Zespół Szkół nr 5 Mistrzostwa Sportowego XV Liceum Ogólnokształcące w Bydgoszczy

Wstęp do algorytmiki i programowania w języku C++ z wykorzystaniem kompilatora Dev C++

Opracowanie: mgr Marcin Maćkiewicz

Opracowanie przygotowane dla uczniów klas z rozszerzoną informatyką o profilach informatycznym i politechnicznym XV Liceum Ogólnokształcącego w Bydgoszczy

Spis treści

1. Definicja algorytmu

Słowo „algorytm” pochodzi od nazwiska Algorismi (Al.-Chorezmi), wybitnego arabskiego matematyka i astronoma, żyjącego na przełomie VIII i IX wieku. Algorytm kojarzy Ci się zapewne z informatyką, ewentualnie z matematyką. Czy zdajesz sobie jednak sprawę, że z algorytmami masz kontakt na co dzień? Wyjaśnijmy więc najpierw co to jest algorytm. Mówiąc ogólnie:

Algorytm to skończony ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego rodzaju zadań.

Gdzie możesz spotkać algorytmy? Praktycznie wszędzie!

Kupując nowe urządzenie otrzymujesz instrukcję obsługi, w której krok po kroku omówione są kolejne operacje, jak np. ustawienie zegara.

Piekąc placek sięgasz po przepis, który jest niczym innym jak algorytmem opisującym potrzebne składniki, ich ilości, kolejność i sposób ich łączenia oraz czas pieczenia. Im dokładniejszy mamy przepis, tym większa jest szansa na udane ciasto.

Wstając rano z łóżka również zaczynasz działać według pewnego algorytmu, np. wstajesz, idziesz do łazienki, myjesz się, idziesz do kuchni, jesz śniadanie, wychodzisz do szkoły, uczestniczysz w lekcjach zgodnie z obowiązującym planem lekcji, wracasz do domu, itd.

Często nie zdajesz sobie z tego sprawy, ale nasze codzienne czynności są niczym innym, jak wyuczonymi algorytmami. Również w przyszłej pracy to właśnie algorytmy będą dla Ciebie podstawą do działania. Algorytmami posługuje się murarz, malarz, stolarz, ślusarz czy górnik. Gdy algorytm nie jest poprawny lub pracownik nie wykona go dokładnie może dojść do wypadku przy pracy, np. zawalenia się szybu kopalni z powody złego, tzn. niezgodnego z algorytmem montażu stempla podpierającego strop chodnika.

Jak widzisz, to właśnie od algorytmu zależy bardzo często powodzenie naszego przedsięwzięcia. Poza czynnościami powinniśmy więc również opisać np. narzędzia, które musimy wykorzystać czy potrzebne składniki. Podając przepis na placek nie wystarczy zapisać informacji, że potrzebny jest kilogram mąki, ale trzeba również określić jej rodzaj, np. kilogram mąki pszennej tortowej. Gdy zabraknie tej informacji użytkownik może zamiast mąki pszennej użyć mąki żytniej czy kukurydzianej, a to może doprowadzić do nieudanego wypieku. Biorąc powyższe pod uwagę możemy sprecyzować definicję algorytmu:

Algorytmem nazywamy skończony ciąg czynności przekształcający zbiór danych wejściowych na zbiór danych wyjściowych (wyników).

2. Etapy konstruowania algorytmu

Zabierzmy się teraz za skonstruowanie naszego pierwszego algorytmu. W poprawnej budowie algorytmu z pewnością pomocne będą poniższe punkty, przedstawiające etapy konstruowania algorytmu:

1. **Sformułowanie zadania – ustalamy, jaki problem ma rozwiązać algorytm.**
2. **Określenie danych wejściowych oraz ich typu.**
3. **Określenie wyniku oraz sposobu jego prezentacji.**
4. **Ustalenie najlepszej z możliwych metody wykonania zadania.**
5. **Zapisanie algorytmu za pomocą wybranej metody.**
6. **Analiza poprawności rozwiązania.**
7. **Testowanie rozwiązania dla różnych danych.**
8. **Ocena skuteczności algorytmu.**

PRZYKŁAD 1:

Załóżmy, że chcemy skonstruować przepis (algorytm) na najlepszą na świecie jajecznicę. Sugerując się więc powyższymi punktami zabieramy się do pracy:

1. Przepis na najlepszą jajecznicę na świecie.
2. Potrzebne nam będą:
   - 4 jajka kurze średniej wielkości
   - 20 dag boczku
   - 2 łyżki oleju
   - 2 łyżki mleka
   - sól i pieprz według upodobań
3. Jajecznicę dobrze ściętą podajemy na talerzu śniadaniowym z pieczywem
4. Mleko i jajka wlewamy do miseczki i dokładnie mieszamy. Wlewamy na patelnię olej i czekamy, aż się zagrzeje, następnie lekko podsmażamy na nim boczek. Wlewamy na patelnię jajka, dodajemy pieprz i sól i smażymy na małym ogniu ok. 5-6 min.
5. Nasz algorytm został już zapisany metodą słowną, przechodzimy więc do kolejnego etapu.
6. Ponownie czytamy nasz przepis, sprawdzając, czy nie zapomnieliśmy o żadnym składniku, następnie przechodzimy do smażenia jajecznicy.
7. Po usmażeniu jajecznicy smakujemy ją i zastanawiamy się, czy wsypaliśmy odpowiednią ilość soli i pieprzu. Jeśli tak, przekładamy ją na talerz śniadaniowy, jak opisane zostało w punkcie 3 powyższego algorytmu.
8. Zjadamy jajecznicę analizując jej smak. Na tym etapie podejmujemy decyzję o pozostawieniu przepisu lub jego zmodyfikowaniu. i przebudowaniu naszego algorytmu.

Zanim przejdziemy do konstruowania bardziej złożonych algorytmów musimy omówić jeszcze takie zagadnienia, jak sposoby zapisywania algorytmów i specyfikacja problemu algorytmicznego.

3. Sposoby zapisywania algorytmów

Wyróżniamy 5 sposobów zapisywania algorytmów, które przedstawimy na przykładzie algorytmu obliczania wartości bezwzględnej podanej liczby.

Wartość bezwzględna liczby rzeczywistej
to wartość liczbowa, która nie uwzględnia znaku liczby.
Przykładowo liczba 5 jest wartością bezwzględną liczby 5 i -5.

1. **Opis słowny** algorytmu to zazwyczaj pierwszy i podstawowy opis algorytmu. Jest on mało ścisłą reprezentacją algorytmu. Służy głównie do ukierunkowania rozwiązania oraz wskazania technik przydatnych w rozwiązaniu.

   PRZYKŁAD:

   Prosimy użytkownika o podanie liczby. Sprawdzamy, czy jest ona mniejsza od zera, a jeśli tak, to zmieniamy jej znak na przeciwny. Wyświetlamy liczbę, która jest wartością bezwzględną podanej przez użytkownika liczby.

2. **Lista kroków** nazywana jest często metodą „**krok po kroku**”. To sposób dokładnie opisujący wymagane operacje oraz ich kolejność. Każdy krok opisuje realizację kolejnej czynności i jest zawarty w osobnym wierszu. Elementem charakterystycznym dla tej metody jest numerowanie wierszy.

   PRZYKŁAD:

   Krok 1: Podaj liczbę *x* której wartość bezwzględną chcesz obliczyć.
   Krok 2: Jeśli *x < 0* wypisz (*-x*).
   Krok 3: Jeśli *x >= 0* wypisz *x*.
   Krok 4: Zakończ działanie algorytmu.

3. **Pseudojęzyk** to metoda pośrednia między listą kroków a zapisem w języku programowania. Notacja algorytmów za pomocą pseudojęzyka jest zbliżona do zapisu w języku programowania ale mniej formalna.

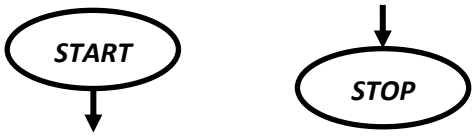
   PRZYKŁAD:

   Początek;
   Rzeczywiste x;
   Jeśli x >= 0 Wypisz (x);
   Jeśli x < 0 Wypisz (-x);
   Koniec.

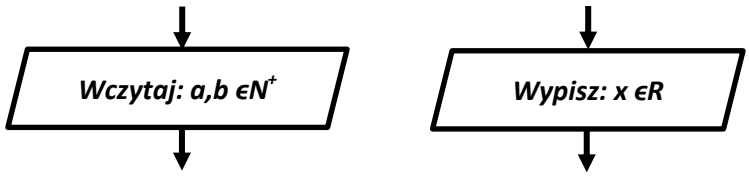
4. **Schemat blokowy** to graficzny sposób przedstawienia algorytmu. Podaje szczegółowo wszystkie operacje arytmetyczne, logiczne, sterujące, przesyłania i pomocnicze wraz z kolejnością ich wykonywania. Jest on podstawą do napisania programu, znacznie ułatwiającą późniejszy zapis w określonym języku programowania.

