

Piotr Broda  
Danuta Smołucha

# Informatyka

**część 1**

**PODRĘCZNIK**  
dla liceum ogólnokształcącego



Gdynia 2008

Projekt okładki: *Krzysztof Godlewski*  
Redaktor prowadzący: *Sebastian Przybyszewski*  
Redakcja językowa: *Elżbieta Pałasz*  
Redakcja graficzna i sktad: *Zespół*  
Korekta techniczna: *Agata Bugiel, Magdalena Ktniotek*

Płyta CD dołączona do podręcznika przygotowali: *Marcin Płocki* (projekt i wykonanie),  
*Sebastian Przybyszewski* (wybór i opracowanie).

Podręcznik dopuszczony do użytku szkolnego przez ministra właściwego do spraw oświaty i wychowania i wpisany do wykazu podręczników przeznaczonych do kształcenia ogólnego do nauczania informatyki na poziomie liceum ogólnokształcącego na podstawie opinii rzeczników: mgr. Włodzimierza Kruszwickiego, prof. dr. hab. Jana Madeya, dr. Leszka Rudaka, dr. Wojciecha Kaliszewskiego.

**Numer dopuszczenia: 146/05**

© Copyright by Wydawnictwo Pedagogiczne OPERON Sp. z o.o. & Piotr Broda, Danuta Smołucha  
Gdynia 2005  
Wszelkie prawa zastrzeżone. Kopiowanie w całości lub we fragmentach bez zgody wydawcy zabronione.

Zastrzeżonych nazw firm i produktów użyto w tym podręczniku wyłącznie w celu identyfikacji.  
7-8/XIII

Wydawca:  
Wydawnictwo Pedagogiczne OPERON Sp. z o.o.  
81-212 Gdynia, ul. Hutnicza 3  
tel. centrali 058 679 00 00  
e-mail: [info@operon.pl](mailto:info@operon.pl)  
<http://www.operon.pl>

**ISBN 978-83-7461-736-9**

# Spis treści

Od Wydawcy.....	6
<b>1. Algorytmy.....</b>	<b>7</b>
1.1. Definicja algorytmu .....	7
1.2. Sposoby zapisu algorytmu .....	9
1.3. Specyfikacja problemu algorytmicznego	11
1.4. Opis stosowanych operatorów .....	13
1.5. Instrukcja warunkowa - algorytmy rozgałęzione.....	15
1.6. Instrukcja iteracji.....	18
1.7. Poprawność algorytmu .....	23
1.8. Złożoność obliczeniowa algorytmu .....	24
1.9. Realizacja algorytmów w arkuszu kalkulacyjnym .....	28
<b>2. Wstęp do programowania w języku C++.....</b>	<b>35</b>
2.1. Reprezentacja danych w komputerze - podstawowe operacje w systemie binarnym .....	35
2.2. Proces wytwarzania programu.....	38
2.3. Podstawowe typy danych .....	41
2.4. Budowa programu.....	42
2.5. Instrukcje sterujące.....	50
2.5.1. Instrukcja warunkowa if ... else .....	51
2.5.2. Instrukcja wyboru switch .....	56
2.5.3. Instrukcja pętli for.....	59
2.5.4. Instrukcja pętli while .....	61
2.5.5. Instrukcja pętli do ... while .....	62
2.5.6. Instrukcje break i continue sterujące wykonaniem pętli.....	64
<b>3. Funkcje w C++.....</b>	<b>69</b>
3.1. Definiowanie i wywoływanie funkcji .....	69
3.2. Zakres ważności zmiennych .....	71
3.3. Sposoby przekazywania parametrów funkcji .....	73
<b>4. Implementacja klasycznych algorytmów iteracyjnych</b>	<b>89</b>
4.1. Znajdowanie najmniejszego lub największego elementu w ciągu liczb ,	89
4.2. Liczba pierwsza czy złożona - algorytm testujący .....	91
4.3. Wyznaczanie NWD i NWW dla dwóch liczb naturalnych (algorytm Euklidesa).....	94
4.4. Obliczanie pierwiastka kwadratowego z liczby dodatniej - metoda Newtona-Raphsona .....	99

Spis treści

4.5. Obliczanie pola obszaru ograniczonego wykresem funkcji .....	102
4.6. Znajdowanie przybliżonej wartości miejsca zerowego funkcji ciągłej metodą połowienia przedziałów .....	106
4.7. Metody Monte Carlo .....	109
4.7.1. Obliczanie wartości liczby % metodą Monte Carlo .....	109
4.7.2. Modelowanie ruchów Browna .....	113
 5. Tablice danych - przykłady i wykorzystanie w algorytmach .....	117
5.1. Charakterystyka tablic .....	117
5.1.1. Tablice jednowymiarowe .....	118
5.1.2. Tablice wielowymiarowe .....	122
5.2. Klasyczne algorytmy działające na tablicach .....	128
5.2.1. Przeszukiwanie liniowe tablicy jednowymiarowej .....	128
5.2.2. Przeszukiwanie liniowe tablicy jednowymiarowej z wartownikiem .....	130
5.2.3. Poszukiwanie elementu maksymalnego (minimalnego) w tablicy .....	132
5.2.4. Zapisywanie liczb w różnych systemach liczbowych .....	134
5.2.5. Sito Eratostenesa .....	140
5.3. Sortowanie tablicy .....	142
5.3.1. Sortowanie bąbelkowe .....	142
5.3.2. Sortowanie przez wstawianie .....	145
5.3.3. Sortowanie przez wybór .....	148
5.4. Praktyczne wykorzystanie tablicy dwuwymiarowej .....	150
5.5. Tablice tekstowe .....	152
5.5.1. Modyfikacje dokonywane na tekście .....	155
5.5.2. Szyfr Cezara .....	156
5.6. Rozwiązywanie układów równań metodą eliminacji Gaussa .....	159
 6. Rekurencja .....	167
6.1. Funkcje rekurencyjne w informatyce .....	167
6.1.1. Silnia liczby naturalnej .....	167
6.1.2. Potęga o wykładniku naturalnym liczby rzeczywistej .....	170
6.1.3. Obliczanie wartości wyrazów ciągów zdefiniowanych rekurencyjnie .....	171
6.1.4. Ciąg Fibonacciego .....	172
6.1.5. Schemat Homera obliczania wartości wielomianu .....	174
6.1.6. Wieże Hanoi .....	175
6.1.7. Zamiana liczby na postać dwójkową - rozwiązywanie rekurencyjne .....	178
6.1.8. Rekurencyjne odwracanie wprowadzonego ciągu znaków .....	179
6.2. Metoda „dziel i zwyciężaj” .....	180
6.2.1. Przeszukiwanie binarne .....	181
6.2.2. Sortowanie tablicy przez scalanie .....	182
6.2.3. Sortowanie szybkie .....	187
6.3. Problem skoczka szachowego i problem ośmiu hetmanów - algorytmy z powrotami .....	188

<b>7. Struktury - typ definiowany przez użytkownika . . . . .</b>	<b>195</b>
7.1. Definicja struktury.....!	195
7.2. Tablice o elementach typu strukturalnego	200
7.3. Struktury jako argumenty funkcji.....	202
<b>8. Typ wskaźnikowy - zastosowania.....</b>	<b>207</b>
8.1. Deklaracja zmiennej wskaźnikowej i podstawowe operacje na wskaźnikach.....	207
8.2. Przekazywanie argumentów funkcji przez wskaźnik	212
8.3. Zastosowanie wskaźników w tablicach.....	213
<b>9. Zmienne i struktury dynamiczne.....</b>	<b>219</b>
9.1. Tablica dynamiczna jednowymiarowa	219
9.2. Tablica dynamiczna dwuwymiarowa	222
9.3. Lista jednokierunkowa - tworzenie listy i wprowadzanie do niej elementów.....	227
9.4. Lista jednokierunkowa - usuwanie listy z pamięci.....	234
9.5. Lista dwukierunkowa	236
<b>10. Zapis do plików i odczyt z plików</b>	<b>243</b>
10.1. Zapis do pliku.....	243
10.2. Odczyt danych z pliku	246
<b>11. Wstęp do programowania obiektowego</b>	<b>255</b>
11.1. Obiekt a klasa.....	255
11.2. Hermetyzacja - rola metod publicznych . . . . .	257
11.3. Rola konstruktora i destruktora.....	260
Indeks.....	266
Literatura pomocnicza.....	269

## Od Wydawcy

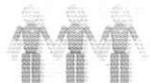
Podręcznik *Informatyka, część I* jest przeznaczony dla uczniów uczących się w liceum ogólnokształcącym. Jest on częścią pakietu edukacyjnego, w którego skład wchodzą: program nauczania, przewodnik metodyczny dla nauczyciela (z płytą CD) i wybrane scenariusze lekcji. Wszystkie te pozycje są ze sobą ścisłe zintegrowane i pozwalają na pełne wykorzystanie nowoczesnych metod dydaktycznych w nauczaniu informatyki.

Dzięki przejrzystej i powtarzalnej strukturze podręcznika przekazywana w nim wiedza jest doskonale uporządkowana. Liczne przykłady ułatwiają zrozumienie omawianych tematów, a utrwaleniu wiadomości służą zamieszczone na końcu każdego podrozdziału pytania kontrolne i ćwiczenia.

Do podręcznika jest dołączona płyta CD, na której znajdują się kompilatory języka C++ (w wersji freeware) oraz kody źródłowe z podręcznika.

Prosimy o nadsyłanie pod adresem Wydawnictwa wszelkich uwag i sugestii. Będą one niezwykle przydatne w pracach nad kolejnymi publikacjami.

Oznaczenia pictogramów:



- przykład



- pytania kontrolne



- ciekawostki

# 1. Algorytmy

Po przeczytaniu tego rozdziału dowiesz się, co to jest algorytm, jak zapisujemy algorytmy i określamy ich złożoność obliczeniową. Poznasz różne przykłady algorytmów, łącznie z ich poprawnym zapisem. Nauczysz się, jak przeprowadzić właściwą analizę zadania wraz z jego specyfikacją, czyli dokładnym opisem wymagań. Na koniec zaprezentujemy ci sposób realizacji algorytmów w arkuszu kalkulacyjnym.

## 1.1. Definicja algorytmu

Często w życiu stajesz przed koniecznością rozwiązania jakiegoś zadania. Na lekcji matematyki musisz na przykład rozwiązać równanie i w tym celu wykonujesz szereg czynności: od określenia i wypisania danych, przez zastosowanie odpowiedniej metody rozwiązywania, aż do uzyskania wyniku i jego sprawdzenia. Pewien ciąg czynności możesz określić do rozwiązania wielu zadań, które nie mają nic wspólnego z matematyką. Aby zrobić na przykład jajecznice, trzeba wziąć jajko, łyżeczkę masła i szczyptę soli. Należy rozgrzać patelnię, rozpuścić na niej przygotowane masło, wbić jajko, dodać sól. Teraz zawartość patelni mieszaj przez 3 minuty, wyłącz palnik - i jajecznica gotowa. W ten sposób zdefiniowany został uporządkowany zbiór czynności, który „rozwiązał zadanie” przygotowania jajecznicy. Jeśli na drugi dzień znów poczujesz apetyt na to danie, naprawdopodobnie będziesz postępować według tego samego schematu, który jest właśnie algorytmem. Definicja algorytmu brzmi:

### Definicja

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

W naszym drugim przykładzie danymi wejściowymi były: jajko, masło, sól, a wynikiem działania - gotowa jajecznica.

### Czy wiesz, że...

Słowo „algorytm” pochodzi od nazwiska Algorismi (Al-Chorezmi)\* - wybitnego arabskiego matematyka i astronoma, żyjącego na przełomie VIII i IX wieku. Dzięki jego pracom upowszechnił się system dziesiętny i wyjaśnione zostało znaczenie zera.



\* Spotyka się też nieco inną pisownię tego nazwiska w niektórych źródłach.

## 1. Algorytmy

Zadania, dla których ustalamy rozwiązania algorytmiczne, często dotyczą danych liczbowych. Jeśli chcemy na przykład policzyć średnią ocen uzyskanych na koniec roku, przyjmiemy metodę konsekwentnego postępowania według schematu: dodamy wartości liczbowe ocen ze wszystkich przedmiotów, a otrzymaną sumę podzielimy przez liczbę przedmiotów. **Danymi wejściowymi** (dane podawane na wejście algorytmu) są tu oceny ze wszystkich przedmiotów, **danymi wyjściowymi** (wyniki działania algorytmu) jest jedna liczba, będąca średnią ocen.

### Definicja

Algorytmy, które wykonują działania matematyczne na danych liczbowych, nazywamy **algorytmami numerycznymi**.

Oczywiście, nie wszystkie zadania dają się rozwiązać za pomocą algorytmu, na przykład nie istnieje przepis na dzieło sztuki. Jest też wiele zadań, których rozwiązanie uzyskamy dzięki szczęśliwemu przypadkowi lub intuicji, ale na pewno nie w wyniku działania jakiegoś algorytmu.

#### Etapy konstruowania algorytmu

Postępowanie przy rozwiązywaniu danego zadania za pomocą algorytmu możemy podzielić na kilka kolejnych etapów:

1. Sformułowanie zadania • ustalamy, jaki problem ma rozwiązywać algorytm.
2. Określenie danych wejściowych - ich typu (w typie określamy, czy dane są na przykład liczbami rzeczywistymi, całkowitymi czy znakami).
3. Określenie wyniku oraz sposobu jego prezentacji.
4. Ustalenie metody wykonania zadania (zauważ, że: piszemy „ustalenie”, a nie „znalezienie”, gdyż możesz znaleźć kilka metod na rozwiązanie zadania, a spośród nich wybierzesz jedną, która według ciebie będzie najbardziej odpowiednia).
5. Zapisanie algorytmu za pomocą wybranej metody.
6. Analiza poprawności rozwiązania.
7. Testowanie rozwiązania dla różnych danych - algorytm musi być uniwersalny, aby służyć do rozwiązywania zadań dla różnych zestawów danych wejściowych.
8. Ocena skuteczności algorytmu (ocenia się tu praktyczną przydatność algorytmu, np. wybiera z kilku możliwych rozwiązań metodę najszybszą lub odrzuca zaproponowane rozwiązanie jako nieefektywne).

Pamiętaj, że komputer to maszyna, która potrafi wykonać tylko jednoznaczne polecenia - nie jest w stanie samodzielnie wybrać właściwej drogi postępowania. Kolejność czynności musi być zatem ściśle określona, a każda możliwa droga algorytmu powinna być w pełni opisana przez odpowiednie czynności. Każde zastosowanie algorytmu dla zestawu tych samych poprawnych danych wejściowych musi prowadzić do uzyskania takich samych wyników.

## 1.2. Sposoby zapisu algorytmu

### Czy wiesz, że...

Jednym z najstarszych algorytmów jest pochodzący z przełomu IV i III wieku przed naszą erą tak zwany algorytm Euklidesa, służący do znajdowania największego wspólnego dzielnika dwóch liczb całkowitych.

### 1.2. Sposoby zapisu algorytmu

Algorytm powinien zostać zapisany w celu dalszej analizy. Do najczęściej używanych zapisów algorytmu należą: lista kroków, pseudojęzyk zwanego też pseudokodem, graficzna prezentacja algorytmu za pomocą schematu blokowego i wreszcie zapis w danym języku programowania.

Elementem charakterystycznym dla **listy kroków** jest numeracja wierszy. Każdy krok opisuje realizację kolejnej czynności i jest zawarty w osobnym wierszu. Oto algorytm podziału odcinka na cztery równe części zapisany w postaci listy kroków:

1. Podziel odcinek na dwie równe części: lewą i prawą.
2. Podziel część lewą na dwie równe części.
3. Podziel część prawą na dwie równe części.
4. **Zakończ.**

Kolejny przykład przedstawia zapis algorytmu znajdującego się średnia arytmetyczną dwóch liczb rzeczywistych podanych na wejściu. Wynik zostaje wypisany na ekranie monitora.

- 1 Wczytaj pierwszą z liczb.
- 2 Wczytaj drugą z liczb.
- Dodaj liczby do siebie.
4. Wynik dodawania podziel przez 2.  
Wypisz otrzymaną wartość.  
Zakończ.

**Pseudojęzyk** jest metodą pośrednią między zapisem za pomocą listy kroków a zapisem w języku programowania. Notacja algorytmów za pomocą pseudojęzyka jest zbliżona do zapisu w którymś z popularnych języków programowania, ale mniej formalna. Poniżej umieściliśmy przykład zapisu algorytmu w pseudojęzyku, pozostawiamy go jednak bez wyjaśnień, ponieważ nie będziemy rozwijać tego sposobu - przejdziemy bezpośrednio do zapisu w języku programowania.

Zapis w pseudojęzyku algorytmu wypisującego wartość bezwzględną liczby\* wygląda następująco:

Początek;  
Rzeczywiste x;  
Jeśli  $x \geq 0$  Wypisz (x);  
**Jeśli**  $x < 0$  Wypisz (-x);  
Koniec.

Zapis algorytmu  
w postaci listy kroków

Zapis algorytmu  
w pseudojęzyku

## 1. Algorytmy

Przedstawione do tej pory przykłady prezentują zapis algorytmów krótkich i nieskomplikowanych, stąd też analizowanie ich zapisu za pomocą listy kroków czy też pseudokodu nie stanowi trudności. Zwrót ponadto uwagi na fakt, że w algorytmach tych instrukcje wykonujemy zawsze w tej samej kolejności, niezależnie od wartości danych wejściowych.

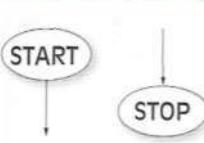
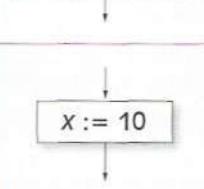
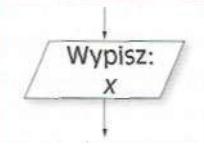
### Definicja

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

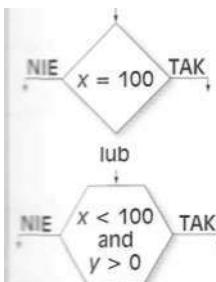
Częściej jednak mamy do czynienia z problemami algorytmicznymi o wiele bardziej złożonymi i ich zapis któryś z podanych już metod, choć możliwy, byłby długi, skomplikowany i mało czytelny. Takie złożone algorytmy najlepiej zapisywać - ze względu na przejrzystość i prostotę - za pomocą schematu blokowego.

**Schemat blokowy** przedstawia algorytm w postaci kilku symboli graficznych. Podaje szczegółowo wszystkie operacje arytmetyczne, logiczne, przesyłania, sterujące i pomocnicze wraz z kolejnością ich wykonywania. Jest on podstawą do napisania programu, znacznie ułatwiającą późniejszy zapis w określonym języku programowania. Schemat blokowy składa się z wielu elementów, wśród których podstawowym jest blok (tab. 1.1).

Zapis algorytmu za pomocą schematu blokowego

wygląd elementu	Opis
	<b>Blok graniczne - początek algorytmu, koniec algorytmu</b> - mają kształt ovalu. Z bloku START wychodzi tylko jedno połączenie, a nie wchodzi do niego żadne. W każdym schemacie blokowym musi być dokładnie jeden blok START. Blok STOP nie ma połączenia wychodzącego, kończy algorytm. Każdy schemat blokowy musi mieć co najmniej jeden blok STOP
	<b>Połączenie</b> pomiędzy blokami - określa kolejność wykonywanych działań lub kierunek przepływu danych (element ten inaczej zwany jest ścieżką sterującą).
	<b>Blok kolekcyjny</b> - łączy kilka różnych dróg algorytmu.
	<b>Blok operacyjny</b> - operacja lub grupa operacji, w których wyniku ulega zmianie wartość zmiennej (wartości zmiennych). W tym przykładzie operacją wykonywaną jest nadanie zmiennej x wartości 10. Bloki operacyjne mają kształt prostokąta. Do bloku operacyjnego wchodzi jedno połączenie i wychodzi z niego również jedno połączenie.
	<b>Bloki wejścia/wyjścia</b> - odpowiadają za wykonanie operacji wprowadzania i wyprowadzania danych, wyników oraz komunikatów. Są w kształcie równoległoboków. Do bloku wejścia lub wyjścia wchodzi jedno połączenie i wychodzi z niego również jedno połączenie.

### 1.3. Specyfikacja problemu algorytmicznego



**Blok decyzyjny** - określa wybór jednej z dwóch możliwych dróg działania. Ma kształt rombu lub sześciokąta. Wychodzą z niego dwa połączenia: jedno podpisane **TAK**, gdy warunek badany wewnątrz jest spełniony, drugie - **NIE**, gdy warunek nie jest spełniony (w 1. bloku pytamy, czy wartość zmiennej  $x$  wynosi 100, w drugim bloku pytamy o prawdziwość warunków dla zmiennych:  $x$  oraz  $y$ ). Wybór kształtu bloku zależy od dtugości zapisu warunku, który chcemy umieścić wewnątrz bloku (zauważ, że warunek złożony łatwiej zapisać w drugim z bloków).

#### ~3b. 1.1. Podstawowe elementy schematu blokowego

W dużych projektach stosuje się również inne bloki, których celem jest zreguły zwiększenie przejrzystości rysunku, łączenie kartek, komentarzy. My jednakże korzystać z podstawowych elementów schematu blokowego.

Aby zapisać algorytm w wybranym języku programowania, trzeba ~ ać zreguły tego języka. W tym miejscu podajemy tylko krótki przykład **zapisu** w językach Pascal i C++ algorytmu wyprowadzającego na ekran - mość bezwzględną liczbę rzeczywistej (tab. 1.2).

Zapis algorytmu  
w języku  
programowania

Język Pascal	Język C++
<pre>if (x&gt;=0) then writeln (x) else writeln (-1*x);</pre>	<pre>if (x&gt;=0) cout &lt;&lt; x else cout &lt;&lt; -1*x;</pre>

Tab. 1.2. Fragmenty programów wyprowadzających na ekran monitora wartość bezwzględną liczby zapisane w dwóch językach programowania: Pascal i C++

## 1.3. Specyfikacja problemu algorytmicznego

Wiesz już, co to jest algorytm, jakie wymogi powinien spełniać i jak go opisać. Brakuje jeszcze jednego czynnika, aby zadanie, które chcesz rozwiązać za pomocą algorytmu, było przedstawione w pełni precyzyjnie. Jeśli

bowiem zamierzasz napisać program, w którym zastosujesz utworzony przez siebie algorytm, to musisz wiedzieć, jakiego typu są dane wejściowe., wyniki, czyli dane wyjścia itp. Wszystkie te informacje powinny zostać umieszczone w **specyfikacji problemu algorytmicznego**.

### Definicja

**Specyfikacją problemu algorytmicznego** nazywamy dokładny opis problemu algorytmicznego, który ma zostać rozwiązany, oraz podanie danych wejściowych i danych wyjściowych wraz z ich typami.

Najczęściej specyfikacja składa się z opisu danych wejściowych wraz ze wszystkimi warunkami, jakie mają one spełniać, oraz danych wyjściowych (wyników) z uwzględnieniem warunków, jakie mają spełniać i ich związku z danymi wejściowymi. Pokażmy to na przykładzie.

### Specyfikacja problemu algorytmicznego

**Problem algorytmiczny:** Obliczanie potęgi liczby naturalnej o wykładowniku naturalnym

**Dane wejściowe:**  $a \in N$  - podstawa potęgi,  $b \in N$  wykładownik potęgi

**Dane wyjściowe:**  $w \in N$  - wartość  $a^b$   
Dane w algorytmie są najczęściej przedstawiane za pomocą liter lub dłuższych nazw, które nazywamy **zmiennymi**.

### Definicja

**Zmienną** nazywamy obiekt występujący w algorytmie, określony przez nazwę i służący do zapamiętywania pewnych danych.

#### Typy zmiennych

Jeśli używamy w algorytmie zmiennej, musimy dokładnie określić, jakiego rodzaju wartości może ona przechowywać - mówimy, że określamy tak zwany **typ zmiennej**. Mogą to być liczby całkowite, liczby rzeczywiste, litery, znaki klawiaturowe. Różne typy będziemy omawiać w kolejnych rozdziałach tego podręcznika.

W czasie działania algorytmu można swobodnie zmieniać wartość przypisaną zmiennej w zakresie jej typu.

**Zmienna pomocnicza** jest zmienną wprowadzoną do zapisu algorytmu w celu umożliwienia jego realizacji. Zmienne pomocnicze służą w algorytmie do pamiętania **danych przejściowych**, czyli danych potrzebnych do działania algorytmu, ale niebędących danymi wejściowymi ani wynikami. Na przykład w algorytmie obliczającym średnią arytmetyczną wielu liczb potrzebujemy dwóch zmiennych pomocniczych - jednej do zapamiętywania ilości podanych liczb, drugiej zaś do zapamiętywania sumy podanych liczb.

Prócz zmiennych w algorytmie możemy używać **stałych**. Mogą to być wartości podane wprost, na przykład liczba 35, albo wartości reprezentowane przez nazwy - podobnie jak zmienne. Wartość stałej w algorytmie nie może ulec zmianie, na przykład: używamy stałej  $n$  mającej wartość 3,14 i oczywiście zależy nam, aby ta wartość nie została przypadkowo zmieniona.

Bezpośrednio po specyfikacji należy umieścić zapis algorytmu - wybór metody zapisu należy do ciebie. My preferujemy zapis za pomocą schematu blokowego, jako najbardziej czytelny dla odbiorcy. Jeśli więc do powyższej specyfikacji dodasz zapis algorytmu sporządzony w postaci na przykład schematu blokowego, to dokumentacja służąca rozwiązaniu problemu byłaby niemal pełna. Należy bowiem jeszcze przeprowadzić ocenę poprawności algorytmu, o czym będzie mowa w dalszej części tego rozdziału.



### Przykład

Zapiszmy specyfikację problemu i schemat blokowy algorytmu służącego do obliczania średniej arytmetycznej dwóch liczb podanych na wejściu i wypisania wyniku na ekranie.

Zacznijmy od sporządzenia specyfikacji problemu.

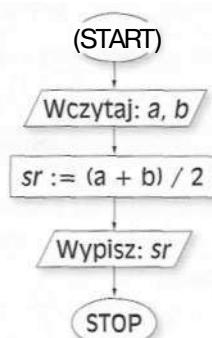
#### Specyfikacja problemu algorytmicznego

**Problem algorytmiczny:** Obliczanie średniej arytmetycznej dwóch liczb rzeczywistych i wypisanie wyniku na ekranie monitora

**Dane wejściowe:**  $a, b \in R$

**Dane wyjściowe:**  $sr \in R$  – średnia liczb  $a, b$

Teraz zapiszmy algorytm w postaci schematu blokowego (ryc. 1.1):



Ryc 1.1. Schemat blokowy algorytmu obliczającego średnią dwóch liczb

Schemat blokowy jest bardzo prosty, dlatego łatwo zauważyc, że po---- poprawne wyniki dla wszystkich danych wejściowych zgodnych ze specyfikacją problemu.

#### 1.4. Opis stosowanych operatorów

Jeśli zmienne lub stałe poprawnie połączymy operatorami zgodnie z regułami matematycznymi, to otrzymamy wyrażenie. Zmienne i stałe występujące w wyrażeniu nazywamy **operandami**. Na przykład:  $a + 4 - b$  jest wyrażeniem składającym się z trzech operandów i dwóch operatorów. Liczba 4 jest operandem stałym,  $a, b$  to operandy zmienne. Operatory występujące w tym wyrażeniu to operatory arytmetyczne. Proponujemy ci zapoznanie się już teraz z zapisem operatorów, gdyż pomyłki w ich stosowaniu są przyczyną wielu błędów popełnianych zwłaszcza przez początkujących programistów.

W tabeli 1.3 zestawiliśmy operatory arytmetyczne występujące w językach Pascal i C++. Zwykle w algorytmach przedstawianych za pomocą schematu blokowego będziemy się posługiwać operatorami zgodnymi z notacją Pascala - jest to tradycyjny sposób zapisywania algorytmów. Podaliśmy też operatory zgodne z językiem C++, ponieważ będziemy ich używać przy zapisie algorytmów w języku programowania.

## 1. Algorytmy

Klasyfikacja operatorów	Pascal	C/C++	Znaczenie
	+	+	dodawanie
	-	-	odejmowanie
	*	*	mnożenie
	/	/	dzielenie
	div	/	dzielenie całkowite, czyli zaokrąglające wynik dzielenia do liczby całkowitej, np. $7/5 = 1$
	mod	%	reszta z dzielenia liczb całkowitych

Tab. 1.3. Operatory arytmetyczne obowiązujące w językach Pascal i C/C++

Kolejna grupa operatorów to **operatory relacji**, czyli operatory badające relacje pomiędzy wyrażeniami (tab. 1.4).

Pascal	C/C++	Znaczenie
=	==	równy
>	>	większy
$\geq$	$\geq$	większy lub równy
<	<	mniejszy
$\leq$	$\leq$	mniejszy lub równy
$\diamond$	$\neq$	różny, czyli nieprawda, że równy

Tab. 1.4. Operatory relacji obowiązujące w językach Pascal i C/C++

Operatory relacji służą do tworzenia **wyrażeń logicznych**. Wyrażenia logiczne charakteryzują się tym, że możemy im przypisać wartość logiczną „prawda” lub „fałsz”. Na przykład stwierdzamy:  $a < 4$  lub  $b > -c + 1$ , albo badamy warunek prawdziwości relacji  $x < y$ . Wówczas dla zadanych wartości zmiennych możemy w sposób jednoznaczny odpowiedzieć: tak lub nie, co odpowiada wartościom logicznym: prawda lub fałsz.

Zapis w notacji Pascal: „ $a = b$ ” zinterpretujemy jako pytanie: Czy wartości argumentów  $a$  i  $b$  są sobie równe? Oczywiście, również tu odpowiedź jest jednoznaczna: albo  $a$  jest równe  $b$ , albo  $a$  nie jest równe  $b$ .

**Operator przypisania (podstawienia)** opisuje operację nadania zmiennej umieszczonej po lewej stronie operatora wartości wyrażenia występującego po prawej stronie tego operatora -jest to instrukcja przypisania. Do oznaczania tej instrukcji przyjmiemy umownie oznaczenie operatora zgodnie ze składnią języka Pascal, czyli znak równości poprzedzony dwukropkiem. W składni C++ ten sam operator jest reprezentowany przez znak równości (tab. 1.5).

Pascal	C/C++ +	Znaczenie
<code>:=</code>	<code>=</code>	operator przypisania

Tab. 1.5. Operator przypisania w językach Pascal i C/C++ +

Jeśli na przykład  $x$  jest zmienną przeznaczoną do przechowywania liczb całkowitych, to operację  $x := 7$  zinterpretujemy jako nadanie zmiennej  $x$  wartości 7. Po prawej stronie operatora przypisania może być umieszczone również wyrażenie algebraiczne. W takim wypadku najpierw będzie obliczone to wyrażenie, a następnie wartość wyrażenia zostanie przypisana zmiennej występującej po lewej stronie operatora. Na przykład zapis  $x := x + 5$  działa w następujący sposób: najpierw jest pobierana wartość zmiennej  $x$ , potem liczona wartość wyrażenia  $x + 5$  i na końcu wartość ta zostaje przypisana zmiennej  $x$ . Jeśli więc przed wykonaniem instrukcji zmieniona  $x$  miała wartość 2, to po jej wykonaniu ma wartość 7.

Po lewej stronie operatora przypisania można umieścić tylko zmienną. Nie może tam być stałej ani wyrażenia.

Kolejną ważną grupą operatorów, za pomocą których budujemy złożone wyrażenia logiczne, są **operatory logiczne** (tab. 1.6):

Pascal	C/C++ +	Znaczenie	zapis matematyczny
and	<code>&amp;&amp;</code>	koniunkcja (iloczyn zdań)	$A$
or	<code>  </code>	alternatywa (suma zdań)	$V$
not	<code>!</code>	negacja (zaprzeczenie zdania)	$\sim$

Tab. 1.6. operatory logiczne stosowane w językach Pascal i C/C++ oraz zapis matematyczny tych operatorów

Dla przykładu, jeśli  $x, y$  są współrzędnymi punktu  $P$  w kartezjańskim układzie współrzędnych, to zapis  $(x > 0) \text{ and } (y > 0)$  oznacza, że punkt  $P(x,y)$  należy do I ćwiartki układu. Jest to koniunkcja dwóch wyrażeń logicznych prostych.

### 1.5. Instrukcja warunkowa – algorytmy rozgałęzione

Do tej pory mówiliśmy o algorytmach, w których sposób wykonania kolejnych instrukcji nie zależał od otrzymywanych wyników ani danych wejściowych. Poznasz teraz instrukcję, która pozwala uzależnić dalszą drogę postępowania w algorytmie od otrzymanego wyniku. Jest to instrukcja warunkowa, będąca „rozgałęźnikiem” w algorytmie.

## 1. Algorytmy

Sformułujmy problem w sposób następujący:



### Przykład

Daną wejściową niech będzie dowolna liczba rzeczywista. Na wyjściu chcemy otrzymać informację, czy liczba ta jest dodatnia czy nie.

Zapiszmy algorytm rozwiązuający to zadanie za pomocą listy kroków:

1. Wczytaj  $x$ .
2. Jeżeli  $x > 0$ , to wypisz: „ $x$  jest liczbą dodatnią” i zakończ.
3. Jeżeli  $x < 0$ , to wypisz: „ $x$  nie jest liczbą dodatnią” i zakończ.

Możemy ten zapis zredukować do dwóch kroków, przy czym obie listy kroków będą poprawne:

1. Wczytaj  $x$ .
2. Jeżeli  $x > 0$ , to wypisz: „ $x$  jest liczbą dodatnią”, w **przeciwnym wypadku** wypisz: „ $x$  nie jest liczbą dodatnią” i zakończ.

Wykonanie instrukcji jest zatem uzależnione od tego, czy zajdzie bany warunek (tu warunek  $x > 0$ ). Pamiętasz z matematyki, że zaprzeczeniem warunku  $x > 0$  jest warunek  $x < 0$ .

### Definicja

**Instrukcja warunkowa** to taka instrukcja, która w zależności od wartości warunku logicznego  $W$  umieszczonego w tej instrukcji umożliwia wykonywanie lub nie innych instrukcji - tak zwanych instrukcji wewnętrznych **A** i **B**.

Instrukcję tę będziemy zapisywać w algorytmach następująco:

**Jeśli** spełniony jest warunek  $W$ , **to** wykonaj instrukcję **A**; lub:  
**Jeśli** spełniony jest warunek  $W$ , **to** wykonaj instrukcję **A**,  
**w przeciwnym wypadku** wykonaj instrukcję **B**.

Zauważ, że pierwszy rodzaj instrukcji warunkowej jest realizacją drugiego zapisu w wypadku, gdy instrukcja **B** jest **instrukcją pustą**, czyli taką, która nie wykonuje żadnej konkretnej czynności.

Warunek w instrukcji warunkowej może być warunkiem prostym (jak w powyższym przykładzie) lub warunkiem złożonym, będącym koniunkcją lub alternatywą kilku innych warunków.

### Definicja

Algorytm, w którym występują instrukcje warunkowe, nazywa się | **algorytmem rozgałęzionym**.



### Przykład

Napiszmy schemat blokowy algorytmu podejmowania decyzji, czy po-brana liczba jest dodatnia.

Schemat blokowy tego algorytmu może wyglądać tak jak na rycinie 1.2:

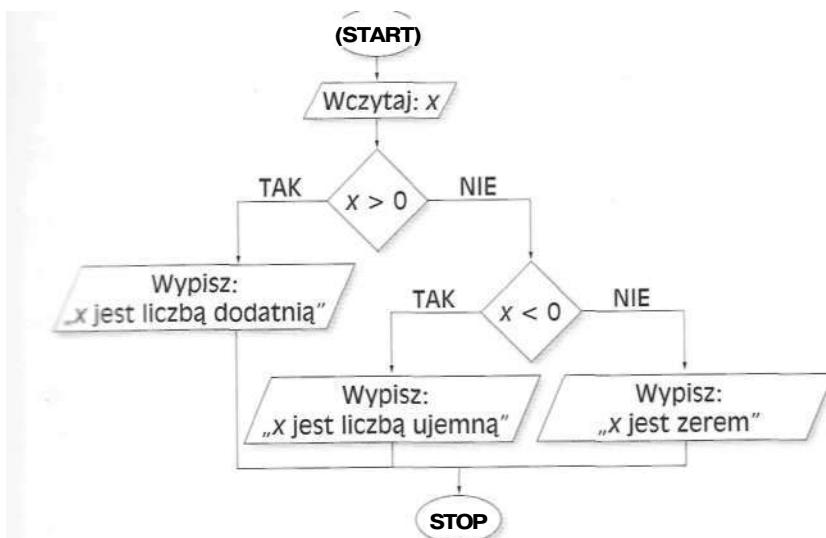


r.t. 1.2. Schemat blokowy algorytmu sprawdzającego, czy liczba jest dodatnia.

Zauważ, że w schemacie pojawi się blok decyzyjny. Przy realizacji algorytmu wybór ścieżki TAK lub NIE zależy od prawdziwości warunku unieszczonego w tym bloku.

Warunek musi być tak określony, aby ocena jego prawdziwości była jednoznaczna.

Modyfikacja algorytmu pozwoli nam otrzymać odpowiedź nie tylko na pytanie, czy liczba jest dodatnia czy nie, ale uzyskamy również информацию o wartości zerowej, jeśli taka została pobrana (ryc. 1.3):

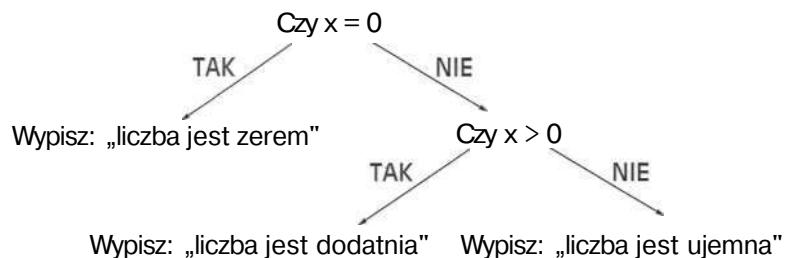


1.3. Schemat blokowy algorytmu badającego znak podanej liczby

## 1. Algorytmy

Mamy tu już do czynienia z pełną wersją algorytmu określającego znak pobranej liczby, ponieważ przewidziany został przypadek, że liczba **jest zerem**. Zwróć uwagę, że kolejność podejmowanych decyzji lub określenie badanych warunków można zmienić. Na przykład: **już** w pierwszym bloku pytamy o to, czy liczba jest zerem, i łączymy z blokiem wyjściowym wyprowadzającym napis: „liczba jest zerem” w wypadku prawdziwości warunku. W kolejnym bloku decyzyjnym pytamy, czy jest większa od zera - tu w zależności od odpowiedzi łączymy ją z blokami wyjścia odpowiednio wyprowadzającymi napisy: „liczba jest dodatnia”, „liczba jest ujemna”.

Spróbuj zapisać rozwiązanie tego zadania za pomocą listy kroków. Możesz wykorzystać do tego **drzewo postępowania** zamieszczone na rycinie 1.4:



Ryc. 1.4. Drzewo postępowania ilustrujące schemat czynności przy badaniu znaku liczby

### 1.6. Instrukcja iteracji

Spróbujmy teraz zapisać algorytm, którego zadaniem jest wypisanie wszystkich liczb dwucyfrowych. Lista kroków dla tak sformułowanego zadania wygląda następująco:

1. Wypisz 10.
2. Wypisz 11.
3. Wypisz 12.

90. Wypisz 99.

91. Zakończ.

Nie trzeba nikogo przekonywać, że taki zapis jest zbyt długi i w związku z tym mało użyteczny, zwłaszcza gdybyśmy na jego podstawie chcieli napisać program realizujący to zadanie (pomyśl, jak wyglądałby ten zapis, gdybyśmy chcieli wypisać na przykład wszystkie liczby siedmiocyfrowe). Korzystając z numeracji kroków algorytmu, można zapis zdecydowanie skrócić:

1. Zmiennej  $n$  przypisz wartość 10.
2. Wypisz  $n$ .

### 3. Zwiększ $n$ o 1.

- Jeśli  $n$  jest mniejsze niż 100, to wróć do kroku 2.

5 Zakończ.

Przedstawiona lista kroków rozwiązuje ten sam problem algorytmiczny, ale jest dużo krótsza od liniowego wypisania wszystkich elementów. Efekt ten osiągnęliśmy dzięki temu, że przy spełnieniu danego warunku (liczba mniejsza od pierwszej trzycyfrowej, czyli 100) powracamy do wykonania czynności opisanej w dwóch poprzednich krokach. Jeśli zaś warunek nie jest spełniony (czyli liczba jest już równa 100 lub większa), algorytm kończy swoje działanie.

Punkt czwarty jest punktem decyzyjnym: jeśli spełniony jest zadany warunek, to dana czynność zostaje wykonana, w przeciwnym wypadku algorytm wykonuje inną czynność (w naszym przykładzie jest to zakończenie algorytmu).

### Przykład

Pobierzmy dwie liczby i policzmy ich iloraz. Wynik zostanie wypisany na ekranie monitora.

**lit**

#### Specyfikacja problemu algorytmicznego

**Problem algorytmiczny:** Obliczanie ilorazu dwóch liczb rzeczywistych

**Dane wejściowe:**  $a \in R, b \in R$

**Dane wyjściowe:**  $\frac{a}{b} \in R$

Jest to przykład z pozoru łatwy, a schemat postępowania wydaje się bardzo prosty:

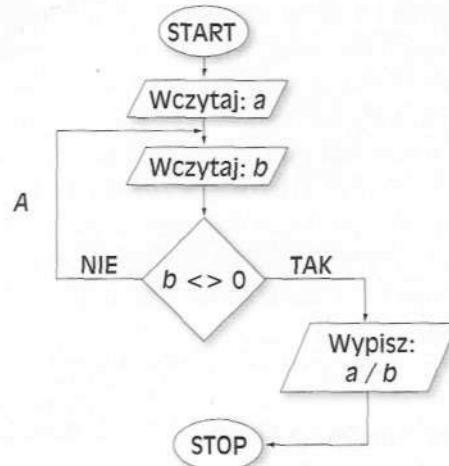
1. Wczytaj dzielną  $a$ .
- 1 Wczytaj dzielnik  $b$ .
3. Wpisz wartość ilorazu:  $a/b$ .
4. Zakończ.

Nie przewidzieliśmy jednak, że użytkownik jako wartość dzielnika może podać zero, a wówczas dzielenie staje się niewykonalne. Powinniśmy więc albo założyć w specyfikacji problemu, że zmienna  $b$  jest różna od zera, albo określić w algorytmie sposób dalszego postępowania w wypadku wprowadzenia zera dla zmiennej  $b$ . Powyższy zapis algorytmu nie jest więc zupełny.

Jakie czynności powinien zatem wykonać algorytm, jeśli podamy zerową wartość dzielnika? Jedną z najprostszych reakcji może być wyświetlenie komunikatu o niedozwolonym dzieleniu przez zero i zakończenie działania. My zaś chcemy, aby dzielenie zostało wykonane, a wynik był

## 1. Algorytmy

wyprowadzony na ekran. Algorytm powinien więc pobrać dzielnik jeszcze raz i powtarzać tę czynność tak długo, aż zostanie podana wartość różna od zera (ryc. 1.5).



Ryc. 1.5. Schemat blokowy algorytmu obliczającego iloraz dwóch liczb

W miejscu oznaczonym na schemacie blokowym literą - moglibyśmy umieścić blok wyjścia wyprowadzający napis: „Nieprawidłowa wartość dzielnika. Podaj wartość dzielnika jeszcze raz”. A zatem instrukcja pobierania wartości dzielnika (i warunkowo wyprowadzania napisu, o którym mowa powyżej) będą się powtarzać tyle razy, ile razy zostanie wprowadzona niepoprawna wartość.

Poprawny algorytm zapiszemy w postaci listy kroków:

1. Wczytaj dzielną a.
2. Wczytaj dzielnik b.
3. Jeśli  $b$  jest różne od zera, wypisz na ekranie monitora wartość ilorazu:  $a / b$  i zakończ.
4. Wróć do kroku 2.

Taką instrukcję złożoną z powtarzania danego ciągu operacji nazywamy iteracją, a jej dokładną definicję poznasz po analizie kolejnego przykładu.

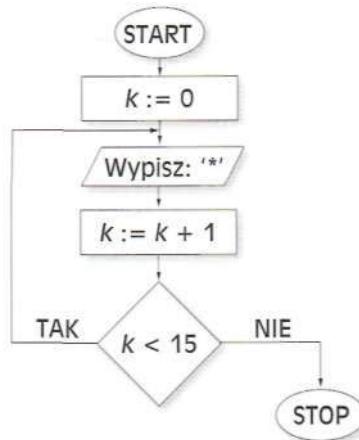


### Przykład

Skonstruujmy algorytm rysujący szlaczek składający się z piętnastu znaków gwiazdki.

Często liczbę powtórzeń znamy już wcześniej, czasem jeszcze przed rozpoczęciem wykonywania algorytmu. Przykładem jest właśnie rozpatrywany przez nas algorytm. W celu zliczania rysowanych gwiazdek wprowadzimy zmienną pomocniczą (tzw. licznikową)  $k$ .

Schemat blokowy algorytmu dla tego przykładu może wyglądać jak na rycinie 1.6:



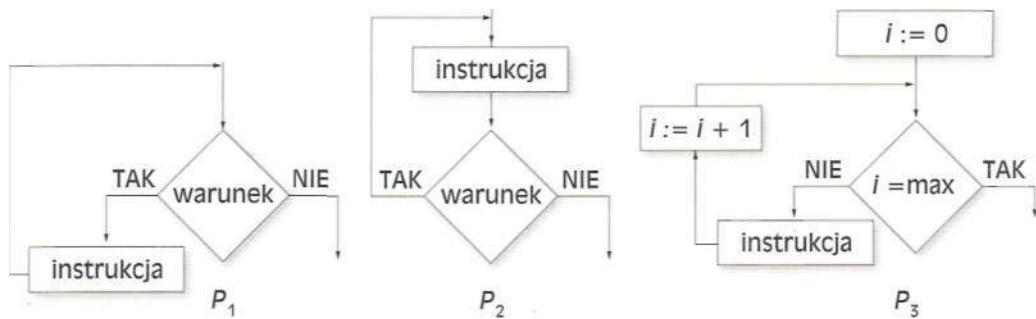
Ryc. 1.6. Schemat blokowy algorytmu z zastosowaniem iteracji

Po każdej wypisanej gwiazdce wartość licznika  $k$  jest zwiększana o jeden, czyli **inkrementowana**. Skoro gwiazdek ma być piętnaście, to sekwencja powtarzanych instrukcji wypisania gwiazdki i inkrementacji zmiennej liczącej nastąpi właściwie piętnaście razy.

### Definicja

**iteracją** nazywamy instrukcję powtarzania danego ciągu operacji. Liczba powtórzeń może być ustalona przed wykonaniem instrukcji lub może zależeć od spełnienia pewnego warunku, który jest sprawdzany w każdej iteracji. Iteracja inaczej zwana jest **pętlą**.

Na rycinie 1.7 przedstawiono graficzną prezentację podstawowych przypadków iteracji:



Ryc. 1.7. Trzy podstawowe przypadki iteracji stosowanych w algorytmach. P<sub>1</sub> - najpierw sprawdzany jest warunek, potem wykonywana instrukcja; P<sub>2</sub> - najpierw jest wykonywana instrukcja, a potem sprawdzany warunek; P<sub>3</sub> - instrukcja jest wykonywana max razy.

## 1. Algorytmy

Zapis iteracji  
w schematach  
blokowych

Pierwsze dwa schematy są zapisem pętli, w której liczba przebiegów jest zależna od spełnienia danego warunku. Zwróć uwagę na różnicę pomiędzy schematami  $P_1$  i  $P_2$ . W  $P_1$  najpierw sprawdzany jest warunek, a dopiero później wykonana zostaje instrukcja (instrukcja lub blok instrukcji), w  $P_2$  kolejność jest odwrotna. A zatem w pierwszym przypadku może się zdarzyć, że instrukcja nigdy nie zostanie wykonana, w drugim przypadku instrukcja zawsze zostanie wykonana przynajmniej raz. W pseudojęzyku iteracje te mają postać:

$P_1$ : **dopóki** spełniony jest warunek **wykonuj** instrukcję

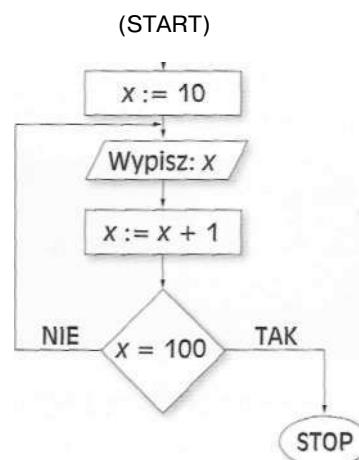
$P_2$ : **wykonuj** instrukcję **dopóki** spełniony jest warunek

W schemacie  $P_1$  liczba iteracji jest z góry ustalona i wynosi  $max$ .



### Przykład

Zapiszmy za pomocą schematu blokowego algorytm wypisujący wszystkie liczby dwucyfrowe.



Ryc. 1.8. Schemat blokowy algorytmu wypisującego wszystkie liczby dwucyfrowe z zastosowaniem pętli

Algorytm ten omówiliśmy już wcześniej i zapisaliśmy go w postaci listy kroków. Teraz przyjrzyj się jeszcze schematowi blokowemu tego algorytmu (ryc. 1.8). Zmienną pomocniczą jest  $x$  i przypisujemy jej wartość, która aktualnie ma zostać wypisana. W każdym przebiegu pętli liczba ta jest zwiększana o jeden (inkrementowana), a następnie sprawdzany jest warunek, czy osiągnęła już wartość 100. Jeśli nie, następuje jej wypisanie i kolejna inkrementacja, w przeciwnym wypadku kończymy działanie algorytmu. Jest to typowy przykład iteracji o z góry wiadomej liczbie powtórzeń. Przecież bez zbytniego zgłębiania tajników matematyki każdy wie, że liczb dwucyfrowych jest 90.

## 1.7. Poprawność algorytmu

Dokonamy teraz drobnej zmiany (a jak wielkiej, jeśli chodzi o sposób działania, to za chwilę się przekonasz) w zapisie algorytmu, którego założeniem jest wypisanie kolejnych liczb dwucyfrowych:

- 1.Zmiennej  $n$  przypisz wartość 10.
- 2.Wypisz  $n$ .
- 3.Zwiększ  $n$  o 1.
- 4.Jeśli  $n$  jest większe od 5, to wróć do kroku 2.
- 5.Zakończ.

Gdy zaczniemy wypisywać liczby zgodnie z działaniem tej wersji algorytmu, szybko dojdziemy do wniosku, że warunek określony w czwartym kroku jest zawsze spełniony, a zatem w nieskończoność wracalibyśmy do kroku drugiego. Do kroku piątego, w którym następuje zakończenie algorytmu, nie doszlibyśmy nigdy. Taki algorytm nie jest poprawny, gdyż nigdy nie kończy swojego działania.

Dokonajmy innej modyfikacji wyjściowego algorytmu:

- Zmiennej  $n$  przypisz wartość 10.  
 Wypisz  $n$ .  
 Zwiększ  $n$  o 2.  
 Jeśli  $n$  jest mniejsze od 100, to wróć do kroku 2.  
 Zakończ.

Zobaczmy, co zostanie wypisane zgodnie z działaniem tego algorytmu: 10, 12, 14, ..., 98. Czy o to chodziło w treści zadania? Nie, zatem ten algorytm również nie jest poprawny, ponieważ nie wykonuje określonego w specyfikacji zadania: nie wypisuje wszystkich liczb dwucyfrowych, tylko te, które są parzyste.

