Matlab-2

Wersja: 1.0

W tym zadaniu dokonasz refaktoryzacji kodu programu *Matlab-1* wprowadzając do niego elementy obiektowości – m.in. zdefiniujesz strukturę danych reprezentującą matematyczny wektor.

Uwagi wstępne

Ważne

Ten kurs ma Cię nauczyć pewnych dobrych nawyków programistycznych oraz stosowania pewnych dobrych technik programistycznych.

W związku z tym w rozwiązaniach *nie korzystaj* z instrukcji „`using namespace std;`" – odwołuj się do wszystkich nazw funkcjonalności z biblioteki standardowej za pomocą nazwy kwalifikowanej, czyli nazwy odpowiedniej przestrzeni nazw (`std`) oraz operatora zasięgu `::`, np.:

```
#include <vector>

// BARDZO ŹLE -- zły nawyk, który zaciemnia kod
//  (która funkcjonalność z biblioteki standardowej, a która własna?)
using namespace std;
vector<int> vi_nostd;

// POPRAWNIE -- zwróć uwagę na `std::`
std::vector<int> vi_std;
```

Informacja

Ten kurs ma Cię nauczyć pewnych dobrych nawyków programistycznych oraz stosowania pewnych dobrych technik programistycznych – w szczególności, że:
W programie nie powinny występować nieużywane nigdzie (efektywnie) zmienne.

Jednak na początkowym etapie rozwiązywania zadania możesz chcieć dopuścić sytuację, gdy niektóre zmienne (np. parametry jeszcze niezaimplementowanych funkcji) nie są w praktyce wykorzystywane.
W tym celu w projekcie danego zadania dodaj w pliku `CMakeLists.txt` po poleceniu

```
set(PROJECT_ID ...)
```

polecenie

```
add_compile_options(-Wno-error=unused-variable)
```

(Pamiętaj, aby usunąć/zakomentować powyższe polecenie przed ostatecznym uznaniem zadania za rozwiązane.)

1. Tworzenie projektu CLion na podstawie istniejących źródeł

Za punkt wyjścia do rozwiązania zadania *Matlab-2* należy przyjąć wzorcowe rozwiązanie zadania *Matlab-1*. W tym celu wykonaj poniższe kroki:

1. Skopiuj *zawartość* katalogu `$REPO$/solutions/matlab-1` (a *nie* cały katalog!) do katalogu `$REPO$/skeletons/matlab-2`.
2. Otwórz CLion.
3. Utwórz nowy projekt na podstawie istniejących plików programu:
4. Wybierz z menu *File* → *New Project*.
5. Jako *Location* wybierz katalog `$REPO$/skeletons/matlab-2`.
6. Kliknij przycisk *Create*. Pojawi się okno dialogowe z informacją, że katalog `$REPO$/skeletons/matlab-2` nie jest pusty i z pytaniem, czy chcesz utworzyć projekt na podstawie istniejących źródeł – kliknij przcisk *Yes*.
7. Otwórz plik `CMakeLists.txt` z głównego katalogu projektu i zmień wszystkie wystąpienia `matlab_1` na `matlab_2`.
8. U góry okna edytora pojawi się żółty pasek – kliknij na znajdujący się na nim link „*Load CMake project*" (albo „*Reaload Changes*" – jeśli projekt CMake został już uprzednio załadowany).
9. Poczekaj, aż projekt CMake zostanie załadowany – lista konfiguracji budowania powinna zostać uzupełniona m.in. pozycjami „*matlab_2__debug*" i „*matlab_2__test*".

2. Definiowanie klasy i jej metod

Ćwiczenie na: definiowanie klasy, definiowanie metod

Zdefiniuj w pliku `include/matlab.hpp` klasę `MatVect` (reprezentującą matematyczny wektor).

