Testowanie kodu

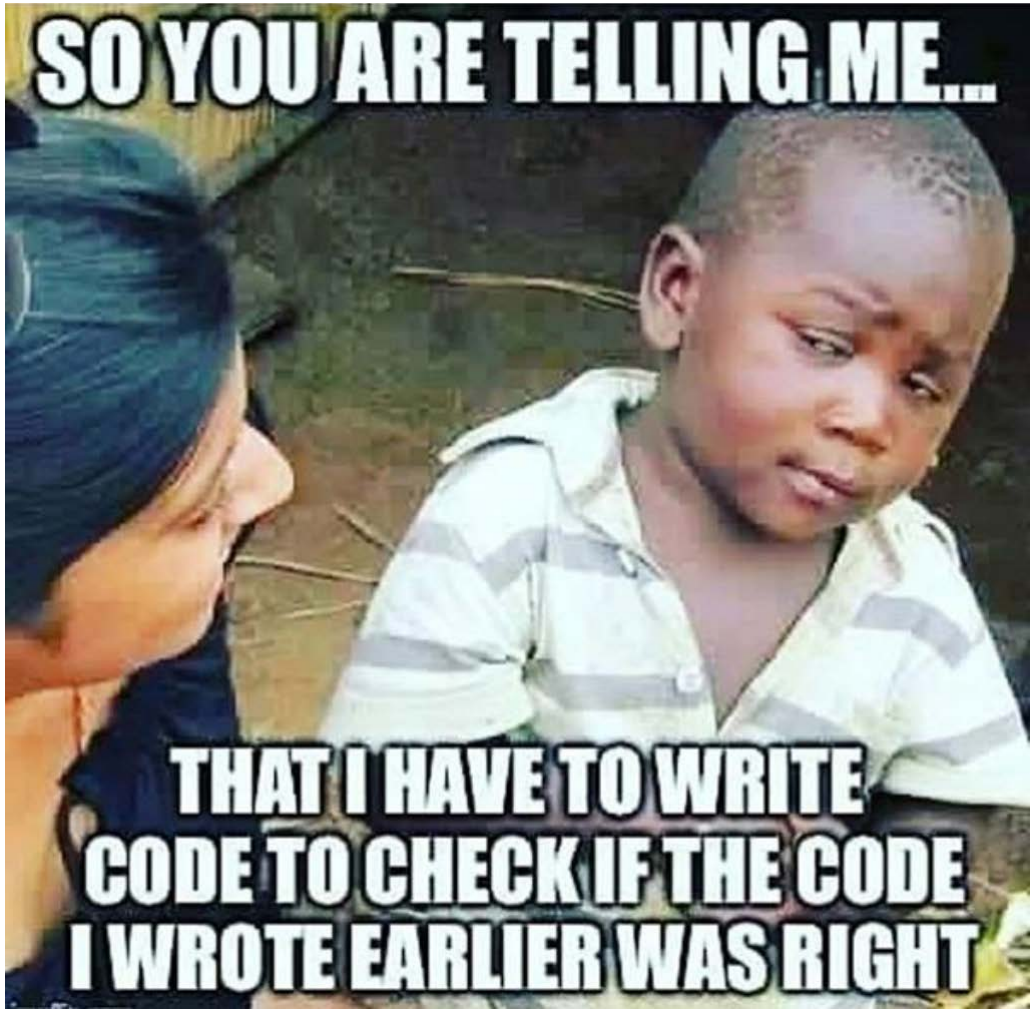
Tak, napisany kod trzeba przetestować i w tym ćwiczeniu podejmiemy kilka kroków, które to zadanie będą miały ułatwić.

Zaczniemy jednak od zmiany ustawień środowiska:

- ustawienie wysokiego poziomu ostrzegania,

- identyfikacji autora kodu w plikach tworzonych w środowisku,

- przyjrzymy się także możliwościom uruchamiania programu.



