



**Pascal C, C++**  
nie tylko dla uczniów  
i studentów

# Od matematyki do programowania

**Wszystko, co każdy programista wiedzieć powinien**

*Wiesław Rychlicki*



## » Idź do

- Spis treści
- Przykładowy rozdział
- Skorowidz

## » Katalog książek

- Katalog online
- Zamów drukowany katalog

## » Twój koszyk

- Dodaj do koszyka

## » Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

## » Czytelnia

- Fragmenty książek online

## » Kontakt

Helion SA  
ul. Kościuszki 1c  
44-100 Gliwice  
tel. 32 230 98 63  
e-mail: helion@helion.pl  
© Helion 1991–2011

## Od matematyki do programowania. Wszystko, co każdy programista wiedzieć powinien

Autor: Wiesław Rychlicki  
ISBN: 978-83-246-3210-7  
Format: 168×237, stron: 320



### „Wędrówka do źródła kodu”

Popularna definicja programowania określa je jako „proces projektowania, tworzenia, testowania i utrzymywania kodu źródłowego programów komputerowych lub urządzeń mikroprocesorowych”.

Wspomniany kod źródłowy może być napisany w różnych językach programowania, z użyciem określonych reguł. Każdy z języków pozwala na wykorzystanie odpowiednich stylów programowania, a wybór konkretnego języka może zależeć od indywidualnych upodobań, polityki firmy lub funkcji, jakie końcowa aplikacja ma realizować. W zasadzie nie istnieje odpowiedź na pytanie, który z języków jest najlepszy.

Dlatego w tej książce nie znajdziesz typowego abecadła. Zapoznasz się za to z danym problemem, a następnie programem komputerowym służącym do jego rozwiązania. Jeśli chcesz wreszcie rozpocząć przygodę z programowaniem i nawiązać dialog ze swoim komputerem, ta publikacja jest właśnie dla Ciebie! Różnorodne obliczenia, mniej lub bardziej skomplikowane, znane Ci z lekcji matematyki lub nieznacznie wykraczające poza program nauczania, stanowią tutaj podstawę do zdobywania informacji na temat programowania w wybranych językach.

Wybrane zadania zaprezentowane są w popularnych językach programowania: Pascal, C i C++. Stosowane algorytmy wymagają także sięgnięcia po różne funkcje matematyczne, dostępne standardowo w bibliotekach języków programowania oraz konstruowane na podstawie wzorów.

**Zostań informatycznym poliglotą. Programuj każdego dnia!**



# SPIS TREŚCI



<b>Wstęp</b>	<b>6</b>
<b>Rozdział 1. Podstawowe pojęcia, czyli mały elementarz...</b>	<b>9</b>
Co wiemy o liczbach?	9
Systemy zapisu liczb	16
Od problemu do programu... — słownik początkującego programisty	21
Kilka zdań o językach programowania	27
Pierwszy program — klasyczne przykłady w popularnych językach	32
Edycja, kompilacja i uruchomienie programu	36
<b>Rozdział 2. Proste obliczenia — pola i obwody figur geometrycznych</b>	<b>40</b>
Programy o strukturze liniowej	40
Instrukcje warunkowe i sprawdzanie poprawności danych	49
Pętle, czyli powtarzanie sekwencji wykonywanych czynności	52
Porównania, operatory logiczne i budowanie warunków złożonych	59
Stosowanie wybranych funkcji matematycznych i definiowanie własnych funkcji	70
<b>Rozdział 3. Podejmowanie decyzji, czyli nieco więcej o instrukcjach warunkowych</b>	<b>75</b>
Różne przypadki w prostych równaniach	75
Algorytm rozwiązywania równania kwadratowego	79
Rozwiązywanie równań wyższych stopni	85
Wybór jednej z wielu opcji...	98
Dialog programu z użytkownikiem — dane tekstowe	106

<b>Rozdział 4. Instrukcje iteracyjne bez tajemnic</b>	<b>112</b>
Pętla o stałej liczbie powtórzeń — przykłady tablicowania funkcji	112
Pętla ze sprawdzaniem warunku na końcu	120
Pętla ze sprawdzaniem warunku na początku	123
Która pętla lepsza, czyli krótkie porównanie instrukcji	124
Przerywanie działania pętli	127
<b>Rozdział 5. Budujemy własne funkcje i procedury</b>	<b>131</b>
Zmienne globalne i lokalne	131
Przekazywanie danych do procedur i funkcji, zwracanie wyników	132
Obliczanie potęg o wykładniku całkowitym	142
Konwersja jednostek miary kątów	145
Funkcje trygonometryczne i funkcje do nich odwrotne	151
To się jeszcze może przydać, czyli jak stworzyć własny moduł lub bibliotekę	156
<b>Rozdział 6. Funkcje i procedury rekurencyjne</b>	<b>163</b>
Kilka funkcji znanych ze szkoły	163
Symbol Newtona i trójkąt Pascala	168
Algorytm Euklidesa — wersja rekurencyjna	170
Liczby Fibonacciego	172
Koniec świata i wieże Hanoi	173
Rekurencja zamiast iteracji...	174
<b>Rozdział 7. Liczby w matematyce i komputerze</b>	<b>178</b>
Liczby naturalne i całkowite	178
Ułamki zwykłe — cztery podstawowe działania	187
Ułamki łańcuchowe	196
Liczby zmiennoprzecinkowe	199

<b>Rozdział 8. Strukturalne typy danych — tablice i rekordy</b>	<b>208</b>
Działania na tekstach — łańcuchowy typ danych	208
Tablicowe typy danych — tablice jedno- i wielowymiarowe	217
Rekordy i struktury	223
Tablica struktur	232
<b>Rozdział 9. Liczby niewymierne i ich przybliżenia dziesiętne</b>	<b>236</b>
Pierwiastek drugiego stopnia z 2	236
Sposoby obliczania pierwiastków drugiego stopnia	245
Obliczanie pierwiastków trzeciego stopnia	247
Obliczanie pierwiastków wyższych stopni	251
Złoty podział odcinka, liczba $\varphi$ i ciąg Fibonacciego	252
Rozwinięcie dziesiętne liczby pi	258
Podstawa logarytmu naturalnego — liczba e	265
<b>Rozdział 10. Ciągi i szeregi liczbowe</b>	<b>268</b>
Sumowanie wyrazów ciągu liczbowego	268
Rozwinięcie funkcji w szereg liczbowy — szeregi funkcyjne	276
<b>Rozdział 11. Podstawowe operacje na plikach</b>	<b>287</b>
Zapisywanie i odczytywanie plików tekstowych	287
Sformatowane dane liczbowe w plikach tekstowych	295
Zapisywanie danych liczbowych w plikach binarnych	298
Modyfikacja danych w pliku binarnym	305
<b>Rozdział 12. Co dalej?</b>	<b>312</b>
<b>Bibliografia</b>	<b>314</b>
<b>Skorowidz</b>	<b>315</b>



# INSTRUKCJE ITERACYJNE BEZ TAJEMNIC

## Pętle o stałej liczbie powtórzeń — przykłady tablicowania funkcji

### Przykład 4.1.

Obliczymy i wyświetlimy na ekranie kwadraty i sześciany liczb naturalnych w zakresie od 1 do 15. Zauważmy, że czynność obliczania i wyświetlania należy powtórzyć 15 razy. Nie mamy prostej instrukcji w stylu *powtarzaj 15 instrukcja*<sup>1</sup>.

W tym celu musimy utworzyć tzw. zmienną sterującą pełniącą najczęściej rolę licznika powtórzeń i ustalić zakres przyjmowanych przez tę zmienną wartości, np. od 1 do 15 (*odliczanie z dołu do góry*).

Następnie sprawdzamy, czy wartość zmiennej sterującej mieści się w podanym zakresie:

- jeśli **tak**, wykonujemy instrukcje (te do powtarzania), zwiększamy wartość zmiennej sterującej (lub zmniejszamy — *odliczanie z góry w dół*) i ponownie sprawdzamy, czy zmienna sterująca mieści się w zakresie;
- jeśli **nie**, kończymy działanie pętli.

Możemy to zrealizować przy użyciu pętli `for`:

<b>Pascal</b>	<pre>for i := 1 to 15 do   instrukcja_do_powtarzania;</pre>
<b>C, C++</b>	<pre>int i; for (i = 1; i &lt;= 15; ++i)   instrukcja_do_powtarzania;</pre>
<b>C++</b>	<pre>for (int i = 1; i &lt;= 15; ++i)   instrukcja_do_powtarzania;</pre>

Zmienną sterującą oznaczyliśmy identyfikatorem `i`. W Pascalu zmienna sterująca musi być zmienną typu porządkowego (liczbą całkowitą — typy: `byte`, `short`, `word`, `integer`, `long`, znakiem — typ `char` lub wartością logiczną — typ `boolean`) zadeklarowaną w części deklarycyjnej (przed głównym blokiem programu, procedury lub funkcji). W składni instrukcji widzimy wyraźnie ustalenie zakresu zmiennej `for i := 1 to 15...` i wskazanie instrukcji do

<sup>1</sup> W języku LOGO taka procedura istnieje: `repeat 15 [lista_czynności_do_wykonania]`.

wykonania ...do *instrukcja\_do\_powtarzania*;. Słowo kluczowe to sygnalizuje, że w cyklu wartość zmiennej sterującej będzie zwiększana i o to już nie musimy się troszczyć. Aby dokonać odliczania *z góry w dół* (zmniejszania zmiennej sterującej), należałoby użyć słowa kluczowego `downto` i konstrukcji `for i := 15 downto 1 do instrukcja`;

W C i C++ instrukcja cyklu typu `for` ma inną składnię i o wiele szersze możliwości. W tym przykładzie wiernie naśladujemy instrukcję `for` z Pascala. Zmienną sterującą zadeklarowaliśmy lokalnie jako liczbę typu całkowitego `int i`; i nadaliśmy jej wartość początkową `i = 0`; (w języku C++ możliwe jest zadeklarowanie zmiennej sterującej „wewnątrz” instrukcji cyklu `for (int i = 1; ...; ...)` ...; i wtedy jej zasięg ogranicza się tylko do tej instrukcji). Górny zakres wartości zmiennej sterującej zawieramy w warunku do sprawdzenia `i <= 15`; , a zwiększenie zmiennej sterującej (na koniec każdego cyklu) realizujemy instrukcją `++i` (preinkrementacja), `i++` (postinkrementacja), `i += 1` lub `i = i + 1` (wszystkie te instrukcje zwiększą w efekcie wartość zmiennej `i` o 1 — różnicę pomiędzy post- i preinkrementacją omówimy nieco później). Odliczanie *z góry w dół* można zrealizować tak: `int i; for (i = 15; i >= 1; --i) instrukcja_do_powtarzania`;

Do zrealizowania pozostało obliczanie i wyświetlanie wartości kwadratów i sześciątów, a argumentem będzie zmieniająca się wartość zmiennej sterującej.

<b>Pascal</b>	<pre>for i := 1 to 15 do begin   kw := i*i;   sz := kw*i;   writeln(i:3, kw:5, sz:7); end;</pre>
<b>C, C++</b>	<pre>int i; for (i = 1; i &lt;= 15; ++i) {   int kw, sz;   kw = i*i;   sz = kw*i;   printf("%3d%5d%7d\n", i, kw, sz); }</pre>
<b>C++</b>	<pre>for (int i = 1; i &lt;= 15; ++i) {   int kw, sz;   kw = i*i;   sz = kw*i;   cout.width(3); cout &lt;&lt; i;   cout.width(5); cout &lt;&lt; kw;   cout.width(7); cout &lt;&lt; sz &lt;&lt; endl; }</pre>

Zwróćmy uwagę na sposób ustawienia wyświetlania szerokości kolumn zawierających liczbę, jej kwadrat i sześciąt na odpowiednio: 3, 5 i 7 znaków. W C++ realizujemy to przez wywołanie dla strumienia wyjściowego (obiekту) `cout` metody<sup>2</sup> `width(n)` z odpowiednim parametrem

<sup>2</sup> Łatwiej to będzie zrozumieć, gdy Czytelnik pozna programowanie obiektowe. Teraz po prostu przyjmijmy, że tak trzeba zrobić.

i nawet gdyby wszystkie kolumny miały stałą szerokość (np. 7 znaków), metodę tę musimy wywołać za każdym razem przed wstawieniem wartości do strumienia (operator <<). W C i Pascalu kod jest dostatecznie czytelny. Kompletne kody zawarte są w plikach na FTP: *p4\_1.pas*, *p4\_1.c*, *p4\_1.cpp*.