   Zanim przejdziemy do przedstawienia przykładu musimy omówić bloki stosowane w schematach blokowych. Wyróżniamy następujące bloki:

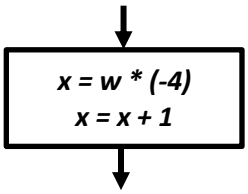
   a) **początku i końca** – oznacza początek, koniec, przerwanie lub wstrzymanie wykonywania działania algorytmu;



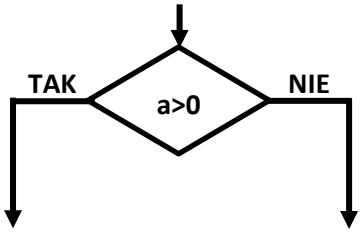
   b) **wejścia i wyjścia** – przedstawia czynność wprowadzania danych do programu i przyporządkowania ich zmiennym dla późniejszego wykorzystania, jak i wyprowadzenia wyników obliczeń;



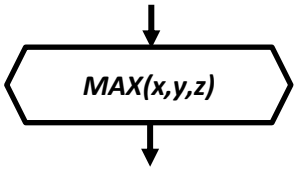
   c) **obliczeniowy** – oznacza wykonanie operacji, w efekcie której zmienią się wartości, postać lub miejsce zapisu danych;



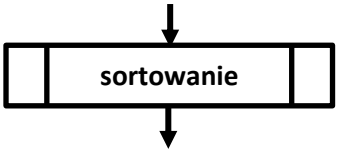
   d) **decyzyjny** – przedstawia wybór jednego z dwóch wariantów wykonywania programu na podstawie sprawdzenia warunku wpisanego w dany blok;



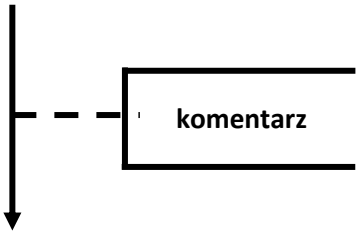
   e) **wywołania podprogramu** – oznacza zmianę wykonywanej czynności na skutek wywołania podprogramu;



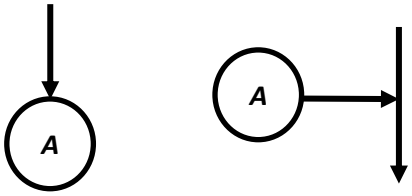
f) **fragmentu** – przedstawia część programu zdefiniowanego odrębnie;



g) **komentarza** – pozwala wprowadzać komentarze wyjaśniające poszczególne części schematu, co ułatwia zrozumienie go czytającemu;

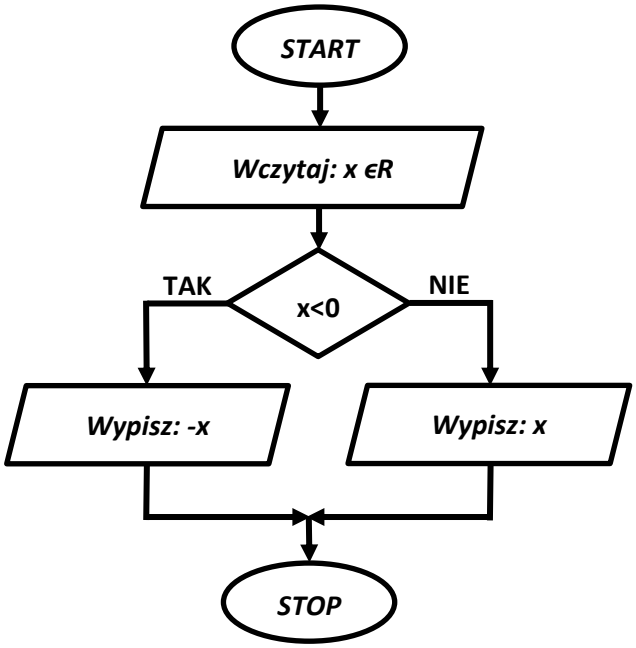


h) **łącznik wewnętrzny** – służy do łączenia odrębnych części schematu znajdujących się na tej samej stronie; powiązane ze sobą łączniki oznaczone są tym samym napisem;



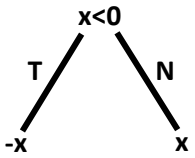
i) **łącznik zewnętrzny** – służy do łączenia odrębnych części schematu znajdujących się na odrębnych stronach; powinien być opisany jak łącznik wewnętrzny, poza tym powinien zawierać numer strony, do której się odwołuje;

PRZYKŁAD:



5. **Drzewo algorytmiczne** to specyficzny zapis algorytmu z wykorzystaniem grafów, odmienny jednak od schematu blokowego. Nie będziemy korzystali z takiego sposobu zapisywania algorytmów, dlatego pomijamy jego dokładny opis. Przedstawimy jednak drzewo naszego przykładowego algorytmu.

PRZYKŁAD:



4. Specyfikacja problemu algorytmicznego

Specyfikacja problemu algorytmicznego to pierwszy i najważniejszy krok do stworzenia poprawnego i skutecznego algorytmu.

Specyfikacja problemu algorytmicznego to dokładny opis problemu algorytmicznego, który ma zostać rozwiązany, oraz określenie danych wejściowych i danych wyjściowych wraz z ich typami.

Łatwo można zauważyć, że w specyfikacji problemu algorytmicznego zawarte są pierwsze cztery punkty etapów konstruowania algorytmu opisane w rozdziale 1.1.

Zanim jednak przejdziemy do przedstawienia przykładu specyfikacji musimy wprowadzić jeszcze pojęcie zmiennej:

Zmienną nazywamy obiekt występujący w algorytmie, określony przez nazwę i służący do zapamiętywania pewnych danych. np. x = 5, liczba = 17, wyraz = „algorytmika”, itp.

Przedstawmy teraz specyfikację problemu algorytmicznego algorytmu zaliczanego do grupy algorytmów liniowych.

Algorytmy, w których kolejność wykonywanych czynności jest zawsze taka sama i niezależna od wartości danych wejściowych, nazywamy algorytmami sekwencyjnymi lub inaczej algorytmami liniowymi.

PRZYKŁAD 2:

1. **Problem algorytmiczny:** obliczanie pola powierzchni prostokąta.
2. **Dane wejściowe:**

   $a,b \in N^+$ – długości boków prostokąta.
3. **Dane wyjściowe:**

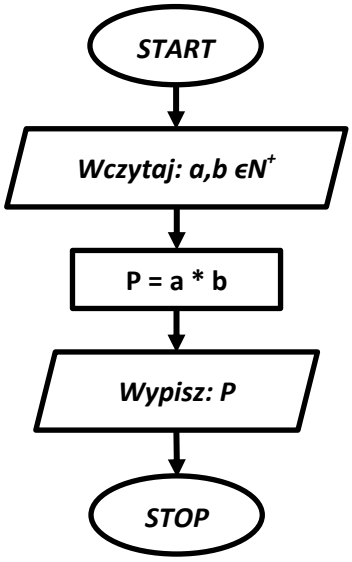
   $P \in N^+$ – *obliczone pole powierzchni prostokąta.*

Jak widać w powyższym przykładzie, w specyfikacji problemu algorytmicznego bardzo dokładnie opisane są nie tylko zadania algorytmu, ale również użyte w nim zmienne, np. *a*, *b* i *P* oraz ich typy, czyli jak w naszym przykładzie informacja, że są to liczby naturalne dodatnie.

Im dokładniejsza będzie specyfikacja problemu algorytmicznego, tym łatwiej będzie zapisać dany algorytm za pomocą jednej z metod zapisywania algorytmów.

To na tym etapie należy przemyśleć, jakie dokładnie kroki ma wykonywać nasz algorytm. Bardzo często okazuje się, że do poprawnego działania algorytmu potrzebne nam będą dodatkowe zmienne czy funkcje. Zagadnienie funkcji omówimy w jednym z dalszych rozdziałów opracowania.

Popatrzmy na schemat blokowy powyższego algorytmu:



Jak widać, bloki powyższego schematu układają się w jednej linii, stąd nazwa „algorytm liniowy". Bez względu na wprowadzone wartości zmiennych *a* i *b* przebieg algorytmu będzie zawsze taki sam.

Poza algorytmami liniowymi (sekwencyjnymi) wyróżniamy również algorytmy rozgałęzione lub inaczej algorytmy z warunkami.

Przykładem takiego algorytmu może być algorytm omawiany w rozdziale 1.2 obliczania wartości bezwzględnej podanej przez użytkownika liczby.

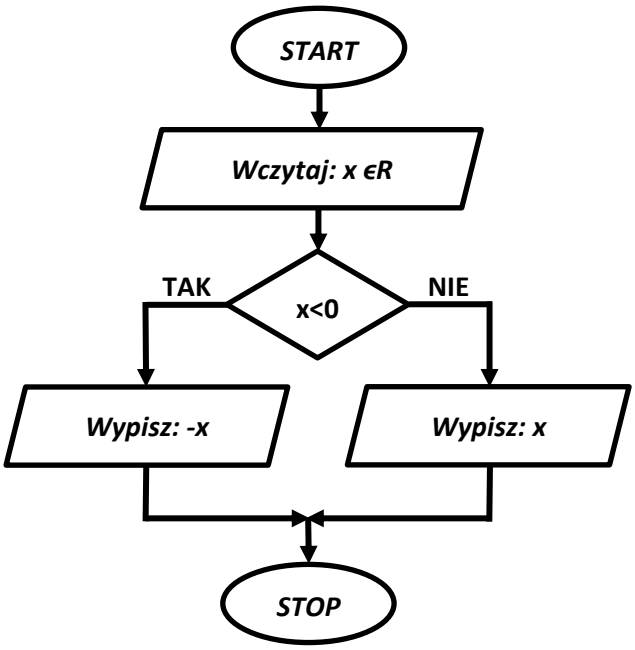
PRZYKŁAD 3:

1. **Problem algorytmiczny:** obliczenie wartości bezwzględnej podanej przez użytkownika liczby.
2. **Dane wejściowe:**

   liczba ϵ *N* – liczba podana przez użytkownika.
3. **Dane wyjściowe:**

   obliczona wartość bezwzględna podanej przez użytkownika liczby.

Popatrzmy na schemat blokowy powyższego algorytmu:



Jak widać po sprawdzeniu warunku x<0 wykonana zostanie prawa lub lewa strona algorytmu. Dalszy przebieg algorytmu uzależniony jest więc od wartości liczby podanej przez użytkownika.

Algorytm nazywamy rozgałęzionym, jeżeli jego realizacja uzależniona jest od wartości danych wejściowych.

Przejdźmy teraz do przedstawienia specyfikacji bardziej złożonego problemu algorytmicznego.

PRZYKŁAD 4:

1. **Problem algorytmiczny:** zliczenie ilości liczb parzystych i liczb nieparzystych spośród 1000 liczb podanych przez użytkownika.
2. **Dane wejściowe:**

   liczba ∈ *N* – liczba podana przez użytkownika.
3. **Dane wyjściowe:**

   p ∈ N – zliczona ilość podanych liczb parzystych.

   n ∈ N – zliczona ilość podanych liczb nieparzystych.
4. **Zmienne pomocnicze:**

   i ∈ *N* – zmienna zliczająca ilość podanych przez użytkownika liczb.

Jak widać w powyższym przykładzie w naszej specyfikacji problemu algorytmicznego pojawiła się dodatkowo zmienna pomocnicza, której zadaniem jest kontrolowanie ilości wprowadzanych przez użytkownika liczb.

Takie zastosowanie zmiennej jednoznacznie określa, iż jest to algorytm iteracyjny, czyli taki, w którym znajduje się pętla.

Iteracja to czynność powtarzania tej samej instrukcji lub wielu instrukcji w pętli. Mianem iteracji określa się także operacje wykonywane wewnątrz takiej pętli.