### Definicja

**Algorytm jest poprawny**, jeśli dla każdego poprawnego zestawu danych wejściowych prowadzi do poprawnych wyników, realizując zadany problem w skończonej liczbie kroków.

Przez poprawny zestaw danych wejściowych i poprawne wyniki rozumiemy wartości zgodne ze specyfikacją algorytmu. Natomiast o **częściowej poprawności algorytmu** mówimy wówczas, jeżeli osiągnie on koniec dla pewnych poprawnych danych wejściowych, a dane wyjściowe będą spełniać warunek końcowy (zauważ więc, że częściowa poprawność nie gwarantuje nam dojścia do punktu końcowego algorytmu).

Jak widać, obie wcześniejsze modyfikacje pierwotnego algorytmu spowodowały, że otrzymaliśmy algorytmy niepoprawne. Zauważ, że druga wersja byłaby poprawna przy inaczej sformułowanej treści zadania - podaj tę treść.

Częściowa  
poprawność  
algorytmu

## 1. Algorytmy

### 1.8. Złożoność obliczeniowa algorytmu

Złożoność obliczeniową algorytmu ukazują dwa różne rozwiązania tego samego problemu, przedstawione poniżej.



#### Przykład

Napiszmy algorytm, który pobiera na wejściu całkowitą dodatnią liczbę mniejszą od 100, a wyprowadza napis z informacją, czy jest to liczba parzysta czy nie.

#### Specyfikacja problemu algorytmicznego i opis użytych zmiennych

**Problem algorytmiczny:** Badanie parzystości liczby podanej na wejściu

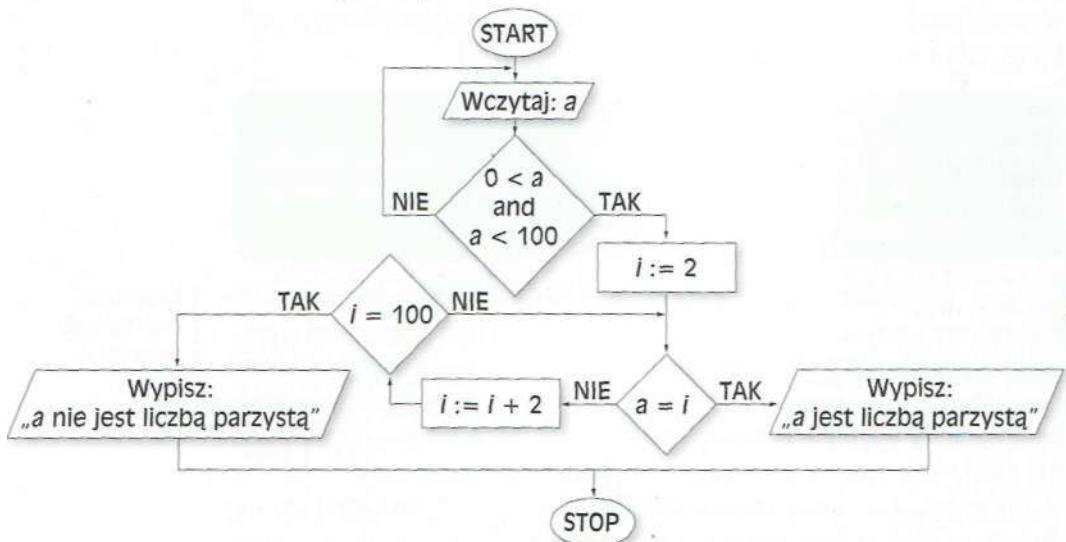
**Dane wejściowe:**  $a \in N, a < 100$  – badana liczba

**Dane wyjściowe:** Napis: „ $a$  jest liczbą parzystą” w wypadku, gdy podana liczba jest parzysta, lub „ $a$  nie jest liczbą parzystą” w wypadku, gdy podana liczba jest nieparzysta

**Zmienne pomocnicze:**  $i \in N$  – zmieniona licznikowa

Zauważ, że do specyfikacji problemu dodaliśmy opis użytych zmiennych pomocniczych, aby ułatwić ci analizę algorytmu. Dalej przedstawiemy dwa rozwiązania problemu (ryc. 1.9 i ryc. 1.10).

#### Rozwiązanie pierwsze

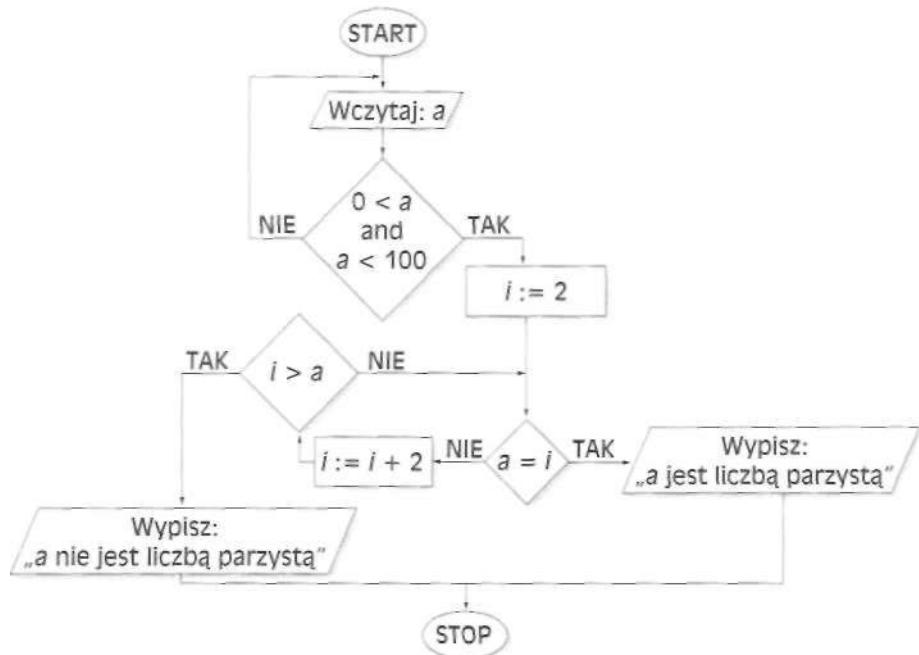


Ryc. 1.9. Przykład algorytmu realizującego badanie, czy liczba z zakresu 1–100 jest parzysta.

Pobieramy liczbę, a następnie sprawdzamy warunek złożony: czy jest to liczba dodatnia i równocześnie mniejsza od 100. Jeśli liczba nie spełnia tych warunków, to pobieramy ją znowu. Jeśli podana została właściwa liczba, czyli dodatnia i mniejsza od 100, porównujemy ją z każdą kolejną liczbą parzystą, począwszy od liczby 2. Jeśli wejściowa liczba byłaby nieparzysta, to w algorytmie wykonalibyśmy 49 porównań zmiennej pomocniczej  $i$  z liczbą 100.

Po przeanalizowaniu działania tego algorytmu musisz przyznać, że pomimo dużej liczby porównań rzeczywiście realizuje on w sposób poprawny postawione przed nim zadanie. Jest więc poprawny zgodnie z definicją poprawności algorytmu.

#### Rozwiążanie drugie



Ryc. 1.10. Przykład algorytmu realizującego badanie, czy liczba z zakresu 1-100 jest parzysta. Algorytm ten jest lepszy niż poprzedni.

Ten schemat blokowy jest niemal identyczny z poprzednim, lecz teraz porównujemy zmienną  $i$  nie z liczbą 100, ale z wartością liczby podanej na wejściu algorytmu. W wypadku podania liczby nieparzystej liczba porównań wynosić będzie  $1$  dla  $a$  większego od 1 i jedno porównanie

dla  $a$  równego 1. Skoro liczba  $a$  jest mniejsza niż 100, to zawsze liczba porównań będzie mniejsza niż 49.

## 1. Algorytmy

Porównując obydwa algorytmy, możemy stwierdzić:

- oba wykonują to samo zadanie,
- oba są poprawne (w rozumieniu definicji o poprawności algorytmu),
- w wypadku podania liczby parzystej w obu algorytmach wykonana zostanie taka sama liczba porównań,
- w wypadku podania liczby nieparzystej w drugim algorytmie wykonana zostanie mniejsza liczba porównań (im mniejsza będzie badana liczba, tym różnica będzie większa).

Dlatego w praktyce wykorzystalibyśmy drugie z proponowanych rozwiązań. Po to, aby ze zbioru różnych rozwiązań zadania algorytmicznego wybierać to, które wykonuje mniejszą liczbę operacji w celu uzyskania wyniku, wprowadzono pojęcie **złożoności obliczeniowej algorytmu**. Na złożoność obliczeniową składają się złożoność pamięciowa i czasowa.

**Złożoność pamięciowa** jest to cecha algorytmu, która wynika z liczby i rozmiaru danych wykorzystywanych w algorytmie. Złożoność ta wyznacza zależność rozmiaru pamięci potrzebnej do realizacji algorytmu od wielkości danych wejściowych (najczęściej chodzi o ilość danych wejściowych). Ten sam algorytm uruchomiony jako program na komputerach o różnych konfiguracjach może wymagać różnej wielkości pamięci (jest to związane z różnym sposobem reprezentacji danych, np. w pewnych konfiguracjach liczba całkowita zajmuje 2, a w innych 4 bajty). Złożoność pamięciowa jest najczęściej proporcjonalna do liczby zmiennych użytych w algorytmie.

**Złożoność czasowa** pozwala oszacować czas potrzebny do wykonania algorytmu. Złożoność tę wyznacza zależność czasu potrzebnego do wykonania algorytmu od rozmiaru danych wejściowych (najczęściej jest to ich ilość). Pojęcie to zostało wprowadzone, aby uniezależnić ocenę szybkości algorytmu od możliwości komputera, na jakim jest on realizowany. Za jednostkę czasu przy obliczaniu złożoności czasowej przyjmuję się wykonanie **operacji dominującej**, to jest operacji charakterystycznej dla danego algorytmu (najczęściej wykonywanej).

Operacjami dominującymi mogą być między innymi:

- dodawanie i mnożenie w wypadku algorytmów numerycznych,
- porównanie i przestawienie elementów w algorytmach porządkujących ciąg liczb.

Jak wyznaczamy złożoność czasową algorytmu? Jeśli na przykład, aby uzyskać wynik, będziemy musieli wykonać dla  $n$  danych wejściowych  $2n$  dodawań i  $4/7$  mnożeń, co w sumie da nam  $6n$  operacji dominujących, to zauważ, że liczbę operacji możemy potraktować jako funkcję zmiennej  $n$  - czyli rozmiaru danych wejściowych. W tym wypadku mielibyśmy  $f(n) = 6n$ , funkcja byłaby więc funkcją liniową. Gdyby funkcja liczby wykonanych operacji dominujących miała postać  $f(n) = 9n + 12$ , to wciąż byłaby funkcją liniową. O takich algorytmach powiemy, że mają **liniową**

Złożoność pamięciowa  
i czasowa algorytmu

Klasy złożoności  
obliczeniowej  
algorytmów - notacja  
Omkron

**złożoność obliczeniową**, ponieważ czas wykonania algorytmu zależy w sposób liniowy od rozmiaru danych wejściowych  $n$ . Tak naprawdę nie interesuje nas dokładna zależność czasu wykonania algorytmu od rozmiaru danych (gdyż na tę dokładną zależność ma duży wpływ sposób ułożenia danych wejściowych, np. ich kolejność), w praktyce ważny jest tylko typ funkcji tej zależności, na przykład dla funkcji  $f(n) = 3n^2 + 5$  mówimy o **kwadratowej złożoności obliczeniowej**.

Dla zagadnienia złożoności obliczeniowej kluczowe jest wyznaczenie **klasy funkcji** zależności czasu obliczeń od rozmiaru danych. **Klasa funkcji** wyznacza ograniczenie funkcji od góry i w przypadku algorytmów jest używana do szacowania czasu działania algorytmu dla pesymistycznego zestawu danych (wymagającego największej liczby operacji). Złożoność obliczeniową wyznaczoną dla pesymistycznego zestawu danych nazywamy **pesymistyczną złożonością obliczeniową**. Ze względu na analizę algorytmów, ta właśnie złożoność jest najbardziej interesująca. Do opisu klasy funkcji służy notacja  $\Theta$  (zwana „O duże”).

Pesymistyczna  
złożoność  
obliczeniowa  
algorytmu

Mówimy, że funkcja  $f(n)$  jest co najwyżej klasą funkcji  $g$ , co zapisujemy:

$$f(n) = \Theta(g(n)),$$

jeżeli istnieje taka stała rzeczywista  $c$  i liczba naturalna  $n_0$ , że dla każdego  $n$  większego lub równego  $n_0$  funkcja  $f(n)$  jest mniejsza lub równa iloczynowi  $c \cdot g(n)$ . O liniowej złożoności obliczeniowej powiemy zatem, że jest klasa  $O(n)$ .

W praktyce, dzięki założeniu o dążeniu do nieskończoności rozmiaru danych, możemy dokonać **znaczących uproszczeń** przy wyznaczaniu klas złożoności algorytmu. Polegają one na:

- pominięciu wszystkich składników funkcji oprócz tego, który ma największy wpływ (a więc najczęściej składnik z  $n$  o najwyższej potędze);
- pominięciu wszelkich stałych współczynników.

Przykładowo, gdy liczba operacji dominujących wynosi  $5n^2 + 6n$ , to mówimy, że algorytm ten ma złożoność rzędu  $\Theta(n^2)$  w pesymistycznym przypadku, ponieważ zgodnie z opisany powyżej uproszczeniami pod uwagę bierzemy tylko najwyższą potegę zmiennej  $n$ , pomijając współczynnik. Ogólnie o algorytmach, których złożoność obliczeniowa wynosi odpowiednio  $\Theta(n^k)$ , gdzie  $k$  jest liczbą naturalną większą od 1, powiemy, że mają **wielomianową złożoność obliczeniową**.

Dla ułatwienia ci w przyszłości określania klas złożoności algorytmu przedstawimy teraz kilka najczęściej spotykanych klas według notacji O duże, wraz z podaniem cech charakterystycznych tych algorytmów. Zestawienie uporządkowaliśmy według malejącej szybkości algorytmu.

Typowe  
złożoności  
algorytmów

#### Najczęściej spotykane złożoności algorytmów

1. **Złożoność stała  $O(1)$**  - algorytm wykonuje stałą liczbę operacji dominujących bez względu na rozmiar danych wejściowych.

- 2. Złożoność logarytmiczna  $\Theta(\log n)$**  - taką złożoność mają na przykład algorytmy, w których problem postawiony dla danych rozmiaru  $n$  da się sprowadzić w pesymistycznym przypadku do problemu z rozmiarem danych o połowę mniejszym. (Uwaga: wielkość podstawy logarytmu najczęściej przyjmuje się tutaj jako 2, choć tak naprawdę w przypadku asymptotycznym - czyli gdy rozmiar danych  $n$  dąży do nieskończoności - wielkość podstawy nie ma znaczenia).
- Złożoność liniowa  $\Theta(n)$**  - taką złożoność mają na przykład algorytmy, które dla każdej danej wykonują w pesymistycznym przypadku stałą liczbę operacji podstawowych. Dwukrotny wzrost ilości danych powoduje dwukrotny wzrost liczby wykonywanych operacji.
- Złożoność liniowo-logarytmiczna  $\Theta(n \log n)$**  - taką złożoność mają na przykład algorytmy, w których problem postawiony dla danych rozmiaru  $n$  da się sprowadzić w liniowej liczbie operacji do rozwiązania dwóch problemów o rozmiarach  $n/2$ .
- 5. Złożoność kwadratowa  $\Theta(n^2)$**  - taką złożoność mają na przykład algorytmy, w których dla każdej pary elementów danych wykonywana jest stała liczba operacji podstawowych (zwykle algorytmy z podwójnymi pętlami).
- 6. Złożoność wykładnicza  $\Theta(2^n)$ ,  $\Theta(n!)$**  - są to algorytmy bardzo wolne, których realizacja w pesymistycznym przypadku jest niewykonalna nawet dla niewielkich rozmiarów danych.

#### Czy wiesz, że...

Wykonanie algorytmu o złożoności  $2^n$  dla rozmiaru danych 100 zajęłoby bardzo szybkiemu komputerowi czas tysiąckrotnie dłuższy niż szacowany wiek Wszechświata!

Działania mające na celu opracowanie algorytmu najlepiej rozwijającego dany problem nazywamy optymalizacją algorytmu. Na etapie optymalizacji bierze się pod uwagę zarówno złożoność czasową, jak i pamięciową.

#### 1.9. Realizacja algorytmów w arkuszu kalkulacyjnym

Arkusze kalkulacyjne wykorzystuje się najczęściej do złożonych zestawień finansowych lub statystycznych, ale można również stworzyć funkcjonalne arkusze pozwalające na sprawne obliczanie zadań z matematyki, fizyki czy chemii. Do zadanego problemu układamy wówczas algorytm, w którym odpowiednio ujmujemy możliwości arkusza kalkulacyjnego.



### Przykład

Opracujmy arkusz kalkulacyjny obliczający pole trójkąta na podstawie długości trzech odcinków. W wypadku, gdy z podanych odcinków nie da się utworzyć trójkąta, powinien zostać wyświetlony odpowiedni komunikat.

Do rozwiązania problemu przydatny będzie tak zwany wzór Herona (Heron z Aleksandrii - grecki matematyk, fizyk, mechanik, wynalazca i konstruktor, żyjący w latach ok. 10-ok. 70 r. n.e.) na pole  $S$  trójkąta o bokach  $a, b, c$ :

$$\sqrt{p(p-a)(p-b)(p-c)}$$

$$\text{gdzie } p = \frac{a+b+c}{2}$$

Wzór Herona

Zanim skorzystamy z tego wzoru, musimy sprawdzić, czy z podanych odcinków można utworzyć trójkąt. W przeciwnym razie pod pierwiastkiem mogłaby się znaleźć liczba ujemna (pamiętasz zapewne, że suma długości dowolnych dwóch boków w trójkącie jest większa od długości trzeciego boku).

Napiszmy najpierw algorytm.

### Specyfikacja problemu i opis użytych zmiennych

**Dane wejściowe:**  $a, b, c$  - długości boków będące liczbami nieujemnymi

**Dane wyjściowe:**  $S$  - pole trójkąta o bokach  $a, b, c$ ; jest to liczba większa od zera lub napis „Z podanych odcinków nie można zbudować trójkąta”

**Zmienne pomocnicze:**  $p$  - dodatnia liczba rzeczywista

Lista kroków algorytmu dla tego problemu wygląda następująco:

1. Wczytaj  $a, b, c$ .
2. Jeżeli  $a + b > c \wedge a + c > b$  oraz  $c + b > a$ , przejdź do kroku 3, w przeciwnym razie wyświetl napis: „Z podanych odcinków nie można zbudować trójkąta” i zakończ.
3. Przypisz zmiennej  $p$  wartość wyrażenia  $p = \frac{a+b+c}{2}$ .
4. Przypisz zmiennej  $S$  wartość wyrażenia  $S = \sqrt{p(p-a)(p-b)(p-c)}$ .
5. Wypisz wartość  $S$  i zakończ.

## 1. Algorytmy

Rycina 1.11 przedstawia ekran arkusza kalkulacyjnego realizującego to zadanie.

The screenshot shows a Microsoft Excel spreadsheet titled "roz\_1". The formula bar displays:  $=JEŻELI(ORAZ(D6+F6>H6,F6+H6>D6,D6+H6>F6);(D6+F6+H6)/2,0)$ . The spreadsheet contains the following text and data:

	A	B	C	D	E	F	G	H	I
1									
2	Arkusz służy do obliczania pola trójkąta na podstawie długości jego boków.								
3									
4	W niebieskie pola wpisz odpowiednio długości odcinków, z których chcesz zbudować trójkąt.								
5									
6	a=		4	b=		4	c=		5
7									
8									
9									7,806
10									
11									6,5
12									
13									
14									
15									
16									
17									

Ryc. 1.11. Przykład realizacji prostego algorytmu z warunkami w arkuszu kalkulacyjnym

Zwróć uwagę na komórkę B11 -jest tam napisana formuła warunkowa, która oblicza wartość zmiennej pomocniczej  $p$ , jeśli z odcinków da się stworzyć trójkąt. Jeśli trójkąta nie można zbudować, w komórce B11 wstawiana jest wartość 0. Wartość obliczonego pola wyświetlna jest w komórce B9. Formuła wpisana w tę komórkę ma postać:

$=JEŻELI(B11=0;"Nie da się z tych odcinków zbudować trójkąta! Podaj inne wartości.>";PIERWIASTEK(B11*(B11-D6)*(B11-F6)*(B11-H6)))$

Obliczenia iteracyjne \\ w arkuszu kalkulacyjnym

Zajmijmy się teraz realizacją pętli, czyli obliczeń iteracyjnych, w arkuszu kalkulacyjnym. Obliczenia takie można zrealizować w arkuszu MS Excel za pomocą kopiowania formuł w tabeli. Prześledźmy przykład obliczający wartość lokaty bankowej o początkowej wartości  $W$ , złożonej na  $n$  miesięcy. Oprocentowanie w skali miesiąca wynosi  $p$ , a kapitalizacja, to jest dopisanie odsetek do lokaty, następuje po każdym miesiącu. Wartość lokaty w każdym kolejnym miesiącu jest powiększana o odsetki naliczane na koniec miesiąca według wzoru:

$$W = W_{\text{aktualna}} + (W_{\text{aktualna}} \cdot p)$$

gdzie:  $W_{\text{aktualna}}$  - wartość lokaty po  $n$  miesiącach

$p$  - oprocentowanie w skali miesiąca

Rycina 1.12 przedstawia ekran arkusza kalkulacyjnego realizującego to zadanie.

### 1.9. Realizacja algorytmów w arkuszu kalkulacyjnym

Microsoft Excel - roz_1							
<input type="checkbox"/> Plik <input type="checkbox"/> Edycja <input type="checkbox"/> Widok <input type="checkbox"/> Wstaw <input type="checkbox"/> Format <input type="checkbox"/> Narzędzia <input type="checkbox"/> Dane <input type="checkbox"/> Okno <input type="checkbox"/> Pomoc							
<input type="button"/> <span style="margin-left: 10px;">Σ</span> <input type="button"/> <span style="margin-left: 10px;">Ariał CE</span> <span style="margin-left: 10px;">10</span>							
D16	$=JEŻELI(C16<=C$9,D15*C$8+D15,0)$						
A	B	C	D	E	F	G	H
1							
2	Arkusz służy do obliczania wartości lokaty bankowej, zawartej na określony termin.						
3	Lokata kapitalizuje się co miesiąc.						
4	W pola zaznaczone na niebiesko wpisz odpowiednio:						
5	wysokość wkładu początkowego, oprocentowanie w skali miesiąca, termin lokaty w miesiącach:						
6							
7	wkład	1 000,00 zł					
8	oprocentowanie	0,50%					
9	termin	90					
10							
11							
12	Na koniec okresu oszczędzania otrzymasz: <b>1 566,55 zł</b>						
13							
14							
15							
16							
17							
18							
19							

Ryc. 1.12. Arkusz kalkulacyjny obliczający wartość lokaty po określonym terminie. Wszystkie wartości (czyli wkład, oprocentowanie, termin) użytkownik wpisuje w odpowiednie pola.

Obliczenia pomocnicze umieszczone są w ukrytej tabelce w kolumnach C i D. Pierwszą kolumnę tej tabelki stanowią liczby reprezentujące kolejne miesiące trwania lokaty, a w drugiej kolumnie są obliczane wartości lokaty po każdym miesiącu oszczędzania według formuły wyświetlanej u góry arkusza. Formuła ta jest kopiuowana w kolejnych wierszach tabelki. Warunek zastosowany w tej formule powoduje, że w miesiącach przekraczających czas trwania lokaty wpisywana jest wartość 0. Dzięki temu do sprawnego wyświetlania wartości lokaty na koniec okresu oszczędzania możemy wykorzystać funkcję MAX. Funkcja ta z drugiej kolumny tabeli pomocniczej wyznaczy wartość lokaty na koniec okresu oszczędzania (ponieważ będzie to największa liczba w tej kolumnie). W tym celu w komórce F12 jest wpisana formuła: = MAX (D14:D1314). Wynika z niej, że arkusz został przystosowany do obliczeń lokat trwających nie dłużej niż 1300 miesięcy.

### Pytania kontrolne

3

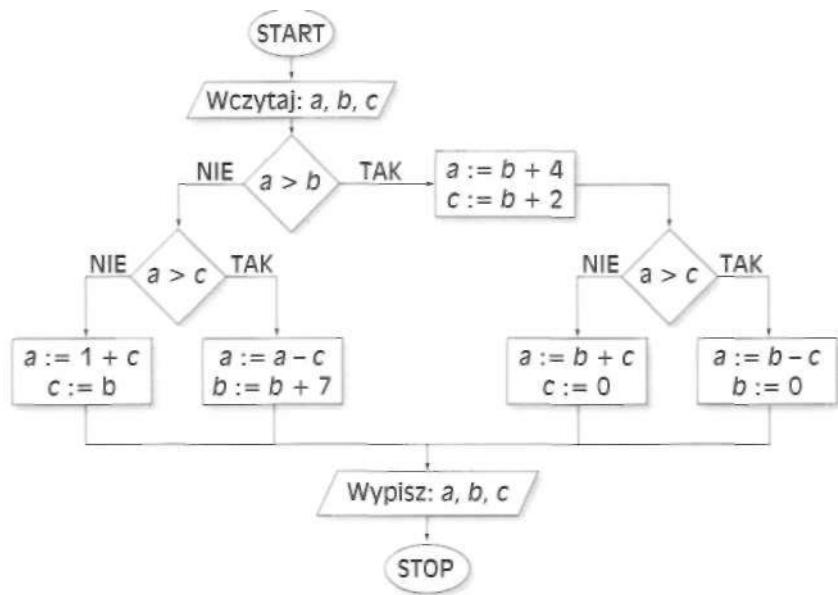
X

- Podaj definicję algorytmu. Czy przepis na pieczenie ciasta jest algorymem? Odpowiedź uzasadnij.
- Jaki algorytm jest poprawny?
- Co to znaczy, że algorytm powinien być uniwersalny?

4. Omów krótko poznane sposoby zapisu algorytmów.
5. Co to jest specyfikacja algorytmu i czemu służy?
6. Co to jest instrukcja wyboru? Podaj parę przykładów jej zastosowania.
7. Co to jest iteracja? Podaj przykład algorytmu, w którym liczba iteracji jest z góry ustalona, i przykład algorytmu, w którym liczba iteracji zależy od spełnienia warunku.  
Od czego zależy pamięciowa złożoność obliczeniowa?
9. Jakie znasz klasy złożoności obliczeniowej? Krótko je scharakteryzuj.
10. Jak można zrealizować obliczenia iteracyjne w arkuszu kalkulacyjnym?

### Ćwiczenia

1. Narysuj schemat blokowy algorytmu obliczającego moduł (wartość bezwzględną) dowolnej liczby rzeczywistej.
2. Podaj specyfikację zadania algorytmicznego obliczającego rozwiązanie równania pierwszego stopnia z jedną niewiadomą i narysuj odpowiedni schemat blokowy.
3. Zapisz specyfikację i narysuj schemat blokowy algorytmu pobierającego kolejne liczby naturalne jednocyfrowe dotąd, aż ich suma przekroczy 30. Na wyjściu algorytmu chcemy otrzymać informację, ile liczb zostało podanych oraz ile wyniosła ich suma.
4. Dany jest schemat blokowy algorytmu. Podaj, jakie wartości otrzymamy na wyjściu, jeśli wprowadzimy odpowiednio:  $a = 4$ ,  $b = 2$ ,  $c = 0$ .

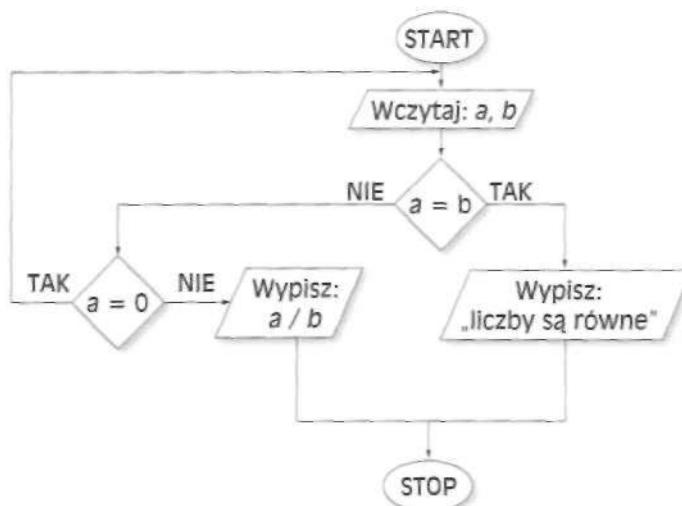


5. Narysuj schemat blokowy algorytmu, który zlicza sumę wszystkich liczb całkowitych z przedziału  $[a, b]$ , gdzie  $a, b$  są liczbami całkowitymi podanymi na wejściu. Uwaga: musisz przewidzieć przypadek, gdy druga z wprowadzonych liczb będzie nie większa od pierwszej wprowadzonej liczby.
6. Narysuj schemat blokowy algorytmu, którego wynikiem działania jest wartość najmniejszej z trzech różnych liczb podanych na wejściu.
7. Narysuj schemat blokowy algorytmu, którego zadanie polega na wypisaniu tabliczki mnożenia dla czynników należących do przedziału  $[1, 10]$ .
8. Podaj specyfikację i narysuj schemat blokowy algorytmu obliczającego pole powierzchni i obwód trójkąta prostokątnego. Długości boków przy kącie prostym są podawane na wejściu algorytmu.
9. Poniżej podane są specyfikacje i schematy blokowe algorytmów. Sprawdź ich poprawność i wskaż ewentualne błędy.

a)

**Specyfikacja problemu algorytmicznego****Problem algorytmiczny:**

Podanie ilorazu dwóch liczb różnych od siebie lub komunikatu o równości tych liczb

**Dane wejściowe:** $a, b \in R$ **Dane wyjściowe:** $\frac{a}{b}$ , jeśli  $a \neq b$ , lub napis „liczby są równe”, jeśli  $a = b$ 

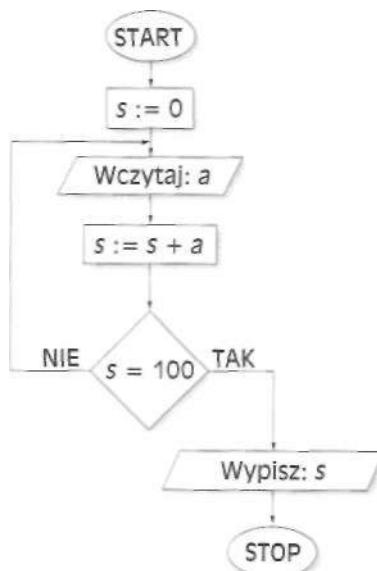
b)

**Specyfikacja problemu algorytmicznego**

**Problem algorytmiczny:** Wypisanie liczby 100, gdy taką wartość osiągnie suma pobieranych liczb

**Dane wejściowe:** Liczby całkowite dodatnie

**Dane wyjściowe:** Liczba 100



10. Zaprojektuj arkusz kalkulacyjny, który oblicza wartość wyrażenia  $a^b$ , gdzie  $a$  jest podstawą potęgi oraz liczbą rzeczywistą,  $b$  zaś to wykładnik potęgi będący liczbą całkowitą dodatnią. Wykorzystaj fakt, że na przykład  $3^4 = 3 \cdot 3 \cdot 3 \cdot 3$ .

## 2. Wstęp do programowania w języku C++

W tym rozdziale prezentujemy podstawy programowania w języku C++. Nauczysz się zapisywać w nowoczesnym języku programowania instrukcje: warunkową i pętli. Poznasz również metody konstrukcji algorytmów w języku C++.

### 2.1. Reprezentacja danych w komputerze - podstawowe operacje w systemie binarnym

Wszystkie dane w komputerze (w tym również programy) są reprezentowane przez liczby zapisane w systemie binarnym (dwójkowym), czyli systemie liczbowym o podstawie dwa. Możliwe cyfry tego systemu to 0 i 1. Na co dzień posługujemy się zwykle systemem dziesiętnym, a więc systemem liczbowym, w którym jest dziesięć cyfr: od 0 do 9. Liczbę w systemie dziesiętnym można łatwo zamienić na postać binarną (ograniczamy się do liczb całkowitych dodatnich). Metoda zamiany polega na kolejnym dzieleniu liczby dziesiętnej przez 2 i zapisywaniu reszt z dzielenia. Dzielenie kończymy, gdy wynik dzielenia jest równy 0.

Zacznijmy od przykładu - zamieńmy liczbę 53 na system dwójkowy:

```
53 : 2 = 26, reszta: 1  
26 : 2 = 13, reszta: 0  
13 : 2 = 6, reszta: 1  
6 : 2 = 3, reszta: 0  
3 : 2 = 1, reszta: 1  
1 : 2 = 0, reszta: 1
```

Zapis liczby  
w systemie binarnym

Wynik odczytujemy od dołu: 110101. Umownie zapisujemy, że  $53_{(10)} = 110101_{(2)}$ . Każdą pozycję w liczbie binarnej nazywamy bitem. Liczba 53 została zapisana na 6 bitach (ma 6 cyfr).

#### Definicja

**Bit** to najmniejsza jednostka informacji potrzebna do zakodowania jednego z dwóch równie prawdopodobnych stanów układu. Bit może przyjąć jedną z dwóch wartości, które zwykle określa się jako 0 i 1, choć można przyjąć dowolną inną parę wartości, na przykład: prawda i falsz lub -1 i 1.

System binarny jest **systemem pozycyjnym** (podobnie jak system dziesiętny), co oznacza, że wartość cyfry zależy od jej pozycji w liczbie. W systemie binarnym pierwszą cyfrę z lewej nazywamy **bitem najbardziej znaczącym**, natomiast pierwszą cyfrę z prawej - **bitem najmniej znaczącym**.

## 2. Wstęp do programowania w języku C++

Aby obliczyć wartość liczby binarnej w systemie dziesiętnym, należy cyfry tej liczby przemnożyć przez odpowiednie potęgi liczby 2, a następnie wszystko zsumować:

$$110101_{(2)} = 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 53_{(10)}$$

Bit jest bardzo małą jednostką informacji, dlatego w praktyce posługujemy się najczęściej bajtem (ang. *byte*). Bajt (B) składa się z ośmiu bitów. Ponieważ każdy z ośmiu bitów może przyjąć wartość 1 lub 0, zatem na jednym bajcie można zapisać  $2^8$  różnych wartości.

### Klasyfikacja jednostek informacji

Kolejne jednostki informacji są wielokrotnościami bajta:

kilobajt (kB);  $1 \text{ kB} = 2^{10} \text{ B} = 1024 \text{ B}$

megabajt (MB);  $1 \text{ MB} = 2^{20} \text{ B} = 1024 \text{ kB} = 1048576 \text{ B}$

gigabajt (GB);  $1 \text{ GB} = 2^{30} \text{ B} = 1024 \text{ MB} = 1073741824 \text{ B}$

terabajt (TB);  $1 \text{ TB} = 2^{40} \text{ B} = 1024 \text{ GB} = 1099511627776 \text{ B}$

Na liczbach zapisanych w systemie binarnym możemy wykonywać działania arytmetyczne podobnie jak na liczbach zapisanych w systemie dziesiętnym.

#### Dodawanie binarne

Zacznijmy od dodawania. Aby dodać dwie liczby binarne, wystarczy wiedzieć, że:

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$0 + 0 = 0$$

$$1 + 1 = 10$$

Liczby binarne dodajemy do siebie podobnie jak liczby dziesiętne. Sumowane liczby zapisujemy jedna pod drugą tak, aby w kolejnych kolumnach znalazły się cyfry umieszczone na tych samych pozycjach w liczbie. Wynik zapisujemy pod kreską. Przeanalizujmy przykład:

$$\begin{array}{r} 10010 \\ + \underline{1011} \\ \hline \end{array}$$

$$11101$$

Najpierw dodajemy liczby znajdujące się w pierwszej kolumnie od końca:  $0 + 1 = 1$ . W drugiej kolumnie od końca dodajemy  $1 + 1 = 10_{(2)}$ , wpisujemy więc 0, a 1 zostaje przeniesione do następnej kolumny. W trzeciej kolumnie dodajemy  $0 + 0$  i otrzymujemy 0, ale dodajemy jeszcze 1 z przeniesienia, więc jako wynik wpisujemy 1. Wyniki dodawania w kolejnych kolumnach to odpowiednio 1 i 1.

$$10010_{(2)} + 1011_{(2)} = 11101_{(2)}, \text{czyli w dziesiętnym kodzie } 18 + 11 = 29.$$

#### Odejmowalni binarne

Tablica odejmowania binarnego wygląda następująco:

$$1 - 1 = 0$$

$$0 - 0 = 0$$

$$1 - 0 = 1$$

$$10 - 1 = 1$$

Ograniczamy się do przykładu, gdy odjemna jest większa od odjemnika:

$$\begin{array}{r} 10101 \\ - \underline{01111} \\ 00110 \end{array}$$

W pierwszej kolumnie od prawej strony odejmujemy 1 od 1 i otrzymujemy 0. W kolejnej kolumnie od 0 nie da się odjąć 1, dokonujemy więc przeniesienia z kolumny trzeciej od końca. Wzięte stamtąd 1 jest w obecnej kolumnie liczbą 10<sub>(2)</sub>, zatem w wyniku odejmowania otrzymujemy 1. W kolumnie trzeciej od końca, w której teraz mamy 0 (1 przenieśliśmy wcześniej), chcemy odjąć 1 od 0 i w tym celu musimy dokonać kolejnego przeniesienia. W kolumnie sąsiedniej nie ma 1, wobec tego przenosimy 1 z najbardziej znaczącego bitu odjemnej. Przenosimy 1 do kolumny sąsiedniej i otrzymujemy w niej liczbę 10<sub>(2)</sub>, po czym przenosimy z niej 1 do następnej kolumny (po przeniesieniu w pozycji zostanie 1). W kolumnie trzeciej od końca przeniesione 1 staje się liczbą 10<sub>(2)</sub>. Dalej jest już prosto: odejmujemy 1 od 10<sub>(2)</sub>, i otrzymujemy 1. Następnie w kolumnie czwartej od końca odejmujemy 1 od 1, co daje 0. Jako bit najbardziej znaczący wpisujemy 0, bo tyle wynosi 0-0. Oto wynik w postaci binarnej i odpowiadający mu wynik dziesiętny:

$$10101_{(2)} - 1111_{(2)} = 110_{(2)} \text{ czyli w dziesiętnym kodzie } 21 - 15 = 6.$$

### Mnożenie binarne

Do mnożenia binarnego dwóch liczb korzystamy z faktu, że:

$$0 \times 1 = 0$$

$$1 \times 0 = 0$$

$$0 \times 0 = 0$$

$$1 \times 1 = 1$$

Kolejne wyniki mnożeń podkreślamy i dodajemy do siebie odpowiednio kolumny (zgodnie z zasadą opisaną już wcześniej):

$$\begin{array}{r} 1110 \\ \times \quad 101 \\ \hline 1110 \\ 0000 \\ 1110 \\ \hline 10001 \quad 10 \end{array} \qquad \begin{array}{r} 1111 \\ \times \quad 111 \\ \hline 1111 \\ 1111 \\ \hline 1101001 \end{array}$$

W drugim z przykładów mnożenia tak dobraliśmy obie liczby, aby można było przeanalizować sposób przenoszenia potęg liczby 2 do kolejnych kolumn na etapie dodawania poszczególnych wyników mnożeń. Zauważ, że wykonywanie działań na liczbach binarnych jest analogiczne do działań na liczbach dziesiętnych. Różnica polega na przenoszeniu do kolejnych kolumn wielokrotności liczby 2, a nie wielokrotności liczby 10.

## 2.2. Proces wytwarzania programu

Zanim przystąpisz do pisania programów, zapoznaj się z miejscem, jakie zajmuje algorytm w języku programowania w całym procesie tworzenia programu komputerowego.

#### **Etapy wytwarzania programu:**

1. Definicja problemu (specyfikacja).
  2. Analiza wymagań i znalezienie odpowiedniej metody rozwiązania (opracowanie algorytmu).
  3. Implementacja programu (zapis algorytmu w języku programowania).
  4. Uruchomienie i testowanie programu.
  5. Konserwacja programu.

Etapy 1 i 2 zostały już omówione w poprzednim rozdziale. Techniczna strona realizacji etapu 3 będzie treścią tego rozdziału. Etapu 4 nie trzeba tłumaczyć - program najlepiej sprawdzić w działaniu na zestawach typowych i nietypowych danych. **Konserwacja programu** określamy naprawianie błędów ujawniających się w czasie praktycznego użytkowania i dalszy rozwój związany z sytuagami nieuwzględnionymi w początkowych planach.

Programowanie pierwszych komputerów polegało na żmudnym przedstawianiu przełączników oraz wpinaniu i wypinaniu odpowiednich wtyczek. Uruchomienie zatem innego programu wymagało w praktyce zmiany budowy komputera. Dane były zaś, podobnie jak dziś, przechowywane w postaci binarnej w pamięci komputera. Dopiero w 1949 roku powstał pierwszy komputer, w którym wykorzystano genialny pomysł Johna von Neumanna, polegającym na przechowywaniu programów w tej samej postaci co dane. Architektura oparta na tym pomyśle jest stosowana w niemal wszystkich obecnie produkowanych komputerach.

Jak wiesz, najważniejszym elementem komputera jest procesor. Może on wykonywać polecenia wyrażone wyłącznie w tak zwanym kodzie maszynowym.

Kod maszynowy to jedyny język zrozumiały dla procesora. Polecenia i ich atrybuty są w tym kodzie reprezentowane poprzez sekwencje liczbowe. Każdy typ procesora ma swój własny kod maszynowy wraz z listą instrukcji, które rozumie, zatem programy napisane na jeden typ procesora nie mogą być w łatwy sposób przeniesione na inny.

Kod maszynowy jest dla człowieka nieprzejrzysty, a programowanie w nim żmudne i narażone na pomyłki. Program zapisany w kodzie maszynowym może wyglądać następująco:

Aby ułatwić programowanie komputerów, wymyślono inne języki programowania. Dzielą się one na tak zwane generacje:

I generacja: kod maszynowy;

II generacja: język symboliczny, niskiego poziomu (asembler);

III generacja: język wysokiego poziomu (np.: ALGOL, BASIC, Pascal, C, FORTRAN, C++, Java);

IV generacja: narzędzia, które umożliwiają budowę prostych aplikacji przez zestawienie „prefabrykowanych” modułów, tak zwane generatory aplikacji (np.: SQL, Delphi);

V generacja: języki sztucznej inteligencji, języki systemów ekspertowych, najbardziej zbliżone do języka naturalnego (np. PROLOG).

Im wyższa generacja, tym język programowania jest bardziej podobny do języka naturalnego. Dzięki temu programista może się bardziej skupić na rozwiązyaniu problemu niż na sposobie jego zapisu.

Lista instrukcji asemblera jest identyczna z listą instrukcji procesora w kodzie maszynowym. Instrukcje w asemblerze mają jednak postać tak zwanych **mnemoników** (krótkich wyrazów zrozumiałych dla człowieka), a nie sekwencji liczbowych jak w kodzie maszynowym i są elementarne - zatem pisanie w tym języku jest czasochłonne, ponieważ zaprogramowanie nawet nieskomplikowanej czynności wymaga użycia wielu instrukcji. Program napisany w asemblerze musi być przed uruchomieniem przetłumaczony na kod maszynowy. Operacja taka nazywa się **asemblacja**, a odwrotna - **deasemblacja**. Ponieważ tłumaczenie odbywa się według zasady: jedna instrukcja asemblera - jedna instrukcja kodu maszynowego, asemblacją jest szybka i prosta. Języka tego używa się w sytuacjach, gdy bardzo ważna jest szybkość programu.

Języki III i wyższych generacji nazywamy językami wysokiego poziomu. W językach tych mamy do dyspozycji instrukcje, które odpowiadają od jednej do kilku tysięcy instrukcji procesora.

Wszystkie przedstawione w podręczniku algorytmy implementujemy za pomocą języka C++. Jest to nowoczesny, bardzo popularny język dający wiele możliwości. Pozwala na pisanie programów operujących na instrukcjach dość niskiego poziomu (co daje programistie większą kontrolę nad sposobem wykonania programu), ale udostępnia też szereg mechanizmów pozwalających na programowanie obiektowe, w którym definiuje się własne typy danych (o takim programowaniu dowiecie się z lektury ostatniego rozdziału tego podręcznika).

Program komputerowy zapisany w języku programowania wysokiego poziomu, jakim jest C++, to ciąg instrukcji zapisanych jedna po drugiej w pliku tekstowym, zwanym dalej **kodem źródłowym**. W każdym języku pliki takie mają swoje charakterystyczne rozszerzenie, na przykład w języku C++ jest to rozszerzenie .cpp, a w Pascalu - pas. Aby program za-

Kompilacja  
i linkowanie  
programu

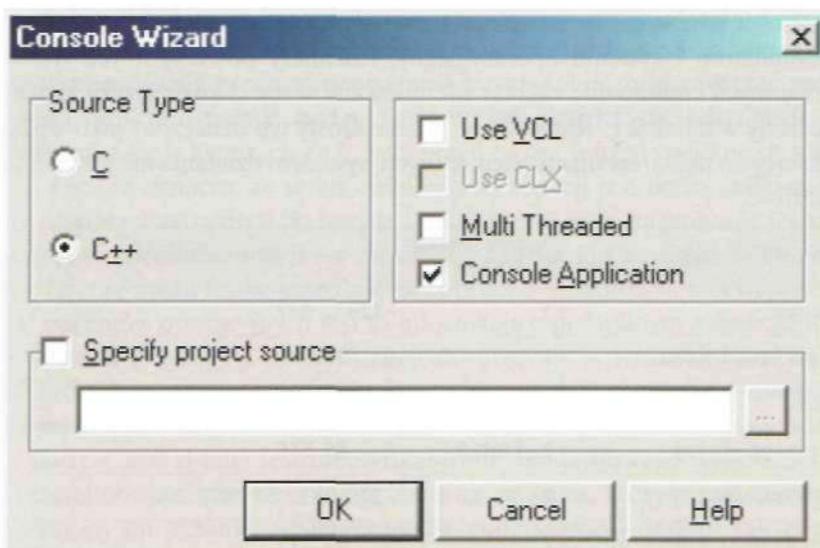
## 2. Wstęp do programowania w języku C++

pisany w takiej postaci był zrozumiały dla procesora, musi zostać **skompilowany**, czyli przetłumaczony na kod maszynowy, w którym instrukcje wyrażone są przez ciąg liczb w systemie dwójkowym. Takim tłumaczeniem zajmuje się specjalny program zwany **kompilatorem**. Po skompilowaniu programu powinna nastąpić konsolidacja, czyli połączenie programu z bibliotekami, popularnie zwane **linkowaniem** od nazwy **linker** – programu, który to wykonuje. **Bibliotekami** są gotowe zestawy programów, podprogramów i procedur, które programista często wykorzystuje, pisząc własny program. Przydatnym narzędziem stosowanym podczas sprawdzania programów jest **debuger** (w języku polskim nazywany „odpluskwiaczem”), który umożliwia między innymi śledzenie wykonywania programu krokowo, linijka po linijce – czyli instrukcja po instrukcji. Na rynku jest dostępnych wiele różnych kompilatorów. Do własnych środków możesz używać dowolnego kompilatora C++. Programy omówione w podręczniku oraz dołączone na płycie CD testowaliśmy za pomocą darmowych (do użytku niekomercyjnego) kompilatorów: Borland Builder Personal 6.0 oraz Dev-C++, które również zamieszczone na płycie CD. Obydwa programy są rozbudowanymi środowiskami programistycznymi i oprócz komplikacji oferują również wygodne edytory, które w czasie pisania programu zaznaczają różnymi kolorami i krojami czcionki zmienne, słowa kluczowe, funkcje, co w znacznym stopniu ułatwia pracę programisty.

### Czy wiesz, że...

W literaturze anglojęzycznej błąd w programie komputerowym nosi nazwę **bug**, czyli owad, robak (na język polski wyraz ten bywa często tłumaczony jako „pluskwa”). Termin ten upowszechnił się w słownictwie komputerowym w 1945 roku, kiedy to w czasie poszukiwania przyczyny błędного działania komputera MARK II okazało się, że wszystkiemu winna jest éma, która spowodowała spięcie w przekaźnikach elektromagnetycznych.

Przed rozpoczęciem pisania i uruchamiania przykładowych programów zawartych w tym podręczniku przekażemy ci kilka praktycznych rad (ograniczamy się do omówienia postępowania w środowiskach programistycznych dołączonych do podręcznika). Wszystkie programy zamieszczone w tym podręczniku powinieneś pisać i uruchamiać w trybie konsolowym, to znaczy w środowisku tekstowym, a nie graficznym. W programie Borland Builder Personal, aby uruchomić tryb konsolowy, należy zamknąć środowisko graficzne, które zwykle otwiera się samoczynnie po uruchomieniu Budlera. Wykonuj to za pomocą polecenia Close All z menu File. Następnie otwórz nowy projekt polecienniem **File->New->Other->Console Wizard**. W okienku, które się pojawi, wybierz opcje jak na rycinie 2.1.



Ryc. 2.1. Okno kreatora nowego projektu

Teraz już możesz wpisać swój program w arkuszu, który się pojawi po naciśnięciu przycisku OK.

Nie zapominaj o zapisywaniu swoich projektów. Ponieważ w procesie komplikacji jest tworzonych kilka plików, warto zapisywać każdy program w osobnym folderze - pozwala to na uniknięcie bałaganu. Skompilowanie i uruchomienie napisanego przez ciebie programu nastąpi po naciśnięciu przycisku Run z paska narzędzi lub po naciśnięciu klawisza F9.

W przypadku kompilatora Dev-C++ zaczynasz pisać swój kod programu w okienku, które pojawi się po wybraniu polecenia Plik->Nowy->->Plik źródłowy. Kompilujesz i uruchamiasz swój program klawiszem F9.

### 2.3. Podstawowe typy danych

W poprzednim rozdziale omówiliśmy tworzenie algorytmów rozwiązyjących różne problemy. Zwykle algorytmy te wymagały zastosowania zmiennych. Podobnie jak w wielu innych językach programowania również w C++ każda zmienna ma swój typ, który musi być określony przez programistę przed pierwszym użyciem zmiennej. W C++ występuje pięć podstawowych typów zmiennych. Są to: znak, liczba całkowita, liczba zmiennopozycyjna, liczba zmiennopozycyjna podwójnej precyzji i wartość logiczna. Wymienione typy oznacza się w C++ odpowiednio: `char`, `int` (skrót od `integer`), `float`, `double`, `bool` (skrót od `boolean`). Inne typy danych utworzone są na podstawie tych pięciu typów. Typy danych różnią się pomiędzy sobą rozmiarem zajmowanej pamięci i zakresem

Podstawowe typy  
zmiennych C++

wartości (wielkości te zmieniają się w zależności od użytego procesora, kompilatora i systemu operacyjnego). Rozmiary poszczególnych typów zmiennych i minimalne zakresy zdefiniowane w standardzie języka zamieściliśmy w tabeli 2.1. Jest w C++ jeszcze szósty typ oznaczony jako void służący do deklarowania funkcji, których wynikiem działania nie jest żadna wartość.

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

**Tab. 2.1.** Typy zmiennych zdefiniowanych w standardzie C++. Znakim gwiazdki oznaczono przykładowe zakresy; ponieważ nie są one ścisłe i określone w standardzie języka, dlatego w twojej konfiguracji sprzętowo-programowej mogą być inne. Zapis 3,4E38 znaczy tyle samo co  $3,4 \cdot 10^{38}$ .