Problem formatowania wyników w C++ możemy też rozwiązać poprzez włączenie pliku nagłówkowego `#include <iomanip>` i wykorzystanie manipulatora `setw(n)` (FTP *p4\_1b.cpp*).

```
cout << setw(3) << i << setw(5) << kw << setw(7) << sz << endl;
```

Możemy sobie darować używanie zmiennych `kw` i `sz`. Wyrażenia `i*i` oraz `i*i*i` możemy umieścić jako parametry wywołania funkcji (procedury) wyświetlającej wynik, co uprości kod naszego programu, chociaż nieznacznie wzrośnie przez to liczba wykonanych mnożeń (FTP: *p\_4\_1a.pas*, *p\_4\_1a.c*, *p\_4\_1a.cpp*).

<b>Pascal</b>	<pre>for i := 1 to 15 do   writeln(i:3, i*i:5, i*i*i:7);</pre>
<b>C, C++</b>	<pre>int i; for (i = 1; i &lt;= 15; ++i)   printf("%3d%5d%7d\n", i, i*i, i*i*i);</pre>
<b>C++</b>	<pre>for (int i = 1; i &lt;= 15; ++i)   cout &lt;&lt; setw(3) &lt;&lt; i &lt;&lt; setw(5) &lt;&lt; i*i &lt;&lt; setw(7) &lt;&lt;   ↪ i*i*i &lt;&lt; endl;</pre>

## Przykład 4.2.

Obliczymy i wyświetlimy na ekranie pierwiastki kwadratowe z liczb od 0 do 2 z krokiem 0,1. Wynik wyświetlimy na ekranie z dokładnością do 8 miejsc po przecinku. Użycie instrukcji cyklu typu `for` w tym przypadku jest również możliwe, jeśli w jakiś sposób powiążemy całkowitą wartość zmiennej sterującej z ułamkowym argumentem funkcji. Zauważmy, że łącznie będzie to 21 liczb (0,0; 0,1; ...; 1,9; 2,0). Gdyby w tym ciągu liczb „nie widzieć” przecinka, moglibyśmy zmieniać zmienną sterującą w zakresie od 0 do 20. Wtedy argumentem funkcji obliczającej pierwiastek byłaby liczba 10 razy mniejsza od zmiennej sterującej.

<b>Pascal</b>	<pre>for i := 0 to 20 do writeln(i/10:5:1, sqrt(i/10):12:8);</pre>
<b>C, C++</b>	<pre>int i; for (i = 0; i &lt;= 20; ++i)   printf("%5.1lf%12.8lf\n", i/10.0, sqrt(i/10.0);</pre>
<b>C++</b>	<pre>cout.setf(ios::fixed); for (int i = 0; i &lt;= 20; ++i) {   cout &lt;&lt; setw(5) &lt;&lt; setprecision(1) &lt;&lt; i/10.0;   cout &lt;&lt; setw(12) &lt;&lt; setprecision(8) &lt;&lt; sqrt(i/10.0) &lt;&lt;   ↪ endl; }</pre>

O formatowaniu liczb zmiennoprzecinkowych wspominaliśmy w przykładzie 2.3.



Pełne kody programów zawarte są w plikach na FTP: *p4\_2.pas*, *p4\_2.c*, *p4\_2.cpp*. W podanych programach możemy wprowadzić dodatkową zmienną *x* typu zmiennoprzecinkowego, która posłuży nam do wyliczania argumentów dla funkcji *sqrt* (FTP: *p4\_2a.pas*, *p4\_2a.c*, *p4\_2a.cpp*).

<b>Pascal</b>	<pre> for i := 0 to 20 do   begin     x := i/10;     writeln(x:5:1, sqrt(x):12:8);   end; </pre>
<b>C, C++</b>	<pre> double x; int i; for (i = 0; i &lt;= 20; ++i) {   x = i/10.0;   printf("%5.1lf%12.8lf\n", x, sqrt(x)); } </pre>
<b>C++</b>	<pre> double x; cout.setf(ios::fixed); for (int i = 0; i &lt;= 20; ++i) {   x = i/10.0;   cout &lt;&lt; setprecision(1) &lt;&lt; setw(5) &lt;&lt; x;   cout &lt;&lt; setprecision(8) &lt;&lt; setw(12) &lt;&lt; sqrt(x) &lt;&lt; endl; } </pre>

Sposób wyświetlania wyników w stylu języka C++ (z użyciem klasy obiektów klasy *iostream* i manipulatorów — potrzebna dyrektywa `#include <iomanip>`) Czytelnik może sam wykorzystać w kolejnych przykładach. W kolejnych wariantach rozwiązania problemu podamy jedynie zmiany w sposobie użycia zmiennej sterującej lub zmiany w warunku kontynuacji pętli.

Można też w inny sposób powiązać zmienną sterującą z argumentem funkcji (Pascal:  $x := 0.1*i$ ; C/C++:  $x = 0.1*i$ ). Możemy również „oddzielić” licznik powtórzeń (zmienną sterującą) od argumentu funkcji. Wystarczy zmiennej *x* nadać wartość początkową (w tym przypadku 0) i w każdym cyklu po wykonaniu obliczeń zwiększać jej wartość o krok 0.1 (FTP: *p4\_2b.pas*, *p4\_2b.c*, *p4\_2b.cpp*).

<b>Pascal</b>	<pre> x := 0; for i := 0 to 20 do   begin     writeln(x:5:1, sqrt(x):12:8);     x := x + 0.1;   end; </pre>
<b>C, C++</b>	<pre> double x = 0; int i; for (i = 0; i &lt;= 20; ++i) {   printf("%5.1lf%12.8lf\n", x, sqrt(x));   x += 0.1; } </pre>

Jak wcześniej wspomnieliśmy, w C/C++ zmienna sterująca pętlą nie musi być liczbą całkowitą. Możemy zatem ze zmiennej *i* zrezygnować i po nieznaczącej modyfikacji kodu wykorzystać w tym celu zmienną *x* (FTP: *p4\_2c.c*, *p4\_2c.cpp*).

```
C, C++
double x;
for (x = 0; x <= 2; x += 0.1)
    printf("%5.11f%12.8lf\n", x, sqrt(x));
```

Uważny Czytelnik testujący wszystkie podane przykłady zauważy różnicę pomiędzy działaniem programów: *p4\_2b.c* i *p4\_2c.c* (podobnie: *p4\_2b.cpp* i *p4\_2c.cpp*). Obie wersje programu realizują ten sam algorytm, ale w przykładzie *b* ostatnim wynikiem jest pierwiastek z liczby 2, a w przykładzie *c* — z liczby 1,9.

**Skąd ta różnica?** W przykładzie *b* pętla została wykonana dokładnie 21 razy (zmiana *i* od 0 do 20), a w przykładzie *c* zmienna *x* zwiększała się od 0 do 2 co 0,1. Dla człowieka (liczącego w układzie dziesiętkowym) wszystko jest w porządku. Komputer wykonuje obliczenia na liczbach binarnych, a w tym systemie wartość ułamka 0,1 (jedna dziesiąta) jest ułamkiem okresowym, dodawanie odbywa się na wartościach przybliżonych i błąd się zwiększa. Przekonajmy się o tym, testując realizację następującego kodu (FTP: *p4\_2d.c* lub *p4\_2d.cpp*):

```
C, C++
for (x = 0; x <= 2; x += 0.1)
    printf("%21.18lf%12.8lf\n", x, sqrt(x));
printf("%21.18lf\n", x);
```

W relacji porównania  $x \leq 2$  o przekroczeniu przez zmienną *x* wartości 2 zadecydowała cyfra na 16. miejscu po przecinku. O takich pułapkach należy pamiętać. Przy porównywaniu wartości zmiennoprzecinkowych należy przewidzieć możliwe odchylenie wyniku od wartości dokładnej (oczekiwanej przez nas) i odpowiednio zmodyfikować warunek (FTP: *p4\_2e.c*, *p4\_2e.cpp*).

```
C, C++
double x;
for (x = 0; x < 2.01; x += 0.1)
    printf("%5.11f%12.8lf\n", x, sqrt(x));
```

### Przykład 4.3.

Sporządźmy tabliczkę mnożenia, która może się stać wzorem do budowy tablic wielu różnych funkcji. Zastosujemy w tym celu kilka instrukcji cyklu, dzieląc pracę na etapy:

- wydrukowanie wiersza nagłówkowego — znak działania  $\times$  i czynniki od 1 do 10; razem 11 kolumn o szerokości 5 znaków;

```
Pascal
write(' x ');
for i := 1 to 10 do write(i:5);
writeln;
```

<b>C, C++</b>	<pre>printf(" x "); for(i = 1; i &lt;= 10; ++i)     printf("%5d", i); printf("\n");</pre>
<b>C++</b>	<pre>cout &lt;&lt; " x "; for(int i = 1; i &lt;= 10; ++i)     cout &lt;&lt; setw(5) &lt;&lt; i; cout &lt;&lt; endl;</pre>

- wydrukowanie 10 wierszy tabelki dla czynników od 1 do 10 (pętla zewnętrzna) zawierających czynnik i 10 iloczynów (pętla wewnętrzna — zagnieżdżona).

<b>Pascal</b>	<pre>for j := 1 to 10 do begin write(j:3, ' '); for i := 1 to 10 do write(j*i:5); writeln; end;</pre>
<b>C, C++</b>	<pre>for(j = 1; j &lt;= 10; ++j) { printf("%3d ", j); for(i = 1; i &lt;= 10; ++i) printf("%5d", j*i); printf("\n"); }</pre>
<b>C++</b>	<pre>for(j = 1; j &lt;= 10; ++j) { cout &lt;&lt; setw(3) &lt;&lt; j &lt;&lt; " "; for(i = 1; i &lt;= 10; ++i) cout &lt;&lt; setw(5) &lt;&lt; j*i; cout &lt;&lt; endl; }</pre>

Pisząc programy samodzielnie, pamiętajmy o zadeklarowaniu potrzebnych zmiennych (dotyczy wszystkich języków) i włączeniu niezbędnych plików nagłówkowych. Pełne kody programów znajdują się w plikach na FTP: *p4\_3.pas*, *p4\_3.c* i *p4\_3.cpp*.

#### Przykład 4.4.

Dostosujemy programy z przykładu 4.3 do wyświetlania pierwiastków kwadratowych z liczb całkowitych w zakresie od 0 do 99. Wiersze tabeli będą zawierały po 10 wartości z zakresu: od 0 do 9, od 10 do 19 itd. Wiersze będą opisane pełnymi dziesiątkami, kolumny jednostkami liczby podpierwiastkowej. Wyniki (pierwiastki kwadratowe) zaokrąglone do 4 miejsc po przecinku będą umieszczane na przecięciu się wiersza dziesiątek z kolumną jednościami (co jest częstą praktyką w bardziej obszernych tablicach wartości funkcji).

Wiersz nagłówkowy:

<b>Pascal</b>	<pre>write(' sqrt'); for i := 0 to 9 do write(i:7); writeln;</pre>
<b>C, C++</b>	<pre>printf(" sqrt "); for(i = 0; i &lt;= 9; ++i)     printf("%7d", i); printf("\n");</pre>

Wiersze tabeli:

<b>Pascal</b>	<pre>for j := 0 to 9 do begin write(10*j:5, ' '); for i := 0 to 9 do write(sqrt(10*j+i):7:4); writeln; end;</pre>
<b>C, C++</b>	<pre>cout &lt;&lt; setprecision(4); cout.setf(ios::fixed); for(j = 0; j &lt;= 9; ++j) { printf("%5d ", 10*j); for(i = 0; i &lt;= 9; ++i) printf("%7.4lf", sqrt(10*j+i)); printf("\n"); }</pre>
<b>C++</b>	<pre>cout &lt;&lt; setprecision(4); cout.setf(ios::fixed); for(j = 0; j &lt;= 9; ++j) { cout &lt;&lt; setw(5) &lt;&lt; 10*j &lt;&lt; " "; for(i = 0; i &lt;= 9; ++i) cout &lt;&lt; setw(7) &lt;&lt; sqrt(10*j+i)); cout &lt;&lt; endl; }</pre>

Należy zwrócić uwagę na zastosowanie odstępów w używanych łańcuchach znaków, gdyż każdy z nich wpływa na format całej tabeli (FTP: *p4\_4.pas*, *p4\_4.c* i *p4\_4.cpp*). Przypomnijmy: użycie funkcji `sqrt` wymaga dodania pliku nagłówkowego *math.h* (C) i *cmath* (C++).

