Od matematyki do programowania

Wszystko, co każdy programista wiedzieć powinien

Wiesław Rychlicki

Helion

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!

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

SPIS TREŚCI

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*;[1].

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`:

Pascal	`for i := 1 to 15 do` `  instrukcja_do_powtarzania;`

C, C++	`int i;`
	`for (i = 1; i <= 15; ++i)`
	`  instrukcja_do_powtarzania;`

C++	`for (int i = 1; i <= 15; ++i)` `  instrukcja_do_powtarzania;`

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 deklaracyjnej (przed głównym blokiem programu, procedury lub funkcji). W składni instrukcji widzimy wyraźnie ustalenie zakresu zmiennej `for i := 1 to 15...` i wskazanie instrukcji do

[1] 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 sześcianów, a argumentem będzie zmieniająca się wartość zmiennej sterującej.

Pascal

```
for i := 1 to 15 do
begin
  kw := i*i;
  sz := kw*i;
  writeln(i:3, kw:5, sz:7);
end;
```

C, C++

```
int i;
for (i = 1; i <= 15; ++i) {
    int kw, sz;
    kw = i*i;
    sz = kw*i;
    printf("%3d%5d%7d\n", i, kw, sz);
}
```

C++

```
for (int i = 1; i <= 15; ++i) {
    int kw, sz;
    kw = i*i;
    sz = kw*i;
    cout.width(3); cout << i;
    cout.width(5); cout << kw;
    cout.width(7); cout << sz << endl;
}
```

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

[2] 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*).

Pascal

```
for i := 1 to 15 do
  writeln(i:3, i*i:5, i*i*i:7);
```

C, C++

```
int i;
for (i = 1; i <= 15; ++i)
  printf("%3d%5d%7d\n", i, i*i, i*i*i);
```

C++

```
for (int i = 1; i <= 15; ++i)
  cout<< setw(3) << i << setw(5) << i*i << setw(7)<<
  ↪i*i*i <<endl;
```

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.

Pascal

```
for i := 0 to 20 do writeln(i/10:5:1, sqrt(i/10):12:8);
```

C, C++

```
int i;
for (i = 0; i <= 20; ++i)
    printf("%5.1lf%12.8lf\n", i/10.0, sqrt(i/10.0));
```

C++

```
cout.setf(ios::fixed);
for (int i = 0; i <= 20; ++i) {
    cout << setw(5) << setprecision(1) << i/10.0;
    cout << setw(12) << setprecision(8) << sqrt(i/10.0) <<
    ↪endl;
}
```

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*).

Pascal

```
for i := 0 to 20 do
  begin
    x := i/10;
    writeln(x:5:1, sqrt(x):12:8);
  end;
```

C, C++

```
double x;
int i;
for (i = 0; i <= 20; ++i) {
    x = i/10.0;
    printf("%5.1lf%12.8lf\n", x, sqrt(x));
}
```

C++

```
double x;
cout.setf(ios::fixed);
for (int i = 0; i <= 20; ++i) {
    x = i/10.0;
    cout << setprecision(1) << setw(5) << x;
    cout << setprecision(8) << setw(12) << sqrt(x) << endl;
}
```

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*).

Pascal

```
x := 0;
for i := 0 to 20 do
  begin
    writeln(x:5:1, sqrt(x):12:8);
    x := x + 0.1;
  end;
```

C, C++

```
double x = 0;
int i;
for (i = 0; i <= 20; ++i) {
    printf("%5.1lf%12.8lf\n", x, sqrt(x));
    x += 0.1;
}
```

Jak wcześniej wspomnieliśmy, w C/C++ zmienna sterująca pętlą nie musi być liczbą całkowitą. Możemy zatem ze zmiennej `i` zrezygnować i po nieznacznej 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("%5.1lf%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 dziesiątkowym) 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 <= 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("%5.1lf%12.8lf\n", x, sqrt(x));
```

Przykład 4.3.

Sporządzimy 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 `x` 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;
```

C, C++

```
printf("  x  ");
for(i = 1; i <= 10; ++i)
    printf("%5d", i);
printf("\n");
```

C++

```
cout << "  x  ";
for(int i = 1; i <= 10; ++i)
    cout << setw(5) << i;
cout << endl;
```

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

Pascal

```
for j := 1 to 10 do
  begin
    write(j:3, '  ');
    for i := 1 to 10 do write(j*i:5);
    writeln;
  end;
```

C, C++

```
for(j = 1; j <= 10; ++j) {
    printf("%3d  ", j);
    for(i = 1; i <= 10; ++i)
        printf("%5d", j*i);
    printf("\n");
}
```

C++

```
for(j = 1; j <= 10; ++j) {
    cout << setw(3) << j << "  ";
    for(i = 1; i <= 10; ++i)
        cout << setw(5) << j*i;
    cout << endl;
}
```

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 jedności (co jest częstą praktyką w bardziej obszernych tablicach wartości funkcji).

