Debugowanie w Code::Blocks

Poznałeś już wiele przydatnych technik programistycznych, ale namierzanie błędów w złożonych programach nadal może być trudne. Na szczęście istnieje narzędzie służące w tym pomocą. Jest to **debugger**. Debugger umożliwia sprawdzanie stanu programu podczas jego działania, ułatwiając tym samym zrozumienie, co tak naprawdę się w nim dzieje. Początkujący programiści często odkładają na później naukę stosowania tego narzędzia, ponieważ korzystanie z niego wydaje się im uciążliwe albo zbędne. To fakt, aby korzystać z debuggera, trzeba go poznać, ale ignorowanie go to jak dostrzeganie samych tylko drzew tam, gdzie rośnie wielki las. Debuggery pozwalają zaoszczędzić mnóstwo czasu, a w porównaniu z wcześniejszym pełzaniem korzystanie z nich przypomina chodzenie. Będzie Ci potrzebna praktyka i na początku będziesz się potykać, ale gdy już zaprzęgniesz debugger do pracy, będziesz mknąć do przodu niczym wiatr.

W rozdziale tym omówię debugger środowiska Code::Blocks, ponieważ jeśli pracujesz w systemie Windows i przeprowadziłeś konfigurację Code::Blocks zgodnie z wcześniejszymi wskazówkami, powinien on być już u Ciebie zainstalowany. Istnieje wiele różnych debuggerów, ale większość pojęć zawiązanych z ich użyciem jest wspólna. Zamieściłem tu wiele zrzutów ekranowych, dzięki czemu nawet jeśli nie korzystasz z Windows, będziesz mógł przeczytać ten rozdział i zobaczyć, jak wygląda debugger. Twoje środowisko programistyczne niemal na pewno jest wyposażone we własny debugger[1].

W rozdziale tym będę korzystał z programów zawierających błędy w celu zaprezentowania rzeczywistego procesu debugowania. Jeśli tylko zechcesz prześledzić opisywane przeze mnie przypadki, będziesz mógł utworzyć w Code::Blocks (albo w innym środowisku, w którym pracujesz) odrębny projekt dla każdego z tych przykładów.

Pierwszy program powinien dla podanej kwoty naliczać odsetki na podstawie procenta składanego. Niestety, zawiera on jakiś błąd, wskutek czego wyświetla niewłaściwą kwotę.

Przykładowy kod 44.: *debugowanie1.cpp*

```
#include <iostream>
using namespace std;

double obliczOdsetki (double podstawa, double oprocentowanie, int lata)
{
```

[1] Jeśli korzystasz z Linuksa, będziesz mógł użyć GDB. Jeżeli pracujesz z Visual Studio albo Visual Studio Express, będziesz mieć do dyspozycji dołączone do nich bardzo dobre debuggery. Istnieją także debuggery samodzielne, z których możesz korzystać, ale ich omówienie wykracza poza zakres tej książki. Należy do nich na przykład WinDbg, który stanowi część Debugging Tools for Windows Microsoftu: *http://www.microsoft.com/whdc/devtools/debugging/default.mspx*. Xcode firmy Apple również udostępnia debugger.

```
        double koncowy_mnoznik;
        for ( int i = 0; i < lata; i++ )
        {
              koncowy_mnoznik *= (1 + oprocentowanie);
        }
        return podstawa * koncowy_mnoznik;
}

int main ()
{
        double podstawa;
        double oprocentowanie;
        int lata;
        cout << "Podaj podstawę: ";
        cin >> podstawa;
        cout << "Podaj oprocentowanie: ";
        cin >> oprocentowanie;
        cout << "Podaj liczbę lat: ";
        cin >> lata;
        cout << "Po " << lata << " latach będziesz mieć " << obliczOdsetki( podstawa,
        ➥oprocentowanie, lata ) << " złotych" << endl;
}
```

Oto przykładowy rezultat działania tego programu:

```
Podaj podstawę: 100
Podaj oprocentowanie: .1
Podaj liczbę lat: 1
Po 1 latach będziesz mieć 1.40619e-306 złotych
```

Niedobrze! 1.40619e-306 złotych to z pewnością nieprawidłowa kwota. Najwidoczniej mamy w programie jakiś błąd. Spróbujmy uruchomić kod w debuggerze, aby stwierdzić, skąd biorą się nasze problemy.