### Przykład 4.5.

Od *a* do *z* i z powrotem — na przykładzie wyświetlania znaków alfabetu pokazemy *odliczanie w górę* (z inkrementacją zmiennej sterującej) i *odliczanie w dół* (z dekrementacją zmiennej

sterującej). W Pascalu dysponujemy typem znakowym `char` (znaki o kodach od 0 do 255). Z typem tym związane są dwie funkcje:

- `chr(x)` — argument `x` typu `byte` jest kodem ASCII, a wynikiem jest odpowiadający mu znak `chr(65) = 'A'`;
- `ord(z)` — argument `z` jest znakiem (typu `char`), a wynikiem jest jego numer porządkowy (kod) w tabeli znaków ASCII, np. `ord('a') = 97`.

Jak widać, istnieje powiązanie pomiędzy typem znakowym `char` i typem liczbowym `byte` (jednobajtowe liczby całkowite bez znaku).

Nieco inaczej wygląda to w językach C i C++. Typ `char` jest typem liczbowym — są to liczby jednobajtowe ze znakiem i na elementach tego typu można wykonywać działania arytmetyczne (w zakresie od `-128` do `127`). Podczas wyświetlania znaków przy użyciu funkcji `printf` należy używać specyfikatora `%c`, natomiast przy wstawianiu znaku do strumienia może być potrzebne rzutowanie na typ `char`.

Wyświetlmy znaki alfabetu od `a` do `z`:

<b>Pascal</b>	<code>var znak: char; {deklaracja zmiennej}</code>
	<code>for znak := 'a' to 'z' do   write(znak);</code>
<b>C, C++</b>	<code>char znak; /* deklaracja zmiennej */ for(znak = 'a'; znak &lt;= 'z'; ++znak)   printf("%c", znak);</code>
<b>C++</b>	<code>for(char znak = 'a'; znak &lt;= 'z'; ++znak)   cout &lt;&lt; znak;</code>

oraz w drugą stronę — od `z` do `a`<sup>3</sup> (FTP: `p4_5.pas`, `p4_5.c` i `p4_5.cpp`):

<b>Pascal</b>	<code>for znak := 'z' downto 'a' do   write(znak);</code>
<b>C, C++</b>	<code>for(znak = 'z'; znak &gt;= 'a'; --znak)   printf("%c", znak);</code>
<b>C++</b>	<code>for(char znak = 'z'; znak &gt;= 'a'; --znak)   cout &lt;&lt; znak;</code>

Analogiczne efekty można uzyskać, stosując zmienną sterującą typu całkowitego i konwersję kodu ASCII na znak (FTP: `p4_5a.pas`, `p4_5a.c` i `p4_5a.cpp`).

<b>Pascal</b>	<code>var z: byte; {deklaracja zmiennej}</code>
	<code>for z := 97 to 122 do   write(chr(z));</code>

<sup>3</sup> Korzystamy z tej samej zmiennej `znak`, już wcześniej (Pascal, C). W C++ zmienną możemy każdorazowo deklorować jako zmienną lokalną w instrukcji `for`.

<b>C, C++</b>	<pre>int z; for(z = 97; z &lt;= 122; ++z)     printf("%c", z); printf("\n"); for(z = 122; z &gt;= 97; --z)     printf("%c", z);</pre>
<b>C++</b>	<pre>for(int z = 97; z &lt;= 122; ++z)     cout &lt;&lt; (char) z; cout &lt;&lt; endl; for(int z = 122; z &gt;= 97; --z)     cout &lt;&lt; (char) z;</pre>

W przykładach w C/C++ celowo użyto czterobajtowej liczby typu `int`, by podkreślić liczbowy charakter zmiennej sterującej. Można było użyć zmiennej typu `char`:

```
for(char z = 97; z <= 122; ++z) cout << z;
```

— nie jest w tym przypadku potrzebne rzutowanie na typ `char` przy wstawianiu wartości do strumienia `cout`).

### Przykład 4.6.

Kontynuując temat wyświetlania znaków, napiszmy program wyświetlający alfabet w postaci: AaBbCc... Zz. W jednym przebiegu pętli należałoby wyświetlić dwa znaki — wielką i małą literę. Wykorzystajmy fakt, że różnica pomiędzy kodem wielkiej i odpowiadającej jej małej litery wynosi 32 (FTP: *p4\_6.pas*, *p4\_6.c* i *p4\_6.cpp*).

<b>Pascal</b>	<pre>var z: byte; for znak := 'A' to 'Z' do     write(znak, chr(ord(znak) + 32));</pre>
<b>C, C++</b>	<pre>char znak; for(znak = 'A'; znak &lt;= 'Z'; ++znak)     printf("%c%c", znak, znak + 32);</pre>
<b>C++</b>	<pre>for(char znak = 'A'; znak &lt;= 'Z'; ++znak)     cout &lt;&lt; znak &lt;&lt; (char)(znak + 32);</pre>

## Pętle ze sprawdzaniem warunku na końcu

Ten rodzaj pętli poznaliśmy już w rozdziale 2. (przykłady 2.6, 2.7 i 2.12) i stosowaliśmy ją do kontroli poprawności wprowadzanych danych. Cechą charakterystyczną pętli ze sprawdzaniem warunku na końcu jest to, że instrukcja wykona się co najmniej raz (np. wprowadzamy dane, sprawdzamy: dane poprawne — idziemy dalej, dane błędne — powracamy do wprowadzania danych).

Przypomnijmy: w Pascalu (składnia: `repeat instrukcja until warunek;`) wyjście z pętli następuje, gdy warunek jest spełniony, a w języku C lub C++ — przeciwnie, wychodzimy z pętli, gdy warunek nie jest spełniony (składnia: `do instrukcja while warunek;`).

**Przykład 4.7.**

Użytkownik wprowadza z klawiatury pewną liczbę (z góry nieznaną) liczb dodatnich. Sygnałem zakończenia operacji jest wprowadzenie zera (zero do tego ciągu liczb już nie należy). Obliczymy sumę i średnią arytmetyczną tych liczb. Podczas wprowadzania danych (w pętli) będziemy wykonywać sumowanie liczb (zmienna suma) i zliczać ich ile ich jest (zmienna licznik). Na koniec pozostanie obliczenie średniej i wyświetlenie wyniku. Każdą liczbę wprowadzimy oddzielnie z klawiatury (po wpisaniu liczby naciśniemy *Enter*).

<b>Pascal</b>	<pre> var x, suma: real; licznik: integer; {Deklaracja zmiennych}  suma := 0; {Początkowa wartość sumy} licznik := -1; {Początkowa wartość licznika} repeat   write('x = ');   readln(x);   suma := suma+x; {Dodanie składnika do sumy}   licznik := licznik+1; {Zwiększenie licznika} until x = 0; writeln('Suma liczb: ', suma:0:5); if licznik &gt; 0 then   writeln('Średnia arytmetyczna: ', suma/licznik:0:5); </pre>
<b>C, C++</b>	<pre> double x, suma = 0; int licznik = -1; do {   printf("x = ");   scanf("%lf", &amp;x);   suma += x;   ++licznik; } while (x != 0); printf("Suma liczb: %0.5lf\n", suma); if (licznik &gt; 0)   printf("Średnia arytmetyczna: %0.5lf\n", suma/licznik); </pre>
<b>C++</b>	<pre> double x, suma = 0; int licznik = -1; do {   cout &lt;&lt; "x = ";   cin &gt;&gt; x;   suma += x;   ++licznik; } while (x != 0); cout &lt;&lt; "Suma liczb: " &lt;&lt; suma &lt;&lt; endl; if (licznik &gt; 0)   cout &lt;&lt; "Średnia arytmetyczna: " &lt;&lt; suma/licznik   &lt;&lt;&lt;endl; </pre>

Ustawienie początkowej wartości licznika równej  $-1$  jest spowodowane tym, że po wprowadzeniu liczby  $0$  licznik jest zwiększany o  $1$ , a ta liczba do ciągu nie należy (dodajemy ją co prawda do sumy, ale — jak wiemy — nie wpłynie to na wartość końcową tej sumy). Unikamy w ten sposób instrukcji warunkowych wewnątrz pętli lub korygowania końcowej wartości licznika (FTP: `4_7.pas`, `4_7.c` i `4_7.cpp`).

### Przykład 4.8.

Mamy dwie różne liczby naturalne dodatnie  $m$  i  $n$ . Będziemy obliczali (w pewnym sensie naprzemiennie) ich wielokrotności, aż do uzyskania równych liczb, czyli ich wspólnej wielokrotności. Zaczynamy od liczby mniejszej, liczymy jej kolejne wielokrotności, jeżeli zostanie przekroczona aktualna wielokrotność drugiej liczby, to zaczynamy obliczać wielokrotności drugiej liczby... Postępowanie kontynuujemy aż do chwili, gdy wielokrotności obu liczb zrównają się (co musi kiedyś nastąpić, gdyż wspólną wielokrotnością tych liczb jest ich iloczyn — może jednak znajdziemy mniejszą wartość). Wyznaczona w ten sposób wspólna wielokrotność jest najmniejszą wspólną wielokrotnością liczb  $m$  i  $n$  — oznaczamy ją symbolem  $NWW(m, n)$ .

<b>Pascal</b>	<pre>wm := m; {wielokrotność liczby m} wn := n; {wielokrotność liczby n} repeat   if wm &lt; wn then wm := wm+m; {kolejna wielokrotność m}   if wn &lt; wm then wn := wn+n; {kolejna wielokrotność n} until wm = wn; writeln('NWW(', m, ', ', m, ') = ', wm);</pre>
<b>C, C++</b>	<pre>wm = m; /* Wielokrotność liczby m */ wn = n; /* Wielokrotność liczby n */ do {   if (wm &lt; wn)     wm += m; /* Kolejna wielokrotność m */   if (wn &lt; wm)     wn += n; /* Kolejna wielokrotność n */ } while(wm != wn); printf("NWW(%d, %d) = %d\n", m, n, wm);</pre>
<b>C++</b>	<pre>/* Jak wyżej */ cout &lt;&lt; "NWW(" &lt;&lt; m &lt;&lt; ", " &lt;&lt; n &lt;&lt; ") = " &lt;&lt; wm &lt;&lt; endl;</pre>

Również podczas wprowadzania danych z klawiatury możemy kontrolować, czy użytkownik podaje liczby dodatnie.



Pascal	<pre>repeat   write('m = ');   readln(m) until m &gt; 0; repeat   write('n = ');   readln(n) until n &gt; 0;</pre>
C, C++	<pre>do {   printf("m = ");   scanf("%d", &amp;m); } while (m &lt;= 0); do {   printf("n = ");   scanf("%d", &amp;n); } while (n &lt;= 0);</pre>
C++	<pre>do {   cout &lt;&lt; "m = ";   cin &gt;&gt; m; } while (m &lt;= 0); do {   cout &lt;&lt; "n = ";   cin &gt;&gt; n; } while (n &lt;= 0);</pre>

Z tego typu pętlami spotkamy się jeszcze wielokrotnie. Kompletne kody programów umieszczono na FTP: *p4\_8.pas*, *p4\_8.c* i *p4\_8.cpp*.

## Pętla ze sprawdzaniem warunku na początku

### Przykład 4.9.

Praktycznym i szybkim sposobem obliczania największego wspólnego dzielnika dwóch liczb całkowitych (nieujemnych, z których co najmniej jedna jest różna od zera) jest algorytm Euklidesa. Jest to jeden z najstarszych algorytmów — został opisany przez Euklidesa ok. roku 300 p.n.e. Opiera się on na spostrzeżeniu, że jeśli od większej liczby odejmiemy mniejszą, to mniejsza liczba i otrzymana różnica będą miały największy wspólny dzielnik taki sam jak pierwotne liczby. Jeśli w wyniku kolejnego odejmowania otrzymamy parę równych liczb, oznacza to, że znaleźliśmy ich największy wspólny dzielnik.