Wiersz nagłówkowy:

Pascal

```
write(' sqrt');
for i := 0 to 9 do write(i:7);
writeln;
```

C, C++

```
printf(" sqrt  ");
for(i = 0; i <= 9; ++i)
    printf("%7d", i);
printf("\n");
```

Wiersze tabeli:

Pascal

```
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;
```

C, C++

```
cout << setprecision(4);
cout.setf(ios::fixed);
for(j = 0; j <= 9; ++j) {
    printf("%5d  ", 10*j);
    for(i = 0; i <= 9; ++i)
        printf("%7.4lf", sqrt(10*j+i));
    printf("\n");
}
```

C++

```
cout << setprecision(4);
cout.setf(ios::fixed);
for(j = 0; j <= 9; ++j) {
    cout << setw(5) << 10*j << "  ";
    for(i = 0; i <= 9; ++i)
        cout << setw(7) << sqrt(10*j+i));
    cout << endl;
}
```

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 pokażemy *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:

Pascal

```
var znak: char; {deklaracja zmiennej}
```

```
for znak := 'a' to 'z' do
  write(znak);
```

C, C++

```
char znak; /* deklaracja zmiennej */
for(znak = 'a'; znak <= 'z'; ++znak)
    printf("%c", znak);
```

C++

```
for(char znak = 'a'; znak <= 'z'; ++znak)
    cout << znak;
```

oraz w drugą stronę — od z do a[3] (FTP: *p4_5.pas*, *p4_5.c* i *p4_5.cpp*):

Pascal

```
for znak := 'z' downto 'a' do
  write(znak);
```

C, C++

```
for(znak = 'z'; znak >= 'a'; --znak)
    printf("%c", znak);
```

C++

```
for(char znak = 'z'; znak >= 'a'; --znak)
    cout << znak;
```

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*).

Pascal

```
var z: byte; {deklaracja zmiennej}
```

```
for z := 97 to 122 do
  write(chr(z));
```

[3] Korzystamy z tej samej zmiennej `znak`, już wcześniej (Pascal, C). W C++ zmienną możemy każdorazowo deklarować jako zmienną lokalną w instrukcji `for`.

C, C++

```
int z;
for(z = 97; z <= 122; ++z)
    printf("%c", z);
printf("\n");
for(z = 122; z >= 97; --z)
    printf("%c", z);
```

C++

```
for(int z = 97; z <= 122; ++z)
    cout << (char) z;
cout << endl;
for(int z = 122; z >= 97; --z)
    cout << (char) z;
```

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*).

Pascal

```
var z: byte;
for znak := 'A' to 'Z' do
  write(znak, chr(ord(znak) + 32));
```

C, C++

```
char znak;
for(znak = 'A'; znak <= 'Z'; ++znak)
    printf("%c%c", znak, znak + 32);
```

C++

```
for(char znak = 'A'; znak <= 'Z'; ++znak)
    cout << znak << (char)(znak + 32);
```

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*).

Pascal

```
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 > 0 then
  writeln('Średnia arytmetyczna: ', suma/licznik:0:5);
```

C, C++

```
double x, suma = 0;
int licznik = -1;
do {
    printf("x = ");
    scanf("%lf", &x);
    suma += x;
    ++licznik;
} while (x != 0);
printf("Suma liczb: %0.5lf\n", suma);
if (licznik > 0)
    printf("Średnia arytmetyczna: %0.5lf\n", suma/licznik);
```

C++

```
double x, suma = 0;
int licznik = -1;
do {
    cout << "x = ";
    cin >> x;
    suma += x;
    ++licznik;
} while (x != 0);
cout << "Suma liczb: " << suma << endl;
if (licznik > 0)
    cout << "Średnia arytmetyczna: " << suma/licznik
    ↪<<endl;
```

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 $NWW(m, n)$.

