КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра інтелектуальних програмних систем

Кваліфікаційна робота

на здобуття ступеня бакалавра

За спеціальністю 121 Інженерія програмного забезпечення
на тему:

Реалізація python-подібних декораторів в C++ шляхом розширення компілятора clang

Виконав студент 4-го курсу
Ілля ГАЙДУЧЕНКО

(підпис)

Науковий керівник:
асистент, кандидат фізико-математичних наук
Констянтин ЖЕРЕБ

(підпис)

Засвідчую, що в цій кваліфікаційній роботі
немає запозичень з праць інших авторів
без відповідних посилань

Студент

(підпис)

Роботу розглянуто й допущено до захисту
на засіданні кафедри інтелектуальних
програмних систем

25 травня 2022 р.,
протокол № 10
Завідувач кафедри
О. ПРОВОТАР

(підпис)

Київ — 2022

РЕФЕРАТ

Робота містить 26 сторінок, 1 ілюстрація, 6 використаних джерел, 1 додаток.

Ключові слова: КОМПІЛЯТОР, ДЕКОРАТОР, РОЗШИРЕННЯ МОВИ, АТРИБУТИ, C++, CLANG, LLVM.

Стислий опис роботи

Об'єктом розроблення є реалізація декораторів для функцій в мові C++ шляхом написання розширення для компілятора Clang. **Мета роботи** — покращення зручності написання коду на мові C++ завдяки розробленому в цій роботі розширенню для компілятора та ознайомитись з програмним кодом Clang. Результати цієї роботи мають **новизну**, тому що не існує популярних розширень, які реалізують декоратори для компілятора Clang або будь-якого існого компілятора. **Сфера застосування** є дуже широкою, адже C++ є однією з найпопулярніших мов програмування в світі і більшість досвідчених користувачів C++ очікують нових функціональних розширень мови. Шляхів подальшого **розвитку** роботи є багато: доцільно збільшувати кількості елементів мови, для яких можна застовувати декоратори (класи, змінні, константи тощо) або покращити інформативності помилок при неправильному використанні декораторів.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ 4

ВСТУП 5

РОЗДІЛ 1. ОГЛЯД ДЕКОРАТОРІВ В PYTHON 7

1.1 Визначення декоратора функції . . . 7

1.2 Мотивація додання нового синтаксису для декорування . . . 8

1.3 Огляд додаткових можливостей синтаксису декораторів . . . 9

1.3.1 Декоратори функцій з аргументами . . . 9

1.3.2 Застовування декількох декораторів до функції . . . 9

РОЗДІЛ 2. ОГЛЯД ПРОПОНОВАНОГО СИНТАКСИСУ ДЕКОРАТОРІВ ДЛЯ C++ 10

2.1 Атрибути в C++ . . . 10

2.2 Синтаксис та обмеження атрибуту decorate . . . 11

2.3 Семантика атрибуту decorate . . . 12

РОЗДІЛ 3. ВИКОРИСТАНІ ТЕХНОЛОГІЇ ДЛЯ НАПИСАННЯ РОЗШИРЕННЯ КОМПІЛЯТОРА 14

3.1 Вибір компілятора . . . 14

3.2 Методи додання нової функціональності в Clang . . . 14

3.3 Мова програмування та середовище розробки . . . 14

РОЗДІЛ 4. ВНУТРІШНЯ БУДОВА ТА ПРОЦЕС НАПИСАННЯ РОЗШИРЕННЯ 16

4.1 Реєстрація плагіну в регістрі плагінів Clang . . . 16

4.2 Додання нового атрибуту . . . 16

4.3 Пошук функцій з атрибутом в синтаксичному дереві . . . 16

4.4 Генерація коду . . . 18

4.5 Перекомпіляція . . . 19

ВИСНОВКИ 20

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 21

ДОДАТОК А 22

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

Патерн проектування	—	типовий спосіб вирішення певної проблеми, що часто зустрічається при проектуванні архітектури програм
Плагін	—	це динамічна бібліотека, яка завантажується програмою під час виконання
Парсинг	—	процес аналізу вхідної послідовності символів, з метою розбору граматичної структури згідно із заданою формальною граматикою. Також називають синтаксичним аналізом
AST	—	абстрактне синтаксичне дерево
Буфер	—	частина оперативної пам'яті, призначена для тимчасового зберігання даних
Одиниця трансляції	—	одиниця компіляції в C++. Вона складається зі вмісту одного вихідного файлу, а також вмісту будь-яких заголовних файлів, прямо чи опосередковано включених до нього

ВСТУП

Декоратори в мові Python надають зручний спосіб додавати нову поведінку до існуючих функцій та класів. В багатьох інших популярних мовах програмування також існують синтаксичні можливості зручно додавати функціональність до сутностей в коді. Наприклад, так звані анотації в мові Java дають можливість додавати метадані до сутностей, за допомогою яких можна змінити або доповнити поведінку цих сутностей під час виконання програми.