Po realizacji tego ćwiczenia klasa `MatVect` powinna posiadać następującą minimalną funkcjonalność (szczegóły „techniczne" tych funkcjonalności podano dalej):

- możliwość przechowywania składowych wektora (tj. wartości współrzędnych kartezjańskich opisujących dany wektor)
- możliwość tworzenia wektora dla przestrzeni o zadanej liczbie wymiarów (np. $\mathbb{R}^2$, $\mathbb{R}^3$ itd.)
- dostęp do składowych wektora (zarówno w celu ich odczytu, jak i modyfikacji)
- uzyskiwanie informacji o wymiarowości przestrzeni, do której należy dany wektor
- uzyskiwanie informacji o normie wektora

Jak widzisz, utworzenie specjalnej klasy reprezentującej matematyczny wektor jest wskazane, gdyż:

- „matematyczny wektor" wiąże ze sobą ściśle dane (składowe wektora) oraz operacje na tych danych
- użytkowników „matematycznego wektora" nie powinien interesować sposób przechowywania składowych wektora w pamięci komputera (to tzw. hermetyzacja)

Klasa ta powinna zawierać następujące pola (prywatne):

- `std::vector<int> v_` – przechowuje składowe wektora (zob. `std::vector`)[1]

oraz następujące metody (publiczne):

- konstruktor jednoargumentowy przyjmujący liczbę składowych wektora – konstruktor powinien zainicjalizować pole `v_` odpowiednią liczbą elementów o wartości 0[2]
- `get_elem(pos)` – zwraca składową na pozycji o indeksie `pos`
- `set_elem(pos, val)` – przypisuje składowej na pozycji o indeksie `pos` wartość `val`
- `size()` – zwraca rozmiar wektora (tj. liczbę jego składowych)
- `norm()` – zwraca normę wektora (norma wektora $\mathbf{v}$ w przestrzeni euklidesowej wynosi $\|\mathbf{v}\| = \sqrt{v_1^2 + \cdots + v_n^2}$, gdzie v_i to i-ta składowa wektora v)

Uwagi odnośnie implementacji:

- Co do zasady w definicji klasy, w pliku nagłówkowym, powinny znajdować się jedynie prototypy metod[3].
  Jednak w tym ćwiczeniu metody `get_elem()`, `set_elem()` i `size()` są tak krótkie (jedna instrukcja, która nie wymaga dodatkowych bibliotek), że możesz zdefiniować je od razu w definicji klasy.
- Pamiętaj u umieszczeniu w pliku nagłówkowym tzw. **strażnika** (ang. *header guard*).
- Pamiętaj o stosowaniu typu `std::size_t`, gdy zmienna ma wyrażać indeks elementu lub liczbę elementów (np. tablicy, kontenera `std::vector` itd.).
- Pamiętaj o dołączeniu biblioteki `<vector>` (inaczej np. instrukcja `std::vector<int> vi;` zwróci mało czytelny błąd – *'vector' in namespace 'std' does not name a template type*).
- Konstruktor domyślny kontenera `std::vector` tworzy *pusty* kontener – czyli nie zawiera on żadnego elementu! Poniższy kod z pewnością zakończy się błędem wykonania:

```cpp
#include <cstdlib>
#include <vector>

int main() {
  std::vector<int> v;  // wywołanie konstruktora domyślnego
  v[0] = 1;  // BŁĄD: Kontener jest pusty, nie można odwołać się do 1. elementu!

  return EXIT_SUCCESS;
}
```

- Zwróć uwagę, że metoda `size()` z klasy `MatVect` oraz metoda `size()` dla typu `std::vector<int>` to dwie różne metody!
- W implementacji metody `norm()` skorzystaj z funkcji `sqrt()` z biblioteki `<cmath>` – funkcja ta zwraca pierwiastek kwadratowy z argumentu.

W przypadku definiowania metody poza definicją klasy pamiętaj, że nazwa metody w miejscu jej definicji powinna być **nazwą kwalifikowaną** (tj. musi być poprzedzona nazwą klasy i operatorem zasięgu `::`), np.:

[1] Na dobrą sprawę bardziej wskazane byłoby użycie kontenera `std::array` – obiektowego odpowiednika tablicy statycznej. Kontener `std::vector` został użyty ze względów dydaktycznych – chodzi o to, aby dobrze poznać obsługę jednego typu kontenera (a nie „skakać z kwiatka na kwiatek").