Modyfikacje środowiska pracy

Jakie konkretnie opcje kompilacji są używane możemy sprawdzić w zakładce `Build log` w dolnej części okna `C::B`. (Jeśli nie widzicie potężnej baterii zakładek na dole, to poleceniem `View > Log` można przełączać ich wyświetlanie).

U mnie wywołanie kompilacji wygląda tak:

```
g++.exe -fexceptions -g  -c  (…)
```

Tymczasem widać tutaj wstawianie dodatkowego kodu obsługi wyjątków (jeszcze ich nie używamy), dokładanie informacji dla debuggera (bardzo dobrze!) i wreszcie ograniczenie się tylko kompilacji kodu (konsolidacja pójdzie później).

Zmiany ustawień możemy wykonać w dwóch miejscach:

- dla projektu `Project > Build options…`
- ogólnie dla `C::B Settings > Compiler…`

Ustawienia kompilacji

Ustawienia kompilacji zmienimy dla całego środowiska, żeby nie trzeba było pamiętać o zmianach przy okazji tworzenia każdego projektu. W oknie Global compiler settings, które otwieramy poleceniem ustawiamy następujące opcje:

`Have g++ follow the C++11 ISO C++ language standard`

`Enable all common compiler options`

`Enable extra compiler warnings`

To powinno zapewnić nam stosunkowo wysoki poziom ostrzegania, wspólny dla wszystkich uczestników. Pociąga to za sobą dość istotne konsekwencje: przy sprawdzaniu rozwiązań będę marudził (co znajdzie wyraz w dodaniu do wyniku -1 punktu).

Zmiany w środowisku (domowym!)

Już w poprzednim zadaniu sugerowałem, żeby w oddawanych plikach umieszczać identyfikację autora: imię, nazwisko i nr albumu. Nie będę tego specjalnie sprawdzał, ale ułatwia mi to pracę, kiedy w jednej serii oceniam kilkanaście rozwiązań.

W oknie `Configure editor`, które otwieramy poleceniem `Settings > Editor`…, szukamy na liście po lewej stronie pozycji `Deafault code`.

W dużym pustym edytorze możemy wpisać dowolny tekst, który będzie wstawiany w każdym nowotworzonym pliku źródłowym i nagłówkowym.

Ja w obu przypadkach ustawiłem ten sam tekst:

```
// Rajmund Kożuszek (121528)
```

Każdy oczywiście wpisuje tutaj swoje dane. Za moment popatrzymy, jak to działa w praktyce.

Obliczenia zmiennopozycyjne

Jesteśmy od lat przyzwyczajeni do pewnych oczywistości matematycznych, np.:

$$\sqrt{2} * \sqrt{2} = 2$$

W momencie, kiedy przyjdzie nam wykonywać obliczenia z ograniczoną precyzją, sprawa przestaje być oczywista. Zaczniemy od napisania prościutkiego programu, który sprawdza, że to zacne równanie nie jest spełnione. Pójdziemy jednak krok dalej i napiszemy także funkcję, która sprawdza, czy dwie wartości zmiennoprzecinkowe są sobie równe *w przybliżeniu*.

Obliczenia zmiennopozycyjne, cd.

A jak to jest z mniej zaawansowanymi obliczeniami? Czy jeśli zsumuję 7 razy wartość $\frac{1}{7}$ to dostanę w wyniku 1?

A co z innymi wartościami ułamkowymi ($\frac{1}{2}$, $\frac{1}{3}$, itd.)?

Żeby to sprawdzić dla większej liczby przypadków na raz napiszemy pętlę, która przejdzie przez odpowiednie wartości mianownika (powiedzmy od 2 do 9), wykona stosowne obliczenia i sprawdzi, czy wynik jest 1, czy nie. Wykorzystamy najpierw pętlę `while`, a kiedy uruchominy ten kod, zastąpimy pętlę `while` pętlą `for`.