## 2.4. Budowa programu

Poznawanie podstaw języka, w którym będziemy implementować wszystkie algorytmy, zacznijmy od napisania krótkiego programu:

```
#include <iostream> // (1)
#include <cstdio> // (2)
using namespace std; // (3)

int main() // (4)
{
    cout << "To jest moj pierwszy program"; // (6)
    getchar(); // (7)
    return 0; // (8)
}
```

Jednym zadaniem tego programu jest wyświetlenie na standardowym wyjściu, jakim jest ekran monitora, napisu: To jest moj pieirwszy program. Każdy program napisany w języku C++ musi zawierać specjalną funkcję o nazwie main. Treść funkcji (każdej, nie tylko main), umieszcza się w klamrach {i}. Jeśli przed nazwą funkcji znajduje się słowo i n t , to oznacza, że wynik działania tej funkcji jest liczbą całkowitą. W nawiasach okrągłych po nazwie funkcji jest miejsce na wpisanie jej argumentów. W linii ósmej jest umieszczona instrukcja r e t u r n 0, która powoduje, że nasza funkcja main kończy pracę z wartością 0. Dla większości systemów operacyjnych jest to informacja o pomyślnym zakończeniu programu (więcej na temat funkcji dowiesz się w rozdziale trzecim). W szóstej linii naszego programu na ekran monitora zostaje wyprowadzony napis. Zastosowany operator « nazywamy operatorem wyjścia, a c o u t (czyt. si-aut) jest identyfikatorem standardowego wyjścia, najczęściej powiązanym z ekranem. Zauważ, że napis, który ma zostać w y - wyświetlony na ekranie, umieszczamy w cudzysłowie - jest to tak zwana stała tekstowa, z języka angielskiego zwana string.

Całość szóstej linii możemy nazwać strumieniem danych skierowanym na ekran.

Definicja operacji wyjścia i wejścia znajduje się w bibliotece o nazwie `iostream`. Aby korzystać z biblioteki, musimy do kodu programu dodać linię `#include <iostream>`. Linie pierwsza i druga są tak zwanymi dyrektywami preprocessora.

". Biblioteka  
; <iostream>

### Definicja

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

Dyrektwa preprocessora `#include` zapoznaje kompilator z nagłówkiem odpowiedniej biblioteki (aby kompilator mógł sprawdzić składnię programu) i informuje preprocesor, że program używa procedur z podanej biblioteki.

W siódmej linii kodu programu użyliśmy funkcji `getchar()`. Jest to Użycie funkcji `getchar()` funkcja, która wypisuje na ekran znak wpisany na klawiaturze aż do wcisnięcia klawisza Enter. Funkcja ta przydaje się, kiedy nasz program uruchamiamy w środowisku Windows, ponieważ pozwala na zatrzymanie okienka konsoli. Program nasz, podobnie jak wszystkie inne zamieszczone w tym podręczniku, pisany jest w trybie konsolowym (tekstowym). Oznacza to, że w środowisku Windows po uruchomieniu programu zostanie otwarte okienko konsoli, w którym pojawią się wyniki działania programu. Okienko to znika natychmiast po zakończeniu programu. Gdybyśmy nie zastosowali funkcji `getchar()`, nie zdążylibyśmy obejrzeć efektu działania naszego programu. Funkcja ta definiowana jest w bibliotece `stdio`, dlatego w drugiej linii zapisana została odpowiednia dyrektywa. W linii trzeciej

## 2. Wstęp do programowania w języku C++

programu jest zawarta informacja dla kompilatora, że w programie będzie wykorzystana przestrzeń o nazwie std. Użycie tej przestrzeni ułatwia dostęp do całej biblioteki standardu C++, która jest w niej zadeklarowana.

Dokonajmy teraz modyfikacji w zapisie kodu naszego pierwszego programu.

Program działałby tak samo, gdyby jego kod wyglądał następująco:

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

int main()
{
    cout << "To jest ";
    cout << "moj pierwszy ";
    cout << "program";
    getchar();
    return 0;
}
```

---

lub:

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

int main()
{
    cout << "To jest " << "moj pierwszy " << "program";
    getchar();
    return 0;
}
```

---

Za każdym razem zobaczymy na ekranie napis:

To jest moj pierwszy program

Przechodzenie do nowej linii

Fakt, że kolejne części zdania, które ma być wyświetlane na ekranie, umieszczamy w następnych liniach, nie ma żadnego wpływu na sposób wyświetlania. Przejście do nowej linii w kodzie programu nie powoduje przejścia do nowej linii w tekście wyświetlonym. Jeśli jednak chcemy wyświetlić ten tekst w trzech różnych liniach, musimy zaznaczyć to w pisanym kodzie za pomocą **znaku nowej linii**. Można to zrobić na dwa sposoby:

- za pomocą tak zwanego manipulatora endl, który wyprowadza znak nowej linii;
- przez wpisanie znaku nowej linii \n.

### Definicja

**Manipulatorami** nazywamy specjalne funkcje, które włączone do wyrażeń związanych z operacjami wejścia/wyjścia zmieniają sposób formatowania tekstu.

Dodajmy więc znaki nowej linii do kodu:

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

int main()
{
    cout << "To jest\n";
    cout << "moj pierwszy" << endl;
    cout << "program";
    getchar();
    return 0;
}
```

Efektem zastosowania znaków nowej linii będzie napis na ekranie:

```
To jest
moj pierwszy
program
```

Aby twój program był czytelny zarówno dla ciebie, jak i pozostałych osób, zapoznamy cię ze sposobem umieszczania w kodzie źródłowym **komentarzy**, czyli napisów, które kompilator ignoruje. Komentarze są umieszczane w celu wyjaśniania poszczególnych fragmentów kodu, na przykład: do czego służą definiowane zmienne, jakie zadanie mają wywoływanie funkcje itp.

W C++ są dwa rodzaje komentarzy. Pierwszy to komentarze, które mogą obejmować wiele linii. Jeśli chcesz objąć komentarzem kilka linijek kodu, wówczas tekst komentarza rozpoczynasz znakiem /\* i kończysz znakiem \*/.

Umieszczenie  
komentarzy w kodzie  
programu

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

/* To jest komentarz. Mogę tu na przykład napisać, co program wykonuje,
   kto jest autorem programu i kiedy został napisany */

int main()
{
    cout << "witaj w programie";
    getchar();
    return 0;
}
```

Drugi rodzaj komentarza rozpoczyna się znakiem // i obejmuje tekst do końca linii:

```
#include <iostream>          // dołączam biblioteki
#include <cstdio>           // używana jest przestrzeń nazw std
using namespace std;        // bezparametrowa funkcja główna programu

int main()                  // klamra otwierająca główną funkcję
{                          // wprowadzenie napisu na ekran
    cout << "witaj w programie" // oczekuje na znak 'entera' z klawiatury
    getchar();                // przekazuje do systemu wartość 0
    return 0;                 // klamra zamkająca główną funkcję
}
```

## 2. Wstęp do programowania w języku C++

Możesz oczywiście stosować w kodzie źródłowym obydwa rodzaje komentarzy. Nie wolno jednak zagnieździć komentarzy objętych znakami /\* i \*/, czyli umieszczać komentarzy tego rodzaju wewnątrz bloku będącego już komentarzem. Umiejętnie stosowanie komentarzy zwiększa czytelność kodu i znacznie ułatwia jego późniejszą analizę.

Spójrz jeszcze na program, w którym stosujemy zarówno znaki przejścia do nowej linii, jak i komentarze:

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

int main()
{
    cout << "To jest" << endl;           // tu pojawił się znak nowej linii (1)
    cout << "moj pierwszy\n    program" ;   // i tu też jest znak nowej linii (2)
    getchar();
    return 0;
}
```

---

Efektem działania programu będzie napis na ekranie monitora:

```
To jest
moj pierwszy
program
```

W linii oznaczonej (1) został wyprowadzony na ekran monitora znak nowej linii, a więc dalsza część tekstu znajduje się już w linii następnej. Linia druga realizuje przejście do nowej linii bezpośrednio po słowie pierwszy. Kolejna linia zaczyna się od czterech znaków spacji, ponieważ tyle ich jest przed słowem program.

Przeanalizujmy drugi, równie prosty przykład:

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

int main()
{
    cout << "Ile masz lat? ";
    int lata;           // deklaracja zmiennej (1)
    cin >> lata;        // pobranie z klawiatury (2)
                        // wartości dla zmiennej lata
    cout << "Juz wiem: masz " << lata << " lat";
    cin.ignore();       // nie czekamy na naciśnięcie 'enter'
    getchar();
    return 0;
}
```

Ten program wykazuje już pewną interakcję z użytkownikiem: nie tylko wyprowadza na standardowe wyjście jakieś informacje, ale również

pobiera informacje od użytkownika, korzystając ze standardowego wejścia, jakim jest klawiatura.

W linii oznaczonej w komentarzu numerem (1) pojawiła się deklaracja zmiennej. Zmienna ma nazwę `1 a t a` i będzie przechowywała wartości typu całkowitego. Równocześnie został dla niej zarezerwowany obszar pamięci odpowiadający wielkości typu `int`.

Sposób deklarowania  
zmiennych

### Definicja

**Deklaracja zmiennej** to nadanie zmiennej nazwy i określenie typu wartości, jaka będzie w niej przechowywana.

Z deklaracją zmiennej wiąże się rezerwacja w pamięci miejsca, w którym będzie przechowywana wartość związana z zadeklarowaną zmienną.

Jeśli chcesz się dowiedzieć, ile bajtów zajmują poszczególne typy zmiennych w twoim komputerze, możesz się posłużyć następującą instrukcją:

```
cout << sizeof(int);
```

W nawiasie możesz wpisać nazwę innego interesującego cię typu.

Nazwa zmiennej nie może być słowem kluczowym języka, czyli takim, które w języku programowania ma już swoje przypisane znaczenie. Nazwa zmiennej musi spełniać jeszcze kilka warunków: pierwszy jej znak powinien być literą lub znakiem podkreślenia, pozostałe zaś znaki muszą być literami, cyframi lub znakami podkreślenia. Inne od wymienionych znaków nie mogą być użyte w nazwie zmiennej. Wielkość liter użytych w nazwie zmiennej ma w C++ znaczenie, tak więc zmienna `licznik` i `LiczniK` są zupełnie innymi zmiennymi.

Budowa poprawnej  
nazwy zmiennej

Błędnymi nazwami zmiennych są na przykład:

`12malp` - cyfra na początku

`int` - słowo kluczowe

`Ala!` - wykrzyknik w nazwie

`liczba kroków` - spacja w nazwie

Poprawnymi nazwami zmiennych są na przykład:

`_7krasnali`

`Int`

`aLa`

`liczba_krokow`

`Archiwum_nr_125`

Możemy zadać sobie pytanie, gdzie należy umieszczać deklarację zmiennej. Najważniejszą zasadą jest zadeklarowanie zmiennej przed jej pierwszym użyciem. W naszym programie zadeklarowaliśmy zmienną bezpośrednio przed linią, w której jej użyliśmy - pozwala na to język C++ (w większości innych języków zmienne deklaruje się tylko na początku funkcji lub programu). Mogliśmy zrobić to wcześniej, bezpośred-

## 2. Wstęp do programowania w języku C++

nio po klamrze otwierającej treść funkcji main (albo przed słowem main), ale nie mogliśmy zadeklarować zmiennej po odwołaniu się do niej.

### Zapamiętaj

Każda nazwa zmiennej użyta w języku C++ musi zostać zadeklarowana przed jej pierwszym użyciem.

v \_\_\_\_\_ )

Przypisywanie : W linii z komentarzem opatrzonym numerem (2) pojawiła się nowa wartości zmiennym : instrukcja: `cin >> lata`. Operator `>>` nazywamy **operatorem wejścia**, a `cin` (czyt. si-in) jest **identyfikatorem standardowego wejścia**, najczęściej powiązanym z klawiaturą. Całą drugą linię możemy nazwać声明iem danych płynącym ze standardowego wejścia do zmiennej `lata`. Od tego momentu zmienna ma wartość, jaką wprowadziliśmy z klawiatury, aż do czasu, gdy wartość tę zmienimy przez przypisanie jej innej wartości (np. `lata = 13`). Przypisanie wartości zmiennej może być wykonane równocześnie z jej deklaracją, mówimy wówczas, że zadeklarowaną zmienną **inicjujemy** początkową wartością. Deklaracja zmiennej wraz z nadaniem jej jednocześnie wartości bywa nazywana **definicja**.

Oto przykładowa deklaracja zmiennych wraz z ich inicjacją:

```
int i= 7;           // deklaracja zmiennej całkowitej,  
                   // jej inicjacja wartością 7  
float w=8.4;       // deklaracja zmiennej rzeczywistej,  
                   // inicjacja wartością 8.4  
char znak= 'p';    // deklaracja zmiennej znakowej  
                   // i inicjacja wartością 'p'
```

Zmienne te służą odpowiednio do przechowywania liczb całkowitych, liczb rzeczywistych i znaków.

Powróćmy do poprzedniego przykładu programu, do linii:

```
cout << "Juz wiem: masz " << lata << " lat"; // informacja dla  
                                         // użytkownika
```

Zauważ, że zmienna `lata` nie jest ujęta w cudzysłów. Gdyby tak było, to zostałby wyświetlony ciąg znaków nim ujęty, czyli:

Juz wiem: masz lata lat

Znaki sterujące : Pamiętaj, że tekst, który umieszczasz w cudzysłowie, zostaje w tej samej postaci wyświetlony na ekranie oprócz kilku znaków sterujących wypisywaniem tekstu, rozpoczynających się ukośnikiem \, które modyfikują sposób wyświetlania tekstu. Jeden ze znaków sterujących wyświetlaniem tekstu już poznaleś, jest nim znak przejścia do nowej linii: \n. Inne spośród najczęściej używanych znaków to:  
\t - tabulator poziomy,  
\r - powrót na początek wiersza (po tym znaku tekst wypisywany jest od początku bieżącego wiersza, nawet jeśli w tym wierszu był już jakiś tekst),  
\b - cofnięcie o jeden znak (cofa kursor o jeden znak w wierszu).

Jeśli chcesz na ekran monitora wyprowadzić wartość zmiennej, podajesz jej nazwę nieujętą w cudzysłów.

Instrukcja `cin.ignore()` ignoruje znak ostatnio wprowadzony na standartowe wejście. Chodzi w tym wypadku o zignorowanie znaku końca wiersza (`\n`), którego użyliśmy przy wpisywaniu wieku. Znak ten oczywiście nie został wpisany do zmiennej `lata` i ciągle tkwi w buforze klawiatury. Jeśli nie pominiemy go instrukcją `cin.ignore()`, zostanie wychwycony przez funkcję `getchar()`, przez co program nie będzie czekał na naciśnięcie **Enter** na zakończenie.

Jeśli w programie korzystamy ze strumienia wejściowego, to instrukcji `cin.ignore()`; zawsze będziemy używać przed wywołaniem funkcji `getchar()`.

Przy pisaniu programu najczęściej popełnianymi błędami są błędy syntaktyczne. Błąd syntaktyczny (ang. *syntax error*) polega na naruszeniu reguł opisujących składnię języka. Zrobienie błędu syntaktycznego spowoduje, że kompilator nie wygeneruje programu wykonywalnego, czyli pliku z rozszerzeniem .exe. Kompilacja zostanie przerwana przy wykryciu pierwszego błędu, po czym nastąpi wyprowadzenie komunikatu wyjaśniającego rodzaj błędu. Zazwyczaj jest to:

- pominięcie średnika,
- użycie niezadeklarowanej zmiennej,
- pominięcie klamry kończącej program,
- zniekształcenie słowa kluczowego lub zmiennej,
- pominięcie słowa kluczowego w instrukcji.

W programie występują też inne rodzaje błędów. Są one o wiele trudniejsze do wykrycia, gdyż kompilator ich nie sygnalizuje. Bardzo częstym błędem popełnianym przez początkujących adeptów C++ jest niezamierzone użycie operatora przypisania w wyrażeniu logicznym (znak = zamiast ==).

Do wyszukiwania błędów w programie bardzo przydatny jest debugger. Pozwala on między innymi na uruchomienie programu w trybie krokowym. W trybie tym możesz w swoim programie prześledzić, jak zmieniają się wartości interesujących cię zmiennych w czasie wykonywania kolejnych linii kodu. Aby jakąś zmienną dołączyć do zmiennych obserwowanych w trybie krokowym w środowisku kompilatora C++ Builder, należy wykonać następujące czynności:

- w kodzie programu ustawić kurSOR przed nazwą zmiennej albo po niej i kliknąć prawym przyciskiem myszy;
- z menu kontekstowego, które się rozwinię, należy wybrać zakładkę **Debug->Add Watch at Cursor**, klikając na niej lewym przyciskiem myszy (nazwa wybranej zmiennej powinna pojawić się w okienku pod kodem twojego programu w zakładce **Watches**). W podobny sposób możesz dołączyć inne zmienne - ograniczaj się do tych najważniejszych, podejrzanych o to, że są przyczyną błędnego działania programu.

## 2. Wstęp do programowania w języku C ++

Gdy już wskazałeś zmienne, których wartości chcesz śledzić, uruchom wykonywanie programu w trybie krokowym - wystarczy z menu Run wybrać funkcję Step Over lub po prostu nacisnąć klawisz skrótu F8. Każda linia programu będzie teraz interpretowana i wykonywana, a w celu wykonania następnej linii należy ponownie nacisnąć klawisz F8. Wartości obserwowanych zmiennych są aktualizowane po każdym wykonaniu kolejnej linii. Narzędzia debugera są szczególnie przydatne w skomplikowanych programach zawierających pętle (o pętlach dowiesz się w dalszej części tego rozdziału).

### 2.5. Instrukcje sterujące

Instrukcje sterujące kierują przebiegiem wykonywania programu. Pozwalają one na zmianę kolejności wykonywania innych instrukcji zależnie od otrzymywanych wyników. Bez instrukcji sterujących moglibyśmy pisać tylko programy liniowe, czyli realizujące algorytmy liniowe. Dzięki instrukcjom sterującym możemy pisać programy realizujące algorytmy rozgałęzione i iteracyjne.

#### 2.5.1. Instrukcja warunkowa if... else

Same operacje wejścia i wyjścia to jeszcze zbyt mało, aby napisać program wykonujący złożone zadanie. Zapoznajmy się z zapisem w języku C ++ instrukcji, które poznaliśmy przy omawianiu algorytmów. Zanim przedstawimy formalną składnię instrukcji, omówimy przykładowe programy i ich poszczególne elementy.

Zacznijmy od instrukcji warunkowej:

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

int main()
{
    int liczba;                                // deklaracja zmiennej
    cout << "Podaj liczbę ";                   // przypisanie wartości z klawiatury
    cin >> liczba;                            // instrukcja warunkowa
    if (liczba>0)
        cout << "Liczba jest dodatnia";
    else
        cout << "Liczba nie jest dodatnia";
    cin.ignore();
    getchar();
    return 0;
}
```

Program ten sprawdza, czy liczba jest dodatnia (algorytm tego problemu zamieściliśmy w poprzednim rozdziale). Znajduje się w nim instruk-

cia warunkowa, której zapis umiesz zrealizować zarówno na schemacie blokowym, jak i w pseudojęzyku. W języku C++ instrukcja warunkowa ma dwie postaci:

**Składnia instrukcji warunkowej if ... else**

```
if (wyrażenie) instrukcja1;
lub
if (wyrażenie) instrukcja1;
else instrukcja2;
```

Wyrażenie może przyjąć jedną z dwóch wartości logicznych (popularnie – boolowskich): albo jest prawdziwe (wartość `true`), albo jest fałszywe (wartość `false`). W C++ przyjmuje się, że 0 odpowiada fałszowi, a każda inna liczba odpowiada wartości logicznej prawdy. Wyrażenie w instrukcji może być wyrażeniem prostym lub złożonym, będącym koniunkcją (operator `&&`) bądź alternatywą (operator `||`) kilku warunków. W miejscu wyrażenia może też znajdować się zmienna. Oto kilka przykładów (wraz z objaśnieniami) wyrażeń mających wartość `true`:

- (`a==3`); jeśli wartość zmiennej `a` wynosi 3,
- (`c+12*d<g`); jeśli wartość wyrażenia `c+12*d` jest mniejsza od zmiennej `g`,
- (`a+1<7&&i!=4`); jeśli suma `a+1` jest mniejsza od 7 i jednocześnie zmiana `i` jest różna od 4,
- (`z`); jeśli zmiana `z` ma wartość różną od 0.

W pierwszej postaci instrukcji warunkowej: „`if (wyrażenie) instrukcja1;`” instrukcja1 wykonana się tylko w wypadku, gdy wyrażenie jest prawdziwe. Kolejne instrukcje programu będą się wykonywały niezależnie od wyniku wyrażenia.

W drugiej postaci instrukcji warunkowej: „`if (wyrażenie) instrukcja1; else instrukcja2;`” wykonana się tylko jedna z tych dwóch instrukcji. Jeśli wyrażenie będzie prawdziwe, to wykonana zostanie instrukcja1, a nie instrukcja2, w przeciwnym wypadku wykonana zostanie instrukcja2, a nie instrukcja1. Kolejne instrukcje programu wykonają się niezależnie od wartości wyrażenia.

Instrukcja może być pojedyncza lub złożona z kilku instrukcji.

#### Zapamiętaj

Skończoną liczbę instrukcji pojedynczych ujętych w klamry { i } nazywamy instrukcją złożoną lub inaczej blokiem instrukcji.

---

\* Zobaczmy, jak bardzo różnić się będą efekty działania dwóch programów, których kody różnią się tylko jedną parą nawiasów klamrowych.

## 2. Wstęp do programowania w języku C++

Jako pierwszy rozważmy program zawierający nawiasy klamrowe:

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

int main()
{
    int dlugosc;
    cout << "Podaj liczbę całkowitą "; // (1)
    cin >> dlugosc; // (2)
    if (dlugosc > 0) // (3)
    {
        cout << endl; // (4)
        cout << "Podajesz dodatnią liczbę" << endl; // (5)
        cout << "Dobrze, bo to ma być długość odcinka" << endl; // (6)
        cout << "Narysuje więc odcinek o długości " << dlugosc; // (7)
        cout << endl; // (8)
    }
    cout << "\n\nDziękuje, koncze działanie..."; // (9)
    cin.ignore();
    getchar();
    return 0;
}
```

Program prosi użytkownika o podanie liczby całkowitej. Jeśli podana liczba jest dodatnia, to wykonają się wszystkie instrukcje bloku otwartego w linii oznaczonej (4), a zamkniętego w linii (9). Klamry są jakby spółwkiem całości instrukcji złożonej, która ma zostać wykonana, jeśli jest spełniony warunek z linii (3). Przykładowo, jeśli podamy liczbę dodatnią, na ekranie monitora po zakończeniu działania programu będzie wyświetlony napis:

```
Podaj liczbę całkowitą 9
Podajesz dodatnią liczbę
Dobrze, bo to ma być długość odcinka
Narysuje więc odcinek o długości 9
Dziękuje, koncze działanie...
```

O to właśnie nam chodziło. Program ma nas poinformować, że podana liczba będzie mu przydatna, bo narysuje odcinek o takiej długości. W wypadku wpisania liczby, która nie jest dodatnia, na ekranie możemy zobaczyć:

```
Podaj liczbę całkowitą -8
Dziękuje, koncze działanie...
```

Gdy podaliśmy liczbę ujemną, działanie programu kończy się tylko po dziękowaniem, nic ponadto nie zostanie wykonane w programie - wszak długość odcinka nie może być liczbą ujemną.

Co się zmieni, jeśli usuniemy klamry wiążące kilka instrukcji w jeden blok instrukcji? Sprawdźmy działanie programu po wykonaniu tej modyfikacji:

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

int main()
{
    int dlugosc;
    cout << "Podaj liczbę całkowitą "; // (1)
    cin >> dlugosc; // (2)
    if (dlugosoo)
        // tu usuneliśmy klamrę otwierającą! (4)
        cout << endl; // (5)
    cout << "Podajeś dodatnią liczbę" << endl; // (6)
    cout << "Dobrze, bo to ma być długość odcinka" << endl; // (7)
    cout << "Narysuje więc odcinek o długości " << dlugosc; // (8)
        // tu usuneliśmy klamrę zamkającą! (9)
    cout << "\n\nDziękuje, koncze działanie...";
    cin.ignore();
    getchar();
    return 0;
}
```

Uruchomimy teraz program dla tych samych wartości. Najpierw podamy tak samo jak poprzednio liczbę dodatnią:

```
Podaj liczbę całkowitą 9
Podajeś dodatnią liczbę
Dobrze, bo to na być długość odcinka
Narysuje więc odcinek o długości 9
Dziękuje, koncze działanie...
```

Nic się nie zmieniło. Podajmy więc liczbę -8:

```
Podaj liczbę całkowitą -8
Podajeś dodatnią liczbę
Dobrze, bo to na być długość odcinka
Narysuje więc odcinek o długości -8
Dziękuje, koncze działanie...
```

Skoro ciąg instrukcji nie był ujęty w klamry, instrukcja, która ma się wykonać w wypadku spełnienia warunku z linii (3), jest instrukcją pojedynczą, czyli wyprowadzeniem na ekran monitora nowej linii określonej w linii (5), pozostałe zaś nie dotyczą już instrukcji warunkowej. Dlatego też pozostałe instrukcje zostaną wykonane za każdym razem, jako nieodnoszące się do instrukcji warunkowej. W przykładach prostych, jak nasz powyżej, szybko zauważysz pomyłkę, ale błąd tego samego ro-

## 2. Wstęp do programowania w języku C++

dzaju w programach bardziej złożonych może być czasem trudny do znalezienia. W unikaniu tego typu błędów bardzo pomocne jest stosowanie wcięć w pisany kodzie. Przejrzystość zapisu i grupowanie instrukcji pomaga w analizie kodu, a w razie nieprawidłowego działania programu - w znalezieniu błędu. Zasady robienia wcięć łatwo poznasz, śledząc sposób formatowania tekstów w programach, które zamieściliśmy w podręczniku. Możesz również wypracować własny sposób formatowania treści programu, ważne jest, aby konsekwentnie się go trzymać.

Zagnieżdżanie :  
instrukcji warunkowej •

Instrukcje warunkowe mogą być zagnieżdżane wewnątrz siebie. Przeanalizuj przykład wzięty z życia:

*Jeśli do jutra wyzdrowię, to*

*jeśli będzie ładna pogoda, pójdę na spacer,*

*w przeciwnym wypadku będę czytał książkę;*

*w przeciwnym wypadku // czyli jeśli nie wyzdrowię*

*jeśli będę się czuł bardzo źle, będę leżał w łóżku,*

*w przeciwnym wypadku zaproszę kolegę do domu*

Oczywiście, moglibyśmy tę wypowiedź jeszcze bardziej rozbudować (np.: *Jeśli będzie ładna pogoda, to jeśli kolega pożyczy mi rower, pojadę nad nekę, w przeciwnym wypadku pójdę na spacer*), tworząc kolejne poziomy zagnieżdżenia instrukcji warunkowych. Liczba zagnieżdżeń powinna być jednak niezbyt duża, gdyż wraz z kolejnymi zagnieżdżeniami kod źródłowy robi się coraz mniej czytelny. Jeśli już je stosujesz, nie zapomnij o wcięciach w zapisie i komentarzach, które pomogą ci później analizować kod własnego programu.

Napiszemy teraz program, który wykorzystuje zagnieżdzoną instrukcję warunkową.



### Przykład

Program wyprowadza na ekran monitora najmniejszą z trzech podanych przez użytkownika wartości całkowitych.

W programie założyliśmy, że podawane przez użytkownika liczby będą całkowite, a zmienne nazwiemy *a*, *b*, *c*.

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

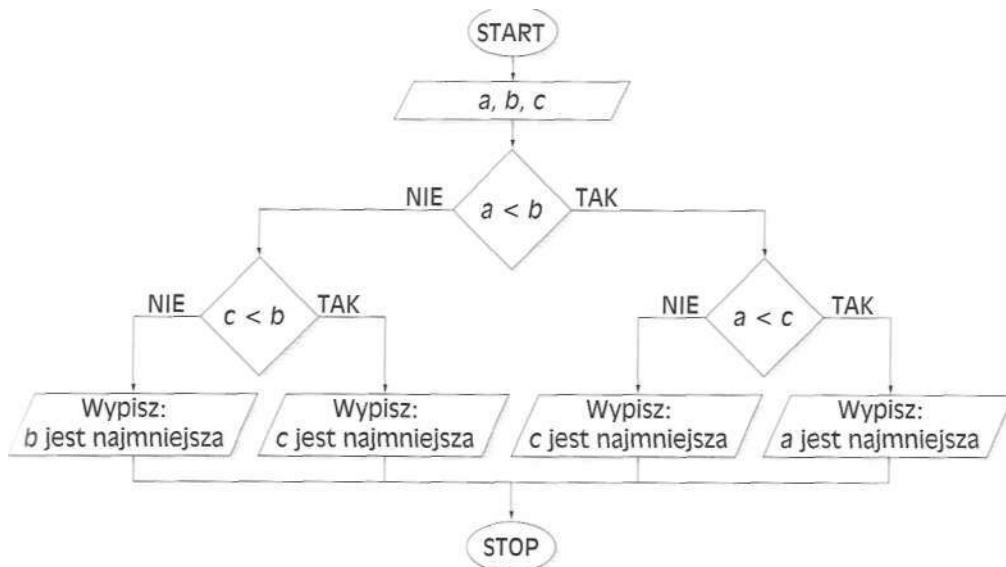
int main()
{
    int a,b,c;
    cout << "Podaj pierwszą liczbę ";
    cin >> a;
    cout << "Podaj drugą liczbę ";
    cin >> b;
    cout << "Podaj trzecią liczbę ";
    cin >> c;
    if (a < b)
        cout << "Najmniejsza wartość to " << a;
    else
        cout << "Najmniejsza wartość to " << b;
    return 0;
}
```

```

    cin >> b;
    cout << "Podaj trzecia liczbę ";
    cin >> c;
    if (a<b)
        if (a<c)
            cout << "Najmniejsza z podanych liczb jest " << a;
        else
            cout << "Najmniejsza z podanych liczb jest " << c;
    else
        if (c<b)
            cout << "Najmniejsza z podanych liczb jest " << c;
        else
            cout << "Najmniejsza z podanych liczb jest " << b;
    cin.ignore();
    getchar();
    return 0;
}

```

Analiza kodu programu nie powinna ci sprawić trudności; dla ułatwienia przedstawiamy schemat blokowy algorytmu (ryc. 2.2), który jest realizowany w programie. Wskaż na nim warunki, które są zagnieżdżone, i warunek zewnętrzny.



Ryc. 2.2. Schemat blokowy algorytmu wyświetlającego na ekranie najmniejszą liczbę z trzech podanych na wejściu

### 2.5.2. Instrukcja wyboru switch

W programie może się zdarzyć, że należy wybrać jeden z wielu sposobów postępowania, zależnie od wartości zmiennej. Spytajmy na przykład ucznia, która godzina lekcyjna właśnie się zaczęła, i na podstawie odpowiedzi wypiszmy informację o aktualnie odbywającym się przedmiocie (używać tu będziemy operatora relacji `==`, który został zasygnalizowany w rozdziale pierwszym). Oto stosowny program:

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

int main()
{
    int lekcja;
    cout << "Która godzina lekcyjna się zaczęła?
    cin >> lekcja;
    if (lekcja == 1) cout << "Masz teraz matematykę";
    if (lekcja == 2) cout << "Masz teraz fizykę";
    if (lekcja == 3) cout << "Masz teraz j. polski";
    if (lekcja == 4) cout << "Masz teraz historię";
    if (lekcja == 5) cout << "Masz teraz geografie";
    if (lekcja == 6) cout << "Masz teraz informatykę";
    if (lekcja > 6) cout << "Jesteś już po lekcjach";
    cin.ignore();
    getchar();
    return 0;
}
```

Program prosty i czytelny, ale jego działanie jest nieoptymalne. Założymy, iż użytkownik poda, że właśnie zaczęła się pierwsza lekcja. Już przy pierwszej instrukcji warunkowej zachodzi prawdziwość określonego warunku, zostanie więc wypisany komunikat, że jest to lekcja matematyki. Teraz wykona się kolejna instrukcja programu, a zatem będzie sprawdzany warunek, czy zmieniona lekcja ma wartość 2. Oczywiście nie ma tej wartości, a więc wykona się sprawdzanie kolejnego warunku, czyjej wartość wynosi 3, aż do sprawdzenia, czy jest większa od 6. Można teraz zapytać, po co sprawdzać kolejne warunki, skoro już pierwszy z nich jest prawdziwy. Ponieważ warunki określone w kolejnych instrukcjach wykluczają się wzajemnie, to spełnienie jednego z nich powinno zablokować sprawdzanie kolejnych, których zajście jest już i tak niemożliwe. Masz rację, ale do zadań procesora należy wykonywanie wszystkich instrukcji po kolei. Jak zatem skonstruować kod, aby uniknąć badania warunków, które i tak nie zajdą, jeśli jeden z wcześniejszych został spełniony? Problem ten rozwiązuje dodanie po każdym warunku if instrukcji else - możesz samodzielnie zapisać taki kod programu. Z pewnością zauważysz, że takie postępowanie wymaga wiele pisania i jest mało czytelne. Zawią konstrukcję warunków zagnieżdżonych możemy zastąpić instrukcją switch.

**Składnia instrukcji switch**

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

Jeśli wartość wyrażenia odpowiada którejś z wartości podanej przy jednej z etykiet `case`, wówczas wykonana zostanie instrukcja przy tej właśnie etykiecie. Po napotkaniu instrukcji `break` następuje przekazanie sterowania do pierwszej instrukcji następującej po `switch` – a zatem nie są sprawdzane kolejne przypadki, których zajście i tak jest już niemożliwe. Jeśli wyrażenie nie przyjmuje żadnej z wartości określonych w etykietach `case`, to wykonana zostanie instrukcja oznaczona etykietą `default`. Etykietą `default` i instrukcja przy niej nie są konieczne, wtedy – jeśli żadna z etykiet `case` nie ma wartości odpowiadającej warunkowi – po prostu nie wykona się żadna instrukcja wewnętrz instrukcji `switch`. Na schemacie blokowym zamieszczonym na następnej stronie (ryc. 2.3) przedstawiamy sposób działania instrukcji `switch`.

Zastosujmy instrukcję wyboru `switch` do poprzedniego przykładu:

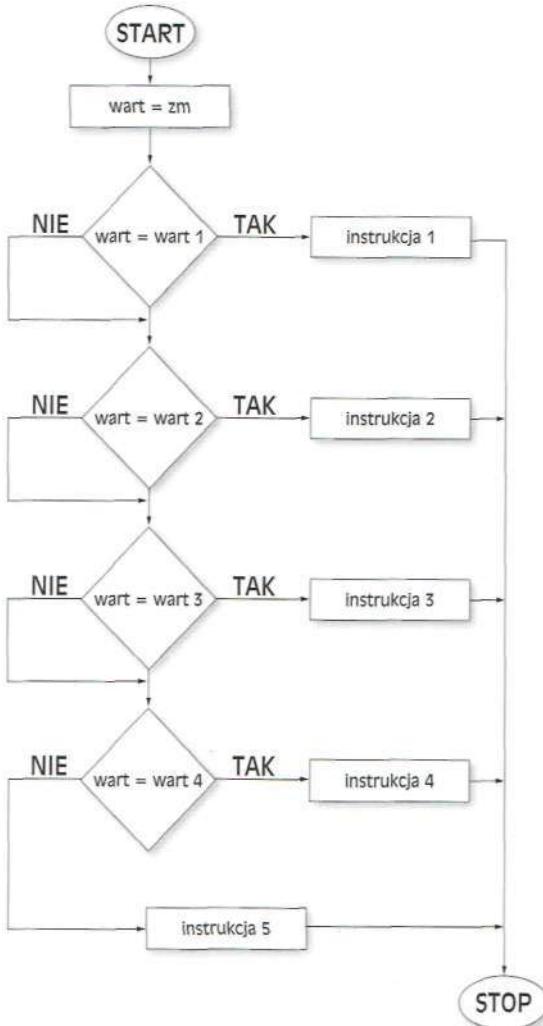
---

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

int main()
{
    int lekcja;
    cout << "Ktora godzina lekcyjna sie zaczela? ";
    cin >> lekcja;
    switch (lekcja)
    {
        case 1: cout << "Masz teraz matematyke"; break;
        case 2: cout << "Masz teraz fizyke"; break;
        case 3: cout << "Masz teraz j. polski"; break;
        case 4: cout << "Masz teraz historie"; break;
        case 5: cout << "Masz teraz geografie"; break;
        case 6: cout << "Masz teraz informatyke"; break;
        default: cout << "Jestes juz po lekcjach"; break;
    }
    cin.ignore();
    getchar();
    return 0;
}
```

---

## 2. Wstęp do programowania w języku C++



Ryc. 2.3. Schemat blokowy instrukcji switch

Jeśli skonstrujesz swój program w ten sposób, zmniejszysz liczbę porównań zmiennej z wartościami, które mogła przyjąć, a przy tym twój kod źródłowy stanie się bardziej czytelny.

Instrukcję switch można również wykorzystać bez używania instrukcji break. Robimy tak w sytuacjach, gdy chcemy, aby wykonały się instrukcje zarówno występujące po etykiecie znalezionej wartości, jak i wszystkie występujące po kolejnych etykietach. Dla przykładu popatrz na efekt działania programu, umieszczony bezpośrednio po jego kodzie:

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

int main()
{
    int lekcja;
    cout << "Ktora godzina lekcyjna sie zaczela? ";
    cin >> lekcja;
    cout << "Pozostaly ci dzisiaj jeszcze nastepujace lekcje:" << endl << endl;
    switch (lekcja)
    {
        case 1: cout << "matematyka" << endl;
        case 2: cout << "fizyka" << endl;
        case 3: cout << "j. polski" << endl;
        case 4: cout << "historia" << endl;
        case 5: cout << "geografia" << endl;
        case 6: cout << "informatyka" << endl;
    }
    cin.ignore();
    getchar();
    return 0;
}
```

2.15

```
Ktora godzina lekcyjna sie zaczela? 4
Pozostaly ci dzisiaj jeszcze nastepujace lekcje:
historia
geografia
informatyka
```

### 2.5.3. Instrukcja pętli for

Znasz już sposób postępowania w wypadku, gdy dana czynność ma się powtórzyć wiele razy: stosujesz wtedy instrukcje iteracyjne, czyli pętle. W języku C++ mamy trzy instrukcje pętli. Pierwszą z nich jest pętla **for**, której będziemy zwykle używać w sytuacjach, gdy liczba przebiegów jest ustalona jeszcze przed wejściem do pierwszego obiegu pętli. Patrz na poniższy przykład:

```
for (int i=0; i<10; i=i+1)
{
    cout << i << ", ";
}
```

Zamieszczony fragment kodu odpowiada za wypisanie na ekranie monitora wszystkich dziesięciu cyfr z przecinkiem umieszczonym po każdej z nich. Po słowie kluczowym **for** występuje nawias, w którym wpisuje się warunek sterujący pętlą. Po nawiasie podaje się instrukcję,

która ma być powtarzana w pętli. Można w tym miejscu umieścić blok instrukcji, wtedy wszystkie one będą się wykonywały w pętli. Chociaż w naszym przykładzie w pętli wykonuje się tylko jedna instrukcja, ujęliśmy ją w klamry, nie jest to jednak konieczne.

### Składnia pętli for

for (instrukcja początkowa; warunek sterujący; instrukcja kroku)  
instrukcja;  
gdzie:

#### instrukcja początkowa

- instrukcja wykonana przed pierwszym obiegiem pętli, zwana również inicjującą
- wyrażenie, którego logiczna wartość jest badana przed każdym obiegiem pętli - jeśli jego wartość jest true (czyli jest różna od zera), to pętla wykona się kolejny raz, w przeciwnym wypadku następuje wyjście z pętli
- instrukcja wykonana po każdym przebiegu pętli, najczęściej modyfikuje tak zwanego licznik pętli

warunek sterujący

#### instrukcja kroku

Rozważmy teraz prosty przykład wyświetlający w kolumnie liczby od 0 do 20. Przy liczbach niepodzielnych przez 3 znajduje się odpowiedni komentarz. Oto program:

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

int main()
{
    for (int i=0; i<20; i++)
    {
        cout << i;
        if (i%3!=0) // jeśli reszta z dzielenia przez 3 jest różna od zera
            cout << " - nie jest podzielne przez 3, ";
        cout << endl;
    }
    getchar();
    return 0;
}
```

2.17

Instrukcja kroku postaci `i++` jest instrukcją inkrementacji zmiennej `i` odpowiada wyrażeniu `i = i + 1`. Ten bardzo wygodny skrót bywa używany nie tylko w pętlach.

instrukcja inkrementacji w C++

`i++` odpowiada `i = i+1`

instrukcja dekrementacji w C++

`i--` odpowiada `i = i-1`

Pętla **for** w C++ jest bardzo elastyczna. Możemy na przykład w tej retli pominąć dowolną instrukcję występującą w nawiasie, musimy jednak zachować średnik. Pętla ta może wyglądać tak: `for( ; ; )`, co oznacza pętlę nieskończoną. Brak warunku sterującego oznacza, że jest on zawsze prawdziwy. Pętla ta nie wymaga żadnego licznika, dzięki czemu można ją upodobnić w działaniu do pętli, które omawiamy w dalszej części rozdziału.

#### 2.5.4. Instrukcja pętli while

Kolejną pętlą w C++ jest pętla `while`. Pętla ta wykonuje się kolejny raz, gdy wyrażenie jest prawdziwe (czyli ma wartość różną od zera).

##### Składnia pętli while

`while (wyrażenie) instrukcja;`

gdzie:

wyrażenie            -wyrażenie przyjmujące wartość logiczną  
                       „prawda” albo „fałsz”

instrukcja            - instrukcja wykonywana w pętli

Napiszmy prosty program wyświetlający na ekranie znaki podane z klawiatury do momentu, aż podamy znak „k”.

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

int main()
{
    char z;
    cout << "Podaj znak ";
    cin >> z;
    while (z != 'k')
    {
        cout << "Podaj kolejny znak ";
        cin >> z;
    }
    cout << "Podales " << z << " wiec koncze";
    cin.ignore();
    getchar();
    return 0;
}
```

2.18

Jeśli w tym programie użytkownik poda na początku znak „k”, pętla w `while` nie wykoná się nawet jeden raz.

Przedstawimy teraz nieco bardziej skomplikowany przykład z zastosowaniem pętli `while`: Program oblicza wynik dzielenia całkowitego dwóch liczb podanych z klawiatury. Algorytm polega na wielokrotnym odejmowaniu dzielnika od dzielnej i liczeniu wykonanych powtórzeń w pętli `while`.

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

int main()
{
    int a, b;                      // a - dzielna,   b - dzielnik
    int k = 0;                      // krotność
    cout << "Podaj dzielna i dzielnik ";
    cin >> a >> b;
    if (b==0)                      // sprawdzenie warunku, czy nie dzielimy przez zero
        cout << "Nie wolno dzielic przez zero!!!";
    else
    {
        while (a>=b)              // tu rozpoczyna się właściwy algorytm dzielenia
        {
            a = a-b;
            k++;
        }
        cout << "a/b wynosi " << k;
    }
    cin.ignore();
    getchar();
    return 0;
}
```

2.19

Pętla while sprawdza warunek przed wykonaniem instrukcji wewnętrz pętli, instrukcja ta zatem może nie wykonać się wcale.

### 2.5.5. Instrukcja pętli do ... while

Poznasz teraz pętlę do ... while. Pętla ta wykonuje się kolejny raz, gdy wyrażenie jest prawdziwe (czyli ma wartość różną od zera).

Składnia pętli do ... while

do instrukcja while (wyrażenie);

gdzie:

instrukcja

- instrukcja wykonywana w pętli

wyrażenie

- wyrażenie przyjmujące jedną z dwóch wartości logicznych: prawda albo fałsz

Zacznijmy od prostego programu rysującego na ekranie 4 gwiazdki:

---

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

int main()
{
    int i = 0;           // definicja zmiennej
    do                 // rób...
    {
        cout << '*';   // klamra otwierająca blok instrukcyjny
        i++;            // wypisz gwiazdkę
    }                   // zwiększą wartość zmiennej i
    while (i<4);       // ...dopóki i jest mniejsze od czterech
    getchar();
    return 0;
}
```

Program będzie działał tak samo, jeśli zamiast warunku, którego prawdziwość jest sprawdzana, umieścimy w ostatniej linii wyrażenie, którego wartość będzie sprawdzana. Zgodnie z semantyką pętli, czyli opisem jej działania, dopóki wyrażenie nie ma wartości 0, będzie się wykonywał kolejny przebieg tej pętli.

---

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

int main()
{
    int i = 4;           // definicja i nadanie wartości początkowej zmiennej
    do                 // rób...
    {
        cout << "*";   // klamra otwierająca blok instrukcyjny
        i--;            // zmniejsz wartość zmiennej i
    }                   // klamra zamknięcia bloku instrukcyjnego
    while (i);          // ...dopóki wyrażenie ma wartość różną od zera
    getchar();
    return 0;
}
```

2.21

Zadanie, które zostanie wykonane (narysowanie 4 gwiazdek), pokazuje mechanizm działania pętli do ... while, ale równie dobrze może być zrealizowane za pomocą pętli for. Z góry założyliśmy, że chcemy otrzymać 4 gwiazdki na ekranie monitora.

Przyjrzymy się następnemu przykładowi, który ilustruje bardzo częsty sposób użycia pętli do ... while:



### Przykład

Napiszmy program obliczający pole kwadratu o podanej długości boku. W wypadku podania przez użytkownika niewłaściwej wartości, to jest: ujemnej lub zero, program powinien prosić o podanie właściwej wartości.

Przeanalizuj poniższy kod programu:

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

int main()
{
    float bok;
    do // rób...
    {
        cout << "Podaj dodatnia dlugosc boku! ";
        cin >> bok; // zmiennej bok została przypisana wartość
        if (bok<=0)
            cout << "Niepoprawna dlugosc boku - podaj jeszcze raz!" << endl;
    }
    while (bok<=0); // ...dopóki bok nie jest wartością dodatnią
    cout << "Pole kwadratu wynosi: " << bok*bok;
    cin.ignore();
    getchar();
    return 0;
}
```

2.22

Warunek pętli do ... while jest sprawdzany po jej wykonaniu, a więc bez względu na to, jaką wartość podamy, pętla wykona się chociaż raz. Pętla wykona się jednokrotnie, gdy za pierwszym razem podamy wartość dodatnią. W wypadku podania wartości, która nie może być bokiem kwadratu, pojawi się komunikat o błędnej wartości i ponownie wykona się pętla. Tak będzie aż do momentu, gdy podamy dodatnią długość boku.

Pętla do ... while wykonuje się co najmniej jeden raz w programie, ponieważ warunek wyjścia jest sprawdzany na końcu.

Warto wiedzieć, że w C++ każdą pętlę można zastąpić inną. Twórcy języka przewidzieli trzy rodzaje pętli, ponieważ pewne zagadnienia łatwiej i przejrzystiej można zapisać jednym rodzajem pętli, natomiast w innych wypadkach bardziej użyteczne mogą być pozostałe pętle.

#### 2.5.6. Instrukcje break i continue sterujące wykonaniem pętli

Instrukcje wymienione w tym rozdziale nie powinny się pojawić w dwóch programach. Używanie tych instrukcji nie jest zgodne z kanonem dc

brego programisty i sprawia, że program staje się mało czytelny. Podajemy je jednak dla lepszego zrozumienia innych programów napisanych z ich użyciem.

Przy okazji instrukcji `switch` przedstawiliśmy typowe użycie instrukcji `break`. Teoretycznie, instrukcja ta może być zastosowana we wszystkich wymienionych wcześniej pętlach w celu ich natychmiastowego przerywania, jednak odradzamy takie jej użycie.

Spójrz na przykład programu, który wyświetla liczby parzyste, dopóki ich suma nie przekroczy 20:

```
=include <iostream>
=include <cstdio>
using namespace std;

int main()
{
    int sum = 0;
    for (int i=0; i<10; i++)
    {
        cout << 2*i <<, ";
        sum = sum+2*i;
        if (sum>20)
            break;                                // jeśli warunek spełniony, wyjście z pętli
    }
    cout << "Jestem poza pętlą";           // tu zaczyna się realizacja zadań
    getchar();
    return 0;
}
```

W pierwszej linii znajdującej się wewnętrz pętli `for` wyprowadzana jest na ekran monitora podwojona wartość zmiennej `i`. Drugą instrukcją w pętli jest obliczanie sumy liczb już wyprowadzonych na ekran monitora. Następnie sprawdzany jest warunek, czy suma ta przekroczyła wartość 20. Jeśli tak, występuje wyjście z pętli, realizowane za pomocą instrukcji `break`, czyli przerwana zostaje instrukcja iteracji. Na ekran monitora zostaną wyprowadzone liczby: 0, 2, 4, 6, 8, 10. Przy szóstym przebiegu pętli warunek określający wyjście z pętli został spełniony, a więc nie będzie już kolejnej iteracji. Wyświetlony zostanie napis: `Jestem poza pętlą` i wykonywać się będą pozostałe instrukcje, których już w przykładzie nie ujęliśmy.

Liczba instrukcji wykonanych w pojedynczym przebiegu pętli można modyfikować za pomocą instrukcji `continue` - po jej napotkaniu nie zostaje przerwane działanie całej pętli, a jedynie tego konkretnego przebiegu, który jest realizowany, i następuje przejście do kolejnego. Zatem w przebiegu pętli część instrukcji zostaje pominiętych.

Instrukcje sterujące  
pętlą `break`  
i `continue`

Oto przykład programu:

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

int main()
{
    for (int i=0; i<5; i++)
    {
        cout << endl << i << ", ";
        if (i%3 == 0)
            continue;          // przy spełnieniu warunku przechodzimy
                                // natychmiast do kolejnego przebiegu pętli
        cout << " - nie jest podzielne przez 3, ";
    }
    getchar();
    return 0;
}
```

2.24

Na ekranie odczytamy:

```
0,
1, - nie jest podzielne przez 3,
2, - nie jest podzielne przez 3,
3, - nie jest podzielne przez 3,
```

Dla liczb 0 i 3 spełniony był warunek zerowania się reszty z dzielenia liczb przez 3 (% jest operatorem reszty z dzielenia), a więc wykonała się instrukcja `continue` - ignorujemy dalsze instrukcje określone w pętli (czyli wyprowadzenie na ekran monitora napisu: `nie jest podzielne przez 3`) i przechodzimy do jej następnego obiegu. Dlatego dla tych właśnie liczb nie został wyświetlony tekst. Pamiętaj, że to jest tylko przykład na zastosowanie instrukcji `continue`. Gdybyśmy jednak rzeczywiście chcieli programowo rozwiązać to zadanie, czytelniej wyglądałaby implementacja w sposób przedstawiony poniżej:

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

int main()
{
    for (int i=0; i<5; i++)
    {
        cout << endl << i << ", ";
        if (i%3 != 0)
            cout << " - nie jest podzielne przez 3,";
    }
    getchar();
    return 0;
}
```

2.



### Pytania kontrolne

1. Co to jest bit?
2. Ile bajtów ma 1 kilobajt?
3. Wymień kolejne etapy tworzenia programu.
4. Wyjaśnij pojęcia: kompilator, linker, debugger.
5. Co to jest kod maszynowy?
6. Wymień podstawowe typy danych w języku C++.
7. Wśród wymienionych poniżej nazw zmiennych, oddzielonych przecinkami, wskaż niepoprawne: liczba, k2 3, moja liczba, iks, x.
8. Co to jest błąd syntaktyczny?
9. Podaj składnię instrukcji warunkowej i przykłady jej zastosowania.
10. Co to jest iteracja? Podaj rodzaje pętli w języku C++.
11. Podaj składnię wszystkich znanych ci instrukcji sterujących.

### Ćwiczenia

1. Jaką największą liczbę binarną można zapisać w siedmiu bitach?  
Podaj jej dziesiętną wartość.
2. Oblicz wartość następujących wyrażeń:
  - a)  $11101010_{(2)} + 111101_{(2)}$ ;
  - b)  $1000101_{(2)} - 1111_{(2)}$ ;
  - c)  $1001101_{(2)} * 101_{(2)}$ .
3. Narysuj schemat blokowy algorytmu obliczającego pole prostokąta o długościach boków podanych przez użytkownika i napisz program realizujący to zadanie (program ma zadbać, aby były to poprawne wartości).
4. Narysuj schemat blokowy algorytmu, który pobiera z klawiatury liczby podane przez użytkownika aż do momentu, gdy ich suma będzie większa niż 50. Napisz program realizujący to zadanie.
5. Napisz program, który wypisze na ekranie monitora kolejne liczby naturalne z przedziału (23, 45) oddzielone spacjami.
6. Narysuj schemat blokowy algorytmu wypisującego na ekranie monitora wszystkie dwucyfrowe liczby parzyste podzielne przez 3. Napisz program realizujący to zadanie.
7. Narysuj schemat blokowy algorytmu, który wyświetla na ekranie monitora kwadraty kolejnych liczb naturalnych, począwszy od zera a skończywszy na kwadracie liczby, która jest podana na początku działania. Napisz program realizujący to zadanie.

8. Przeanalizuj poniższy program i powiedz, jaka liczba zostanie wyprodukowana na ekran monitora:

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

int main()
{
    int a = 1, s = 0;
    for (int i=1; i<5; i++)
    {
        if (i<3)
        {
            a = 4;
            s = s+a;
        }
        else
        {
            a = 2;
            s = s+2*a;
        }
    }
    cout << s;
    getchar();
    return 0;
}
```

## 3. Funkcje w C++

W tym rozdziale omawiamy mechanizm tworzenia programów opartych na zdefiniowanych funkcjach. Dowiesz się, co to jest funkcja i w jaki sposób się ją definiuje. Nauczysz się wykorzystywać w programie zdefiniowane przez siebie funkcje. Zapoznamy cię też z różnymi sposobami przekazywania argumentów do funkcji i pokażemy przykłady funkcji, które przyjmują jakąś wartość, oraz takich, które wartości nie przyjmują. Rozważania na temat użycia funkcji zilustrujemy wieloma przykładami.