Актуальність роботи та підстави для її виконання В мові C++ на данний момент часу недостатньо подібних синтаксичних можливостей в порівнянні з популярними мовами програмування. Так, цілком можливо досягнути такої ж поведінки програми стандартними засобами, яка досягається в Python за допомогою декораторів, але це не так зручно та лаконічно, як це робиться в Python та інших мовах програмування. Також в стандартній бібліотеці Python присутні багато стандартних декораторів, які часто вживаються при вирішенні абсолютно різноманітних задач, на відміну від C++, де більшість простих речей реалізуються не так швидко і лаконічно, як в Python.

Отже, додання подібного синтаксису в мову C++ є актуальним.

Метою роботи є розробка плагіну для компілятора Clang, який би додав можливість компілювати код, в якому використовується синтаксис, схожий до синтаксису декораторів в Python. Поведінка програми, яка використовує цей синтаксис і яка була скомпільована за допомогою розробленого плагіну, має бути подібною до аналогічного коду на мові Python.

Об'єкт, методи й засоби розроблення Об'єктом розробки плагіну є підтримка нового синтаксису декораторів в мові C++.

Розробці розширення передувала розробка та формалізація синтаксису декораторів, що опирається на існуючий синтаксис атрибутів в мові C++.

Для реалізації плагіну було використано Clang API, компілятор Clang та текстовий редактор VS Code. В якості мови програмування було обрано C++, тому що це єдина мова, що може бути використана для написання плагіну для компілятора Clang.

Можливі сфери застосування Об'єкт розробки плагіну охоплює всі сфери програмування на C++, адже декоратори використовуються в різноманітних програмах на Python, як і їх аналоги в інших мовах програмування. Тому цей плагін може бути використаний розробником, що використовує мову C++ в програмах будь-якої предметної області.

РОЗДІЛ 1. ОГЛЯД ДЕКОРАТОРІВ В PYTHON

Перед тим, як розпочати роботу по реалізації синтаксису Python-подібних декораторів для C++, було проведено огляд синтаксису та семантики декораторів в Python, вивчення випадків їх використання та існуючих функцій-декораторів в стандартній бібліотеці Python та сторонніх бібліотеках.

1.1 Визначення декоратора функції

Декоратор – це функція, яка приймає як аргумент іншу функцію(функція, яку декорують) та повертає нову функцію, яка доповнює поведінку функції, яку декорують. Інакше кажучи, декоратор «декорує» ту функцію, яку йому передали як аргумент.

Нижче наведено приклад декоратора, який «декорує» функцію, заміряючи її час виконання та виводить його в стандартний вивід:

```
1  from datetime import datetime
2
3  def measure_time(func):
4      def inner():
5          start_time = datetime.now()
6          func()
7          finish_time = datetime.now()
8          print("function executed in {}".format(finish_time -
    ↪  start_time))
9      return inner
10
11 def print_hello_world():
12     print("Hello, world!")
13
14 print_hello_world = measure_time(print_hello_world)
15
16 print_hello_world()
```

В цьому коді функція *measure_ time* «декорує» функцію *print_ hello_ world*

в рядку 14, додаючи після неї вивід часу її виконання. Під час виконання коду програма виконала такий вивід:

```
Hello, world!
function executed in 0:00:00.000038
```

Вивід рядка 'Hello, world!' — це поведінка, визначена функцією *print_hello_world*, а рядок 'function executed in 0:00:00.000038' вивівся через те, що функція декорована декоратором *measure_time*.