Zaczynamy

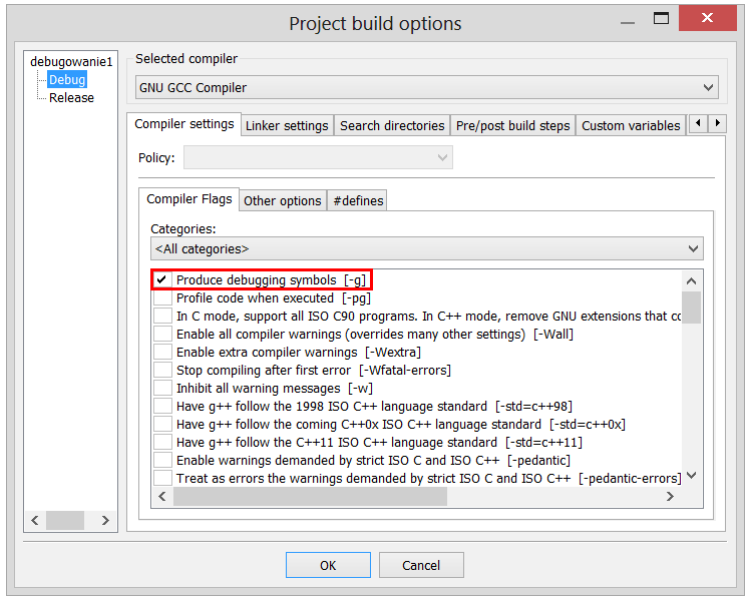
Najpierw, aby ułatwić sobie czynności debugowania, upewnijmy się, że środowisko Code::Blocks jest poprawnie skonfigurowane. W tym celu musimy utworzyć tak zwane **symbole debugowania**. Pomagają one debuggerowi zorientować się, który wiersz kodu jest akurat wykonywany, dzięki czemu będziemy wiedzieć, gdzie w programie się znajdujemy. Aby określić symbole debugowania, z menu *Project* wybierz polecenie *Build options*. Powinno wyświetlić się poniższe okno dialogowe (zobacz pierwszy rysunek na kolejnej stronie).

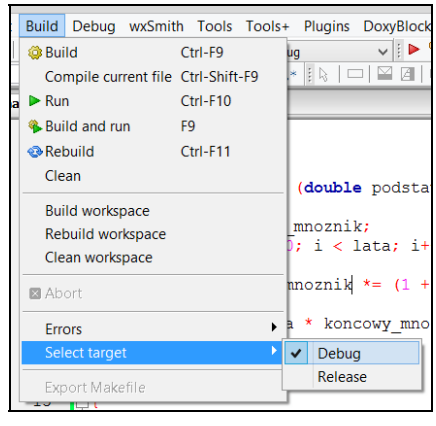
Upewnij się, czy dla pozycji *Debug* zaznaczona jest opcja *Produce debugging symbols [-g]*. Powinieneś także sprawdzić, czy jako cel kompilacji zaznaczono *Debug* (menu *Build/Select target/Debug*) (zobacz drugi rysunek na kolejnej stronie).

Dzięki temu zagwarantujesz, że Twój program będzie kompilowany przy użyciu symboli debugowania, które zostały skonfigurowane dla celu *Debug*.

Jeśli nie widzisz ani celu *Debug*, ani *Release*, zaznacz po prostu pole wyboru *Produce debugging symbols [-g]* dla swojego bieżącego celu kompilacji[2]. Upewnij się też, że opcja *Strip all symbols*

[2] Jeśli korzystasz z kompilatora g++, w celu utworzenia symboli kompilacji będziesz musiał w wierszu poleceń wpisać argument `-g`. Jeżeli używasz Xcode, on sam zatroszczy się o dołączenie odpowiednich symboli.





from binary (minimizes size) [-s] NIE jest zaznaczona (zazwyczaj cele kompilacji są definiowane podczas tworzenia projektu; najłatwiej zagwarantujesz, że poszczególne ustawienia będą poprawne, jeśli pozostawisz domyślne opcje proponowane przez Code::Blocks podczas konfigurowania projektu).

