Rozdział 2. Zaawansowane operacje w środowisku programistycznym

Celem niniejszego rozdziału jest zaprezentowanie Studentom możliwości testowania i śledzenia poprawności zakodowanych w programie obliczeń. Proces tworzenie nawet bardzo prostych programów wymaga często od programisty wprowadzenia wielu poprawek, gdyż trudno jest być bezbłędnym w pisaniu programów (jest to wręcz niemożliwe). Błędy powstałe w trakcie tworzenia programu dzielimy na 3 grupy:

* **Błędy kompilacji (składniowe)** - czyli niezgodności kodu programu ze standardem języka C++ (np. nieumieszczenie średnika tam, gdzie być powinien),
* **Błędy wykonania (semantyczne)** - czyli błędy, jakie wywołuje dany program w trakcie jego wykonywania (np. zapętlenie, dzielenie przez 0 itp.),
* **Błędy logiczne** - czyli niezgodności między zamiarem programisty a tym, co program faktycznie robi (np. przypisanie pewnej zmiennej innej niż zamierzona wartości początkowej, wywołanie procedury ze złymi parametrami itp.).

2.1. Testowanie i usuwanie usterek z programu

Z poprzedniego rozdziału wiemy, że środowisko Code::Blocks w jasny sposób sygnalizuje zaistnienie błędów kompilacji, jesteśmy również informowani o ewentualnym wystąpieniu błędu wykonania. Niestety, nie ma uniwersalnej metody radzenia sobie z błędami logicznymi, które zresztą najtrudniej znaleźć i usunąć. W dalszej części rozdziału na przykładach zaprezentowana zostanie metoda radzenia sobie z tym problemem, udostępniona w środowisku Code::Blocks.

***Przykład 2.1.***

*Dane są dwie dowolne liczby całkowite. Napisz program obliczający iloczyn tych dwu liczb korzystając z operatorów addytywnych.*

W przypadku tego zadania Studentom należą się pewne wyjaśnienia. Dlaczego w zadaniu zabroniono korzystania z operatora mnożenia („\*”)? Otóż, programowanie składa się z szeregu operacji, które nie występują w standardzie języka – przykładem mogą być operacje wejścia/wyjścia, inne to obliczanie wartości funkcji trygonometrycznych jak np. sin(x), cos(x), obliczanie sumy elementów tablicy czy znalezienie maksymalnego elementu z liczb zapisanych w pliku. W efekcie programując rozwiązanie zadania prawie zawsze spotykamy się z sytuacją, w której brakuje nam jakiegoś typu operacji. Zaletą problemu wybranego do rozwiązania jest jego prostota, co jest ważne na początku nauki programowania.

***Rozwiązanie 1 Przykładu 2.1*** (źródło: opracowanie własne)

Należy zastanowić się nad rozwiązaniem. Pierwsze podejście można zapisać następująco:

1. Wczytaj dane: dwie liczby,
2. Oblicz iloczyn liczb nie korzystając z mnożenia,
3. Wypisz wynik.

Zaprezentowany powyżej sposób postępowania nie nadaje się jeszcze do implementacji (kodowania). Wobec tego należy doprecyzować wylistowane wyżej operacje. I tak:

* Ad.1. Należy zaproponować nazwy zmiennych, niech to będą zmienne **x** oraz **y**;
* Ad.3. Konieczne jest wypisanie wyniku, więc najłatwiej pamiętać go w zmiennej, niech to będzie zmienna o nazwie **iloczyn**;
* Ad.2. Ta operacja jest najtrudniejsza. W takiej sytuacji dobrze jest wziąć przykładowe dane wejściowe[[1]](#footnote-1). Niech to będą dane postaci: 3 i 4. Czyli w dalszych rozważaniach przyjmiemy, że na wejściu do programu podane są x=3, y=4. Po wymyśleniu rozwiązania dla tych danych zostanie ono uogólnione dla dowolnego zestawu danych:
* Dla tych danych wejściowych rozwiązanie wydaje się proste, należy 3 razy dodać liczbę 4 (lub odwrotnie: 4 razy liczbę 3).
* Jak to zrobić w przypadku danych uogólnionych? Też wydaje się to proste. Należy x-razy dodawać liczbę y, a precyzyjniej: do zmiennej iloczyn należy dodać zmienną y powtarzając to x-razy. Na każdym etapie wynik przechowujemy w zmiennej iloczyn. Moglibyśmy to zapisać jako: iloczyn=iloczyn + y;[[2]](#footnote-2) . I tak x-razy.
* Pojawia się nowy problem. Mianowicie, tekst programu jest strukturą statyczną, a tu instrukcję „iloczyn = iloczyn + y;” będziemy musieli zapisać w liczbie zależnej od danych. Nie jest to dobry sposób.
* Lepszym sposobem jest uzależnienie liczby operacji dodawania od wartości danych, w ten sposób, że po każdej operacji zmieniamy wartość zmiennej, która określa, ile razy dodajemy. Można to zrobić za pomocą instrukcji „x = x - 1;”. Kiedy przerwać powtarzanie tych instrukcji? Wtedy, kiedy zmienna x będzie równa zero.

***Rozwiązanie 2 Przykładu 2.1*** (Poprawione rozwiązanie, źródło: opracowanie własne)

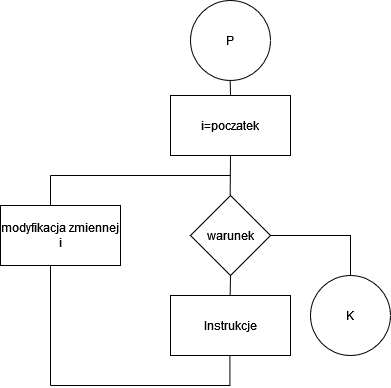
Po tych wyjaśnieniach oraz dysponując schematami pokazanymi w poprzednim przykładzie można rozwiązanie zapisać w postaci następujący kroków:

1. Wczytaj dwie liczby, dla wygody nazwijmy je x oraz y,
2. x razy wykonaj następujące kroki:
   1. iloczyn = iloczyn + y;
3. wypisz wartość zmiennej iloczyn.