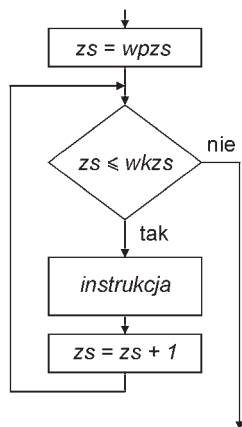
W czasie wykonywania operacji odejmowania zmieniają się wartości liczb, więc bezpiecznie będzie te działania wykonywać na kopiach liczb (zachowując oryginały do dalszych działań).

Dopóki liczby (kopie) są różne, od większej odejmujemy mniejszą (FTP: *p4\_9.pas*, *p4\_9.c* i *p4\_9.cpp*).

<b>Pascal</b>	<pre> km := m; {Kopia liczby m} kn := n; {Kopia liczby n} while (km &lt;&gt; kn) do   if km &gt; kn then km := km-kn else kn := kn-km; writeln('NWD(', m, ', ', n, ') = ', km); </pre>
<b>C, C++</b>	<pre> km = m; /* Kopia liczby m */ kn = n; /* Kopia liczby n */ while(km != kn)   if (km &gt; kn) km -= kn; else kn -= km; printf("NWD(%d, %d) = %d\n", m, n, km); </pre>
<b>C++</b>	<pre> /* Jak wyżej */ cout &lt;&lt; "NWD(" &lt;&lt; m &lt;&lt; ", " &lt;&lt; n &lt;&lt; ") = " &lt;&lt; km &lt;&lt; endl; </pre>

Zastosowana tu pętla ze sprawdzaniem warunku na początku różni się składnią (różnica pomiędzy Pascalą i C lub C++), natomiast sposób interpretowania warunku pozostaje we wszystkich językach taki sam — dopóki warunek jest spełniony, wykonywana jest instrukcja. Jeśli instrukcja jest złożona (a tak najczęściej bywa), to musimy pamiętać o nawiasach (begin end w Pascalu i { } w C i C++).

Pętla ze sprawdzaniem warunku na początku może nie wykonać się wcale, gdy podczas pierwszego sprawdzania warunek będzie fałszywy. Należy pamiętać o tym, żeby instrukcja zawarta w pętli modyfikowała zmienne mające wpływ na ocenę logiczną warunku, gdyż w przeciwnym razie nie będzie możliwości zakończenia cyklu (program „zawiesi się”).



Rysunek 4.1. Pętla ze zmienną sterującą — odliczanie w górę

## Która pętla lepsza, czyli krótkie porównanie instrukcji

Sytuację przeanalizujemy w kilku charakterystycznych fragmentach algorytmów przedstawionych w postaci schematu blokowego i realizujących pętlę. Rozpocznijmy od pętli, w których *zmienna sterująca* spełnia rolę licznika powtórzeń.

Na rysunku 4.1 przedstawiono pętlę realizującą następujący algorytm:

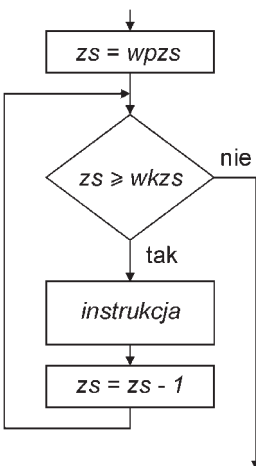
1. Zmienna *zs* (*zmienna sterująca*) przyjmuje pewną wartość *wpzs* (*wartość początkowa zmiennej sterującej*).
2. Badany jest warunek: czy wartość zmiennej *zs* jest mniejsza lub równa pewnej wartości *wkzs* (*wartość*

końcowa zmiennej sterującej). Jeśli warunek nie jest spełniony, przechodzimy do kolejnej instrukcji po instrukcji cyklu.

3. Wykonywana jest instrukcja.
4. Wartość zmiennej sterującej jest zwiększana o 1.
5. Przechodzimy do punktu 2.

Algorytmowi temu odpowiada idealnie klasyczna pętla for (odliczanie w górę — od mniejszej wartości do większej). Łatwo to również zapisać przy użyciu pętli ze sprawdzaniem warunku na początku.

Pascal	for zs:= wpzs to wkzs do instrukcja;
	<pre> zs := wpzs; {Wartość początkowa zmiennej sterującej} while wpzs &lt;= wkzs do   begin     instrukcja;     zs := zs+1;   end; </pre>
C, C++	for(zs = wpzs; zs <= wkzs; ++i) instrukcja;
	<pre> zs = wpzs; /* Wartość początkowa zmiennej sterującej */ while (zs &lt;= wkzs) {   instrukcja;   ++zs }; </pre>
	<pre> zs = 1; /* Wartość początkowa zmiennej sterującej */ while (zs++ &lt;= wkzs)   instrukcja; </pre>



Drugi z podanych wariantów (dla C i C++) jest do przyjęcia tylko wtedy, gdy instrukcja nie korzysta z wartości zmiennej zs (wewnątrz pętli zmienna zs będzie miała wartość o 1 większą niż podczas badania warunku — efekt działania preinkrementacji).

Podobne konstrukcje możemy zbudować dla zmniejszającej się wartości zmiennej sterującej (odliczanie w dół). Odpowiedni schemat blokowy przedstawiono na rysunku 4.2.

Algorytm przedstawiony na schemacie blokowym (rysunek 4.2) można zakodować w następujący sposób:

Rysunek 4.2. Pętla ze zmienną sterującą — odliczanie w dół

<b>Pascal</b>	<code>for zs:= wpzs downto wkzs do instrukcja;</code>
	<code>zs := wpzs; {Wartość początkowa zmiennej sterującej} while zs &gt;= wkzs do begin instrukcja; zs := zs-1; end;</code>
<b>C, C++</b>	<code>for(zs = wpzs; zs &gt;= wkzs; ++zs) instrukcja;</code>
	<code>zs = wpzs; /* Wartość początkowa zmiennej sterującej */ while (zs &gt;= wkzs) { instrukcja; --i };</code>
	<code>zs = wpzs; /* Wartość początkowa zmiennej sterującej */ while (i-- = n) instrukcja;</code>

Należy pamiętać, że dla pętli `for` w Pascalu zmienna sterująca powinna być zmienną typu porządkowego (liczba całkowita lub znak). W C i C++ ta zasada nie musi być przestrzegana.

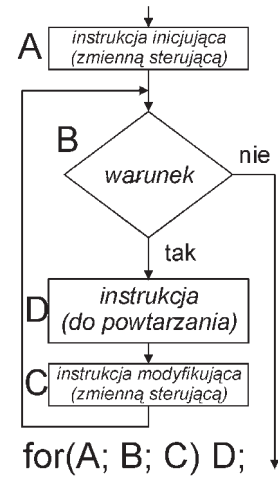
Każdą pętlę typu `for` możemy zastąpić pętlą ze sprawdzaniem warunku na końcu połączoną z instrukcją warunkową (pokażemy to tylko w przypadku zwiększania zmiennej sterującej).

<b>Pascal</b>	<code>for zs:= wpzs to wkzs do instrukcja;</code>
	<code>zs := wpzs; {Wartość początkowa zmiennej sterującej} if zs &lt;= wkzs then repeat instrukcja; zs := zs+1; until zs &gt; wkzs;</code>
<b>C, C++</b>	<code>for(zs = wpzs; zs &lt;= wkzs; ++zs) instrukcja;</code>
	<code>zs = wpzs; /* Wartość początkowa zmiennej sterującej */ if (zs &lt;= wkzs) do { instrukcja; ++zs; } while (zs &lt;= wkzs);</code>
	<code>zs = wpzs; /* Wartość początkowa zmiennej sterującej */ if (zs &lt;= wkzs) do { instrukcja; } while (++zs &lt;= wkzs);</code>

Drugi wariant pętli for (odliczanie w dół) można przedstawić podobnie. Zwróćmy uwagę na różnicę pomiędzy zapisaniem warunku w Pascalu i C lub C++.

Odrębnego omówienia wymaga pętla for w C lub C++. W dotychczasowych przykładach dokonywaliśmy zapisywania w C i C++ pętli for w stylu Pascala. W tych językach pętla for oferuje nam jednak szereg innych, niespotykanych w Pascalu możliwości. Oznaczone na schemacie (rysunek 4.3.) elementy **A**, **C** i **D** mogą być dowolnymi instrukcjami, a warunek **B** może być dowolnym wyrażeniem arytmetycznym (wartość 0 interpretowana jest jako fałsz, a różna od zera jako prawda).

Instrukcja **A** wykonywana jest tylko raz, na początku. Następnie badany jest warunek **B** — wartość 0 powoduje zakończenie cyklu, wartość różna od 0 wywołuje wykonanie instrukcji **D**, później **C** i powrót do sprawdzania warunku **B**. W zapisie instrukcji (z użyciem symboli) `for(A; B; C) D;` możemy pominąć (oczywiście licząc się z konsekwencjami) dowolny z elementów, a nawet wszystkie cztery. Oczywiście instrukcja w postaci `for(;;);` skompiluje się, po uruchomieniu nie będzie niczego widocznego robić, ale też trudno będzie przerwać jej działanie.



Rysunek 4.3. Schemat działania instrukcji cyklu typu for w językach C i C++

## Przerywanie działania pętli

Pętle wykonują się z góry określoną liczbą razy albo wykonują się, gdy jakiś warunek jest spełniony, albo do czasu spełnienia (niespełnienia) określonego warunku. Są jednak sytuacje, w których wypadałoby przerwać działanie pętli podczas jej pracy (gdzieś w środku między kolejnymi instrukcjami), po spełnieniu (niespełnieniu) jakiegoś warunku.

### Przykład 4.10.

Przeanalizujmy działanie prostego programu. Przy użyciu pętli for wyświetlamy na ekranie liczby od 1 do 10. Przy osiągnięciu pewnej wartości (np. 5) przerywamy proces wyświetlania. Program wyświetli na ekranie tylko liczby od 1 do 4 (w kolejnych wierszach:  $n = 1$ ,  $n = 2$ ,  $n = 3$  i  $n = 4$ ) oraz niedokończony piąty wiersz ( $n =$ ), a następnie komunikat: *Pętla została przerwana dla  $n = 5$*  (FTP: `p4_10.pas`, `p4_10.c` i `p4_10.cpp`).

Pascal

```

for n:= 1 to 10 do
  begin
    write('n = ');
    if n = 5 then exit;
    writeln(n);
  end;
writeln('Pętla została przerwana dla n = 5');
  
```

<b>C, C++</b>	<pre> for(n = 1; n &lt;= 10; ++n) {     printf("n = ");     if (n == 5)         break;     printf("%d\n", n); } printf("Pętla została przerwana dla n = 5\n"); </pre>
<b>C++</b>	<pre> for(int n = 1; n &lt;= 10; ++n) {     cout &lt;&lt; "n = ";     if (n == 5)         break;     cout &lt;&lt; n &lt;&lt; endl; } cout &lt;&lt; "Pętla została przerwana dla n = 5\n"; </pre>

Funkcja `break` w C i C++ działa zgodnie z naszymi oczekiwaniami. Użycie do tego celu w Pascalu instrukcji `exit` nie jest dobrym pomysłem, gdyż ta instrukcja realizuje wyjście z bieżącego bloku programu lub podprogramu (nie dotyczy bloku instrukcji złożonej) i w tym przypadku zamknie nasz program — nie zobaczymy już komunikatu *Pętla została przerwana dla n = 5*.

Nie mamy jednak w Pascalu odpowiednika funkcji `break` z języka C. Musimy wymyślić coś innego.

Jednym wyjściem jest zbudowanie bezparametrowej procedury `loop` (rodzaj „opakowania”), w której ciele umieścimy naszą pętlę `for`. Użycie `exit` spowoduje opuszczenie bloku procedury `loop` i powrót do naszego programu.