### 3.1. Definiowanie i wywoływanie funkcji

Pamiętasz zapewne, że każdy program napisany w języku C++ musi zawierać funkcję o nazwie main. Jest ona w programie funkcją główną, ale zwykle nie jedyną. Aby programowanie było łatwiejsze, poszczególne fragmenty programu (stanowiące logiczne całości) będziemy definiować w postaci funkcji. Napisany w ten sposób program łatwiej się analizuje, a część zdefiniowanych funkcji możemy wykorzystać w innych pisanych przez siebie programach, kopiując po prostu fragment kodu do nowego programu. Funkcję warto stosować w programie również w sytuacji, gdy jest w nim często wykorzystywany jakiś ciąg instrukcji. Możemy go wówczas wyodrębnić i umieścić w zdefiniowanej funkcji, co dodatkowo zmniejszy długość kodu programu. Teraz za każdym razem gdy chcemy, aby wykonał się ciąg instrukcji, uruchomimy funkcję, w której go umieściliśmy - nie musimy więc kolejno wpisywać tych instrukcji.

Funkcja w języku C++ jest jakby podprogramem, który możesz wykorzystać w dowolnym miejscu pisanego przez siebie programu. Aby ednak korzystać z funkcji, musisz ją najpierw zdefiniować, a definicję umieścić w ścisłe określonym miejscu kodu programu. Definicja funkcji musi się znajdować przed pierwszym jej użyciem.

Zalety użycia zdefiniowanych funkcji w programie

#### Składnia definicji funkcji

```
typ_wyniku nazwa_funkcji(zestaw parametrów formalnych)
{
    wnętrze funkcji;
}
```

### 3. Funkcje w C++

Parametry formalne funkcji :

Zestawem parametrów formalnych są nazwy obiektów przekazywanych do funkcji wraz z podaniem ich typów, oddzielone przecinkami, występujące w definicji funkcji. Parametry funkcji umieszcza się w nawiasach. Może się zdarzyć, że funkcja nie ma żadnych parametrów - jest bezparametrowa. Wtedy po prawej stronie nazwy funkcji wpisuje się puste nawiasы.

Wnętrze funkcji, zwane też treścią funkcji, to instrukcje i operacje, które mają zostać wykonane w trakcie działania funkcji, wraz ze zbiorem zmiennych pomocniczych koniecznych do ich wykonania. Zasady tworzenia nazw funkcji są takie same, jak zasady tworzenia nazw zmiennych (opisane w poprzednim rozdziale). Poprawna jest na przykład nazwa funkcja\_moduł, niepoprawna zaś: funkcja\_moduł.

Przyjrzyjmy się zamieszczonemu niżej przykładowi.



#### Przykład

Zdefiniujmy funkcję obliczającą średnią arytmetyczną dwóch liczb rzeczywistych.

Oto przykład tej funkcji:

```
float srednia_aryt(float a, float b) // typ wyniku nazwa(parametry formalne)
{
    return (a+b)/2; // wynikiem funkcji jest średnia arytmetyczna
                    // dwóch zmiennych
} // klamra zamykająca wnętrze funkcji
```

3.1

Tak zdefiniowaną funkcję będziemy mogli użyć w programie, czyli wywołać ją. Wywołanie funkcji to napisanie jej nazwy łącznie z nawiasem, w którym znajdują się ewentualne argumenty przekazywane funkcji.

Argumenty przekazane wywoływanej funkcji nazywamy inaczej parametrami aktualnymi.

Przymieranie wartości przez funkcję

Zauważ, że wnętrze funkcji kończy instrukcja `return (a+b)/2;` Pojawienie się w jakiekolwiek funkcji (również main) instrukcji `return;` natychmiast kończy jej działanie, a sterowanie programem zostaje przekazane do funkcji, która wywołała właśnie zakończoną funkcję. Jeśli instrukcja zostanie wpisana w sposób: `return zmienna;`, to funkcja zakończy działanie i dodatkowo przyjmie wartość równą zmiennej.

Zdefiniowana przez nas funkcja pobiera dwie liczby rzeczywiste i przyjmuje wartość ich średniej arytmetycznej (będącą również liczbą rzeczywistą, o czym informuje nas nagłówek funkcji).

Oto program, który wykorzystuje zdefiniowaną funkcję:

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

float srednia_aryt(float a, float b)
{
    return (a+b)/2;
}

int main()
{
    cout << "Srednia arytmetyczna liczb 2.7 i 5 = ";
    cout << srednia_aryt(2.7,5);
    getchar();
    return 0;
}
```

Zwróć uwagę na budowę tego krótkiego programu. Na początku zostały dołączone pliki biblioteczne, następnie umieściliśmy definicję funkcji `srednia_aryt`, a dopiero potem napisaliśmy funkcję `main`. Jest to zgodne z zasadą obowiązującą przy programowaniu w języku C++, mówiącą, że każda nazwa musi zostać zadeklarowana przed jej pierwszym użyciem. Skoro więc w funkcji `main` wywołujemy funkcję `Srednia_aryt`, zdefiniowaliśmy ją przed funkcją główną.

**Nie wolno definiować funkcji we wnętrzu innej funkcji.**

W funkcji `main` funkcja `srednia_aryt` została wywołana wraz z liczbami, dla których ma się wykonać -jak już wiesz, są to argumenty funkcji. Argumenty umieszcza się w nawiasie po nazwie funkcji. Podajemy je bez ich typów.

### 3.2. Zakres ważności zmiennych

Rozpatrzmy teraz przykład, w którym zamiast wartości przekazaliśmy funkcji zmienne. Funkcja `main` ma teraz postać:

```
int main()
{
    float x, y; // zmienne lokalne funkcji main
    cout << "Podaj dwie liczby: ";
    cin >> x >> y;
    cout << "Srednia arytmetyczna tych liczb wynosi: ";
    cout << srednia_aryt(x,y);
    cin.ignore();
    getchar();
    return 0;
}
```

### 3. Funkcje w C++

Zmienne lokalne

Zmienne, które są zadeklarowane we wnętrzu funkcji, to **zmienne lokalne** funkcji. Nie są one znane poza wnętrzem funkcji. **We wnętrzu funkcji zmienna jest znana od miejsca jej deklaracji do klamry zamkającej blok, w którym zmienna została zadeklarowana.**

Zmienne globalne

Zwróc więc uwagę, że zmienne zadeklarowane w funkcji main również są zmiennymi lokalnymi, tyle że funkcji głównej. Nie można się więc do nich odwoływać z wnętrza jakiekolwiek innej zdefiniowanej funkcji. Jeśli zaś zadeklarujemy zmienne poza wszystkimi funkcjami, to otrzymamy **zmienne globalne**. Będą one dostępne dla wszystkich funkcji zdefiniowanych pod nimi.

W powyższym przykładzie zmienne *x* i *y* były zmiennymi lokalnymi funkcji głównej, aby więc funkcja *srednia\_aryt* miała do nich dostęp, musiały zostać jej przekazane jako argumenty. Gdybyśmy zadeklarowali te zmienne jako globalne, nie byłoby potrzeby przekazywania jakichkolwiek argumentów do funkcji. W takim wypadku funkcja działałaby tylko i wyłącznie na zmiennych globalnych *a* i *b*, co jednak znacznie zawężałoby wykorzystanie funkcji w bardziej rozbudowanych programach, a także mogłoby zachętać do niewłaściwego stylu programowania.

W poniższym przykładzie funkcja nie pobiera żadnych argumentów, działa na zmiennych globalnych.

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

float a, b; // zmienne globalne - wszystkie
// zdefiniowane poniżej funkcje mają do nich dostęp

float srednia_aryt()
{
    return (a+b)/2; // funkcja srednia_aryt ma dostęp do a i b
}

int main()
{
    cout << "Podaj dwie liczby: ";
    cin >> a >> b; // funkcja main ma dostęp do a i b
    cout << "Srednia arytmetyczna tych liczb wynosi: " << srednia_aryt();
    cin.ignore();
    getchar();
    return 0;
}
```

3.4

Funkcja, która nie operuje na zmiennych globalnych, jest dużo bardziej uniwersalna od funkcji z tymi zmiennymi. Staraj się pisać programy, które nie korzystają ze zmiennych globalnych.

Dla lepszego zrozumienia charakteru zmiennych lokalnych i globalnych funkcji oraz zakresu ważności zmiennych przyjrzyj się przykładowi poniżej:

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

int a; // zmienna globalna, znają ją wszystkie funkcje programu

void wyswietl()
{
    cout << "Bok kwadratu ma długość: " << a;
}

float pole_kwadratu()
{
    return a*a;
}

float b; // tę zmienną globalną znają funkcje pole_prostokata i main

float pole_prostokata()
{
    int pole = a*b; // zmienna lokalna funkcji pole_prostokata
    return pole;
}

int main()
{
    int c; // zmienna lokalna funkcji main
    // tu znajduje się dalsza treść funkcji głównej
}
```

3.5

### 3.3. Sposoby przekazywania parametrów funkcji

Teraz zapoznamy cię z dwoma podstawowymi sposobami przekazywania argumentów do funkcji, która nie jest bezparametrowa: przez wartość lub przez referencję. Omówimy też funkcje bezparametrowe, to znaczy takie, które nie pobierają żadnych argumentów.

Zdefiniujmy funkcję, która oblicza wartość bezwzględną liczby rzeczywistej, inaczej moduł liczby. Funkcję tę wykorzystamy w programie obliczającym odległość punktów na osi liczbowej.

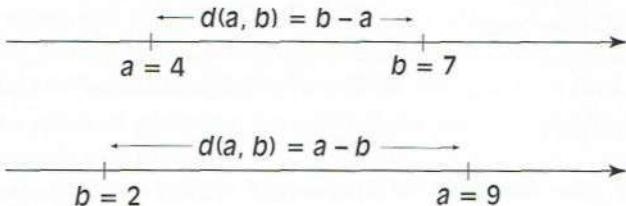
#### Przykład

Napiszemy program służący do obliczania odległości pomiędzy dwoma punktami na osi liczbowej.

**m**

### 3. Funkcje w C++

Program ten będzie obliczał odległość pomiędzy punktami na osi liczbowej podanymi przez użytkownika. Odległość dwóch punktów na osi dana jest wzorem  $d(a, b) = \begin{cases} a - b, & a \geq b \\ b - a, & a < b \end{cases}$ , co ilustruje rycina 3.1.



Ryc. 3.1. Wyznaczanie odległości pomiędzy dwoma punktami na osi liczbowej

Ogólnie zatem odległość pomiędzy punktami liczymy ze wzoru  $d(a, b) = |a - b|$ .

Napiszemy więc funkcję obliczającą moduł liczby rzeczywistej:

```
float modul(float a) // funkcja modul pobiera argument typu float
{
    if (a>=0)
        return a;
    else
        return -a;
}
```

3.6

Wynik działania funkcji zależy od wartości logicznej wyrażenia określonego w instrukcji warunkowej, stąd słowo `return` pojawiło się w definicji funkcji dwukrotnie, i program zostanie zakończony na jeden z dwóch sposobów. Teraz wykorzystamy tę funkcję w programie:

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

float modul(float x)
{
    if (x>=0)
        return x;
    else
        return -x;
}

int main()
{
    float a, b;
    cout << "Podaj dwa punkty na osi: ";
    cin >> a >> b;
```

```

cout << "Odleglosc pomiedzy punktami wynosi: " << modul(a-b);
cin.ignore();
getchar();
return 0;
}

```

3.7

Funkcja moduł została w funkcji głównej wywołana z argumentem będącym różnicą liczb podanych przez użytkownika.

Teraz zajmiemy się przykładem niematematycznego zastosowania funkcji: będziemy rysować szlaczki za pomocą funkcji napisanej na kilka różnych sposobów.

### Przykład

Zdefiniujmy funkcję rysującą szlaczek o podanej długości znakiem przekazanym funkcji.

/\ /&% /\\  
**n li n**

```

void szlaczek_dowolny(int i, char znak)
{
    for (int li=0; li<i; li++)           // li: zmienna pomocnicza
        cout << znak;
}

```

3.8

Funkcja ta pobiera znak oraz dodatnią liczbę całkowitą i rysuje szlaczek składający się z  $i$  znaków przekazanych funkcji jako wartość argumentu znak. Funkcji nie zostaje przypisana żadna wartość, o czym świadczy słowo void, oznaczające **typ pusty**. Zauważ, że nie pojawiło się też słowo return, ponieważ w funkcjach typu void nie musimy wpisywać instrukcji return;. Taka funkcja kończy się po napotkaniu klamry zamkającej treść funkcji.

### Przykład

Zdefiniujmy funkcję rysującą szlaczek składający się z dziesięciu gwiazdek - nie przekazujemy jej żadnych argumentów.

**ITT**

```

void szlaczek_10_gwiazdek(void)
{
    for (int p=0; p<10; p++)
        cout << "*";
}

```

3.9

Ten przykład to modyfikacja poprzedniej funkcji, tylko że w tym wypadku określone zostały znak (gwiazdka) i liczba powtórzeń, a zatem nie ma potrzeby pobierania tych wartości z zewnątrz. Dlatego w nawiasie, w którym umieszczamy parametry przekazywane do funkcji, znajduje się

: Funkcja  
: bezparametrowa

### 3. Funkcje w C++

stów void. Możemy pozostawić puste miejsce w nawiasach, będzie to tak samo odczytane przez kompilator.

Równoważne są zapisy:

```
void nazwa_funkcji() oraz void nazwa_funkcji(void)
```

Nie jesteśmy jednak ograniczeni do szłaczka złożonego z określonej liczby gwiazdek, nawet jeśli nie przekażemy funkcji żadnych argumentów z zewnętrz. Możemy skonstruować funkcję, której nie podamy z zewnątrz żadnych informacji, a funkcja sama zatroszczy się o ich pobranie:

```
void szlaczek_inaczej()
{
    cout << "podaj, jaki znak mam rysowac ";
    char znak;                                // to jest zmienna lokalna funkcji
    cin >> znak;                             // pobieramy informację, z jakich znaków
                                              // będzie się składał szlaczek
    cout << "ile mam narysowac znakow: " << znak << "? ";
                                              // długość szlaczka
    int ile;                                 // kolejna zmienna lokalna funkcji
    cin >> ile;                            // pobieramy informację, z ilu znaków
                                              // będzie się składał szlaczek
    for (int p=0; p<ile; p++)
        cout << znak;                      // rysowanie szlaczka
}
```

3.10

Zwróć uwagę, że do funkcji nie są przekazywane argumenty dotyczące długości szłaczka oraz znaku, z jakiego szlaczek będzie się składał. Te informacje pobiera funkcja w swoim wnętrzu. W tym celu zadeklarowane zostały dwie zmienne: znak oraz ile. Są to zmienne lokalne, z zakresem ważności w bloku funkcji. Są one znane funkcji, ale nie są znane nigdzie poza nią.

Przeanalizujmy pełny kod programu wykorzystującego przykłady wszystkich funkcji rysujących szłaczki, które wyżej zdefiniowaliśmy:

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

***** Definicje funkcji *****
void szlaczek_10_gwiazdek(void)
{
    for (int p=0; p<10; p++)
        cout << "*";
    cout << endl;
}

void szlaczek_dowolny(int i, char znak)
{
```

### 3.3. Sposoby przekazywania parametrów funkcji

```
for (int li=0; li<i; li++)
    cout << znak;
    cout << endl;
}

void szlaczek_inaczej()
{
    cout << "Podaj, jaki znak mam rysowac ";
    char znak;
    cin >> znak;           // znak, na którego podstawie tworzymy szlaczek
    cout << "Ile mam narysowac znakow: " << znak << "? ";
    int ile;
    cin >> ile;
    for (int p=0; p<ile; p++)
        cout << znak;
    cout << endl;
}

//***** Początek funkcji głównej programu *****/
int main()
{
    cout << "Wywoluje funkcje szlaczek_10_gwiazdek()" << endl;
    szlaczek_10_gwiazdek();
    cout << endl << "Chce miec teraz szlaczek skladajacy sie";
    cout << "z pieciu wykryznikow" << endl;
    cout << "Wywoluje wiec funkcje szlaczek_dowolny(5,'!')" << endl;
    szlaczek_dowolny(5,'!');
    int k;
    char zn;
    cout << endl << "Zapytam, jaki chcesz miec szlaczek" << endl;
    cout << "Ile elementow ma miec szlaczek? (to jest zmienna k) ";
    cin >> k;
    cout << "Jakie to maja byc znaki? (to jest zmienna zn) ";
    cin >> zn;
    cout << endl << "Wywoluje funkcje szlaczek_dowolny(k,zn)" << endl;
    szlaczek_dowolny(k,zn);
    cout << endl << "Wywoluje funkcje szlaczek_inaczej()" << endl;
    szlaczek_inaczej();
    cin.ignore();
    getchar();
    return 0;
}
//***** Koniec funkcji głównej programu *****/

```

3.11

Tak wygląda przykładowy ekran po wykonaniu programu:

```
Wywoluje funkcje szlaczek_10_gwiazdek ()
*****
Chce miec teraz szlaczek skladajacy sie z pieciu wykryznikow
Wywoluje wiec funkcje szlaczek_dowolny (5,'!')
!!!!!

Zapytam, jaki chcesz miec szlaczek
Ile elementow ma miec szlaczek? (to jest zmienna k) 12
Jakie to maja byc znaki? (to jest zmienna zn) @
```

### 3. Funkcje w C++

```
Wywołuje funkcję szlaczek_dowolny (k,zn)
oooooooooooo
```

```
Wywołuje funkcję szlaczek_inaczej ()
Podaj, jaki znak mam rysować #
Ile mam narysować znaków: #? 8
#####
```

Powyższy przykład pokazuje, że dany problem można rozwiązać na kilka różnych sposobów, w zależności od tego, czy przekazujemy funkcji argumenty, czy też nie.

Powróćmy do przykładów matematycznych zastosowań definiowanych funkcji.

Funkcje, którym zostaje przypisana wartość, mogą być użyte w wyrażenях na przykład arytmetycznych lub logicznych podobnie jak inne zmienne. W szczególności wartość funkcji może zostać przypisana zmiennej.

Napiszemy programy, z których pierwszy będzie rozwiązywał równanie liniowe, drugi zaś rozwiąże układ dwóch równań liniowych.



#### Przykład

Napiszmy program rozwiązujący równanie liniowe z jedną niewiadomą.

Przypomnijmy, że równaniem liniowym z jedną niewiadomą nazywamy równanie postaci  $A * x = B$ , gdzieś,  $B$  są współczynnikami równania należącymi do zbioru liczb rzeczywistych. Równanie to ma jedno rozwiązanie, gdy  $A$  jest liczbą różną od zera, nieskończoność wiele rozwiązań, gdy oba współczynniki są zerami, bądź nie ma rozwiązań, jeśli jest zerem, a współczynnik  $B$  jest różny od zera. Skoro liczba rozwiązań zależy od wartości współczynników, to funkcji, którą zdefiniujemy, przekażemy oba współczynniki równania. Funkcja nie przyjmie żadnej wartości, wypisze informację o rozwiązaniach równania.

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

void rozwiazanie(float a, float b)
{
    if (a!=0)
        cout << "Równanie ma dokładne jedno rozwiązanie, rowne: " << b/a;
    else
        if (b!=0)
            cout << "Równanie nie ma rozwiązań";
        else
            cout << "Równanie ma nieskończenie wiele rozwiązań";
}
int main()
{
    float A, B;
```

```

cout << "Podaj wspolczynniki rownania: " << endl;
cin >> A >> B;
roziazanie(A,B);
cin.ignore();
getchar();
return 0;

```

3.12

Zanim napiszemy program, który określi liczbę rozwiązań układu dwóch równań liniowych za pomocą metody wyznaczników, przypomnijmy tę metodę. Dany jest układ równań z dwoma niewiadomymi  $x$  i  $y$ .

$$\begin{cases} a_1 \cdot x + b_1 \cdot y = c_1 \\ a_2 \cdot x + b_2 \cdot y = c_2 \end{cases}$$

Wyznacznikiem głównym układu jest liczba  $W$ , której wartość obliczamy ze wzoru:

$$W = a_1 \cdot b_2 - a_2 \cdot b_1$$

Wyznaczniki dla niewiadomych  $x$  i  $y$  obliczamy ze wzorów:

$$W_x = b_2 \cdot c_1 - c_2 \cdot b_1$$

$$W_y = a_1 \cdot c_2 - a_2 \cdot c_1$$

Twierdzenie mówi, że jeśli wyznacznik główny  $W$  jest różny od zera, to układ ma dokładnie jedno rozwiązanie dane wzorami:  $x = \frac{W_x}{W}$ ,  $y = \frac{W_y}{W}$ .

Jeśli wszystkie trzy wyznaczniki są równe zeru, to układ ma nieskończenie wiele rozwiązań. Jeśli zaś wyznacznik główny jest równy zeru, a któryś z wyznaczników dla zmiennych jest różny od zera, to układ nie ma rozwiązań.

### Przykład

Napiszemy program rozwiązujący układ dwóch równań liniowych z dwiema niewiadomymi.

**m**

W programie tym zdefiniujemy trzy funkcje: jedną, której przekażemy pobrane współczynniki układu, funkcja zaś wypisze układ na ekranie; drugą, której przekażemy cztery współczynniki, a funkcja obliczy ich wyznacznik, oraz trzecią, której zadaniem będzie określić liczbę rozwiązań, w wypadku zaś, gdy rozwiązanie jest jedno, dodatkowo zostanie ono obliczone i wyprowadzone na ekran. Tej funkcji przekażemy trzy argumenty - policzone wyznaczniki układu. Przeanalizujmy gotowy program:

### 3. Funkcje w C++

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

void wyswietl_uklad(float a, float b, float c, float d, float e, float f)
{
    cout << "Rozwiązujemy układ:" << endl;
    cout << a << " * x + " << b << " * y = " << c << endl;
    cout << d << " * x + " << e << " * y = " << f << endl;
}
float wyzn(float a, float b, float c, float d)
{
    return a*d-b*c;
}

void rozwiazanie(float g, float x, float y)
{
    if (g!=0)
    {
        cout << "Układ ma dokładnie jedno rozwiązanie ";
        cout << " x = " << x/g << " , y = " << y/g;
    }
    else
        if ((x!=0) || (y!=0))
            cout << "Układ nie ma rozwiązań";
        else
            cout << "Układ ma nieskończenie wiele rozwiązań";
}
int main()
{
    float a1, a2, b1, b2, c1, c2;
    cout << "Podaj współczynniki do pierwszego równania:" << endl;
    cin >> a1 >> b1 >> c1;
    cout << "Podaj współczynniki do drugiego równania:" << endl;
    cin >> a2 >> b2 >> c2;
    wyswietl_uklad(a1,b1,c1,a2,b2,c2);
    rozwiazanie(wyzn(a1,b1,a2,b2), wyzn(c1,b1,c2,b2), wyzn(a1,c1,a2,c2)); // (1)
    cin.ignore();
    getchar();
    return 0;
}
```

3.13

W funkcji głównej pobrane zostały współczynniki układu, dalsza część to wywołania zdefiniowanych funkcji. W linii z komentarzem (1) wywoływana została funkcja `rozwiazanie`, a jako argumenty przekazaliśmy jej wyniki funkcji `wyzn`, czyli wszystkie trzy wyznaczniki obliczone dla układu. Skoro powiedzieliśmy już wcześniej, że wyniki funkcji traktujemy jak każdą zmienną, to mogą być one również argumentami w wywołaniu innej funkcji.

W następnym rozdziale omówimy dokładnie problem znajdowania największej (najmniejszej) wartości w ciągu liczb, tu zajmiemy się prostym, szczególnym przypadkiem, gdy zbiór liczb jest trzyelementowy.

## Przykład

Napiszemy program znajdujący największą liczbę z trzech wczytanych liczb.

Program ten ma wykonać dwa zadania: pobrać od użytkownika trzy liczby, a następnie znaleźć i wyświetlić największą z nich. Zadanie znajdowania elementu największego wśród trzech liczb powierzymy funkcji, którą zdefiniujemy. Funkcję tę wykorzystamy w programie. Kod programu będzie dzięki temu łatwy w analizie i czytelny, a ponadto, jeśli kiedykolwiek później pisząc jakiś program, będziemy musieli wyznaczyć największą z trzech liczb, skopiujemy już zdefiniowaną funkcję do nowo tworzonego kodu programu.

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

float maxi(float a, float b, float c)
{
    float max = a;
    if (b>max)
        max = b;
    if (c>max)
        max = c;
    return max;
}

int main()
{
    int x, y, z;
    cout << "Podaj trzy liczby, a znajde największa ";
    cin >> x >> y >> z;
    cout << "Największa z nich to: " << maxi(x,y,z);
    cin.ignore();
    getchar();
    return 0;
}
```

Teraz zwróciły uwagę na pewien bardzo ważny fakt związany z obsługą przez funkcję argumentów, które są jej przekazywane. Zaczniemy od przykładu. Przekażemy funkcji zmienną, której wartość zostanie zmieniona wewnętrz funkcji. Następnie wyświetlimy wartość tej zmiennej wewnętrz funkcji i po wyjściu z niej.

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

### 3. Funkcje w C++

```
void zmien(int a)
{
    a = a+3;
    cout << "Wewnatrz funkcji zmienna ma wartosc: " << a << endl;
}

int main()
{
    int a = 3;
    cout << "Przed wywolaniem funkcji zmienna ma wartosc: " << a << endl;
    zmien(a); // wywolanie funkcji zmien z parametrem a
    cout << "Po wyjsciu z funkcji zmienna ma wartosc: " << a;
    getchar();
    return 0;
}
```

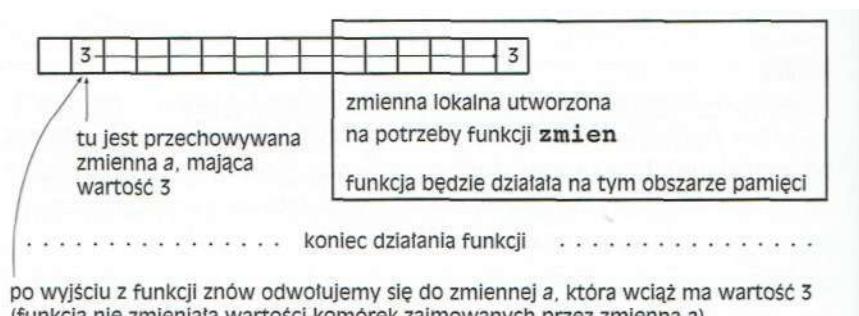
3.15

Po analizie kodu tego krótkiego programu można by przypuszczać, że skoro liczba 3 (wartość zmiennej *a*) została przekazana funkcji, a jedyną instrukcją, którą miała funkcja wykonać, jest zwiększenie wartości zmiennej A o 3, to po wywoaniu funkcji otrzymamy informację o nowej wartości zmiennej wynoszącej 6. Tymczasem działanie programu wygląda następująco:

```
Przed wywoaniem funkcji zmienna ma wartość: 3
Wewnatrz funkcji zmienna ma wartość: 6
Po wyjściu z funkcji zmienna ma wartość: 3
```

Przekazywanie argumentów funkcji przez wartość

Dlaczego więc, pomimo że działanie funkcji polegało na zmianie wartości argumentu funkcji, zmieniła *a* wciąż zachowuje swoją poprzednią wartość? Zgodnie z regułami języka C++, w momencie wywołania funkcji jest tworzona na jej potrzeby zmieniona lokalna o podanej nazwie i do niej jest kopowana wartość przekazana funkcji. Po zakończeniu działania funkcji zmienne powiązane z argumentami przekazanymi do funkcji przestają istnieć. Po wyjściu z funkcji znów odwołujemy się do oryginalnej zmiennej, która nie została zmodyfikowana. Graficznie sposób działania funkcji z argumentami przekazywanymi przez nazwę zmiennej przedstawia rycina 3.2.



Ryc. 3.2. Schematyczny rysunek komórek pamięci w komputerze. Graficzna ilustracja odniesienia się funkcji do zmiennej przekazanej jej przez wartość

### 3.3. Sposoby przekazywania parametrów funkcji

Co należy zrobić, jeśli chcemy, aby zmiany wykonywane w funkcji na zmiennych zostały zachowane? Wówczas możemy podać funkcji informację, aby działała bezpośrednio na zmiennej. Za pomocą znaku & umieszczonego przed nazwą zmiennej odwołujemy się bezpośrednio do adresu w pamięci, pod którym zmienna jest przechowywana. Wszystkie operacje będą więc dokonywane na oryginalnej zmiennej, a nie na zmiennej, która została powołana tylko na czas działania funkcji. Oto przykład funkcji zachowującej zmiany:

- Przekazywanie argumentów do funkcji przez referencję

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

void zmien(int &a) // tu jest zmiana!
{
    a = a+3;
    cout << "Wewnatrz funkcji zmienna ma wartosc: " << a << endl;
}

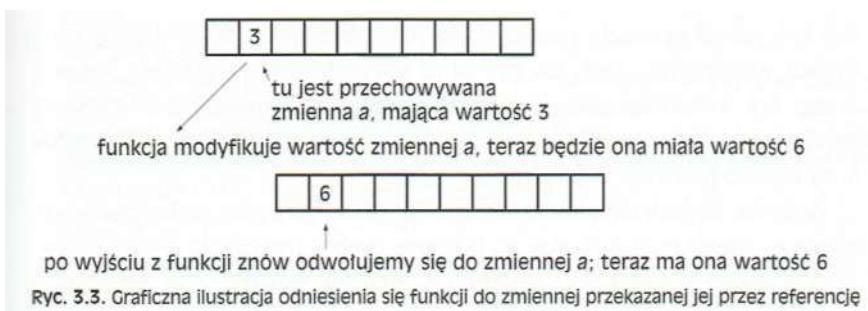
int main()
{
    int a = 3;
    cout << "Przed wywolaniem funkcji zmienna ma wartosc: " << a << endl;
    zmien(a);
    cout << "Po wyjsciu z funkcji zmienna ma wartosc: " << a;
    getchar();
    return 0;
}
```

3.16

Teraz po wyjściu z funkcji zmienna  $a$  ma wartość 6 - funkcja zmodyfikowała wartość przechowywaną pod adresem, który został jej przekazany. A to jest przecież miejsce, gdzie zapisana jest zmienna  $a$ .

Przed wywołaniem funkcji zmienna ma wartość: 3  
Wewnątrz funkcji zmienna ma wartość: 6  
Po wyjściu z funkcji zmienna ma wartość: 6

Ten sposób przekazywania funkcji argumentów nazywamy **przekazaniem przez referencję**. Na rycinie 3.3 przedstawiono sposób działania funkcji, do której przekazujemy argumenty przez referencję:



Ryc. 3.3. Graficzna ilustracja odniesienia się funkcji do zmiennej przekazanej jej przez referencję

Przyjrzymy się teraz przykładowi, w którym wartość zmiennej przekazywana do funkcji powinna zostać zmieniona:



### Przykład

Pracownik firmy otrzymuje dziesięcioprocentową podwyżkę po każdym przepracowanym roku. Równocześnie zmienia się jego staż pracy liczony w latach. Napiszmy funkcję, która zmieni zarówno wartość wypłaty pracownika, jak i zaktualizuje jego staż pracy po przepracowanym roku.

Ponieważ chcemy zmodyfikować wartości dwóch zmiennych, napiszemy funkcję, której argumenty przekażemy przez referencję.

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

void zmiany(float &zarobek, int &staz)
{
    zarobek = zarobek*1.1;
    staz = staz+1;
}

int main()
{
    float zarobek = 1000;
    int staz = 0;
    cout << "Twój staż pracy: " << staz << " Zarabiasz: " << zarobek << endl;
    zmiany(zarobek,staz);
    cout << "Twój staż pracy: " << staz << " Zarabiasz: " << zarobek << endl;
    getchar();
    return 0;
}
```

3.17

Twój staż pracy: 0 Zarabiasz: 1000  
 Twój staż pracy: 1 Zarabiasz: 1100

Każde wywołanie funkcji zmiany sprawi, że ulegnie zmianie zarówno wartość zmiennej zarobek, jak i zmiennej staż.

Zamiana wartości pomiędzy dwoma elementami

W wielu algorytmach porządkujących ciąg elementów według zadanej kolejności operacją podstawową jest przedstawienie pomiędzy sobą dwóch elementów. Jeśli na przykład chcielibyśmy tą metodą ustawić liczby 7, 6, 1 w kolejności od najmniejszej do największej, musielibyśmy zamienić miejscami elementy pierwszy z drugim, drugi z trzecim, a na koniec pierwszy z drugim.

Zauważ, że jeśli chcemy zamienić wartości pomiędzy zmiennymi  $a$ ,  $b$ , to nie możemy tego dokonać za pomocą dwóch instrukcji:  $a = b, b = a$ . Jeśli bowiem początkowo zmienna  $a$  miałaby wartość 7, a zmienna  $b$

wartość 6, to po pierwszym przypisaniu nadalibyśmy zmiennej *a* wartość 6 i taką miałyby ona w momencie przejścia do instrukcji drugiej. Teraz zmiennej *b* mającej wartość 6 przypisalibyśmy wartość zmiennej *a*, która w tej chwili też już ma wartość 6. Co więc należy zrobić, aby dokonać zamiany pomiędzy wartościami dwóch zmiennych? Trzeba wprowadzić zmienną pomocniczą, która przechowa wartość początkową zmiennej *a*. Sama zaś zamiana wartości pomiędzy dwoma zmiennymi składać się będzie z trzech instrukcji:

```
temp = a;
a = b;
b = temp;
```

### Przykład

Napiszmy program porządkujący trzy liczby w sposób niemalejący.

Aby trzy liczby ustawić obok siebie w zadanej kolejności, musimy zamieniać pomiędzy sobą te pary liczb, w których liczba mniejsza występuje za liczbą większą. Napiszemy więc funkcję z dwoma argumentami, których wartości chcemy pomiędzy sobą zamienić. Przekażemy je do funkcji przez referencję.

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

void zamien_miejscami(int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}

int main()
{
    int a, b, c;
    cout << "Podaj wartości a, b, c ";
    cin >> a >> b >> c;
    if (a>b) zamien_miejscami(a,b);    // tu dokonują się zamiany elementów
    if (a>c) zamien_miejscami(a,c);    // pomiędzy sobą, tak aby ostatecznie
    if (b>c) zamien_miejscami(b,c);    // zmienne a, b, c spełniały warunek a<=b<=c
    cout << "Wartości liczb a, b, c po zamianach: " << a << " " << b << " " << c;
    cin.ignore();
    getchar();
    return 0;
}
```

3.18

### 3. Funkcje w C++

Ten fragment kodu funkcji głównej, w którym sprawdzane jest ustalenie elementów i ewentualne zamiany, można wyodrębnić i również umieścić w kolejnej zdefiniowanej funkcji. Ponieważ wykorzystamy w niej funkcję zamien\_miejscami, jej definicja musi znajdować się powyżej obecnie definiowanej funkcji, którą nazwiemy ustaw\_rosnaco:

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

void zamien_miejscami(int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}

void ustaw_rosnaco(int &a, int &b, int &c)
{
    if (a>b) zamien_miejscami(a,b);
    if (a>c) zamien_miejscami(a,c);
    if (b>c) zamien_miejscami(b,c);
}

int main()
{
    int a, b, c;
    cout << "Podaj wartosci a, b, c ";
    cin >> a >> b >> c;
    ustaw_rosnaco (a,b,c);
    cout << "Wartosci liczb a, b, c po zamianach: " << a << " " << b << " " << c;
    cin.ignore();
    getchar();
    return 0;
}
```

3.19

Niebezpieczeństwo przekazywania argumentów do funkcji przez referencję

Musisz jednak wiedzieć, że przekazywanie argumentów do funkcji przez referencję jest niebezpieczne, gdyż nieopatrznie możemy zmienić wartości tych zmiennych, których wcale zmieniać nie chcemy. Zauważ, że z wywołania jakiejkolwiek funkcji w funkcji głównej nie widać, który argument został jej przekazany przez wartość, a który przez referencję. Pomyłka w przekazaniu argumentu do funkcji może się stać powodem błędu, który trudno zlokalizować przy analizie kodu programu. Staraj się więc raczej unikać przekazywania argumentów do funkcji przez referencję.

## Pytania kontrolne

1. Podaj składnię funkcji zdefiniowanej w C++.
  2. Czym się różnią parametry formalne funkcji od parametrów aktualnych?
  3. Czy funkcja może przyjmować wartość? Jeśli tak, to ile wartości może przyjąć?
  4. Czym się różni przekazywanie funkcji argumentów przez wartość zmiennej od przekazywania argumentów przez referencje?
  5. Czy może istnieć poprawna funkcja, która nie pobiera żadnych argumentów i nie przyjmuje żadnej wartości? Jeśli tak, podaj przykład.
  6. Podaj przykład funkcji, która pobiera dwa argumenty typu float, a przyjmuje wartość typu integer.
  7. Czy może istnieć funkcja, która nie pobiera żadnych argumentów, a przyjmuje wartość? Jeśli tak, to podaj przykład.

## Ćwiczenia

1. Określ, czy poprawne są definicje funkcji:

```
a)  
void usmieszek()  
{  
    cout << ":-)";  
}
```

```
b)  
int dodaj (int a, int b)  
{  
    return a/b;  
}
```

```
c)
float szlaczek (float &w)
{
    for (int i=1; i<w; i++)
        cout << "*";
}
```

Jeśli zauważysz błąd w definicji którejś z funkcji, popraw go.

4. Napisz program, który dla pobranych trzech liczb będących długościami odcinków bada, czy da się z nich zbudować trójkąt. Jeśli tak, to określa powstały trójkąt jako rozwartokątny, ostrokątny, równoramienny lub prostokątny. W programie wykorzystaj zdefiniowane przez siebie funkcje.
5. Napisz program, który wyznacza objętość sześciangu o długości boku podanej przez użytkownika (w wypadku podania niedodatniej wartości boku pojawia się komunikat o błędzie i prośba o ponowne podanie długości boku). W programie wykorzystaj zdefiniowaną przez siebie funkcję, która pobiera długość krawędzi i oblicza objętość sześciangu.
6. Napisz funkcję, która pobiera dwa argumenty typu całkowitego  $a, b$  takie, że  $a < b$ , oraz oblicza wartość sumy wszystkich liczb całkowitych należących do przedziału obustronnie domkniętego  $\{a, b\}$ .
7. Napisz program pobierający z klawiatury dwie liczby całkowite i pytający użytkownika o ich iloczyn. W wypadku podania złego wyniku użytkownik otrzymuje komunikat o błędzie i jest proszony o ponowne podanie wyniku dotąd, aż poda poprawny. W programie wykorzystaj zdefiniowaną przez siebie funkcję, która pobiera dwie liczby i przyjmuje wartość ich iloczynu.

## 4. Implementacja klasycznych algorytmów iteracyjnych

Na początku tego rozdziału pokażemy, jak iteracyjnie wyznaczyć najmniejszą wartość w ciągu liczb pobieranych z klawiatury. Następnie omówimy kilka klasycznych algorytmów badających własności liczb. Nauczysz się badać, czy liczba jest pierwsza, i poznasz algorytm Euklidesa, za którego pomocą można wyznaczyć NWD i NWW dwóch liczb. Dowiesz się, jak napisać funkcję obliczającą przybliżoną wartość pierwiastka kwadratowego z liczby nieujemnej, nie korzystając z pliku bibliotecznego, w którym zdefiniowane są funkcje matematyczne. Zapoznasz się też z metodą obliczania pola obszaru o nieregularnym kształcie. Pokażemy również jedną z metod znajdowania przybliżonej wartości miejsca zerowego funkcji. Na koniec powiemy ci, czym są metody Monte Carlo.

### 4.1. Znajdowanie najmniejszego lub największego elementu w ciągu liczb

Wyznaczenie najmniejszego lub największego elementu zbioru znajduje zastosowanie w programach, które służą do wskazania na przykład wieku najstarszego człowieka, najniższej temperatury albo długości skoku najlepszego skoczka. Ten problem omówiliśmy w poprzednim rozdziale dla szczególnego przypadku - zbioru trzyelementowego.

#### Przykład

Napiszmy program, który z  $k$  wczytanych liczb wyznacza liczbę najmniejszą. Program powinien wczytać najpierw liczbę  $k$  oznaczającą, ile liczb znajduje się w badanym ciągu. Następnie program powinien wczytać  $k$  liczb i wyznaczyć najmniejszą z nich.

Najmniejsza wartość będzie zapamiętywana w zmiennej  $min$ . Po wprowadzeniu pierwszej liczby z badanego ciągu przyjmujemy ją za tymczasowe minimum, nie sprawdzając żadnych dodatkowych warunków. Po wprowadzeniu każdej następnej liczby sprawdzamy, czy jest ona mniejsza od dotychczasowego minimum. Jeśli tak, to jej wartość przyjmujemy od tą za nowe minimum. Po wprowadzeniu i sprawdzeniu ostatniej liczby pozostaje już tylko wypisanie wartości najmniejszego elementu i zakończenie programu. Oczywiście w programie należy zastosować licznik,



#### 4. Implementacja klasycznych algorytmów iteracyjnych

który będzie kontrolował ilość wprowadzonych liczb. Na początku programu licznik przyjmie wartość równą ilości badanych liczb, a po każdej wprowadzonej liczbie jego wartość zmniejszymy o jeden.

##### Specyfikacja problemu algorytmicznego

**Problem algorytmiczny:** Znalezienie minimum w ciągu liczb

**Dane wejściowe:**  $k \in N_+$  – ilość liczb

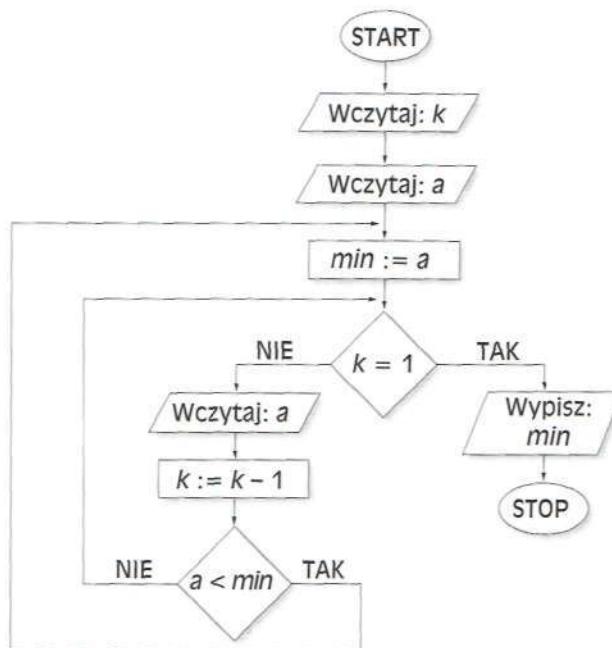
$a_1, a_2, \dots, a_k; a_{1\dots k} \in R$  – ciąg liczb

**Dane wyjściowe:**  $\min \in R$

Algorytm w postaci listy kroków zapiszemy następująco:

1. Wczytaj  $k$ .
2. Wczytaj  $a$ .
3. Zmiennej  $\min$  przypisz wartość  $a$ .
4. Jeśli  $k$  jest równe 1, wypisz  $\min$  i zakończ.
5. Wczytaj  $a$ .
6. Zmniejsz  $k$  o 1.
7. Jeśli  $a < \min$ , przejdź do kroku 3.
8. Przejdz do kroku 4.

Gotowy schemat blokowy prezentuje rycina:



Ryc. 4.1. Schemat blokowy algorytmu wyszukiwania wartości najmniejszej w ciągu  $k$  liczb podanych na wejściu

Pozostało nam już tylko napisać program, który wykorzystuje ten algorytm:

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

int main()
{
    int k, a, min; // deklaracja zmiennych
    cout << "ile chcesz podać liczby - co najmniej jedna ";
    cin >> k; // ilość liczb do wprowadzenia
    cout << "wprowadź liczbę ";
    cin >> a; // wprowadzamy pierwszą liczbę
    min = a; // i przyjmujemy ją jako bieżące minimum
    while (k>1) // pętla będzie się wykonywała, dopóki k>1
    {
        cout << "wprowadź liczbę ";
        cin >> a; // wprowadzamy następną liczbę
        k--;
        if (a<min) // najważniejsza część programu
            min = a;
    } // koniec pętli
    cout << "najmniejsza wartość w podanym ciągu to: " << min;
    cin.ignore();
    getchar();
    return 0;
}
```

4.1

W analogiczny sposób możemy znaleźć największy element w ciągu oprowadzanych liczb. Sprawdzalibyśmy tylko, czy kolejno wprowadzona liczba jest większa od aktualnie ustalonego maksimum.

## 4.2. Liczba pierwsza czy złożona - algorytm testujący

Zacznijmy od przypomnienia definicji liczby pierwszej: **liczba pierwsza to taka liczba naturalna, która ma tylko dwa różne dzielniki: liczbę jeden i samą siebie.**

Liczby pierwsze stanowią zbiór nieskończony. Są nimi 13 i 7, a nie jest liczbą pierwszą na przykład 14, gdyż dzieli się przez 1, samą siebie, ale też przez 2 i 7. Liczba, która nie jest liczbą pierwszą, nazywa się **liczbą złożoną**.

**Liczby 1 i 0 nie są ani liczbami pierwszymi, ani złożonymi.**

Pojęcie liczby pierwszej i złożonej

### Czy wiesz, że...

Największą znaną obecnie liczbę pierwszą odnalazł 18 lutego 2005 roku uczestnik projektu **GIMPS**, czyli Wielkiego Internetowego Poszukiwania Liczb Pierwszych Mersenne'a (*Great Internet Mersenne Prime Search*). Jest ona równa  $2^{25964931} - 1$  (liczy prawie 8 milionów cyfr). Tak ogromne liczby pierwsze nie mają obecnie praktycznego zastosowania. Więcej na temat projektu GIMPS dowiesz się ze strony [www.mersenne.org](http://www.mersenne.org).

Duże liczby pierwsze służą do konstruowania trudnych do odczytania szyfrów. Są także bardzo użyteczne przy tworzeniu kodów korekcyjnych do wyszukiwania błędów w przekazie danych, na przykład z satelity.

### Przykład

Napiszmy program, który udzieli odpowiedzi na pytanie, czy podana na wejściu liczba  $n$  jest liczbą pierwszą.

Aby to zbadać, będziemy liczbę  $n$  dzielić przez kolejne liczby naturalne mniejsze od  $n$ , począwszy od liczby 2, i sprawdzać **resztę z dzielenia**. Jeśli znajdziemy choćby jeden dzielnik, dla którego reszta z dzielenia wynosi 0, to liczba  $n$  jest liczbą złożoną (nie musimy sprawdzać wszystkich możliwych dzielników mniejszych od liczby  $n$ ).

Jeśli liczba nie dzieli się przez żadną liczbę całkowitą równą jej pierwiastkowi kwadratowemu lub od niego mniejszą, to jest ona liczbą pierwszą.

Dowód tego twierdzenia jest bardzo prosty: jeżeli liczba  $n$  jest liczbą złożoną, to co najmniej jeden dzielnik jest nie większy od  $\sqrt{n}$ , ponieważ jeżeli jest większy, wówczas  $n$  byłoby równe iloczynowi  $ab$ , gdzie  $a > \sqrt{n}$  i  $b > \sqrt{n}$ , zatem  $a \cdot b$  byłoby większe od  $\sqrt{n} \cdot \sqrt{n}$ , czyli większe od  $n$ , co jest sprzeczne z założeniem.

W schematach blokowych do obliczenia reszty z dzielenia używamy operatora mod, na przykład  $7 \bmod 2 = 1$ . W języku programowania C++ operator reszty z dzielenia przyjmuje postać `%`, na przykład  $7 \% 2 = 1$ .

### Specyfikacja problemu algorytmicznego i opis użytych zmiennych

**Problem algorytmiczny:** Badanie, czy podana liczba jest pierwsza

**Dane wejściowe:**  $n \in N; n > 1$  – badana liczba

**Dane wyjściowe:** Napis „liczba pierwsza”, gdy  $n$  jest liczbą pierwszą, lub napis „liczba złożona”, gdy  $n$  nie jest liczbą pierwszą

**Zmienne pomocnicze:**  $i \in N$  – potencjalny dzielnik;  $i > 1$

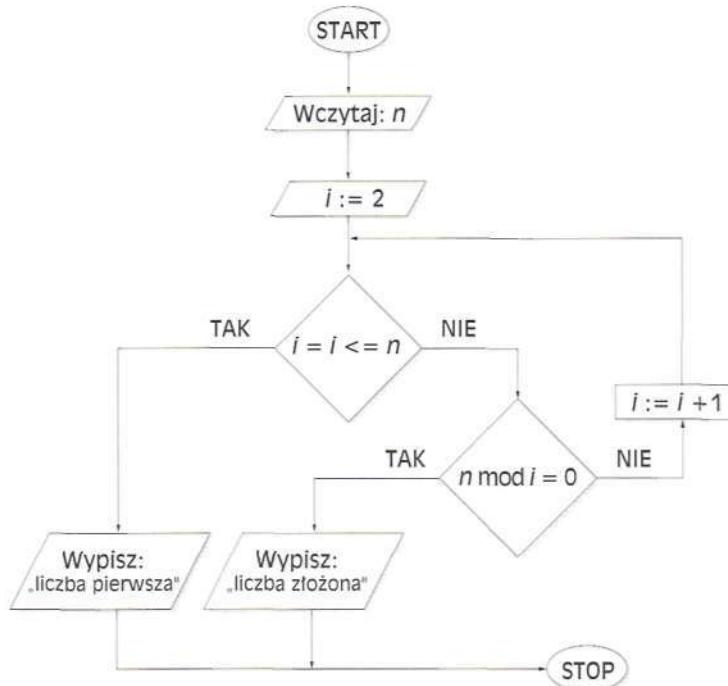
#### 4.2. Liczba pierwsza czy złożona - algorytm testujący

Ogólnie algorytm możemy zapisać za pomocą listy kroków:

- I. Wczytaj  $n$ .
- Zmiennej pomocniczej  $i$  przypisz wartość 2.
- Jeśli kwadrat liczby  $i$  jest większy od liczby  $n$ , to wypisz „liczba pierwsza” i zakończ.
- Jeśli reszta z dzielenia  $n$  przez  $i$  wynosi 0, wypisz „liczba złożona” i zakończ.
- Zwiększ o 1 wartość zmiennej pomocniczej  $i$ .
- Przejdź do kroku 3.

Zauważ, że zamiast badać warunek, czy wartość zmiennej  $i$  jest nie większa od liczby  $n$ , badamy, czy kwadrat zmiennej  $i$  jest nie większy od  $n$ . W ten sposób unikniemy błędów zaokrągleń, które pojawiłyby się przy zastosowaniu funkcji  $\text{sqrt}(n)$ , w wyniku będącym pierwiastkiem kwadratowym liczby  $n$ . Definicja tej funkcji znajduje się w bibliotece `cmath`.

Zapiszmy ten algorytm jeszcze w schemacie blokowym:



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

Oto program wykorzystujący omówiony algorytm:

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

int main()
{
    int i = 2, n;
    cout << "podaj liczbę całkowitą większą od 1 ";
    cin >> n;
    while (n%i!=0 && i*i<=n) // pętla skończy się, jeśli jeden
        i++; // z tych warunków nie będzie spełniony
    if (i*i<=n)
        cout << "liczba złożona";
    else
        cout << "liczba pierwsza";
    cin.ignore();
    getchar();
    return 0;
}
```

4.2

Wyjaśnienia wymaga warunek zastosowany w pętli while, który oznacza: wykonuj pętlę, dopóki reszta z dzielenia  $n$  przez  $i$  jest różna od 0 oraz kwadrat liczby  $i$  jest mniejszy lub równy liczbie  $n$ . Wyjście z pętli nastąpi, gdy nie będzie spełniony choćby jeden z warunków: albo reszta z dzielenia przez zmienną pomocniczą będzie równa 0, albo zmienna pomocnicza podniesiona do kwadratu osiągnie wartość większą od badanej liczby.