Zajmiemy się teraz drugim problemem. Spróbujmy myślowo wykonać (zamiast komputera) powyższy tekst programu. Otóż, w instrukcji iloczyn = iloczyn + y; nie wiemy co mamy wstawić w miejsce zmiennej iloczyn za pierwszym razem. By ten problem rozwiązać, należy przed pierwszym wywołaniem tego wyrażenia zainicjować wartość początkową zmiennej iloczyn taką wartością, by końcowy wynik nie został zmieniony. Ponadto należy to zrobić jednokrotnie na początku programu. Rozwiązanie polega na dodaniu instrukcji iloczyn = 0; przed pętlą „for”.

Kolejnym problemem jest to, jak zapisać fakt, że jakąś operację mamy wykonać x razy. Jest oczywiste, że liczbę wykonywanych operacji musimy kontrolować za pomocą zmiennej, ten rodzaj zmiennej jest nazywany zmienną sterującą, każda zmienna sterująca zmienia się w pewnym zakresie wynikającym z danego algorytmu – częstymi wartościami są albo 0 albo 1. Wykonanie każdej operacji powoduje zwiększanie tej zmiennej o 1 (w przypadku różnych algorytmów bardziej logicznym rozwiązaniem może być inna modyfikacja). A liczbę wykonywanych iteracji możemy kontrolować za pomocą odpowiedniego warunku. W rozważanym przykładzie warunek kontrolujący liczbę wykonanych operacji można zapisać za pomocą i<=x (gdy wartość początkowa i=1) lub za pomocą i<x (gdy i=0).

Pętla for umożliwia wielokrotne powtarzanie grupy instrukcji w sytuacji, gdy znana jest liczba powtórzeń. Jej działanie można zilustrować następującym schematem blokowym:



Ilustracja 2.1. Schemat blokowy instrukcji for.

Natomiast składnia może być przedstawiona następująco:

for (i=początek; warunek; modyfikacja zmiennej i)

Instrukcje

gdzie „i=początek” służy do nadania wartości początkowej zmiennej sterującej (tu i), warunek określa moment przerwania iteracji (warunek prawdziwy oznacza powtarzanie instrukcji, fałszywy wyjście z pętli) a „modyfikacja zmiennej sterującej i” oznacza zmianę wartości zmiennej, która wpływa na moment zakończenia pętli.

***Rozwiązanie 3 Przykładu 2.1*** (Kolejne uszczegółowione rozwiązanie, źródło: opracowaniewłasne)

Biorąc pod uwagę zaprezentowane powyżej wyjaśnienia można poprawić poprzednie rozwiązanie następująco:

1. wczytaj x; wczytaj y;
2. iloczyn=0;
3. for (i=1; i<=x; i++)
4. iloczyn=iloczyn+y;
5. wypisz iloczyn;

Zanim, na podstawie powyższego rozwiązania 3 zostanie napisany program, należy sprawdzić jego działanie dla danych, od których zaczęto konstruować rozwiązanie, czyli x=3, y=4. Numerowanie wierszy ułatwi śledzenie działania zaproponowanego rozwiązania przedstawionego w Tabeli 2.1.

|  |  |
| --- | --- |
| **Numer wiersza** | **Operacja** |
| 1 | x=3, y=4[[3]](#footnote-3) |
| 2 | iloczyn=0, i=1 |
| 3 | 1<=3?=>tak[[4]](#footnote-4) |
| 4 | iloczyn=0+4[[5]](#footnote-5) |
| 5 | i=1+1<=2 |
| 3 | 2<=3?=>tak |
| 4 | iloczyn=4+4=8 |
| 5 | i=2+1=3 |
| 3 | 3<=3?=>tak |
| 4 | iloczyn=8+4=12 |
| 5 | i=3+1=4 |
| 3 | 4<=3?=>nie |
| 8 | 12 |

**Tabela 2.1**. Sprawdzenie poprawności Rozwiązania 3 Przykładu 2.1

Należy zaznaczyć, że wynik wyszedł poprawny. Ponadto, łatwo zauważyć, że wiersze o numerach 3, 4 i 5 były wykonywane 3-krotnie. Nie powinno to dziwić, gdyż ogólnie dążono do x - krotnego wykonania pewnych instrukcji. Takie powtórzenie nazywane jest **pętlą**, pojedyncze wykonanie pętli to **instrukcja**, tu złożona.

Program, będący formalnym zapisem Rozwiązania 3 Przykładu 2.1 przedstawia,   
Listing 2.1. Zrzut ekranu został pokazany na Rysunku 2.1.

#include <iostream>

using namespace std;

int main()

{

int x,y,iloczyn;

cout<<"Program oblicza iloczyn dwu liczb calkowitych bez operatora \*"<<endl;

cout<<"podaj 1 liczbe:"; cin>>x;

cout<<"podaj 2 liczbe:"; cin>>y;

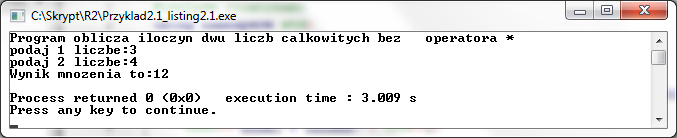
iloczyn=0;

for (int i=1; i<=x; i=i+1)

iloczyn=iloczyn+y;

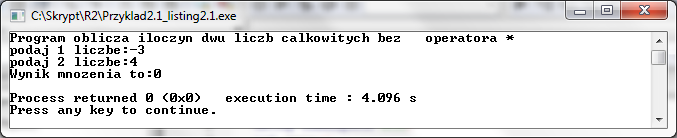
cout<<"Wynik mnozenia to:"<<iloczyn<<endl;

}**Listing 2.1.** Kod programu uwzględniający Rozwiązanie 3 Przykładu2.1



**Rysunek 2.1**. Wynik działania programu pokazany na Listingu 2.1, źródło: opracowanie własne

Jak widać z Rysunku 2.1 wynik obliczeń jest taki sam, jak przedstawiony w Tabeli 2.1. Ponieważ jeden zestaw testowych danych to zbyt mało, aby można było wnioskować o poprawności stworzonego programu, to program zostanie uruchomiony dla innych danych, np. x=-3, y=4. Wynik przedstawia Rysunek 2.2.