Załóżmy, że mamy narysować na piasku kwadrat. Możemy więc wykonać poniższe czynności:

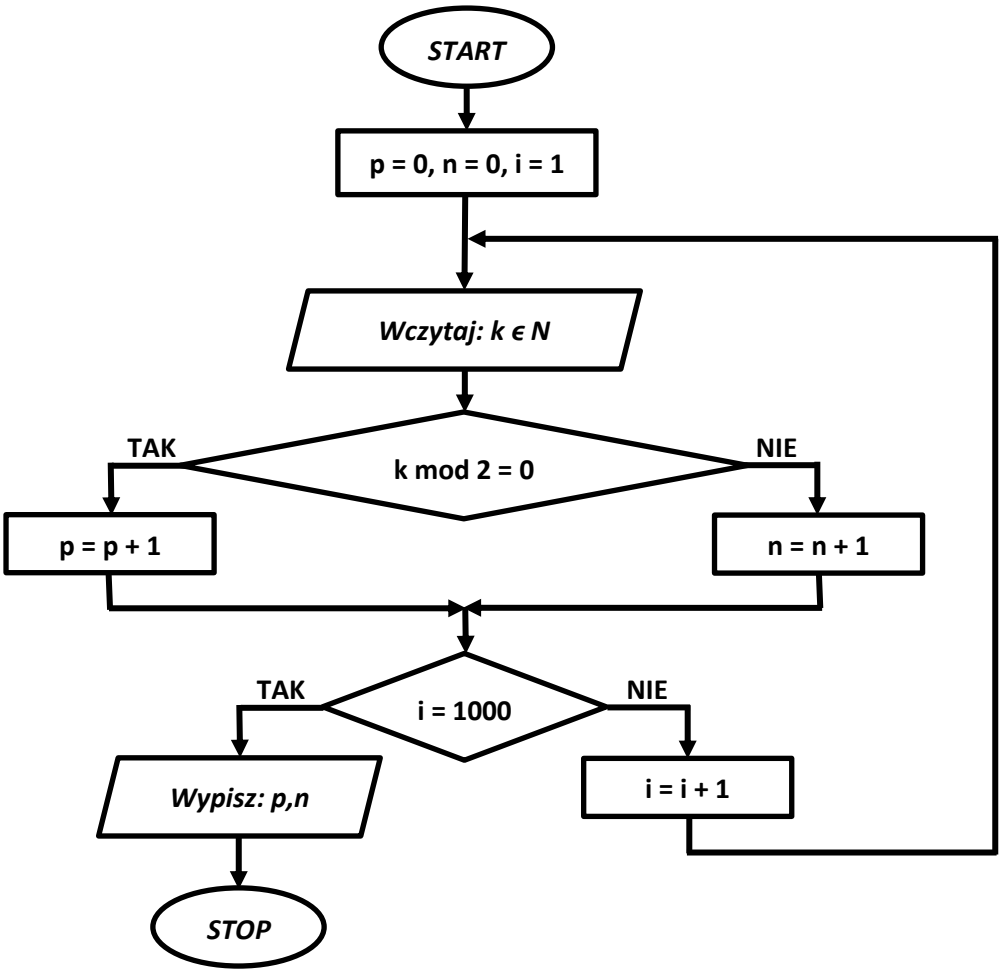
- narysuj linię długości 4 metrów i obróć się w prawo o 90 stopni,
- narysuj linię długości 4 metrów i obróć się w prawo o 90 stopni,
- narysuj linię długości 4 metrów i obróć się w prawo o 90 stopni,
- narysuj linię długości 4 metrów i obróć się w prawo o 90 stopni,

Można powyższe czynności zapisać jednak znacznie krócej, np.

- narysuj linię długości 4 metrów i obróć się w prawo o 90 stopni,
- powyższą czynność powtórz jeszcze 3 razy.

Efektem realizacji obu algorytmów będzie narysowany kwadrat, ale zapis drugiego algorytmu jest znacznie krótszy od zapisu pierwszego algorytmu.

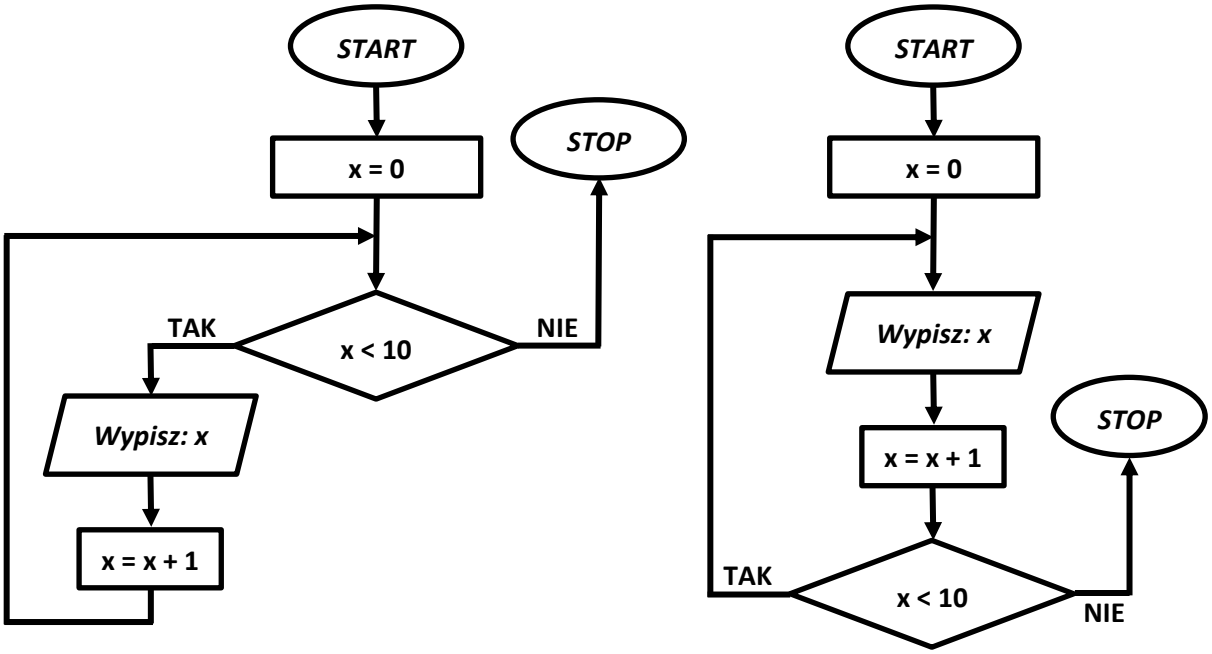
Popatrzmy teraz na schemat blokowy algorytmu z przykładu 4, pokazujący zastosowanie pętli w algorytmice:



5. Rodzaje pętli

Pętle możemy podzielić na dwa rodzaje. Ważne jest posiadanie umiejętności rozróżniania ich w opisach algorytmów.

Popatrz na poniższe schematy blokowe algorytmu wypisującego wszystkie liczby naturalne jednocyfrowe:



Łatwo można zauważyć, że na schemacie blokowym po lewej stronie najpierw sprawdzany jest warunek, a dopiero później wykonywane są polecenia znajdujące się wewnątrz pętli. Na schemacie po prawej stronie sytuacja jest odwrotna, tzn. najpierw wykonywane są polecenia znajdujące się wewnątrz pętli, a dopiero później sprawdzany jest warunek i gdy jest on spełniony pętla wykonuje się ponownie. Efekt działania obu algorytmów jest jednak identyczny. Kiedy więc stosuje się którą z powyższych pętli? Pętla po prawej stronie daje nam pewność, że zostanie uruchomiona przynajmniej jeden raz, podczas gdy pętla po lewej stronie może nie wykonać się ani razu jeśli warunek nie będzie spełniony.

6. Przykłady i ćwiczenia

Przykład 1

Określ specyfikację i zapisz za pomocą listy kroków, pseudojęzyka i schematu blokowego algorytm sprawdzający warunek istnienia trójkąta.

W każdym trójkącie suma długości dwóch dowolnych boków jest większa od długości trzeciego boku.

ROZWIĄZANIE:

Specyfikacja problemu algorytmicznego:

1. **Problem algorytmiczny**: sprawdzić, czy z podanych przez użytkownika trzech odcinków można zbudować trójkąt.
2. **Dane wejściowe**:

   a, b, $c \in N$ – długości odcinków, z których ma zostać zbudowany trójkąt
3. **Dane wyjściowe**:

   komunikat: „Można zbudować trójkąt" lub „Nie można zbudować trójkąta".

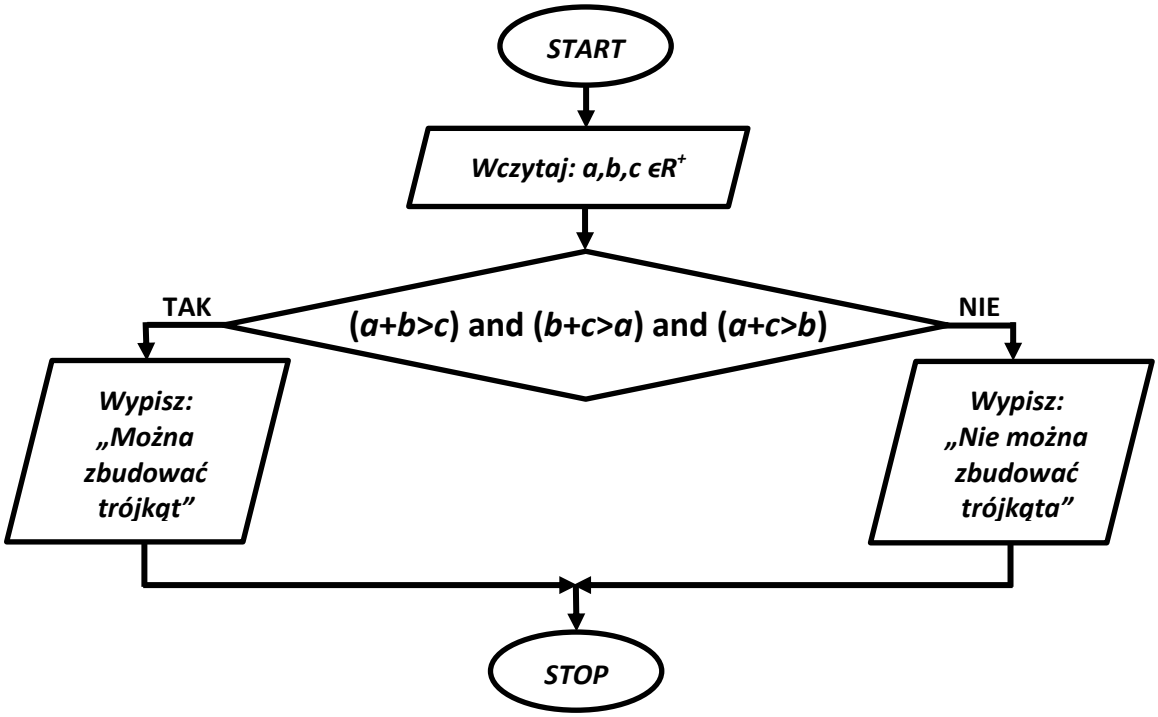
Algorytm zapisany za pomocą **listy kroków**:

Krok 1: Podaj długości odcinków a, b i c, z których ma zostać zbudowany trójkąt.
Krok 2: Jeśli $a+b>c$ i $b+c>a$ i $a+c>b$ wypisz („*Można zbudować trójkąt*"), w przeciwnym wypadku wypisz („*Nie można zbudować trójkąta*").
Krok 3: Zakończ działanie algorytmu.