<b>Pascal</b>	<pre> procedure loop;     var n: byte; begin     for n:= 1 to 10 do         begin             write('n = ');             if n = 5 then exit;             writeln(n);         end; end; begin     loop; {Procedura „opakowująca” naszą pętlę}     writeln('Pętla została przerwana dla n = 5'); end. </pre>
---------------	--

Kod umieszczono na FTP: [p4\\_10a.pas](#), [p4\\_10a.c](#) i [p4\\_10a.cpp](#). W C i C++ zamiast instrukcji `exit` wykorzystano `return`. Takich kombinacji jednak nie polecamy, skoro można zrobić to łatwiej przy użyciu `break`, co też właściwie nie jest zalecane. Pokazaliśmy jednak możliwość przeniesienia pewnego pomysłu dostępnego w C do Pascala, a później odwrotnie — z Pascala do C.

Inną możliwością jest ingerencja w wartość zmiennej sterującej i warunkowe jej ustawienie na wartość końcową. W **Turbo Pascalu** program kompiluje się i działa poprawnie. Kompilator **FPC** tego przykładu nie skompiluje (FTP: *p4\_10b.pas*, *p4\_10b.c* i *p4\_10b.cpp*).

<b>Pascal</b>	<pre> var n: byte; begin   for n:= 1 to 10 do     begin       write('n = ');       if n = 5 then n := 10         else writeln(n);     end;   writeln('Pętla została przerwana dla n = 5');   readln; end. </pre>
---------------	--

Zamiast instrukcji `for` możemy użyć innej instrukcji cyklu, która ją zastąpi. Wtedy problemu nie będzie (FTP: *p4\_10c.pas*, *p4\_10c.c* i *p4\_10c.cpp*). Podobnie można postąpić z pętlą ze sprawdzaniem warunku na początku — instrukcja `break`; (C, C++) działa tak samo, niezależnie od typu przerywanej pętli. Nasze „sztuczki” ze zmienną sterującą, związane z przerywaniem pętli, również są skuteczne (FTP: *p4\_10d.pas*, *p4\_10d.c* i *p4\_10d.cpp*).

<b>Pascal</b>	<pre> n:= 1; repeat   write('n = ');   if n = 5 then n := 10     else writeln(n);   n := n+1; until n &gt; 10; writeln('Pętla została przerwana dla n = 5'); </pre>
	<pre> n:= 1; while n &lt;= 10 do begin   write('n = ');   if n = 5 then n := 10     else writeln(n);   n := n+1; end; writeln('Pętla została przerwana dla n = 5'); </pre>

Analogiczne rozwiązania w C i C++ Czytelnik może sobie zbudować samodzielnie (ale nie ma takiej konieczności — instrukcja `break` usuwa problem). Mimo to na serwerze FTP umieszczono wszystkie sygnalizowane rozwiązania zapisane w trzech językach.

Inną przydatną czynnością może być warunkowe pomijanie fragmentu kodu wewnątrz bloku pętli. Oczywiście możliwe jest dokonanie tego przy użyciu instrukcji warunkowych, ale czasem

wygodne może być posłużenie się instrukcją `continue` (C lub C++ — w Pascalu takiej instrukcji nie znajdziemy).

### Przykład 4.11.

Użytkownik wprowadza z klawiatury ciąg liczb całkowitych. Program sumuje wyłącznie liczby dodatnie i kończy obliczenia, gdy suma osiągnie lub przekroczy 100 (FTP: *4\_11.pas*, *4\_11.c*, *4\_11a.c*, *4\_11.cpp* i *4\_11a.cpp*).

<b>Pascal</b>	<pre> suma := 0; repeat   write('n = ');   readln(n);   if n &gt; 0 then     begin {Dodawanie wyłącznie liczb dodatnich}       suma := suma+n;       writeln('S = ', suma);     end; until suma &gt;= 100; </pre>
<b>C, C++</b>	<pre> suma = 0; do {   printf("n = ");   scanf("%d", &amp;n);   if (n &gt; 0) { /* Dodawanie wyłącznie liczb dodatnich */     suma += n;     print("S = %d\n", suma);   } } while (suma &lt; 100); </pre>
	<pre> suma = 0; do {   printf("n = ");   scanf("%d", &amp;n);   if (n &lt;= 0)continue;   /* continue – pominięcie dalszych instrukcji w pętli, dodawanie wyłącznie   ↪liczb dodatnich */   suma += n;   print("S = %d\n", suma); } while (suma &lt; 100); </pre>
<b>C++</b>	<pre> <i>Jw., wiersze:</i> printf("n = "); scanf("%d", &amp;n); <i>można zastąpić wierszami:</i> cout &lt;&lt; "n = "; cin &gt;&gt; n; <i>Podobnie wiersz:</i> print("S = %d\n", suma); <i>zastąpimy wierszem:</i> cout &lt;&lt; "S = " &lt;&lt; suma &lt;&lt; endl; </pre>

Funkcja `continue` powoduje pominięcie kolejnych instrukcji i przejście do miejsca w pętli, w którym badany jest warunek. Dotyczy to pętli wszystkich typów w językach C i C++.



# BUDUJEMY WŁASNE FUNKCJE I PROCEDURY



## Zmienne globalne i lokalne

W Pascalu deklarujemy potrzebne zmienne przed głównym blokiem programu i ich zasięg jest globalny (możemy z nich korzystać w dowolnym miejscu programu). Budując własne procedury lub funkcje, możemy deklarować w nich zmienne, których zasięg będzie lokalny — w obrębie głównego bloku procedury lub funkcji.

Zadaniem procedury jest realizacja jakiegoś fragmentu algorytmu (programu), szczególnie wtedy, gdy ta czynność jest wielokrotnie wykonywana w programie. Może tutaj wystąpić kilka istotnie różnych sytuacji (co zilustrujemy standardowymi procedurami):

- procedura wykonuje czynność, która nie wymaga żadnych dodatkowych informacji (parametrów) wpływających na sposób jej wykonania, np. `writeln` — przeniesie kursor na początek nowego wiersza ekranu;
- procedura wykonuje czynność w sposób zależny od przekazanego parametru (lub parametrów), np. `write(5)` — wyświetla na ekranie liczbę 5, `write(x)` — wyświetla na ekranie wartość zmiennej `x`, nie wpływa jednak w żaden sposób na wartość zmiennej;
- procedura zmienia wartość wskazanej przez parametr zmiennej (globalnej lub lokalnej), np. `readln(x)` — odczytuje wartość wprowadzoną przez użytkownika z klawiatury i przypisuje ją do zmiennej `x`; `dec(n)` — zmniejszenie wartości zmiennej `n` (typu porządkowego) o jeden „krok”<sup>1</sup>;
- procedura działa niezależnie od wartości pewnej zmiennej globalnej — brak standardowego przykładu jest chyba zrozumiałe, rozwiązanie nie jest zbyt eleganckie, ale czasem można z niego skorzystać;
- procedura zmienia wartość pewnej zmiennej globalnej (niewskazanej jako parametr) — tego raczej bym nie polecał (popołniony błąd może być trudny do odnalezienia).

Funkcja może działać w sposób podobny do procedury (w zakresie przekazywania parametrów, działania na zmiennych...), lecz jej podstawowym zadaniem jest zwrócenie obliczonej wartości do wyrażenia, w którym została wywołana, np. wyrażenie `n := pred(n)` odpowiada wywołaniu procedury `dec(n)` dla zmiennej typu porządkowego; `y := 2 * sin(x) - 1` nie wymaga chyba komentarza.

<sup>1</sup> Dla liczb całkowitych odpowiada to podstawieniu `n := n - 1`, natomiast dla znaków (typ `char`) po wykonaniu sekwencji instrukcji `znak := 'd'`; `dec(znak)`; zmienna `znak` ma wartość `'c'` (znak o kodzie o 1 mniejszym).

Parametry formalne procedury lub funkcji są dla tej funkcji zmiennymi lokalnymi, z wyłączeniem sytuacji, gdy do funkcji przekazywany jest adres zmiennej i procedura (funkcja) działa bezpośrednio na wskazanej zmiennej. Jeśli nazwa zmiennej lokalnej jest identyczna z nazwą zmiennej globalnej, to na czas działania procedury (funkcji) występuje tzw. przysłanianie zmiennej. Wszystko to omówimy wkrótce na przykładach. Należy dodać, że procedury i funkcje w Pascalu mogą mieć zadeklarowane własne stałe, zmienne, procedury lub funkcje o charakterze dla nich lokalnym, a poza nimi zupełnie niedostępne.

Nieco odmienna sytuacja występuje w języku C lub C++. Zmienne lokalne możemy definiować w obrębie każdego bloku (instrukcji złożonej) i ich zasięg nie wychodzi poza parę nawiasów {...} ograniczających ten blok. Przy czym w języku C zmienne deklarujemy na początku bloku, a w C++ można to zrobić w dowolnym momencie — wtedy, kiedy chcemy użyć danej zmiennej. Ponadto nie rozróżniamy pojęcia procedury i funkcji — w C i C++ stosujemy wyłącznie funkcje, jednak dla tych, które nie zwracają wartości, mamy zdefiniowany specjalny pusty typ `void`. Przekazane wcześniej uwagi o procedurach pozostają aktualne (z wyjątkiem przykładów) dla funkcji w języku C (C++). W kolejnych przykładach postaramy się przedstawić szczegóły.

## Przekazywanie danych do procedur i funkcji, zwracanie wyników

Zacznijmy od przedstawienia ogólnej budowy procedury i funkcji. Użyte nazwy (*nazwa\_procedury*, *nazwa\_funkcji*, *lista\_parametrów*, *typ\_wartości*) i komentarze jasno opisują miejsce i znaczenie poszczególnych elementów. Resztę wyjaśnią kolejne przykłady.

	<p>Procedura:</p> <pre>procedure <i>nazwa_procedury</i>(<i>lista_parametrów</i>);     {Deklaracje lokalnych stałych, zmiennych, procedur lub funkcji} begin     {Instrukcje – ciało procedury} end;</pre>
Pascal	<p>Funkcja:</p> <pre>function <i>nazwa_funkcji</i>(<i>lista_parametrów</i>): <i>typ_wartości</i>;     {Deklaracje lokalnych stałych, zmiennych, procedur lub funkcji} begin     {Instrukcje – ciało procedury}     {Wynik obliczeń jest na koniec przypisany do nazwy funkcji}     <i>nazwa_funkcji</i> := <i>wyrażenie</i>; end;</pre>

C, C++	<p>Funkcja niezwracająca wyniku — odpowiednik procedury w Pascalu:</p> <pre>void nazwa_funkcji(lista_parametrów) {     /* Deklaracja zmiennych lokalnych */     /* Instrukcje – ciało funkcji */     return; }</pre>
	<p>Funkcja:</p> <pre>typ_wyniku nazwa_funkcji(lista_parametrów) {     /* Deklaracja zmiennych lokalnych */     /* Instrukcje – ciało funkcji */     return wyrażenie; /* Zwrócenie obliczonego wyniku */ }</pre>

### Przykład 5.1.

Bezparametrowa procedura komunikat wyświetla na ekranie napis *To jest komunikat!*. Wywołanie procedury (funkcji wg terminologii C i C++) jest we wszystkich tych językach identyczne — `komunikat()`; — jedynie w Pascalu można opuścić nawiasy, gdy procedura nie posiada parametrów. W językach C i C++ nazwa funkcji bez nawiasów interpretowana jest jako adres tej funkcji w pamięci, a nie jej wywołanie (FTP: *p5\_1.pas*, *p5\_1.c* i *p5\_1.cpp*).

Pascal	<pre>procedure komunikat(); begin     writeln('To jest komunikat!'); end;</pre>
	Wywołanie procedury: <code>komunikat()</code> ; lub <code>komunikat</code> ;
C, C++	<pre>void komunikat(void) {     printf("To jest komunikat!\n"); }</pre>
	Wywołanie procedury: <code>komunikat()</code> ;
C++	<pre>void komunikat(void) {     cout &lt;&lt; "To jest komunikat!\n"; }</pre>
	Wywołanie procedury: <code>komunikat()</code> ;

**Przykład 5.2.**

Procedura `suma` z dwoma parametrami (liczby całkowite  $m$  i  $n$ ) wyświetla na ekranie sumę podanych liczb. Użyte w przykładzie zmienne  $a$ ,  $b$  i  $x$  są zadeklarowane jako zmienne typu całkowitego. Przekazywanie parametrów następuje przez wartość (stała liczbowa lub wartość zmiennej). Obliczony wynik pojawia się na ekranie i nie jest nigdzie zapamiętywany.