**Rysunek 2.2.** Wynik działania programu pokazanego na listingu 2.1 dla innych danych testowych, źródło: opracowanie własne

Niestety, okazuje się, że wynik jest niepoprawny. Wykonując operacje pokazane w Tabeli 2.1 dla nowych danych (to zostawiamy Studentom do samodzielnego wykonania) łatwo znaleźć miejsce, które jest źródłem błędu. Otóż, jak nietrudno zauważyć, dla x=-3 nie wchodzimy w ogóle do pętli for ponieważ nie jest prawdą, że 0>=-3. Skoro tak, to pętla (zgodnie z opisaną wyżej semantyką) jest pomijana i wykonywana następna instrukcja - wypisuje ona wartość zmiennej iloczyn, która w tym przypadku wynosi 0 i taka wartość pojawia się na monitorze.

Oznacza to, że dla x ujemnych (dociekliwym Studentom pozostawiamy do samodzielnej analizy, dlaczego dodatnia lub ujemna wartość zmiennej y nie ma znaczenia dla poprawności działania algorytmu) został zaproponowany niepoprawny algorytm. Jak go można poprawić? Najprościej będzie w przypadku ujemnych wartości x zmienić znak x na przeciwny, wykonać operacje i na końcu ustawić znak dla zmiennej iloczyn w zależności od znaku zmiennej x, dla dodatnich nic nie robimy, dla ujemnych zmieniamy znak na przeciwny. I tu pojawia się kolejny problem – po wykonaniu pętli while tracimy wartość zmiennej x (w poprzednim rozwiązaniu też traciliśmy, ale zmienna x po wykonaniu pętli nie była nam do niczego potrzebna). Ten problem z kolei możemy rozwiązać wprowadzając zmienną pomocniczą, np. o identyfikatorze (nazwie) px. Wynika z tego, że pewne operacje musimy wykonać warunkowo. Można je zapisać następująco:

* jeśli x<0, to px=-x; na przeciwny w przeciwnym razie px=x;

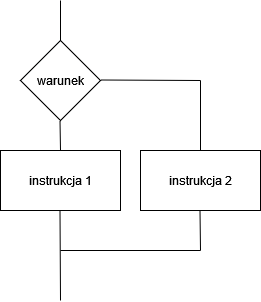
oraz

* jeśli x<0, to iloczyn=-iloczyn;

Takie sytuacje w programowaniu są częste, stąd istnieje **instrukcja warunkowa „if”**, która dopuszcza dwie składnie:

1. if (warunek) instrukcja; //w tym przypadku instrukcja jest wykonywana jeśli warunek jest spełniony;
2. if (warunek) instrukcja1; else instrukcja2; //w tym przypadku instrukcja1 jest wykonywana jeśli warunek jest spełniony, w przeciwnym przypadku wykonywana jest instrukcja2;

Instrukcja warunkowa if umożliwia warunkowe wykonywanie instrukcji. Jej ogólna postać jest postaci:



A składnia instrukcji warunkowej jest postaci:

if (warunek) instrukcja1; else instrukcja2;

Rozwiązanie uwzględniające powyższe uwagi zostało zaprezentowane na poniższym Listingu 2.2.

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int i,x,y,iloczyn;
6. cout<<"Program oblicza iloczyn dwu liczb calkowitych   
    bez operatora \*"<<endl;
7. cout<<"podaj 1 liczbe:"; cin>>x;
8. cout<<"podaj 2 liczbe:"; cin>>y;
9. iloczyn=0;
10. int px;
11. if (x>0)
12. px=x;
13. else
14. px=-x;
15. for (i=1; i<=px; i=i+1)
16. {
17. iloczyn=iloczyn+y;
18. }
19. if (x<0) iloczyn=-iloczyn;
20. cout<<"Wynik mnozenia to:"<<iloczyn<<endl;
21. }

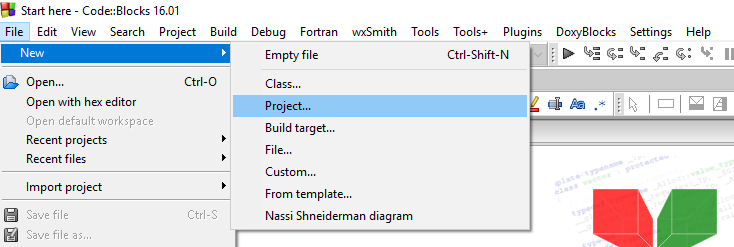
**Listing 2.2** Poprawiony kod programu z listingu 2.1, źródło: opracowanie własne

W ramach pracy własnej można sprawdzić, że powyższy program wykonuje poprawnie obliczenia dla: x=3 i y=4 oraz x=-3 i y=4. Podobnie jest i dla innych liczb całkowitych z odpowiedniego zakresu.

W przypadku testowania większych programów jest oczywiste, że metoda przedstawiona w Tablicy 2.1 jest uciążliwa i podatna na błędy rachunkowe, zwłaszcza w przypadku danych rzeczywistych (spróbujmy wyobrazić sobie testowanie programu dla danych: x=127, y=-345). Naprzeciw tym ograniczeniom wychodzi środowisko Code::Blocks wraz ze swoimi narzędziami. Narzędzie to, nazywane jest z angielska debugowaniem a narzędzie, z którego będziemy korzystać debuggerem. Opiera się ona na krokowym (instrukcja po instrukcji) wykonywaniu programu i analizie wartości poszczególnych zmiennych.