Kiedy już wszystko skonfigurujesz, będziesz mógł przystąpić do działania. Jeśli kompilowałeś swój program wcześniej, ale musiałeś zmienić jego konfigurację, będziesz musiał ponownie go skompilować. Kiedy już to zrobisz, możesz zabrać się za debugowanie!

Wstrzymywanie działania programu

Jedną z zalet debuggera jest możliwość podejrzenia tego, co robi program — który fragment kodu jest wykonywany i jakie są wartości jego zmiennych. Aby zapoznać się z tymi informacjami, będziemy musieli **wstrzymać** działanie programu. W tym celu ustawimy gdzieś w kodzie

punkt wstrzymania i uruchomimy nasz program pod kontrolą debuggera. Debugger zacznie wykonywać program, aż natrafi na linię z punktem wstrzymania. W tym miejscu będziesz mógł rozejrzeć się po swoim kodzie albo przejść go wiersz po wierszu i sprawdzić, w jaki sposób instrukcje w tych wierszach wpływają na Twoje zmienne.

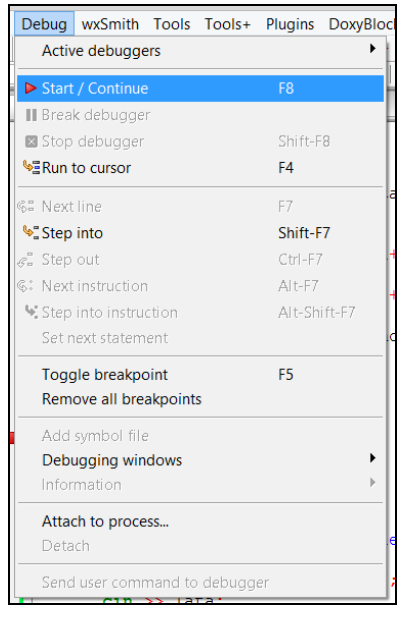
Zdefiniujmy dość wczesny punkt wstrzymania, na samym początku funkcji `main`, żebyśmy mogli prześledzić wykonywanie się całego programu. W tym celu ustaw kursor w wierszu

```
double podstawa;
```

i wybierz polecenie *Debug/Toggle breakpoint* (albo naciśnij *F5*). Na pasku bocznym, obok wybranego wiersza kodu, zostanie umieszczona mała czerwona kropka — oznacza ona, że w tym miejscu znajduje się punkt wstrzymania.

```
15  int main ()
16  {
17      double podstawa;
18      double oprocentowanie;
19      int lata;
20      cout << "Podaj podstawe: ";
21      cin >> podstawa;
22      cout << "Podaj oprocentowanie: ";
23      cin >> oprocentowanie;
24      cout << "Podaj liczbe lat: ";
25      cin >> lata;
```

Punkt ten można włączać i wyłączać, korzystając z polecenia *Toggle breakpoint*; można go także klikać. Teraz, kiedy masz już punkt wstrzymania, możesz uruchomić program! Wywołaj polecenie *Debug/Start* albo naciśnij klawisz *F8*.



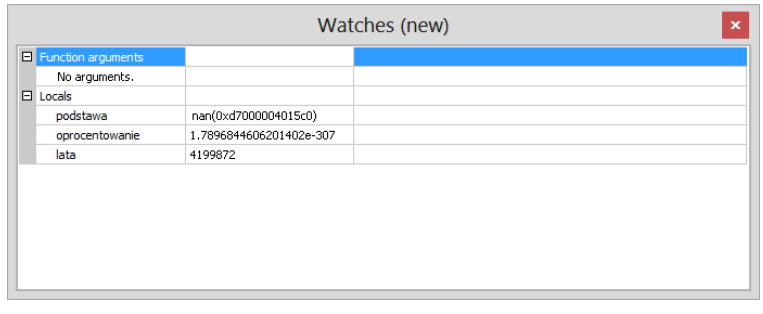
Gdy tak postąpisz, program zacznie wykonywać się jak zwykle, aż dotrze do punktu wstrzymania. W naszym przypadku stanie się to praktycznie od razu, ponieważ punkt wstrzymania znajduje się w pierwszej linii programu.