Jeśli wyjście z pętli nastąpi wskutek otrzymania zerowej reszty z dzielenia, oznacza to, że na wejściu podaliśmy liczbę złożoną, ponieważ właśnie znaleźliśmy dzielnik tej liczby różny zarówno od 1, jak i od niej samej. W przeciwnym wypadku mamy do czynienia z liczbą pierwszą.

**złożoność:** Podstawowe operacje wykonywane w tak skonstruowanym algorytmie to dzielenie i porównanie. Gdy mamy do czynienia z liczbą pierwszą, czas czynności dzielenia jest tyle, ile wynosi wartość całkowita pierwiastka badanej liczby. Gdy na wejściu podana zostanie liczba złożona, ilość operacji dzielenia będzie jeszcze mniejsza. Jest to więc algorytm o klasie złożoności  $O(\sqrt{n})$ .

### 4.3. Wyznaczanie MMD i NWW dla dwóch liczb naturalnych (algorytm Euklidesa)

Jednym z klasycznych algorytmów jest algorytm znajdowania największego wspólnego dzielnika dla pary liczb naturalnych. Największy wspólny dzielnik dwóch liczb całkowitych  $a$  i  $b$  oznaczamy symbolem

#### 4.3. Wyznaczanie NWD i NMW dla dwóch liczb naturalnych (algorytm Euklidesa)

MVD( $a, b$ ). Umiejętność znajdowania największego wspólnego dzielnika przydaje się zwłaszcza w sytuacji, gdy chcemy skrócić ułamek, co znaczą podzielenie licznika i mianownika przez ich największy wspólny dzielnik. Otrzymujemy wówczas ułamek już dalej nieskracalny, gdyż liczniku i mianowniku znajdują się tak zwane liczby względnie pierwsze, czyli takie, których największy wspólny dzielnik wynosi 1.

Największym wspólnym dzielnikiem dwóch liczb naturalnych jest największa z liczb całkowitych, przez którą obie liczby dzielą się bez reszty.

Jak znaleźć największy wspólny dzielnik dwóch liczb za pomocą prostego algorytmu? Można by na przykład sprawdzać, począwszy od wartości mniejszej z dwóch badanych liczb, czy obie dzielą się przez nią bez reszty. Następnie, jeśli tak nie jest, dzielnik zmniejszyć o 1 i znów badać, czy obie liczby dzielą się przez nowy dzielnik. Postępujemy w ten sposób tak długo, aż znajdziemy wartość, która dzieli obie liczby bez reszty. Będzie to ich największy wspólny dzielnik. Opisaną metodę można zakwalifikować do metod zwanych metodami brutalnymi (ang. brute force), ponieważ jest bardzo prosta, ale wymaga wykonania wielu operacji. Dlatego warto poznać algorytm szybszy, a jednocześnie równie prosty w implementacji. Nosi on nazwę pochodzącą od nazwiska jego twórcy: **algorytm Euklidesa** (o algorytmie tym wspomnieliśmy w rozdziale pierwszym). Zaczniemy od słownego opisu algorytmu.

Pobieramy dwie liczby, dla których chcemy wyznaczyć największy wspólny dzielnik. Od większej z nich odejmujemy mniejszą, a następnie większą różnicę zastępujemy otrzymaną różnicą. Postępujemy tak dotąd, aż liczby będą równe. Otrzymana liczba jest największym wspólnym dzielnikiem.

Zanim zapiszemy sformalizowany algorytm, przeanalizujmy przykład liczbowy:

Niech liczba  $a$  ma wartość 12, a liczba  $b$  wartość 20. Ponieważ  $b > a$ , więc liczbę  $b$  zastępujemy różnicą  $b - a$ , czyli nowe  $b$  przyjmie wartość 8. Mamy zatem dane  $a = 12$  oraz  $b = 8$ . Teraz  $a > b$ , więc tym razem zastępujemy  $a$  różnicą  $a - b$ , czyli liczbą 4. Ponieważ  $a$  ma wartość 4, a  $b$  jest równe 8, więc w miejscu  $b$  wstawimy  $b - a$ , czyli 4. Otrzymaliśmy warunek zakończenia pętli, gdyż obie liczby są już sobie równe. A zatem szukany NWD to liczba 4.

##### Przykład

Napiszmy program, który dla dwóch liczb  $a, b$  podanych na wejściu oblicza ich NWD (klasyczna metoda Euklidesa).

Przejdźmy do formalnego opisu algorytmu.

**nu**

#### 4. Implementacja klasycznych algorytmów iteracyjnych

##### Specyfikacja problemu algorytmicznego

<b>Problem algorytmiczny:</b>	Wyznaczenie największego wspólnego dzielnika dwóch liczb
<b>Dane wejściowe:</b>	$a, b \in N_+$
<b>Dane wyjściowe:</b>	$\text{NWD}(a, b)$

Kolejno postępujemy według listy kroków:

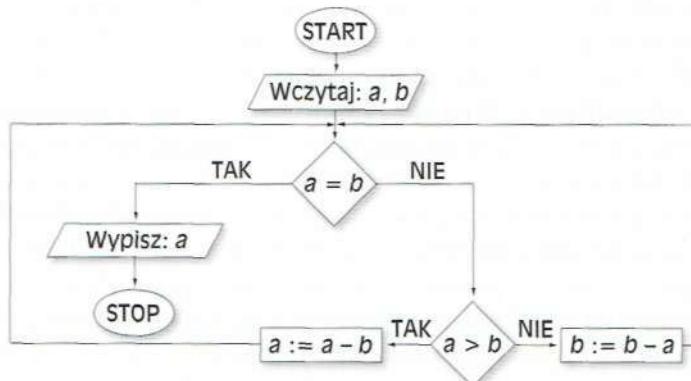
1. Wczytaj  $a, b$ .
2. Jeśli  $a = b$ , wypisz  $a$  i zakończ.
3. Jeśli  $a > b$ , zmiennej  $a$  przypisz  $a - b$  i wróć do kroku 2.
4. Jeśli  $a < b$ , zmiennej  $b$  przypisz  $b - a$  i wróć do kroku 2.

Tak skonstruowany algorytm Euklidesa opiera się na własności:

$$\text{NWD}(a, b) = \text{NWD}(a - b, b), \text{ jeśli } a > b$$

$$\text{NWD}(a, b) = \text{NWD}(a, b - a), \text{ jeśli } b > a$$

Przeanalizujmy jeszcze schemat blokowy algorytmu:



Ryc. 4.3. Schemat blokowy algorytmu wyznaczającego NWD dwóch liczb metodą Euklidesa

Kod samego algorytmu jest krótki i przejrzysty, nie powinno zatem być problemu z jego zrozumieniem:

```

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

int NWD(int a, int b) // funkcja licząca NWD(a,b)
{
    while (a!=b)      // dopóki a jest różne od b
    {
        if(a>b)       // jeśli a jest większe od b, to
            a = a-b;   // w miejsce a podstaw różnicę a-b
        else           // w przeciwnym wypadku
            b = b-a;   // w miejsce b podstaw różnicę b-a
    }
    return a;
}
  
```

#### 4.3. Wyznaczanie NWD i NWV dla dwóch liczb naturalnych (algorytm Euklidesa)

```
int main()          // funkcja główna programu
{
    int a, b;
    cout << "podaj pierwsza liczbe a: ";
    cin >> a;
    cout << "podaj druga liczbe b: ";
    cin >> b;
    cout << "NWD(" << a << "," << b << ") ma wartosc: " << NWD(a,b);
    cin.ignore();
    getchar();
    return 0;
}
```

##### Przykład

Napiszmy program, który podobnie jak poprzedni, dla dwóch liczb  $a, b$  oblicza ich NWD (zoptymalizowana metoda Euklidesa).

**m**

Zmodyfikowany algorytm Euklidesa z użyciem funkcji mod

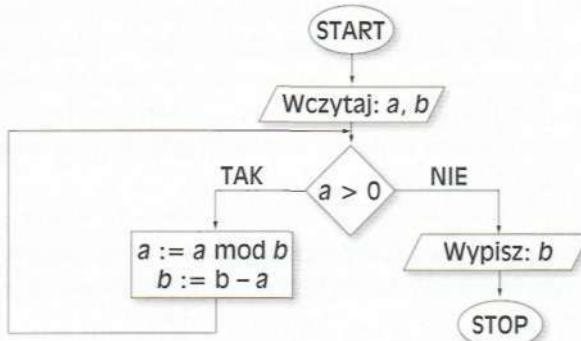
Częściej stosuje się modyfikację algorytmu Euklidesa, wykorzystującą zależność:  $\text{NWD}(a, b) = \text{NWD}(a \bmod b, b - (a \bmod b))$ .

Będziemy iteracyjnie zmieniać wartości liczb  $a$  i  $b$  aż do momentu, gdy  $a$  osiągnie wartość 0. Wówczas największy wspólny dzielnik obu liczb wynosi  $b$ . Jest to oczywiste, gdyż 0 dzieli się przez każdą liczbę całkowitą, a największą z liczb, przez jaką dzieli się liczba różna od 0, jest ona sama. Stąd:  $\text{NWD}(0, b) = b$ .

Zapiszmy algorytm jako listę kroków:

1. Wczytaj  $a, b$ .
2. Jeśli  $a$  nie jest większe od 0, wypisz  $b$  i zakończ.
3. Zmniejsz  $a$  przypisząc resztę z dzielenia  $a$  przez  $b$ .
4. Zmniejsz  $b$  przypisząc różnicę  $b - a$  i wróć do kroku 2.

Spójrzmy na gotowy schemat blokowy:



Ryc. 4.4. Schemat blokowy algorytmu wyznaczającego NWD dwóch liczb zoptymalizowaną metodą Euklidesa

#### 4. Implementacja klasycznych algorytmów iteracyjnych

Funkcja obliczająca NWD według tej metody ma postać:

```
int NWD(int a, int b)
{
    while (a>0)
    {
        a = a%b;
        b = b-a;
    }
    return b;
}
```

4.4

Porównanie  
obu wersji  
algorytmu Euklidesa

Dla porównania obu metod prześledźmy ilość pętli wykonujących się w algorytmach podczas szukania NWD dla dwóch dowolnych liczb.

Pierwsza z metod Euklidesa:

$$\begin{aligned} \text{NWD}(51,60) &= \text{NWD}(51,9) = \text{NWD}(42,9) = \text{NWD}(33,9) = \text{NWD}(24,9) = \\ &= \text{NWD}(15,9) = \text{NWD}(6,9) = \text{NWD}(6,3) = \text{NWD}(3,3) \end{aligned}$$

Druga z metod Euklidesa:

$$\text{NWD}(51,60) = \text{NWD}(51,9) = \text{NWD}(6,3) = \text{NWD}(0,3)$$

Jak widać, dla tak dobranych liczb różnica w ilości wykonanych operacji jest duża. Czy oznacza to, że druga z metod jest szybsza? Nie w każdym przypadku. Weźmy inny zestaw liczb, dla których poszukamy NWD według obu metod.

Pierwsza z metod Euklidesa:

$$\text{NWD}(12, 48) = \text{NWD}(12, 36) = \text{NWD}(12, 24) = \text{NWD}(12, 12)$$

Druga z metod Euklidesa:

$$\begin{aligned} \text{NWD}(12, 48) &= \text{NWD}(12, 36) = \text{NWD}(12, 24) = \text{NWD}(12, 12) = \\ &= \text{NWD}(0, 12) \end{aligned}$$

Druga z metod zdecydowanie zmniejsza ilość obiegów pętli w wypadku, gdy mamy do czynienia z parą liczb względnie pierwszych lub kiedy największym wspólnym dzielnikiem jest liczba stosunkowo mała.

W starszych procesorach ten fakt nie miał większego znaczenia, ponieważ operacje dzielenia (wykorzystywane w drugiej z metod) były znacznie bardziej skomplikowane niż odejmowanie. W najnowszych procesorach to się zmieniło, dlatego częściej stosuje się drugą z metod.

## HtT

### Przykład

Poszukajmy najmniejszej wspólnej wielokrotności dwóch liczb (wykorzystanie algorytmu Euklidesa).

Najmniejszą wspólną wielokrotnością dwóch liczb naturalnych nazywamy najmniejszą z liczb, która jest podzielna przez obie te liczby.

Najmniejszą wspólną wielokrotność liczb  $a$  i  $b$  skrócie oznaczamy jako  $\text{NWW}(a, b)$ . Wyznaczenie jej ma zastosowanie na przykład przy sprowadzaniu ułamków do wspólnego mianownika.

#### 4.4. Obliczanie pierwiastka kwadratowego z liczby dodatniej - metoda Newtona-Raphsona

Nie będziemy konstruować nowego algorytmu do znajdowania  $\text{NWW}(ff. b)$ , ponieważ w istocie opiera się on na algorytmie znajdowania NWD i wzorze:

$$\text{NWW}(a, b) = (a * b) \text{ div } \text{NWD}(a, b)$$

gdzie  $\text{div}$  oznacza dzielenie w zbiorze liczb całkowitych. W języku C++ dzielenie w zbiorze liczb całkowitych jest automatycznie realizowane przez operator dzielenia  $/$ , jeśli dzielna i dzielnik są liczbami całkowitymi, na przykład jeśli  $a = 5$ ,  $b = 2$  i obydwie są zmiennymi typu `int`. to  $a / b = 2$ .

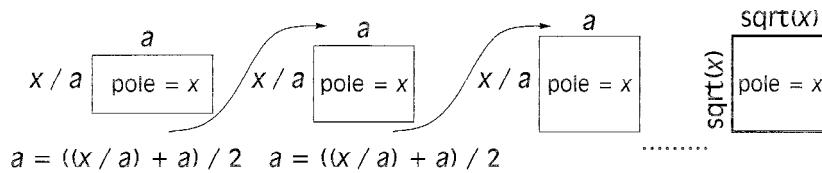
#### 4.4. Obliczanie pierwiastka kwadratowego z liczby dodatniej – metoda Newtona-Raphsona

Do rozwiązywania niektórych problemów algorytmicznych używamy funkcji matematycznych takich jak sinus, potęga liczby czy pierwiastek liczby nieujemnej. Te funkcje zostały już napisane i możecie z nich korzystać, dołączając do swojego kodu bibliotekę `cmath`, gdyż w niej właśnie znajdują się ich definicje. Warto jednak przynajmniej niektóre funkcje umieć napisać samodzielnie. Pokażemy więc, jak zaimplementować algorytm liczący wartość pierwiastka kwadratowego metodą Newtona-Raphsona, znaną również – zwłaszcza matematykom – pod nazwą metody Herona. Metoda polega na znalezieniu kolejnych przybliżeń wartości szukanego pierwiastka.

Pamiętamy, że kwadrat, którego pole wynosi  $*$  jednostek, ma bok o długości  $\sqrt{x}$  jednostek. Nasz problem sprowadza się więc do znalezienia 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  $a * x/a = x$ ).

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).

Opis metody  
Newtona-Raphsona



Ryc. 4.5. Graficzna ilustracja metody Newtona-Raphsona

#### 4. Implementacja klasycznych algorytmów iteracyjnych

Widzimy, że różnica pomiędzy długościami obu boków prostokąta zmalała, pole zaś pozostało niezmienione. Jeżeli boki nadal nie mają tej samej długości, budujemy kolejny prostokąt według tej samej zasady. Postępujemy tak aż do momentu, gdy różnica pomiędzy długościami boków prostokąta będzie mniejsza od zadanej dokładności. Zwróć uwagę, że skoro boki kolejnych prostokątów zbliżają się do siebie długością, to cały prostokąt upodabnia swój kształt do kwadratu. Stąd też długości boków prostokąta zmierzają do długości boku kwadratu, czyli do  $\sqrt{x}$ .

##### Przykład

Obliczmy przybliżoną wartość pierwiastka kwadratowego liczby dodatniej.

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:

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

##### Specyfikacja problemu algorytmicznego

Problem algorytmiczny: Obliczenie przybliżonej wartości pierwiastka kwadratowego liczby dodatniej

Dane wejściowe:  $x \in \mathbb{R}_+$  - liczba, z której pierwiastek obliczamy

$d \in \mathbb{R}_+$  - dokładność wyznaczenia pierwiastka

Dane wyjściowe:  $a \in \mathbb{R}_+$  - przybliżona wartość pierwiastka

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

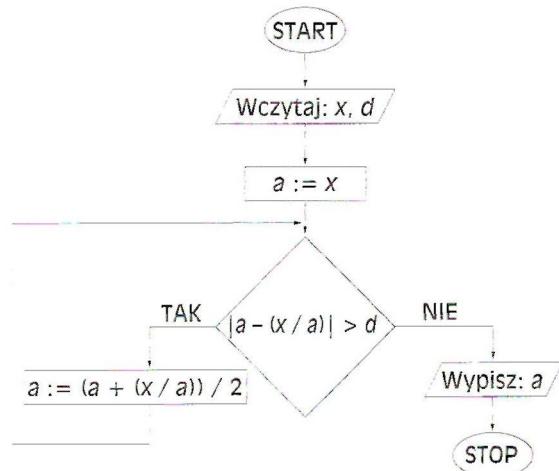
1. Wczytaj  $x$ , wczytaj  $d$ .
2. Zmiennej  $a$  przypisz wartość  $x$ .
3. Jeśli  $|a - (x/a)|$  nie jest większe od wartości zmiennej  $d$ , wypisz wartość zmiennej  $a$  i zakończ,
4. Zmiennej  $a$  przypisz wartość  $(a + (x/a)) / 2$  i wróć do kroku 3.

Zwróć uwagę, że w tym algorytmie jako pierwsze przybliżenie wartości pierwiastka liczby  $x$  przyjęliśmy wartość  $x$ , choć oczywiście równie dobrze moglibyśmy przyjąć każdą inną liczbę dodatnią.

Przeanalizujmy blokowy schemat algorytmu. Przed każdym przebiegiem pętli sprawdzamy, czy różnica boków prostokąta jest już na tyle

#### 4.4. Obliczanie pierwiastka kwadratowego z liczby dodatniej - metoda Newtona-Raphsona

mała, że możemy ten prostokąt uznać za dobre przybliżenie kwadratu, a zatem - podać długość jednego z boków jako wartość pierwiastka z odpowiednio zadaną na wejściu dokładnością (ryc. 4.6).



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

Oto prosty program wykorzystujący zdefiniowaną w nim funkcję do obliczenia pierwiastka z liczby 2 z dokładnością do 0,01:

```

#include <iostream>
#include <cstdio>
#include <cmath>
using namespace std;

double pierwiastek(double x, double d)
{
    double a = x;
    while (fabs(a-(x/a))>d) // dopóki różnica boków jest większa od dokładności
    {
        a = (a+(x/a))/2;      // modyfikujemy długości boków, korzystając ze
                               // średniej arytmetycznej
    }
    return a;                  // funkcji przypisujemy długość boku jako wartość pierwiastka z x
}

int main()
{
    cout << "Pierwiastek wynosi: " << pierwiastek(2,0.01);
    getchar();
    return 0;
}
  
```

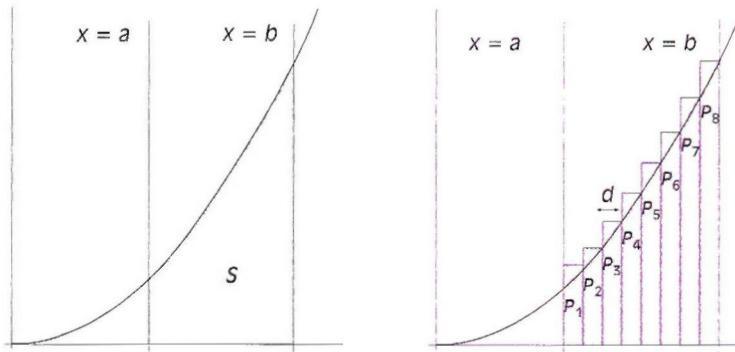
4.5

Przypominamy, że `fabs (a)` jest funkcją, która dla dowolnej liczby rzeczywistej wyznacza jej wartość bezwzględną. Wykorzystanie tej funkcji w programie wymaga wcześniejszego dołączenia biblioteki `cmath`.

#### 4. Implementacja klasycznych algorytmów iteracyjnych

### 4.5. Obliczanie pola obszaru ograniczonego wykresem funkcji

Dana jest funkcja ciągła, przyjmująca wartości dodatnie w przedziale  $(a; b)$  (mówiąc obrazowo, funkcja ciągła to taka, której wykres można narysować bez odrywania ołówka od kartki). Naszym zadaniem jest obliczenie pola obszaru  $S$  ograniczonego wykresem funkcji, osią  $OX$  oraz prostymi  $x = a$ ,  $x = b$ . Zakładamy przy tym, że wzór funkcji jest nam znany. Pole, które będziemy obliczać, jest równocześnie interpretacją tak zwanej **oznaczonej całki Riemanna** (ryc. 4.7).



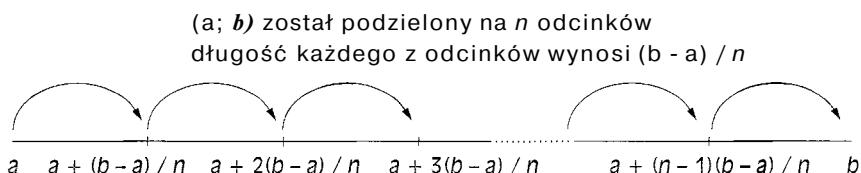
Ryc. 4.7. Ilustracja graficzna rozważanego problemu Ryc. 4.8. Pokrycie obszaru  $S$  prostokątami o podstawie  $d$

Opis metody obliczania pola powierzchni ograniczonej wykresem funkcji

Jednym ze sposobów znalezienia pola nierregularnego obszaru jest przybliżenie jego kształtu prostokątami o różnych wnętrzach (ryc. 4.8). Suma pól prostokątów  $P$  jest w przybliżeniu równa polu figury  $S$ :

$$P \approx P_1 + P_2 + P_3 + P_4 + P_5 + P_6 + P_7 + P_8$$

Podzielmy przedział  $(a; b)$  na zadaną liczbę równych odcinków (ryc. 4.9), a następnie wyznaczmy wartości funkcji dla prawych krańców odcinków.



Ryc. 4.9. Podział przedziału  $(a; b)$  na  $n$  odcinków (współrzędne punktów podziału)

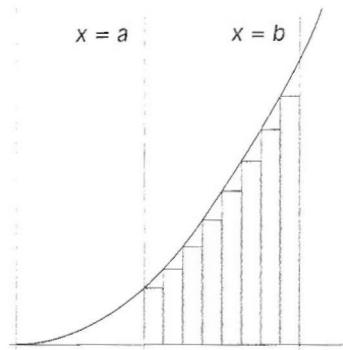
Każdy z prostokątów ma więc podstawę długości  $d = \frac{b-a}{n}$ , gdzie  $n$  jest liczbą odcinków (prostokątów), a drugi bok (wysokość) ma długość równą wartości funkcji w punkcie odpowiadającym prawemu końcowi

(-tego przedziału  $f(a + i \cdot d)$ ). Sumę pól obliczymy zatem w następujący sposób:

$$P = d \cdot f(a) + d \cdot f(a + d) + \dots + d \cdot f(b)$$

W naszym przykładzie tak obliczone pole  $P$  będzie wartością większą niż faktyczna wartość pola obszaru  $S$ . Jest to obliczenie „z nadmiarem”, gdyż suma prostokątów stanowi obszar nieco większy od obszaru  $S$  (co widać na ryc. 4.8).

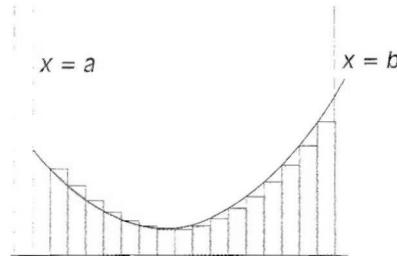
Mogliśmy również obliczać wartości funkcji dla lewych krańców przedziałów, na które podzieliliśmy początkowy przedział  $\langle a; b \rangle$ . W takim wypadku prostokąty, którymi przybliżamy obszar  $S$ , nie pokrywają go w całości i przybliżoną wartość pola wyznaczylibyśmy „z niedomiarem” (ryc. 4.10), korzystając ze wzoru:



$$P = d \cdot f(a) + d \cdot f(a + d) + \dots + d \cdot f(a + (n - 1) \cdot d)$$

Ryc. 4.10. Przykład na wyznaczenie pola z niedomiarem

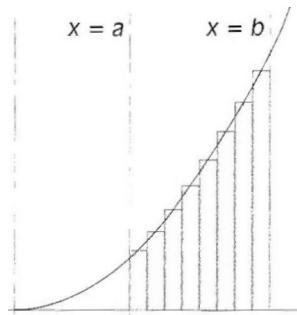
Na rycinie 4.11 pole obszaru pod wykresem funkcji przybliżamy sumą pól prostokątów, których wysokość jest równa wartości funkcji liczonej dla lewych krańców. Zauważ, że część obszaru będzie miała pole liczone z niedomiarem (tam, gdzie funkcja jest rosnąca), a część z nadmiarem (tam, gdzie funkcja maleje).



Ryc. 4.11. Obliczanie pola w przypadku funkcji, która nie jest monofoniczna w przedziale  $\langle a; b \rangle$ .

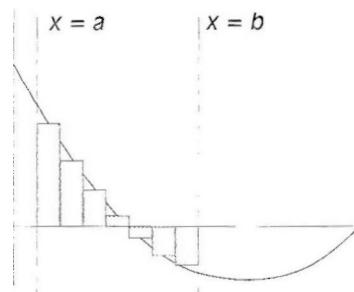
#### 4. Implementacja klasycznych algorytmów iteracyjnych

Przy zadanej liczbie prostokątów najdokładniejszy wynik uzyskamy, wyznaczając wysokości prostokątów na podstawie wyboru jakiegoś punktu pośredniego pomiędzy lewym a prawym krańcem przedziału, na przykład punktu leżącego w środku przedziału. W tej metodzie części prostokątów wystające poza obszar 5 są w przybliżeniu równoważone przez puste miejsca podobne do trójkątów, które powstały pod wykresem funkcji (ryc. 4.12).



Ryc. 4.12. Wysokość prostokątów jest obliczana w środkach ich podstawa.

Aby skonstruować uniwersalny algorytm, musimy uwzględnić również te funkcje, które przyjmują wartości ujemne (ryc. 4.13). Nie wiedząc z góry, czy funkcja przyjmuje dla danego argumentu wartość dodatnią czy ujemną, jako drugi bok prostokąta będziemy przyjmować wartość bezwzględną z wartości funkcji (długość boku prostokąta nie może być wartością ujemną).



Ryc. 4.13. Obliczanie obszaru ograniczonego wykresem funkcji, która zmienia swój znak.

Przejdźmy więc do konstrukcji algorytmu:

Specyfikacja problemu algorytmicznego i opis użytych zmiennych  
 Problem algorytmiczny: Wyznaczenie pola obszaru ograniczonego wykresem funkcji  $y = x^2$ , prostymi  $x = a$ ,  $x = b$  i osią  $OX$

#### 4.5. Obliczanie pola obszaru ograniczonego wykresem funkcji

Dane wejściowe:  $a, b \in \mathbb{R}$ ,  $n \in \mathbb{N}_+$  - liczba przedziałów, na które dzielimy przedział wyjściowy  $\langle a; b \rangle$

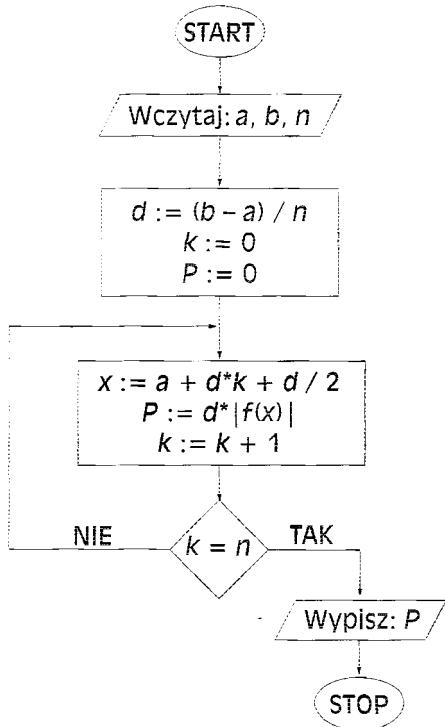
Dane wyjściowe:  $P \in \mathbb{R}_+$

Zmienne pomocnicze:  $d \in \mathbb{R}_+$  - długość przedziału;  $x \in \mathbb{R}$ ;  $k \in \mathbb{N}_+$

Zapiszmy algorytm za pomocą listy kroków:

1. Wczytaj wartości  $a$ ,  $b$ ,  $n$ .
2. Zmiennej  $d$  przypisz wartość  $\frac{b-a}{n}$ .
3. Zmiennej  $k$  przypisz 0, w miejsce  $P$  podstaw 0.
4. Zmiennej  $x$  przypisz wartość  $a + d \cdot k + \frac{d}{2}$ .
5. Do wartości  $P$  dodaj wartość  $d \cdot |f(x)|$ .
6. Zwięksź wartość  $k o 1$ .
7. Jeśli  $k$  jest równe  $n$ , wypisz wartość  $P$  i zakończ.
8. Wróć do kroku 4.

Gotowy schemat blokowy przedstawiono na rycinie 4.14:



**Ryc. 4.14.** Schemat blokowy algorytmu obliczającego przybliżoną wartość pola obszaru ograniczonego wykresem funkcji  $y = x^2$  w przedziale  $\langle a; b \rangle$

#### 4. Implementacja klasycznych algorytmów iteracyjnych

Oto pełny kod programu:

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

double funkcja(double x)      //funkcja, dla której liczymy całkę, tu x^2
{
    return x*x;
}

double pole_obszaru(int n, double a, double b)
{
    double P = 0;           //zmienna, która sumuje pola prostokątów
    double d = (b-a)/n;     //długość przedziałów, na jakie dzielimy <a; b>
    double x;               //punkty pośrednie przedziałów
    for (int k=0; k<n; k++)
    {
        x = a+(d*k)+(d/2);
        P = P+d*fabs(funkcja(x)); //suma pół prostokątów
    }
    return P;               //suma pół wszystkich prostokątów
}

int main()
{
    int ilosc;
    double a, b;
    cout << "Program oblicza pole obszaru ograniczonego";
    cout << "wykresem funkcji w przedziale <a; b>" << endl;
    cout << "podaj wartość lewego krańca przedziału: a ";
    cin >> a;
    cout << "podaj wartość prawego krańca przedziału: b ";
    cin >> b;
    cout << "Na ile przedziałów podzielić wyjściowy przedział? ";
    cin >> ilosc;
    cout << "wartość pola : " << pole_obszaru(ilosc,a,b);
    cin.ignore();
    getchar();
    return 0;
}
```

4.6

#### 4.6. Znajdowanie przybliżonej wartości miejsca zerowego funkcji ciągłej metodą połowienia przedziałów

Zajmiemy się teraz problemem znajdowania miejsca zerowego funkcji ciągłej. Wykorzystamy do tego **metodę połowienia przedziałów**, inaczej **bisekcji** - metoda ta polega na cyklicznym połowieniu przedziału i wybiorze tej części, w której znajduje się miejsce zerowe, aż do osiągnięcia żądanej dokładności w oszacowaniu miejsca zerowego funkcji lub natrafienia na argument będący miejscem zerowym.

#### 4.6. Znajdowanie przybliżonej wartości miejsca zerowego funkcji ciągiej metodą połowienia przedziałów

##### Przykład

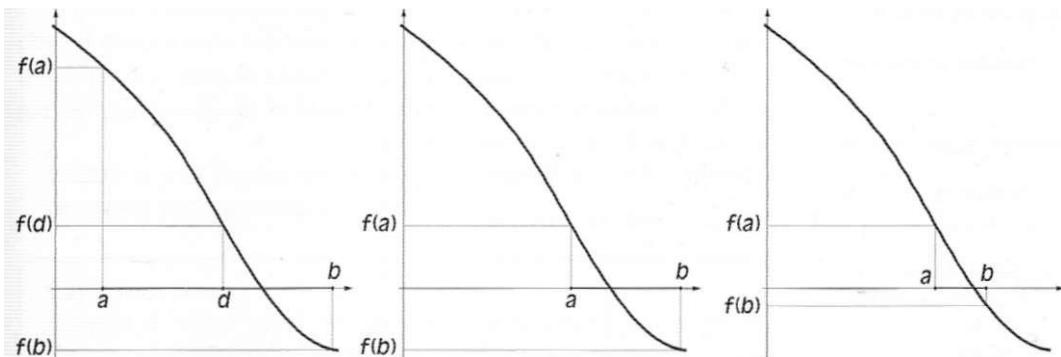
Znajdźmy w zadanym przedziale miejsce zerowe funkcji ciągiej.



Przypomnijmy, że miejscem zerowym funkcji nazywamy argument, dla którego funkcja przyjmuje wartość 0. Założmy, że mamy dane funkcję oraz przedział, na którego krańcach wartości funkcji są różnych znaków. Wynika stąd, że wewnątrz tego przedziału znajduje się argument, dla którego funkcja przyjmuje wartość 0. Oczywiście, może się zdarzyć, że wewnątrz takiego przedziału funkcja ma dwa miejsca zerowe lub więcej; my ograniczymy nasze zadanie do znalezienia jakiegokolwiek miejsca zerowego oraz wyznaczenia go zadaną dokładnością.

Mamy daną funkcję  $f(x)$  oraz dwa argumenty, dla których funkcja przyjmuje wartości przeciwnych znaków (ryc. 4.15). Dzielimy przedział na pół i dla środkowego argumentu badamy wartość funkcji. Jeśli wynosi 0, to algorytm kończy swe działanie. Jeśli zaś wartość funkcji dla środka przedziału nie jest równa 0, to będziemy postępować zgodnie ze schematem: jeśli jest ona dodatnia, to podmieniamy tym argumentem krańec, dla którego poprzednia wartość funkcji była dodatnia, w przeciwnym wypadku - drugi z krańców.

Opis metody połowienia przedziałów



Ryc. 4.15. Kolejne połówienia przedziału, w którym szukamy miejsca zerowego funkcji.

Nadal będziemy mieć do czynienia z przedziałem, w którego wnętrzu znajduje się miejsce zerowe funkcji, gdyż na krańcach funkcja wciąż przyjmuje wartości o przeciwnych znakach, teraz jednak badany przedział jest już dwukrotnie krótszy. Postępujemy tak, aż długość przedziału będzie odpowiednio mała, czyli będziemy mogli uznać, że jakakolwiek liczba należąca do niego jest przybliżonym miejscem zerowym, lub do momentu, gdy natrafimy dokładnie na miejsce zerowe funkcji.

Zapiszmy metodę połowienia przedziałów za pomocą listy kroków:

1. Wczytaj  $a, b$  - krańce wejściowego przedziału, oraz dokładność.
2. Jeżeli  $(a \cdot f(b)) > 0$ , to wypisz „W zadanym przedziale nie mogę znaleźć miejsca zerowego” i zakończ.

#### 4. Implementacja klasycznych algorytmów iteracyjnych

3. Jeśli  $f(a)$  jest równe 0, wypisz  $a$  i zakończ.
4. Jeśli  $f(b)$  jest równe 0, wypisz  $b$  i zakończ.
5. Jeśli  $b-a$  jest mniejsze lub równe dokładność, wypisz  $\frac{a+b}{2}$  i zakończ.
6. W miejsce zmiennej  $d$  podstaw wartość  $\frac{a+b}{2}$ .
7. Jeśli  $(a \cdot f(d)) < 0$ , to zmiennej  $b$  przypisz  $d$ , w przeciwnym wypadku zmiennej  $a$  przypisz  $d$  i wróć do kroku 2.

Poniżej umieszczamy pełny kod programu wykorzystującego omówiony algorytm. W funkcji głównej obsługujemy przypadek, gdy użytkownik poda krańce przedziału, dla których wartości funkcji przyjmują jednakowy znak, lub gdy podane zostaną takie krańce, że prawy ma mniejszą wartość od lewego. Funkcja zostaje wywołana już dla przedziału, w którym istnieje pierwiastek funkcji. W naszym przykładzie szukamy miejsca zerowego funkcji kwadratowej  $f(x) = 2x^2 - 4x$  (możesz oczywiście w tym miejscu wpisać jakąkolwiek inną funkcję, byle ciągłą w badanym przedziale).

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

double f(double x)
{
    return 2*x*x-4*x;
}

double m_zerowe(double a, double b, double dokladosc)
{
    double d = (a+b)/2;
    while ((b-a)>dokladosc && f(a)!=0 && f(b)!=0)
    {
        d = (a+b)/2;
        if (f(a)*f(d)<0)
            b = d;
        else
            a = d;
    }
    if (f(a)==0) return a;
    if (f(b)==0) return b;
    return d;
}

int main()
{
    double lewy, prawy, dokl;
    do
    {
        cout << "Podaj lewy kraniec przedzialu ";
        cin >> lewy;
        cout << "Podaj prawy kraniec przedzialu ";
        cin >> prawy;
    }
```

```

while (f(lowy)*f(prawy)>0 || prawy<=lewy);
cout << "Podaj dokladnosc ";
cin >> dokl;
cout << "Przyblizona wartosc miejsca zerowego: ";
cout << m_zerowe(lowy,prawy,dokl) << endl;
cin.ignore();
getchar();
return 0;

```

4.7.

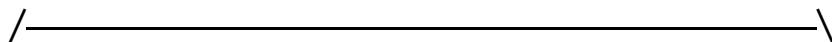
## 4.7. Metody Monte Carlo

Metody Monte Carlo to ogólna nazwa metod rozwiązywania złożonych problemów występujących w różnych dziedzinach, od fizyki, przez matematykę, teorię gier, do ekonomii, z wykorzystaniem komputerowego generowania liczb pseudolosowych. Jako ilustrację tych metod obliczymy przybliżenie liczby  $\pi$  i przeprowadzimy uproszczoną symulację ruchów Browna.

### 4.7.1. Obliczanie wartości liczby $\pi$ metodą Monte Carlo

Z liczbą  $\pi$  zetknęliśmy się na pewno wielokrotnie, na przykład obliczając obwód okręgu o promieniu  $r$  ze wzoru  $O = 2\pi r$ . Liczba  $\pi$ , dawniej zwana ludolfiną, ma wartość około 3,14. Jest to liczba przestępna, co oznacza, że nie jest pierwiastkiem żadnego równania o współczynnikach całkowitych.

Liczba  $\pi$  jest liczbą określającą stosunek długości okręgu do długości jego średnicy.



Czy wiesz, że...

Symbol  $\pi$  został użyty po raz pierwszy w 1706 roku przez angielskiego matematyka Williama Jonesa. Upowszechnił się dopiero w połowie XVIII wieku po wydaniu *Anality* Leonharda Eulera.

Wartość liczby  $\pi$  z dokładnością 50 cyfr po przecinku wynosi 3,14159265358979323846264338327950288419716939937510.



Korzystając z metody Monte Carlo, możemy określić przybliżoną wartość liczby  $\pi$ . Wyobraź sobie, że na kwadratową kartkę papieru, w której wpisane jest koło, pada drobny deszczek. Gdy policzysz stosunek liczby kropli wewnętrzko koła do liczby kropli, jakie spadły na całą kartkę, to w przybliżeniu okaże się on równy stosunkowi pola koła do pola kwadratu. Jeśli założymy, że kwadrat ma bok długości  $2r$ , to pole kwadratu wynosi  $4r^2$ . Pole koła wpisanego w ten kwadrat dane jest

wzorem  $\pi r^2$ . Zatem z zależności  $\frac{\pi r^2}{4r^2} = \frac{T}{R}$ , gdzie  $T$  jest liczbą kropli we

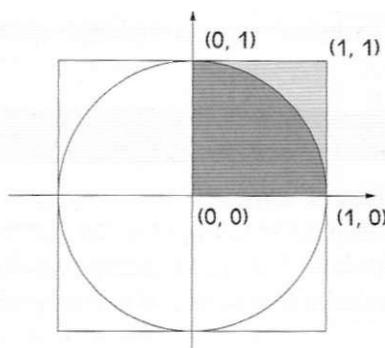
Opis metody  
wyznaczenia liczby  $\pi$

#### 4. Implementacja klasycznych algorytmów iteracyjnych

wnętrzu koła, a  $R$  - liczbą wszystkich kropli na kwadratowej kartce,łatwą jest wyliczyć wartość liczby  $\pi$ . Oczywiście, im więcej kropli deszczu, tym wynik będzie dokładniejszy. My nie będziemy rysować koła ani czeKać na deszcz. Generowanie punktów pozostawimy programowi, który napiszemy. Zakładamy, że zadanie rozwiązujemy w kartezjańskim układzie współrzędnych na płaszczyźnie, gdzie każdy punkt jest identyfikowany przez dwie współrzędne.

Opis metody  
wyznaczania  
liczby  $\pi$

W naszym programie możemy ustalić, że środek koła leży w początku układu współrzędnych, a promień wynosi jedną jednostkę. Stąd pole koła ma wartość  $\pi$  jednostek kwadratowych. Punkty należące do koła o środku w punkcie  $(0, 0)$  i promieniu 1 mają współrzędne spełniające nierówność:  $x^2 + y^2 \leq 1$ . Koło wpisane jest w kwadrat o boku dwóch jednostek, a wierzchołki kwadratu leżą w punktach tak, jak zaznaczono na rycinie 4.16.



Ryc. 4.16. Aby wyznaczyć liczbę  $\pi$ , wystarczy przeprowadzić obliczenia dla jednej ćwiartki koła.

Pole kwadratu wynosi cztery jednostki kwadratowe, a zatem wartość liczby  $\pi$  jest równa odpowiednio  $4 \frac{T}{R}$  (gdzie  $R$  jest liczbą wszystkich wygenerowanych punktów, a  $T$  - liczbą punktów należących do koła). Teraz wystarczy skonstruować pętlę, która  $R$ -krotnie wylosuje punkt (losowanie punktu będzie wygenerowaniem obydwu jego współrzędnych z przedziału  $<0; 1>$ , gdyż zamiast badać całą konfigurację obydwu figur, wystarczy wziąć pod uwagę tylko ich ćwiartki). Dla każdego punktu sprawdzimy, czy leży wewnątrz koła. Jeśli tak, zwiększymy wartość  $T$ .

Zacznijmy od specyfikacji problemu:

#### Specyfikacja problemu algorytmicznego i opis użytych zmiennych

**Problem algorytmiczny:** Znajdowanie przybliżonej wartości liczby  $\pi$

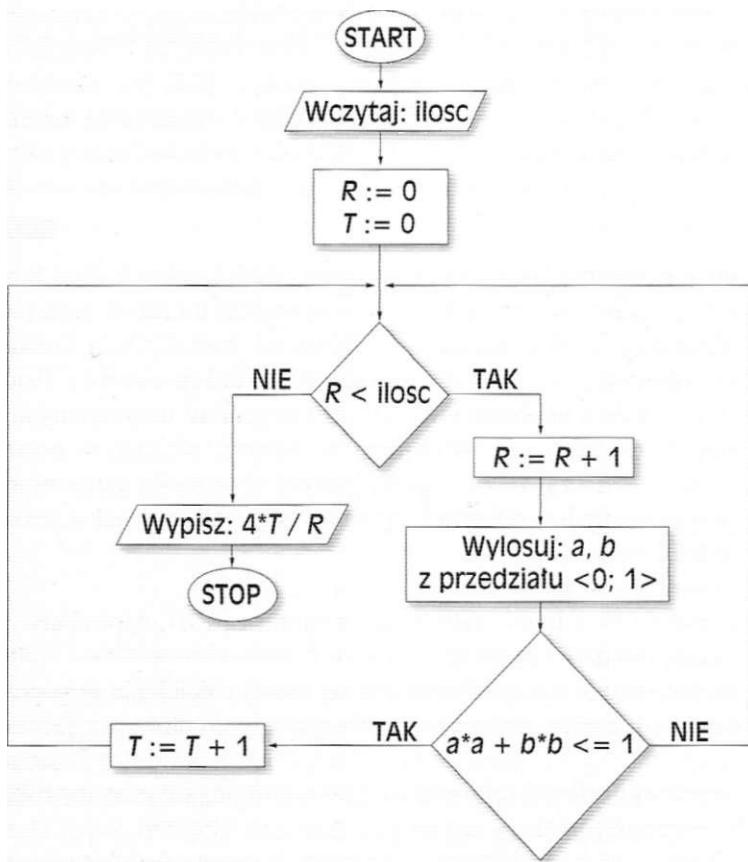
**Dane wejściowe:**  $ilosc \in N$  - ilość generowanych punktów

**Dane wyjściowe:** Przybliżona wartość %  
**Zmienne pomocnicze:**  $T, R \in \mathbb{V}_+$

Zapiszmy algorytm w postaci listy kroków:

1. Wczytaj  $ilosć$ .
2. Zmiennym  $R$  i  $T$  przypisz wartość 0.
3. Jeśli  $R$  nie jest mniejsze od  $ilosć$ , wypisz wartość wyrażenia  $\frac{4 \cdot T}{R}$  i zakończ.
4. Zwięksź wartość licznika  $R$ .
5. Wylosuj punkt należący do zaznaczonego fragmentu kwadratu.
6. Jeśli wylosowany punkt należy do zaznaczonej ćwiartki koła, zwięksź wartość licznika  $T$ .
7. Wróć do kroku 3.

Przeanalizuj jeszcze schemat blokowy algorytmu:



Ryc. 4.17. Schemat blokowy algorytmu wyznaczania przybliżonej wartości liczby  $\pi$  metodą Monte Carlo

#### 4. Implementacja klasycznych algorytmów iteracyjnych

Na koniec zapiszmy kod programu realizującego nasze zadanie:

```
#include <iostream>
#include <cstdio>
#include <cstdlib> // używamy w programie funkcji rand()
using namespace std;

int main()
{
    double a, b; // zmienne pomocnicze - współrzędne losowanego punktu
    long T = 0; // dzięki typowi long możemy losować wiele punktów
    long R;
    long ilosc;
    srand(time(NULL)); // inicjacja funkcji rand()
    cout << "Na podstawie ilu punktów znajdę wartość liczby pi? ";
    cin >> ilosc;
    for (R=0; R<ilosc; R++)
    {
        a = (double)rand()/(RAND_MAX); // losujemy liczby rzeczywiste
        b = (double)rand()/(RAND_MAX); // z zakresu <0,1>
        if (a*a+b*b<=1) T++; // jeśli punkt należy do koła, to zwiększamy licznik
    }
    cout << "Liczba pi ma wartość: " << (double)(4*T)/R;
    cin.ignore();
    getchar();
    return 0;
}
```

4.8

**Uwaga o losowaniu liczb.** Do generowania liczb losowych służy funkcja `rand()`. Generuje ona liczbę całkowitą między 0 i `RAND_MAX` (stałą symboliczną zdefiniowaną w bibliotece `cstdlib`). Funkcja `srand(time(NULL))` służy do zainicjowania funkcji `rand()`, dzięki czemu przy każdym uruchomieniu naszego programu uzyskujemy inną sekwencję liczb losowych. Argument w funkcji `srand` w postaci `time(NULL)` oznacza, że za wartość bazową w procesie generowania liczby przyjmowany jest odczytany z zegara czas, jaki upłynął w sekundach od 1970 roku.

Przykłady  
zastosowania  
funkcji `rand()`  
do generowania  
liczb losowych

Przyjrzyjmy się przykładowi:

Aby wygenerować liczbę naturalną  $d$  z zakresu  $(0; a)$ , napiszemy:  
`d = rand()%a;`

Po prawej stronie równości znajduje się reszta z dzielenia wygenerowanej liczby całkowitej przez  $a + 1$ ; takie wyrażenie może przyjąć wartości  $0, 1, 2, 3, \dots, a - 1, a$ .

Do generowania liczby całkowitej  $d$  z zakresu  $(a; b)$  użyjemy instrukcji:  
`d = a + rand()%(b - a + 1);`

Jeśli chcemy wygenerować liczbę rzeczywistą  $c$  z zakresu  $[a; b]$ , napiszemy:  
`c = a + ((float)rand()/(RAND_MAX))*(b - a);`

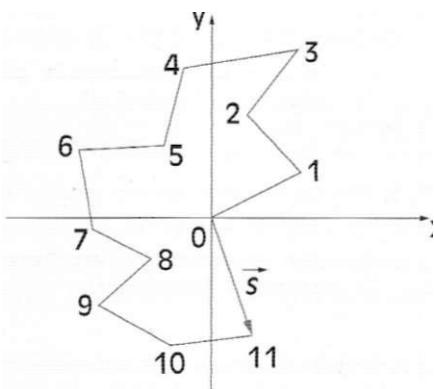
W tej instrukcji (tzw. instrukcji rzutowania) informujemy kompilator, że mimo iż funkcja `rand()` daje liczby całkowite, ma on je traktować tak, jakby to były liczby typu `float` (w szczególności więc użyć dzielenia rzeczywistego).

