Dyrektywy

Dyrektywy preprocesora to specjalne polecenia stosowane w kodzie programu, które powodują odpowiednią kompilację programu, jednak nie wpływają na sam przebieg programu po jego skompilowaniu.

Preprocesor jest programem, dokonującym wstępnej obróbki kodu źródłowego przed jego kompilacją.

#include

Dyrektywa ta powoduje, że kompilator w miejscu jej wystąpienia wstawia wskazany plik z kodem źródłowym. Ogólne, równoważne postacie:

#include "*nazwa_pliku*"
#include <*nazwa_pliku*>

Możliwe jest stosowanie samej nazwy pliku (bez ścieżki dostępu) znajdującego się w katalogu pliku, z którego dyrektywa jest wywołana, w katalogu plików nagłówkowych ustawionych w kompilatorze, lub też pełna nazwa ze ścieżką pliku.

Dyrektywa
#include<iostream>
Informuje kompilator, aby pobrał plik „iostream”,
który zawiera deklaracje strumieni wejścia i wyjścia: cin, cout,
operatora wyjścia(<<) i operatora ekstrakcji (>>)
Umożliwia to wykonanie poprawnego łączenia kodu wynikowego
biblioteki „iostream” z instrukcjami wejscia/wyjscia w programie.

Operatory arytmetyczne

Operator	Działanie	Przykład
+	dodawanie	a = b + c;
-	odejmowanie	a = b - c;
*	mnożenie	a = b * c;
/	dzielenie	a = b / c;
%	reszta z dzielenia (modulo)	a = b % c;

Uwagi

Wszystkie operatory za wyjątkiem operatora % można stosować zarówno do argumentów *całkowitych*, jak i *zmiennoprzecinkowych*. Operatory + i - można również stosować jako operatory *jednoargumentowe*.

Jeśli przy dzieleniu liczb całkowitych iloraz zawiera część ułamkową, jest ona odrzucana.

Przykłady:

25 / 7	→	3;
25 / 7.	→	3.571428
35. / 5	→	7.0
1 / 4	→	0
19 % 6	→	1
0 % 5	→	0
18 % 6	→	0

Operatory relacji

Operator	Działanie	Przykład
<	mniejszy	a < b
<=	mniejszy lub równy	a <= b
>	większy	a > b
>=	większy lub równy	a >= b
==	równy	a ==b
!=	nie równy	a != b

Operatory logiczne

Operator	Działanie	Przykład
!	negacja	! a
&&	koniunkcja (iloczyn logiczny)	a && b
||	alternatywa (suma logiczna)	a || b

Operatory logiczne

Wyrażenia połączone dwuargumentowymi operatorami logicznymi koniunkcji i alternatywy zawsze *są wartościowane od strony lewej do prawej*. Kompilator oblicza wartość wyrażenia dotąd, dopóki na pewno nie wie jaki będzie wynik.

Oznacza to, że w wyrażeniu

```
( a == 0 ) && ( m == 5 ) && ( x > 23 )
```

kompilator będzie obliczał od lewej do prawej, a jeśli pierwszy czynnik koniunkcji nie będzie prawdziwy, dalsze obliczanie zostanie przerwane.

Typ	Typowy rozmiar w bajtach	Minimalny zakres
`char`	1	–128 ... 127
`unsigned char`	1	0 ... 255
`int`	2, 4 lub 8	–32 768 ... 32 767
`unsigned int`	2, 4 lub 8	0 ... 65 535
`short int`	2	–32 768 ... 32 767
`unsigned short int`	2	0 ... 65 535
`long int`	4	–2 147 483 648 ... 2 147 483 647
`unsigned long int`	4	0 ... 4 294 967 295
`float`	4	sześć cyfr znaczących w zapisie (–3,4E38 ... 3,4E38)*
`double`	8	dziesięć cyfr znaczących w zapisie (–1,7E308 ... 1,7E308)*
`long double`	12	dziesięć cyfr znaczących w zapisie (–1,2E4932 ... 1,2E4932)*
`bool`	1	true, false

Operator sizeof

- Operator sizeof pozwala nam rozpoznać zachowania kompilatora i komputera, z którymi przyszło nam pracować. Jest to ważne z dwóch powodów:
- Te same typy obiektów (np. zmiennych) mogą mieć w różnych implementacjach różne wielkości.
- C++ pozwala użytkownikowi na definiowanie własnych typów obiektów. Często ważne jest, by wiedzieć ile pamięci zajmuje zdefiniowany obiekt.
- Operator sizeof ma następującą składnię:

  sizeof (*nazwa_typu*)

albo

  sizeof (*nazwa_obiektu*)

Wyrażenie warunkowe

w zależności od spełnienia lub niespełnienia warunku przyjmuje jedną z dwóch postaci:

(*warunek*) ? *wartość1* : *wartość2*

Przykładowo:

(i > 5) ? 15 : 20

Jeśli warunek jest spełniony, to wyrażenie przyjmuje wartość 15, natomiast jeśli warunek nie jest spełniony, to wyrażenie przyjmuje wartość 20.

Jest to bardzo wygodna konstrukcja, ponieważ pozwala zapakować ją do wnętrza innych instrukcji, np:

c = (x > y) ? 17 : 56;

Operator rzutowania

Operator rzutowania umożliwia przekształcenie typu obiektu. Działa on w ten sposób, że bierze obiekt jakiegoś typu i jako wynik zwraca obiekt innego typu. Operator ten może mieć jedną z dwóch postaci:

(*nazwa_typu*) *obiekt*

lub

nazwa_typu (*obiekt*)

Instrukcja warunkowa if

Instrukcja if może występować w jednej z dwóch postaci:

if (*wyrażenie*) *instrukcja1*;

Najpierw oblicza się wartość wyrażenia. Jeśli jest ono prawdziwe (różne od 0), to wykonywana jest *instrukcja1*. Jeśli wartość wyrażenia jest 0 (*fałsz*), to *instrukcja1* nie jest wykonywana.

if (*wyrażenie*) *instrukcja1*;
else *instrukcja2*;

Jeśli wartość wyrażenia jest 0, to wykonywana jest *instrukcja2*.

Wybór wielowariantowy

Możemy stosować wybór wielowariantowy używając zagnieżdżoną instrukcję if..else:

if (*warunek1*) *instrukcja1*;
else if (*warunek2*) *instrukcja2*;
else if (*warunek3*) *instrukcja3*;
....;
else if (*warunekN*) *instrukcjaN*;

Zadanie 2

- Opracuj program obliczania podatku dochodowego według zasad podanych w tabeli. Użytkownik podaje podstawę obliczania podatku.