Teraz powinieneś zobaczyć, jak otwiera się okno debuggera, które będzie wyglądać mniej więcej tak (mogą zostać otwarte jeszcze inne okna, ale o nich opowiem już za chwilę):

```
15  int main ()
16  {
17      double podstawa;
18      double oprocentowanie;
19      int lata;
20      cout << "Podaj podstawe: ";
21      cin >> podstawa;
22      cout << "Podaj oprocentowanie: ";
23      cin >> oprocentowanie;
24      cout << "Podaj liczbe lat: ";
25      cin >> lata;
26
27      cout << "Po " << lata << " latach bedziesz miec " << obliczOdsetki( podstawa, oprocentowanie, lata ) << " zlotych" << endl;
28  }
```

Należy zwrócić uwagę przede wszystkim na żółty trójkąt, widoczny pod czerwoną kropką. Trójkąt ten wskazuje linię kodu, która zostanie wykonana w następnej kolejności. Znajduje się on kilka wierszy pod naszą kropką. Trójkąt nie jest położony bezpośrednio pod nią, ponieważ tak naprawdę nie ma żadnego kodu maszynowego (tj. kodu wykonywanego przez procesor, który jest wynikiem kompilacji kodu w C++), związanego z deklaracjami zmiennych, tak więc nasz punkt wstrzymania, chociaż wydaje się znajdować w wierszu 17., tak naprawdę jest umieszczony w wierszu 20. (liczby widoczne z lewej strony kropki i trójkąta to numery wierszy).

Powinno także zostać otwarte okno *Watches* — będzie ono wyglądać mniej więcej tak (wartości, które zobaczysz, mogą być inne):



Jeśli go nie widzisz, rozejrzyj się za nim; może być schowane za innymi oknami debuggera.

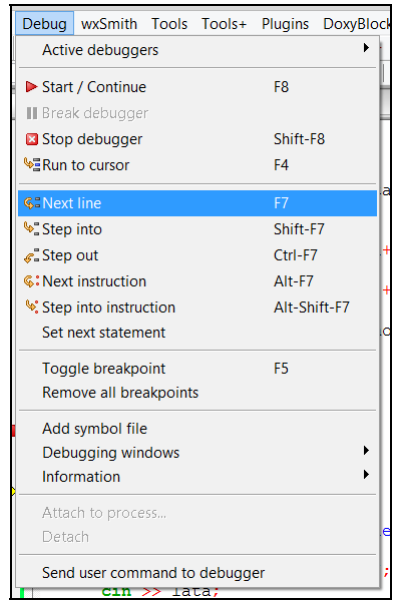
W oknie *Watch* rozwinąłem dwie pozycje — *Locals* oraz *Function Arguments*. Okno to pokazuje wszystkie aktualnie dostępne zmienne — zarówno zmienne lokalne, jak i argumenty funkcji — oraz ich wartości. Zauważ, że wartości wyglądają na pozbawione jakiegokolwiek sensu! Jest tak dlatego, że jeszcze ich nie zainicjalizowaliśmy; stanie się to dopiero w kilku następnych wierszach programu[3].

W celu wykonania kilku następnych wierszy programu musimy poprosić debugger, aby przeszedł do następnej linii kodu. Przejście do kolejnego wiersza spowoduje wykonanie się bieżącej linii (oznaczonej żółtą strzałką). W Code::Blocks służy do tego polecenie *Next line*, czyli następny wiersz (zobacz rysunek na kolejnej stronie).

Zamiast z polecenia *Next line* można skorzystać ze skrótu klawiaturowego *F7*[4].

[3] Pamiętaj, że zmienne nie są inicjalizowane podczas ich deklarowania.

[4] Być może zastanawiasz się, dlaczego istnieje zarówno polecenie *Next line*, jak i *Next instruction*. My zawsze będziemy korzystać z polecenia *Next line*. *Next instruction* jest używane podczas debugowania bez symboli, co wykracza poza zakres tej książki.