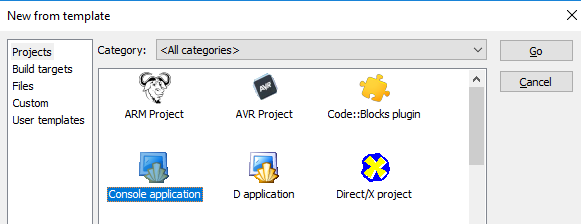
Aby skorzystać z Debuggera w programie Code::Blocks konieczne jest stworzenie Projektu, w którym zaznaczymy, że wymagamy dostępności Debuggera.

Projekt tworzymy wybierając z menu File->New->Project (Rysunek 2.3).



**Rysunek 2.3** Menu środowiska Code::Blocks.

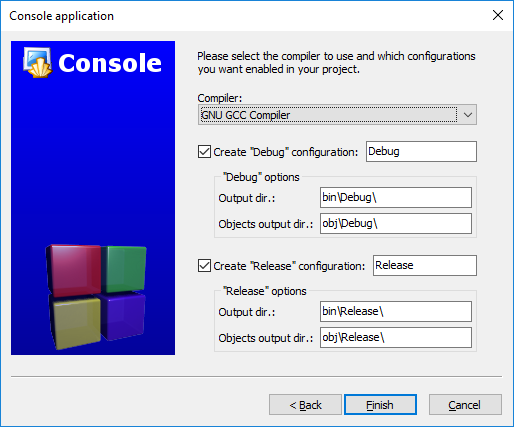
Następnie wybieramy Console Aplication



**Rysunek 2.4** Etap tworzenia nowego projektu w środowisku Code::Blocks.

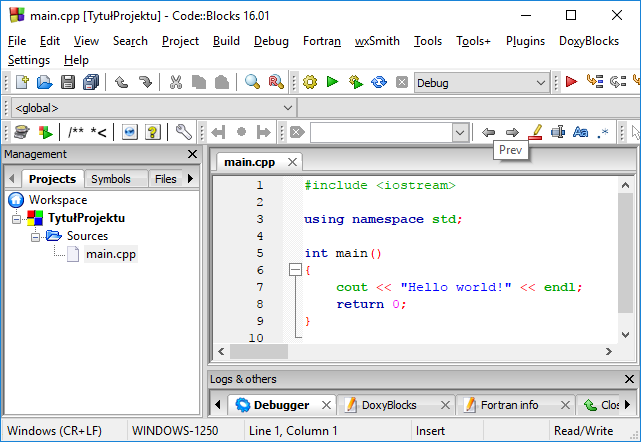
W kolejnym oknie określamy język programowania. W naszym przypadku wybieramy C++. Następnie określamy tytuł projektu oraz lokalizację.

W kolejnym oknie z listy rozwijanej można wybrać kompilator, który będzie stosowany, w tym przypadku GNU GCC Complier. Aby móc korzystać z Debuggera należy to zaznaczyć zgodnie z Rysunkiem 2.5.



**Rysunek 2.5** Konfiguracja nowego projektu w środowisku Code::Blocks.

Efektem będzie stworzenie projektu zawierającego częściowo wypełniony plik main.cpp.



**Rysunek 2.6** Okna środowiska Code::Blocks wraz z otwartym projektem.

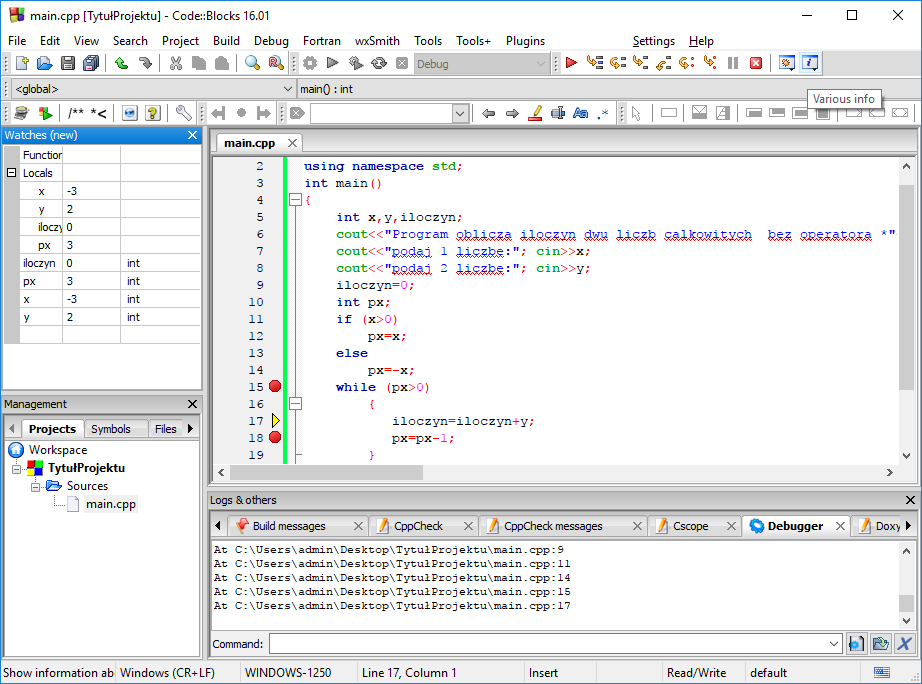
Sterowanie Debuggerem można przeprowadzać poprzez polecenia z menu Debug, lub poprzez pasek narzędzi mający postać:



**Rysunek 2.7.** Pasek narzędzi Debuggera w środowisku Code::Blocks.

Aby wstawić/usunąć „Punty Przerwań” (ang. Toggle breakpoint) klikamy po lewej stronie danej linii wiersza lub mając aktywny kursor w danej linii nacisnąć F5, lub wybrać z menu Debbug->Toggle breakpoint. Aktywny punkt przerwania zaznaczony jest czerwonym okręgiem po lewej stronie linii kodu.