Podatek dochodowy od osób fizycznych

Skala podatkowa - 2007r

Podstawa obliczenia podatku (w zł)		Wartość podatku
poniżej	**43 405**	19% podstawy obliczenia minus kwota 572 zł 54 gr.
43 405	**85 528**	7 674 zł 41 gr. + 30% nadwyżki ponad 43.405 zł
85 528		20 311 zł 31 gr. + 40% nadwyżki ponad 85.528 zł

Kwota wolna od podatku **3.013 zł**

Instrukcja switch

Instrukcja switch służy do podejmowania wielowariantowych decyzji.

```
switch (wyrażenie)
{
   case wart1 :  instr1;
                 break;
   case wart2 :  instr2;
                 break;
   ...
   case wartn :  instrn;
                 break;
   default    :  instrn+1;
                 break;
}
```

```
/*------------------------------------------------------------------------------------------------*/
/* Program oblicza stopień na podstawie liczby otrzymanych punktów */
/* Kryteria: 0.. 49 pkt. - 2                                        */
/*            50.. 59 pkt. - 3                                      */
/*            60.. 69 pkt. - 3.5                                    */
/*            70.. 79 pkt. - 4                                      */
/*            80.. 89 pkt. - 4.5                                    */
/*            90..100 pkt. - 5                                      */
/*------------------------------------------------------------------------------------------------*/
int main ()
{
 int lp;
 float stopien;

   cout << "Podaj liczbę punktów (0 <= lp <= 100): ";
   cin >> lp;
   lp = lp/10;
   switch (lp)
    {
    case 5    : { stopien = 3; break;}
    case 6    : { stopien = 3.5; break;}
    case 7    : { stopien = 4; break;}
    case 8    : { stopien = 4.5; break;}
    case 9,10 : { stopien = 5; break;}
    default   : { stopien = 2; break;}
    }
    cout << "Twoja ocena: ";
    cout.width(3);
    cout.precision(1);
    cout << stopien << endl;
    return 0;}
```

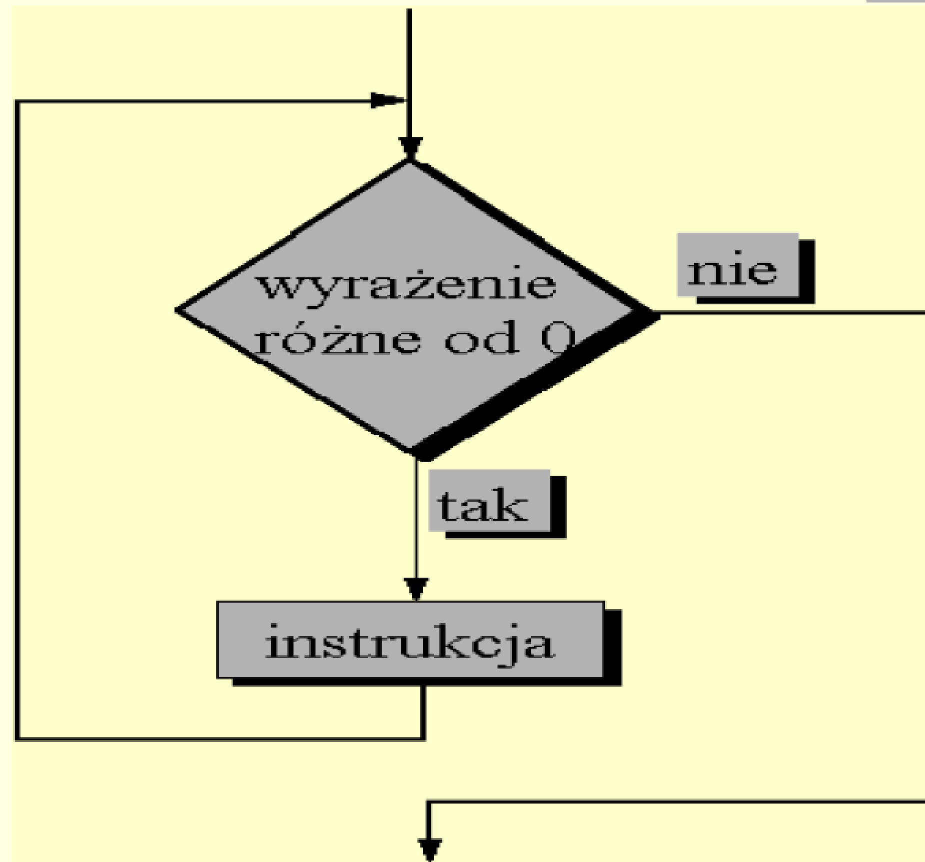
Instrukcje sterujące

Instrukcja while

while (*wyrażenie*) *instrukcja1*;

Najpierw obliczana jest wartość wyrażenia w nawiasach. Jeśli wartość ta jest *prawdziwa* (niezerowa), to następuje wykonywanie *instrukcji1* w pętli tak długo, aż wyrażenie przyjmie wartość *zerową* (fałsz). Należy zwrócić uwagę, że wartość wyrażenia jest obliczana *przed* wykonaniem instrukcji.

Instrukcja while

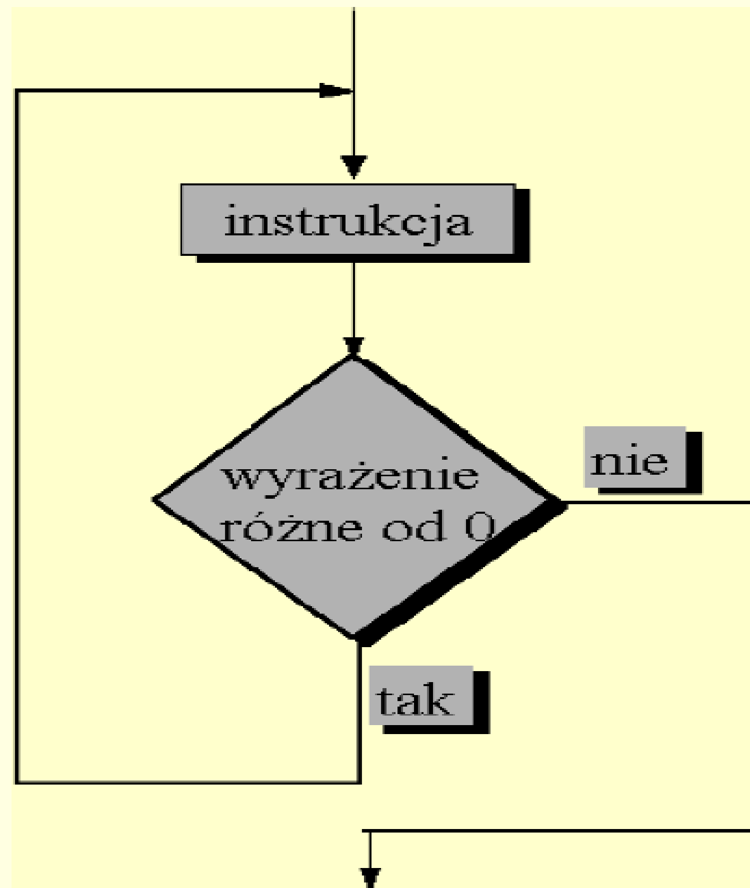


Instrukcja do...while...