Algorytm zapisany za pomocą **pseudojęzyka**:

Początek;
Rzeczywiste dodatnie a, b, c;
Jeśli ($a+b>c$) i ($b+c>a$) i ($a+c>b$) Wypisz („*Można zbudować trójkąt*")
w przeciwnym wypadku Wypisz („*Nie można zbudować trójkąta*");
Koniec.

Algorytm zapisany za pomocą **schematu blokowego**:



Przykład 2

Określ specyfikację i zapisz za pomocą listy kroków, pseudojęzyka i schematu blokowego algorytm obliczający silnię podanej przez użytkownika liczby.

Silnią liczby naturalnej *n* (*n*!, co czytamy „n silnia”) nazywamy iloczyn wszystkich liczb naturalnych nie większych niż *n*, np.

3! = 1*2*3 = 6 , 2! = 1*2 = 2 , 1! = 1 , 0! = 1

ROZWIĄZANIE:

Specyfikacja problemu algorytmicznego:

1. **Problem algorytmiczny**: obliczyć silnię liczby podanej przez użytkownika.
2. **Dane wejściowe**:

   $a \in N$ – liczba podana przez użytkownika.
3. **Dane wyjściowe**:

   $s \in N$ – obliczona silnia podanej przez użytkownika liczby a.
4. **Zmienne pomocnicze**:

   $x \in N$ – zmienna zliczająca ilość wykonanych pętli.

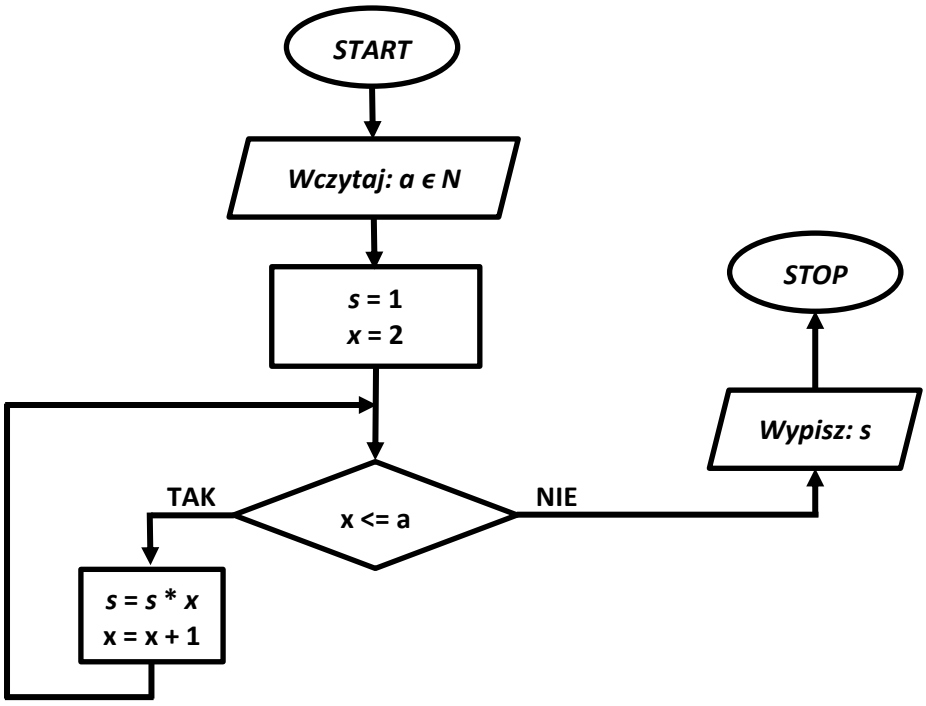
Algorytm zapisany za pomocą **listy kroków**:

Krok 1: Podaj liczbę *a*, której silnię chcesz obliczyć.
Krok 2: Zmiennej *s* przypisz wartość 1.
Krok 3: Zmiennej *x* przypisz wartość 2.
Krok 4: Jeśli x > a przejdź do kroku 8.
Krok 5: Zmiennej *s* przypisz wynik iloczynu *s* * *x*.
Krok 6: Zwiększ o 1 wartość zmiennej *x*.
Krok 7: Wróć do kroku 4.
Krok 8: Wypisz wartość zmiennej *s*.
Krok 9: Zakończ działanie algorytmu.

Algorytm zapisany za pomocą **pseudojęzyka**:

```
Początek;
Naturalne a;
s ← 1;
x ← 2;
Dopóki x<=a wykonuj:
    s ← s * x;
    x ← x + 1;
Wypisz s;
Koniec.
```

Algorytm zapisany za pomocą **schematu blokowego**:



Ćwiczenie 1. Określ specyfikację i zapisz za pomocą listy kroków, pseudojęzyka i schematu blokowego algorytm obliczający obwód trójkąta.

Ćwiczenie 2. Określ specyfikację i zapisz za pomocą listy kroków, pseudojęzyka i schematu blokowego algorytm sprawdzający czy podana przez użytkownika liczba jest liczbą parzystą.

Ćwiczenie 3. Określ specyfikację i zapisz za pomocą listy kroków, pseudojęzyka i schematu blokowego algorytm wypisujący wszystkie liczby dwucyfrowe podzielne przez 7.

Ćwiczenie 4. Określ specyfikację i zapisz za pomocą listy kroków, pseudojęzyka i schematu blokowego algorytm obliczający dowolną potęgę naturalną dowolnej liczby naturalnej.

Ćwiczenie 5. Określ specyfikację i zapisz za pomocą listy kroków, pseudojęzyka i schematu blokowego algorytm obliczający średnią arytmetyczną 10 liczb podanych przez użytkownika.

Ćwiczenie 6. Określ specyfikację i zapisz za pomocą listy kroków, pseudojęzyka i schematu blokowego algorytm wypisujący gwiazdki w ilości podanej przez użytkownika.

Ćwiczenie 7. Określ specyfikację i zapisz za pomocą listy kroków, pseudojęzyka i schematu blokowego algorytm wypisujący wszystkie dzielniki podanej przez użytkownika liczby.

7. Budowa programu komputerowego

Struktura podstawowego programu napisanego w języku C++ jest dość prosta i składa się z 4 części:

`#include <cstdlib>` `#include <iostream>`	część 1
`using namespace std;`	część 2
`int main()`	część 3

`{`
`  instrukcje funkcji main`
`  return 0;`
`}`
	część 4

Część 1 – biblioteki

W tej części programu podajemy nazwy bibliotek, które mają być dołączone do programu. Biblioteki to pliki o nazwach podanych w nawiasach **<>**, w których zapisane są m. in. przydatne funkcje. Dołączamy je do programu za pomocą dyrektywy preprocesora **#include**. Dzięki bibliotekom programista nie musi pisać całego programu od podstaw, lecz może wykorzystać zawarte w bibliotekach funkcje, np. zamiast pisać program obliczający wartość bezwzględną liczby wystarczy wykorzystać funkcję **fabs()** zawartą w bibliotece **cmath**.

W powyższym przykładzie dołączamy dwie biblioteki:

cstdlib – zawiera funkcje ogólne, takie jak konwersje, alokacje pamięci czy niektóre funkcje matematyczne,

iostream – (input/output stream) jest standardową biblioteką wejścia/wyjścia w języku C++. Jeśli chcemy coś wyświetlać na ekranie za pomocą obiektu **cout** i operatora "**<<**" lub pobrać dane z klawiatury za pomocą obiektu **cin** i operatora "**>>**" musimy dodać ją do nagłówka programu.

W dalszej części opracowania korzystać będziemy również z innych funkcji, które omówimy zawsze przed ich pierwszym użyciem.

Część 2 – przestrzenie nazw

Dyrektywę **using namespace** używamy, żeby nie pisać wywołania obiektu **cout** czy **cin** z przedrostkiem **std::** Dzięki omawianej powyżej dyrektywie zamiast instrukcji: **std::cout<<"Ala ma kota"<<std::endl;** możemy napisać: **cout<<"Ala ma kota"<< endl;**

Instrukcje języka C++ omówimy dokładniej w kolejnych rozdziałach opracowania więc ograniczymy się teraz wyłącznie do podania powyższego przykładu.

Część 3 – podstawowa funkcja programu

Funkcja **int main()** jest charakterystyczną funkcją języka C++, która musi występować w każdym konsolowym programie. To właśnie od tej funkcji każdy program zaczyna się uruchamiać. Instrukcje funkcji **int main()** zawarte są między dwoma nawiasami klamrowymi **{ … }** i nazywamy je „**ciałem**" funkcji – patrz część 4.

Część 4 – ciało funkcji

Jest to blok funkcji rozpoczynający się nawiasem **{** i kończący nawiasem **}**. Wewnątrz tego bloku znajdują się wszystkie instrukcje naszego programu. Z wnętrza funkcji **int main()** możemy również wywoływać inne funkcje, co omówimy w dalszej części opracowania. W ciele funkcji **int main()** musi pojawić się informacja zwrotna **return 0** (lub **return EXIT_SUCCESS**), która zwraca do systemu operacyjnego informacje o zakończeniu działania danej aplikacji.