Debuggowanie rozpoczynamy z menu Debug->Start/Continue, poprzez wciśnięcie klawisza F8 lub z paska narzędzi. Po rozpoczęciu debuggowania można wybrać zmienne, których wartości w puntach przerwań będą „śledzone”. Okno wyświetlające zmienne aktywujemy z menu Debugg->Debugging windows->Watches. Dodanie zmiennej do okna Watches można przeprowadzić zaznaczając daną zmienną i wybierając z menu kontekstowego „Watch nazwa\_zmiennej”. Aby wykonywać program krok po kroku należy wybrać Shift-F7 lub przycisk „Step into” z paska narzędzi. Żółty trójkąt po lewej stronie oznacza linię kodu, która jest aktualnie wykonywana. W oknie Watches widzimy wartości obserwowanych zmiennych w każdym kroku programu.



**Rysunek 2.3** Okno środowiska Code::Blocks z aktywnym Debuggerem.

Warto zwrócić uwagę na to, jakie dane powinno się wybierać do testowania programu (Debugger nie rozwiązuje za nas tego problemu, tylko ułatwia wykonanie obliczeń). Oczywiście, nie ma ogólnej zasady, jakie wartości zmiennych powinny być użyte do testowania, doświadczenie w tym zakresie zdobywa się dzięki praktyce. Tym niemniej, w przypadku ostatniego zadania możemy wyodrębnić cztery klasy danych, które powinniśmy rozważać. Są to:

* {(x,y): x>=0, y>=0}, np. x=3, y=4,
* {(x,y): x>=0, y<0}, np. x=3, y=-4,
* {(x,y): x<0, y>=0}, np. x=-3, y=4,
* {(x,y): x<0, y<0}, np. x=-3, y=-4.

Jak wspomniano powyżej, znak zmiennej y nie mają większego znaczenia dla poprawności programu – taki wniosek można wyciągnąć czytając tekst programu i zauważając, że zmienna y jest tylko dodawana do zmiennej iloczyn (jedynym potencjalnym źródłem błędów mogą być zbyt duże lub zbyt małe liczby). Natomiast zmienna x (w pierwotnej wersji rozwiązania) była modyfikowana. I właśnie takie modyfikacje mogą być źródłem błędu. Wobec tego wystarczy rozważyć dwie klasy danych:

* {(x,y): x>=0, y dowolne},
* {(x,y): x<0, y dowolne}.

Warto podkreślić, że zmienne obserwowane nie muszą być typu prostego (liczbowe, znakowe itd.) - mogą to być np. tablice, w przypadku których zostanie wyświetlona cała zawartość (albo ich początek, o ile rozmiar jest duży). Zamiast całej tablicy możemy także obserwować tylko jedną jej pozycję (podając do podglądu tablica[indeks]).

Więcej informacji na temat testowania programów znajduje się w Dodatku A. Istnieją także metody formalnego dowodzenia poprawności, ale wykraczają one istotnie poza zakres niniejszego skryptu.

***Zadanie 2.2***

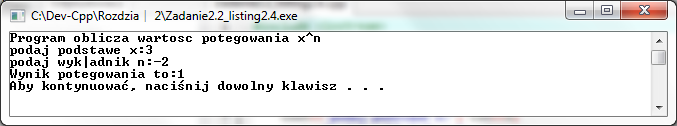
*Napisz program wczytujący z klawiatury dwie liczby* ***x*** *i* ***n****, a następnie obliczający i wyświetlający na ekranie wartość potęgi* ***xn****, przy założeniu, że* ***x*** *jest liczbą zmiennoprzecinkową,* ***n*** *jest liczbą całkowitą (dodatnią, ujemną lub równą 0).*

Zadanie wydaje się podobne do zadania poprzedniego. W rozwiązaniu należy zamiast dodawania jak to było w Zadaniu 2.1 wykonać n-razy operację mnożenia zmiennej x. Załóżmy, że chcemy przechowywać wynik potęgowania pod zmienną o nazwie potega, analogicznie więc należy wykonać n-razy operację potęga=potęga\*x. Oczywiście należy zainicjować wartość zmiennej przed wejściem do pętli. Tu można postawić pytanie czy można podobnie jak w Zadaniu 2.1 zainicjalizować ją z wartością 0 tj. potęga=0 ? Otóż gdybyśmy tak zrobili zawsze na wynikiem działania potęga=potęga\*x będzie wartość 0. W tym przypadku niezmiennikiem operacji mnożenia jest wartość 1, a nie 0 tak jak w przypadku dodawania i odejmowania. Wstępny kod programu będącego rozwiązaniem przedstawia Listing 2.3.

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int i,x,n,potega;
6. cout<<"Program oblicza wartosc potegowania x^n"<<endl;
7. cout<<"podaj podstawe x:"; cin>>x;
8. cout<<"podaj wykładnik n:"; cin>>n;
9. potega=1;
10. for (i=0; i<n; i=i+1)
11. potega=potega\*x;
12. cout<<"Wynik potęgowania to:"<<potega<<endl;
13. system ("pause");
14. }

**Listing 2.3** Rozwiązanie 1 Zadania 2.2, źródło: opracowanie własne

Czy powyższy program działa poprawnie? W przypadku gdy obie zmienne wejściowe są dodatnie wyniki są poprawne. Co jednak, gdy np. n=0? W takim przypadku instrukcje znajdujące się w liniach 12-13 nigdy nie zostaną wykonane, gdyż warunek z linii 10 nigdy nie będzie spełniony. Czy więc wynik będzie poprawny? Tak, dzięki inicjalizacji potęga =1 w linii 9, gdyż dla każdego x mamy, że x0=1. Jak natomiast będzie w przypadku gdy n<0? Taką sytuacje przedstawia Rysunek 2.5.