[2] Na kolejnych laboratoriach użyjesz w tym celu odpowiedniego konstruktora szablonu klasy `std::vector` – tzw. **konstruktora wypełniającego** (ang. *fill constructor*).

[3] Jeśli dana metoda ma puste ciało albo ciało długości jednej instrukcji (i jej implementacja nie wymaga korzystania z dodatkowych bibliotek), dopuszczalne jest jej zdefiniowanie (tj. zaimplementowanie) od razu w definicji klasy – w przeciwnym razie jej definicja powinna zostać umieszczona w pliku źródłowym.

```
class Dummy {
public:
  void foo();  // `foo` w definicji klasy -- kwalifikacja nie jest konieczna
};

// `foo` jako nazwa kwalifikowana (`Dummy::foo`)
void Dummy::foo() {
  // ...
}
```

Wzorując się na istniejących testach jednostkowych zdefiniowanych w pliku `test/test_vector.cpp`, utwórz w tym pliku analogiczny scenariusz testowy `MatlabVectorTest.norm` dla metody `norm()` – możesz sprawdzić, czy wywołanie tej metody dla wektora $\mathbf{v} = (3, 4, 0)$ zwróci wartość 5:

```
// Zapis 'MatlabVectorTest.norm' oznacza:
// * MatlabVectorTest -- nazwa zbioru testów
// * norm -- nazwa przypadku testowego
TEST(MatlabVectorTest, norm) {
  // ... logika testu (czy ||[3, 4, 0]|| = 5?) ...
}
```

Pamiętaj, że możesz w wygodny sposób inicjalizować kontenery z użyciem nawiasów klamrowych:

```
  std::vector<int> v{1, 2};  // `v` zawiera dwa elementy: 1 i 2
```

Wykonaj (tj. uruchom) testy jednostkowe aby upewnić się, że metody zostały zaimplementowane zgodnie z oczekiwaniami.

3. Refaktoryzacja funkcji

Ćwiczenie na: testy jednostkowe

Dokonaj refaktoryzacji funkcji `add_vectors()` – w nowej wersji funkcja ta powinna przyjmować jako argumenty dwa obiekty klasy `MatVect` i zwracać *nowy* obiekt klasy `MatVect` (sumowanie można zrealizować np. tworząc najpierw nowy obiekt klasy `MatVect` o odpowiedniej liczbie elementów o wartości `0`, a następnie skorzystać z metod `get_elem()` i `set_elem()`).
Aby dokonać refaktoryzacji, skorzystaj z odpowiedniej funkcjonalności IDE – ze możliwości zmiany sygnatury.

Zmień test `MatlabVectorTest.add` tak, aby oddawał nową rzeczywistość (tj. istnienie klasy `MatVect` oraz zmienione parametry funkcji `add_vectors()`).

Dodaj w pliku `test/test_vector.cpp` poniższy scenariusz testowy:

```
TEST(MatlabVectorTest, createWithSize) {
  MatVect v(2U);

  ASSERT_EQ(v.size(), 2U);  // przyrostek ,,U'' oznacza wartość bez znaku (ang. unsigned)
  EXPECT_EQ(v.get_elem(0), 0);
  EXPECT_EQ(v.get_elem(1), 0);
}
```

Wykonaj testy jednostkowe aby upewnić się, że metody zostały zaimplementowane zgodnie z oczekiwaniami.

Wzorując się na teście `MatlabVectorTest.createWithSize` utwórz w tym pliku analogiczny scenariusz testowy `MatlabVectorTest.createFromVector` dla konstruktora jednoargumentowego przyjmującego obiekt typu `std::vector<int>`.
Wykonaj testy jednostkowe aby upewnić się, że metody zostały zaimplementowane zgodnie z oczekiwaniami.