do *instrukcja1* while (*wyrażenie*);

Działanie instrukcji: *instrukcja1* jest wykonywana
w pętli tak długo póki wyrażenie ma wartość *niezerową* (*prawda*). Z chwilą, gdy wyrażenie przyjmie wartość *zerową* (*fałsz*), działanie instrukcji zostaje zakończone.

Instrukcja do...while...



Instrukcja for

for (*instr_ini*; *wyraz_warunkowe*; *instr_krok*)
***treść_pętli*;**

instr_ini - jest to instrukcja wykonywana przed wykonaniem treści pętli;

wyraz_warunkowe - jest to wyrażenie obliczane przed każdym obiegiem pętli. Jeśli jest ono różne od zera, to wykonywane zostaną instrukcje będące treścią pętli;

instr_krok - jest to instrukcja wykonywana na zakończenie każdego obiegu pętli. Jest to ostatnia instrukcja wykonywana bezpośrednio przed obliczeniem wyrażenia warunkowego *wyraz_warunkowe;*

Działanie instrukcji for

1. najpierw wykonywana jest *instrukcja inicjalizująca* pracę pętli;
2. obliczane jest *wyrażenie warunkowe*; jeśli jest ono równe 0 - praca pętli jest przerywana;
3. jeśli *wyrażenie warunkowe* jest *różne od zera*, wówczas wykonywane zostaną instrukcje będące *treścią pętli*;
4. po wykonaniu treści pętli wykonana zostanie instrukcja *instr_krok*, po czym następuje powrót do p. 2.

Uwagi:

- *instr_ini* nie musi być tylko jedną instrukcją. *Może być ich kilka, wówczas muszą być one oddzielone przecinkami*. Podobnie jest w przypadku instrukcji *instr_krok*.
- Wyszczególnione elementy: *instr_ini*, *wyraz_warunkowe*, *instr_krok* nie muszą wystąpić. Dowolny z nich można opuścić, zachowując jednak średnik oddzielający go od sąsiada. Opuszczenie wyrażenia warunkowego jest traktowane tak, jakby stało tam wyrażenie *zawsze prawdziwe*.

Zadanie 5

- Określ, co otrzymamy w wyniku działania następujących instrukcji:

```
for (int i=1; i<=10; i++);
cout<<"*";
cout<<endl;
```

Zadanie 6

- Niech będzie dany następujący program:

```
main()
{
    int i, n, sum;
    sum = 0;
    for ( i=0; i <4; i++)
    {
        cout <<"Podaj liczbe calkowita : ";
        cin>> n;
        sum+=n;
    }
    cout <<"Suma :  " << sum;
    return 0;
}
```

Zamień instrukcję for na

1. while
2. do … while

Zadanie 7

Napisz programy (stosując podział na funkcje), których efekty działania będą następujące:

a).

```
*
**
***
****
```

b).

```
****
***
**
*
```

c).

```
****
 ***
  **
   *
```

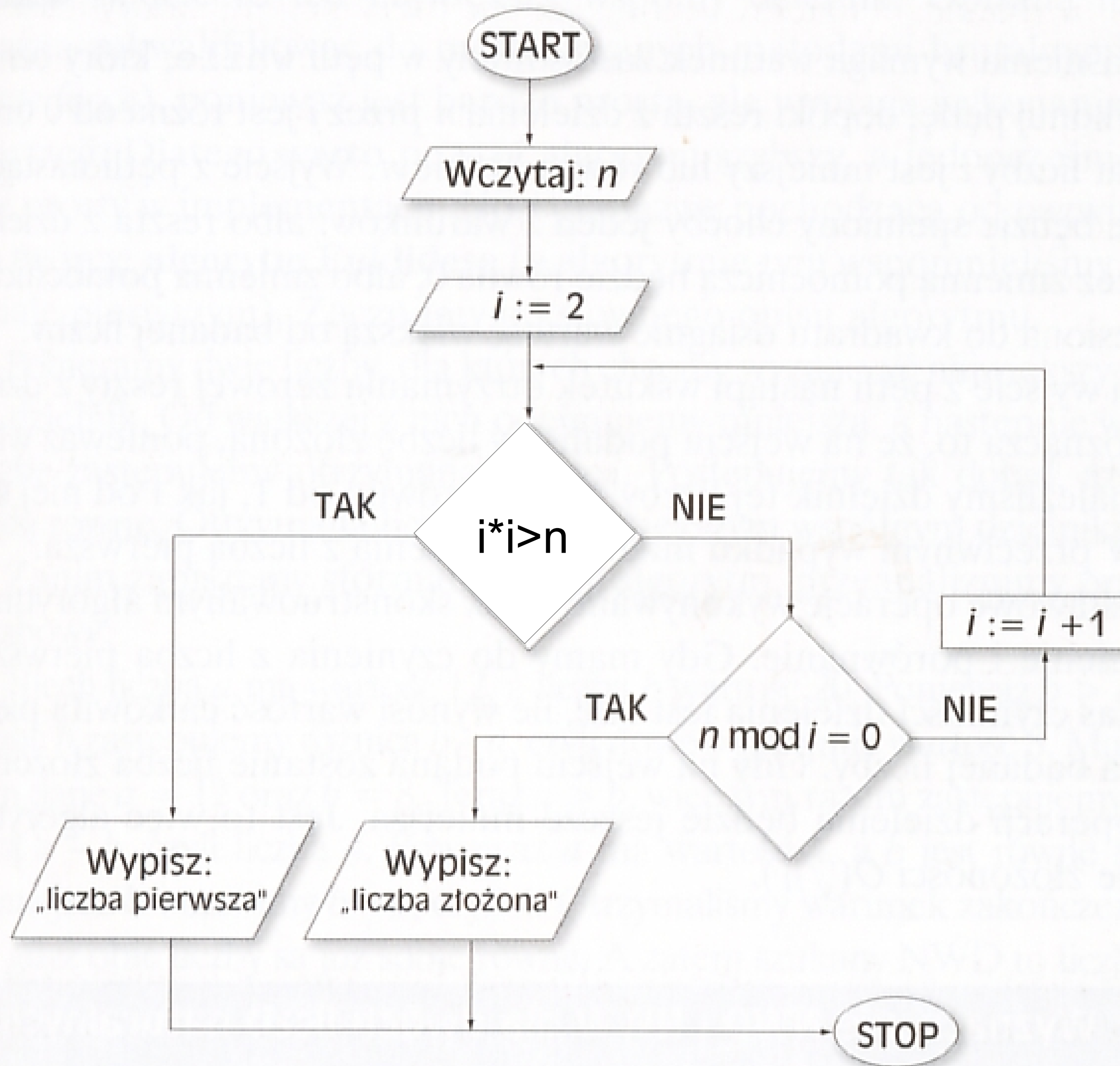
d).

```
   *
  **
 ***
****
```

e).

```
  *
 ***
*****
```

Liczbę wyświetlanych wierszy podaje użytkownik.



Ryc. 4.2. Schemat blokowy algorytmu badającego, czy podana na wejściu liczba n jest liczbą pierwszą.

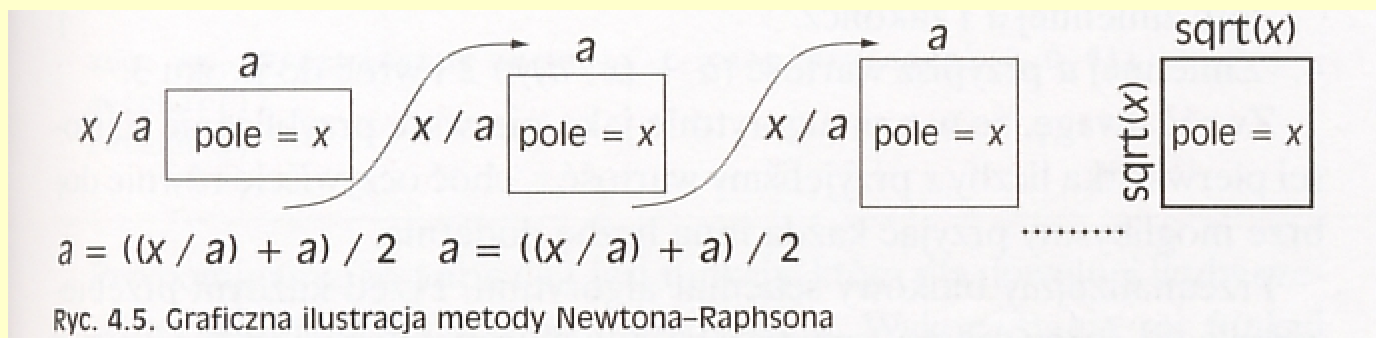
Zadanie 8

Napisz program, który wyznaczy wszystkie liczby pierwsze z przedziału 1 do n. Wartość n pobieramy od użytkownika.

4.4. Obliczanie pierwiastka kwadratowego z liczby dodatniej – metoda Newtona–Raphsona

Pamiętamy, że kwadrat, którego pole wynosi x jednostek, ma bok o długości $\sqrt{x}$ jednostek. Nasz problem sprowadza się więc do znajdowania długości boku kwadratu o polu x. Dowolną liczbę a większą od zera przyjmijmy za jeden z boków prostokąta o tym samym polu, co kwadrat o boku $\sqrt{x}$. Aby zachować pole takiego prostokąta równe x, drugi bok musi mieć długość x / a (ponieważ $a \cdot x / a = x$).

Jeżeli boki prostokąta nie są sobie równe, rozważamy następny prostokąt, którego jeden z boków jest średnią arytmetyczną długości boków poprzedniego prostokąta, czyli $a_1 = (a + x / a) / 2$, drugi bok ma więc teraz długość x / a_1 (ryc. 4.5).



Ryc. 4.5. Graficzna ilustracja metody Newtona–Raphsona

Na wejściu podajemy liczbę, z której chcemy wyznaczyć pierwiastek, oraz dokładność, z jaką chcemy uzyskać wynik. Podsumowując wcześniejsze rozważania, możemy napisać wzór będący kluczem metody Newtona–Raphsona:

$a_n = \frac{1}{2} \cdot \left(a_{n-1} + \frac{x}{a_{n-1}} \right)$, gdzie a_n jest kolejnym przybliżeniem pierwiastka z x (a_1 będziemy najczęściej przyjmować jako x).

Specyfikacja problemu algorytmicznego

Problem algorytmiczny: Obliczenie przybliżonej wartości pierwiastka kwadratowego liczby dodatniej

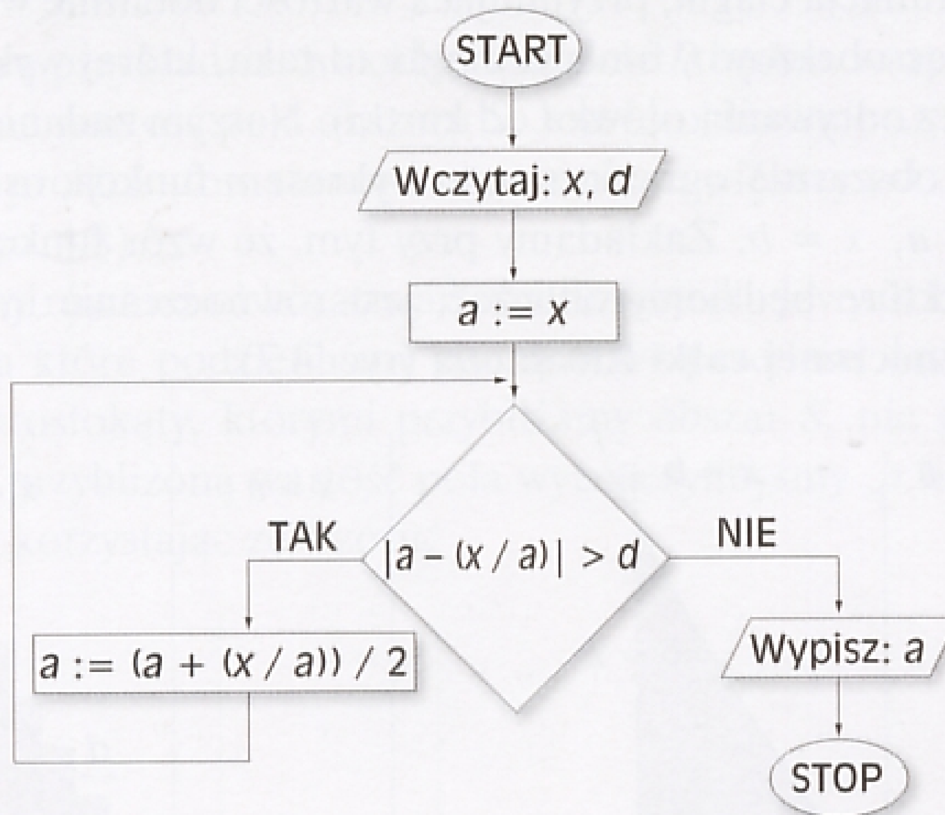
Dane wejściowe: $x \in R_+$ – liczba, z której pierwiastek obliczamy