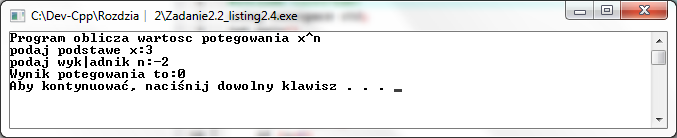
**Rysunek 2.5**. Wynik działania kodu przedstawionego na listingu 2.3 dla przykładowych danych, źródło: opracowanie własne

Widzimy, że wynik nie jest prawidłowy, gdyż również w tym przypadku warunek z linii 10 nigdy nie będzie spełniony. Aby naprawić błąd programu należy uwzględnić, że w przypadku ujemnego wykładnika mamy zależność x-n=1/(xn). W przypadku ujemnego wykładnika należy więc wykonać działania tak jakby wykładnik był ≥0, a na końcu jako wynik wypisać 1/potega. W tym celu należy dodać nową zmienną np. znak, która przechowa informację o wyjściowym znaku wykładnika n. Poprawiony kod prezentuje Listing 2.4.

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int i,x,n, potega;
6. cout<<"Program oblicza wartosc potegowania x^n"<<endl;
7. cout<<"podaj podstawe x:"; cin>>x;
8. cout<<"podaj wykładnik n:"; cin>>n;
9. short znak=0;
10. if (n<0)
11. {
12. n=-n;
13. znak=1;
14. }
15. potega=1;
16. for (i=0; i<n; i=i+1)
17. potega=potega\*x;
18. cout<<"Wynik potegowania to:";
19. if (znak==1)
20. potega=(1/potega);
21. cout<<potega <<endl;
22. }

**Listing 2.4**. Rozwiązanie 2 Zadania 2.2, źródło: opracowanie własne

Wynik uruchomienia powyższego kodu dla zmiennych wejściowych x=3 i n=-2 prezentuje Rysunek 2.6. Widzimy, że wynik jest nieprawidłowy , gdyż otrzymujemy 0, a powinno być 1/9 czyli 0.111(1).



**Rysunek 2.6.** Wynik działania kodu przedstawionego na Listingu 2.4 dla przykładowych danych, źródło: opracowanie własne

Jaki jest powód tego złego wyniku ? Dokładne prześledzenie kodu i użycie debuggera pokaże, że wszystkie działania wykonywane są poprawnie.

W tym przypadku należy zwrócić uwagę na typ zmiennej potęga. W przypadku ujemnego wykładnika potęgowania jako zmienną otrzymujemy liczbę rzeczywistą ≤1. A więc w tym zadaniu wszystkie działania wykonywane są poprawnie, natomiast ponieważ zmienna potega jest typu całkowitego w linii 23 Listingu 2.4 następuje zrzutowanie wyniku na typ całkowity, czyli obcięcie części liczby po przecinku. Aby program zadziałał poprawnie wystarczy zmienić typ zmienne potega na float. Poprawny kod przedstawia Listing 2.5.

Przykład ten ilustruje, że nawet za pomocą debuggera nie zawsze jesteśmy w stanie znaleźć błąd działania programu, zwłaszcza, gdy jest to błąd logiczny.

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int i,x,n;
6. float potega;
7. cout<<"Program oblicza wartosc potegowania x^n"<<endl;
8. cout<<"podaj podstawe x:"; cin>>x;
9. cout<<"podaj wykładnik n:"; cin>>n;
10. short znak=0;
11. if (n<0)
12. {
13. n=-n;
14. znak=1;
15. }
16. potega=1;
17. while (n>0)
18. {
19. potega=potega\*x;
20. n=n-1;
21. }
22. cout<<"Wynik potegowania to:";
23. if (znak==1) potega=(1/potega);
24. cout<<potega <<endl;