#### 4.7.2. Modelowanie ruchów Browna

W 1828 roku szkocki botanik Robert Brown opublikował sprawozdanie ze swoich wieloletnich badań. Posługując się mikroskopem, zaobserwował, że drobiny kurzu, sadzy i pyłki kwiatowe zawieszone w wodzie lub innych płynach wykonują nieustanne i nieregularne ruchy podobne do błądzenia. Po wielu eksperymentach Brown wysnuł wniosek, że ruchy te występują samoczynnie.

Spróbujmy za pomocą symulacji komputerowej zaprezentować przykładową trajektorię cząsteczki pływającej po powierzchni cieczy i obliczyć przemieszczenie tej cząsteczki w określonym czasie. Ponieważ interesują nas ruchy cząsteczki na powierzchni, obliczenia przeprowadzimy w przestrzeni dwuwymiarowej (2D). Założymy, że nasza cząsteczka startuje w początku kartezyjńskiego układu współrzędnych. Ustalmy też, że w każdym kroku cząsteczka przemieszcza się w dowolnym kierunku o wektor stałej długości - dla uproszczenia przyjmujemy 1 za długość kroku (ryc. 4.18).

- Opis problemu



Ryc. 4.18. Trajektoria cząsteczki po 11 ruchach oraz wektor przesunięcia  $\vec{s}$

Interesuje nas wypadkowy wektor przesunięcia  $\vec{s}$  cząsteczki po  $n$  ruchach. Czy średnia długość tego wektora będzie różna od zera oraz czy długość ta zależy od liczby wykonanych kroków?

Zadanie zrealizujemy następująco: cząsteczka startująca w początku układu współrzędnych może wykonać ruch w dowolnym kierunku. Musimy wyznaczyć współrzędną  $x$ , i  $y$ , nowego położenia cząsteczki po pierwszym ruchu. Z położenia o współrzędnych  $x$ , i  $y$ , cząsteczka wykonuje

- Opis metody

#### 4. Implementacja klasycznych algorytmów iteracyjnych

następny ruch w dowolnym kierunku, obliczamy wówczas położenie  $x_i$  i  $y_i$ , częsteczki po drugim kroku itd. Na koniec wyznaczamy położenie częsteczki po  $n$  krokach  $x_n$  i  $y_n$ . Pozostaje nam już tylko wyznaczyć długość wektora przesunięcia częsteczki po  $n$  krokach. Ponieważ częsteczka startowała z punktu o współrzędnych  $(0, 0)$ , długość ta wyniesie  $|\vec{s}| = \sqrt{x_n^2 + y_n^2}$ .

Kierunek przesunięcia częsteczki w każdym kroku obliczymy za pomocą generatora liczb losowych, który wylosuje kąt z przedziału  $(0; 2\pi)$ . Następnie dla wyznaczenia współrzędnych posłużymy się wzorami:

$$x_k = x_{k-1} + r \cdot \cos \varphi$$

$$y_k = y_{k-1} + r \cdot \sin \varphi$$

gdzie:  $k = 0, 1, \dots, n$

$r$  - długość jednego kroku (my przyjęliśmy 1)

$\varphi$  - kąt między kierunkiem ruchu częsteczki a osią  $OX$

Zapiszmy to w programie:

```
#include <iostream>
#include <cstdio>
#include <cstdlib>           // do zastosowania funkcji rand()
#include <cmath>              // do zastosowania funkcji sin(), cos() i sqrt();
#include <fstream>            // do obsługi plików
using namespace std;

int main()
{
    ofstream wyj("brown.xls");      // deklaracja i otworzenie pliku wyjściowego
    const float pi = 3.14159;       // deklaracja stałej pi
    float x, y, s, fi;
    long i, n;
    cout << "Ile chcesz ruchow: ";
    cin >> n;
    x = 0; y = 0;
    wyj << x << "\t" << y << endl; // zapis początkowych współrzędnych do pliku
    srand(time(NULL));             // inicjacja generatora liczb
    for (i=0; i<n; i++)
    {
        fi = (float)rand()/(RAND_MAX +1)*2*pi; // losowanie kierunku
        x = x+cos(fi);                      // nowa współrzędna x
        y = y+sin(fi);                      // nowa współrzędna y
        wyj << x << "\t" << y << endl; // kolejne współrzędne zapisujemy do pliku
    }
    s = sqrt(x*x+y*y);             // długość przemieszczenia
    cout << endl << "Czasteczka przemiescila sie na odleglosc: " << s;
    wyj.close();                   // zamykamy plik
    cin.ignore();
    getchar();
    return 0;
}
```

4.9

Na czerwono zaznaczyliśmy linie, których w tej chwili jeszcze nie musisz rozumieć (wystarczy, że je przepiszesz). Linie te służą do zapisania do pliku **brown.xls** współrzędnych poszczególnych ruchów cząsteczki. Dzięki temu plikowi można łatwo wykreślić trajektorię cząsteczki w arkuszu kalkulacyjnym, stosując wykres XY-punktowy (polecamy tę pouczającą zabawę). Będzie można również sporządzić wykres odległości, na jaką przemieściła się cząsteczka w funkcji czasu (czas jest proporcjonalny do liczby kroków). Oczywiście, jeśli interesuje cię tylko odległość, na jaką przemieściła się nasza cząsteczka od początku układu współrzędnych, nie musisz wpisywać linii fioletowych.

### Pytania kontrolne

1. Omów algorytm znajdujący element maksymalny w ciągu liczb podanych z klawiatury.
2. Powiedz, w jaki sposób można obliczyć przybliżoną wartość pola nie-regularnego obszaru ograniczonego wykresem funkcji, osią  $OX$  oraz prostymi prostopadłymi do osi  $OX$ .
3. Omów metodę połowienia przedziałów w celu wyznaczenia miejsca zerowego funkcji ciągiej.
4. Co to są metody Monte Carlo? Podaj dwa przykłady problemów, które można rozwiązać, korzystając z tej metody.

### Ćwiczenia

1. Napisz program, który skraca ułamek podany z zewnątrz w postaci parę liczb: pierwszą z nich traktujemy jako licznik ułamka, drugą jako jego mianownik. Wynik powinien być wypisany w postaci  $a : b$ , gdzie  $a$  jest nowym licznikiem,  $b$  - nowym mianownikiem ułamka po skróceniu.
2. Pan Nowak wpłacił w styczniu do banku 100 zł, a każdego następnego miesiąca będzie wpłacał o 10 zł więcej. Napisz algorytm, który oblicza stan konta pana Nowaka po  $n$  miesiącach oraz kwotę ostatniej wpłaty.
3. Narysuj schemat blokowy algorytmu obliczający ilość całkowitych dzielników liczby podanej przez użytkownika i napisz odpowiedni program.
4. Napisz algorytm, który wypisuje największą z liczb pobranych z klawiatury wraz z informacją, ile razy liczba ta wystąpiła w ciągu. Informacja o ilości liczb jest wartością podawaną na początku działania algorytmu.



4. Implementacja klasycznych algorytmów iteracyjnych

5. Napisz program, który pobiera na wejściu ciąg liczb rzeczywistych aż do momentu, gdy wprowadzone zostanie 0, i jako wynik wyświetla największą ilość podanych kolejno liczb dodatnich.
6. Napisz program, który obliczy pole powierzchni ograniczone wykresem funkcji  $y = |x| - 8$ , osią  $OX$  oraz prostymi  $x = 2, x = 8$ .
7. Wykorzystując metodę Monte Carlo, oblicz pole obszaru ograniczonego wykresem funkcji  $y = x^2$ , prostymi  $x = 0, x = 1$  oraz osią  $OX$ .

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

W tym rozdziale omawiamy złożony typ danych, jakim jest tablica. Można w niej przechowywać w sposób uporządkowany ciąg danych tego samego typu. Poznasz też algorytmy działające na tablicach, między innymi algorytm przeszukujący tablicę w celu znalezienia wyróżnionego elementu w tablicy - maksymalnego lub minimalnego. Nauczysz się sortować tablicę, czyli ustawać elementy w zadanej kolejności (od najmniejszego do największego).

### 5.1. Charakterystyka tablic

Wyobraź sobie, że chcesz napisać program, który pobiera od użytkownika jego oceny, oblicza ich średnią, a na koniec wypisuje wszystkie oceny, z których ta średnia została obliczona. Przy założeniu, że wprowadzimy  $k$  liczb, realizacja tego zadania wymagałaby zadeklarowania  $k$  zmiennych, a operowanie na nich w wypadku dużych  $k$  byłoby niezwykle uciążliwe (zastanów się, jak wyglądałaby deklaracja zmiennych, jeśli wprowadzilibyśmy 100 ocen). Tu właśnie przychodzi nam z pomocą złożona struktura danych, jaką jest tablica.

#### Definicja

Tablicą nazywamy ciąg ustalonej liczby elementów tego samego typu, do których odwołujemy się za pośrednictwem wspólnej nazwy. Dostęp do konkretnego elementu tablicy uzyskuje się za pomocą nazwy i indeksów określających pozycję, jaką element zajmuje w tablicy.

Tablica może mieć kilka wymiarów lub jeden, w zależności od problemu algorytmicznego. W **tablicy jednowymiarowej** każdy element jest identyfikowany przez nazwę tablicy i jeden indeks. Na przykład lista uczniów w klasie może być przedstawiona jako jednowymiarowa tablica nazwisk, gdzie indeksem jest numer w dzienniku, czyli jedna liczba. W **tablicy dwuwymiarowej** zaś każdy element jest jednoznacznie identyfikowany przez nazwę tablicy i parę indeksów. Na przykład w tablicy dwuwymiarowej może być przedstawiony zbiór pionków na szachownicy: każda figura ma podane dwie współrzędne, czyli dwie liczby.

W C++ pierwszy element tablicy jednowymiarowej ma indeks równy零.

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

### 5.1.1. Tablice jednowymiarowe

Deklaracja tablicy : Aby pracować z tablicą, musimy najpierw ją zadeklarować, tak jak każdą zmienną w programie. Deklaracja tablicy ma postać:

```
typ_elementow nazwa_tablicy[liczba_elementow];
```

**Stosowane oznaczenia:**

**typ\_elementow** - należy podać nazwę typu elementów w tablicy;

**nazwą tablicy** — jest dowolną nazwą (podlega tym samym regułom poprawności, co nazwa każdej innej zmiennej);

**liczba\_elementow** - w nawiasie kwadratowym należy podać liczbę elementów tablicy; musi to być liczba podana wprost (tak zwany literal), na przykład 7, lub stałą typu całkowitego (zdefiniowana wcześniej).

Załóżmy, że chcemy zadeklarować tablicę, w której przechowywać będziemy osiem liczb całkowitych. Jeśli tablica ma mieć nazwę **tab**, to deklaracja takiej tablicy w programie będzie wyglądać następująco:

```
int tab[8];
```

Na rycinie 5.1 przedstawiono graficznie część obszaru pamięci komputera, gdzie zadeklarowana została tablica ośmioelementowa liczb całkowitych. Elementy tablicy przedstawiono w postaci prostokątów, a nad prostokątami zapisano przykładowe adresy komórek pamięci komputera - zauważ, że adresy te zmieniają się o 4. Liczba 4 jest rozmiarem typu **int** elementów tej tablicy. **Elementy tablicy zajmują w pamięci komputera obszar ciągły**, to znaczy, że każdy kolejny element następuje bezpośrednio po poprzednim.

7840	7844	7848	7852	7856	7860	7864	7868
3	247	0	0	12547	-1	1	-247

tab[0] tab[1] tab[2] tab[3] tab[4] tab[5] tab[6] tab[7]

**Ryc. 5.1.** Tablica jednowymiarowa ośmioelementowa po deklaracji, przed wypełnieniem żądanymi wartościami

Kolejne elementy tablicy są identyfikowane jako **tab[0], tab[1], ..., tab[7]**. Pierwszy element ma indeks o wartości 0, dlatego ósmy element ma indeks o wartości 7.

Rycina 5.1 przedstawia tablicę po deklaracji, ale jeszcze przed przypisaniem wartości do tablicy. Zauważ, że tablica nie jest pusta, lecz ma już jakieś wartości. **Liczby te są przypadkowymi wartościami**, jakie mogą pojawić się w tablicy, zanim wpiszemy do niej właściwe wartości.

Wypełnianie tablicy : Tablicę można zadeklarować i od razu nadać wartości jej elementom za pomocą instrukcji (przy tak skonstruowanej deklaracji nie trzeba wpisywać rozmiaru tablicy, ponieważ kompilator poznaje po liczbie elementów w klamrze):

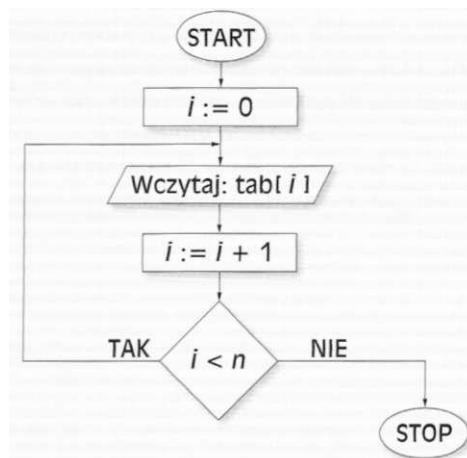
```
int tab[] = {6,8,7,2,3,5,7,2};
```

Tablica będzie teraz wyglądać jak na rycinie 5.2:

6	8	7	2	3	5	7	2
---	---	---	---	---	---	---	---

Ryc. 5.2. Tablica jednowymiarowa ośmioelementowa po deklaracji i nadaniu wartości elementom tablicy

Ten sposób wypełniania tablicy można wykorzystać tylko na etapie pisania kodu programu. Do wypełnienia tablicy podczas wykonywania programu używać będziemy najczęściej sposobu iteracyjnego, przedstawionego na schemacie blokowym (ryc. 5.3). Schemat przedstawia wypełnianie tablicy o nazwie `tab`, która ma  $n$  elementów (gdzie  $n$  oraz zmieniona pomocnicza  $i$  mają wartości całkowite dodatnie).



Ryc. 5.3. Schemat blokowy wypełniania tablicy jednowymiarowej

Zmienna  $i$  jest licznikową zmienną pomocniczą, która przyjmując wartości indeksów kolejnych elementów tablicy, pozwala przypisać tym elementom pobrane z zewnątrz wartości.

Przypominamy, że jeśli tablica o nazwie `tab` ma  $n$  elementów, to jej elementami są: `tab[0]`, `tab[1]`, ..., `tab[n-1]`. Indeksami tablicy są liczby ze zbioru  $\{0, \dots, n-1\}$ . Odwołanie się do elementu o indeksie innym aniżeli liczba z tego zbioru nazywamy **przekroczeniem zakresu tablicy** (lub wyjściem poza zakres tablicy). Przekroczenie zakresu, na przykład przy wypełnianiu tablicy, spowoduje zapis poza obszarem pamięci przeznaczonym na zadeklarowaną tablicę. W obszarze tym mogą się znajdować inne zmienne tego programu lub nawet dane innych programów. Wpisanie tam nieprawidłowych wartości grozi więc błędem w programie, zawieszeniem uruchomionego programu lub nawet całego systemu. Jest to niezgodne z regułami języka C++.

Załóżmy, że tablica jest  $n$ -elementowa, gdzie  $n$  jest dodatnią liczbą całkowitą, ustaloną w trakcie deklarowania tablicy. Fragment kodu realizujący wypełnianie tablicy liczbami podanymi z klawiatury ma postać:

Problem  
przekroczenia  
zakresu tablicy

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

```
for (int i=0; i<n; i++)
{
    cout << "Podaj wartość elementu";
    cin >> tab[i];
}
```

Tablica jako |  
argument funkcji :

Jeśli w programie chcemy wypełnić kilka tablic, to wygodniejsze będzie umieszczenie powyższego kodu w funkcji, którą nazwiemy na przykład **wypełnij**. Wówczas parametrami przekazywanymi do funkcji będą: tablica, którą chcemy wypełnić, oraz liczba elementów tablicy do wypełnienia.

Funkcja, po otrzymaniu tablicy jako argumentu, ma pełen dostęp do zawartości komórek, w których przechowywane są elementy tablicy - działa bezpośrednio na tablicy. Nie jest więc potrzebne odwołanie przez referencję, aby z wnętrza funkcji modyfikować wartości elementów tablicy.

W celu sprawdzenia efektów działania funkcji **wypełnij** zdefiniowaliśmy funkcję **wyswietl**, która wyświetli nam na ekranie wypełnione tablice:

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

void wypełnij(int tab[8])
{
    for (int i=0; i<8; i++)
    {
        cout << "Podaj wartość elementu ";
        cin >> tab[i];
    }
}

/***********************
void wyswietl(int tab[8])
{
    for (int i=0; i<8; i++)
        cout << tab[i] << " ";
    cout << endl;
}

***** Początek funkcji głównej programu *****
int main()
{
    int tablica1[8], tablica2[8];
    wypełnij(tablica1); // wywołanie funkcji wypełnij dla tablica1
    wypełnij(tablica2); // wywołanie funkcji wypełnij dla tablica2
    wyswietl(tablica1); // wywołanie funkcji wyswietl dla tablica1
    wyswietl(tablica2); // wywołanie funkcji wyswietl dla tablica2
    cin.ignore();
    getchar();
    return 0;
}
```

5.1

Zwróć uwagę na postać parametru aktualnego przy wywołaniu funkcji `wypełnij` oraz funkcji `wyświetl`.

Przekazując funkcji tablicę jako argument (czyli w wywołaniu funkcji), podajemy tylko i wyłącznie jej nazwę, bez nawiasów kwadratowych i bez podawania rozmiaru.

Do obsługi obydwu tablic zastosowaliśmy zdefiniowane przez nas funkcje. Zauważ, że argumentem przekazywanym do funkcji jest tu właśnie tablica, na której funkcje wykonają zdefiniowane operacje: a zatem jedna z nich wypełni tablicę, drugą ją wyświetli. Moglibyśmy za pomocą tych funkcji zainicjować i wyświetlić każdą tablicę o elementach całkowitych, ale pod warunkiem, że miałyby ona osiem elementów. Co więc w wypadku, gdy chcemy wypełnić i wyświetlić kilka tablic, ale o różnej liczbie elementów? Czy musimy napisać kilka różnych funkcji, przeznaczonych dla tablic o różnych liczbach elementów? Oczywiście, nie. Napiszemy jedną funkcję, ale oprócz tablicy przekażemy do jej wnętrza informację o rozmiarze tej tablicy:

```
void wypelnij(int tab[], int rozmiar)
{
    for (int i=0; i<rozmiar; i++)
    {
        cout << "Podaj wartosc elementu";
        cin >> tab[i];
    }
}
```

5.2

Zauważ, że deklarując tablicę jako argument funkcji, nie podaliśmy w nawiasach kwadratowych jej rozmiaru - nie ma takiej potrzeby. W wypadku tablicy jednowymiarowej na etapie definicji funkcji kompilatorowi wystarczy znajomość typu elementów tablicy (nieco inaczej wygląda to w tablicach o większej liczbie wymiarów).

Fragment funkcji głównej z zastosowaniem zdefiniowanej funkcji `wypełniającej` dwie tablice o różnych liczbach elementów wygląda następująco:

```
int main()
{
    int tablica1[10];
    int tablica2[4];
    wypelnij(tablica1,10);
    wypelnij(tablica2,4);
    return 0;
}
```

5.3

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

Skoro już wiesz, że tablice mogą być argumentami przekazywanymi do funkcji, to analiza działania poniższej funkcji pozwoli ci łatwo określić, jakie zadanie ma ona do wykonania:

```
float oblicz(float tab_1[5], float tab_2[5])  
{  
    float suma = 0;  
    for (int i=0; i<5; i++)  
        suma = suma+tab1[i]+tab2[i];  
    return suma;  
}
```

5.4

Funkcja ta pobiera dwie jednowymiarowe tablice pięcioelementowe, a jej wynikiem jest suma wszystkich elementów obu tablic. Moglibyśmy napisać funkcję, która sumuje elementy pojedynczej tablicy, wywołać tę funkcję dla dwóch tablic i zsumować oba wyniki. My dokonaliśmy tego w jednej funkcji, aby pokazać, że funkcji można przekazać więcej niż jedną tablicę.

### 5.1.2. Tablice wielowymiarowe

Elementami tablicy dwuwymiarowej są jednowymiarowe tablice, z których każda ma taką samą liczbę elementów tego samego typu. Taka „tablica tablic” jest w matematyce zwana **macierzą**. Na rycinie 5.4 przedstawiliśmy dwuwymiarową tablicę o czterech wierszach i czterech kolumnach. Aby zadeklarować tablicę dwuwymiarową, nadajemy jej nazwę, podajemy typ elementów, jakie będzie przechowywać, oraz określamy liczbę wierszy i liczbę kolumn. Jeśli liczba wierszy i kolumn jest taka sama, to mówimy, że **tablica jest kwadratowa**.

4	5	7	9
2	4	8	2
1	5	6	8
6	4	78	5

Ryc. 5.4. Dwuwymiarowa tablica kwadratowa  $4 \times 4$

Deklarowanie :      Oto przykładowe deklaracje tablic dwuwymiarowych:  
tablicy :      int tabl[4][4];                  // tablica kwadratowa  
dwuwymiarowej :      // o elementach całkowitych  
                    char tab2[2][3];                  // tablica znaków  
                    // o 2 wierszach i 3 kolumnach  
                    float tab3[2][4];                  // tablica liczb rzeczywistych:  
                    // ma 2 wiersze i 4 kolumny

Deklaracja `tabl[w][k]` oznacza deklarację tablicy o nazwie: `tabl`; litera `w` oznacza liczbę wierszy tablicy, litera `k` – liczbę kolumn tablicy, gdzie `w` i `k` są stałymi całkowitymi.

Podobnie w tablicach jednowymiarowych również elementom tablicy dwuwymiarowej możemy przypisać wartości początkowe już w momencie deklaracji:

```
int liczby[2][3] = {{2,3,4},{7,1,5}};
```

W ten sposób otrzymujemy tablicę dwuwymiarową z dwoma wierszami i trzema kolumnami o wartościach takich jak w klamrach.

Nadanie początkowych wartości poszczególnym elementom tablicy dwuwymiarowej, którą nazwaliśmy liczbą, może wyglądać na przykład:

```
liczby[0][0] = 2;  
liczby[0][1] = 3;
```

•

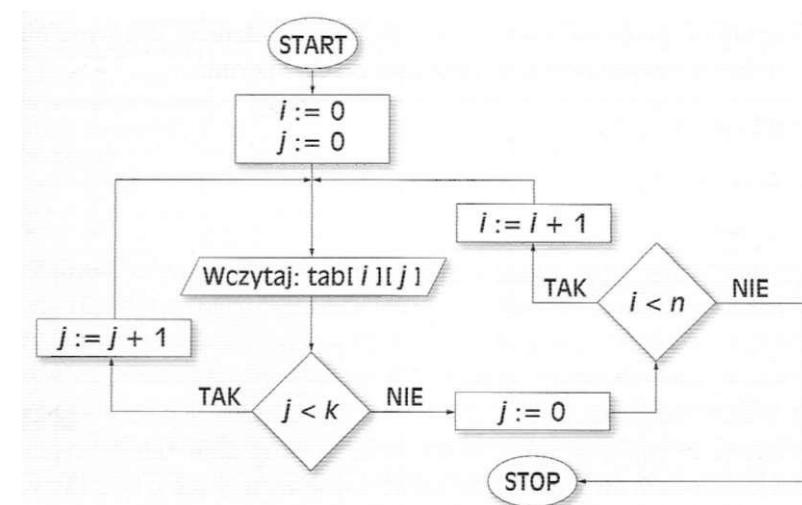
```
liczby[2][3] = 5;
```

Wypełnienie tablicy dwuwymiarowej polega na wypełnieniu wartości wszystkich wierszy, przy czym każdy wiersz jest właśnie tablicą jednowymiarową. Do zapełnienia elementów tablicy żądanymi wartościami zastosujemy pętlę. W każdym przebiegu pętli musimy nadać wartości tylu elementom, ile ich jest w jednym wierszu, co również wykonamy za pomocą pętli. Otrzymamy więc pętlę **zagnieżdzoną**, czyli pętlę wykonującą się w innej pętli.

Przeanalizuj schemat blokowy algorytmu wypełniającego tablicę dwuwymiarową (rye. 5.5) o wymiarach  $n \times k$ , gdzie  $n$  jest liczbą wierszy tablicy, a  $k$  - liczbą kolumn.

## Zainicjowanie tablicy dwuwymiarowej na etapie deklaracji

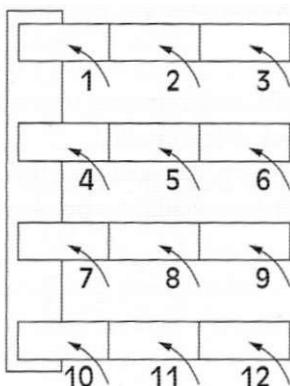
## Wypełnianie tablicy dwuwymiarowej



Ryc. 5.5. Schemat blokowy algorytmu wypełniającego tablicę dwuwymiarową

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

Łatwo ci będzie zrozumieć ten algorytm, jeśli spojrzesz na tablicę dwuwymiarową jak na tablicę jednowymiarową, której elementami są również tablice jednowymiarowe. Na rycinie 5.6 została zaznaczona kolejność wypełniania komórek w dwuwymiarowej tablicy, która ma cztery wiersze i trzy kolumny:



Ryc. 5.6. Kolejność wypełniania tablicy w schemacie blokowym z ryciną 5.5

Tablice : Analogicznie wypełniamy tablicę trójwymiarową - jest to tablica jednowymiarowa, której elementami są tablice jednowymiarowe o elementach będących również tablicami jednowymiarowymi. Nie powinno ci zatem sprawić problemu skonstruowanie algorytmu wypełniającego elementy tablicy o dowolnym, skończonym wymiarze - zwiększeniu ulegnie tylko liczba pętli zagnieżdżonych. W praktyce bardzo rzadko mają zastosowanie tablice o większych wymiarach niż trzy - najczęściej używane są tablice jedno- i dwuwymiarowe.

Zacznij od przeanalizowania funkcji, której zadaniem jest wypełnienie tablicy dwuwymiarowej wartościami rzeczywistymi:

```
void wypelnij(float tab[4][3])
{
    for (int i=0; i<4; i++)
    {
        for (int j=0; j<3; j++)
        {
            cout << "Podaj wartosc elementu ";
            cin >> tab[i][j];
        }
    }
}
```

5.5

Funkcję tę możemy teraz wykorzystać w programie, który wypełni w ten sam sposób dwie lub więcej tablic dwuwymiarowych o tych samych rozmiarach, a ponadto wyświetli je na ekranie monitora za pomocą zdefiniowanej przez nas funkcji.

```

#include <iostream>
#include <cstdio>
#include <iomanip>
using namespace std;

void wypelnij(float tab[5][7])
{
    for (int i=0; i<5; i++)
    {
        for (int j=0; j<7; j++)
        {
            cout << "Podaj wartosc elementu: ";
            cin >> tab[i][j];
        }
    }
}

void wyswietl(float tab[5][7])
{
    for (int i=0; i<5; i++)
    {
        for (int j=0; j<7; j++)
        {
            cout << setw(4) << tab[i][j]; // na zapis zmiennej przeznaczamy 4 miejsca
                                         // aby każdy następny wiersz pojawił się
                                         // w nowej linii
            cout << endl;
        }
    }
}

int main()
{
    float tab1[5][7], tab2[5][7];
    wypelnij(tab1);
    wypelnij(tab2);
    cout << endl;
    wyswietl(tab1);
    cout << endl;
    wyswietl(tab2);
    cin.ignore();
    getchar();
    return 0;
}

```

5.6

Zauważ, że dołączliśmy w tym programie jeszcze jeden plik biblioteczny: `iomanip`. W tym pliku są zdefiniowane tak zwane **manipulatory**, służące do formatowania wejścia i wyjścia, my skorzystaliśmy z jednego z nich: funkcji `setw`, za pomocą której ustalamy ilość miejsca przeznaczonego do zapisu wartości zmiennej. Chcemy, aby wyświetlona tablica miała wyrównane kolumny, dlatego każdej liczbie przekształćmy tę samą ilość miejsca do zapisu.

Skoro potrafisz już wypełnić tablicę dwuwymiarową, to zmodyfikujmy funkcję wypełniającą tak, aby tablica została wypełniona w pewien szczególny sposób.

Wykorzystanie  
funkcji z biblioteki  
`iomanip`

### Przykład

Dane są tablica dwuwymiarowa oraz dwie liczby:  $a$  oraz  $r$ . Wypełnimy kolejne elementy tablicy liczbami: pierwszy element liczbą  $a$ , każdy następny zaś sumą wartości poprzednio wypełnianego elementu i liczby  $r$ .

Wypełnianie  
tablicy  
dwuwymiarowej  
wyrazami ciągu

Jeśli znasz z lekcji matematyki ciągi liczbowe, to wiesz, że kolejne elementy tablicy będą tworzyć wyrazy ciągu arytmetycznego o pierwszym wyrazie równym  $a$  i różnicą wynoszącą  $r$ . Konstruując funkcję wypełniającą tablicę w ten charakterystyczny sposób, przekażemy do funkcji trzy parametry: tablicę, pierwszy wyraz ciągu arytmetycznego  $a$ , oraz  $r$  - różnicę ciągu. Otrzymamy następujący zapis:

```
void wypelnij(float tab[5][7], float a, float r)
{
    for (int i=0; i<5; i++)
    {
        for (int j=0; j<7; j++)
        {
            tab[i][j] = a;
            a = a+r;
        }
    }
}
```

5.7

Pierwszy z wypełnianych elementów tablicy otrzyma więc wartość zmiennej  $a$ , każdy następny wypełniony zostanie wartością poprzedniego elementu zwiększoną o  $r$ .

Po wykonaniu funkcji przy wywołaniu

```
wypełnij(tablica,2,3)
```

tablica wyświetlona według zdefiniowanej w poprzednim programie funkcji wyświetli będzie wyglądać tak:

2	5	8	11	14	17	20
23	26	29	32	35	38	41
44	47	50	53	56	59	62
65	68	71	74	77	80	83
86	89	92	95	98	101	104

Widzimy więc, że elementy tablic zachowują się jak zwykłe zmienne i tak należy je traktować i obsługiwać.

### Przykład

Program znajduje średnią ocen, zapisanych w jednowymiarowej dwunastoelementowej tablicy, wyświetla tę średnią, a następnie podaje wszystkie oceny, z których obliczył średnią.

W celu wyznaczenia średniej policzymy sumę wszystkich ocen, a następnie otrzymany wynik podzielimy przez liczbę ocen; w naszym przykładzie jest ich maksymalnie dwanaście.

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

void wypelnij(float tab[], int ilosc)
{
    for (int i=0; i<ilosc; i++)
    {
        cout << "Podaj ocene: ";
        cin >> tab[i];
    }
}

float zsumuj(float tab[], int ilosc)
{
    float suma = 0;
    for (int i=0; i<ilosc; i++)
        suma = suma+tab[i];
    return suma;
}

void wyswietl(float tab[], int ilosc)
{
    for (int i=0; i<ilosc; i++)
        cout << setw(3) << tab[i];
}

int main()
{
    float oceny[12];
    int ile;
    cout << "Ile podasz ocen? ";
    cin >> ile;
    wypelnij(oceny,ile);
    cout << "Osiagnales srednia ocen: " << zsumuj(oceny,ile)/ile;
    cout << " z nastepujacych stopni czastkowych: " << endl;
    wyswietl(oceny,ile);
    cin.ignore();
    getchar();
    return 0;
}
```

5.8

Jeśli mielibyśmy pewność, że użytkownik zawsze będzie chciał policzyć średnią z dwunastu ocen, moglibyśmy zdefiniować wszystkie funkcje od razu dla tablicy dwunastoelementowej. W naszym przykładzie jako argumenty przekazujemy funkcjom zarówno tablicę, jak i liczbę elementów, aby uwzględnić sytuację, gdy uczeń na przykład jest z jakiegoś przedmiotu zwolniony lub nie jest klasyfikowany - wówczas średnia ocen jest liczona odpowiednio z mniejszej ilości ocen.

## 5.2. Klasyczne algorytmy działające na tablicach

Poznałeś już tablicę i podstawową obsługę zmiennych tego typu, pora więc przejść do realizacji algorytmów, które działają na tablicach.

### 5.2.1. Przeszukiwanie liniowe tablicy jednowymiarowej



Opis metody  
przeszukiwania  
liniowego

#### Przykład

Zajmijmy się problemem poszukiwania wyróżnionego elementu w jednowymiarowej tablicy. Zakładamy przy tym, że element, którego szukamy, może się w tablicy znajdująć lub nie.

W tym celu kolejno, począwszy od pierwszego elementu, będziemy sprawdzać, czy napotkana wartość jest elementem szukanym. Nazwijmy go szuk. Jeśli już pierwszy element tablicy (oczywiście pierwszym elementem jest element indeksowany zerem!) okaże się poszukiwanym elementem, przeszukiwanie zakończymy, a na wyjściu pojawi się informacja, że element szukany znajduje się w tablicy. W przeciwnym wypadku sprawdzamy, czy drugi element jest równy elementowi szukanemu, i tak postępujemy aż do ostatniego elementu tablicy. Przeszukiwanie tablicy metodą element po elemencie nazywamy **przeszukiwaniem liniowym**. Po każdym przebiegu pętli musimy oczywiście skontrolować, czy nie sprawdzamy ostatniego elementu tablicy, aby nie wyjść poza zakres tablicy.

#### Specyfikacja problemu algorytmicznego i opis użytych zmiennych

**Problem algorytmiczny:** Znalezienie wyróżnionego elementu w tablicy

**Dane wejściowe:** Tablica  $\text{tab}[n]$ , gdzie  $n$  to rozmiar tablicy, szuk  $\in R$  - szukany element w tablicy  $\text{tab}$

**Dane wyjściowe:** Informacja o znalezieniu (bądź nie) elementu szuk

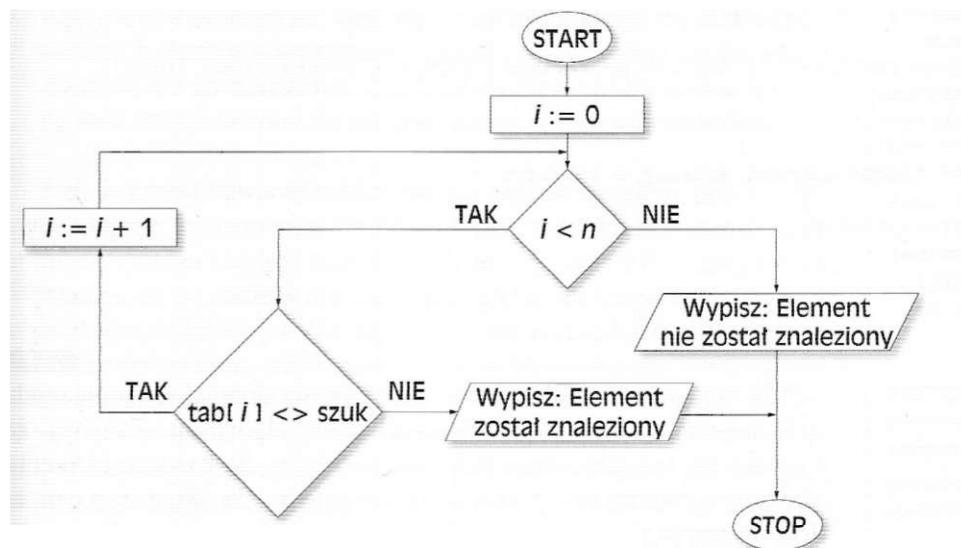
**Zmienne pomocnicze:**  $i \in \{0, \dots, n-1\}$

Zapis za pomocą  
listy kroków

Zapiszmy algorytm w postaci listy kroków:

1. Wczytaj tablicę  $n$ -elementową, pobierz wartość elementu szukanego szuk.
2. Zmiennej pomocniczej  $i$  przypisz wartość 0.
3. Jeśli  $i$  jest mniejsze od  $n$ , przejdź do kroku 4, w przeciwnym wypadku wypisz „Element nie został znaleziony” i zakończ.
4. Jeśli  $\text{tab}[i]$  jest różne od szuk, zwięksź o 1 wartość zmiennej  $i$  i przejdź do kroku 3.
5. Wypisz „Element został znaleziony” i zakończ.

## 5.2. Klasyczne algorytmy działające na tablicach



Ryc. 5.7. Schemat blokowy wyszukiwania w tablicy wyróżnionego elementu

Porównaj schemat blokowy algorytmu (ryc. 5.7) z zapisem w postaci listy kroków, a następnie przeanalizuj program:

```

#include <iostream>
#include <cstdio>
#include <cstdlib>
using namespace std;

void wypełnij(int tab[10])
{
    for (int i=0; i<10; i++)
        tab[i] = rand()%21;
}

void wyświetl(int tab[10])
{
    for (int i=0; i<10; i++)
        cout << tab[i] << " ";
}

void szukaj(int tablica[10], int szukany)
{
    int i = 0;
    while (i<10 && tablica[i]!=szukany)
        i++;
    if (i==10)
        cout << "Element nie zostal znaleziony";
    else
        cout << "Element zostal znaleziony";
}

int main()
{
  
```

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

```
int tab[10];
int szuk;
srand(time(NULL)); // nadanie początkowej wartości generatorowi liczb
wypelnij(tab);
wyswietl(tab);
cout << endl;
cout << "Podaj element szukany w tablicy: ";
cin >> szuk;
szukaj(tab,szuk);
cin.ignore();
getchar();
return 0;
}
```

5.9

Analiza złożoności obliczeniowej algorytmu przeszukiwania liniowego

Policzmy, ile w tym algorytmie wykonuje się operacji dominujących, czyli mających wpływ na szybkość działania algorytmu. Założymy, że w tablicy nie ma szukanego elementu. Wówczas dla każdego elementu tablicy wykonujemy dwa porównania: czyjego indeks świadczy o tym, że element mieści się w zakresie tablicy, oraz czy element jest elementem szukanym. Przeanalizuj kod funkcji, która przeszukuje tablicę w celu znalezienia elementu:

```
while (i<10 && tablica[i]!=szukany)
{
    i++;
}
```

A zatem dla  $n$  elementów wykonujemy w najgorszym wypadku  $2n$  porównań. Złożoność obliczeniowa tego algorytmu jest więc złożonością liniową. Liczbę porównań da się zmniejszyć dzięki zastosowaniu algorytmu przeszukiwania liniowego z wartownikiem.

### 5.2.2. Przeszukiwanie liniowe tablicy jednowymiarowej z wartownikiem

Podczas przeszukiwania tablicy w poprzednim przykładzie musielibyśmy za każdym obiegiem pętli wykonywać dwie czynności: porównywać kolejne elementy tablicy z wartością poszukiwaną i sprawdzać, czy wartość indeksu tablicy nie przekracza dozwolonego zakresu. Poznasz teraz metodę przeszukiwania liniowego tablicy z tak zwanym wartownikiem pozwalającą nam zrezygnować z „pilnowania” wyjścia poza zakres tablicy.

Opis metody przeszukiwania tablicy z wartownikiem

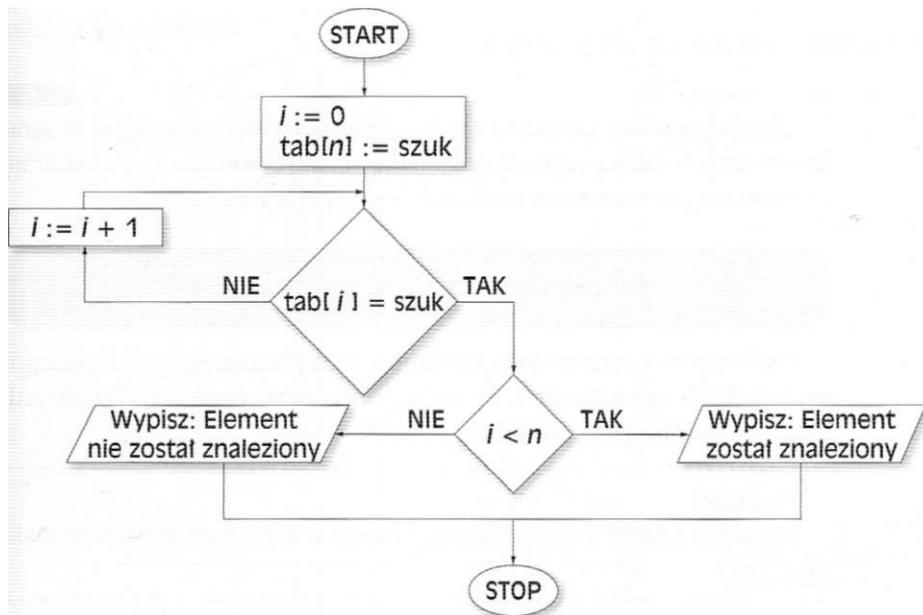
Dodajmy do tablicy dodatkowy element o poszukiwanej wartości i ustawmy go na końcu. Ten właśnie element nazywamy **wartownikiem**. W tak zmodyfikowanej tablicy zawsze znajdziemy element szukany (przecież sami wpisaliśmy go na końcu tablicy), mamy więc pewność, że algorytm skończy swoje działanie, zanim wyjdzie poza zakres tablicy.

## 5.2. Klasyczne algorytmy działające na tablicach

**Aby móc wykorzystać metodę z wartownikiem, musisz użyć tablicy o liczbie elementów o jeden większej niż liczba elementów do sprawdzenia. Dodatkowy element na końcu tablicy jest przeznaczony na umieszczenie w nim wartownika.**

Jeśli w przeszukiwanej tablicy nie ma szukanego elementu, to algorytm znajdzie ten ostatni, dodatkowy. Dzięki takiemu rozwiązaniu algorytm nie musi za każdym razem sprawdzać, czy dotarł do końca tablicy. Wystarczy, że po znalezieniu szukanego elementu sprawdza, czy jest on elementem dodatkowym (to znaczy: czy ma indeks  $n$ ). Jeśli tak, to wprowadzony zostanie napis mówiący, że w tablicy nie ma szukanej wartości (pamiętasz, że wartownik nie jest elementem badanej tablicy). W sytuacji pesymistycznej, gdy elementu szukanego nie było w wejściowej tablicy, algorytm wykona  $n$  operacji porównań oraz jedno dodatkowe - sprawdzi, czy znaleziony element jest wartownikiem. Mimo że zmniejszyliśmy dwukrotnie liczbę operacji, to klasa złożoności tego algorytmu się nie zmieniła i wynosi  $O(n)$ , czyli jest liniowa - podobnie, jak w przeszukiwaniu liniowym.

Na schemacie blokowym z ryciny 5.8 wyraźnie zauważalna jest mniejsza liczba dokonywanych porównań niż w zapisie algorytmu w postaci listy kroków (specyfikacja jak w przykładzie wcześniejszym).



Ryc. 5.8. Schemat blokowy wyszukiwania z wartownikiem wyróżnionego elementu w tablicy

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

Lista kroków algorytmu:

1. Utwórz tablicę o rozmiarze o **1** większym niż wielkość badanej tablicy i wypełnij ją elementami przeszukiwanej tablicy.
2. Wczytaj wartość elementu szukanego szuk.
3. Na końcu (po ostatnim elemencie) przeszukiwanej tablicy dopisz element o wartości szuk.
4. Zmiennej pomocniczej *i* (*indeks tablicy*) przypisz wartość 0.
5. Jeśli *i* [i] = szuk, przejdź do kroku 7.
6. Zwiększ zmienną *z* o 1 i przejdź do kroku 5.
7. Jeśli *i* < *n*, wypisz „Element został znaleziony” i zakończ.
8. Wypisz „Element nie został znaleziony” i zakończ.

Poniżej przedstawiamy samą funkcję przeszukującą tablicę z wartownikiem (reszta programu wyglądałaby analogicznie jak w metodzie przeszukiwania liniowego, bez użycia wartownika):

```
void szukaj_z_wartownikiem(int tablica[11], int szukany)
{
    int i;
    tablica[10] = szukany;
    i = 0;
    while (tablica[i]!=szukany)
        i++;
    if (i<10)
        cout << "Element został znaleziony";
    else
        cout << "Element nie został znaleziony";
}
```

5.10

Chociaż szukamy elementu w tablicy dziesięcioelementowej, w programie tworzymy tablicę jedenastoelementową, gdyż pamiętamy, że ostatnim elementem jest wartownik niebędący elementem badanej tablicy.

### 5.2.3. Poszukiwanie elementu maksymalnego (minimalnego) w tablicy

Problemem poszukiwania elementu maksymalnego (minimalnego) w ciągu liczb zajmowaliśmy się już w poprzednim rozdziale. Wtedy jednak nie znaleźliśmy jeszcze pojęcia tablicy.



#### Przykład

Dana jest tablica *n*-elementowa. Chcemy w niej znaleźć element maksymalny.

W tym celu zapiszemy algorytm w postaci listy kroków, narysujemy schemat blokowy algorytmu oraz zakodujemy metodę poszukiwania elementu maksymalnego.

**Specyfikacja problemu algorytmicznego i opis użytych zmiennych**

**Problem algorytmiczny:** Znalezienie elementu maksymalnego w tablicy

**Dane wejściowe:** Tablica  $\text{tab}[n]$ , gdzie  $\text{tab}[i] \in R$ ;  $n$  to rozmiar tablicy,  $n > 0$

**Dane wyjściowe:**  $\max$  - element maksymalny tablicy

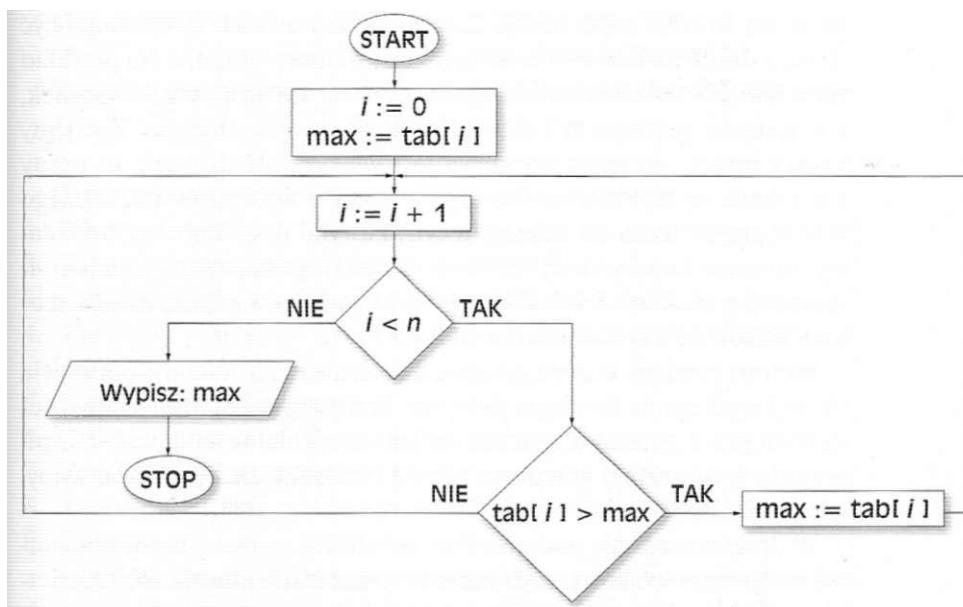
**Zmienne pomocnicze:**  $i \in N$  - zmienna licznikowa

Różnica pomiędzy postępowaniem w przypadku szukania maksimum w ciągu liczb podawanych z zewnątrz a poszukiwaniem maksimum w tablicy polega w zasadzie tylko na tym, że poprzednio liczby pobierane były na bieżąco, teraz zaś zostały zapisane jako kolejne elementy tablicy.

Oto zapis algorytmu za pomocą listy kroków:

1. Zmiennej pomocniczej  $i$  przypisz 0.
2. Zmiennej  $\max$  przypisz wartość  $\text{tab}[i]$ .
3. Zwiększ  $i$  o 1.
4. Jeśli  $i < n$ , to przejdź do kroku 6.
5. Wypisz  $\max$  i zakończ.
6. Jeśli  $\text{tab}[i] > \max$ , zmiennej  $\max$  przypisz wartość  $\text{tab}[i]$ .
7. Przejdz do kroku 3.

Pozostało nam już tylko zapisanie algorytmu za pomocą schematu blokowego (ryc. 5.9), a następnie zakodowanie omówionej metody w języku programowania.



**Ryc. 5.9.** Schemat blokowy wyszukiwania największego elementu w tablicy

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

Oto funkcja, która realizuje znajdowanie największego elementu w tablicy:

```
int maksymalny(int tab[10])
{
    int max = tab[0];
    for (int i=1; i<10; i++)
        if (tab[i]>max) max = tab[i];
    return max;
}
```

Najpierw za element maksymalny przyjmujemy wartość pierwszego elementu tablicy. Następnie iteracyjnie sprawdzamy, czy kolejny element tablicy jest większy od aktualnego max - jeśli tak, ustalamy go jako nowy element maksymalny. Wynikiem działania funkcji jest maksymalny element.

### 5.2.4. Zapisywanie liczb w różnych systemach liczbowych

Wiesz już, że każda informacja zapisana w komputerze jest oparta na dwójkowym systemie liczenia. Teraz nauczysz się, jak liczbę z postaci dziesiętnej zamienić na liczbę w systemie dwójkowym, trójkowym, szesnastkowym i na odwrót.

Postać liczby  
w danym  
systemie liczenia

Liczba dziesiętna to inaczej liczba o podstawie liczenia dziesięć, liczbą dwójkową jest liczba o podstawie liczenia dwa, a szesnastkowa postać liczby to liczba o podstawie liczenia szesnaście.

Przypomnijmy, że podstawa danego systemu informuje, z jakich cyfr może się składać zapis liczby. Zawsze są to wartości, które mogą być resztą z dzielenia liczby całkowitej przez podstawę systemu. Na przykład zapis liczby w systemie dwójkowym składa się wyłącznie z zer i jedynek, a w systemie piątkowym z cyfr od zera do czwórki włącznie. Zwróćmy jednak uwagę, że jeżeli podstawa jest większa niż dziesięć, to reszty z dzielenia są liczbami zarówno jedno-, jak i dwucyfrowymi. Dlatego w tych przypadkach do dziesięciu cyfr, którymi dysponujemy, dokładamy umownie kolejne litery alfabetu. Dziesiątkę oznaczamy znakiem<sup>^1</sup>, jedenastkę znakiem *B* itd. Dlatego liczba podana w zapisie szesnastkowym składa się z cyfr oraz liter od *A* do *F*.

Metoda zamiany liczby z systemu dziesiętnego na system o podstawie równej *n* polega na kolejnym dzieleniu liczby zapisanej w systemie dziesiętnym przez podstawę systemu, na jaki zamieniamy daną liczbę, i spisywaniu w odwrotnej kolejności reszt z dzielenia. Dzielenie kończymy, gdy wynik dzielenia jest zerem.

W drugim rozdziale podręcznika omówiliśmy zamianę liczby dziesiętej na system dwójkowy, czyli binarny. Teraz zamienimy liczbę dziesiętną na liczbę w innym systemie. Sposób jest analogiczny.

Zamienimy liczbę 47 z zapisu w systemie dziesiętnym na zapis w systemie o podstawie trzy:

$$47 : 3 = 15, \text{ reszta: } 2$$

$$15 : 3 = 5, \text{ reszta: } 0$$

$$5 : 3 = 1, \text{ reszta: } 2$$

$$1 : 3 = 0, \text{ reszta: } 1$$

A zatem liczba 47 w systemie trójkowym ma postać 1202.

### Przykład

Dana jest liczba dziesiętna. Zamienimy ją na liczbę zapisaną w systemie o podstawie liczenia równej  $n$ , gdzie  $n \in \{2, 9\}$ .