Zwróćmy uwagę na sposób zadeklarowania parametrów formalnych w nagłówkach. W Pascalu możemy to zrobić na dwa sposoby:

- `procedure suma(m, n: integer)` — lista zmiennych (bez słowa kluczowego `var`) oddzielonych przecinkami, dwukropek i typ zmiennej (zmienne są tego samego typu);
- `procedure suma(m: integer; n: integer)` — deklaracje pojedynczych zmiennych (bez słowa kluczowego `var`) oddzielone średnikami (zmienne mogą być różnych typów).

W języku C (C++) typ każdej zmiennej określamy osobno, a deklaracje poszczególnych parametrów oddzielamy przecinkami: `void suma(int m, int n)`.

<b>Pascal</b>	<pre>procedure suma(m, n: integer); begin   writeln(m+n); end;</pre>
	Wywołanie procedury: <code>suma(14, 29); suma(x, 13); suma(a, b);</code>
<b>C, C++</b>	<pre>void suma(int m, int n) {   printf("%d\n", m+n); }</pre>
	Wywołanie funkcji: <code>suma(14, 29); suma(x, 13); suma(a, b);</code>
<b>C++</b>	<pre>void suma(int m, int n) {   cout &lt;&lt; m+n &lt;&lt; endl; }</pre>
	Wywołanie funkcji: <code>suma(14, 29); suma(x, 13); suma(a, b);</code>

**Przykład 5.3.**

Dodawanie można wykonywać, korzystając z procedury `suma` z trzema parametrami — dwa z nich to wartości składników, trzeci parametr wskazuje nazwę (adres) zmiennej, do której ma być przekazany wynik dodawania. Tym razem parametry są liczbami zmiennoprzecinkowymi.

<b>Pascal</b>	<pre>procedure suma(x, y: real; var s: real); begin     s := x+y; end;</pre>
	<p>Wywołanie procedury:</p> <pre>suma(14, 2.9, z); suma(x, 13.1, a); suma(a, b, x);</pre>
<b>C, C++</b>	<pre>void suma(double x, double y, double* s) {     *s = x+y; }</pre>
	<p>Wywołanie funkcji (procedury):</p> <pre>suma(14, 2.9, &amp;z); suma(x, 13.1, &amp;a); suma(a, b, &amp;x);</pre>

W procedurze `suma` (Pascal) trzeci parametr `s` zadeklarowano przy użyciu słowa kluczowego `var` (`var s: real`). Oznacza to, że w chwili wywołania procedury nie jest tworzona zmienna lokalna `s` typu `real` z kopią wartości przekazanego parametru, ale do procedury przekazywany jest adres zmiennej (tzw. *przekazywanie przez referencję*), np. dla wywołania `suma(14, 2.9, z)` jest to adres zmiennej `z`. Wszelkie zmiany wartości zmiennej `s` wewnątrz procedury dotyczą w tym wywołaniu procedury zmiennej `z`.

Nieco inaczej rozwiązany jest problem w C lub C++:

`double* s` — w nagłówku zadeklarowano parametr `s` typu wskaźnik na zmienną typu `double`,  
`*s = ...` — w ciele funkcji przypisano zmiennej wskazywanej przez wskaźnik wartość wyrażenia ...,

`suma(14, 2.9, &z)` — w wywołaniu funkcji jako jeden z parametrów przekazano adres zmiennej `&z`.

Wskaźnik jest po prostu adresem (w pamięci komputera) zmiennej określonego typu. Jednoargumentowy operator `*` zwany operatorem wyłuskania określa wartość zmiennej wskazywanej przez wskaźnik (np. `*s = 2` — zmiennej wskazywanej przez wskaźnik `s` przypisano wartość 2; `y = 2*s` — zmiennej `y` przypisano wartość zmiennej wskazywanej przez wskaźnik `s` pomnożoną przez 2). Ta minimalna informacja o wskaźnikach powinna wystarczyć do zrozumienia tego i podobnych przykładów (FTP: [p5\\_3.pas](#), [p5\\_3.c](#), [p5\\_3.cpp](#)).

Rozwiązanie ze wskaźnikami w stylu języka C można również zrealizować w Pascalu. Wymaga to tylko zdefiniowania nazwy dodatkowego typu wskaźnikowego: `type preal = ^real;`

Pascal	<pre> <b>type preal = ^real;</b> {Typ wskazujący na real} procedure suma(x, y: real; <b>s: preal</b>); begin     <b>s^</b> := x+y; end;</pre>
	<p>Wywołanie procedury:</p> <pre> suma(14, 2.9, @z); suma(x, 13.1, @a); suma(a, b, @x);</pre>

Symbol `s^` oznacza wartość zmiennej wskazywanej przez wskaźnik `s`, a w wywołaniu procedury `@z`, `@a` i `@x` są adresami zmiennych `z`, `a` i `x` (`@` — operator adresu). Ten przykład (FTP: `p5_3a.pas`) potraktujmy jako ciekawostkę, gdyż działania na wskaźnikach zawsze niosą jakieś ryzyko zamieszania w pamięci. W Pascalu mamy wcześniej wspomnianą bezpieczną możliwość przekazywania parametrów przez referencję (tego nie ma w C — w nim konieczne jest użycie wskaźników).

W C++ wprowadzono pojęcie **referencji** i ten przykład można zapisać w następujący sposób:

C++	<pre> void suma(double x, double y, double&amp; s) {     s = x+y; }</pre>
	<p>Wywołanie funkcji (procedury):</p> <pre> suma(14, 2.9, z); suma(x, 13.1, a); suma(a, b, x);</pre>

Po zadeklarowaniu `double& s` parametr `s` jest *referencją (adresem)* zmiennej typu `double`. W chwili wywołania funkcji `suma(14, 2.9, z)` tworzona jest lokalnie referencja `s` zawierająca adres zmiennej `z`. Wszelkie operacje wykonywane na `s` dotyczą zmiennej `z` (FTP: `p5_3a.cpp`).

#### Przykład 5.4.

Funkcja `suma` z dwoma parametrami oblicza i zwraca wartość sumy podanych liczb. Wynik może być przypisany do zmiennej odpowiedniego typu lub stać się elementem innego wyrażenia (FTP: `p5_4.pas`, `p5_4.c` i `p5_4.cpp`).

Pascal	<pre> function suma(x, y: real): real; begin     suma := x+y; end;</pre>
	<p>Wywołanie funkcji:</p> <pre> z := suma(14, 2.9); a := suma(x, 13.1); x := 2*suma(a, b);</pre>
C, C++	<pre> double suma(double x, double y) {     return x+y; }</pre>
	<p>Wywołanie funkcji:</p> <pre> z = suma(14, 2.9); a = suma(x, 13.1); x = 2*suma(a, b);</pre>

W podanych przykładach pokazaliśmy dwa sposoby przekazywania danych do procedury lub funkcji (przez wartość i przez zmienną) oraz trzy sposoby przekazywania wyników: bezpośrednio na konsolę (ekran monitora), przez zmianę wartości wskazanej zmiennej, podstawienie wartości do zmiennej lub bezpośrednio do wyrażenia. Pominęliśmy niezalecane i nienależące do dobrego stylu programowania korzystanie wprost ze zmiennych globalnych.

Przytoczone przykłady były na tyle proste, że nie wymagały deklarowania i stosowania dodatkowych zmiennych lokalnych (w funkcji) do realizacji zadania.

### Przykład 5.5.

Napiszemy procedurę zamieniającą wartości dwóch zmiennych w pamięci komputera. W tym celu zastosujemy dodatkową zmienną pomocniczą i następujący schemat:

$x \rightarrow \text{pomocnik} \mid y \rightarrow x \mid \text{pomocnik} \rightarrow y$

Pascal	<pre> procedure zamiana(var x, y: real);   var pom: real; {Lokalna zmienna pomocnicza} begin   pom := x; {Zapamiętanie wartości x w zmiennej pom}   x := y; {Podstawienie y w miejsce x}   y := pom; {Podstawienie zapamiętanej wartości do y} end;</pre>
	Wywołanie procedury: <code>zamiana(a, b);</code>
C, C++	<pre> void zamiana(double * x, double * y) {   double pom; /* Lokalna zmienna pomocnicza */   pom = *x; /* Zapamiętanie wartości x w zmiennej pom */   *x = *y; /* Podstawienie y w miejsce x */   *y = pom; /* Podstawienie wartości pom w miejsce y */   return; }</pre>
	Wywołanie funkcji: <code>zamiana(&amp;a, &amp;b);</code>

Kompletne pliki umieszczono na FTP: `p5_5.pas`, `p5_5.c` i `p5_5.cpp`. Procedura zamiany wartości zmiennych przyda się np. podczas realizacji operacji sortowania danych liczbowych.

### Przykład 5.6.

Na podstawie przykładów 3.3 i 3.5 zbudujemy funkcję `rkw` rozwiązującą równanie kwadratowe. Funkcja ta zwróci wartość całkowitą określającą liczbę i „rodzaj pierwiastków” wg kodu: -2 — równanie nie jest równaniem kwadratowym ( $a = 0$ ); -1 — równanie ma dwa pierwiastki zespolone będące liczbami sprzężonymi (funkcja przekaże ich część rzeczywistą i urojoną); 0 — równanie ma pierwiastek dwukrotny; 1 — równanie ma dwa różne pierwiastki rzeczywiste. Trzy początkowe parametry wywołania funkcji to współczynniki ( $a$ ,  $b$ ,  $c$ ) równania kwadratowego, kolejne dwa to adresy zmiennych, do których zostaną przekazane wyniki.

Pascal

```

function rk(a,b,c: real; var x1, x2: real): integer;
var delta: real;
begin
  if a = 0 then rk := -2
  else
    begin
      delta := b*b-4*a*c;
      if delta > 0 then
        begin
          x1 := (-b-sqrt(delta))/(2*a);
          x2 := (-b+sqrt(delta))/(2*a);
          rk := 1;
        end
      else if delta = 0 then
        begin
          x1 := -b/(2*a);
          x2 := x1;
          rk := 0;
        end
      else
        begin
          {Pierwiastki zespolone - liczby sprzeczne}
          x1 := -b/(2*a); {Część rzeczywista}
          x2 := abs(sqrt(-delta))/(2*a); {Część urojona}
          rk := -1;
        end
    end;
end;

```

Wywołanie procedury: rk(a, b, c, x1, x2)

C, C++

```

int rk(double a, double b, double c, double * x1, double *
↳x2)
{
  if (a == 0)
    return -2;
  double delta = b*b-4*a*c;
  if (delta > 0) {
    *x1 = (-b-sqrt(delta))/(2*a);
    *x2 = (-b+sqrt(delta))/(2*a);
  }
  return 1;
} else if (delta == 0) {
  *x1 = *x2 = -b/(2*a);
  return 0;
} else { /* Pierwiastki zespolone - liczby sprzeczne */
  *x1 = -b/(2*a); /* Część rzeczywista */
  *x2 = fabs(sqrt(-delta))/(2*a); /* Część urojona */
  return -1;
}
}

```

Wywołanie funkcji: rk(a, b, c, &amp;x1, &amp;x2)



W programie do wywołania funkcji rozwiązującej równanie kwadratowe i interpretacji wyników wygodnie będzie posłużyć się instrukcją selekcji.

