Programowanie II R

Zadania – seria 11.

Nowoczesny C++

Cel zajęć

Zapoznanie się z najnowszymi mechanizmami języka C++ (standardy C++20 oraz C++23). Poniżej przedstawiono fragmenty kodu źródłowego napisane w starszych standardach (C++11/14/17). Waszym zadaniem jest ich refaktoryzacja, czyli przepisanie w taki sposób, aby uzyskać identyczny wynik działania programu, wykorzystując nowoczesne, zwięzłe i bezpieczniejsze konstrukcje.

UWAGA: zadania wymagają użycia nowych wersji kompilatora (gcc16, clang17). Jeśli nie są one dostępne lokalnie, można użyć kompilatora online: `https://wandbox.org`.

Zadanie 1: Moduły

Tradycyjnie, zawartość plików nagłówkowych (`.h`, `.hpp`) była bezpośrednio wklejana do kodu programu za pomocą `#include ...`. Powoduje to zwiększenie czasu kompilacji i rozmiaru skompilowanych programów oraz może prowadzić do konfliktów nazw. Standard C++20 zastępuje ten mechanizm za pomocą niezależnie kompilowanych *modułów*.

Stary kod

"Biblioteka" `math_utils` służąca do dodawania dwóch zmiennych typu `int`:

Plik `math_utils.hpp`:

```hpp
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int add(int a, int b);

#endif
```

lub

```hpp
#pragma once

int add(int a, int b);
```

Plik `math_utils.cpp`:

```cpp
#include "math_utils.h"

int add(int a, int b) {
    return a + b;
}
```

Plik `main.cpp`:

```cpp
#include <iostream>
#include "math_utils.h"

int main() {
    std::cout << "Wynik: " << add(5, 7) << std::endl;
    return 0;
}
```

Instrukcja refaktoryzacji

Przepisz powyższy kod definiując moduł `MathUtils`, pozbywając się pliku `.hpp` oraz dyrektywy `#include "math_utils.hpp"`.

Wskazówki

1. Utwórz jeden plik dla modułu (np. `math_utils.cppm`). Na samej górze zadeklaruj nazwę modułu poprzez: `export module MathUtils;`.
2. Słowo kluczowe `export` postaw bezpośrednio przed definicją funkcji, którą chcesz udostępnić na zewnątrz: `export int add(int a, int b) { ... }`.
3. W pliku `main.cpp` zamiast `#include` zaimportuj swój moduł: `import MathUtils;`.
4. Standard definiuje, że cała biblioteka standardowa powinna być dostępna jako jeden moduł: `import std;`, jednak wiele kompilatorów jeszcze tego nie obsługuje.
5. Kompilacja (na przykładzie `gcc`):
   (a) Kompilacja modułu: `g++ -std=c++23 -c math_utils.cppm -fmodules`
   (b) Kompilacja programu: `g++ -std=c++23 main.cpp math_utils.o -fmodules`

Zadanie 2: Formatowanie tekstu (std::print i std::println)

Stary kod

Wypisywanie sformatowanego tekstu (np. wyrównanie pól tekstowych, określenie stałej liczby miejsc po przecinku) przy użyciu `std::cout` zmusza do stosowania mało czytelnych i skomplikowanych manipulatorów z biblioteki `<iomanip>`.

```cpp
#include <iostream>
#include <iomanip>
#include <string>

struct Student {
    std::string name;
    double grade_average;
};

int main() {
    Student s{"Jan Kowalski", 4.7562};

    // Wypisanie imienia na 15 znakach (wyrownane do lewej)
    // oraz oceny z dokladnoscia do 2 miejsc po przecinku
    std::cout << "Student: " << std::left << std::setw(15) << s.name
              << " | Srednia: " << std::fixed << std::setprecision(2) << s.grade_average << std::endl;

    return 0;
}
```

Instrukcja refaktoryzacji

Zastąp manipulatory strumieniowe nową funkcją `std::println`, wprowadzającą intuicyjną składnię formatowania wzorowaną na językach takich jak Python czy Rust.

Wskazówki

1. Zamiast nagłówków `<iostream>` i `<iomanip>` dołącz dedykowany dla nowej funkcjonalności nagłówek `<print>` (lub, jeśli to możliwe, skorzystaj z `import std;`).
2. Wywołaj funkcję w formacie: `std::println("szablon_tekstu", argument1, argument2);`. Funkcja ta automatycznie dodaje znak nowej linii na końcu.
3. Miejsca na zmienne oznaczasz klamrami {}. Formatowanie zaawansowane podajesz po dwukropku wewnątrz klamer: np. {`:<15`} wyrównuje tekst do lewej strony na polu o szerokości 15 znaków, a {`:.2f`} ogranicza wyświetlanie liczby zmiennoprzecinkowej do dwóch miejsc po przecinku.

Zadanie 3: Operacje na zbiorach danych (std::ranges i std::views)

Stary kod

Przetwarzanie zbiorów danych (np. filtrowanie elementów, przekształcanie ich wartości) wymagało dotychczas pisania zagnieżdżonych pętli lub użycia skomplikowanej składni `STL`. Często wymagało też tworzenia zbędnych zmiennych tymczasowych i kontenerów pomocniczych. Poniższy program wypisuje kwadraty liczb parzystych zawartych w wektorze.