Skonstruujemy algorytm, który będzie zamieniał liczbę z systemu dziesiętnego na system dwójkowy. Potem odpowiednio go zmodyfikujemy, tak aby zamieniać daną liczbę na liczbę w zapisie o podstawie liczenia ze zbioru  $\{3, \dots, 9\}$ . Jednym ze sposobów rozwiązania tego zadania jest przechowanie reszt z dzielenia w przeznaczonej do tego tablicy, a następnie wypisanie ich na ekranie.

### Specyfikacja problemu algorytmicznego i opis użytych zmiennych

**Problem algorytmiczny:** Zamiana liczby dziesiętnej na liczbę w systemie o podstawie liczenia równej 2

**Dane wejściowe:** Liczba  $x \in C$ , zapisana w systemie dziesiętnym

**Dane wyjściowe:** Napis przedstawiający liczbę  $x$  zapisaną w systemie o podstawie liczenia 2

**Zmienne pomocnicze:** Tablica liczb całkowitych przechowująca cyfry wynikowej liczby

Zapiszmy na początku algorytm za pomocą listy kroków:

1. Wczytaj liczbę  $x$ .
2. Zmiennej pomocniczej  $i$  przypisz wartość 0.
3. Zmiennej tab [ i ] przypisz resztę z dzielenia  $x$  przez 2.
4. Zmiennej  $x$  przypisz wynik dzielenia w zbiorze liczb całkowitych  $x$  przez 2.
5. Zwiększ wartość zmiennej  $i$ .
6. Jeśli  $x$  jest różne od 0, wróć do kroku 3.
7. Zmniejsz  $i$  o 1 oraz wypisz tab [ i ].
8. Jeśli  $i$  jest większe od 0, to przejdź do kroku 7.
9. Zakończ.

Algorytm ten nie jest skomplikowany, ale pojawia się w nim problem dotyczący tablicy, w której będziemy przechowywać reszty z dzielenia, by

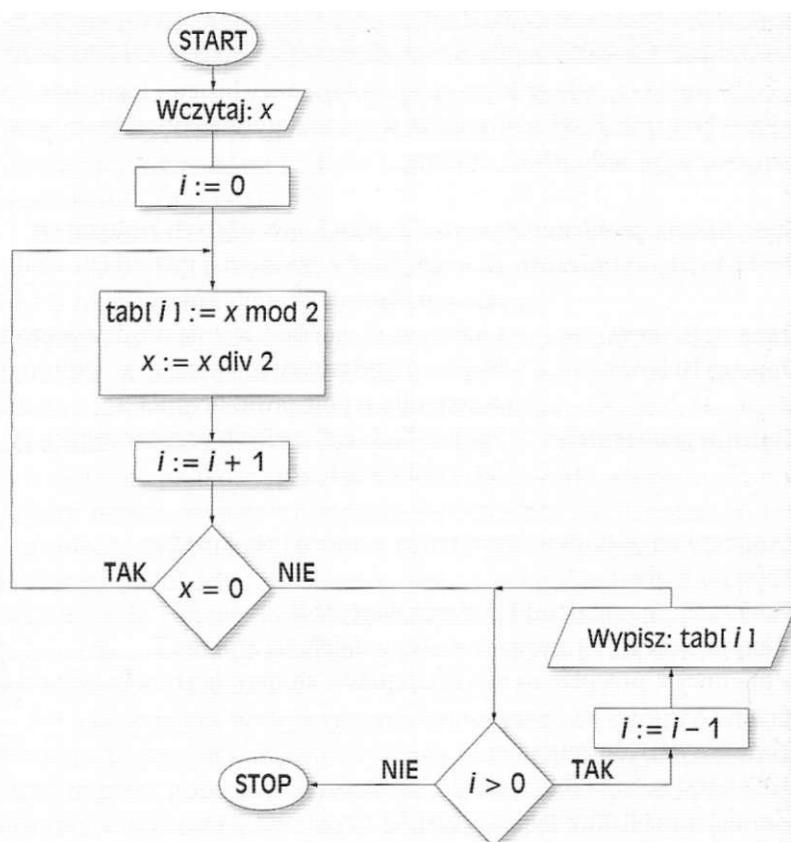
## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

je następnie wypisać. Problem polega na tym, że musimy zdefiniować tablicę, nie znając liczby jej elementów, zależy to bowiem od tego, ile reszt z dzielenia wpiszemy do tablicy, zanim wynik dzielenia będzie zerem. Efektem jest ograniczenie wartości liczby, którą zamieniamy na postać dwójkową. Założymy, że największą liczbą, jaką chcemy zmienić, będzie 255, wówczas wystarczy nam tablica ośmioelementowa.

Jeśli dla zadanego rozmiaru tablicy  $n$  chcesz sprawdzić, jak dużą liczbę dziesiętną  $L$  możesz zapisać w systemie o podstawie  $k$ , wystarczy skorzystać ze wzoru:

$$L = k^n - 1$$

Zakodujmy omówioną metodę i porównajmy działanie algorytmu ze schematem blokowym (ryc. 5.10):



Ryc. 5.10. Schemat blokowy algorytmu zamiany liczby zapisanej w systemie dziesiętnym na system dwójkowy

## 5.2. Klasyczne algorytmy działające na tablicach

```
void zamiana_na_postac_dwojkowa(int x)
{
    int t[8], i = 0;
    do
    {
        t[i] = x%2;
        x = x/2;
        i++;
    }
    while (x!=0)
    // tablica została wypełniona resztami
    // z dzielenia liczby x przez 2
    // teraz wartości wpisane do tablicy
    // zostaną wypisane w odwrotnej kolejności
    while (i>0)
    {
        i--;
        cout << t[i] << " ";
    }
}
```

5.11

Zdefiniowaliśmy funkcję zamieniającą podstawy zapisu liczb, jako zadanie dla ciebie pozostawiamy użycie jej w programie.

Jeśli chcemy otrzymać funkcję uniwersalną, która zamienia liczbę z postaci dziesiętnej nie tylko na postać dwójkową, ale i na trójkową, czwórkową aż do postaci dziewiątkowej włącznie, wystarczy poddać powyższą funkcję tylko drobnej modyfikacji. Do parametrów pobieranych przez funkcję dodajemy podstawę systemu liczenia, a następnie tam, gdzie we wnętrzu pętli pojawia się dzielenie przez dwa i obliczanie reszty z dzielenia przez dwa, wprowadzamy zmienną skojarzoną z podstawą liczenia:

```
void zamiana(int x, int podstawa)
{
    int t[8], i = 0;
    do
    {
        t[i] = x%podstawa;
        x = x/podstawa;
        i++;
    }
    while (x!=0)
    while (i>0)
    {
        cout << t[i] << " ";
        i--;
    }
}
```

5.12

Teraz zastanówmy się, co zmienić w funkcji, aby można ją było stosować do zamiany liczby z postaci dziesiętnej na taką postać, w której liczby zapisywane są cyframi bądź cyframi i literami.

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

Wiesz, że zmienne, które przechowują litery, muszą być zadeklarowane jako typ `char`. Czy koniecznie trzeba zdefiniować tablicę typu `char`? Jeśli reszta z dzielenia w programie wynosiłaby 10, to można wpisać do tablicy znak „A”, jeśli zaś 11, wówczas znak „B” itd. Lepiej jednak pozostać przy zdefiniowaniu tablicy jako liczbowej i do niej wpisywać dwucyfrowe reszty z dzielenia, a dopiero przy wypisywaniu wymieniać na litery - odpowiednie wartości.

Teraz zajmiemy się krótko problemem odwrotnym: jak zamienić liczbę zapisaną w systemie o podanej podstawie na liczbę w systemie dziesiętnym.

Każdą liczbę naturalną (mówimy tu o systemie dziesiętnym) możesz przedstawić jako sumę kolejnych potęg liczby 10. Dla przykładu:  
 $234 = 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$ .

Zauważ, że cyfry na poszczególnych pozycjach mówią nam o tym, ile w liczbie znajduje się potęg dziesiątki (tu dziesiątka jest podstawą liczenia) o wykładniku będącym numerem pozycji cyfry, licząc od prawej strony i rozpoczynając liczenie od zera.

Zamiana liczby zapisanej w dowolnym systemie liczenia na system dziesiętny

Każdą liczbę zapisaną w dowolnym systemie zamienimy na system dziesiętny, za pomocą tej samej metody, tylko że cyfry informują nas wówczas o ilości kolejnych potęg podstawy liczenia, a nie liczby 10, jak powyżej. Przeliczmy więc kilka liczb z danych systemów na dziesiątkowy (dolny indeks liczby informuje nas, w jakim systemie została ona zapisana):

$$101101_{(2)} = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 45$$

$$210_{(3)} = 2 \cdot 3^2 + 1 \cdot 3^1 + 0 \cdot 3^0 = 21$$

$$1A2_{(16)} = 1 \cdot 16^2 + 10 \cdot 16^1 + 2 \cdot 16^0 = 418$$

Żeby napisać program, który będzie przeliczał liczbę z dowolnego systemu na dziesiątkowy, musimy najpierw podać podstawę liczenia, a następnie kolejne cyfry zamienianej liczby. Kolejne cyfry wprowadzamy do uprzednio przygotowanej tablicy. Potem tworzymy sumę zgodnie z przykładami obliczeniowymi przedstawionymi powyżej i wyświetlamy ją.

Poniżej znajduje się program, który zamienia liczbę z postaci binarnej na postać dziesiętną:

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

int zamien_na_dziesietna(int tab[], int cyfry)
{
    int i, j, k;
    int liczba_dziesietna = 0;
    for (j=0; j<cyfry; j++)
    {
        i = 1;
```

```

        for (k=j+1; k<cyfry; k++)
            i = i*2;
        liczba_dziesietna = liczba_dziesietna+tab[j]*i;
    }
    return liczba_dziesietna;
}

int main()
{
    int binarna[20];
    int i, ilosc_cyfr;
    cout << "Z ilu cyfr bedzie sie skladac liczba?: ";
    cin >> ilosc_cyfr;
    cout << "Podaj liczbe w postaci binarnej; po kazdej cyfrze nacisnij Enter \n";
    for (i=0; i<ilosc_cyfr; i++)
        cin >> binarna[i];
    cout << "W systemie dziesietnym to: " << zamien_na_dziesietna(binarna,ilosc_cyfr);
    cin.ignore();
    getchar();
    return 0;
}

```

5.13

Założyliśmy, że maksymalnie możemy w programie podać liczbę binarną dwudziestocyfrową. Jeżeli chcielibyśmy zamienić na postać dziesiętną nie tylko liczbę binarną, lecz każdą o razej podstawie liczenia (gdzie  $n$  jest liczbą całkowitą większą od dwóch), wówczas funkcja zamieniająca miałaby postać:

```

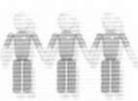
int zamien_na_dziesietna(int tab[], int cyfry, int podstawa)
{
    int i, j, k;
    int liczba_dziesietna = 0;
    for (j=0; j<cyfry; j++)
    {
        i = 1;
        for (k=j+1; k<cyfry; k++)
            i = i*podstawa;
        liczba_dziesietna = liczba_dziesietna+tab[j]*i;
    }
    return liczba_dziesietna;
}

```

5.14

Oczywiście, zastosowanie tak zmodyfikowanej funkcji wymagałoby dokonania drobnych zmian w kodzie głównej funkcji. Najpierw spróbuj wykonać to zadanie samodzielnie, a potem możesz sprawdzić swoje rozwiązanie z kodem pełnego programu znajdującym się na płycie CD dołączonej do podręcznika (5.15).

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach



Ponownie wracamy do problemu badania, czy dana liczba jest pierwsza. Zapewne pamiętasz, że liczba pierwsza to taka liczba naturalna różna od 1, która nie ma innych naturalnych dzielników prócz 1 i samej siebie. Prześledźmy ten problem na przykładzie:

### Przykład

Dany jest przedział liczbowy  $<2, b>$ . Znajdziemy i wypiszemy wszystkie liczby pierwsze należące do tego przedziału.

Bazując na zdobytej wcześniej wiedzy, moglibyśmy badać kolejno każdą liczbę całkowitą z zadanego przedziału, czy jest liczbą pierwszą czy złożoną, ale takie postępowanie jest mało optymalne. Jeśli bowiem badanym przedziałem jest na przykład  $(4, 30)$ , to nie ma sensu sprawdzać żadnej z całkowitych wielokrotności liczby 4, ponieważ będą się one dzieliły przez 4. Możemy więc od razu wyrzucić z badanego przedziału liczby 8, 12, 16, 20, 24, 28 (4 też wyrzucimy, gdyż jest liczbą złożoną). Tak samo odrzucimy wszystkie całkowite wielokrotności liczb 3, 5 oraz każdej następnej liczby jeszcze niewyryzuonej z przedziału. Taka metoda postępowania nosi nazwę sita Eratostenesa, od skojarzenia, że z przedziału „odsiewamy” wszystkie liczby złożone, a zatem pozostawiamy w nim tylko liczby pierwsze. Twórcą metody jest Eratostenes z Cyreny (ok. 276 r. p.n.e.-ok. 194 r. p.n.e.) - grecki matematyk, astronom, filozof, geograf i poeta.

Przykład  
zastosowania  
metody  
odsiewania liczb

Załóżmy, że chcemy znaleźć wszystkie liczby pierwsze z przedziału od 2 do 20. Będziemy ich szukać wśród liczb ustawionych w ciągu:

**2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20**

Liczba 2 jest liczbą pierwszą, więc ją pozostawiamy, ale pozbywamy się jej wszystkich wielokrotności. Pozostaje więc ciąg:

**2 3 5 7 9 11 13 15 17 19**

Kolejną liczbą znajdująca się w ciągu jest 3 - musi być ona liczbą pierwszą, gdyż inaczej dzieliłaby się przez 2 i zostałaby wykreślona już w poprzednim kroku. Zatem liczbę 3 pozostawiamy, ale wykreślamy wszystkie jej całkowite wielokrotności. Oto ciąg wynikowy:

**2 3 5 7 11 13 17 19**

Zauważ, że część tych wielokrotności była już wcześniej wykreślona jako liczby, które oprócz podzielności przez 3 dzieliły się również przez 2. Następną niewykreślzoną liczbą jest 5, która z tych samych względów co powyżej musi być liczbą pierwszą (na pewno nie dzieli się przez 2, bo już by jej w ciągu nie było, na pewno nie dzieli się przez 3, ponieważ zostałaby również wykreślona, nie przez 4, bo wówczas zostałaby wykreślona jako podzielna przez 2 - dzieli się tylko przez 1 i przez samą siebie - jest więc liczbą pierwszą).

Teraz powinniśmy wykreślić wielokrotności liczby 5, ale już ich w tablicy nie ma. Czy musimy dojść aż do górnej granicy badanego przedziału, aby mieć pewność, że pozostały tylko liczby pierwsze? Odpowiedź na to pytanie znasz z przykładu w rozdziale czwartym: wystarczy badać liczby pierwsze nie większe od  $\sqrt{N}$ , gdzie N jest prawym krańcem badanego przedziału. W naszym przykładzie prawym krańcem, czyli największą liczbą w przedziale, jest 20. Jej pierwiastek to około 4,4. A zatem liczbą, do której dojdziemy i będziemy wykreślać całkowite jej wielokrotności, będzie liczba 4. Tylko że do niej w zasadzie też nie doszliśmy, gdyż została ona już wcześniej skreślona jako wielokrotność dwójki.

Poniżej prezentujemy pełny kod programu, który wypisuje wszystkie liczby pierwsze mniejsze od podanej z zewnątrz lub jej równe. W naszym programie podana liczba musi być mniejsza od 100, gdyż na tyle pozwala nam rozmiar zadeklarowanej tablicy.

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

int main()
{
    int i, j, zakres, b;
    bool tablica[100];
    cout << "Podaj gorny zakres, max 99" << endl;
    cin >> zakres;
    b = sqrt((float)zakres);
    for (i=2; i<zakres+1; i++)
        tablica[i] = true; // (1)
    for (i=2; i<=b; i++)
        if (tablica[i]!=false)
            for (j=i+i; j<zakres+1; j=j+i)
                tablica[j] = false; // (2)
    cout << "Liczby pierwsze z zakresu od 1 do " << zakres << " to:" << endl;
    for (i=2; i<zakres+1; i++)
        if (tablica[i]!=false) // (3)
            cout << i << " ";
    cin.ignore();
    getchar();
    return 0;
}
```

5.16

W linii opatrzonej komentarzem (1) wypełniliśmy fragment tablicy wartością logiczną `true`. W linii oznaczonej (2) wszystkie elementy tablicy, których indeksy są liczbami złożonymi, zostają zamienione na `false`. Na ekran wyprowadziliśmy tylko indeksy elementów, które przechowują wartość logiczną `true`, nie zostały zamienione na `false`, a więc są liczbami pierwszymi.

### 5.3. Sortowanie tablicy

Ważną czynnością związaną z tablicą jest sortowanie umieszczonych w niej danych, ponieważ nieporównywalnie łatwiej operuje się na danych, które są posortowane, czyli ustawione w zadanym porządku (wyobraź sobie np. korzystanie ze słownika, w którym hasła nie zostały ułożone w kolejności alfabetycznej). Bez względu na to, jakiego typu są elementy tablicy, problem i tak zawsze sprowadza się do sortowania liczb. Jeśli sortujemy tablicę przechowującą nazwiska, to każdej literze przyporządkowany jest liczbowy kod ASCII (omawiany dokładniej w dalszej części rozdziału), zatem sortując litery, sortujemy związane z nimi liczby. W wypadku, gdy pierwsze litery nazwiska są takie same, sortujemy według drugiej litery itd.

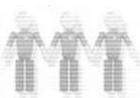
#### Definicja

Sortowaniem nazywamy ustawianie elementów ciągu w ustalonym porządku według zadanego kryterium (klucza).

Możemy na przykład sortować nazwiska według alfabetu od A do Z lub na odwrót: od Z do A. Można posortować liczby od najmniejszej do największej lub na odwrót. W naszych algorytmach będziemy omawiać sortowanie liczb w celu ustawienia ich od najmniejszej do największej, czyli w porządku niemalejącym. Zakładamy, że tablice, które sortujemy, mają co najmniej dwa elementy.

#### 5.3.1. Sortowanie bąbelkowe

Sortowanie bąbelkowe (przez prostą zamianę, ang. *bubblesort*) jest jedną z najprostszych metod sortowania, zarówno ze względu na schemat działania metody, jak i jej implementację. Założymy, że chcemy posortować tablicę liczbową w porządku od najmniejszego elementu do największego.



#### Przykład

Dana jest n-elementowa tablica liczb całkowitych. Elementy tablicy ustawimy w porządku niemalejącym.

#### Specyfikacja problemu algorytmicznego i użytych zmiennych

**Problem algorytmiczny:** Ustawienie elementów tablicy w porządku niemalejącym

**Dane wejściowe:** Tablica  $\text{tab}[n]$ , gdzie  $\text{tab}[i] \in \mathbb{C}$ ;  
 $i \in \{0, n - 1\}$ ,  $n > 1$

**Dane wyjściowe:** Posortowana tablica

**Zmienne pomocnicze:**  $i, j \in N$  - zmienne licznikowe,  $temp \in C$   
- zmienna pomocnicza przy zamianie elementów pomiędzy sobą

Sortowanie rozpoczynamy od pierwszego elementu tablicy. Bieżący element porównujemy z elementem bezpośrednio po nim następującym. Jeśli kolejny element jest mniejszy od poprzedniego, to zamieniamy je miejscami. Jeśli bieżący element nie jest przedostatnim elementem zbioru, to przesuwamy się do kolejnego elementu i kontynuujemy porównywanie. Zanim przejdziesz do analizy zapisu algorytmu, prześledź kolejne kroki na rycinie przedstawiającej graficznie tę metodę dla tablicy pięcioelementowej.

Przejście 1. Porównujemy elementy **pierwszy z drugim**. Ponieważ większy znajduje się przed mniejszym, zamieniamy je miejscami. Teraz porównujemy element **drugi z trzecim**. Są ustawione w nieodpowiedniej kolejności, a więc je również zamieniamy miejscami. Kolejno porównujemy elementy **trzeci z czwartym**. Nie zamieniamy ich, gdyż mniejszy znajduje się przed większym. Porównujemy **czwarty z piątym** elementem i dokonujemy zamiany. Po dokonaniu czterech porównań i odpowiednich zamian (w zależności od danych w tablicy) mamy pewność, że największy element tablicy znajduje się już na końcu.

Przejście 2. Teraz rozpoczynamy wszystko od początku, tyle że nie musimy już przeходить całej tablicy - ostatni element stanowi jej posortowaną część.

Przechodząc przez tablicę drugi raz, tylko trzykrotnie porównamy elementy i ewentualnie zamienimy je pomiędzy sobą. Po drugim przejściu dwa ostatnie elementy tablicy stanowią jej posortowaną część.

Przejście 3. Trzeci przebieg pętli zewnętrznej to już tylko dwa przebiegi pętli wewnętrznej. Trzy ostatnie elementy znajdują się bowiem na swoich miejscach.

Przejście 4. Pętla zewnętrzna rusza po raz czwarty - a w pętli wewnętrznej wykonujemy tylko jeden przebieg - zostają porównane pierwsze dwa elementy i w razie potrzeby przedstawione, tak jak w naszym przykładzie. Tablica została posortowana.

Opis metody sortowania bąbelkowego

Przejście 1
7 3 2 9 1
3 7 2 9 1
3 2 7 9 1
3 2 7 9 1
3 2 7 1 9

Przejście 2
3 2 7 1 9
2 3 7 1 9
2 3 7 1 9
2 3 1 7 9

Przejście 3
2 3 1 7 9
2 3 1 7 9
2 1 3 7 9

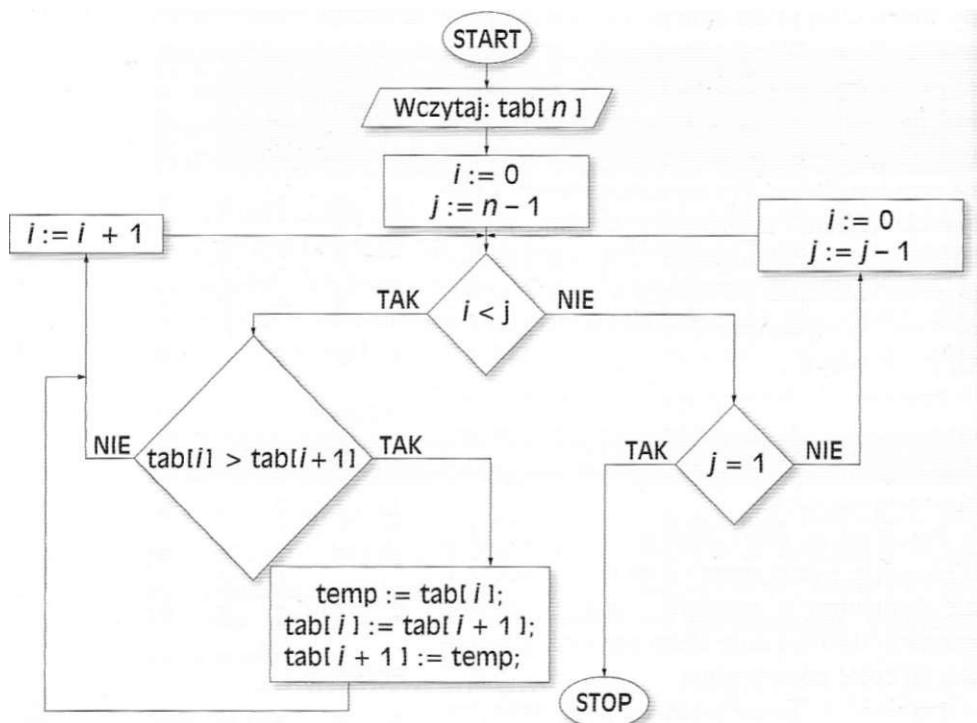
Przejście 4
2 1 3 7 9
1 2 3 7 9

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

Zapiszmy algorytm za pomocą listy kroków:

1. Zmiennej  $j$  przypisz wartość  $n - 1$ .
2. Zmiennej  $i$  przypisz wartość 0.
3. Jeśli  $\text{tab}[i + 1] < \text{tab}[i]$ , to zamień je miejscami.
4. Zwiększ wartość zmiennej  $z o 1$ .
5. Jeśli  $i$  jest mniejsze niż  $j$ , to wróć do kroku 3.
6. Jeśli  $j$  jest równe 1, to zakończ.
7. Zmniejsz wartość zmiennej pomocniczej  $j o 1$  i wróć do kroku 2.

Porównaj zapis algorytmu zarówno w postaci listy kroków, jak i schematu blokowego (ryc. 5.11) z przykładem, który omówiliśmy. Zwróć uwagę, że zmienną  $j$  została wprowadzona w celu uniknięcia dokonywania porównań w już posortowanej części tablicy.



Ryc. 5.11. Schemat blokowy algorytmu sortowania bąbelkowego

W funkcji realizującej metodę sortowania bąbelkowego argumentem przekazywanym do funkcji jest tablica przeznaczona do posortowania oraz jej rozmiar. Pamiętaj, że zmiany dokonane na tablicy wewnętrz funkcji znajdują zachowane po zakończeniu jej działania. Dla czytelności kodu zdefiniowaliśmy osobno funkcję, której zadaniem jest zamiana dwóch elementów miejscami. Argumenty przekazaliśmy funkcji zamień przez referencję, aby zamiana elementów została zachowana po wyjściu z funkcji.

Od momentu wywołania funkcji `sortowanie_babelkowe` operujemy tablicą już posortowaną.

```
void zamien(int &a, int &b)
{
    int temp = a;          // zamiana elementów
    a = b;                // a z b
    b = temp;              // z użyciem zmiennej pomocniczej temp
}

void sortowanie_babelkowe(int tab[], int n)
{
    int temp;
    for (int j=n-1; j>0; j--)           // (1)
    {
        for (int i=0; i<j; i++)         // (2)
            if (tab[i]>tab[i+1])
                zamien(tab[i],tab[i+1]);
    }
}
```

5.17

Przypominamy, że nie można zamienić wartości pomiędzy zmiennymi  $a$ ,  $b$  za pomocą operacji:  $a = b$ ;  $b = a$ . Dlatego wprowadziliśmy zmienią pomocniczą, aby przechowywała początkową wartość zmiennej  $a$ .

Przeanalizujmy na koniec złożoność obliczeniową algorytmu sortowania bąbelkowego. Założymy w tym celu, że tablica, którą chcemy posortować, ma  $n$  elementów. A zatem będzie  $n - 1$  przebiegów pętli zewnętrznej. Odpowiednio w pętlach wewnętrznych będzie kolejno:  $n - 1$ ,  $n - 2$ , 2 porównań i maksymalnie tyle samo zamian. Samych tylko porównań będzie:  $(n - 1) + (n - 2) + \dots + 2$ . Korzystając ze wzoru na sumę elementów ciągu arytmetycznego, możemy policzyć, że liczba porównań wynosi

$$\frac{n^2 - n}{2}$$

Sortowanie bąbelkowe jest więc metodą o złożoności obliczeniowej  $O(n^2)$ .

Złożoność obliczeniowa sortowania bąbelkowego

### 5.3.2. Sortowanie przez wstawianie

Sortowanie przez wstawianie (ang. *inseitionsort*) to jeden z prostszych i bardziej znanych algorytmów. Postępowanie przy tym sortowaniu przypomina zachowanie gracza, który układają w ręku karty podnoszone ze stołu, wstawiając je w odpowiednie miejsca, tak aby trzymane karty już były ustalone według zadanej kolejności.

Zacznijmy od przykładu. Założymy, że mamy dane liczby, które będziemy kolejno pobierać od lewej strony i tworzyć z nich posortowany ciąg:

7    3    0    1    5

Bierzemy liczbę 7 i wstawiamy ją jako pierwszy element tablicy, gdyż jest to na razie najmniejsza z liczb (jako jedyna):

3    0    1    5              7

[operan.pl](http://operan.pl)

Opis metody sortowania przez wstawianie na podstawie przykładu

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

Kolejną pobraną liczbą jest liczba 3 i teraz ona zajmie pierwszą pozycję w tablicy jako mniejsza od liczby 7, a 7 zostaje przeniesione na pozycję drugą:

**0    1    5              3    7**

Teraz pobieramy 0. Jest ono mniejsze od dwóch liczb już znajdujących się w tablicy, aby więc znalazło się na pierwszej pozycji, najpierw musimy przesunąć obie liczby tablicy na kolejne pozycje.

**1    5              0    3    7**

Pobieramy liczbę 1 i sprawdzamy, w którym miejscu powinna się ona znaleźć. Jest większa od 0, a mniejsza od 3, powinna zatem zostać umieszczona pomiędzy nimi. W tym celu 3 i 7 znów przesuwamy na dalsze pozycje tablicy, zwalniając w ten sposób miejsce, które zajmie 1:

**5              0    1    3    7**

Przeglądamy tablicę od pierwszego elementu i szukamy miejsca, w którym powinna się znaleźć ostatnia z liczb do posortowania. Porównując liczby już umieszczone w tablicy z liczbą 5, która jeszcze czeka na umieszczenie, znajdujemy dla niej miejsce - będzie to czwarty element tablicy, zatem 5 znajdzie się za mniejszym elementem, a przed elementem większym, 7 zostanie przesunięte na następne miejsce w tablicy:

**0    1    3    5    7**

Otrzymaliśmy ciąg posortowany metodą wstawiania. Jak już wspomnieliśmy, jest to metoda prosta i łatwa do zrozumienia. Implementowana za pomocą tablicy, wymaga jednak wielu przestawień elementów, gdyż pozyskiwanie miejsca w tablicy na umieszczenie jakiegoś elementu wiąże się z przesunięciem wszystkich pozostałych, począwszy od pierwszego elementu większego od tego, który chcemy wstawić.

Przejdźmy do zapisu i kodowania metody sortowania przez wstawianie.

### Specyfikacja problemu algorytmicznego i opis użytych zmiennych

**Problem algorytmiczny:** Ustawienie elementów tablicy w porządku niemalejącym

**Dane wejściowe:** Tablica  $\text{tab}[n]$ , gdzie  $\text{tab}[i] \in C$ ,  $i \in \{0, n - 1\}, n > 1$

**Dane wyjściowe:** Posortowana tablica

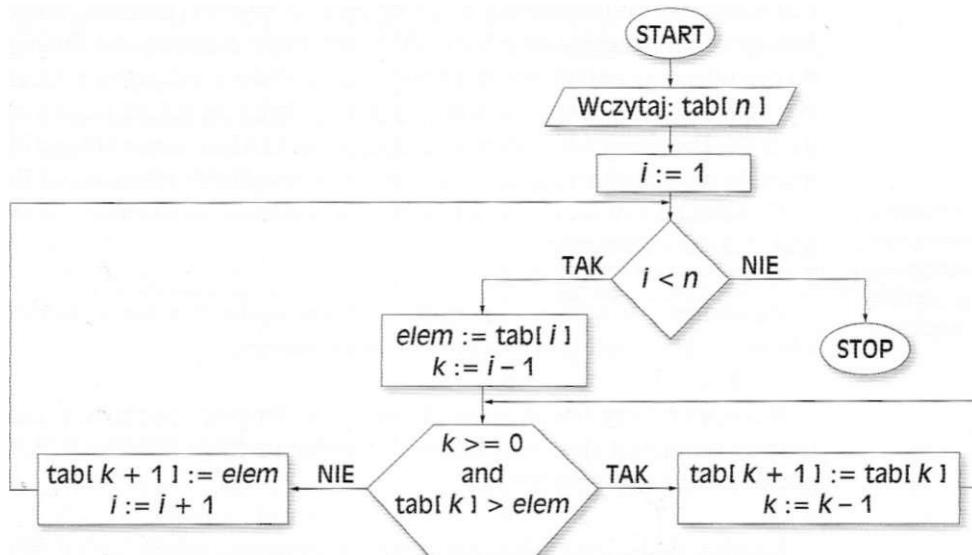
**Zmienne pomocnicze:**  $i, k \in N$  - zmienne licznikowe,  $\text{elem} \in C$  - zmienność pomocnicza, element aktualnie ustawiany w sortowanej tablicy

Lista kroków algorytmu sortującego:

1. Wczytaj tablicę «-elementową  $\text{tab}[n]$ .
2. Zmiennej  $i$  przypisz 1.
3. Jeżeli  $i$  nie jest mniejsze od liczby elementów w tablicy, to zakończ.

4. Wartość  $\text{tab}[i]$  zapamiętaj w zmiennej pomocniczej  $\text{elem}$ , pod zmienną  $k$  podstaw  $(i - 1)$ .
5. Jeżeli  $k$  jest większe lub równe 0 i  $\text{tab}[k]$  jest większe od  $\text{elem}$ , to przejdź do kroku 6, w przeciwnym wypadku przejdź do kroku 7.
6. Wartość  $\text{tab}[k]$  przesuń o jedno miejsce w prawo w tablicy ( $\text{tab}[k + 1] := \text{tab}[k]$ ), zmniejsz  $k$  o 1, przejdź do kroku 5.
7. Wstaw wartość zmiennej  $\text{elem}$  w znalezione miejsce ( $\text{tab}[k + 1] := \text{elem}$ ), zwiększ  $i$  o 1, przejdź do kroku 3.

Zapiszmy algorytm za pomocą schematu blokowego (ryc. 5.12):



Ryc. 5.12. Schemat blokowy algorytmu sortowania przez wstawianie

Zobaczmy na koniec, jak wygląda funkcja, której jako pierwszy argument podajemy tablicę do posortowania. Drugim argumentem jest rozmiar tablicy:

```

void sortowanie_przez_wstawianie(int tab[], int n)
{
    int i, k, elem;
    for (i=1; i<n; i++)
    {
        elem = tab[i];
        k = i-1;
        while (k>=0 && tab[k]>elem)
        {
            tab[k+1] = tab[k];
            k--;
        }
        tab[k+1] = elem;
    }
}

```

5.18

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

Przeanalizuj i policz podstawowe operacje algorytmu sortowania przez wstawianie. W notacji  $O$  sortowanie to ma złożoność kwadratową  $O(n^2)$ .

### 5.3.3. Sortowanie przez wybór

Kolejny typ sortowania, o tej samej złożoności obliczeniowej co dwa poprzednie, to sortowanie przez wybór, znane również pod nazwą sortowania przez selekcję (ang. *selectionsort*). Idea tej metody jest dość prosta: w tablicy znajdujemy najmniejszy element i zamieniamy go miejscem z pierwszym elementem tablicy. Jeśli najmniejszy element występuje wielokrotnie, to będziemy braли pod uwagę ten, który znajduje się najbliżej początku tablicy. Następnie we fragmencie tablicy, który obejmuje elementy od drugiego do ostatniego, poszukujemy minimum i zamieniamy je miejscem z drugim elementem tablicy. Dwa pierwsze elementy tablicy to już jej posortowana część. Będziemy tak postępować, dodając za każdym razem kolejny element do posortowanej części, aż dojdziemy do ostatniego elementu tablicy.

Opis metody sortowania przez wybór na podstawie przykładu

Przeanalizujmy przykład, na podstawie którego omawialiśmy sortowanie przez wstawianie:

7    3    0    1    5

Znajdujemy w tablicy minimum, jest nim liczba 0, i zamieniamy ją z liczbą 7, która jest nierwesząm elementem tablicy.

0    3    7    1    5

W części tablicy obejmującej elementy od drugiego poczawszy, znajdujemy minimum (jest nim liczba 1) i zamieniamy ją z liczbą 3, która znajduje się na drugiej pozycji.

0    1    7    3    5

Kolejne znalezione minimum w nieposortowanej jeszcze części tablicy to liczba 3 - zamieniamy ją z liczbą 7.

0    1    3    7    5

Pozostała nam do posortowania część tablicy, która składa się z jej dwóch ostatnich elementów. Minimum tej części to liczba 5. Zamieniamy ją więc miejscem ze znajdującą się przed nią siódemką.

0    1    3    5    7

Mamy pewność, że na ostatniej pozycji stoi liczba nie mniejsza od wszystkich pozostałych, gdyż w przeciwnym wypadku zmieniłaby swoje miejsce jako znalezione w którymś z kroków algorytmu minimum.

Oto lista kroków algorytmu sortującego przez wybór:

1. Zmiennej pomocniczej  $i$  przypisz wartość 0.
2. Znajdź minimum we fragmencie tablicy od  $\text{tab}[i]$  do  $\text{tab}[n-1]$ .
3. Zamień miejscami  $\text{tab}[i]$  z minimum położonym najbliżej początku tablicy.
4. Wartość zmiennej  $i$  zwiększ o 1.
5. Jeśli  $i < n - 1$ , to wróć do kroku 2.
6. Zakończ.

Ponieważ jest to ostatni z typów sortowań, które w tym rozdziale omawiamy, dlatego umieścimy kod funkcji wraz z całym programem, aby można było go przeanalizować.

```
#include <iostream>
#include <cstdlib>
#include <cstdio>
#include <iomanip>
using namespace std;

void zamien(int &a, int &b)
{
    int temp = a;           // zamiana elementów
    a = b;                 // a z b
    b = temp;              // z użyciem zmiennej pomocniczej temp
}

void sortowanie_przez_wybor(int tab[], int n)
{
    int i, j, k, temp;
    for (i=0; i<n; i++)
    {
        k = i;
        for (j=i+1; j<n; j++) // w tej pętli szukamy indeksu najmniejszego elementu
            if (tab[j]<tab[k]) // znajdującego się najbliżej początku tablicy
                k = j;
        zamien(tab[k],tab[i]);
    }
}

void wypisz(int tab[], int n)
{
    for (int i=0; i<n; i++)
        cout << setw(3) << tab[i];
    cout << endl;
}

int main()
{
    int tablica[20];
    int i;
    srand (time(NULL));
    for (i=0; i<20; i++)
        tablica[i] = rand()%100;
    cout << "Tablica wypełniona losowo liczbami z przedziału <0,99>:" << endl;
    wypisz(tablica,20);
    cout << "Posortowana tablica:" << endl;
    sortowanie_przez_wybor(tablica,20);
    wypisz(tablica,20);
    getchar();
    return 0;
}
```

5.19

Pozostawiamy ci jako ćwiczenie sporządzenie schematu blokowego algorytmu sortowania przez wybór. Wykorzystaj w tym celu jego zapis za pomocą listy kroków i schemat blokowy algorytmu znajdowania elementu minimalnego, który znajduje się w tym rozdziale.

## 5.4. Praktyczne wykorzystanie tablicy dwuwymiarowej

Praktyczne zastosowanie tablic przedstawiamy na przykładzie programu, którego działanie opierać się będzie na zdefiniowanej tablicy dwuwymiarowej.



### Przykład

Napiszmy program, który wprowadzi do tablicy dwuwymiarowej oceny końcoworoczne uczniów z kilku przedmiotów, wypisze najlepszego ucznia (czyli tego, który uzyskał najwyższą średnią) i jego średnią, a także przedmiot, z którego uczniowie uzyskali najwyższą średnią, wraz z podaniem tej wartości.

Na potrzeby programu napisaliśmy dwie krótkie funkcje: wypełniającą tablicę oraz ją wyświetlającą. Kolejne dwie funkcje są do siebie podobne: pierwsza z nich oblicza średnie poszczególnych uczniów, druga – średnie poszczególnych przedmiotów. Aby przykład programu nie był długi, zastosowano ograniczenia: jest przeznaczony dla małej liczby uczniów i przedmiotów.

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

const int w = 8; // liczba uczniów
const int k = 5; // liczba przedmiotów
void wypelnij(int tab[w][k])
{
    for (int i=0; i<w; i++)
    {
        cout << "Podaj oceny ucznia: uczen" << i+1 << endl;
        cout << "w kolejnosci: j.pol. j.ang. matem. biol. inf." << endl;
        for (int j=0; j<k; j++)
            cin >> tab[i][j];
    }
}

void wyswietl(int tab[w][k])
{
    for (int i=0; i<w; i++)
    {
        cout << "u" << i+1 << "    ";
        for (int j=0; j<k; j++)
            cout << setw(7) << tab[i][j];
        cout << endl;
    }
}

void max_srednia_ucznia(int tab[w][k])
{
    float srednia_ucznia, max_srednia=0;
```

#### 5.4. Praktyczne wykorzystanie tablicy dwuwymiarowej

```
int i, j, nr_ucznia;
for (i=0; i<w; i++)
{
    srednia_ucznia = 0;
    for (j=0; j<k; j++)
        srednia_ucznia = srednia_ucznia+tab[i][j];
    srednia_ucznia = (float)srednia_ucznia/k;
    if (srednia_ucznia>max_srednia)
    {
        max_srednia = srednia_ucznia;
        nr_ucznia = (i+1);
    }
}
cout << endl << "Najwyzsza srednia ucznia to: " << max_srednia;
cout << " Osiagnal ja uczen u" << nr_ucznia;
}

void max_srednia_przedmiot(int tab[w][k])
{
    float srednia_przedmiot, max_srednia = 0;
    int i, j, nr_przedmiotu;
    for (j=0; j<k; j++)
    {
        srednia_przedmiot = 0;
        for (i=0; i<w; i++)
            srednia_przedmiot = srednia_przedmiot+tab[i][j];
        srednia_przedmiot = (float)srednia_przedmiot/w;
        if (srednia_przedmiot>max_srednia)
        {
            max_srednia = srednia_przedmiot;
            nr_przedmiotu = j;
        }
    }
    cout << endl << "Najwyzsza srednia z przedmiotu: " << max_srednia;
    cout << " Jest to ";
    switch (nr_przedmiotu)
    {
        case 0: cout << "j.pol."; break;
        case 1: cout << "j.ang."; break;
        case 2: cout << "matem."; break;
        case 3: cout << "biol."; break;
        case 4: cout << "inf."; break;
    }
}

int main()
{
    int oceny[w][k];
    wypełnij(oceny);
    cout << "          j.pol.  j.ang.  matem.  biol.  inf." << endl;
    wyswietl(oceny);
    max_srednia_ucznia(oceny);
    cout << endl;
    max_srednia_przedmiot(oceny);
    cin.ignore();
    getchar();
    return 0;
}
```

5.20

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

Poniżej prezentujemy końcowy fragment efektu działania programu:

	j - pol.	j . a n g .	matem.	b i o l .	i n f
u1	2	3	4	2	3
u2	4	4	3	4	5
u3	6	6	5	6	5
u4	2	3	3	3	2
u5	4	5	4	4	4
u6	5	5	4	5	6
u7	4	4	4	5	4
u8	5	3	3	4	5

**Najwyższa średnia ucznia to: 5.6 Osiągnął ja uczeń u3**

**Najwyższa średnia s przedmiotu: 4.25 Jest to inf.**

Spróbuj zmodyfikować program, aby pobierał od użytkownika nazwiska uczniów i nazwy przedmiotów.

### 5.5. Tablice tekstowe

Deklaracja tablicy tekstowej

W języku C++ teksty są przechowywane w tablicach o elementach typu char. Tablicę taką deklarujemy w sposób następujący:

```
char nazwa_tablicy[ilosc_elementow];
```

Podobnie jak w przypadku tablicy przechowującej liczby możemy zadeklarować i równocześnie w tej samej linii zainicjować tablicę, co przedstawiliśmy poniżej:

```
char imie[] = "Karolina"
```

Zadeklarowana w ten sposób tablica o nazwie imie będzie dziewięciu-elementowa, ponieważ w słowie „Karolina” jest 8 liter, ajako ostatni znak jest dodawany znacznik końca tekstu \0. Dlatego tablica, która ma przechować tekst n-znakowy, musi liczyć  $n + 1$  elementów.

Pracując ze zmiennymi, które przechowują teksty, możemy skonstruować algorytm, pobierający dane osób w taki oto sposób:

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

int main()
{
    char osoba[20];
    cout << "Przedstaw sie: ";
    cin >> osoba ;
    cout << "Juz wiem, jestes: " << osoba << endl;
    cin.ignore();
    getchar();
    return 0;
}
```

5.21

Wynik działania uruchomionego programu możemy zobaczyć na następnej stronie:

```
Przedstaw sie: Janek
Juz wiem, jesteś: Janek
```

Wydaje się, że program działa poprawnie. I jest tak, dopóki nie będziemy chcieli uzyskać od przedstawiającej się osoby zarówno jej imienia, jak i nazwiska i aby wartości te były równocześnie podstawione w miejsce zmiennej osoba. Co się stanie, gdy ktoś przedstawi się właśnie w ten sposób? Zobaczmy:

```
Przedstaw sie: Janek Kowalski
Juz wiem, jesteś: Janek
```

Zmienna osoba nie przyjęła wartości „Janek Kowalski”, tylko „Janek”. W wypadku przedstawienia się za pomocą ciągu znaków, w którym wystąpiła spacja, w miejsce zmiennej zostaje podstawiony tylko fragment do pierwszej napotkanej spacji. Nie uzyskaliśmy więc takiej wartości zmiennej osoba, jakiej się spodziewaliśmy.

**Za pomocą wyrażenia `cin>>tekst` (gdzie tekstem jest zmieniona będąca tablicą znaków) można pobrać tekst tylko do pierwszej spacji, reszta znaków jest ignorowana.**

Jakw takim razie w miejsce zmiennej zdefiniowanej jako tablica znaków (o ustalonej długości) można wprowadzić tekst składający się z dwóch lub więcej słów oddzielonych spacją? Zróbmy to za pomocą instrukcji: `cin.getline(osoba, 20);` zamiast `cin > osoba;`

Pobrała zostanie wówczas cała linia o długości 20 znaków (bez względu na to, czy pobieranym znakiem jest spacja, czy jakikolwiek inny znak) i wstawiona w miejsce tablicy znaków o nazwie osoba. Oczywiście, można podać maksymalnie 19 znaków, dwudziesty znak to wspomniany wcześniej znacznik końca tekstu.

Oto program, w którym wykorzystana zostanie instrukcja pobrania całej linii:

---

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

int main()
{
    char osoba[20];
    cout << "Przedstaw sie: ";
    cin.getline(osoba,20);
    cout << "Juz wiem, jesteś: " << osoba << endl;
    getchar();
    return 0;
}
```

---

5.22

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

Teraz efekt działania programu jest następujący:

```
Przedstaw sie: Janek Kowalski  
Juz wiem, jestes: Janek Kowalski
```

Zmienna osoba przyjęła wartość tekstową wraz ze spacją oddzielającą imię od nazwiska.

**Zapamiętaj**

Teksty, w których nie będzie białych znaków (czyli znaków niewidocznych, np.: spacji, tabulatora), możesz pobierać do tablicy za pomocą składni:

```
cin >> napis
```

Kiedy jednak w pobieranym ciągu mogą wystąpić białe znaki, stosuj:

```
cin.getline(napis, ilosc_znakow)
```

gdzie:

`napis` to nazwa tablicy znaków,

`ilosc_znakow` to liczba znaków, jaka zostanie wpisana do tablicy `napis`

Odczytywanie :  
tablicy tekstopowej ;

Wiesz już, jak pobierać i wstawiać do tablicy napisy, teraz pora się dowieźć, jak je poprawnie odczytywać w każdej sytuacji. Jeśli chcemy wypisać tablicę znak po znaku, to odczyt umieszczaćmy w pętli, przy czym liczba przebiegów tej pętli musi być równa liczbie wprowadzonych znaków. Odczytywanie lub wypisywanie tekstu znak po znaku jest nam potrzebne na przykład do tego, aby każdy znak uzyskać w nowej linii, zapisać tekst od tyłu do przodu bądź dokonać operacji na każdym znaku (np. zamienić go na inny). My wykorzystamy ten zapis do zamiany znaków działań matematycznych na znaki zapytania - można w ten sposób tworzyć proste zagadki matematyczne.



**Przykład**

Napiszmy program, który w pobranym od użytkownika równaniu matematycznym zamieni wszystkie znaki działań matematycznych na znaki zapytania i w tej postaci wypisze równanie na ekranie monitora.