<b>Pascal</b>	<pre> case rk(a, b, c, x1, x2) of 1: begin     writeln('Równanie ma dwa pierwiastki rzeczywiste:');     writeln('x = ', x1:0:4);     writeln('x = ', x2:0:4); end; 0: begin     writeln('Równanie ma pierwiastek dwukrotny:');     writeln('x = ', x1:0:4); end; -1: begin     writeln('Równanie ma dwa pierwiastki zespolone:');     writeln('x = ', x1:0:4, ' - ', x2:0:4, 'i');     writeln('x = ', x1:0:4, ' + ', x2:0:4, 'i'); end; -2: writeln('a = 0, to nie jest równanie kwadratowe.');</pre>
<b>C, C++</b>	<pre> switch (rk(a, b, c, &amp;x1, &amp;x2)) { case 1:     printf("Równanie ma dwa pierwiastki rzeczywiste:\n");     printf("x = %lf\nx = %lf\n", x1, x2);     break; case 0:     printf("Równanie ma pierwiastek dwukrotny:\n");     printf("x = %lf\n", x1);     break; case -1:     printf("Równanie ma dwa pierwiastki zespolone:\n");     printf("x = %lf - %lfi\n", x1, x2);     printf("x = %lf + %lfi\n", x1, x2);     break; case -2:     printf("a = 0, to nie jest równanie kwadratowe.\n");     break; }</pre>
<b>C++</b>	Jw., z ewentualną zmianą sposobu wyświetlania wyników.

Kompletne rozwiązanie przedstawiono na FTP: *p5\_6.pas*, *p5\_6.c* i *p5\_6.cpp*.

### Przykład 5.7.

Na podstawie programu z przykładu 5.6 zbudujemy procedurę pierwiastki z trzema parametrami, wyświetlającą pierwiastki równania kwadratowego na podstawie kodu i wartości pierwiastków otrzymanych z funkcji *rk*. Zauważmy, że przekazywanie parametrów *x1* i *x2*

przez referencję w procedurze pierwiastki (Pascal) wydaje się zbędne. Jednak jeśli później parametr kod zastąpimy wywołaniem funkcji `rkw`, która z kolei umieści pierwiastki równania w zmiennych `x1` i `x2`, będzie to miało istotne znaczenie (najpierw odczytane będą wartości `x1` i `x2`, sporządzone zostaną ich kopie lokalne, a dopiero później zostanie wywołana procedura, która wartości `x1` i `x2` zmieni globalnie, nie na kopiach — stąd potrzeba użycia referencji). Czytelnik może sam sprawdzić efekty, usuwając `var` z nagłówka procedury. W C i C++ tego problemu nie ma (wynika to zapewne z innej kolejności wywoływania funkcji).

Pascal	<pre> procedure pierwiastki(kod: integer; var x1, x2: real); begin   case kod of     1: begin       writeln('x = ', x1:0:4);       writeln('x = ', x2:0:4);     end;     0: writeln('x = ', x1:0:4, '( dwukrotny)');     -1: begin       writeln('x = ', x1:0:4, ' - ', x2:0:4, 'i');       writeln('x = ', x1:0:4, ' + ', x2:0:4, 'i');     end;     -2: writeln('a = 0, to nie jest równanie kwadratowe.');</pre>
C, C++	<pre> void pierwiastki(int kod, double x1, double x2) {   switch (kod) {     case 1:       printf("x = %lf\nx = %lf\n", x1, x2);       break;     case 0:       printf("x = %lf (pierwiastek dwukrotny)\n", x1);       break;     case -1:       printf("x = %lf - %lf\n", x1, x2);       printf("x = %lf + %lf\n", x1, x2);       break;     case -2:       printf("a = 0, to nie jest równanie ↳kwadratowe.\n");       break;   } }</pre>
C++	Jw., z ewentualną zmianą sposobu wyświetlania wyników.

Rozwiązanie równania kwadratowego i wyświetlenie jego pierwiastków sprowadzi się do wywołania procedury pierwiastki i funkcji rkW (FTP: *p5\_7.pas*, *p5\_7.c* i *p5\_7.cpp*).

<b>Pascal</b>	<code>pierwiastki(rkW(a, b, c, x1, x2), x1, x2);</code>
<b>C, C++</b>	<code>pierwiastki(rkW(a, b, c, &amp;x1, &amp;x2), x1, x2);</code>

Niewiele bardziej skomplikowane będzie rozwiązanie równania dwukwadratowego:

$$ax^4 + bx^2 + c = 0, a \neq 0.$$

Po podstawieniu  $y = x^2$  rozwiążemy równanie kwadratowe  $ay^2 + by + c = 0$  (wywołanie funkcji `rkW(a, b, c, y1, y2)` w Pascalu i `rkW(a, b, c, &y1, &y2)` w C i C++). Następnie rozwiążemy kolejne równania kwadratowe:  $x^2 = y_1$  i  $x^2 = y_2$ , wywołując dwukrotnie funkcję `rkW` z odpowiednimi parametrami (FTP: *p5\_7a.pas*, *p5\_7a.c* i *p5\_7a.cpp*).

<b>Pascal</b>	<pre> case rkW(a, b, c, y1, y2) of   1: begin     {Cztery pierwiastki rzeczywiste, dwa rzeczywiste i dwa zespolone lub     ↪cztery zespolone}     pierwiastki(rkW(1, 0, -y1, x1, x2), x1, x2);     pierwiastki(rkW(1, 0, -y2, x1, x2), x1, x2);   end;   0: begin     {Dwa podwójne pierwiastki – rzeczywiste, rzeczywisty i zespolony lub     ↪dwa zespolone}     if y1 = 0 then writeln('x = 0 (pierwiastek czterokrot     ↪ny)')     else pierwiastki(rkW(1, 0, -y1, x1, x2), x1, x2);   end;   -1: begin     {Tych pierwiastków na razie nie umiemy obliczyć...}     writeln('Cztery pierwiastki zespolone...');   end;   -2: writeln('To nie jest równanie dwukwadratowe!'); end; </pre>
---------------	---

C, C++

```

switch (rkw(a, b, c, &y1, &y2)) {
  case 1:
    /* Dwa podwójne pierwiastki: rzeczywiste, rzeczywisty i zespolony lub
    ↪ dwa zespolone */
    pierwiastki(rkw(1, 0, -y1, &x1, &x2), x1, x2);
    pierwiastki(rkw(1, 0, -y2, &x1, &x2), x1, x2);
    break;
  case 0:
    /* Cztery pierwiastki rzeczywiste, dwa rzeczywiste i dwa zespolone lub
    ↪ cztery zespolone */
    if (y1 == 0)
      printf("x = 0 (pierwiastek czterokrotny)\n");
    else
      pierwiastki(rkw(1, 0, -y1, &x1, &x2), x1, x2);
    break;
  case -1:
    /* Tych pierwiastków na razie nie umiemy obliczyć... */
    printf("Cztery pierwiastki zespolone...");
    break;
  case -2:
    printf("To nie jest równanie dwukwadratowe!\n");
    break;
}

```

Do pełnego rozwiązania brakuje nam jeszcze umiejętności obliczania pierwiastka kwadratowego z liczby zespolonej.

# SKOROWIDZ

## A

algorytm Euklidesa, 123, 170, 171  
algorytmy, 22

- o strukturze liniowej, 40

obliczanie pierwiastków drugiego stopnia, 245, 246, 247  
obliczanie pierwiastków trzeciego stopnia, 247, 248, 249, 250  
obliczanie pierwiastków wyższych stopni, 251, 252  
równanie kwadratowe, 79, 80, 81, 82  
równanie liniowe, 75, 76, 77  
równanie trzeciego stopnia, 95, 96  
schemat blokowy, 23  
w postaci listy kroków, 22

alternatywa, 62, 63

- wykluczająca, 63

AND, Patrz koniunkcja

Archimedesesa, przybliżenie, 258

assembler, 28

## B

BASIC, 28

- wyświetlanie tekstów, 36

biblioteka uruchomieniowa, 27

bit, 17

- znaku, 18

Brouncker, William, 263

## C

C, język, 29

- #define, 58
- abs(), 68
- acos(), 91
- atan(), 286
- break, 100, 128
- ceil(), 148
- char, 71, 119
- chcp 1250, 42
- conio.h, 36, 110
- continue, 130

cos(), 91  
cosh(), 91  
do, 53  
double, 298  
fabs(), 68  
feof(), 292  
fflush(), 107  
fgetc(), 107, 292  
fgets(), 291, 292  
FILE, 288  
float, 298  
floor(), 148  
fopen(), 288, 289, 290, 299  
for, 112, 113, 127  
fprintf(), 288  
fputs(), 288  
fread(), 303  
fscanf(), 107  
fseek(), 311  
fwrite(), 299  
getc(), 107  
getch(), 36, 110  
getchar(), 107  
int, 42, 298  
itoa(), 211  
komentarze, 34  
log(), 92  
math.h, 45, 286  
modf(), 148  
operatory logiczne, 63  
pierwszy program, 33, 34  
plik nagłówkowy, 159  
pow(), 45, 94  
printf(), 35  
rand(), 223  
scanf(), 43, 48, 70, 107  
sinh(), 91  
sprintf(), 211  
sqrt(), 48, 70  
srand(), 223  
stdio.h, 35, 42  
stdlib.h, 36, 42  
strcat(), 209, 211

- string.h, 208
- strlen(), 210
- switch, 99, 100
- system(), 36
- time.h, 223
- tolower(), 109
- void, 70
- while, 53
- wyświetlanie tekstów, 35
- C++, język, 29
- #define, 58
- abs(), 68
- atan(), 286
- break, 100, 128
- ceil(), 148
- char, 71, 119
- cin, 44, 107
- cin.get(), 107
- cin.ignore(), 107
- clear(), 306
- close(), 288
- cmath, 45, 286
- conio.h, 110
- continue, 130
- cos(), 91
- cosh(), 91
- cout, 36
- cout.precision(), 49
- cout.setf(), 49
- cstdio, 35
- cstring, 208
- ctime, 223
- do, 53
- double, 298
- endl, 56
- eof(), 292
- fabs(), 68
- feof(), 292
- fflush(), 107
- fgetc(), 107, 292
- fgets(), 291, 292
- FILE, 288
- float, 298
- floor(), 148
- fopen(), 288, 289, 290, 299
- for, 112, 113, 127
- fprintf(), 288
- fputs(), 288
- fread(), 303
- fscanf(), 107
- fseek(), 311
- fstream, 306
- fwrite(), 299
- get(), 292
- getc(), 107
- getch(), 36, 110
- getchar(), 107
- getline(), 291, 292
- ifstream, 290, 292, 304
- int, 298
- ios::binary, 300
- ios::in, 306
- ios::out, 306
- iostream, 36, 43
- itoa(), 211
- komentarze, 34
- length(), 210
- log(), 92
- modf(), 148
- ofstream, 288, 289, 300
- operatory logiczne, 63
- pierwszy program, 33, 34
- plik nagłówkowy, 159
- pow(), 45, 94
- rand(), 223
- referencja, 136
- scanf(), 107
- seekg(), 311
- seekp(), 311
- sinh(), 91
- size(), 210
- sprintf(), 211
- sqrt(), 48, 70
- srand(), 223
- standardowa przestrzeń nazw, std, 44
- string, 208
- strlen(), 210
- switch, 99, 100
- system(), 36
- tolower(), 109
- void, 70
- while, 53
- wyświetlanie tekstów, 35, 36
- cecha, 20, 200
- Ceulen, Ludolf van, 258
- ciąg liczbowy
  - arytmetyczny, 166
  - geometryczny, 167
  - harmoniczny, 268
- ciąg Fibonacciego, 172, 257
  - drzewo rekurencyjne, 173
- CodeBloks, 38
  - tworzenie programu, 39
- continued fraction, Patrz ułamki łańcuchowe

**D**

DevC++, 38  
 tworzenie programu, 39  
 dialog z użytkownikiem, 106, 107, 110  
 Diofantos, 9  
 dwumian Newtona, 168, 169  
 dyrektywy preprocesora, #define, 58  
 działania arytmetyczne, własności, 14, 15

**E**

e, liczba, 92, 265  
 przybliżenie, 275  
 Euklides, 9  
 Euler, Leonard, 262, 266, 267  
 Eulera, wzór, 267

**F**

FORTTRAN, 28  
 Free Pascal, 38  
 kompilacja, 38  
 funkcja wykładnicza, 277  
 funkcja, w języku programowania, 70, 131  
 przekazywanie danych, 132, 133, 134  
 funkcje trygonometryczne, 151, 278, 279, 282  
 dla kąta mierzonego w radianach, 153  
 dla kąta mierzonego w stopniach, 151, 152  
 wzory redukcyjne, 281  
 zależność, 152  
 funktory, 62

**G**

goto, Patrz instrukcja skoku  
 Grossmann, H. G., 10

**H**

Hamilton, William, 15  
 Herona, wzór, 47  
 Hilbert, David, 10  
 Hornera, schemat, 215