8. Zintegrowane środowisko programistyczne IDE

Każdy początkujący programista staje przed wyborem środowiska, w którym będzie tworzył swoje aplikacje. Nasz wybór padł na polskojęzyczny program Dev-C++ w wersji 5.5.3. Dev-C++ jest darmowym i niezwykle popularnym wśród użytkowników systemu Windows zintegrowanym środowiskiem programistycznym IDE, obsługującym język C i C++. Z jednej strony stanowi ono doskonałe narzędzie pracy dla osób stawiających swoje pierwsze kroki w świecie programowania, z drugiej natomiast dobrze sprawdza się również podczas tworzenia rozbudowanych projektów. Dev-C++ wyposażony został we wszystkie niezbędne programiście narzędzia i funkcje, pozwalające usprawnić wykonywaną przez niego pracę. Środowisko, pomimo dużej liczby opcji jest proste w obsłudze, na co w dużym stopniu wpływa przejrzysty i intuicyjny interfejs graficzny. Umożliwia tworzenie zarówno niewielkich programików konsolowych, jak również aplikacji z interfejsem graficznym, statycznych i dynamicznych bibliotek oraz programów wykorzystujących możliwości bibliotek graficznych Direct3D lub OpenGL.

Wśród oferowanych funkcji znalazł się m.in. mechanizm kart, pozwalający pracować z wieloma plikami jednocześnie (także w trybie pełnoekranowym), wbudowany klient systemu kontroli wersji CVS itp. Nie zabrakło również wysoce konfiguralnej opcji kolorowania składni, przeglądarki klas wykorzystywanych w projekcie, a także niezwykle przydatnego w tego typu oprogramowaniu mechanizmu skrótów klawiszowych, pozwalającego na sprawne i szybkie

wykonywanie częstych operacji np. kompilacji kodu. Dev-C++ daje możliwość importu projektów z programu Visual C++.

Środowisko bazuje na kompilatorze GCC, a jego podstawowa funkcjonalność może być przez użytkownika dowolnie rozszerzana za pośrednictwem tzw. DevPacków o obsługę kolejnych bibliotek, szablonów itp.

Powyższy opis programu pochodzi ze strony: http://www.dobreprogramy.pl/DevC,Program,Windows,28400.html, z której można pobrać omawiany program. Zachęcam również do odwiedzenia strony domowej projektu, znajdującej się pod adresem: http://www.bloodshed.net/.

9. Pierwszy program w języku C++

Wiemy już jak zbudowany jest program w języku C++. Przejdźmy więc do napisania pierwszego prostego programu komputerowego, którego zadaniem jest wyświetlenie komunikatu na ekranie monitora.

```
#include <iostream>
using namespace std;
int main()
{
    cout << "To jest nasz pierwszy program komputerowy" << endl;
    cout << "Zycze przyjemnej pracy z jezykiem C++";
    return 0;
}
```

Łatwo można zauważyć, że w komunikatach oznaczonych kolorem niebieskim nie ma znaków polskiego alfabetu. Z powodu innej wersji języka polskiego w systemie Windows i trybie konsolowym zamiast polskich znaków pojawiłyby się dziwne znaczki, więc zaleca się dla celów edukacyjnych pomijanie polskich liter.

Wróćmy jednak do omówienia naszego programu. Pojawiła się w nim pierwsza nowa instrukcja języka C++, choć wspominaliśmy już o niej w rozdziale 7:

cout << "To jest nasz pierwszy program komputerowy" << endl;

Obiekt **cout** jest strumieniem wyjścia konsoli (ang. console output). Pozwala on wypisywać teksty (oznaczone kolorem niebieskim) na ekranie konsoli. Oprócz danych do strumieni można przesyłać również tzw. manipulatory, które ingerują w sposób działania strumienia. Takim manipulatorem jest w naszym programie **endl**, którego zadaniem jest przeniesienie wydruku tekstu na początek nowego wiersza. Chcąc wyeliminować manipulator **endl** możemy powyższą instrukcję zapisać w inaczej: **cout << "To jest nasz pierwszy program komputerowy \n";**

Jak widać w powyższej instrukcji zamiast operatora **endl** użyliśmy **\n** zapisanego wewnątrz cudzysłowu, które spełnia dokładnie takie samo zadanie jak **endl**.

10. Zmienne w języku C++

Biblioteka **<iostream>** pozwala nam nie tylko wypisywać tekst, ale również wczytywać dane do zmiennych. Zmienna (patrz rozdział 4), jak sama nazwa wskazuje będzie się zmieniać w trakcie działania programu. Zmienna to pewien stosunkowo mały obszar w pamięci, w którym możemy przechowywać dane różnego typu np. liczby całkowite, liczby rzeczywiste (zmiennoprzecinkowe), znak, tekst, itp. Nie można jednak wszystkiego zapisywać do jednej zmiennej. Każda zmienna ma swoje przeznaczenie, wielkość i właściwości. Na zmiennych liczbowych możemy wykonywać operacje matematyczne, w innych z kolei możemy przechowywać tekst.

Zmienne możemy podzielić na 4 typy:

1. Zmienne całkowite

Jak sama nazwa mówi, przechowują tylko liczby całkowite. Różnią się one rozmiarem, czyli zakresem przechowywanych liczb. Im większy rozmiar, tym większe liczby mogą być przechowane.

Nazwa	Wielkość (bajty)	Zakres
short	2	$-2^{15} \div 2^{15} - 1$, czyli przedział [-32768, 32767]
int	4	$-2^{31} \div 2^{31} - 1$, czyli przedział [-2147483648, 2147483647]
long	4	$-2^{31} \div 2^{31} - 1$, czyli przedział [-2147483648, 2147483647]
long long	8	$-2^{63} \div 2^{63} - 1$, czyli przedział [-9223372036854775808, 9223372036854775807]
unsigned short	2	$0 \div 2^{16} - 1$, czyli przedział [0, 65535]
unsigned int	4	$0 \div 2^{32} - 1$, czyli przedział [0, 4294967295]
unsigned long	4	$0 \div 2^{32} - 1$, czyli przedział [0, 4294967295]
unsigned long long	8	$0 \div 2^{64} - 1$, czyli przedział [0, 18446744073709551615]

2. Zmienne rzeczywiste

Przechowują liczby zmiennoprzecinkowe. Gdy mamy zamiar w naszym programie wykorzystać ułamki, ten typ będzie najbardziej odpowiedni.

Nazwa	Wielkość (bajty)	Zakres
float	4	pojedyncza precyzja - dokładność 6 - 7 cyfr po przecinku
double	8	podwójna precyzja - dokładność 15 - 16 cyfr po przecinku
long double	12	liczby z ogromną dokładnością - 19 - 20 cyfr po przecinku

3. Zmienne znakowe

Przechowują znaki, które są kodowane kodem ASCII. Znak w pamięci nie może być przechowany jako znak, tylko jako pewna liczba, dlatego każdy znak ma swój odpowiednik liczbowy z zakresu [0, 255], który nazywamy kodem ASCII. Dla przykładu litera "d" ma wartość 100, "!" = 33, a "spacja" = 32.

Nazwa	Wielkość (bajty)	Zakres
char	1	-128 ÷ 127
unsigned char	1	0 ÷ 255

4. Zmienne logiczne

Przechowują jedną z dwóch wartości - true (prawda) albo false (fałsz). Wartość logiczna true jest równa 1, natomiast false ma wartość 0.

Nazwa	Wielkość (bajty)	Wartości
bool	1	true (1) false (0)

Uwaga!!! Dane w tabelach określają typy zdefiniowane w kompilatorze 32 bitowym Dev C++. W kompilatorach 64 bitowych, zakresy niektórych zmiennych mogą być większe.

UWAGA! Przed pierwszym użyciem zmiennej musimy ją zadeklarować!

Zmienne możemy podzielić na **zmienne lokalne** i **zmienne globalne**.

Programowanie w języku C++ opiera się na idei korzystania ze **zmiennych lokalnych**. Zmienne tego typy „widoczne" są tylko w określonym bloku (ograniczonym nawiasami klamrowymi), np. wewnątrz utworzonej funkcji czy wewnątrz pętli. Po opuszczeniu bloku zadeklarowane wewnątrz niego zmienne tracą swoją ważność.

Zmienne globalne widoczne są w każdym „zakątku" programu. Oznacza to, że można z nich korzystać w każdym miejscu programu. Trzeba jednak zwrócić uwagę, że przy tworzeniu tego typu zmiennych, istnieje niebezpieczeństwo przypadkowego nadpisania jej wartości, co może spowodować nieprawidłowe działanie programu. Zmienne globalne deklarujemy przed blokiem funkcji **main()**.

Popatrzmy na poniższy przykład przedstawiający przykładowe miejsce deklarowania zmiennych lokalnych i zmiennych globalnych:

```
/*
    Program obliczajacy pole powierzchni prostokata
    o bokach dlugosci 5 i 3
*/

#include <iostream>
using namespace std;

int a=5;        // DEKLARACJA ZMIENNEJ GLOBALNEJ

int main()
{
    int b=3;    // DEKLARACJA ZMIENNEJ LOKALNEJ

    cout << "Pole prostokata o bokach a=" << a << " i b=" << b;
    cout << " jest rowne " << a*b;

    return 0;
}
```

W kodzie źródłowym programu można używać komentarzy, które są pomocne w tworzeniu programu, ale pomijane są przez kompilator. Przykłady takich komentarzy widać w powyższym programie. Komentarze wstawiamy poprzedzając je dwoma ukośnikami **//** w przypadku komentarzy jednolinijkowych lub tworząc blok komentarza **/*** KOMENTARZ ***/** tworząc komentarz wielolinijkowy.

Jak widać w powyższym przykładzie, zarówno zmienna *a*, jak i zmienna *b* są zmiennymi całkowitymi i korzystając z nich możemy obliczyć pole powierzchni naszego prostokąta. Jedyną różnicą jest zakres ważności zmiennych, czyli zmienna *a* widziana jest w całym programie, natomiast zmienna *b* wyłącznie w funkcji **main()** naszego programu.

Czasami zachodzi potrzeba zmiany typu zmiennej wykorzystanej w naszym programie. Możemy w takiej sytuacji wykorzystać **rzutowanie typów** polegające na konwersji jednego typu na drugi. W przypadku konwersji liczby typu rzeczywistego na całkowitą, obetniemy część ułamkową liczby rzeczywistej, a w przypadku konwersji char -> int, zamiast znaku będziemy mieli do dyspozycji liczbę (czyli kod ASCII), pod jaką kryje się dany znak itd.

