування для подвійної диспетчеризації модельних сигнатур / В. В. Бублик // Наукові записки НаУКМА. – 2021. – Т. 4 (2021). Комп'ютерні науки.
3. Boccara J. What the Curiously Recurring Template Pattern can bring to your code [Електронний ресурс] / Jonathan Boccara. – 2017. – Режим доступу до ресурсу: https://www.fluentcpp.com/2017/05/16/what-the-crtp-brings-to-code/
4. Martin, Robert C. Agile Software Development, Principles, Patterns, and Practices. Pearson Education, 2002.
5. Design Patterns: Elements of Reusable Object-Oriented Software / E. Gamma, R. Helm, R. Johnson, and J. Vlissides. – 1994. – Addison-Wesley Professional.
6. Arena M. Use CRTP for polymorphic chaining [Електронний ресурс] / Marco Arena. – 2012. – Режим доступу до ресурсу: https://marcoarena.wordpress.com/2012/04/29/use-crtp-for-polymorphic-chaining/
7. Bendersky E. The cost of dynamic (virtual calls) vs. static (CRTP) dispatch in C++ [Електронний ресурс] / Eli Bendersky. – 2013. – Режим доступу до ресурсу: https://eli.thegreenplace.net/2013/12/05/the-cost-of-dynamic-virtual-calls-vs-static-crtp-dispatch-in-c
8. Google. protobuf [Електронний ресурс]. – Режим доступу до ресурсу: https://github.com/protocolbuffers/protobuf
9. Facebook. folly [Електронний ресурс]. – Режим доступу до ресурсу: https://github.com/facebook/folly
10. Boccara J. An Implementation Helper For The Curiously Recurring Template Pattern [Електронний ресурс] / Jonathan Boccara. – 2017. –

Режим доступу до ресурсу: https://www.fluentcpp.com/2017/05/19/crtp-helper/

ДОДАТОК

Код проекту додано в архіві.