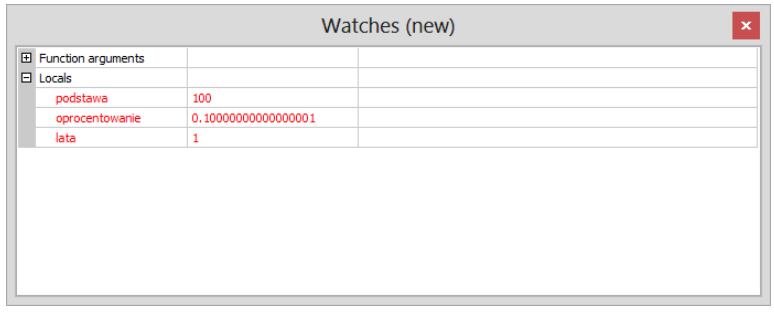


Kiedy już przejdziemy do następnego wiersza, program wywoła instrukcję `cout` i wyświetli na ekranie komunikat z prośbą o wprowadzenie wartości. Jeśli jednak spróbujesz wpisać jakąś kwotę, nie uda Ci się; program powróci do debuggera. Jeszcze raz naciśnij *F7*, aby wykonać następny wiersz kodu. Teraz program będzie czekać, aż użytkownik poda kwotę, ponieważ funkcja `cin` nie zwróciła jeszcze wartości — do tego potrzebne będą informacje uzyskane od użytkownika. Dla zachowania zgodności z naszym raportem dotyczącym błędów wpisz 100, po czym podaj wartości dwóch kolejnych zmiennych, które wprowadziliśmy za pierwszym razem: .1 dla oprocentowania i 1 dla liczby lat do obliczenia procenta składanego.

Teraz dotarliśmy do następującego wiersza:

```
cout << "Po " << lata << " latach będziesz mieć " << obliczOdsetki( podstawa,
↪oprocentowanie, lata ) << " złotych" << endl;
```

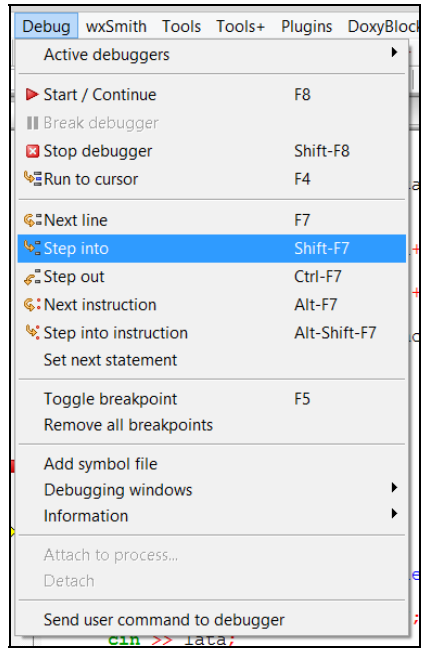
Sprawdźmy jeszcze raz, czy prawidłowo wprowadziliśmy dane. W tym celu można skorzystać z okna *Watch* i przyjrzeć się wartościom zmiennych lokalnych.



Jak dotąd wszystko jest w porządku. Podstawa wynosi 100, oprocentowanie .1, a liczba lat 1. Co mówisz? Że oprocentowanie nie jest równe .1? Masz rację. Oprocentowanie w rzeczywistości wynosi .10000000000000001, ale ta mała jedynka na końcu to tylko zaburzenie wyni-

kające ze sposobu, w jaki reprezentowane są liczby zmiennoprzecinkowe. Pamiętaj, że liczby te nie są idealnie dokładne. Różnice wynikające z reprezentacji tych liczb są jednak na tyle małe, że w większości programów nie będą one mieć znaczenia[5].

Teraz, kiedy już wiemy, że jak do tej pory wszystko jest w porządku, sprawdźmy, co się dzieje wewnątrz funkcji `oblicz0dsetki`. W tym celu należy wywołać kolejne polecenie debuggera, którym jest *Step into*:



Polecenie *Step into* służy do wejścia do funkcji, która ma zostać wywołana z bieżącego wiersza — w przeciwieństwie do *Next line*, które powoduje wykonanie się całej funkcji i zwrócenie jej wyniku, jak to widzieliśmy w przypadku funkcji `cin`. Z polecenia *Step into* należy korzystać w celu debugowania wnętrza funkcji, co już za chwilę zrobimy.