$d \in R_+$ – dokładność wyznaczenia pierwiastka

Dane wyjściowe: $a \in R_+$ – przybliżona wartość pierwiastka

Zanim napiszemy program realizujący to zadanie, stwórzmy algorytm za pomocą listy kroków:

1. Wczytaj x, wczytaj d.
2. Zmiennej a przypisz wartość x.
3. Jeśli $|a - (x / a)|$ nie jest większe od wartości zmiennej d, wypisz wartość zmiennej a i zakończ.
4. Zmiennej a przypisz wartość $(a + (x / a)) / 2$ i wróć do kroku 3.



Ryc. 4.6. Schemat blokowy algorytmu wyznaczającego metodą Newtona–Raphsona pierwiastek kwadratowy z liczby x

fabs(k) jest funkcje, która dla dowolnej liczby rzeczywistej wyznacza jej wartośc bezwzględną

Funkcja znajduje się w bibliotece cmath, którą dołączamy:

#include<cmath>

Podprogramy

- Jedną z najważniejszych cech nowoczesnych języków programowania jest to, że można w nich posługiwać się podprogramami.
- Jeśli napiszemy podprogram realizujący np. operację obliczenia pola koła na podstawie zadanego promienia - to tak, jakbyśmy język programowania wyposażyli w nową instrukcję umiejącą właśnie to obliczać.
- Podprogram, który jako wynik zwraca pojedynczą wartość, nazywamy funkcją.
- Podprogram, który nie zwraca żadnej wartości nazywamy procedurą.
- W języku C++ wszystkie podprogramy nazywane są funkcjami.

Pojęcie funkcji

- Funkcja jest pewną wyróżnioną częścią programu, realizującą pewne ściśle określone zadanie.
- Program w języku C++ składa się ze zbioru funkcji.
- Ponadto, może on korzystać z funkcji napisanych przez:
  - twórców systemu operacyjnego,
  - twórców kompilatora,
  - także inne osoby.
- Funkcje umieszczone są w specjalnych plikach nazywanych bibliotekami.

Pojęcie funkcji w języku C++

Funkcja w C++ stanowi analogię do funkcji w matematyce:

- otrzymuje pewne parametry (np. liczby, zbiory, itp),
- wykonuje na nich pewną operację,
- zwraca wynik swojego działania (np. liczbę).

Deklaracja , definicja funkcji

- Funkcja ma swoją nazwę, która ją identyfikuje.
- Wszelkie nazwy - przed pierwszym odwołaniem się do nich - muszą być zadeklarowane.
- Przed odwołaniem się do nazwy wymagana jest jej deklaracja.
  Deklaracja, ale niekoniecznie definicja.
- Sama funkcja może być zdefiniowana później, nawet w zupełnie innym pliku.
- Zdefiniować funkcję, to znaczy napisać jej treść.

Wywołanie funkcji

- Wywołanie funkcji, to napisanie jej nazwy wraz z listą argumentów przesyłanych do funkcji, ujętych w nawiasy okrągłe.
- Ponieważ funkcja zwraca wartość, przypisujemy ją do zmiennej.

Budowa funkcji

Funkcja składa się z nagłówka i ciała.

Nagłówek ma postać:

<typ_wartości> <nazwa_funkcji> ([parametry_formalne])

Ciało funkcji składa się z dowolnej ilości deklaracji i instrukcji zamkniętych w nawiasach klamrowych.

Przykłady deklaracji funkcji

```
float kwadrat (int);
void fun (int stopien, char znak, int nachlenie);
int przypadek (void);
char znak_x( );
void funk (...);
```

- *kwadrat* jest funkcją, wywoływaną z jednym argumentem typu int, która zwraca wartość typu float;
- *fun* jest funkcją wywoływaną z 3 argumentami typu: int, char, int, która nie zwraca żadnej wartości. Słowo void służy tu właśnie do zaznaczenia tego faktu;
- *przypadek* jest funkcją, która wywoływana jest bez żadnego argumentu, a która zwraca wartość typu int;
- *znak_x* to funkcja, która wywoływana jest bez żadnych argumentów, a która zwraca wartość typu char;
- *funk* to funkcja, którą wywołuje się z bliżej nieokreślonymi jeszcze argumentami, a która nie zwraca żadnej wartości.

Deklaracje funkcji

Deklaracja :

f( )

oznacza w C++ brak jakichkolwiek argumentów, czyli to samo co:

f (void)

Deklaracje funkcji

Nazwy argumentów umieszczone w nawiasach przedstawionych deklaracji są nieistotne dla kompilatora i można je pominąć. Nazwy ale nie typy argumentów. Dlatego deklarację funkcji:

void *fun* (int *stopien*, char *znak*, int *nachylenie*);

można napisać także jako:

void *fun* (int, char, int);

W deklaracji powiadamiamy kompilator o liczbie i typie argumentów.