Pascal

```
wm := m; {wielokrotność liczby m}
wn := n; {wielokrotność liczby n}
repeat
  if wm < wn then wm := wm+m; {kolejna wielokrotność m}
  if wn < wm then wn := wn+n; {kolejna wielokrotność n}
until wm = wn;
writeln('NWW(', m, ', ', m, ') = ', wm);
```

C, C++

```
wm = m; /* Wielokrotność liczby m */
wn = n; /* Wielokrotność liczby n */
do {
    if (wm < wn)
        wm += m; /* Kolejna wielokrotność m */
    if (wn < wm)
        wn += n; /* Kolejna wielokrotność n */
} while(wm != wn);
printf("NWW(%d, %d) = %d\n", m, n, wm);
```

C++

```
/* Jak wyżej */
cout << "NWW(" << m << ", " << n << ") = " << wm << endl;
```

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

Pascal

```
repeat
  write('m = ');
  readln(m)
until m > 0;
repeat
  write('n = ');
  readln(n)
until n > 0;
```

C, C++

```
do {
    printf("m = ");
    scanf("%d", &m);
} while (m <= 0);
do {
    printf("n = ");
    scanf("%d", &n);
} while (n <= 0);
```

C++

```
do {
    cout << "m = ";
    cin >> m;
} while (m <= 0);
do {
    cout << "n = ";
    cin >> n;
} while (n <= 0);
```

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*).

Pascal

```
km := m; {Kopia liczby m}
kn := n; {Kopia liczby n}
while (km <> kn) do
  if km > kn then km := km–kn else kn := kn-km;
writeln('NWD(', m, ', ', n, ') = ', km);
```

C, C++

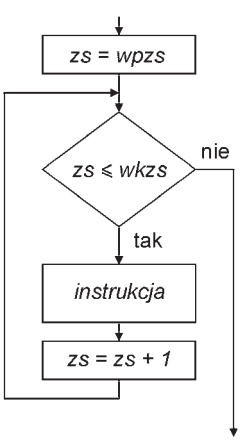
```
km = m; /* Kopia liczby m */
kn = n; /* Kopia liczby n */
while(km != kn)
    if (km > kn) km -= kn; else kn -= km;
printf("NWD(%d, %d) = %d\n", m, n, km);
```

C++

```
/* Jak wyżej */
cout << "NWD(" << m << ", " << n << ") = " << km << endl;
```

Zastosowana tu pętla ze sprawdzaniem warunku na początku różni się składnią (różnica pomiędzy Pascalem 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 przeanalizujmy 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;
```

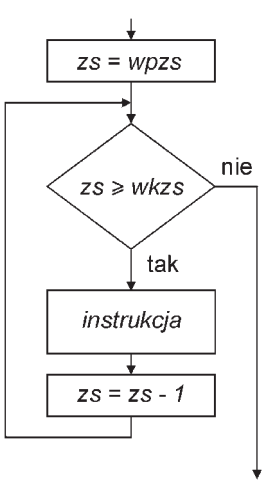
```
zs := wpzs; {Wartość początkowa zmiennej sterującej}
while wpzs <= wkzs do
  begin
    instrukcja;
    zs := zs+1;
  end;
```

C, C++

```
for(zs = wpzs; zs <= wkzs; ++i) instrukcja;
```

```
zs = wpzs; /* Wartość początkowa zmiennej sterującej */
while (zs <= wkzs) {
    instrukcja;
    ++zs
};
```

```
zs = 1; /* Wartość początkowa zmiennej sterującej */
while (zs++ <= wkzs)
    instrukcja;
```



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

Pascal

```
for zs:= wpzs downto wkzs do instrukcja;
```

```
zs := wpzs; {Wartość początkowa zmiennej sterującej}
while zs >= wkzs do
  begin
    instrukcja;
    zs := zs-1;
  end;
```

C, C++

```
for(zs = wpzs; zs >= wkzs; ++zs) instrukcja;
```

```
zs = wpzs; /* Wartość początkowa zmiennej sterującej */
while (zs >= wkzs) {
    instrukcja;
    --i
};
```

```
zs = wpzs; /* Wartość początkowa zmiennej sterującej */
while (i-- = n)
    instrukcja;
```

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).

Pascal