Popatrz na poniższy przykład:

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

int main()
{
    float a = 257.64;    // ZMIENNA TYPU RZECZYWISTEGO
    char b = 'K';        // ZMIENNA TYPU ZNAKOWEGO

    // RZUTOWANIE ZMIENNEJ a Z LICZBY RZECZYWISTEJ NA CALKOWITA
    cout << a << " po rzutowaniu jest rowne " << (int) a << endl;

    // RZUTOWANIE ZNAKU NA JEGO OZNACZENIE LICZBOWE W TABLICY KODÓW ASCII
    cout << b << " po rzutowaniu jest rowne " << (int) b << endl;

    return 0;
}
```

którego efektem działania jest dwulinijkowy komunikat:

```
257.64 po rzutowaniu jest rowne 257
K po rzutowaniu jest rowne 75
```

Przejdźmy teraz do omówienia instrukcji **cin** języka C++, o której wspomnieliśmy już w rozdziale 7:

cin >> a >> b;

Obiekt **cin** jest strumieniem wejścia konsoli (ang. console input). Pozwala on na pobranie danych z klawiatury i przypisanie ich do wcześniej zadeklarowanych zmiennych.

Popatrzmy na poniższy program obliczający pole powierzchni prostokąta o podanych przez użytkownika długościach boków *a* i *b*:

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

int main()
{
    int a,b;
    cout << "Podaj dlugosci bokow a i b prostokata: ";
    cin >> a >> b;
    cout << "Pole prostokata jest rowne : " << a * b;
    return 0;
}
```

Czas przejść do pisania bardziej rozbudowanych programów, wykorzystujących omówione w tym rozdziale zmienne oraz rzutowanie typów.

12. Instrukcje sterujące języka C++

12.1. Instrukcja warunkowa if ... else ...

Z instrukcją warunkową zetknęliśmy już się w rozdziale 3 podczas omawiania bloku decyzyjnego schematu blokowego. W języku C++ instrukcja warunkowa może przyjąć jedną z dwóch postaci:

```
if ( wyrażenie )
{
     instrukcja 1;
     instrukcja 2;
     instrukcja 3;
     ..............
}
else
{
     instrukcja 1;
     instrukcja 2;
     instrukcja 3;
     ..............
}
```

lub

```
if ( wyrażenie )
{
     instrukcja 1;
     instrukcja 2;
     instrukcja 3;
     ..............
}
```

Wyrażenie może przyjąć jedną z dwóch wartości logicznych, tzn. prawda (**true**) lub fałsz (**false**). W języku C++ przyjmuje się dodatkowo, że wartość 0 odpowiada wartości logicznej fałsz, a każda inna liczba wartości logicznej prawda.

Wyrażenie może być wyrażeniem prostym, np. ***x*>3**, ***a*<*b*** czy ***a*%*b*!=0** lub wyrażeniem złożonym będącym koniunkcją „i” (operator **&&**) lub alternatywą „lub” (operator **||**), np. **((*a*<*b*)&&(*a*<*c*))||((*b*==0)&&(*c*==0))**. Przedstawione powyżej wyrażenia omówimy w dalszej części skryptu.

Zwróć uwagę, że instrukcje umieszczone są w blokach. Jeśli instrukcja jest tylko jedna można pominąć bloki. Instrukcja warunkowa będzie miała wówczas postać:

```
if ( wyrazenie )
    instrukcja 1;
else
    instrukcja 1;
```

Nadszedł czas na przedstawienie instrukcji **if … else …** w praktyce.

Przykład 1.

Napisz program obliczający wartość bezwzględną podanej przez użytkownika liczby:

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

int main()
{
    float a;
    cout << "Podaj liczbe, ktorej wartosc bezwzgledna chcesz obliczyc: ";
    cin >> a;
    if(a<0)
        cout << "Wartosc bezwzgledna podanej liczby jest rowna " << -a;
    else
        cout << "Wartosc bezwzgledna podanej liczby jest rowna " << a;
    return 0;
}
```

Możemy wyeliminować instrukcję **else** wykorzystując instrukcję **if** nie do wypisania wyniku lecz do wykonania mnożenie przez -1 w przypadku podania przez użytkownika liczby mniejszej od zera.

Zauważ również, że zarówno w pierwszym jak i drugim programie instrukcja **if** nie posiada bloków, ponieważ wykonywane są pojedyncze instrukcje.

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

int main()
{
    float a;
    cout << "Podaj liczbe, ktorej wartosc bezwzgledna chcesz obliczyc: ";
    cin >> a;
    if(a<0)
        a *= -1;    //  lub inaczej  a = a * (-1);
    cout << "Wartosc bezwzgledna podanej liczby jest rowna " << a;
    return 0;
}
```

Przykład 2.

Napisz program sprawdzający parzystość podanej przez użytkownika liczby:

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

int main()
{
    int a;
    cout << "Podaj liczbe, ktorej parzystosc chcesz sprawdzic: ";
    cin >> a;

    if(a%2==0)
        cout << "Podana liczba jest parzysta.";
    else
        cout << "Podana liczba nie jest parzysta.";
    return 0;
}
```

Zwróć uwagę na wyrażenie instrukcji **if**: ***a*%2==0**. Wykorzystaliśmy w nim operator **%** pozwalający obliczyć resztę z dzielenia liczby *a* podanej przez użytkownika przez liczbę 2. Jeśli reszta z dzielenia jest równa 0 to oznacza, że *a* jest liczbą parzystą.

Popatrz na poniższą tabelę zawierającą operatory języka C++:

Operator	Opis zastosowania operatora
+	dodawanie
–	odejmowanie
*	mnożenie
/	dzielenie liczb rzeczywistych lub część całkowita dzielenia liczb całkowitych
%	reszta z dzielenia liczb całkowitych
==	równy
>	większy
>=	większy lub równy
<	mniejszy
<=	mniejszy lub równy
!=	różny, czyli nie jest równy
=	operator przypisania (przypisywanie wartości zmiennym)
&&	„i” (koniunkcja, czyli iloczyn zdań)
||	„lub” (alternatywa, czyli suma zdań)
!	„nie” (negacja, czyli zaprzeczenie zdań)

Bardzo często dwa operatory: „=” (przypisanie) i „==” (porównanie) mylone są przez początkujących programistów. Operator „=” służy do przypisania wartości zmiennej, np. a = 5, silnia = 120, liczba = 27.21, itp., natomiast operator „==” wykorzystujemy w celu sprawdzenia równości zmiennych lub równości zmiennej i liczby, np. a == b, x == 5, liczba1 == liczba2, itp., podobnie jak a < b czy x >= 100.

W kolejnym przykładzie wykorzystamy operator logiczny „&&”:

Przykład 3.

Napisz program do przykładu 1 z rozdziału 6, czyli sprawdzający warunek istnienia trójkąta dla boków długości *a*, *b* i *c* podanych przez użytkownika.

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

int main()
{
    unsigned int a,b,c;
    cout << "Podaj dlugosci odcinkow a, b i c, z których ma zostac zbudowany trojkat: ";
    cin >> a >> b >> c;

    if((a+b>c)&&(b+c>a)&&(a+c>b))
        cout << "Mozna zbudowac trojkat.";
    else
        cout << "Nie mozna zbudowac trojkata.";

    return 0;
}
```

Porównaj powyższy program z zapisaną wcześniej specyfikacją i schematem blokowym omawianego algorytmu.

12.2. Instrukcja wyboru switch

Za pomocą instrukcji warunkowej **if** możemy określić dokładnie co ma się wydarzyć w zależności od stanu jednej lub kilku zmiennych. Instrukcja **if** daje nam pełną kontrolę nad przebiegiem programu. W języku C++ jest jednak dostępna również instrukcja wielokrotnego wyboru **switch**. W przypadku niej możemy wykonywać decyzje tylko i wyłącznie na podstawie wartości jednej zmiennej. Możliwości instrukcji **switch** są nieporównywalnie mniejsze, jednak używanie jej w niektórych przypadkach jest znacznie korzystniejsze dla szybkości działania programu i estetyki kodu niż użycie instrukcji **if**.

Instrukcja **switch** ma następującą składnię:

```
switch(wyrazenie)
{
    case wartosc1: instrukcja1; break;
    case wartosc2: instrukcja2; break;
    ................................
    default: instrukcja_x;
}
```

Jeśli wartość wyrażenia odpowiada którejś z wartości przy jednej z etykiet **case**, wówczas wykonana zostanie instrukcja przy tej właśnie etykiecie oraz wszystkie instrukcje znajdujące się poniżej. W celu wywołania wyłącznie instrukcji wskazanej przez wyrażenie należy na końcu każdego bloku instrukcji dodać instrukcję **break**, która przekazuje sterowanie do pierwszej instrukcji po instrukcji **switch**.

Popatrz na poniższy przykład programu wypisującego nazwę podanego dnia tygodnia:

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

int main()
{
    int dzien;
    cout << "Ktory mamy dzisiaj dzien tygodnia: ";
    cin >> dzien;

    switch(dzien)
    {
        case 1: cout << "Dzisiaj jest poniedzialek."; break;
        case 2: cout << "Dzisiaj jest wtorek."; break;
        case 3: cout << "Dzisiaj jest sroda."; break;
        case 4: cout << "Dzisiaj jest czwartek."; break;
        case 5: cout << "Dzisiaj jest piatek."; break;
        case 6: cout << "Dzisiaj jest sobota."; break;
        case 7: cout << "Dzisiaj jest niedziela."; break;
        default: cout << "Podaj liczbe od 1 do 7!";
    }
    return 0;
}
```

Zapiszmy teraz powyższy program używając instrukcji **if** zamiast instrukcji **switch**:

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

int main()
{
    int dzien;
    cout << "Ktory mamy dzisiaj dzien tygodnia: ";
    cin >> dzien;

    if((dzien<1)||(dzien>7))
        cout << "Podaj liczbe od 1 do 7!";
    else
    {
        if(dzien==1) cout << "Dzisiaj jest poniedzialek.";
        if(dzien==2) cout << "Dzisiaj jest wtorek.";
        if(dzien==3) cout << "Dzisiaj jest sroda.";
        if(dzien==4) cout << "Dzisiaj jest czwartek.";
        if(dzien==5) cout << "Dzisiaj jest piatek.";
        if(dzien==6) cout << "Dzisiaj jest sobota.";
        if(dzien==7) cout << "Dzisiaj jest niedziela.";
    }
    return 0;
}
```

Pomijając instrukcję **break** oraz **default** możemy zmienić działanie programu w taki sposób, że nasz program zamiast wypisywać nazwę wskazanego dnia tygodnia wypisze wszystkie dni, które pozostały do końca tygodnia:

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

int main()
{
    int dzien;
    cout << "Ktory mamy dzisiaj dzien tygodnia: ";
    cin >> dzien;

    cout << "Do konca tygodnia mamy jeszcze:";
    switch(dzien)
    {
        case 1: cout << " poniedzialek,";
        case 2: cout << " wtorek,";
        case 3: cout << " srode,";
        case 4: cout << " czwartek,";
        case 5: cout << " piatek,";
        case 6: cout << " sobote i";
        case 7: cout << " niedziele.";
    }
    return 0;
}
```

Zapiszmy teraz powyższy program używając instrukcji **if** zamiast instrukcji **switch**:

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

int main()
{
    int dzien;
    cout << "Ktory mamy dzisiaj dzien tygodnia: ";
    cin >> dzien;
    cout << "Do konca tygodnia mamy jeszcze:";
    if(dzien<=1) cout << " poniedzialek,";
    if(dzien<=2) cout << " wtorek,";
    if(dzien<=3) cout << " srode,";
    if(dzien<=4) cout << " czwartek,";
    if(dzien<=5) cout << " piatek,";
    if(dzien<=6) cout << " sobote i";
    if(dzien<=7) cout << " niedziele.";
    return 0;
}
```

Jak widać, program w którym wykorzystano instrukcję **switch** jest znacznie czytelniejszy. Ponadto konstrukcja wyrażenia instrukcji **if** w pierwszym przykładzie może sprawić początkującemu programiście trudność.