**I**

IEEE 754, 200  
 iloczyn logiczny, Patrz koniunkcja  
 implementacja, 23  
 instrukcja decyzyjna, Patrz instrukcja wyboru  
 instrukcja skoku, 54  
 instrukcja wyboru, 98  
 instrukcje iteracyjne, 112

instrukcje warunkowe  
 instrukcja, 49  
 warunek, 49  
 zagnieżdżanie, 75  
 interpreter, 26, 27

**J**

język maszynowy, 27  
 język programowania, 25, 26  
 niskiego poziomu, 28  
 wybór, 6  
 wysokiego poziomu, 28

**K**

kąty, 145  
 Kemeny, John George, 28  
 Kepler, 253  
 Kernighan, B., 29  
 kod maszynowy, 26  
 kod uzupełnień do dwóch, 180  
 kod uzupełnień do jedności, 179  
 kod znak-moduł, 179  
 kod źródłowy, 26  
 komentarze, stosowanie, 34  
 kompilacja, 27  
 kompilator, 27  
 koniunkcja, 62, 63  
 konsolidator, 27  
 Kurtz, Thomas E., 28  
 kwaterniony, 15, 16

**L**

liczba Eulera, Patrz e, liczba  
 liczba Nepera, Patrz e, liczba  
 liczba  $\phi$ , 253, 257  
 liczby  
 algebraiczne, 13  
 całkowite, 9, 10, 179  
 naturalne, 9, 178  
 niewymierne, 13, 15, 236  
 przestępne, 13  
 reprezentacja w komputerze, 20, 178, 179, 180,  
 187, 200, 204  
 rzeczywiste, 13, 14  
 systemy zapisu, 16, 17, 18, 19  
 ujemne, 9  
 urojone, 15  
 własności działań arytmetycznych, 14, 15  
 wymierne, 10, 11, 12  
 zespolone, 15, 82

zmiennoprzecinkowe, 199, 200, 204  
 linker, Patrz konsolidator  
 linkowanie, Patrz konsolidacja  
 Lispu, język, 31  
 LOGO, 31, 32

**Ł**

łańcuchowy typ danych, 208

**M**

makrodefinicja, 58  
 mantysa, 20, 200  
 metoda iteracyjna, 244  
 metoda kolejnych przybliżeń, 244  
 metoda Newtona-Raphsona, 245, 249  
 miara łukowa kąta, 145  
 miara stopniowa kąta, 145  
 MinGW, 38  
 minuta kątowna, 146

**N**

najmniejsza wspólna wielokrotność, 122, 193  
 największy wspólny dzielnik, 123, 170, 171, 193  
 naturalny kod binarny, 178  
 negacja, 62, 63  
 nierówność trójkąta, 66, 67, 68  
 NKB, Patrz naturalny kod binarny  
 NOT, Patrz negacja  
 NWD, Patrz największy wspólny dzielnik  
 NWW, Patrz najmniejsza wspólna wielokrotność

**O**

operator warunkowy, 103, 104  
 operator wyłuskania, 135  
 operatory  
   logiczne, 62, 63  
   porównania, 59  
   równości, 60  
 OR, Patrz alternatywa

**P**

Papert, Seymour, 31  
 Pascal, język, 29  
   abs(), 68  
   append(), 289  
   arctan(), 91, 286  
   array, 218, 219  
   assign(), 288  
   blockread(), 304

blockwrite(), 302  
 case, 99  
 char, 71  
 chr(), 119  
 close(), 288  
 concat(), 209  
 cos(), 91  
 eof(), 291  
 exit, 128  
 exp(), 92  
 for, 112, 113  
 frac(), 147  
 komentarze, 34  
 length(), 210  
 ln(), 92  
 longint, 298  
 lowercase(), 109  
 moduły, 110  
 operatory logiczne, 63  
 ord(), 119  
 Pi, 57  
 pierwszy program, 33  
 random(), 223  
 randomize, 223  
 read(), 40, 41, 107, 292  
 readkey(), 110  
 readln(), 40, 41, 70, 107  
 real, 200, 298  
 repeat, 52  
 reset(), 290  
 rewrite(), 288  
 round(), 104, 148  
 sqr(), 42  
 sqrt(), 48, 70  
 string, 208  
 text, 288  
 trunc(), 148  
 typ wskaźnikowy, 135  
 until, 53  
 var, 40, 135  
 własny moduł, 156, 157  
 write(), 35, 40, 41  
 writeln(), 35, 40, 41  
 wyświetlanie tekstów, 35  
 Peano, Giuseppe, 9  
 pętle  
   instrukcja, 53  
   o stałej liczbie powtórzeń, 112, 113  
   porównanie, 124, 125, 126, 127  
   przerwanie działania, 127, 128, 129  
   schemat blokowy, 53  
   warunek, 53



ze sprawdzaniem warunku na końcu, 52, 120  
ze sprawdzaniem warunku na początku, 123, 124

pi, liczba, 258, 262

pierwiastek dwukrotny, 85

plik beztypowy, 302

plik binarny, 298, 299, 302  
modyfikacja danych, 305  
odczytywanie danych, 302

plik jednorodny, 299

pliki tekstowe  
błędy otwarcia pliku, 295  
dopisywanie do pliku, 289  
koniec pliku, 291, 292  
odczytywanie wierszami, 290, 291  
odczytywanie znak po znaku, 292, 293  
otwieranie do odczytu, 290  
otwieranie do zapisu, 288  
tworzenie pliku, 288

potęga, 142, 164

procedura, 70, 131  
przekazywanie danych, 132, 134  
przekazywanie przez referencję, 135  
przekazywanie przez wartość, 134

procesor, 27

program komputerowy, 26

przysyłanie zmiennej, 132

pseudokod, 24

## R

rad, Patrz radian

radian, 145  
zamiana na stopnie, 146

rekord, 223

rekurencja, 163  
a iteracja, 174  
algorytm Euklidesa, 170, 171  
ciąg Fibonacciego, 172  
ciągi liczbowe, 166, 167  
potęga, 164  
silnia, 163, 164  
symbol Newtona, 169  
wieże Hanoi, 173

Ritchie, Dennis, 29

równania  
czwartego stopnia, 89, 98  
dwukwadratowe, 85, 86, 87, 88  
liniowe, 75  
trzeciego stopnia, 88, 90, 91, 93, 94, 95

równanie kwadratowe, 78, 104, 137, 141  
algorytm, 79, 80, 81, 82  
w zbiorze liczb zespolonych, 83, 84

różnica symetryczna, Patrz alternatywa  
wykluczająca

## S

schemat blokowy, 23  
blok decyzyjny, 24  
blok fragmentu, 24  
blok graniczny, 23  
blok komentarza, 24  
blok obliczeniowy, 24  
blok wejścia-wyjścia, 23  
blok wywołania podprogramu, 24  
instrukcje warunkowe, 51  
łącznik wewnętrzny, 24  
łącznik zewnętrzny, 24

sekunda kątowna, 146

sieć działań, 23

silnia, 163, 164

SM, Patrz kod znak-moduł

spójniki zdaniowe, 62

stałopozycyjny, zapis, 20

stałoprzecinkowy, zapis, Patrz stałopozycyjny,  
zapis

Stewin, Simon, 9

stopień kątowny, 146  
zamiana na radiany, 146

Stroustrup, Bjarne, 29

struktura, 223  
deklaracja, 224

suma logiczna, Patrz alternatywa

suma szeregu, 269, 276

symbol Newtona, 168, 169

systemy zapisu liczb  
dwójkowy, 17, 18  
dziesiętkowy, 16, 17, 18

szereg liczbowy, 269  
funkcyjny, 276  
geometryczny, 270  
harmoniczny, 270

## Ś

średnia harmoniczna, 268

## T

tablice, 217  
struktur, 232  
wielowymiarowe, 222

tablicowy typ danych, 217

translacja, 26

trójkąt Pascala, 168, 169

Turbo Pascal, 36, 38  
kompilacja, 37

## U

U1, Patrz kod uzupełnień do jedności  
U2, Patrz kod uzupełnień do dwóch  
układ  
  binarny, Patrz układ dwójkowy  
  dwójkowy, 17, 18, 179, 180, 183, 184  
  dziesiętkowy, 16, 17, 18  
  heksadecymalny, Patrz układ szesnastkowy  
  oktalny, Patrz układ ósemkowy  
  ósemkowy, 18  
  szesnastkowy, 18  
ułamki, 9, 19  
  dodawanie, 192, 194  
  dzielenie, 191  
  dziesiętne, 9, 11  
  łańcuchowe, 13, 196  
  mnożenie, 190  
  odejmowanie, 192, 194  
  okresowe, 12  
  podstawowe działania, 11  
  reprezentacja w komputerze, 187  
  skrącanie, 190  
  wspólny mianownik, 193  
  zamiana ułamka dziesiętnego na zwykły, 12  
  zamiana ułamka zwykłego na dziesiętny, 11,  
  12  
  zamiana ułamka zwykłego na ułamek  
    łańcuchowy arytmetyczny, 196  
  zwykłe, 9, 11, 187

## V

Viète, François, 264

## W

Wallis, John, 264  
wieże Hanoi, 173  
Wirth, Nielaus, 29  
wskaźnik, 135  
LOGO, 36

## X

XOR, Patrz alternatywa wykluczająca

## Z

zagnieżdżanie instrukcji warunkowych, 75  
zapis

  stałopozycyjny, 20  
  zmiennopozycyjny, 20, 21  
zbieżność punktowa, 276  
złoty podział odcinka, 252, 253, 257  
ZM, Patrz kod znak-moduł  
zmienna sterująca, 112  
zmienne  
  globalne, 131, 132  
  lokalne, 131, 132  
zmiennopozycyjny, zapis, 20, 21  
zmiennoprzecinkowy, zapis, Patrz  
  zmiennopozycyjny, zapis  
ZU1, Patrz kod uzupełnień do jedności  
ZU2, Patrz kod uzupełnień do dwóch

# Wędrówka do źródła kodu

Popularna definicja programowania określa je jako „proces projektowania, tworzenia, testowania i utrzymywania kodu źródłowego programów komputerowych lub urządzeń mikroprocesorowych”. Wspomniany **kod źródłowy** może być napisany w różnych językach programowania, z użyciem określonych reguł. Każdy z języków pozwala na wykorzystanie odpowiednich stylów programowania, a wybór konkretnego języka może zależeć od indywidualnych upodobań, polityki firmy lub funkcji, jakie końcowa aplikacja ma realizować. W zasadzie nie istnieje odpowiedź na pytanie, który z języków jest najlepszy. Dlatego w tej książce nie znajdziesz typowego abecadła. Zapoznasz się za to z danym problemem, a następnie programem komputerowym służącym do jego rozwiązania.

Jeśli chcesz wreszcie rozpocząć przygodę z programowaniem i nawiązać dialog ze swoim komputerem, ta publikacja jest właśnie dla Ciebie! Różnorodne obliczenia, mniej lub bardziej skomplikowane, znane Ci z lekcji matematyki lub nieznacznie wykraczające poza program nauczania, stanowią tutaj podstawę do zdobywania informacji na temat programowania w wybranych językach. Wybrane zadania zaprezentowane są w popularnych językach programowania: Pascal, C i C++. Stosowane algorytmy wymagają także sięgnięcia po różne funkcje matematyczne, dostępne standardowo w bibliotekach języków programowania oraz konstruowane na podstawie wzorów.

## Zostań informatycznym poliglota. Programuj każdego dnia!

40 stronowy 1990



Księgarnia internetowa:  
<http://helion.pl>



Zamówienia telefoniczne:  
**0 801 339900**



**0 601 339900**



**Helion**

Sprawdź najnowsze promocje:  
@ <http://helion.pl/promocje>  
#tagi: najlepszej cenie!  
@ <http://helion.pl/interaktywny>  
Zamów informacje o nowościach:  
@ <http://helion.pl/nowosci>

Helion SA  
ul. Rakoczkowska 1c, 44-100 Gliwice  
tel.: 02 230 19 43  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
<http://helion.pl>

**helion.pl**  
Książki  
Internetowa

ISBN 978-83-246-3210-7



9 788324 632107

Informatyka w najlepszym wydaniu