28. system ("pause");
29. }

**Listing 2.5**. Rozwiązanie 3 Zadania 2.2, źródło: opracowanie własne

2.2 Zadania

1. Dany jest ciąg liczb całkowitych dodatnich zakończony zerem. Napisz program, który sprawdzi czy podany ciąg liczb jest uporządkowany, nierosnąco lub niemalejąco. Wstaw punkt przerwania na początku ciała pętli, śledź wszystkie zmienne, które są zmieniane wewnątrz pętli.
2. Dany jest ciąg n-liczb całkowitych. Napisz program, który obliczy i wypisze ilość podciągów składających się z samych liczb dodatnich. Wstaw punkt przerwania na końcu ciała pętli, śledź wszystkie zmienne, które są zmieniane wewnątrz pętli.
3. Dany jest ciąg n-liczb całkowitych. Napisz program, który wypisze te liczby z ciągu, których ostatnia cyfra jest równa cyfrze podanej z klawiatury. Wstaw punkty przerwania po wprowadzeniu danych, wewnątrz ciała pętli, śledź istotne dla programu zmienne.
4. Dany jest ciąg liczb całkowitych zakończony zerem. Napisz program, który wypisze największą liczbę tego ciągu i jej numer w ciągu. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
5. Dany jest ciąg n-liczb całkowitych podanych z klawiatury. Napisz program, który obliczy ilość liczb tego ciągu, które są dzielnikami liczby podanej z klawiatury. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
6. Dany jest ciąg n-elementowy liczb całkowitych. Opracuj algorytm, narysuj schemat blokowy programu wypisującego najmniejszą różnicę pomiędzy dwoma kolejnymi liczbami w ciągu. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
7. Dany jest ciąg n-elementowy liczb całkowitych. Opracuj algorytm, narysuj schemat blokowy programu wypisującego najmniejszy iloczyn dwóch kolejnych liczb w ciągu. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
8. Dany jest ciąg n-elementowy liczb całkowitych. Napisz program, który wypisze długość najdłuższego podciągu nierosnącego. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
9. Dany jest ciąg n-elementowy liczb całkowitych. Napisz program, który wypisze długość najdłuższego podciągu rosnącego. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
10. Napisz program, który wczyta liczbę od użytkownika i sprawdzi czy jest ona podzielna przez 8, a następnie wypisze stosowny komunikat. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
11. Napisz program, który po wczytaniu 3 liczb: dnia, miesiąca i roku urodzenia umożliwi obliczenie tzw. liczby drogi życia. Jest to liczba, mniejsza od 10, uzyskana z sumowania cyfr z podanej daty. Zastosuj odpowiednie typy danych.  
    np. dla 7.11.1996 7+11+1+9+9+6 = 43 43=4+3= 7
12. Napisz program obliczania iloczynu dwóch liczb całkowitych a i b stosując wyłącznie operatory dodawania i odejmowania. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
13. Napisz program, który wczytuje z klawiatury liczbę punktów z kolokwium w postaci wartości całkowitej z przedziału od 0 do 10, a następnie wyświetla na ekranie napis, będący słowną oceną odpowiadającą podanej liczbie punktów. Przyjmij następujące oceny: mierna (0÷1 pkt.), niedostateczna (2÷3 pkt.), dostateczna(4÷5 pkt.), dobra (6÷7 pkt.), bardzo dobra (8÷9 pkt.), celująca(10 pkt.).
14. Napisz program wczytujący z klawiatury liczbę naturalną i wyświetlający na ekranie komunikat określający, czy wprowadzona liczba jest liczbą doskonałą.   
    Wskazówka: liczba doskonała to liczba naturalna równa sumie swoich podzielników całkowitych właściwych, np. 6, 28, 496 itd. 6 = 1 + 2 + 3.
15. Napisz program wczytujący z klawiatury liczbę naturalną i wyświetlający na ekranie komunikat określający, czy wprowadzona liczba jest liczbą Amstronga.  
    Wskazówka: liczba Amstronga to liczba naturalna równa sumie sześcianów swoich cyfr, np. 153 = 13 + 53 + 33. W programie nie stosuj konwersji liczby na łańcuch znakowy – wszystkie operacje należy wykonać na wartościach liczbowych.
16. Napisz program wczytujący dwie dodatnie liczby całkowite **n** i **m**, i wypisujący w kolejnych wierszach na wyjściu wszystkie dodatnie wielokrotności **n** mniejsze od **m**. Użyj tylko operatorów addytywnych. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
17. Napisz program wczytujący dwie dodatnie liczby całkowite **n** i **m**, i wypisujący na wyjściu ***m*** pierwszych wielokrotności liczby ***n***. Użyj tylko operatorów addytywnych. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
18. Napisz program wczytujący trzy dodatnie liczby całkowite **n**, **m** i **k**, i wypisujący w kolejnych wierszach wszystkie wielokrotności **n** większe od **m** i mniejsze od **k**. Użyj tylko operatorów addytywnych. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
19. Napisz program, który dostaje na wejściu nieujemną liczbę całkowitą **n** i zwraca jako wartość liczbę **2n**. Użyj tylko operatorów addytywnych. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
20. Napisz program, który wczytuje nieujemną liczbę całkowitą **n** i wypisuje na wyjściu liczbę **n!**. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
21. Napisz program, który wczytuje nieujemną liczbę całkowitą **n** i wypisuje na wyjściu sumę kwadratów liczb od **0** do **n**, czyli wartość 02 + 12 + 32 + ... + n2. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
22. Napisz program, który wczytuje liczbę całkowitą **n** (n > 2) i wypisuje na wyjściu iloczyn liczb parzystych z zakresu od **2** do **n** (czyli 2 ∗ 4 ∗ . . . ∗ n , dla **n** parzystych i   
    2 ∗ 4 ∗ . . . ∗ (n − 1) w przeciwnym wypadku). Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
23. Napisz program, który wczytuje dwie liczby całkowite **n** i **m** (zakładamy, że **n < m**) i wypisuje na standardowym wyjściu liczbę **n** ∗ . . . ∗ **m**. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
24. Napisz program, który wczytuje nieujemną liczbę całkowitą **n** i wypisuje na wyjściu element ciągu Fibonacciego o indeksie **n**. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
25. Napisz program, który wczytuje liczby **a, b, c, d** i wypisuje najmniejszą nieujemną liczbę całkowitą **x** taką, że |a| ∗ x2 + b ∗ x + c > d. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
26. Napisz program, który wczytuje ze standardowego wejścia liczby **a, b, c, d** i wypisuje na wyjściu największą nieujemną liczbę całkowitą **x** taką, że 5∗x2+a∗x+b < c.   
    Zakładamy, że taka nieujemna całkowita liczba **x** istnieje. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
27. Napisz program, który wczytuje nieujemną liczbę całkowitą **n** i wypisuje na wyjściu wartość 0! + 1! + . . . + n!. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
28. Dla danych liczb n i k policzy symbol Newtona dla Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
29. Napisz program, który wypisuje kolejne wyrazy wolne dla dwumianu Newtona. Dwumian Newtona zapisujemy postaci:



1. Napisz program, który dla wyrażenia wypisze trójkąt Pascala. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
2. Dane są dwa wielomiany 2-go stopnia. Napisz program, który obliczy i wypisze współczynniki wielomianu otrzymanego w wyniku pomnożenia tych dwu wielomianów. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
3. Dane są t z przedziału <0,1> oraz n. Oblicz wartość wielomianu Bernsteina według wzoru: . Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.