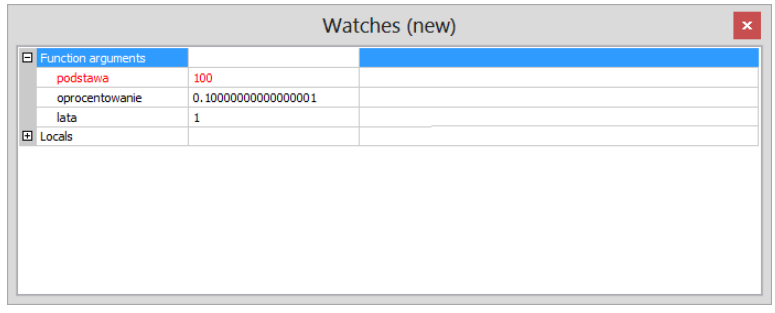
Wkroczmy zatem do funkcji `oblicz0dsetki`. Być może zastanawiasz się, czy w poniższym wierszu nie ma przypadkiem wielu wywołań funkcji.

```
cout << "Po " << lata << " latach będziesz mieć " << obliczOdsetki( podstawa,
↪oprocentowanie, lata ) << " złotych" << endl;
```

Co z tymi wszystkimi wywołaniami funkcji `cout`? Debugger Code::Blocks jest bardzo sprytny — nie wkroczy do funkcji pochodzących z biblioteki standardowej. Możemy po prostu przejść do naszego wiersza i od razu znajdziemy się w funkcji `oblicz0dsetki` z pominięciem funkcji, które nas nie interesują. Zróbmy to teraz.

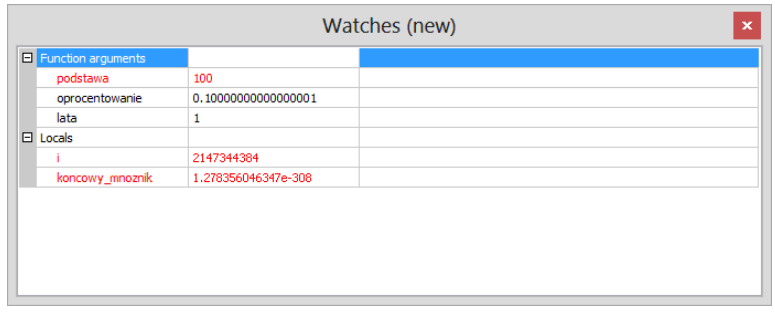
Teraz, gdy znajdujemy się wewnątrz funkcji `oblicz0dsetki`, pierwsze, co musimy zrobić, to sprawdzić, czy argumenty funkcji są prawidłowe — być może pomieszaliśmy kolejność argumentów. Rozszerzmy sekcję *Function Arguments* w oknie *Watch*:

[5] Prawdą jest jednak, że błędy zmiennoprzecinkowe mogą się nawarstwiać, co w przypadku niektórych aplikacji może wywołać poważne konsekwencje. Niemniej naszego programu to nie dotyczy.



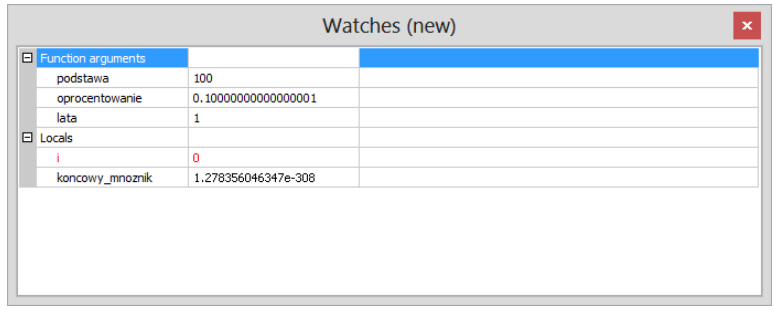
Wszystko wygląda dobrze!

Spójrzmy na zmienne lokalne:



Czy widzisz coś dziwnego? Zarówno zmienna `i`, jak i `koncowy_mnoznik` wyglądają na zupełnie pozbawione sensu! Pamiętaj jednak, że kiedy poprzednim razem zaglądaliśmy do okna *Watch*, widzieliśmy bezsensowne wartości, ponieważ nie były one jeszcze zainicjalizowane. Wybierzmy polecenie *Next line* (*F7*), aby przeprowadzić inicjalizacje związane z naszą pętlą i zobaczyć, co się wtedy stanie.