```
for zs:= wpzs to wkzs do instrukcja;
```

```
zs := wpzs; {Wartość początkowa zmiennej sterującej}
if zs <= wkzs then
  repeat
    instrukcja;
    zs := zs+1;
  until zs > wkzs;
```

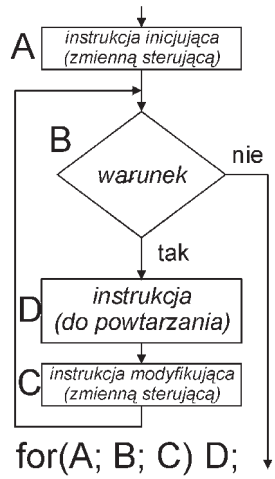
C, C++

```
for(zs = wpzs; zs <= wkzs; ++zs)
    instrukcja;
```

```
zs = wpzs; /* Wartość początkowa zmiennej sterującej */
if (zs <= wkzs)
    do {
        instrukcja;
        ++zs;
    } while (zs <= wkzs);
```

```
zs = wpzs; /* Wartość początkowa zmiennej sterującej */
if (zs <= wkzs)
    do {
        instrukcja;
    } while (++zs <= wkzs);
```

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



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

Przerywanie działania pętli

Pętle wykonują się z góry określoną liczbę 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');
```

C, C++

```
for(n = 1; n <= 10; ++n) {
    printf("n = ");
    if (n == 5)
        break;
    printf("%d\n", n);
}
printf("Pętla została przerwana dla n = 5\n");
```

C++

```
for(int n = 1; n <= 10; ++n) {
    cout << "n = ";
    if (n == 5)
        break;
    cout << n << endl;
}
cout << "Pętla została przerwana dla n = 5\n";
```

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.

Pascal

```
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.
```

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*).

Pascal

```
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.
```

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*).

Pascal

```
n:= 1;
repeat
  write('n = ');
  if n = 5 then n := 10
           else writeln(n);
  n := n+1;
until n > 10;
writeln('Pętla została przerwana dla n = 5');
```

```
n:= 1;
while n <= 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');
```

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*).

Pascal

```
suma := 0;
repeat
  write('n = ');
  readln(n);
  if n > 0 then
    begin {Dodawanie wyłącznie liczb dodatnich}
      suma := suma+n;
      writeln('S = ', suma);
    end;
until suma >= 100;
```

C, C++

```
suma = 0;
do {
    printf("n = ");
    scanf("%d", &n);
    if (n > 0) { /* Dodawanie wyłącznie liczb dodatnich */
        suma += n;
        print("S = %d\n", suma);
    }
} while (suma < 100);
```

```
suma = 0;
do
{
  printf("n = ");
  scanf("%d", &n);
  if (n <= 0)continue;
  /* continue – pominięcie dalszych instrukcji w pętli, dodawanie wyłącznie
  ↪liczb dodatnich */
  suma += n;
  print("S = %d\n", suma);
} while (suma < 100);
```

C++

Jw., wiersze: `printf("n = "); scanf("%d", &n);`
można zastąpić wierszami: `cout << "n = "; cin >> n;`
Podobnie wiersz: `print("S = %d\n", suma);`
zastąpimy wierszem: `cout << "S = " << suma << endl;`

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”[1];
- procedura działa zależnie od wartości pewnej zmiennej globalnej — brak standardowego przykładu jest chyba zrozumiały, 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 (popełniony 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.

[1] 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.

Pascal

Procedura:

```
procedure nazwa_procedury(lista_parametrów);
  {Deklaracje lokalnych stałych, zmiennych, procedur lub funkcji}
begin
  {Instrukcje – ciało procedury}
end;
```

Funkcja:

```
function nazwa_funkcji(lista_parametrów): typ_wartości;
  {Deklaracje lokalnych stałych, zmiennych, procedur lub funkcji}
begin
  {Instrukcje – ciało procedury}
  {Wynik obliczeń jest na koniec przypisany do nazwy funkcji}
  nazwa_funkcji := wyrażenie;
end;
```

C, C++

Funkcja niezwracająca wyniku — odpowiednik procedury w Pascalu:

```
void nazwa_funkcji(lista_parametrów)
   {
        /* Deklaracja zmiennych lokalnych */
        /* Instrukcje – ciało funkcji */
        return;
   }
```

Funkcja:

```
typ_wyniku nazwa_funkcji(lista_parametrów)
   {
        /* Deklaracja zmiennych lokalnych */
        /* Instrukcje – ciało funkcji */
     return wyrażenie; /* Zwrócenie obliczonego wyniku */
   }
```

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

```
procedure komunikat();
begin
  writeln('To jest komunikat!');
end;
```

Wywołanie procedury: `komunikat();` lub `komunikat;`

C, C++

```
void komunikat(void)
{
    printf("To jest komunikat!\n");
}
```

Wywołanie procedury: `komunikat();`

C++

```
void komunikat(void)
{
    cout << "To jest komunikat!\n";
}
```

Wywołanie procedury: `komunikat();`

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)`.