1.2 Мотивація додання нового синтаксису для декорування

Метод застосування перетворення до функції, що описаний в попередньому підрозділі розміщує фактичне перетворення функції після її тіла. Для великих функцій це відокремлює ключовий компонент поведінки функції від визначення решти зовнішнього інтерфейсу функції.

```
def print_hello_world():
    print("Hello, world!")

print_hello_world = measure_time(print_hello_world)
```

Рішення цієї проблеми полягає в тому, щоб перемістити трансформацію методу ближче до власного оголошення методу. Метою нового синтаксису, доданого в версія Python 2.4 є заміна цього коду:

```
def print_hello_world():
    print("Hello, world!")

print_hello_world = measure_time(print_hello_world)
```

альтернативою, яка розміщує декоратор в оголошенні функції:

```
@measure_time
def print_hello_world():
    print("Hello, world!")
```

Ці два приклади коду є ідентичними по поведінці. Новий синтаксис є консенсусом спільноти Python-розробників[1] і є однією з самих зручних та використовуваних можливостей мови Python.

1.3 Огляд додаткових можливостей синтаксису декораторів

1.3.1 Декоратори функцій з аргументами

Декоратори, як і звичайні функції, можуть приймати аргументи. Приклад:

```
@repeat(5)
def print_hello_world():
    print("Hello, world!")
```

Декоратор *repeat* перетворює функцію в функцію, яка виконує її n раз, де n — аргумент декоратора *repeat*. Як результат, функція *print_hello_world* виведе рядок "Hello, world!"в стандартний вивід 5 разів.

1.3.2 Застовування декількох декораторів до функції

```
@measure_time
@repeat(5)
def print_hello_world():
    print("Hello, world!")
```

В цьому прикладі до функції *print_hello_world* будуть застосовуються декоратори в порядку від останнього написаного до першого. Тобто спочатку буде застосовано декоратор *repeat* і після цього *measure_time*. Виклик функції *print_hello_world* спричинить такий вивід:

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
function executed in 0:00:00.000045
```

РОЗДІЛ 2. ОГЛЯД ПРОПОНОВАНОГО СИНТАКСИСУ ДЕКОРАТОРІВ ДЛЯ C++

Для реалізації синтаксису декораторів в C++ було вирішено використати атрибути.

2.1 Атрибути в C++

Атрибути забезпечують уніфікований стандартний синтаксис для визначених реалізацією мовних розширень. Атрибути можна використовувати майже скрізь у C++ програмі і їх можна застосовувати майже до всього: до типів, до змінних, до функцій, до блоків коду[3]. Реалізація конкретного атрибуту визначає до який сутностей в коді його можна застосовувати.

Синтаксис використання атрибуту:

```
[[attr]]
```

Синтаксис використання декількох атрибутів:

```
[[attr1, attr2, attr3]]
```

Синтаксис використання атрибуту з аргументом:

```
[[attr(arg)]]
```

Синтаксис використання атрибуту, який знаходиться в просторі імен (англ. *namespace*):

```
[[namespace::attr]]
```

attr тут – назва самого атрибута. Поведінка атрибуту визначається конкретною реалізацією в компіляторі, або зафіксована в стандарті мови C++, якщо цей атрибут стандартний. Наприклад, стандартний атрибут *nodiscard* застосовується до функцій. *nodiscard* додає наступну поведінку: якщо ця функція повертає значення і результат функції з цим атрибутом ігнорується, то компілятор видає попередження під час компіляції. Код, що використовує цей атрибут:

```
[[nodiscard]]
int foo() {
    return 5;
}
int main() {
    foo();
}
```

В функції *main* результат функції *foo*, до якої застосовується атрибут *nodiscard*, ігнорується, тому компілятор показує попередження під час компіляції цього коду:

```
warning: ignoring return value of 'int foo()', declared with attribute '
    nodiscard'
```

Отже, стандарт C++ надає можливість розробникам компіляторів реалізовувати нову функціональність шляхом реалізації нових атрибутів. Це значно спрощує реалізацію нових можливостей мови в компіляторі, тому що парсингом атрибутів займається компілятор. Якщо при розробці розширень мови піти іншим шляхом – додавати нові синтаксичні елементи, не використовуючи атрибути – розробнику потрібно буде додавати нові правила до лексера та парсера програмного коду. Також вагомою перевагою реалізації атрибутів є те, що це мінімізує конфлікти та помилки при використанні декількох розширень компілятора одночасно.

Самі розробників компіляторів частіше надають перевагу реалізації нових атрибутів для розширення можливостей мови. Так, компілятор Clang містить близько ста нестандартних атрибутів, більшість з яких можна використовувати для оптимізації програмного коду[5].

Беручи до уваги всі ці причини, було вирішено реалізувати синтаксис декораторів для функцій шляхом написання нового атрибуту *decorate*.

2.2 Синтаксис та обмеження атрибуту decorate

Розроблений атрибут *decorate* може застосовуватись до функцій та при-

ймає від 1 аргумента. Ці аргументи є декораторами для функції, до якої застосувається атрибут, тому на ці аргументи накладаються деякі обмеження:

– аргумент повинен бути функтором — об'єктом, в якого можна викликати оператор «()» або вказівником на функцію;

– він повинен приймати один об'язковий аргумент. Цим аргументом буде функція, яку декорують;

– виклик оператора «()» повинен повертати функтор;

– оператор «()» повинен мати «constexpr» специфікацію, тобто мати можливість бути викликаним під час компіляції, адже декорування відбувається під час компіляції.

Також є обмеження для функції, яку декорують (для якої застосовується атрибут *decorate*):

– функція не може бути чисто віртуальною (англ. *pure virtual function*), тому що в неї немає тіла, яке можна декорувати;

– функція не може бути явно видаленою (англ. *explicitly deleted function*);

– атрибут можна застосувати тільки до визначення функції;

– функція не може мати тип компонування «extern "C"».

В випадку порушення будь-якого правила, компілятор виведе помилку компіляції з детальним описом.

2.3 Семантика атрибуту decorate

Атрибут, застосовуючись до функції перетворює її в результат виклику декоратора з цієї функцією як аргументом. Нижче наведений код з вимірювання часу виконання функції *print_hello_world*, переписаний на C++ з використанням декоратора *decorate*:

```
template <typename Func>
constexpr auto measure_time(Func&& func)
{
    return [func = std::forward<Func>(func)] ()
    {
        const auto start = std::chrono::system_clock::now();
        func();
        const auto finish = std::chrono::system_clock::now();
        const auto duration =
        ↪  std::chrono::duration_cast<std::chrono::nanoseconds>(finish
        ↪  - start);
        std::cout << "function executed in " << duration.count() << "
        ↪  ns\n";
    };
}
[[decorate(measure_time)]]
void print_hello_world()
{
    std::cout << "Hello, world!" << std::endl;
}
```

Вивід функції *print_hello_world*:

```
Hello, world!
function executed in 38704 ns
```

Реалізований атрибут *decorate* дозволяє писати код, подібний до коду на Python з використанням декораторів. Цей атрибут підтримує і декоратори з аргументами, і застосування декількох декораторів до функції. На відміну від декораторів в Python, декорування функції відбувається під час компіляції. Завдяки цьому, використання декораторів не збільшує час виконання програми і є «безкоштовною» абстракцією (в англомовних джерелах використовується термін «zero-cost abstraction»[2]).

В додатку А наведено приклади використання атрибута *decorate* та порівняння з декораторами з Python.

РОЗДІЛ 3. ВИКОРИСТАНІ ТЕХНОЛОГІЇ ДЛЯ НАПИСАННЯ РОЗШИРЕННЯ КОМПІЛЯТОРА

3.1 Вибір компілятора

Існують три найпопулярніших компілятора для C++: gcc, Clang, MSVC. Було обрано Clang як об'єкт розширення через його архітектуру на основі бібліотек, завдяки якій компілятор може простіше взаємодіяти з іншими інструментами, які взаємодіють з вихідним кодом, такими як інтегровані середовища розробки (IDE). На відміну від Clang, gcc і MSVC працює в більш тісному процесі; інтегрувати їх з іншими інструментами не завжди легко.

Clang зберігає більше інформації під час процесу компіляції, ніж GCC, і зберігає загальну форму оригінального коду, що полегшує відображення помилок назад у вихідне джерело. Звіти про помилки Clang є більш детальними, конкретними та читабельнішими, тому IDE можуть індексувати вихідні дані компілятора. Дерево розбору також більше підходить для підтримки автоматизованого рефакторингу коду, оскільки воно безпосередньо представляє вихідний код.

3.2 Методи додання нової функціональності в Clang

Існують 2 методи доповнення Clang[4]:

- змінення вихідного програмного коду компілятора. Мінус цього підходу в тому, що для внесення навіть невеликої зміни необхідно компілювати весь компілятор;

- написання плагіну для Clang. Плагіни Clang дозволяють виконувати додаткові дії з AST під час компіляції. Плагіни завантажуються компілятором під час виконання і їх легко інтегрувати в середовище збірки. Цей підхід не має мінусів першого способу.

Тому було обрано другий спосіб додання нового функціоналу в Clang

3.3 Мова програмування та середовище розробки

Clang написаний на мові C++ і надає програмний інтерфейс для написання плагінів на мові на мові C++, тому ця мова була обрана для написання плагіну. При розробці була використана бібліотека *fmt* для зручного та швидкого форматування тексту, адже швидкодія це важлива риса для компілятора. Також було використано систему збірки *CMake* та редактор коду *VS Code*.

РОЗДІЛ 4. ВНУТРІШНЯ БУДОВА ТА ПРОЦЕС НАПИСАННЯ РОЗШИРЕННЯ

4.1 Реєстрація плагіну в регістрі плагінів Clang

Для того, що Clang розпізнав розроблюваний плагін, потрібно пронаслідуватись від класу *clang::PluginASTAction*, визначити спеціальні аргументи, які передаються компілятору для налаштування плагіну та зареєструвати його в реєстрі плагінів компілятора[6].

4.2 Додання нового атрибуту

Для того, щоб Clang розпізнав атрибут *decorate* при побудові абстрактного синтаксичного дерева, потрібно додати цей атрибут. Для цього необхідно пронаслідуватись від класу *ParsedAttrInfo* та зробити декілька дій:

– визначити так звані «Spellings» для цього атрибуту — назви цього атрибуту. В розроблюваного атрибуту є тільки один варіант найменування: *decorate*;

– визначити метод *handleDeclAttribute*, який повертає чи може атрибут застосовуватись до певних мовних визначень. Саме тут знаходиться логіка, яка перевіряє чи атрибут застосований саме до визначення функції і чи функція відповідає обмеженням з розділу **Синтаксис та обмеження атрибуту decorate**;

– визначити кількість обов'язкових аргументів;

– визначити чи атрибут може застосовуватись, враховуючи налаштування мови, які були передані компілятору.

4.3 Пошук функцій з атрибутом в синтаксичному дереві

Clang представляє програмний код у вигляді абстрактного синтаксичного дерева. Якщо скомпілювати наведений нижче код:

```
[[decorate(decorator)]]
int foo()
{
    return 5;
}
```

Компілятор побудує приблизно таке AST (спрощено для наглядності):

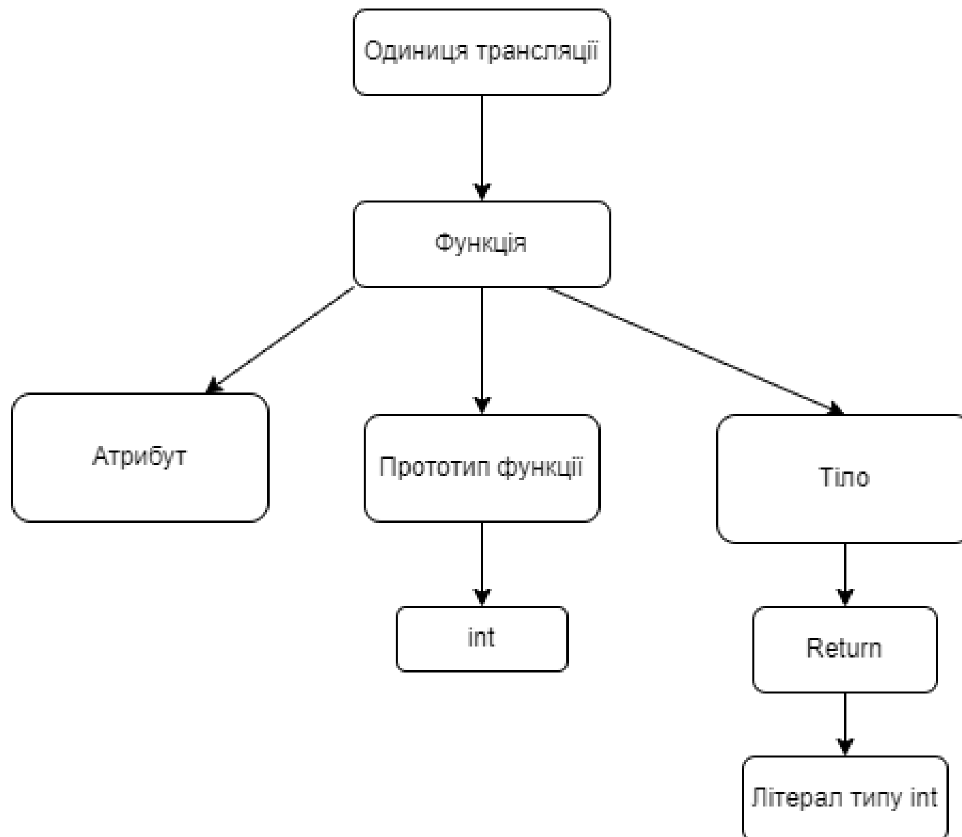

Рис. 1: AST простої програми з використанням атрибуту *decorate*

Для того, щоб зробити декорування функцій, спочатку потрібно знайти всі функції, для яких було застосовано атрибут *decorate*. Для цього потрібно додати «відвідувачів» абстрактного синтаксичного дерева, які будуть обходити всі оголошення функцій. Для кожного знайденого оголошення функції потрібно перевірити чи в них присутній потрібний атрибут та деякі інші умови. Коли підходящий вузол в дереві знайдений плагін переходить до наступного кроку — генерації коду.

4.4 Генерація коду

Для генерації коду плагін використовує клас *clang::Rewriter*. Цей клас є високорівневим програмним інтерфейсом для зміни існуючих файлів з кодом[7]. Для того щоб змінити код якого наявного в коді об'єкту, потрібно дістати його місцезнахоження, яке представляється класом *clang::SourceLocation*, та скористатись методами *Rewriter* для вставки/видалення/зміни коду в якомусь місці в коді. Важливо підкреслити, що *Rewriter* не модифікує файли з кодом, а записує модифіковані частини в тимчасовий буфер.
За допомогою *Rewriter* здійснюється перетворення коду функцій, для яких застосовано атрибут згідно прикладу з розділу. Це перетворення міняє код такого вигляду:

```
[[decorate(decorator1, decorator2, decorator3)]]
void foo()
{
    std::cout << "Hello, world!\n";
}
```

в такий код:

```
void __undecorated_foo()
{
    std::cout << "Hello, world!\n";
}
constinit auto foo =
 ↪  decorator3(decorator2(decorator1(__undecorated_foo)));
```

Плагін міняє оригінальне ім'я функції, яку декорують та зберігає результат виклику декоратора в константу *foo*. Як наслідок, всі звернення до *foo* в програмному коді будуть відсилатись не до оригінальної функції *foo*, а до вже декорованої декоратором *decorator*. Специфікатор «constinit» забезпечує те, що підстановку (або декорування) буде виконано під час компіляції, на відміну від Python, де підстановка робиться на початку роботу програми. Також важливо

звернути увагу, що функцію __*undecorated_foo* не можна викликати з коду.

4.5 Перекомпіляція

Після модифікації коду всіх функцій, для яких було застосовано атрибут плагін ініціює перебудову абстрактного синтаксичного дерева на основі коду з модифікованого буфера для кожної одиниці трансляції, в якій були використані декоратори. Після закінчення перебудови AST плагін закінчує свою роботу і компілятор продовжує процес компіляції.

ВИСНОВКИ

В роботі було розглянуто синтаксис та приклади використання декораторів в мові Python та розроблено синтаксис та очікувану поведінку декораторів для мови C++. Декоратори були реалізовані шляхом додання нового атрибуту. Новий атрибут було додано шляхом написання плагіна для компілятора Clang. Розроблений плагін дозволяє писати код з використанням декораторів на мові C++ схоже до коду з використанням декораторів на Python.

Цей плагін може скомпільовати будь-який бажаючий і використовувати разом з компілятором Clang. Сирцевий код знаходиться в відкритому доступі на репозиторії в Github[8]. Розроблене розширення охоплює всі сфери програмування на C++, адже декоратори використовуються в абсолютно різноманітних програмах на Python, як і їх аналоги в інших мовах програмування.

Шляхів подальшого розвитку роботи є багато: доцільно збільшувати кількості елементів мови, для яких можна застовувати декоратори (класи, змінні, константи тощо) або покращити інформативність помилок при неправильному використанні декораторів. Також можна покращити алгоритм застосування декоратора, а саме: знайти спосіб модифікувати AST напряму без модифікації коду під час компіляції.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

[1] PEP 318 – Decorators for Functions and Methods [Електронний ресурс]. – Режим доступу: URL: - https://peps.python.org/pep-0318/ (переглянуто 20.05.2022) - Назва з екрана.

[2] What does 'Zero Cost Abstraction' mean? [Електронний ресурс]. – Режим доступу: URL: - https://stackoverflow.com/a/69178445 (переглянуто 25.05.2022) - Назва з екрана.

[3] Attribute specifier sequence [Електронний ресурс]. – Режим доступу: URL: - https://en.cppreference.com/w/cpp/language/attributes (переглянуто 21.05.2022) - Назва з екрана.

[4] Choosing the Right Interface for Your Application [Електронний ресурс]. – Режим доступу: URL: - https://clang.llvm.org/docs/Tooling.html (переглянуто 13.05.2022) - Назва з екрана.

[5] Attributes Reference [Електронний ресурс]. – Режим доступу: URL: - https://clang.llvm.org/docs/AttributeReference.html (переглянуто 13.05.2022) - Назва з екрана.

[6] Min-Yih Hsu. LLVM Techniques, Tips, and Best Practices Clang and Middle-End Libraries. Min-Yih Hsu - Велика Британія: «Packt Publishing Ltd.», 2021. - 132 с.

[7] clang::Rewriter Class Reference [Електронний ресурс]. – Режим доступу: URL: - https://clang.llvm.org/doxygen/classclang_1_1Rewriter.html (переглянуто 22.05.2022) - Назва з екрана.

[8] Репозиторій з кодом плагіну [Електронний ресурс]. – Режим доступу: URL: - https://github.com/starquell/decorators-clang-plugin

ДОДАТОК А

Реалізація та використання декоратора який виводить кількість викликів функції, яку декорує на мові Python:

```
def count_calls(func):
    func.calls = 0
    def inner(*args, **kwargs):
        func.calls += 1
        print("this function called {} times".format(func.calls))
        func(*args, **kwargs)
    return inner

@count_calls
def print_something(text):
    print(text)

print_something("Hello!")
print_something("Hello!")
print_something("Hello!")
```

Вивід:

```
this function called 1 times
Hello!
this function called 2 times
Hello!
this function called 3 times
Hello!
```

Аналогічний код на C++ з використанням розробленого атрибуту *decorate*:

```
template <typename Func>
constexpr auto count_calls(Func&& func)
{
    return [func = std::forward<Func>(func), calls = 0] (auto&&... args)
     ↪  mutable {
```

```
            std::cout << "this function called " << ++calls << " times\n";
            func(std::forward<decltype(args)>(args)...);
        };
    }

    [[decorate(count_calls)]]
    void print_something(std::string_view text)
    {
        std::cout << text << '\n';
    }

    int main()
    {
        print_something("Hello!");
        print_something("Hello!");
        print_something("Hello!");
    }
```

Вивід:

```
this function called 1 times
Hello!
this function called 2 times
Hello!
this function called 3 times
Hello!
```

Реалізація та використання декоратора *repeat* на Python:

```python
def repeat(n):
    def deco(func):
        def inner(*args, **kwargs):
            for i in range(n):
                func(*args, **kwargs)
        return inner
    return deco
```

```
@repeat(3)
def print_something(text):
    print(text)

print_something("Hello!")
```

Вивід програми:

Hello!

Hello!

Hello!

Аналогічний код на C++ з використанням розробленого атрибуту *decorate*:

```
consteval auto repeat(int n)
{
    return [n] (auto&& func) constexpr
    {
        return [n, func = std::forward<decltype(func)>(func)] (auto&&...
         ↪  args) mutable
        {
            for (int i = 0; i < n; ++i) {
                func(std::forward<decltype(args)>(args)...);
            }
        };
    };
}

[[decorate(repeat(3))]]
void print_something(std::string_view text)
{
    std::cout << text << '\n';
}

int main()
{
    print_something("Hello!");
}
```

Вивід програми:

Hello!

Hello!

Hello!

Використання декораторів *repeat* і *count_calls* одночасно:

```
@count_calls
@repeat(3)
```

```
def print_something(text):
    print(text)


print_something("Hello!")
```

Вивід програми:

```
this function called 1 times
Hello!
this function called 2 times
Hello!
this function called 3 times
Hello!
```

Аналогічний код на C++ з використанням розробленого атрибуту *decorate*:

```
[[decorate(count_calls, repeat(3))]]
void print_something(std::string_view text)
{
    std::cout << text << '\n';
}


int main()
{
    print_something("Hello!");
    /// Output:
    //   Hello!
    //   Hello!
    //   Hello!
    //   Hello!
    //   Hello!
}
```