Sprawozdanie z projektu z przedmiotu "Programowanie i projektowanie aplikacji obiektowych"

Artur Kuś
Andrzej Zagórski
Michał Lubczyński

29.01.2023

1. Wstęp

Temat projektu dotyczy zaprojektowania oraz zaimplementowania biblioteki narzędzi statystycznych w języku C++ o nazwie **OOStatLib**.

1.1 Założenia projektu

Początkowo biblioteka miała spełniać następujące, ogólne założenia:

1. Być jak najbardziej generyczna, jeżeli chodzi o używane typy danych.
2. Zaimplementować większość prostych algorytmów statystycznych.
3. Być *"szybka"*.
4. Być *"prosta w obsłudze"*.

1.2 Podział obowiązków

- Andrzej Zagórski - struktura programu, diagram UML.
- Michał Lubczyński - dokumentacja kodu, raport, sprawozdanie.
- Artur Kuś - implementacja w języku C++.

2. Struktura OOStatLib

2.2 Generyczność i typy pomocnicze

Biblioteka, aby była użyteczna, potrzebuje być "niezależna" od tego, jakich typów liczbowych, znakowych i enumerycznych używa programista korzystający z niej. Różne architektury procesorów mogą być przystosowane szybszej pracy na różnych typach (np. 64 i 32 bitowe) oraz każdy użytkownik może chcieć posługiwać się z różną dokładnością liczbową. Dlatego postanowiono, aby klasy działały na generycznych typach danych, oraz zwracały wartość zmiennoprzecinkową (float lub double) zależnie od tego, jakiego typu wynik życzy sobie programista.

Wymogiem stało się rozwiązanie zapewniające jak największą generyczność, bez konieczności przepisywania funkcji na nowo z innymi typami argumentów i zwracania. Trzeba było również zapewnić bezpieczeństwo wprowadzanych przez użytkownika typów.

2.1 Diagram UML OOStatLib

3. Rozwiązania implementacyjne OOStatLib

Pełna dokumentacja jak i cała biblioteka została zahostowana na githubie i jest dostępna pod poniższymi linkami:

- link do dokumentacji - https://artqs01.github.io/OOStatLib/,
- link do biblioteki - https://github.com/artqs01/OOStatLib.

W wersji 0.1 biblioteki zostały zaimplementowane proste testy statystyczne:

- test Shapiro-Wilka,
- test dla par obserwacji,
- test istotności dla średniej,
- test istotności dla wariancji.

Zostały również zaimplementowane proste algorytmy pomocnicze:

- średnia,
- wariancja,
- odchylenie standardowe.

Wszystkie testy ustalają poziom istotności za pomocą typu sl::significance - typu wyliczeniowego, który zawiera w sobie wszystkie popularne wartości poziomu istotności:

- 0,01;
- 0,02;

- 0,05;
- 0,1;

Typy zmiennych są zaimplementowane jako koncepty z C++20. Narzędzia z biblioteki standardowej używane jako zależności w bibliotece to:

- type_traits,
- concepts,
- vector,
- concepts,
- cmath,
- cassert,
- algorithm,
- cstddef,
- memory.

Zatem jedynym narzędziem potrzebnym do uruchomienia biblioteki jest kompilator wspierający C++20 (niemalże wszystkie).

3.1 Rozważanie na temat bezpiecznej i wygodnej implementacji biblioteki

Sprawę bezpieczeństwa i niezdefiniowane zachowanie rozwiązuje standard języka C++20, a konkretnie koncepty (*concepts*), które można nakładać na typy generycznie i przez to ograniczanie *at compile time* typów, które podaje się jako argumenty do generycznych klas, bez konieczności wprowadzania własnych typów/klas liczbowych.

W ten sposób też zaimplementowano skale pomiarowe:

- nominalną,
- porządkową,
- różnicową,
- ilorazową.

Podczas rozeznania się z nowym standardem języka zdecydowano, że można zastąpić konceptami funkcje wirtualne, których wywoływanie jest wolniejsze, niż typowe metody składowe klasy. W przypadku, gdy funkcja jest niezdefiniowana, także pojawia się błąd kompilacji z tą różnicą, że błąd kompilatora jest mniej czytelny dla programisty. Wada jest taka, że kod jest trochę bardziej rozbudowany, ale nieznacznie.