Pascal

```
procedure suma(m, n: integer);
begin
  writeln(m+n);
end;
```

Wywołanie procedury: `suma(14, 29); suma(x, 13); suma(a, b);`

C, C++

```
void suma(int m, int n)
{
    printf("%d\n", m+n);
}
```

Wywołanie funkcji: `suma(14, 29); suma(x, 13); suma(a, b);`

C++

```
void suma(int m, int n)
{
    cout << m+n << endl;
}
```

Wywołanie funkcji: `suma(14, 29); suma(x, 13); suma(a, b);`

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.

Pascal

```
procedure suma(x, y: real; var s: real);
begin
    s := x+y;
end;
```

Wywołanie procedury:

```
suma(14, 2.9, z); suma(x, 13.1, a); suma(a, b, x);
```

C, C++

```
void suma(double x, double y, double* s)
{
    *s = x+y;
}
```

Wywołanie funkcji (procedury):

```
suma(14, 2.9, &z); suma(x, 13.1, &a); suma(a, b, &x);
```

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

```
type preal = ^real; {Typ wskazujący na real}
procedure suma(x, y: real; s: preal);
begin
    s^ := x+y;
end;
```

Wywołanie procedury:

```
suma(14, 2.9, @z); suma(x, 13.1, @a); suma(a, b, @x);
```

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++

```
void suma(double x, double y, double& s)
{
    s = x+y;
}
```

Wywołanie funkcji (procedury):

```
suma(14, 2.9, z); suma(x, 13.1, a); suma(a, b, x);
```

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

```
function suma(x, y: real): real;
begin
  suma := x+y;
end;
```

Wywołanie funkcji:

```
z := suma(14, 2.9); a := suma(x, 13.1); x := 2*suma(a, b);
```

C, C++

```
double suma(double x, double y)
{
    return x+y;
}
```

Wywołanie funkcji:

```
z = suma(14, 2.9); a = suma(x, 13.1); x = 2*suma(a, b);
```

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 → pomocnik | y → x | pomocnik → y

Pascal

```
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;
```

Wywołanie procedury: `zamiana(a, b);`

C, C++

```
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;
}
```

Wywołanie funkcji: `zamiana(&a, &b);`

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 rkw(a,b,c: real; var x1, x2: real): integer;
  var delta: real;
begin
  if a = 0 then rkw := -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);
          rkw := 1;
        end
      else if delta = 0 then
        begin
          x1 := -b/(2*a);
          x2 := x1;
          rkw := 0;
        end
      else
        begin
          {Pierwiastki zespolone – liczby sprzężone}
          x1 := -b/(2*a); {Część rzeczywista}
          x2 := abs(sqrt(-delta)/(2*a)); {Część urojona}
          rkw := -1;
        end
    end;
end;
```

Wywołanie procedury: `rkw(a, b, c, x1, x2)`

C, C++

```
int rkw(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 sprzężone */
        *x1 = -b/(2*a); /* Część rzeczywista */
        *x2 = fabs(sqrt(-delta)/(2*a)); /* Część urojona */
        return -1;
    }
}
```

Wywołanie funkcji: `rkw(a, b, c, &x1, &x2)`

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

Pascal

```
case rkw(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.');
end;
```

C, C++

```
switch (rkw(a, b, c, &x1, &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;
}
```

C++

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 `rkw`. 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

```
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.');
  end;
end;
```

C, C++

```
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 - %lfi\n", x1, x2);
            printf("x = %lf + %lfi\n", x1, x2);
            break;
        case -2:
            printf("a = 0, to nie jest równanie
            ↪kwadratowe.\n");
            break;
    }
}
```

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*).

Pascal

```
pierwiastki(rkw(a, b, c, x1, x2), x1, x2);
```

C, C++

```
pierwiastki(rkw(a, b, c, &x1, &x2), x1, x2);
```

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*).

Pascal

```
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;
```

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

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!

ISBN 978-83-246-3210-7