1. Dana jest liczba całkowita. Napisz program, który sprawdzi, czy suma cyfr tej liczby jest większa niż iloczyn cyfr. Sprawdź działanie programu dla liczb dodatnich i ujemnych. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
2. Rozważamy liczby naturalne z zakresu 11…n, n>=11. Wypisz te liczby, dla których suma cyfr jest równa iloczynowi cyfr lub -1, gdy takich liczb nie ma. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
3. Dana jest liczba całkowita dodatnia. Napisz program, który podaje postać tej liczby układzie dwójkowym. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
4. Dana jest liczba całkowita dodatnia oraz podstawa innego układu pozycyjnego, <10.   
   Napisz program, który poda postać tej liczby w tym układzie. Co z sytuacją, gdy podstawa jest >10? Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
5. Dany jest n-elementowy ciąg liczb całkowitych. Napisz program, który wypisze liczbę, mającą największą sumę cyfr. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
6. Dany jest ciąg n-elementowy liczb całkowitych. Napisz program, który obliczy największą liczbę mającą nieparzystą liczbę cyfr. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
7. Dany jest ciąg n liczb. Napisz program, który sprawdzi czy w podanym ciągu istnieją liczby palindromiczne. Program powinien wypisywać znalezione liczby lub komunikat o braku takich liczb w podanym ciągu.  
   Uwaga ! Liczbę naturalną, którą czyta się tak samo od początku i od końca nazywamy palindromem. Przykłady liczb palindromicznych: 55, 494, 30703, 414, 5115...
8. Dany jest ciąg n liczb. Opracuj algorytm, narysuj schemat blokowy oraz napisz program, który sprawdzi czy w podanym ciągu istnieją liczby trójkątne. Program powinien wypisywać znalezione liczby lub komunikat o braku takich liczb w podanym ciągu.  
   Uwaga ! Liczby trójkątne to są liczby postaci tk=k(k+1)/2, gdzie k jest liczbą naturalną. Liczba tk jest sumą k kolejnych liczb naturalnych. Np. t1=1, t2=3, t3=6, t4=10
9. Dane są dwie liczby. Napisz program, który sprawdzi czy podane liczby są to liczby zaprzyjaźnione. Program powinien wypisywać odpowiedni komunikat.  
   Uwaga! Dwie liczby naturalne nazywamy zaprzyjaźnionymi, gdy każda z nich jest równa sumie dzielników właściwych drugiej liczby (dzielnik właściwy liczby to każdy dzielnik mniejszy od tej liczby).   
   Przykładem pary liczb zaprzyjaźnionych są liczby 220 i 284. Dzielniki właściwe liczby 220 to:{1,2,4,5,10,11,20,22,44,55,110} więc 1+2+4+5+10+11+20+22+44+55+110=284  
   Dzielniki właściwe liczby 284 to:{1,2,4,71,142} więc 1+2+4+71+142=220
10. Dany jest ciąg liczb zakończony zerem. Napisz program, który w podanym ciągu wyszuka liczby podzielne przez 4. Program powinien wypisać znalezione liczby oraz ich ilość lub komunikat o braku liczb podzielnych przez 4.
11. Napisz program, który wczyta liczbę naturalną n a następnie wyszukuje podzielniki tej liczby i oblicza ich sumę (załóżmy: mniejszych od n). Np.: po wprowadzeniu liczby 12 na ekranie powinna zostać wypisana liczba 16, ponieważ podzielnikami liczby 12 są liczby 1, 2, 3, 4 oraz 6, zaś 1 + 2 + 3 + 4 + 6 = 16. Napisz program, który wczyta liczbę od użytkownika i sprawdzi czy jest ona podzielna przez 8, a następnie wypisze stosowny komunikat. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
12. Dane są dwie liczby całkowite. Napisz program, który sprawdzi czy są one liczbami lustrzanymi  np.: 125 i 521, 68 i 86. Program powinien wypisać odpowiedni komunikat. Wstaw punkty przerwania w miejscach wybranych przez siebie, śledź wybrane zmienne.
13. Dane są trzy liczby całkowite a, b, c. Napisz program, który wypisze 1 (jeden) jeśli liczby te są trójką pitagorejską oraz 0 (zero) w przeciwnym wypadku.   
    Liczby pitagorejskie spełniają warunek: a\*a+b\*b=c\*c.

1. Nie ma jednego, najlepszego sposobu, jak wymyślać rozwiązania problemów, których jeszcze nie rozwiązano lub rozwiązano, ale my nie znamy tego rozwiązania. Do tego min. służą studia, by poznawać różne schematy postępowania, zdobywać potrzebną wiedzę i umiejętności. Na pewno umiejętność wymyślania rozwiązania problemu jest w pewnym sensie sztuką, ale także można wypracować w sobie pewne umiejętności –same raczej się nie pojawią. [↑](#footnote-ref-1)
2. Zapis: iloczyn=iloczyn+y; nie jest równaniem w matematyce, tzn. nie szukamy takich liczb iloczyn i y, by lewa strona=prawa strona. Jest to zwięzły zapis dwu czynności:(1) najpierw oblicz wyrażenie po prawej stronie, (2) wynik obliczonego wyrażenia jest nową wartością zmiennej iloczyn. Średnik kończący linię oznacza koniec instrukcji. [↑](#footnote-ref-2)
3. Wczytanie danych to nadanie wartości początkowych nie wymagające modyfikacji tekstu programu. [↑](#footnote-ref-3)
4. Sprawdzenie, czy wartość zmiennej x (=3) jest większa od 0. [↑](#footnote-ref-4)
5. Najpierw obliczana jest prawa stronę wyrażenia – do wartości zmiennej iloczyn (=0) dodana zostaje wartość zmiennej y (=4), obliczany jest wynik (=4) i ta wartość przypisana zostaje do zmiennej iloczyn. Oznacza to, że w każdym późniejszym odwołaniu do wartości zmiennej iloczyn będzie brane 4 a nie 0 (poprzednia wartość). [↑](#footnote-ref-5)