Zwracanie wyniku przez funkcję

```
long potega (int stopien, long liczba)
{
   long wynik = liczba;

   for (int i =1; i < stopien; i++)
         wynik = wynik * liczba;

   return wynik;
}
```

Zwracanie wyniku przez funkcję

- Zwracanie wartości funkcji odbywa się przez instrukcję return. Przykładowo:

return wynik;

return (wynik + 6);

return 12.34;

- Jeśli po słowie return stoi wyrażenie, to najpierw obliczana jest wartość tego wyrażenia, a następnie wynik jest przedmiotem zwrotu.
- Należy zwrócić uwagę, że zadeklarowaliśmy funkcje potega typu long. Czy jest błędem napisanie instrukcji return 12.34? Nie zawsze. Nastąpi bowiem próba niejawnej konwersji typu. W naszym przypadku będzie to konwersja typu zmiennoprzecinkowego na typ long, w wyniku której funkcja zwróci wartość 12. *Nie zawsze jednak taka konwersja może się odbyć.*

Zwracanie wyniku przez funkcję

- Jeśli funkcja została zadeklarowana jako zwracająca typ void, to próba użycia jej w wyrażeniu spowoduje błąd, który zostanie zasygnalizowany. Również, gdybyśmy wewnątrz definicji takiej funkcji obok słowa return napisali wyrażenie, to kompilator wykryje błąd.
- Odwrotnie, jeśli zadeklarowaliśmy, że funkcja ma coś zwracać, a przy słowie return stoi sam średnik, kompilator uzna to za błąd.
- Jeśli w obrębie funkcji definiujemy jakieś zmienne, to są one przechowywane w podręcznej pamięci nazywanej *stosem*.

Argumenty domyślne

- Deklarując funkcję w sposób następujący:

```
void temperatura (float stopnie, int skala = 0);
```

- Oznacza to, że parametr skala ma wartość domyślną zero. Wówczas można wywołać funkcję w sposób następujący:

```
temperatura (66.7);
```

Drugi parametr ma wartość domyślną 0.

Argumenty domyślne

- O tym, że argument jest domyślny, informujemy kompilator raz, w deklaracji funkcji. Jeśli definicja funkcji występuje później, to w definicji już się tego nie powtarza.
- Jeśli chcemy, by funkcja miała kilka argumentów domyślnych, to argumenty takie muszą być na końcu listy:

```
int multi (int x, float m, int a = 4, float y = 6.55, int k = 10);
```

- Ostatnie argumenty jako domyślne, mogą być więc w niektórych przypadkach opuszczone. Oto przykłady wywołań tej funkcji:

```
multi (2, 3.14);            // a = 4, y = 6.55, k = 10
multi(2, 3.14, 7);          //a = 7, y = 6.55, k = 10
multi(2, 3.14, 7, 0.3);     // a = 7, y = 0.3, k = 10
multi (2, 3.14, 7, 0,3, 5);         //a = 7, y = 0.3, k = 5
```

- Nie jest możliwe opuszczenie domyślnego argumentu a lub y, a umieszczenie argumentu k. Zatem wywołanie typu:

```
multi (2, 3.14, , 5);
```

jest traktowane jako błąd.

Obiekty globalne

- Obiekt zdefiniowany na zewnątrz wszystkich funkcji ma zasięg *globalny*. Oznacza to, że jest on dostępny wewnątrz wszystkich funkcji znajdujących się w tym pliku. Z jednym zastrzeżeniem - jest znany dopiero od linijki, w której nastąpiła jego deklaracja, w dół do końca programu.
- Oczywiście praktyka jest taka, że deklaracje umieszcza się na samym początku pliku, dzięki czemu obiekt jest dostępny wszystkim funkcjom z tego pliku.

Zadanie 2

- Napisać funkcję, która będzie otrzymywać jako argumenty dwie liczby rzeczywiste i jeden znak, a następnie wykonywać na dwóch otrzymanych liczbach operację arytmetyczną odpowiadającą znakowi i zwracać jej wynik.
  Należy zrealizować następujące operacje:
  - dodawanie (+),
  - odejmowanie (-),
    mnożenie (*),
  - dzielenie (/).

  Każdy inny znak ma powodować wykonanie dodawania.

- Opracować program (funkcję main) wykorzystujący przygotowaną funkcję.

Zadanie 3

- Napisać dwie funkcje, każdą z argumentem i wynikiem całkowitym. Pierwsza funkcja ma sprawdzać, czy argument jest podzielny przez 2, a druga zaś, czy jest podzielny przez 3.
- Zastosować te funkcje w programie, który będzie czytał z wejścia liczbę całkowitą i stwierdzał, czy jest podzielna przez 2, przez 3 i przez 6.

Zadanie 1

Napisz funkcję określającą dla pary liczb całkowitych, czy pierwsza z liczb jest wielokrotnością drugiej.

Zadanie 2

Napisz funkcję, która pozwoli na zamianę wartości przypisanych do dwóch zmiennych.

Przygotuj warianty funkcji dla liczb całkowitych, rzeczywistych oraz wartości logicznych.

Przykład

Przed zamianą int a=10; int b=15;

Po zamianie a=15; b=10;

Zadanie 3

- Przygotuj funkcję pozwalającą na wyznaczenie pierwiastków rzeczywistych równania kwadratowego.
- Dodatkowo przygotuj przykładowy program przedstawiający zastosowanie przygotowanej funkcji.

Zadanie 5

- Największy wspólny dzielnik x i y jest największą liczbą całkowitą przez którą x i y dzielą się bez reszty.
- Napisz funkcję , która będzie pobierała dwa argumenty i zwracała NWD

Zadanie 6

- Wiedząc że 1 litr = 0,264179 galona.
- Napisz program, który pobierze od użytkownika ilość zużytego przez samochód paliwa w litrach oraz liczbę przejechanych kilometrów, a następnie wyświetli, ile kilometrów ten samochód przejechałby na jednym galonie.
- W programie zdefiniuj funkcję, która pozwoli na obliczenie ilości kilometrów przejeżdżanych na jednym galonie paliwa

Zadanie 5

- Największy wspólny dzielnik x i y jest największą liczbą całkowitą przez którą x i y dzielą się bez reszty.
- Napisz funkcję , która będzie pobierała dwa argumenty i zwracała NWD

5. Jaki będzie wynik wykonania podanego poniżej programu

```
int main()
{
        int n = 1;
        do
        {
                if (n%2 == 0)
                {
                        cout << n << " – jest liczbą parzystą.\n";
                        if (n%3 == 0)
                                cout << n << " – jest podzielne przez 3" << endl;
                }
                n +=1;
        }
        while (n <=10);

        return 0;

}
```

6. Program z zadania 5 zapisz używając pętli for

Przekazywanie argumentów do funkcji przez wartość

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

void alarm (int stopien, int wyjscie);