4. Analiza pod kątem wcześniejszych założeń.

4.1 Generyczność

Każdy test i algorytm został zrobiony w taki sposób, że podaje się do niego osobno typ danych wejściowych, oraz typ zmiennoprzecinkowy, na którym wykonywane są obliczenia. Pod tym względem bibliotece nic nie brakuje i jest ona złożona w sposób wystarczający oraz sposób, w jaki domyślane są typy są wykonywane na etapie kompilacji (szybko). Na przykładzie:

```
bool pair_observations_test<DataType1, DataType2, CalcType>::evaluate(paired_container<DataType1, DataType
{
    using DifferenceType = std::conditional_t<std::is_floating_point_v<DataType1> && std::is_floating_poin
        std::conditional_t<sizeof(DataType1) >= sizeof(DataType2),
            DataType1,
            DataType2
        >,
        std::conditional_t<std::is_floating_point_v<DataType1>,
            DataType1,
            DataType2
        >
    >;
    std::vector<DifferenceType> difference(data.size());
```

```
    ...
}
```

Tutaj dedukowany jest typ dynamicznej tablicy (`std::vector`), w której będą przechowywane różnice danych w w kontenerze `data`.

4.2 Funkcjonalność

4.2.1 Przykład użycia

```
#include "data.hpp"
#include "single_data_algorhithms.hpp"
#include <iostream>
#include <memory>

int main()
{
    sl::single_container<double> data1 = {
        26.5,
        27.5,
        27.5,
        26.9,
        27.9,
        27.0,
        27.7,
        27.6,
        27.9,
        27.4,
        28.3,
        27.6,
        26.3,
        25.9,
        27.9
    };
    auto data1_ptr = std::make_shared<sl::single_container<double>>(data1);
    sl::shapiro_wilk<double> sw1(data1_ptr, sl::significance::five_hundredths);
    std::cout <<
        "funkcja statyczna: " <<
        sl::shapiro_wilk<double>::evaluate(*data1_ptr, sl::significance::five_hundredths) << "\n" <<
        "metoda obiektu: " << sw1();
}
```

Wyjście programu:

```
[heniek@komp build]$ ./statlib
funkcja statyczna: 1
metoda obiektu: 1
```

4.2.2 Niezaimplementowane (jeszcze) funkcjonalności

Funkcjonalności potrzebne do zaimplementowania, aby biblioteką dało się używać w sensowny sposób to:

- czytanie i zapisywanie do pliku,
- tryb, w którym test wypisuje dodatkowe dane, jednocześnie licząc,
- zaimplementowanie większej liczby testów.

4.3 Wydajność

Jeżeli chodzi o optymalizację to nie byliśmy na ten moment w stanie znaleźć szczególnie istotne optymalizacje wykonywanych obliczeń, lecz przeanalizowaliśmy program linuxowym narzędziem perf, który analizuje program pod

kątem nieprzewidzianych dla procesora gałęzi (*cache misses*):

```
 Performance counter stats for './statlib':

              1.30 msec task-clock:u                     #    0.745 CPUs utilized
                 0      context-switches:u               #    0.000 /sec
                 0      cpu-migrations:u                 #    0.000 /sec
               130      page-faults:u                    #  100.271 K/sec
         2,205,975      cycles:u                         #    1.702 GHz
         2,669,142      instructions:u                   #    1.21  insn per cycle
           471,494      branches:u                       #  363.670 M/sec
            15,054      branch-misses:u                  #    3.19% of all branches

       0.001740116 seconds time elapsed

       0.001790000 seconds user
       0.000000000 seconds sys
```

Poziom *cache misses* w tym programie wynosił około 3%, a w trybie debug mniej, niż 1%, co jest całkiem dobrym wynikiem, aczkolwiek mogło być lepiej.

5. Podsumowanie

5.1 Bibliografia

- https://en.cppreference.com/w/,
- https://www.youtube.com/watch?v=g-WPhYREFjk,
- https://www.youtube.com/watch?v=gTNJXVmuRRA,
- https://www.youtube.com/watch?v=HqsEHG0QJXU.

5.2 Wnioski

Nie udało nam się zaimplementować wszystkiego, co sobie założyliśmy, lecz po drodze wiele się nauczyliśmy z dziedziny programowania generycznego, które jest trudne i czasem kontrintuicyjne, ale po spędzonym czasie z wszystkimi źródłami, udało nam się napisać solidną bazę, którą teraz można już tylko rozbudowywać.

Aktualny trend w C++, można powiedzieć, jest w kontrapunkcie do innych współczesnych języków. Gdzie np. w Języku Java można znaleźć takie "perełki", jak wyjątek `java.lang.RuntimeException: Uncompilable source code`, to w języku C++ trend polega na pisaniu programów, których jak największa część instrukcji jest realizowana zanim program się wykona, na etapie kompilacji, co jest, naszym zdaniem, warte uwagi i godne pochwały.