Programowanie II R

Zadania – seria 10. rozszerzona

Szablony funkcji i klas - koncepty

Zadanie 1: Koncepty

Wprowadzenie

Na poprzednich zajęciach tworzyliśmy tradycyjne szablony (`template <class T>`), które przyjmują dowolny typ jako argument. Jeśli do solwera RK4 przekazałeś typ, który nie obsługiwał dodawania (np. `std::string`), kompilator orientował się o tym dopiero głęboko w kodzie solwera, wyrzucając setki linii całkowicie nieczytelnych błędów.

Standard **C++20** wprowadza **Koncepty (Concepts)**. Pozwalają one na nałożenie jawnych wymagań na typy przekazywane do szablonów. Zamiast mówić "przyjmij dowolny typ T", możemy powiedzieć kompilatorowi: "przyjmij typ T, ale tylko taki, który da się do siebie dodawać i mnożyć przez liczbę zmiennoprzecinkową". Użycie konceptów daje bardziej czytelny kod, zrozumiałe błędy kompilacji, oraz pozwala na precyzyjną specjalizację szablonów.

Koncept należy najpierw zdefiniować (określić wymagania względem typu), a potem wykorzystać przy pisaniu szablonów. Składnia definicji konceptu bazuje na słowach kluczowych `concept` oraz `requires`. Przykład definicji konceptu, który sprawdza, czy dwa obiekty danego typu można dodawać do siebie i czy wynik dodawania będzie tego samego typu:

```
#include <concepts>

template <typename T>
concept Addable = requires(T a, T b) {
    { a + b } -> std::same_as<T>;
};
```

Przykład wykorzystania konceptu: szablon funkcji, która wymaga, żeby jej argument spełniał koncept "Addable":

```
template <Addable T>
void foo(T arg)
{...}
```

Koncept może zawierać więcej niż jeden argument szablonu:

```
template <typename T, typename real_t>
concept MyConcept = ...
```

Przykład jego wykorzystania:

```
template <MyConcept<double> Type>
void foo(Type arg)
{...}
```

Kompilator uznaje `Type` za pierwszy argument szablonu konceptu (`T`) a `double` za drugi argument szablonu konceptu (`real_t`).

Treść zadania

Zadaniem jest refaktoryzacja kodu z 10. serii zadań. Stwórz i zastosuj dwa koncepty:

1. `VectorSpace`: Typ reprezentujący stan układu, który musi wspierać operację dodawania dwóch stanów oraz mnożenia stanu przez skalar typu `double`, obie zwracające typ "stan układu". Ponadto typ ten powinien obsługiwać operator `[i]`, gdzie `i` to zmienna typu `std::size_t`. Ten operator powinien zwracać zmienną, którą można zamienić na typ `double` (użyj `std::convertible_to`). Przykładem typu spełniającego ten koncept jest klasa reprezentująca wektor napisana w zadaniu 10.

2. `ODESystem`: Obiekt reprezentujący prawe strony równań, który obsługuje operator `()` z jednym argumentem reprezentującym stan (zgodnym z konceptem `VectorSpace`), a zwracającym wartość, która jest tego samego typu co stan. Przykładowa składnia:

```
1       template <typename F, typename V>
2       concept ODESystem = VectorSpace<V> && requires(F f, V u) {...}
3
```

Przepisz szablon funkcji `rk4_step` używając tych konceptów.

Opracowanie: Piotr Dziekan