Z małych zadanie trzecie

Zadanie ma trzy części:

1. Napisanie funkcji liczącej powierzchnię wielokąta foremnego o zadanej długości boku.
2. Napisanie funkcji liczącej powierzchnię koła o zadanym promieniu.
3. Sprawdzenie na trzech zestawach testowych, że funkcje dają poprawne wyniki (z pewnym przybliżeniem oczywiście, bo jesteśmy skazani na obliczenia niedokładne). Do tego przyda się osobna funkcja sprawdzająca, czy dwie wartości `double` mają w przybliżeniu taką samą wartość.

Prace rozdzielamy na podstawie parzystości numeru albumu:

parzysty → sześciokąt foremny (*ang. regular hexagon*)

nieparzysty → pięciokąt foremny (*ang. regular pentagon*)

Szczegóły implementacyjne

Prototypy funkcji powinny wyglądać następująco:

```
double hexagon_area(double edge_len);
double pentagon_area(double edge_len);
double circle_area(double radius);
bool approx_equal(double first, double second,
                  double eps = 1e-6);
```

Pierwsze trzy są dość oczywiste – liczą powierzchnię wielokąta foremnego.

Funkcja `approx_equal` ma sprawdzić, czy jej argumenty nie różnią się o więcej niż podany z wartością domyślną (jedna milionowa) argument `eps`. W funkcji trzeba policzyć różnicę między argumentami `first` i `double`, a następnie sprawdzić, czy wartość bezwzględna tej różnicy jest mniejsza równa `eps` (wartością funkcji jest w tym przypadku `true`; w pozostałych przypadkach `false`).

Szczegóły implementacyjne

Dlaczego jest potrzebna `approx_equal`? Spójrzmy na moje problemy z powierzchnią pięciokąta foremnego (na ćwiczeniach – kwadratu). Znalazłem stronę, na której mogę policzyć powierzchnię pięciokąta i przygotowałem sobie dwa przypadki testowe dla długości boku 1.0 i 2.5:

```
edge_len = 1.0   ->   pentagon_area = 1.72048
edge_len = 2.5   ->   pentagon_area = 10.753
```

Po znalezieniu stosownego wzoru napisałem funkcję `pentagonArea` i następujący test:

```
if (pentagon_area(1.0) != 1.72048)
    cout << "Error!!! pentagon_area(1.0) != 1.72048\n";
```

I dowiedziałem się, że moja funkcja nie działa! A przecież:

```
cout << pentagon_area(1.0) << endl;
```

pokazuje dokładnie `1.72048`

Szczegóły implementacyjne

Operatory == i != świetnie działają dla liczb całkowitych, ale przy zmiennopozycyjnych są zwodnicze. Działamy z ograniczoną dokładnością i nieco inna kolejność obliczeń może dać minimalnie inny wynik. Tu nie mam pojęcia o tym, jak była liczona powierzchnia (nie analizowałem kodu na stronie kalkulatora), a co gorsza spisałem tylko 6 znaczących cyfr wyniku. Nie zaskakuje mnie, że:

```
if (!approx_equal(pentagon_area(1.0), 1.72048))
    cout << "Error!!! pentagon_area(1.0) != 1.72048\n";
```

też daje błąd. Domyślna wartość `eps` to 0.000001, a ja mam tylko 5 cyfr po przecinku. W tym przypadku powinienem zapisać test tak:

```
if (!approxEqual(pentagon_area(1.0), 1.72048, 0.0001))
    cout << "Error!!! pentagon_area(1.0) != 1.72048\n";
```

Teraz wreszcie ten przypadek testowy działa.

Dostarczanie rozwiązania

Rozwiązaniem zadania jest plik **`zad_m4.cpp`** (dokładnie tak ma się nazywać). Proszę złożyć tylko ten jeden plik.

Na początku pliku trzeba umieścić w komentarzu imię, nazwisko i nr albumu autora.

Pod nagłówkiem `Tydzień 3: 18-22.10` znajdziecie Państwo zadanie zatytułowane `Powierzchnie różne`. W tym zadaniu każdy z Was powinien załadować plik `zad_m4.cpp` do:

27 października 2020 23:59