12.3. Instrukcja pętli for

W przykładzie 4 rozdziału 4 oraz w rozdziale 5 omówiliśmy zastosowanie pętli w algorytmice.

W języku C++ mamy trzy rodzaje pętli. Pierwszą z nich jest pętla **for**, stosowana w sytuacjach gdy znamy ilość przebiegów pętli jeszcze przed rozpoczęciem jej pierwszego obiegu.

Przyjrzyjmy się składni instrukcji **for**:

```
for ( instrukcja początkowa ; warunek sterujący ; instrukcja kroku )
        instrukcja;
```

lub

```
for ( instrukcja początkowa ; warunek sterujący ; instrukcja kroku )
{
        instrukcja1;
        instrukcja2;
        ……..
}
```

Instrukcja początkowa zwana jest też instrukcją inicjującą. Wykonywana jest przed pierwszym obiegiem pętli i służy nadaniu początkowej wartości zmiennej sterującej pętlą.

Warunek sterujący to wyrażenie, którego logiczna wartość sprawdzana jest przed każdym obiegiem pętli. Jeśli jego wartość jest równa **true** (czyli jest różna od zera), to pętla wykona się po raz kolejny, w przeciwnym wypadku następuje wyjście z pętli.

Instrukcja kroku wykonywana jest po każdym przebiegu pętli, najczęściej modyfikuje tzw. licznik pętli.

Przykład 1:

Program wypisujący na ekranie monitora wszystkie liczby jednocyfrowe.

w kolejności rosnącej:

```
#include<iostream>
using namespace std;
int main()
{
    for(int i=0;i<10;i++)
        cout << i << " ";
    return 0;
}
```

w kolejności malejącej:

```
#include<iostream>
using namespace std;
int main()
{
    for(int i=9;i>=0;i--)
        cout << i << " ";
    return 0;
}
```

Przykład 2:

Program wypisujący na ekranie monitora wszystkie liczby parzyste trzycyfrowe.

w kolejności rosnącej:

```
#include<iostream>
using namespace std;
int main()
{
    for(int i=100;i<=999;i=i+2)
        cout << i << " ";
    return 0;
}
```

w kolejności malejącej:

```
#include<iostream>
using namespace std;
int main()
{
    for(int i=998;i>=100;i=i-2)
        cout << i << " ";
    return 0;
}
```

12.4. Instrukcja pętli while

Kolejną z omawianych przez nas pętli będzie pętla **while**, stosowana w sytuacjach, gdy nie znamy ilość przebiegów pętli. Możemy jednak stosować ją również w sytuacjach, w których stosujemy pętle **for**.

Przyjrzyjmy się składni instrukcji **while**:

```
while ( wyrażenie )
    instrukcja;
```

lub

```
while ( wyrażenie )
{
    instrukcja1;
    instrukcja2;
    ……..
}
```

Wyrażenie przyjmuje wartość logiczną „prawda” lub „fałsz”.

Przykład 1:

Program obliczający sumę liczb pobieranych z klawiatury do momentu, aż pobrane zostanie zero i wypisujący na ekranie obliczoną sumę.

```
#include<iostream>
using namespace std;
int main()
{
    int liczba, suma=0;
    while(liczba!=0)
    {
        cout << "Podaj liczbe: ";
        cin >> liczba;
        suma += liczba;   // lub suma = suma + liczba;
    }
    cout << "Suma podanych liczb jest rowna " << suma;
    return 0;
}
```

Przykład 2:

Program sprawdzający zgodność hasła zapisanego w programie z hasłem podanym przez użytkownika.

```
#include<iostream>
using namespace std;
int main()
{
    string haslo1="www.liceumxv.edu.pl", haslo2;
    while(haslo1!=haslo2)
    {
        cout << "Podaj haslo: ";
        cin >> haslo2;
    }
    cout << "Podano poprawne haslo!";
    return 0;
}
```

W naszym programie użyte zostały zmienne typu **string**, o których nie wspominaliśmy wcześniej. W rzeczywistości **string** to nie zmienna lecz tablica znaków, którą omówimy w jednym z późniejszych rozdziałów.

W naszym programie hasłem zapisanym w programie (haslo1) jest adres internetowy strony naszej szkoły: ***www.liceumxv.edu.pl*** Dopóki użytkownik nie poda poprawnego hasła (haslo2) program będzie prosił o jego podanie. W Podobny sposób działają programy, które podczas instalacji wymagają podania klucza, choć większość z nich zamiast pobierać hasło z programu pobiera je z internetowej bazy haseł danego programu.

Popatrz na powyższe dwa przykłady. Jak można zauważyć, nie jesteśmy w stanie przewidzieć po ilu pobranych z klawiatury liczbach użytkownik zdecyduje się na podanie liczy zero oraz po ilu nieudanych próbach podane zostanie poprawne hasło. Omawiana w poprzednim podrozdziale pętla **for** nie znajdzie więc tutaj zastosowania. Możemy jednak bez większych trudności wykorzystać pętle **while** zbudowania programów z rozdziału opisującego pętlę **for**, np. porównaj programy z przykładu 2 z poprzedniego podrozdziału z poniższymi programami:

```
#include<iostream>
using namespace std;
int main()
{
    int i=100;
    while(i<=999)
    {
        cout << i << " ";
        i=i+2;
    }
    return 0;
}
```

```
#include<iostream>
using namespace std;
int main()
{
    int i=998;
    while(i>=100)
    {
        cout << i << " ";
        i=i-2;
    }
    return 0;
}
```

12.5. Instrukcja pętli do…while

Nieco inną odmianą pętli **while** jest pętla **do…while**. Również używamy jej w sytuacjach, gdy nie znamy ilość przebiegów pętli. Tym co odróżnia ją od pętli **while** jest to, że uruchamia się ona zawsze przynajmniej jeden raz, bez względu na wartość wyrażenia pętli.

Przyjrzyjmy się składni instrukcji **do…while**:

```
do
    instrukcja;
while ( wyrażenie );
```

lub

```
do
{
    instrukcja1;
    instrukcja2;
    ……..
}
while ( wyrażenie );
```

Wyrażenie przyjmuje wartość logiczną „prawda” lub „fałsz”.

Jako przykład pokażemy dwa programy, z wykorzystaniem pętli **while** i **do…while**, których zadaniem jest pobieranie z klawiatury liczb do czasu, aż podana zostanie liczba większa od zera:

```
#include<iostream>
using namespace std;
int main()
{
    int x;
    while(x<=0)
    {
        cout << "Podaj liczbe: ";
        cin >> x;
    }
    cout << "Podano liczbe " << x;
    return 0;
}
```

```
#include<iostream>
using namespace std;
int main()
{
    int x;
    do
    {
        cout << "Podaj liczbe: ";
        cin >> x;
    }
    while(x<=0);
    cout << "Podano liczbe " << x;
    return 0;
}
```

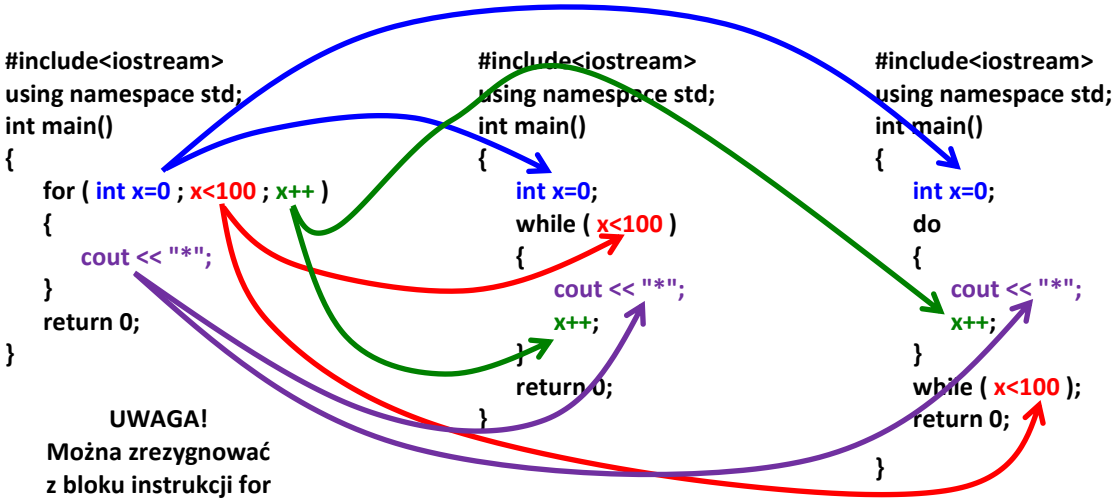
Jak zapewne zauważysz, po uruchomieniu pierwszego programu na ekranie pojawi się informacja, że liczba została już podana, natomiast po uruchomieniu drugiego programu pokaże się komunikat z prośbą o podanie liczby. Pomimo, że programy są prawie identyczne, pierwszy z nich nie działa poprawnie, dlatego, że pętla nie wykona ani jednego przebiegu. Zmienne podczas deklarowania mają przypisane początkowe wartości. Dla przykładu zmienna *x* z programu pierwszego ma wartość w naszym przypadku równą 32763, co widać na poniższym zrzucie ekranu.

```
Podano liczbe 32763
--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

13. Przykłady programów iteracyjnych

Zanim zaczniemy omawiać bardziej rozbudowane programy, porównajmy wszystkie trzy pętle omówione w poprzednim rozdziale.

Napiszmy program wypisujący na ekranie 100 gwiazdek:



Jak widzisz, pomimo iż do napisania naszego programu wykorzystane zostały trzy różne pętle, to w każdej z nich pojawiły się dokładnie te same elementy opisane w składni instrukcji **for**, tzn. **instrukcja początkowa**, **warunek sterujący**, **instrukcja kroku** i **instrukcja**, która ma być wykonywana przy każdym obiegu pętli.

Przejdźmy teraz do przedstawienia przykładowych programów z zastosowaniem wcześniej omawianych instrukcji.

Przykład 1

Napisz program wypisujący wszystkie dzielniki podanej przez użytkownika liczby z wykorzystaniem poznanych pętli.

PROGRAM 1:

```
#include<iostream>
using namespace std;
int main()
{
    int liczba;
    cout << "Podaj liczbe, ktorej dzielniki chcesz wypisac: ";
    cin >> liczba;
    cout << "Dzielnikami liczby " << liczba << " sa: ";
    for(int i=1;i<=liczba;i++)
        if(liczba%i==0)
            cout << i << " ";
    return 0;
}
```

PROGRAM 2:

```
#include<iostream>
using namespace std;
int main()
{
    int liczba;
    cout << "Podaj liczbe, ktorej dzielniki chcesz wypisac: ";
    cin >> liczba;
    cout << "Dzielnikami liczby " << liczba << " sa: ";
    int i=1;
    while(i<=liczba)
    {
        if(liczba%i==0)
            cout << i << " ";
        i++;
    }
    return 0;
}
```

PROGRAM 3:

```
#include<iostream>
using namespace std;
int main()
{
    int liczba;
    cout << "Podaj liczbe, ktorej dzielniki chcesz wypisac: ";
    cin >> liczba;
    cout << "Dzielnikami liczby " << liczba << " sa: ";
    int i=1;
    do
    {
        if(liczba%i==0)
            cout << i << " ";
        i++;
    }
    while(i<=liczba);
    return 0;
}
```

Przykład 2

Napisz program obliczający silnię podanej przez użytkownika liczby z wykorzystaniem poznanych pętli.

W przykładzie 2 rozdziału 6 zapisaliśmy specyfikację oraz algorytm obliczania silni podanej liczby. Nadszedł czas na napisanie programu w języku C++.

Zwróć uwagę jak zmienia się położenie i wartość zmiennej *x* w poniższych programach w zależności od zastosowanej pętli.

PROGRAM 1:

```
#include<iostream>
using namespace std;
int main()
{
    int a, s=1;
    cout << "Podaj liczbe, z ktorej silnie chcesz obliczyc: ";
    cin >> a;
    for(int x=2;x<=a;x++)
        s*=x;   // co mozna zapisac rowniez jako s = s * x;
    cout << "Silnia z liczby " << a << " jest rowna " << s;
    return 0;
}
```

PROGRAM 2:

```
#include<iostream>
using namespace std;
int main()
{
    int a, s=1, x=2;
    cout << "Podaj liczbe, z ktorej silnie chcesz obliczyc: ";
    cin >> a;
    while(x<=a)
    {
        s*=x;   // co mozna zapisac rowniez jako s = s * x;
        x++;
    }
    cout << "Silnia z liczby " << a << " jest rowna " << s;
    return 0;
}
```

PROGRAM 3:

```
#include<iostream>
using namespace std;
int main()
{
    int a, s=1, x=1;
    cout << "Podaj liczbe, z ktorej silnie chcesz obliczyc: ";
    cin >> a;
    do
    {
        s*=x;   // co mozna zapisac rowniez jako s = s * x;
        x++;
    }
    while(x<=a);
    cout << "Silnia z liczby " << a << " jest rowna " << s;
    return 0;
}
```

Przykład 3

Napisz program obliczający dowolną potęgę naturalną dowolnej liczby naturalnej z wykorzystaniem poznanych pętli.

PROGRAM 1:

```
#include<iostream>
using namespace std;
int main()
{
    int podstawa, wykladnik, potega=1;
    cout << "Podaj podstawe potegi: ";
    cin >> podstawa;
    cout << "Podaj wykladnik potegi: ";
    cin >> wykladnik;
    cout << podstawa << "^" << wykladnik << " = ";
    for(wykladnik;wykladnik;wykladnik--)
        potega *= podstawa;
    cout << potega ;
    return 0;
}
```

Zwróć uwagę, że instrukcja **for** wygląda zupełnie inaczej niż w dotychczas omawianych programach. Instrukcja początkowa nie zawiera deklaracji zmiennej, lecz wskazuje zmienną *wykladnik*, która została wcześniej zadeklarowana i ma wartość podaną przez użytkownika.

Warunkiem sterującym jest w naszej pętli również zmienna *wykladnik*. Pamiętasz jak długo wykonywana jest pętla? Zgodnie z opisem instrukcji **for** z rozdziału 12.3 pętla wykonuje się tak długo, dopóki warunek sterujący ma wartość ***true***, czyli jest różny od zera. W naszej pętli, po każdym jej obiegu wartość zmiennej *wykladnik* zmniejsza się o 1, a co za tym idzie, po pewnej ilości wykonanych obiegów jej wartość będzie równa 0. Zmienna *wykladnik* przekaże wówczas pętli wartość ***false*** i program przejdzie do wykonania instrukcji znajdujących się poniżej pętli.

Jeśli zrozumienie tej konstrukcji sprawia Ci problemy, porównaj tę instrukcję:

for (wykladnik ; wykladnik ; wykladnik--)

z instrukcją spełniającą dokładnie takie samo zadanie postaci:

for (wykladnik ; wykladnik>0 ; wykladnik--)

lub instrukcją, którą już znasz:

for (int x=wykladnik ; x>0 ; x--)

gdzie kolorami oznaczone zostały: **instrukcja początkowa**, **warunek sterujący** i **instrukcja kroku**.

Uproszczony zapis wyrażeń zastosujemy również w poniższych programach w instrukcjach **while**, **do…while** i **if**.

PROGRAM 2:

```
#include<iostream>
using namespace std;
int main()
{
    int podstawa, wykladnik, potega=1;
    cout << "Podaj podstawe potegi: ";
    cin >> podstawa;
    cout << "Podaj wykladnik potegi: ";
    cin >> wykladnik;
    cout << podstawa << "^" << wykladnik << " = ";
    while(wykladnik)
    {
        potega *= podstawa;
        wykladnik--;
    }
    cout << potega ;
    return 0;
}
```

PROGRAM 3:

```
#include<iostream>
using namespace std;
int main()
{
    int podstawa, wykladnik, potega=1;
    cout << "Podaj podstawe potegi: ";
    cin >> podstawa;
    cout << "Podaj wykladnik potegi: ";
    cin >> wykladnik;
    cout << podstawa << "^" << wykladnik << " = ";
    do
    {
        if(wykladnik)
        {
            potega *= podstawa;
            wykladnik--;
        }
    }
    while(wykladnik);
    cout << potega ;
    return 0;
}
```

W kolejnych przykładach ograniczymy się do napisania programów z wykorzystaniem wyłącznie jednego rodzaju pętli. Pozostałe dwie wersje programów pozostawiamy Tobie drogi czytelniku do samodzielnego napisania.

Przykład 4

Napisz program obliczający sumę liczb z określonego przez użytkownika przedziału < min ; max >.

```
#include<iostream>
using namespace std;
int main()
{
    int min, max, suma=0;
    cout << "Podaj najmniejsza liczbe przedzialu: ";
    cin >> min;
    cout << "Podaj najwieksza liczbe przedzialu: ";
    cin >> max;
    while(min<=max)
    {
        suma += min;
        min++;
    }
    cout << "Suma podanych liczb jest rowna " << suma;
    return 0;
}
```

Przykład 5

Napisz program obliczający średnią arytmetyczną *x* podanych przez użytkownika liczb, gdzie *x* jest liczbą podaną przez użytkownika.

```
#include<iostream>
using namespace std;
int main()
{
    unsigned int x, liczba, suma=0, i=1;
    cout << "Ile liczb chcesz podac? ";
    cin >> x;
    do
    {
        cout << "Podaj " << i << " liczbe: ";
        cin >> liczba;
        suma += liczba;
        i++;
    }
    while(i<=x);
    cout << "Srednia arytmetyczna podanych liczb jest rowna " << (float)suma/x;
    return 0;
}
```

Zwróć uwagę, że w naszym programie wykorzystaliśmy omówione w rozdziale 10 rzutowanie ilorazu dwóch liczb naturalnych na liczby rzeczywiste.

Przykład 6

Napisz program wyszukujący największą i najmniejszą spośród wprowadzonych przez użytkownika 10 liczb całkowitych.

```
#include<iostream>
using namespace std;
int main()
{
    int min, max, liczba;
    for(int i=1;i<=10;i++)
    {
        cout << "Podaj " << i << " liczbe: ";
        cin >> liczba;
        if(i==1)
        {
            min = liczba;   // przy pierwszym obiegu petli przypisujemy
            max = liczba;   // zmiennym min i max pierwsza podana liczbe
        }
        else
        {
            if(liczba<min) min = liczba;
            if(liczba>max) max = liczba;
        }
    }
    cout << "Najmniejsza liczba jest " << min << ", a najwieksza " << max;
    return 0;
}
```

Ćwiczenie 1. Napisz program wypisujący na ekranie wszystkie liczby dwucyfrowe, parzyste, podzielne przez 7.

Ćwiczenie 2. Napisz program wypisujący na ekranie szlaczek składający się z 40 znaków postaci „+=+=+=+=+=+=…”.

Ćwiczenie 3. Napisz program wypisujący na ekranie 400 gwiazdek po 20 gwiazdek w każdym wierszu.

Ćwiczenie 4. Napisz program, którego zadaniem jest wczytywanie z klawiatury liczb do czasu, aż podana zostanie przez użytkownika liczba większa od zera.

Ćwiczenie 5. Napisz program wypisujący na ekranie wszystkie duże litery angielskiego alfabetu. W realizacji zadania wykorzystaj rzutowanie liczby całkowitej **int** na znak **char**. Przyda Ci się również informacja, że w tabeli kodów ASCII wielkim literom alfabetu odpowiadają liczby z zakresu od 65 do 90.

Ćwiczenie 6. Napisz program, którego zadaniem jest wczytywanie z klawiatury 10 liczb całkowitych oraz wypisanie informacji ile podanych zostało liczb ujemnych oraz liczb nieujemnych.