Inicjalizacja pętli odbywa się w jednym tylko wierszu, tak więc możemy ponownie sprawdzić zmienne lokalne. Powinny wyglądać mniej więcej tak:



Z `i` jest już wszystko w porządku, ale co ze zmienną `koncowy_mnoznik`? Nie wygląda na to, aby była poprawnie zainicjalizowana. Co więcej, już za chwilę w wierszu, w którym się znajdujemy, zostanie ona użyta:

```
koncowy_mnoznik *= (1 + oprocentowanie);
```

Wiersz ten nakazuje wykonać mnożenie `koncowy_mnoznik * (1 + oprocentowanie)` i przypisać otrzymany wynik do zmiennej `koncowy_mnoznik`, ale jak widzimy, jej wartość jest wzięta z sufitu, w związku z czym mnożenie da w rezultacie wynik pozbawiony sensu.

Czy wiesz, jak naprawić taki błąd?

W wierszu, w którym jest deklarowana zmienna `koncowy_mnoznik`, musimy ją zainicjalizować; w naszym przypadku powinniśmy przypisać jej wartość 1.

To tyle. Namierzyliśmy problem i znaleźliśmy rozwiązanie. Dziękujemy ci, debuggerze!

Temat: Debugowanie w C++

Zadania do samodzielnej realizacji:

1. (2p) Program ma za zadanie obliczyć wartość *podstawa* do potęgi *wykl.* Poniżej przedstawiony jest kod programu, który zawiera błędy. Proszę przetestować poniższy program oraz naprawić błędy.
   W pliku tekstowym należy opisać błędy zawarte w programie cytując linię kodu, opisując błąd i proponując poprawną linię kodu.

   Wysyłając rozwiązanie należy załączyć poprawny kod oraz plik tekstowy zawierający komentarze.

```
1   #include <iostream>
2
3   using namespace std;
4
5   int wykladnik (int podstawa, int wykl){
6       int wartosc_narast;
7       for(int i=0; i<wykl; i++){
8           wartosc_narast *= podstawa;
9       }
10      return podstawa;
11  }
12
13  int main(){
14      int podstawa;
15      int wykl;
16
17      cout << "Podaj podstawe:";
18      cin>> podstawa;
19      cout<< "Podaj wykladnik:";
20      cin>> wykl;
21      cout<<wykladnik(wykl, podstawa);
22  }
23
```

2. (3p) Poniższy program powinien sumować wartości tablicy dynamicznej. Proszę przeanalizować kod i znaleźć błędy.
   W pliku tekstowym należy opisać błędy zawarte w programie cytując linię kodu, opisując błąd i proponując poprawną linię kodu.

   Wysyłając rozwiązanie należy załączyć poprawny kod oraz plik tekstowy zawierający komentarze.

```cpp
#include <iostream>
using namespace std;

int sumujWartosci (int *wartosci, int n)
{
    int suma;
    for ( int i = 0; i <=n; i++ )
    {
        suma += wartosci[i];
    }
    return suma;
}

int main()
{
    int rozmiar;
    cout<<"Podaj rozmiar:";
    cin>> rozmiar;
    int *wartosci = new int [rozmiar];
    int i;
    while (i<rozmiar)
    {
        cout<<"Podaj wartosc do dodania: ";
        cin>> wartosci[++i];
    }
    cout<<"Laczna suma wynosi:"<<sumujWartosci(wartosci, rozmiar);
}
```

Dla każdego z poniższych programów proszę przygotować plik w Wordzie (*lab14_z1.docx, lab14_z2.docx*) opisując błędy zawarte w programie wraz z ich rozwiązaniami.

Należy odwołać się do numeru linii, opisać z czego wynika błąd i w jaki sposób go naprawić (podać prawidłową linię kodu).

Prawidłowe rozwiązanie zadania wraz z komentarzami w arkuszu tekstowym.

3. (3p) Zaproponuj zadanie warte 3 punkty, które mogłoby być zadaniem „domowym” w przyszłym roku. Zadanie powinno dotyczyć jednego z tematów:
    1) Łańcuchy znaków w języku C++
    2) Obsługa plików w języku C++
    3) Wskaźniki w C++
    4) Struktury danych w języku C++
    5) Dziedziczenie w języku C++
    6) Przeciążanie operatorów

Rozwiąż zaproponowane zadanie. Napisz (w pliku tekstowym) krótkie omówienie spodziewanych trudności, które studenci mogą mieć przy wykonywaniu zaproponowanego zadania.