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    for (int n : numbers) {
        if (n % 2 == 0) {
            std::cout << n*n << " ";
        }
    }

    std::cout << std::endl;
}
```

Instrukcja refaktoryzacji

Wykorzystaj bibliotekę `<ranges>` (C++20) do wykonania operacji filtrowania i transformacji w sposób czysto deklaratywny. Zastosuj mechanizm widoków (*views*) łączonych operatorem potoku `|`.

Wskazówki

1. Wyrażenie przetwarzające wektor możesz zapisać jako jedną instrukcję: `auto result = numbers | std::views::filter(...) | std::views::transform(...);`. W ten sposób definiujemy jedynie sposób przetwarzania danych, ale przetwarzanie to jeszcze nie jest wykonywane.
2. Jako argumenty widoków przekaż odpowiednie wyrażenia lambda, np. dla filtrowania `[](int n) { return n % 2 == 0; }`.

3. Zawartość widoku `results` można wypisać w oddzielnym kroku używając `println`. Widoki są przetwarzane w sposób leniwy (*lazy evaluation*) – obliczenia zostaną wykonane dopiero, gdy wynik faktycznie będzie potrzebny (przy wypisywaniu).

Zadanie 4: Tablice wielowymiarowe (std::mdspan z C++23)

Stary kod

Standardowe kontenery w C++ (takie jak `std::vector`) oferują wyłącznie strukturę jednowymiarową. Tworzenie macierzy poprzez zagnieżdżenie `std::vector<std::vector<int>>` jest wysoce nieefektywne. Najbardziej wydajnym podejściem jest użycie płaskiego wektora i ręczne przeliczanie indeksów. Poniższy program używa wektora do reprezentacji tablice 2D.

```cpp
#include <iostream>
#include <vector>

int main() {
    int width = 3;
    int height = 3;
    // Plaska tablica reprezentujaca obraz/macierz 3x3
    std::vector<int> data = {1, 2, 3,
                             4, 5, 6,
                             7, 8, 9};

    // Chcemy pobrac element z wiersza y=1, kolumny x=2 (oczekiwany wynik: 6)
    int y = 1;
    int x = 2;
    int value = data[y * width + x];

    std::cout << "Element: " << value << std::endl;
}
```

Instrukcja refaktoryzacji

Nałóż na płaską strukturę jednowymiarową widok wielowymiarowy przy użyciu `std::mdspan` z C++23, usuwając konieczność ręcznego wyliczania przesunięcia pamięci.

Wskazówki

1. Dołącz nagłówek `<mdspan>` (lub użyj `import std;`).
2. Zainicjalizuj dwuwymiarowy widok, przekazując wskaźnik do początku danych oraz rozmiary poszczególnych wymiarów (wiersze, kolumny):
   `std::mdspan matrix{data.data(), height, width};`.
3. Wykorzystaj nową składnię standardu C++23, w której operator indeksowania `[]` zyskał możliwość przyjmowania wielu argumentów rozdzielonych przecinkami. Odczytaj wartość za pomocą czytelnego zapisu: `matrix[y, x]`.

Zadanie 5: Statek kosmiczny (Operator `<=>` z C++20)

Stary kod

Aby obiekt własnej klasy użytkownika mógł być w pełni sortowalny lub poprawnie przechowywany w kontenerach asocjacyjnych (np. `std::set`), programista zmuszony był do ręcznego przeciążania aż sześciu różnych operatorów relacyjnych (`==`, `!=`, `<`, `<=`, `>`, `>=`).

```cpp
#include <iostream>

struct Wzrost {
    int m, cm; // metry i centymetry

    bool operator==(const Wzrost& o) const { return m == o.m && cm == o.cm; }
    bool operator!=(const Wzrost& o) const { return !(*this == o); }
    bool operator<(const Wzrost& o) const {
        if (m != o.m) return m < o.m;
        return cm < o.cm;
    }
    bool operator<=(const Wzrost& o) const { return !(o < *this); }
    bool operator>(const Wzrost& o) const { return o < *this; }
    bool operator>=(const Wzrost& o) const { return !(*this < o); }
};

int main() {
    Wzrost Jan{1, 80}, Marian{0, 90};
    std::cout << (Jan < Marian ? "Jan jest nizszy" : "Marian jest nizszy") << std::endl;
}
```

Instrukcja refaktoryzacji

Zastąp całą sekcję operatorów porównania w strukturze `Wzrost` jedną, zwięzłą linijką kodu, wykorzystującą operator porównania trójdrożnego `<=>`.

Wskazówki

1. Zadeklaruj domyślny operator trójdrożny (tzw. *spaceship operator*): `auto operator<=>(const Wzrost&) const = default;`.
2. Kompilator automatycznie wygeneruje wszystkie sześć operatorów relacyjnych, porównując składowe struktury w kolejności ich definicji (najpierw `m`, potem `cm`).

Opracowanie: Piotr Dziekan