/**********************************************************/
int main()
{
  int a, m;
  alarm (1, 10);
  cout << "\nPodaj stopień zagrożenia: ";
  cin >> a;
  cout << "Podaj numer wyjścia: ";
  cin >> m;
  cout << endl;
  alarm (a, m);
  return 0;
}
/**********************************************************/

void alarm (int stopien, int wyjscie)
{
 cout << "Alarm " << stopien
          << " stopnia\n"
          << "Skierować się do wyjścia nr "
          << wyjscie << endl;
}
```

stopien, wyjscie - **parametry formalne**

1, 10, a, m - **parametry aktualne.**

Argumenty przesłane do funkcji w rozpatrywanym przykładzie są tylko kopiami. Jakiekolwiek działanie na nich nie dotyczy oryginału.

Przekazywanie argumentów do funkcji przez wartość

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

void zwieksz (int formalny);
int main ()
{
   int aktu = 2;
   cout << "Przed wywołaniem funkcji, aktu = "<< aktu <<
            endl;
   zwieksz (aktu);
   cout << "Po wywołaniu funkcji, aktu = "<< aktu << endl;
   return 0;
}
/*****************************************************
*******/
void zwieksz (int formalny)
{
  formalny += 1000; // zwiększenie liczby o 1000
  cout << "Funkcja zwieksz modyfikuje argument formalny\n\t"
            << " i teraz argument formalny = "
            << formalny << endl;
}
```

Po wykonaniu programu otrzymamy:

Przed wywołaniem funkcji, aktu = 2

Funkcja zwieksz modyfikuje argument formalny

i teraz argument formalny = 1002

Po wywołaniu funkcji, aktu = 2

Przekazywanie argumentów przez referencję

```
#include <iostream>
using namespace std;
void zer ( int wart, int &ref);
          //1
/************************************************************/
int main ()
{
 int a = 44,
 b = 77;
 cout << "Przed wywołaniem funkcji: zer \n";
 cout << "a = " << a << ", b = " << b << endl;
 zer (a, b);                                              //2
 cout << "Po powrocie z funkcji: zer \n";
 cout << "a = " << a << ", b = " << b << endl;
          //7
 return 0;
}
/************************************************************/
void zer (int wart, int &ref)
{
          cout      << "\tW funkcji zer przed zerowaniem \n";
          cout      << "\twart = " << wart << ", ref = "
                    << ref << endl;                       //3
          wart = 0;
          ref = 0;                                        //4
          cout      << "\tW funkcji zer po zerowaniu \n";
          cout      << "\twart = " << wart << ", ref = "
                    << ref << endl;                       //5
}
                                                          //6
```

//1 Funkcja zer zależy od dwóch parametrów: parametru *wart* przesyłanego przez wartość i parametru *ref* przesyłanego przez referencję;

//2 Funkcja zer w bloku głównym jest wywoływana z parametrami aktualnymi a i b;

//3 Wewnątrz funkcji zer wypisywane są wartości parametrów formalnych *wart* i *ref*.

//4 Następnie w bloku funkcji następuje zmiana wartości parametrów *wart* i *ref*.

//5 Następuje wypisywanie wartości parametrów *wart* i *ref*.

//6 Działanie funkcji zostaje zakończone. Ponieważ funkcja jest typu void, nie musimy na końcu bloku funkcji pisać instrukcji return (możemy napisać return;).

//7 Po powrocie z funkcji, będąc w bloku głównym main wypisujemy na ekranie wartości zmiennych a i b. Zmienna, którą funkcja odebrała przez referencje została zmodyfikowana.

Przekazywanie argumentów przez referencję

- Zmiana wartości zmiennej b nastąpiła dlatego, że do funkcji *zamiast liczby 77* (wartość zmiennej b) został wysłany *adres zmiennej* b w pamięci komputera. Ten adres funkcja odebrała i na stosie stworzyła referencję, czyli komórce pamięci o przesłanym adresie nadano nazwę ref.
- W //4 do obiektu o nazwie ref wpisano zero. Skoro ref jest nazwą obiektu b, to znaczy, ze odbyło się to na obiekcie b.
- Ponieważ po zakończeniu działania funkcji zmienne lokalne zostają niszczone, zobaczmy, co zostało zlikwidowane:
  - będąca na stosie kopia zmiennej a (kopia początkowo miała wartość 44, a potem 0);
  - drugi argument przesyłany był przez referencję, więc na stosie mieliśmy zanotowany adres tego obiektu, który nazwaliśmy ref. Ten adres został zlikwidowany.
- Wniosek: przesłanie argumentów przez referencję pozwala funkcji na modyfikowanie zmiennych znajdujących się poza tą funkcją.

Zakres ważności nazw deklarowanych wewnątrz funkcji

Zakres ważności nazw deklarowanych w obrębie funkcji *ogranicza się tylko do bloku tej funkcji*.

Nie można więc spoza funkcji za pomocą danej nazwy próbować dotrzeć do zmiennej będącej w obrębie funkcji.

Uwagi

- Należy pamiętać, że *zmienne automatyczne nie są zerowane* w chwili definicji. Jeśli nie zainicjowaliśmy ich jakąś wartością, to przechowują one wartości przypadkowe. Dzieje się tak dlatego, że zmienne automatyczne przechowywane są na stosie. Przydziela się im wymagany dla danego obiektu obszar - i nic więcej. Nie inicjalizuje się tego obszaru.
- *Zmienne globalne* zakładane są w normalnym obszarze pamięci. Ten obszar przed uruchomieniem programu jest zerowany, zatem zmienna globalna, jeśli jej nie zainicjowaliśmy ma wartość zero.
- Z obiektem automatycznym wiąże się słowo kluczowe auto stawiane przed definicją obiektu wewnątrz jakiegoś bloku. Jest ono jednak rzadko używane, gdyż obiekty tak definiowane są automatyczne przez domniemanie. Zatem, jeśli w bloku funkcji definiujemy obiekt:

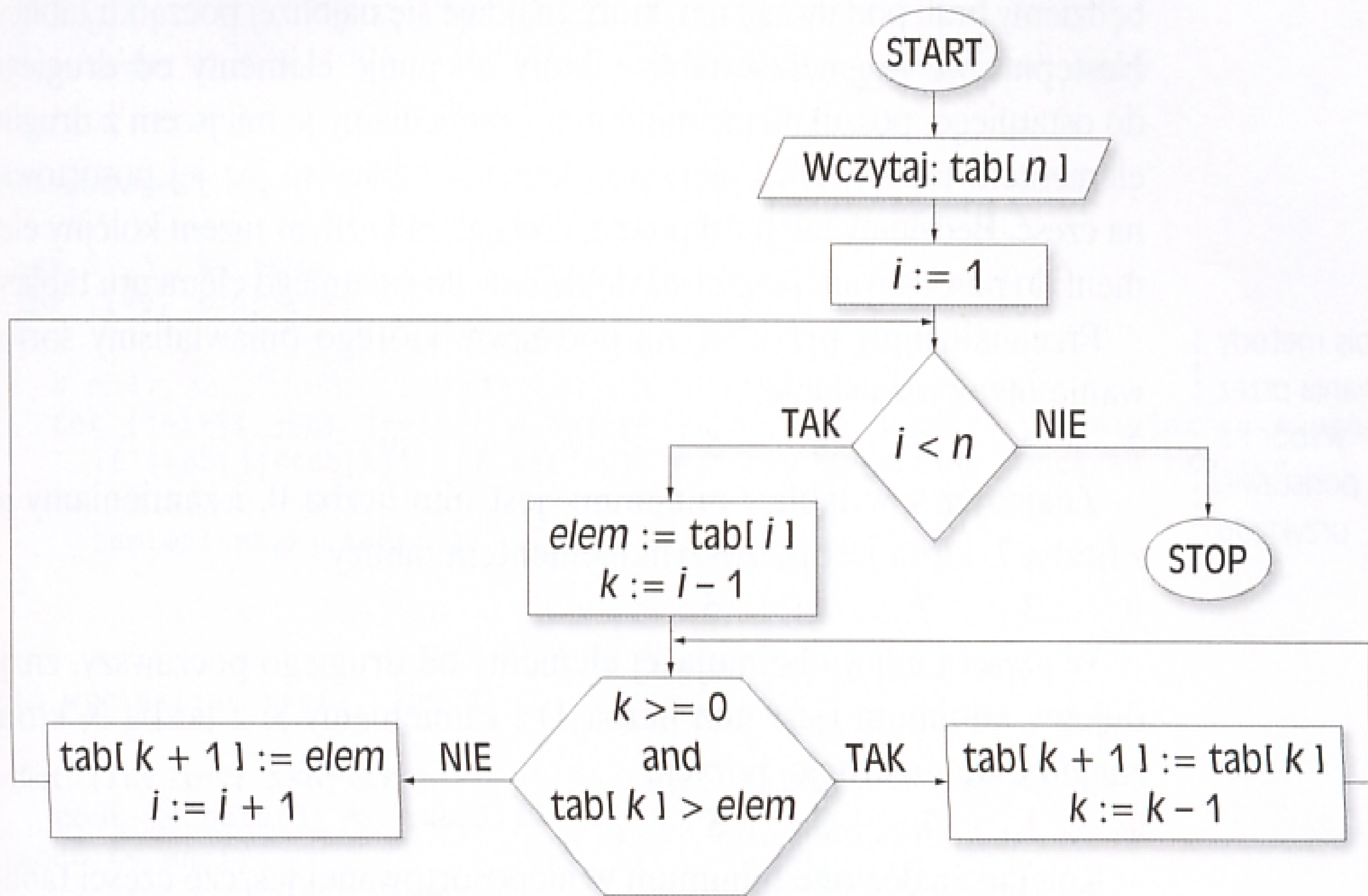
  auto int m;

  to jest to równoważne definicji int m;

Sortowanie przez wstawianie

Specyfikacja problemu algorytmicznego i opis użytych zmiennych

Problem algorytmiczny:	Ustawienie elementów tablicy w porządku niemalejącym
Dane wejściowe:	Tablica `tab[n]`, gdzie `tab[i]` $\in C$, $i \in \{0, ..., n-1\}$, $n > 1$
Dane wyjściowe:	Posortowana tablica
Zmienne pomocnicze:	$i, k \in N$ – zmienne licznikowe, `elem` $\in C$ – zmienna pomocnicza, element aktualnie ustawiany w sortowanej tablicy



Ryc. 5.12. Schemat blokowy algorytmu sortowania przez wstawianie

Sortowanie przez wybór

Algorytm porządkowania przez wybór

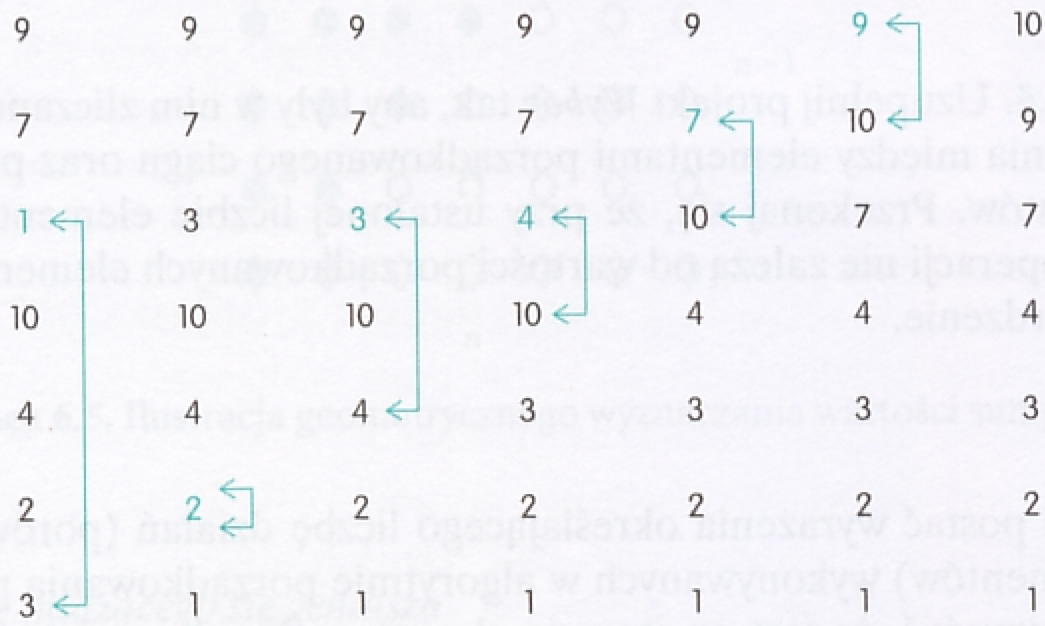
Dane: Liczba naturalna n i ciąg n liczb $x_1, x_2, \ldots, x_n$.

Wynik: Uporządkowanie tego ciągu liczb od najmniejszej do największej.

Krok 1. Dla $i = 1, 2, \ldots, n-1$ wykonaj kroki 2 i 3.

Krok 2. Znajdź k takie, że x_k jest najmniejszym elementem w podciągu $x_i, \ldots, x_n$.

Krok 3. Zamień miejscami elementy x_i oraz x_k. ■



Rysunek 6.3. Przykład działania algorytmu porządkowania przez wybór