```
#include <iostream>  
#include <cstdio>  
using namespace std;  
  
int main()  
{  
    char dz[20]; // linia, do której zostanie wczytane działanie  
    int i = 0;  
    cout << "Rownanie matematyczne do zamiany: ";  
    cin.getline(dz,20);  
    cout << "Rownanie po ukryciu znakow dzialan arytmetycznych: ";  
    while (dz[i]!='\0')  
    {
```

```

if (dz[i]=='+') || dz[i]=='-' || dz[i]=='*' || dz[i]=='/')
    cout << " ? ";           // znaki działań zostaną zamienione na znak zapytania
else
    cout << dz[i];          // inne pozostałe niezmienione
    i++;
}
getchar();
return 0;
}

```

5.23

Efekt działania programu prezentujemy poniżej:

```

Działanie do zamiany: 2*12+200-60 = 264
Działanie po ukryciu znaków działań arytmetycznych:
 2 ? 12 ? 200 ? 60 = 264

```

Zaistniała tu rzeczywista potrzeba wypisywania kolejno znak po znaku, gdyż na części znaków chcieliśmy wykonać instrukcję zamiany znaku na inny, resztę znaków pozostawiając bez zmian.

#### **Zapamiętaj**

Jeśli zadeklarujesz tablicę znaków o długości  $n$ , to możesz do niej wprowadzić tylko  $n - 1$  znaków, gdyż ostatni z elementów tablicy jest znakiem \ 0 - znacznikiem końca tekstu.

### **5.5.1. Modyfikacje dokonywane na tekście**

Z każdym znakiem klawiaturowym jest skojarzony jego kod ASCII będący liczbą całkowitą. Poniżej przedstawiamy fragment tablicy kodów ASCII dla wielkich i małych liter alfabetu łacińskiego. Kod ASCII nie jest jedyną metodą kodowania liter, ale nasze rozważania opierają się właśnie na nim. Nie musisz oczywiście znać na pamięć wartości kodów odpowiadających danym znakom, zawsze możesz programowo otrzymać informację o kodzie dla danego znaku.

Tablica kodów ASCII

Znak	Kod	Znak	Kod	Znak	Kod	Znak	Kod	Znak	Kod
A	65	L	76	W	87	g	103	r	114
B	66	M	77	X	88	h	104	s	115
C	67	N	78	Y	89	i	105	t	116
D	68	O	79	Z	90	j	106	u	117
E	69	P	80	.....	.....	k	107	v	118
F	70	Q	81	a	97	l	108	w	119
G	71	R	82	b	98	m	109	x	120
H	72	S	83	c	99	n	110	y	121
I	73	T	84	d	100	o	111	z	122
J	74	U	85	e	101	p	112		
K	75	V	86	f	102	q	113		

Tab. 5.1. Fragment tablicy kodów ASCII dotyczący liter alfabetu łacińskiego

Jeśli program wykona instrukcję:

```
cout << (int) 'a';
```

to wynikiem jej działania będzie wyświetlenie kodu małej litery *a*. Przyjrzyj się tabeli kodów (tab. 5.1) i zauważ, że wielkie litery mają wartości kodów ASCII z przedziału (65, 90), małe zaś z przedziału (97, 122). Charakterystyczne jest to, że różnica pomiędzy wartościami kodów małej i odpowiadającej jej wielkiej litery jest stała i wynosi 32. Już ta wiedza ci wystarczy, aby napisać program, który tekst wprowadzony małymi literami zamieni na tekst o tej samej treści, ale pisany wielkimi literami. Napiszmy krótką funkcję, która pobierze tablicę znaków z wpisany do niej tekstem, a następnie wszystkie znaki zamieni na znaki o kodach ASCII mniejszych o 32. Zakładamy przy tym, że wprowadzamy tekst pisany małymi literami, niezawierający białych znaków:

```
void zamiana(char os[])
{
    int i = 0;
    while (os[i]!='\0')
    {
        cout << (char)((int)os[i]-32);
        i++;
    }
}
```

5.24

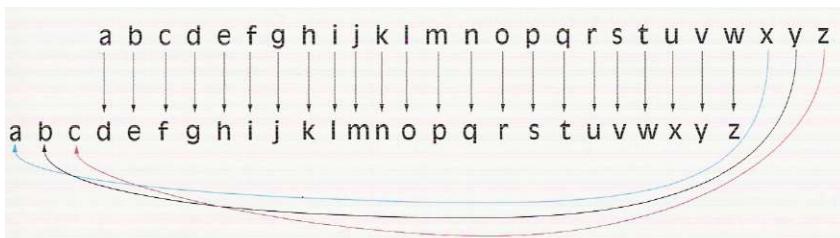
W efekcie działania programu, w którym będzie wykorzystana ta funkcja, otrzymamy przykładowy wynik:

**Wprowadź tekst pisany małymi literami, niezawierający białych znaków:  
Zamieniamy na tekst pisany dużymi literami:  
INFORMATYKA**

### 5.5.2. Szyfr Cezara

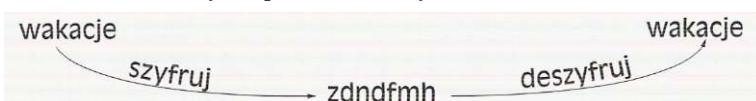
Opis metody  
szyfrowania

Umiejętność wykorzystania w algorytmach kodowania znaków przydaje się w wielu sytuacjach. Jedną z nich jest szyfrowanie tekstu. W pierwotnej swej postaci szyfr Cezara polegał na zastępowaniu każdej litery szyfrowanego tekstu tą, która w alfabetie znajdowała się o trzy miejsca dalej: litera *a* była zmieniana na *d*, litera *b* na *e*, a gdy alfabet dobiegał końca, wracano do początku i litera *z* była zamieniana na *c* (ryc. 5.13 na stronie obok). Tekst szyfrowany był ciągiem znaków niezawierającym spacji. Nazwa szyfru pochodzi od Juliusza Cezara, który przesyłał do Cicerona listy zaszyfrowane właśnie tym sposobem. Klasyczny algorytm opierał się na alfabetie łacińskim dwudziestoszcioznakowym.



Ryc. 5.13. Ilustracja szyfrowania za pomocą szyfru Cezara

W implementacji komputerowej znak będący małą literą alfabetu będziemy podmieniać innym znakiem, którego kod ASCII ma wartość o trzy większą. Wyjątkiem są trzy końcowe litery alfabetu łacińskiego, zamieniane na litery początkowe. Aby szyfr należycie wykorzystać, muszą działać dwie funkcje: jedna szyfrująca tekst, a druga deszyfrująca. Schemat działania szyfru przedstawia rycina 5.14:



Ryc. 5.14. Ilustracja szyfrowania i deszyfrowania tekstu „wakacje”

Funkcje `szyfruj` i `deszyfruj` są względem siebie odwrotne i spełniony jest warunek: `deszyfruj(szyfruj(tekst)) = tekst`.

Zapiszmy obydwie funkcje: szyfrującą i deszyfrującą. Klasyczny kod Cezara opiera się na przesunięciu znaku o trzy "pozycje w przód; my napiszemy funkcje bardziej uniwersalne - wartość przesunięcia będziemy mogli modyfikować w zakresie dodatnich liczb całkowitych (ograniczeniem jest oczywiście maksymalna wartość typu całkowitego). Liczbę pozycji, o które się przesuwamy w celu znalezienia znaku po zaszyfrowaniu, nazwiemy **kluczem**. W zdefiniowanych funkcjach używamy instrukcji, które znakowi przypisują jego kod ASCII, na przykład:

`(int)'a'` - ma wartość 97, czyli kod ASCII litery a.

Instrukcja ta nazywa się **rzutowaniem** na typ `integer` i służy do przedstawienia zmiennej lub stałej jednego typu w reprezentacji typu, na który rzutujemy. Instrukcję tę zapisujemy w postaci: `(nazwa_typeru_na_jaki_typ_rzutujemy) zmienna_rzutowania`. Aby dla zadanej wartości kodu ASCII odczytać, jaka litera się pod nią kryje, wystarczy użyć odpowiedniego rzutowania:

`(char) 100` - jest to znak „d”, gdyż ten właśnie znak ma kod ASCII o wartości 100.

A zatem instrukcja `tekst[i] = (char)((int)tekst[i]+klucz);` pobierze kod ASCII znaku `tab[i]`, doda do wartości kodu wartość klucza i wstawi do tablicy znak, którego kod ASCII ma tak obliczoną wartość.

Funkcja szyfrująca  
i deszyfrująca  
tekst

Instrukcja  
rzutowania  
pomiędzy typami

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

Oto zapis funkcji szyfrującej:

```
void szyfruj(char tekst[], int klucz)
{
    int i = 0;
    klucz = klucz%26; // (1)
    while (tekst[i]!='\0')
    {
        if ((int)tekst[i]>122-klucz) // (2)
            tekst[i] = (char)((int)tekst[i]+klucz-26);
        else
            tekst[i] = (char)((int)tekst[i]+klucz);
        i++;
    }
}
```

5.25

W linii (1) obliczamy resztę z dzielenia wartości wpisanego klucza przez liczbę liter w kodowanym alfabetie. Chodzi o to, że przesunięcie znaku o całkowitą wielokrotność liczby 26 (liczba znaków w analizowanym alfabetie) nie zmienia kodowanego znaku. Dlatego w kluczu interesuje nas tylko reszta z dzielenia. Dla wszystkich kluczy o wartościach mniejszych niż 26 wartość klucza się w tej linii nie zmienia.

Warunek oznaczony komentarzem (2) został wprowadzony po to, aby uniemożliwić wyjście poza zakres dostępnych liter alfabetu przy kodowaniu znaków z końca alfabetu lub dla dużych wartości klucza. Kod ASCII znaku w przykładowym kodowaniu nie może przekroczyć wartości 122.

Do czego jednak przydałby nam się zaszyfrowany tekst, gdybyśmy nie znali metody, jak go na powrót odszyfrować? Będziemy zatem deszyfrować tablicę znaków, aż natrafimy na znacznik końca tekstu. Funkcja deszyfrująca ma postać bardzo zbliżoną do funkcji szyfrującej, tylko znaki zamieniały na te, które w alfabetie stoją o trzy pozycje wcześniej (równiej wartości klucza), przy czym początkowe litery zamieniamy na końcowe, analogicznie jak przy szyfrowaniu tekstu końcowe litery zamienialiśmy na te z początku alfabetu. Zapis funkcji deszyfrującej wygląda następująco:

```
void deszyfruj(char tekst[], int klucz)
{
    int i = 0;
    klucz = klucz%26;
    while (tekst[i]!='\0')
    {
        if ((int)tekst[i]-klucz<97)
            tekst[i] = (char)((int)tekst[i]-klucz+26);
        else
            tekst[i] = (char)((int)tekst[i]-klucz);
        i++;
    }
}
```

5.26

Teraz już możemy użyć zdefiniowane funkcje w krótkim programie. Jeśli z klawiatury wprowadzimy wartość klucza równą 3, to program będzie realizował szyfr Cezara.

## 5.6. Rozwiązywanie układów równań metodą eliminacji Gaussa

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

int main()
{
    char a[50];
    int klucz;
    cout << "Podaj klucz (dla szyfru CEZARA wpisz 3): ";
    cin >> klucz;
    cout << "TEKST DO ZASZYFROWANIA:      ";
    cin >> a;
    cout << "TEKST ZASZYFROWANY:           ";
    szyfruj(a,klucz);
    cout << a << endl;
    cout << "TEKST ODSZYFROWANY:          ";
    deszyfruj(a,klucz);
    cout << a << endl;
    cin.ignore();
    getchar();
    return 0;
}
```

5.27

Jeśli jako klucz wpiszemy 3, a jako tekst do zaszyfrowania wprowadzimy kolejne litery alfabetu, to efekt programu będzie następujący:

TEKST DO ZASZYFROWANIA:	abcdefghijklmnoqrstuvwxyz
TEKST ZASZYFROWANY:	defghijklmnoqrstuvwxyzabc
TEKST ODSZYFROWANY:	abcdefghijklmnoqrstuvwxyz

Przedstawiliśmy proste zastosowanie szyfru Cezara do szyfrowania tekstów o ograniczonej długości (założyliśmy, że maksymalnie zostanie zakodowanych 49 znaków). W zależności od potrzeb można oczywiście zwiększyć lub zmniejszyć liczbę elementów tablicy tekstowej.

### 5.6. Rozwiązywanie układów równań metodą eliminacji Gaussa

Na koniec zajmiemy się problemem rozwiązywania układu  $n$  równań liniowych z  $n$  niewiadomymi metodą eliminacji Gaussa. Zacznijmy od prostego przykładu. Dany jest układ trzech równań z trzema niewiadomymi:

$$\begin{cases} x_1 + x_2 + 2x_3 = 3 \\ 2x_1 + x_2 + x_3 = 4 \\ x_1 + 2x_2 + x_3 = 0 \end{cases}$$

Pamiętasz zapewne z lekcji matematyki, że do rozwiązywania takiego układu równań (tzn. obliczania  $x_1, x_2, x_3$ ) pomocne są dwie podstawowe operacje:

1. Współczynniki każdego równania można przemnożyć przez dowolną liczbę różną od 0, a rozwiązanie układu nie ulegnie zmianie.
2. Dwa dowolne równania układu można dodać stronami, a rozwiązanie układu nie ulegnie zmianie.

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

Czasem w metodzie Gaussa będziemy również korzystać z trzeciej, dość oczywistej właściwości układów równań, mówiącej, że zamiana kolejności równań nie zmienia rozwiązania.

Stosując wymienione wyżej metody, zróbmy przekształcenie: przemnożone przez -2 pierwsze równanie dodajmy stronami do drugiego oraz przemnożone przez -1 pierwsze równanie dodajmy stronami do trzeciego. Wówczas:

$$\begin{cases} x_1 + x_2 + 2x_3 = 3 \\ 0x_1 - x_2 - 3x_3 = -2 \\ 0x_1 + x_2 - x_3 = -3 \end{cases}$$

Teraz wykonajmy jeszcze jedno przekształcenie - dodajmy stronami drugie równanie do trzeciego:

$$\begin{cases} x_1 + x_2 + 2x_3 = 3 \\ 0x_1 - x_2 - 3x_3 = -2 \\ 0x_1 + 0x_2 - 4x_3 = -5 \end{cases}$$

Ostatnia postać układu równań nosi nazwę postaci trójkątnej (niezerowe współczynniki przy niewiadomych tworzą trójkąt - wszystkie współczynniki poniżej tzw. głównej przekątnej są równe 0) i bardzo łatwo uzyskać z niej rozwiązanie. W celu dokończenia obliczeń trzeba wyznaczyć z trzeciego równania  $x_3$ . Wynik wstawiamy do równania drugiego i obliczamy  $x_2$  - podobnie postępujemy, by otrzymać  $x_1$ .

Przedstawiona metoda jest właśnie metodą eliminacji Gaussa (eliminacji dlatego, że z równań eliminujemy niewiadome).

Uogólnijmy teraz problem. Zapiszmy układ  $n$  równań z  $n$  niewiadomymi; założymy przy tym, że współczynniki układu są różne od zera. Rozwiązywanie kwestii istnienia zerowych współczynników omówiono w dalszej części podręcznika.

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = a_{1,n+1} \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = a_{2,n+1} \\ \dots & \dots & \dots & \dots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = a_{n,n+1} \end{cases}$$

Aby wyeliminować z równań drugiego i kolejnych współczynniki przy pierwszej niewiadomej (tzn. wyzerować je), wystarczy do każdego z tych równań dodać stronami pierwsze równanie przemnożone przez odpowiedni czynnik. Dla równania drugiego czynnik ten wynosi  $\left(-\frac{a_{21}}{a_{11}}\right)$ .

W ogólnej postaci dla  $i$ -tego równania czynnikiem będzie  $\left(-\frac{a_{i1}}{a_{11}}\right)$ . Układ równań po takiej operacji uzyska postać:

## 5.6. Rozwiązywanie układów równań metodą eliminacji Gaussa

$$\left\{ \begin{array}{l} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = a_{1,n+1} \\ \left( a_{22} - \frac{a_{21}}{a_{11}}a_{12} \right)x_2 + \cdots + \left( a_{2n} - \frac{a_{21}}{a_{11}}a_{1n} \right)x_n = a_{2,n+1} - \frac{a_{21}}{a_{11}}a_{1,n+1} \\ \dots \quad \dots \quad \dots \\ \left( a_{n2} - \frac{a_{n1}}{a_{11}}a_{12} \right)x_2 + \cdots + \left( a_{nn} - \frac{a_{n1}}{a_{11}}a_{1n} \right)x_n = a_{n,n+1} - \frac{a_{n1}}{a_{11}}a_{1,n+1} \end{array} \right.$$

Ze wszystkich równań oprócz pierwszego zostały wyeliminowane współczynniki przy pierwszej niewiadomej. Nowe współczynniki można w ogólnej postaci zapisać:

$$a_{ij}^{(1)} = a_{ij} - \frac{a_{i1}}{a_{11}}a_{1j}$$

gdzie  $i$  - numer wiersza  $[2, \dots, n]$ ,  $j$  - numer kolumny  $[2, \dots, n+1]$ . Indeks górnny został dodany, aby odróżnić nowe współczynniki od poprzednich.

Następnie należy wyzerować współczynniki przy  $x_2$  w równaniach trzecim i kolejnych. W tym celu dodajemy stronami drugie równanie do kolejnych (leżących niżej), mnożąc je wcześniej przez odpowiednie czynniki. Zauważ, że współczynniki w równaniu pierwszym pozostają niezmienione, współczynniki w równaniu drugim zmieniamy jeden raz, w trzecim - 2 razy (pierwszy raz przy eliminowaniu współczynnika przy  $x_1$  i drugi raz przy eliminowaniu współczynnika przy  $x_2$ ). Ogólnie możemy zapisać układ równań po sprowadzeniu do postaci trójkątnej:

$$\left\{ \begin{array}{l} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = a_{1,n+1} \\ a_{22}^{(1)}x_2 + \cdots + a_{2n}^{(1)}x_n = a_{2,n+1}^{(1)} \\ \dots \quad \dots \quad \dots \\ a_{nn}^{(n-1)}x_n = a_{n,n+1}^{(n-1)} \end{array} \right.$$

Widać z tego zapisu, że współczynniki w ostatnim równaniu są zmieniane  $n - 1$  razy. Z tej postaci równania bardzo łatwo obliczysz niewiadome, jeśli zaczniesz od wyznaczenia ostatniej.

Współczynniki w układzie równań po sprowadzeniu go do postaci trójkątnej mają ogólną postać:

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}a_{kj}^{(k-1)}$$

gdzie:  
 $k = 1, 2, \dots, n - 1$   
 $j = k + 1, k + 2, \dots, n + 1$   
 $i = k + 1, k + 2, \dots, n$

Postać trójkątna  
układu równań

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

Macierzowy zapis układu równań

Naturalną strukturą danych, potrzebną do zaimplementowania tej metody, jest tablica.

$$\text{Zapiszmy zatem nasz układ równań: } \begin{cases} x_1 + x_2 + 2x_3 = 3 \\ 2x_1 + x_2 + x_3 = 4 \\ x_1 + 2x_2 + x_3 = 0 \end{cases}$$

$$\text{w postaci macierzowej: } \begin{bmatrix} 1 & 1 & 2 \\ 2 & 1 & 1 \\ 1 & 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 0 \end{bmatrix}$$

Pierwszą - kwadratową macierz nazywamy **macierzą współczynników**. drugą - jednowymiarową macierz nazywamy **wektorem niewiadomych**, a trzecią - **wektorem wyrazów wolnych**. Macierzowy sposób zapisu równań, nawet jeśli stykasz się z nim po raz pierwszy, powinien być dla ciebie zrozumiały. Aby otrzymać pierwsze równanie w zapisie tradycyjnym, wystarczy dodać iloczyny kolejnych współczynników z pierwszego wiersza i odpowiadających im niewiadomych z drugiej macierzy, a po znaku równości przepisać pierwszą wartość z macierzy wyrazów wolnych:

W ten sam sposób można odtworzyć kolejne równania.

Do tej pory nie wspomnialiśmy w naszych rozważaniach o pewnej komplikacji, często pojawiającej się w czasie sprowadzania układu równań do postaci trójkątnej. Przeanalizujmy innv układ równań:

$$\begin{cases} x_1 + x_2 + 2x_3 = 3 \\ 2x_1 + 2x_2 + x_3 = 4 \\ x_1 + 2x_2 + x_3 = 0 \end{cases}$$

Zauważ, że po wyeliminowaniu współczynników przy  $x_1$  z równań drugiego i trzeciego otrzymujemy:

$$\begin{cases} x_1 + x_2 + 2x_3 = 3 \\ 0x_1 + 0x_2 + -3x_3 = -2 \\ 0x_1 + x_2 - x_3 = -3 \end{cases}$$

Próba mechanicznego przemnożenia drugiego równania przez jakiś czynnik i dodania go do równania trzeciego w celu wyzerowania w równaniu trzecim współczynnika przy  $x_1$ , nie powiedzie się, ponieważ przy  $x_1$ , w poprzednim równaniu jest 0. Przyjrzyj się dokładnie wcześniejszym ogólnym wzorom, a dostrzeżesz, że taka próba doprowadzi do dzielenia przez 0, co zakończy się błędem programu. Istnieje proste rozwiązanie tego problemu, mianowicie zamiast dodawać te równania stronami, wystarczy równanie drugie zamienić kolejnością z trzecim:

## 5.6. Rozwiązywanie układów równań metodą eliminacji Gaussa

$$\begin{cases} x_1 + x_2 + 2x_3 = 3 \\ 0x_1 + x_2 - x_3 = -3 \\ 0x_1 + 0x_2 + -3x_3 = -2 \end{cases}$$

Przedstawiamy poniżej pełny kod programu, w którym przewidziano obliczanie układów maksymalnie 20 równań. Nie jest to ograniczenie metody, wynika ono z faktu, że na tym etapie musisz zadeklarować tablice jeszcze przed uruchomieniem programu.

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

int main()
{
    float a[20][21];
    float x[20];
    float czynnik, temp;
    int i, j, k, n;

    cout << "\nPodaj liczbę równań w układzie (max 20): n = ";
    cin >> n; // Pobranie
    for (i=0 ; i<n ; i++) // współczynników
        for (j=0 ; j<n+1 ; j++) // układu
    {
        cout << "Podaj a[" << i+1 << "] " << "[" << j+1 << "] = "; // równań
        cin >> a[i][j]; // *****
    }
    for (i=0; i<n; i++) // Wyświetlenie
    {
        cout << endl; // tablicy
        for (j=0; j<n+1; j++) // współczynników
            cout << setw(5) << a[i][j]; // układu
    }

    //****Początek eliminacji Gaussa****
    for (k=0; k<n; k++)
    {
        if (a[k][k]==0) // Jeśli istnieje niebezpieczeństwo dzielenia przez zero
        {
            int z = k+1;
            while (a[z][k]==0 && z<n) // Sprawdzamy, czy w kolejnych wierszach jest
                z++; // liczba różna od zera w kolumnie k
            if (z==n)
            {
                cout << endl << "Nie ma rozwiązań lub jest nieskonczenie wiele rozwiązań";
                cin.ignore();
                getchar();
                return 1; // Jeśli w kolumnie k kolejne
                           // liczby są równe zeru, to koniec programu
            }
            else
                for (j=k; j<n+1; j++) // 4 kolejne wiersze programu
                { // to zamiana wierszy miejscami
                    temp = a[z][j];
                    a[z][j] = a[k][j];
                    a[k][j] = temp;
                }
        }
    }
}
```

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

```
        }
    }
    for (i=k+1; i<n; i++)
        for (j=n; j>=0; j--)
            a[i][j] = a[i][j]-a[i][k]*a[k][j]/a[k][k]; // eliminacja współczynników
}
cout << endl;
for (i=0; i<n; i++)           // 4 kolejne wiersze programu to wypisanie tablicy po
{                           // sprowadzeniu układu do postaci trójkątnej
    cout << endl;
    for (j=0; j<n+1; j++)
        cout << setw(5) << a[i][j];
}
float suma;
for (i=n-1; i>=0; i--)       // ten i 5 kolejnych wierszy programu to
{                           // obliczenie niewiadomych układu i wpisanie
    suma = 0;                // ich do tablicy x
    for (j=i+1; j<n; j++)
        suma = suma+a[i][j]*x[j];
    x[i] = (a[i][n]-suma)/a[i][i];
}
cout << endl;
for(i=0; i<n; i++)          // wypisanie rozwiązania
    cout << endl << "x" << i+1 << " = " << x[i];
cin.ignore();
getchar();
return 0;
}
```

5.28

Program pobiera od użytkownika informację o liczbie równań, następnie użytkownik wprowadza kolejno współczynniki równania. Jeśli układ ma jednoznaczne rozwiązanie, to zostaje ono wyprowadzone na ekran monitora. W przeciwnym wypadku zostaje wyprowadzony napis: „Nie ma rozwiązań lub jest nieskończenie wiele rozwiązań”.



### Pytania kontrolne

1. Podaj przykład deklaracji tablicy jednowymiarowej i tablicy dwuwymiarowej.
2. Czy w jednej tablicy można przechowywać elementy różnego typu?
3. Podaj jeden ze sposobów wypełniania tablicy jednowymiarowej.
4. Podaj jeden ze sposobów wypełniania tablicy dwuwymiarowej.
5. Jakiego typu może być indeks tablicy?
6. Jak działa algorytm liniowego wyszukiwania w tablicy elementu o zadanej wartości?
7. Dlaczego algorytm wyszukiwania z wartownikiem jest szybszy od algorytmu liniowego bez wartownika?
8. Na czym polega bąbelkowe sortowanie tablicy?
9. Na czym polega sortowanie tablicy przez wybór?
10. Na czym polega sortowanie tablicy przez wstawianie?

11. Jak można wprowadzać teksty do tablicy znaków? Czy ma znaczenie występowanie znaków spacji w tekście? Jeśli tak, to jakie?
12. Co to jest kod ASCII?
13. Opisz sposób działania szyfru Cezara.

### Ćwiczenia

1. Napisz program, który znajduje wartości najczęściej występujące w jednowymiarowej tablicy 20-elementowej liczb całkowitych wylosowanych z zakresu  $<0, 10>$ . Jeśli takich wartości jest kilka, wówczas podaje wszystkie. Natomiast jeśli wszystkie wartości tablicy występują jednakowo często, to wyświetla komunikat, że w zbiorze nie ma wartości najczęściej występującej.  
Wskazówka: Zadanie to najłatwiej można wykonać, porządkując najpierw elementy zbioru.
2. Oblicz medianę zbioru z ćwiczenia 1. Mediana to wartość środkowa w uporządkowanym zbiorze o nieparzystej liczbie elementów. W zbiorze o parzystej liczbie elementów są dwa elementy środkowe, więc za medianę przyjmij ten z niższym indeksem.
3. Oblicz odchylenie standardowe zbioru 20 liczb wybranych losowo z zakresu  $<0,5>$ . Odchylenie standardowe jest tak zwaną miarą rozproszenia zbioru. Jeśli jej wartość jest mała, to zbiór jest skupiony wokół wartości średniej; jeśli duża, to dane są bardziej rozproszone w porównaniu z wartością średnia. Odchylenie standardowe wyraża się wzorem:

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n-1}}$$

gdzie:  $\bar{x}$  oznacza średnią arytmetyczną wszystkich liczb w zbiorze,  $n$  to liczba wszystkich elementów (w ćwiczeniu jest to 20). W liczniku pod pierwiastkiem jest zapis oznaczający sumę kwadratów różnic kolejnych wartości  $x$  w zbiorze i wartości średniej.

4. Napisz program, który obliczy sumy odpowiadających sobie elementów dwóch tablic jednowymiarowych  $A$ ,  $B$  wypełnionych liczbami losowymi z zakresu od zera do wartości podanej przez użytkownika, a następnie wpisze je do tablicy  $C$ .
5. Napisz program znajdujący minimum w każdej kolumnie tablicy dwuwymiarowej o wymiarach  $5 \times 7$ , wypełnionej liczbami wybranymi losowo z przedziału  $<0, 9>$ , które następnie wypisuje na ekran monitora.
6. Policz ilość elementów przekraczających wartość podaną z klawiatury dla każdej z kolumn tablicy dwuwymiarowej.

## 5. Tablice danych - przykłady i wykorzystanie w algorytmach

7. Napisz program, który dla tablicy kwadratowej  $n \times n$  wypełnionej liczbami wybranymi losowo z przedziału  $<1, 10>$  liczy sumę elementów położonych nad główną przekątną oraz położonych pod główną przekątną, a wynik wyprowadza na ekran monitora.
8. Z tablicy dwuwymiarowej usuń zadany wiersz z jednocześnie komprezją tablicy (tzn. przesunięciem do góry wszystkich wierszy leżących poniżej).
9. Utwórz tablicę dwuwymiarową i wypełnij ją liczbami wybranymi losowo z przedziału  $<0, 50>$ . Następnie w każdą komórkę wpisz sumę liczby, która znajduje się w tej komórce, i wszystkich liczb sąsiadujących (również po przekątnej).
10. Napisz program, który posortuje rosnąco elementy każdego z wierszy w dwuwymiarowej tablicy kwadratowej o wymiarach  $7 \times 7$ .
11. Napisz program, który w ciągu znaków zamienia litery wielkie na małe i odwrotnie, pozostałe znaki pozostawiając bez zmian.
12. Napisz program, który wypełni tablicę dwuwymiarową mającą 5 wierszy i 10 kolumn kolejnymi liczbami od 1 do 50, tak aby liczby te wypełniały kolejno wszystkie kolumny od góry do dołu.
13. Napisz funkcję, która oblicza wartość maksymalnego elementu tablicy dwuwymiarowej o 4 wierszach i 7 kolumnach. Sprawdź jej działanie w napisanym przez siebie programie. Następnie wprowadź zmiany w programie tak, aby znajdowany był element minimalny.
14. Napisz program zamiany  $i$ -tego wiersza z  $j$ -tą kolumną, gdzie  $i$  oraz  $j$  są odpowiednio numerami wiersza i kolumny, na których przecięciu znajduje się element maksymalny w tablicy kwadratowej o wymiarach  $n \times n$ .

## 6. Rekurencja

W tym rozdziale opisujemy inną metodę konstrukcji algorytmu, zwaną rekurencją. Zapoznamy cię z rekurencyjnymi rozwiązaniami wielu klasycznych algorytmów. Dokładnie omówimy dwie metody sortowania: przez scalanie i sortowanie szybkie, dla których rekurencja jest najprostszą metodą implementacji. Poznasz również dwa klasyczne problemy: konika szachowego i ośmiu hetmanów, rozwiązywane za pomocą tak zwanych algorytmów z powrotami.

### 6.1. Funkcje rekurencyjne w informatyce

Zacznijmy od definicji: **metodą rekurencyjną** nazywamy taką metodę, w której stosujemy rekurencję (zwana także rekursją). Być może takie sformułowanie wydaje ci się pozbawione sensu. Tymczasem jest ono charakterystyczne właśnie dla rekurencji, która oznacza odwoływanie się funkcji (lub definicji) do samej siebie.

Definicja rekurencji

W matematyce funkcją rekurencyjną nazywamy funkcję, która jest zdefiniowana za pomocą samej siebie. W informatyce zaś **funkcją rekurencyjną** jest funkcja, która wywołuje samą siebie.

W powyższych definicjach nie wspomnieliśmy jeszcze o najważniejszym elemencie, mianowicie nie ma w nich określenia warunku zakończenia odwołań rekurencyjnych. Prześledźmy na przykładach zastosowanie funkcji rekurencyjnej.

#### 6.1.1. Silnia liczby naturalnej

Silnią dodatniej liczby naturalnej nazywamy iloczyn kolejnych liczb naturalnych, począwszy od liczby 1 aż do liczby, dla której obliczamy wartość silni. Na przykład dla liczby 5 silnia jest równa wartości iloczynu  $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$ , czyli 120. Silnia liczby 0 wynosi 1 (z definicji).

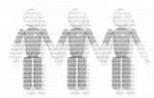
Silnia obliczana iteracyjnie

Formalny zapis matematyczny funkcji silnia może wyglądać następująco:

$$\begin{aligned}n! &= 1 && \text{dla } n \in \{0, 1\} \\n! &= 1 \cdot \dots \cdot n && \text{dla } N_+ - \{1\}\end{aligned}$$

#### Przykład

Napiszmy funkcję iteracyjną, która obliczy silnię dowolnej liczby naturalnej.



## 6. Rekurencja

```
double silnia_iteracyjnie(int n)
{
    double silnia = 1;
    for (int i=2; i<n+1; i++)
        silnia = silnia*i;
    return silnia;
}
```

6.1

Powyzsza funkcja pobierze naturalna liczbę  $n$ , a wynikiem jej działania będzie iloczyn kolejnych liczb naturalnych (obliczenia wykonają się w pętli `for`) aż do  $n$  włącznie. Jeśli chcemy policzyć silnię 0, program nie wykona pętli, a funkcja przyjmie wartość 1. Ponieważ silnia jest funkcją osiągającą bardzo duże wartości dla małych  $n$ , wybraliśmy dla nich typ `double`.

Silnia obliczana rekurencyjnie : Spróbujmy teraz ten sam problem zdefiniować rekurencyjnie. Zauważmy od tego, że przy znanej wartości silni liczby 4 łatwo obliczysz silnię dla 5, ponieważ skoro  $4! = 24$ , to  $5! = 24 \cdot 5$ , co daje 120. W ogólnej postaci możemy więc zapisać:

$$n! = (n - 1)! \cdot n$$

Zwróć uwagę na odwołanie rekurencyjne - silnia liczby  $n$  jest równa wartości silni liczby  $(n - 1)$  pomnożonej przez liczbę  $n$ . Oczywiście, aby zgodnie z naszą definicją rekurencyjną policzyć silnię dla  $(n - 1)$ , należy obliczyć silnię  $(n - 2)$  i pomnożyć ją przez  $(n - 1)$  itd. W celu zakończenia tych czynności musimy postawić warunek zakończenia odwołań rekurencyjnych.

Aby funkcja rekurencyjna była poprawna, musi mieć ścisłe zdefiniowany warunek zakończenia wywołań rekurencyjnych.

W przypadku funkcji `silnia` takim warunkiem jest definicja silni dla  $n = 0$ , czyli  $0! = 1$ . Pełna definicja rekurencyjną funkcji `silnia` ma zatem postać:

$$\begin{aligned} n! &= 1 && \text{dla } n = 0 \\ n! &= (n - 1)! \cdot n && \text{dla } n \in N_+ \end{aligned}$$

Pozostaje nam tylko zapisać za pomocą języka programowania definicję matematyczną w postaci funkcji rekurencyjnej:

```
double silnia(int n)
{
    if (n==0)                  // warunek początkowy rekurencji (1)
        return 1;
    else
        return silnia(n-1)*n;   // rekurencyjne wywołanie funkcji (2)
}
```

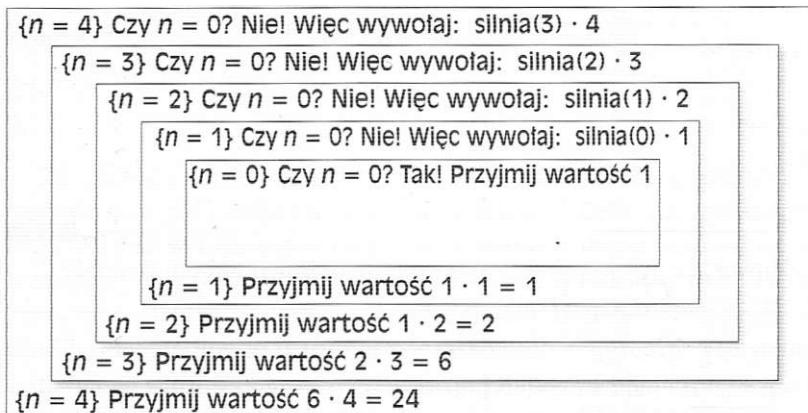
6.2

W linii 2 jest zawarte wywołanie rekurencyjne funkcji `silnia`. We wnętrzu funkcji wywołujemy funkcję, którą właśnie piszemy. Taki zapis powo-

duje, że program po dotarciu do linii numer 2 kodu ma jest w stanie do kończyć instrukcję, musi bowiem wywołać funkcję `silnia` z parametrem `/z o 1 mniejszym`. Każde wywołanie funkcji jest związane z zapamiętaniem kopii wszystkich zmiennych funkcji oraz z umieszczeniem na stosie adresu powrotnego, czyli miejsca w programie, do którego system ma powrócić po zakończeniu funkcji.

**Stosem** nazywamy obszar pamięci, w którym są przechowywane zmienne wywoływanych funkcji oraz adresy powrotne. Stos charakteryzuje się tym, że jest strukturą powoływaną dynamicznie, to znaczy nie ma stałego miejsca w pamięci i jest powiększany w zależności od potrzeb. Zasada działania stosu przypomina sposób postępowania ze stosem książek ułożonych jedna na drugiej - aby się dostać do książki z wnętrza stosu, należy zdjąć wszystkie książki położone wyżej. Na rycinie 6.1 przedstawiono graficznie stos z wywołaniami rekurencyjnymi funkcji `silnia`.

Rekurencja  
wywołania funkcji  
`silnia`



Ryc. 6.1. ilustracja działania rekurencyjnej funkcji obliczającej silnię dla liczby 4

Jeśli obliczamy wartość silni dla liczby większej od zera, wówczas po pewnej ilości odwołań funkcji do samej siebie, w wyniku kolejnego odejmowania od niej liczby 1, w końcu trafiemy na wartość argumentu równą 0. Funkcja `silnia` wywołana z argumentem  $n = 0$  wykona się natychmiast i zgodnie z treścią linii 1 przyjmie wartość 1. Teraz system kolejno może ściągać ze stosu niedokończone wywołania funkcji `silnia` i je obliczać. Rycina 6.1 przedstawia graficzną ilustrację tego procesu dla  $n$  równego 4. Największy prostokąt odpowiada pierwszemu umieszczeniu funkcji `silnia` na stosie. Ostatnim elementem umieszczonym na stosie jest wywołanie funkcji z argumentem 0, potem następuje kolejne ściąganie wywołań funkcji ze stosu wraz z jej wynikiem (pamiętaj, że wynik ten jest argumentem w obliczeniach poprzedniego wywołania funkcji).

## 6. Rekurencja

### 6.1.2. Potęga o wykładniku naturalnym liczby rzeczywistej



#### Przykład

Napiszmy funkcję obliczającą dla liczby rzeczywistej wartość potęgi o wykładniku naturalnym.

#### Potęga obliczana iteracyjnie

Kolejną funkcją o krótskiej i jasnej w zapisie definicji rekurencyjnej jest funkcja obliczająca potęgę liczby rzeczywistej. Zaczniemy jednak od rozwiązania za pomocą iteracji. Funkcja pobiera dwa argumenty: podstawa potęgi będącą liczbą rzeczywistą oraz wykładnik, który jest liczbą naturalną. W sposób iteracyjny tworzymy iloczyn liczby, dla której liczymy wartość potęgi. Założymy, że obsługę przypadku, gdy zarówno podstawa, jak i wykładnik są równe 0, pozostawiamy funkcji głównej main. Wszystkie inne wartości argumentów podanych do funkcji zostaną obsłużone w sposób poprawny przez funkcję, której kod umieszcza się poniżej:

```
float potega(float x, int n)
{
    float wynik = 1;
    for (int i=0; i<n; i++)
        wynik = wynik*x;
    return wynik;
}
```

6.3

Zauważ, że jest to proste zakodowanie definicji matematycznej:

$$\begin{aligned} a^0 &= 1 && \text{dla } a \neq 0 \\ a^1 &= a \\ a^n &= \underbrace{a \cdot \dots \cdot a}_{n-\text{czynników}} && \text{dla } n \in N_+ - \{1\} \end{aligned}$$

Potęga obliczana rekurencyjnie : To samo zadanie wykona funkcja zdefiniowana rekurencyjnie. Zapiszmy najpierw definicję rekurencyjną potęgi, a następnie zakodujmy ją w C++.

$$\begin{aligned} a^0 &= 1 && \text{dla } a \wedge 0 \\ a^n &= a^{n-1} \cdot a && \text{dla } n \in N_+ \end{aligned}$$

Zwróć uwagę, że funkcja pobiera dwa argumenty i wywołuje się rekurencyjnie ze zmodyfikowanym jednym z argumentów, przy czym drugi pozostaje bez zmian.

```
float potega(float x, int n)
{
    if (n==0) // warunek początkowy rekurencji
        return 1;
    else
        return potega(x, n-1)*x; // rekurencyjne wywołanie funkcji
}
```

6.4

Pamiętaj, że w kodzie programu musisz jeszcze zadbać o obsługę przypadku, gdy użytkownik wprowadzi 0 zarówno jako podstawę, jak i wykładnik. Funkcja nie może się wykonać z takimi argumentami (0 do połowy zerowej w matematyce nazywane jest symbolem nieoznaczonym).

### 6.1.3. Obliczanie wartości wyrazów ciągów zdefiniowanych rekurencyjnie

Tym razem zajmiemy się zapisywaniem ciągów. Z lekcji matematyki pamiętasz, że możemy to zrobić za pomocą zdefiniowania n-tego wyrazu ciągu jako funkcji zmiennej naturalnej  $n$  lub wykorzystując definicję rekurencyjną.

Załóżmy, że dany jest ciąg arytmetyczny, którego początkowe wyrazy mają wartości: 3, 5, 7, 9, 11, ... Są to więc kolejne liczby nieparzyste, a pierwszym wyrazem ciągu jest 3. Możemy zapisać wzór na ogólny wyraz ciągu:  $a_n = 2n + 1$ , gdzie  $n$  jest dodatnią liczbą naturalną, i korzystając z tego wzoru, obliczmy wartość dowolnego wyrazu. Możemy również zapisać ciąg w sposób rekurencyjny, wskazujący zależność pomiędzy kolejnymi wyrazami:

$$\begin{cases} a_1 = 3 \\ a_{n+1} = a_n + 2 \end{cases}$$

Z pomocą wzoru rekurencyjnego dość trudno byłoby nam znaleźć wartość jakiegoś konkretnie ustalonego wyrazu. Żeby policzyć na przykład wartość wyrazu czwartego, musielibyśmy policzyć wartość wyrazu trzeciego i dodać do niego 2, a z kolei aby policzyć wartość trzeciego wyrazu, musielibyśmy policzyć wyraz drugi. Aby znaleźć wyraz drugi, odnieśliśmy się już do warunku początkowego rekurencji, który informuje nas o wartości pierwszego wyrazu. Dlatego do obliczeń na kartce wygodniejszy jest wzór na ogólny wyraz ciągu, gdyż wtedy bezpośrednio obliczymy wartość wyrazu, bez konieczności znajdowania wszystkich poprzednich wyrazów. Nas jednak interesują ciągi zdefiniowane rekurencyjnie. Rozpatrzmy zatem ciąg określony wzorem:

$$\begin{cases} a_1 = 2 \\ a_n = a_{n-1} \cdot n^2 + 1 & \text{dla } n > 1 \end{cases}$$

Funkcja obliczająca wartość dowolnego  $n$ -tego wyrazu ciągu będzie miała postać:

---

```
int wartosc(int n)
{
    if (n==1)
        return 2;
    else
        return wartosc(n-1)*n*(n+1);
}
```

6.5

Funkcja rekurencyjną jest bardzo zwięzły i niemal bezpośrednim zapisem ciągu zdefiniowanego rekurencyjnie.

#### 6.1.4. Ciąg Fibonacciego

Kolejnym przykładem funkcji rekurencyjnej jest funkcja obliczająca wartość n-tego wyrazu ciągu Fibonacciego. Ciąg ten jest zdefiniowany dla każdej liczby naturalnej w następujący sposób: /(0) = 1; /(1) = 1, a każdy następny wyraz jest sumą dwóch wyrazów bezpośrednio go poprzedzających. Stąd też /(2) = 2, /(3) = 3, /(4) = 5 itd. Ogólnie więc:  $f(n) = f(n - 1) + f(n - 2)$  dla każdego argumentu większego od jeden.

$$f(n) = 1 \quad \text{dla } n \in \{0, 1\}$$

$$f(n) = f(n - 1) + f(n - 2) \quad \text{dla } n > 1$$

Oto kilka początkowych liczb ciągu Fibonacciego: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987 itd.

Fibonacci odkrył ową sekwencję liczb, analizując rozwój populacji królików w zagrodzie. Włoski matematyk chciał przewidzieć, ile królików będzie w zagrodzie po określonej liczbie miesięcy. Musiał więc przyjąć kilka założeń: u każdej płodnej pary królików nowa para rodzi się w ciągu miesiąca, para staje się płodna po miesiącu od urodzenia, króliki nie zdychają.

Ciąg Fibonacciego ma szerokie zastosowanie nie tylko w matematyce, ale również przy symulowaniu zjawisk przyrodniczych, fizycznych a nawet ekonomicznych (przewidywanie cen giełdowych).



#### Czy wiesz, że...

Liczby w ciągu Fibonacciego mają bardzo ciekawą właściwość: iloraz sąsiednich wyrazów, z wyjątkiem kilku początkowych, wynosi około 0,618 lub 1,618 (w zależności, czy dzielimy następny przez poprzedni, czy na odwrót). Liczba 1,618 jest znana w geometrii jako złota proporcja.

W odcinku podzielonym na dwie części zgodnie z zachowaniem reguł złotej proporcji większa część pozostaje w takiej samej relacji do mniejszej, jak całość do większej.

Złote proporcje od wieków bardzo chętnie wykorzystywano w architekturze. Zgodnie z zasadą złotego podziału wybudowano Wielką Piramidę w Gizie, Partenon w Atenach oraz wiele gotyckich katedr. Badacze z powodzeniem doszukują się liczb Fibonacciego i złotej liczby podziału w dziełach Mozarta i Beethovena. Podobno słynny Stradivarius korzystał ze złotego podziału podczas konstruowania swoich najlepszych wiolonczeli. U większości ludzi wysokość do pępka stanowi 0,618 łącznej wysokości, co zauważyl i na szkicował Leonardo da Vinci.

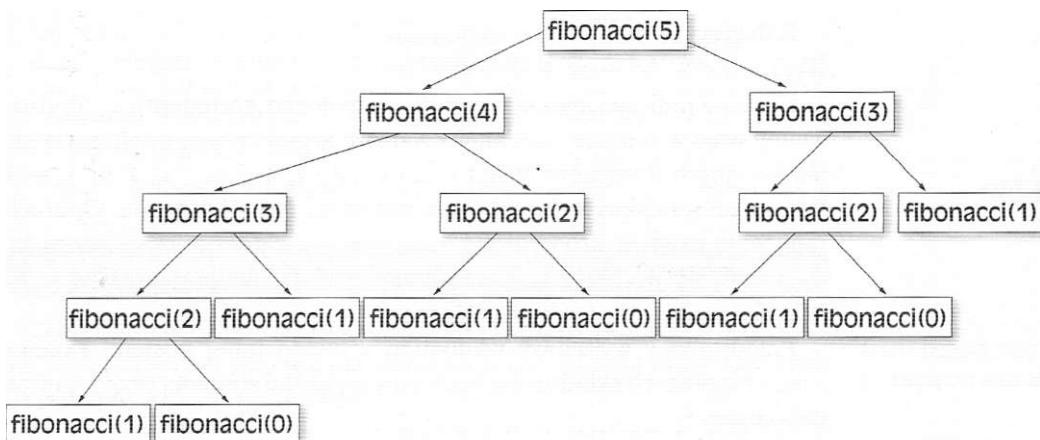
Rozrysuj drzewo genealogiczne par królików, a zauważysz, że po każdym miesiącu ich liczbę możesz wyznaczyć ze wzoru na  $n$ -ty wyraz ciągu Fibonacciego.

Definicja rekurencyjna funkcji wykonującej to zadanie jest równie krótka i czytelna, jak problemów opisanych wcześniej:

```
int fibonacci(int n)
{
    if (n<2)
        return 1;
    else
        return fibonacci(n-1)+fibonacci(n-2);
}
```

6.6

Zapisane i przeanalizowane przez nas przykłady pokazują, że rozwiązania rekurencyjne są krótkie, przejrzyste i dość proste do analizowania. Rekurencja ma jednak również pewne wady. Najczęściej rozwiązywanie problemu tą metodą trwa długo. Rozpiszmy drzewo wywołań funkcji rekurencyjnej z ostatniego przykładu (ryc. 6.2).



Ryc. 6.2. Drzewo wywołań rekurencyjnej funkcji obliczającej piąty wyraz ciągu Fibonacciego

Zauważ, że przy obliczaniu zaledwie piątego wyrazu ciągu wywołanie funkcji dla  $n$  równego na przykład 2 wykonuje się aż trzy razy. A zatem ilość obliczeń jest duża, o wiele większa niż przy realizacji funkcji zdefiniowanej nierekurencyjnie, opartej jedynie na iteracjach. Czasowa złożoność obliczeniowa rozwiązania rekurencyjnego jest wykładnicza, a więc wyjątkowo niska. Tak samo jest ze złożonością pamięciową - już przy niewielkich wartościach  $n$  może nastąpić przepelenie stosu. Złożoność pamięciowa rozwiązania rekurencyjnego jest wykładnicza. Nie zawsze jednak funkcje rekurencyjne są nieoptymalne - przykłady roz-

| Drzewo wywołań rekurencyjnych

## 6. Rekurencja

wiązań rekurencyjnych, których zaletą jest właśnie szybkość działania, poznasz w następnych podrozdziałach.

Wyrazy ciągu Fibonacciego możemy obliczać za pomocą programu z zastosowaniem jedynie pętli - sposób ten jest znacznie szybszy od rekurencyjnego; spróbuj samodzielnie napisać taki program. Rozwiążanie iteracyjne ma liniową złożoność obliczeniową. Złożoność pamięciowa jest minimalna - wystarczy zapamiętywać dwie ostatnie wartości ciągu (jest to więc złożoność stała).

### 6.1.5. Schemat Homera obliczania wartości wielomianu

Schemat Homera jest najszybszym sposobem obliczenia wartości wielomianu, pozwalającym na znaczne zredukowanie liczby mnożeń w obliczeniach.



#### Przykład

Obliczmy rekurencyjnie wartość wielomianu dla podanego argumentu.

*Załóżmy, że mamy dany wielomian:*

Cheemy policzyć jego wartość dla dowolnego argumentu  $x_0$ . Podstawiamy więc w miejsce zmiennej  $x$  wartość argumentu  $x_0$  i obliczamy sumę kolejnych iloczynów:  $W(x_0) = 2 \cdot x_0 \cdot x_0 \cdot x_0 + 3 \cdot x_0 \cdot x_0 + 5 \cdot x_0 + 4$ . W tym celu musimy wykonać sześć mnożeń i trzy dodawania. Operacja mnożenia zajmuje procesorowi dużo czasu, a zatem dobrze byłoby zastosować taki algorytm, który zredukuje liczbę tych operacji. Takim właśnie algorytmem jest schemat Homera.

Opis metody na podstawie przykładu

Przedstawmy wyjściowy wielomian w nieco innej postaci. Kolejno z jego pierwszych składników będziemy wyłączać zmienną przed nawias, uzyskując:

$$\begin{aligned}W(x) &= x \cdot (2 \cdot x^2 + 3 \cdot x + 5) + 4 \\W(x) &= x \cdot (x \cdot (2 \cdot x + 3) + 5) + 4\end{aligned}$$

Aby obliczyć z końcowej postaci wielomianu jego wartość, wykonamy już tylko trzy mnożenia i trzy dodawania. Oszczędność liczby wykonywanych mnożeń rośnie wraz ze wzrostem stopnia wielomianu. Dla wielomianu 77-tego stopnia w zwykłej postaci należy wykonać  $\frac{n \cdot (n+1)}{2}$

mnożeń, a dla wielomianu po zastosowaniu schematu Homera tylko  $n$  mnożeń. Zatem dla dużych  $n$  różnica jest naprawdę spora.

Funkcję, którą omawiamy, moglibyśmy zapisać zarówno w sposób iteracyjny, jak i rekurencyjny. Nas interesuje oczywiście ten drugi sposób.

Funkcja rekurencyjną, która wykorzystuje schemat Homera, jak na funkcję rekurencyjną przystało, jest zdefiniowana krótko i czytelnie:

```
float horner(int k, float tablica_wspolczynnikow[], float x)
{
    if (k==0)
        return tablica_wspolczynnikow[0]*x;
    else
        return horner(k-1,tablica_wspolczynnikow,x)*x+tablica_wspolczynnikow[k];
}
```

6.7

Funkcja pobiera trzy parametry: stopień wielomianu, tablicę współczynników wielomianu i wartość argumentu, dla którego oblicza wartość wielomianu. Przykładowo, dla wielomianu  $W(x) = 4x^3 + 5x^2 + 3x + 2$  tablica współczynników ma postać: [4, 5, 3, 2],

Schemat Homera również możemy skonstruować iteracyjnie, bez użycia rekurencji. Kod funkcji jest równie krótki jak w metodzie rekurencyjnej:

Iteracyjna funkcja realizująca schemat Homera

```
float horner_iteracyjnie(int k, float tablica_wspolczynnikow[], float x)
{
    int i;
    float wartosc = tablica_wspolczynnikow[0];
    for (i=1; i<k+1; i++)
        wartosc = wartosc*x+tablica_wspolczynnikow[i];
    return wartosc;
}
```

6.8

### 6.1.6. Wieże Hanoi

Nazwa: wieże Hanoi oznacza zarówno klasyczny problem o rozwiązyaniu rekurencyjnym, jak i grę dla starszych dzieci. Przedstawmy więc krótko problem, którym się zajmiemy.

#### Przykład

Mamy trzy paliki. Na pierwszy z nich wsunięto n krążków od największego do najmniejszego (ryc. 6.3). Zadanie polega na przeniesieniu wszystkich krążków z pierwszego palika na trzeci. Przestrzegamy przy tym dwóch zasad: zawsze nakładamy mniejsze krążki na większe, a z palika możemy zdjąć jednorazowo tylko jeden krążek. W razie potrzeby korzystamy z drugiego palika.

Przedstawmy na początek rozwiązanie dla małych liczb krążków. Oznaczmy nasze paliki jako: A (palik początkowy), B (palik pomocniczy), C (palik końcowy).

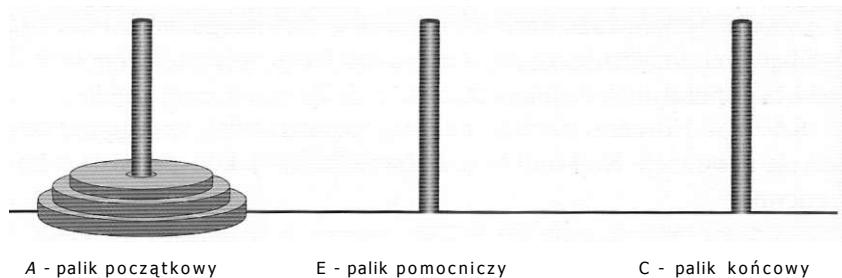
## 6. Rekurencja

Opis metody dla jednego, dwóch i trzech krążków

Jeśli mamy tylko jeden krążek na paliku, to po prostu przenosimy go na palik C i zadanie jest rozwiązane. Przy dwóch krążkach wykonamy ruchy: przeniesiemy mniejszy z krążków z palika<sup>A</sup> na B, następnie większy z palika A na C, a na końcu mniejszy krążek z palika B na C. Zadanie zostało wykonane. Całe rozwiązanie zadania możemy zapisać:

```
przenies(A, B);
przenies(A, C);
przenies(B, C);
```

Przeanalizujmy teraz sposób przenoszenia trzech krążków.



Ryc. 6.3. Stan początkowy problemu wież Hanoi

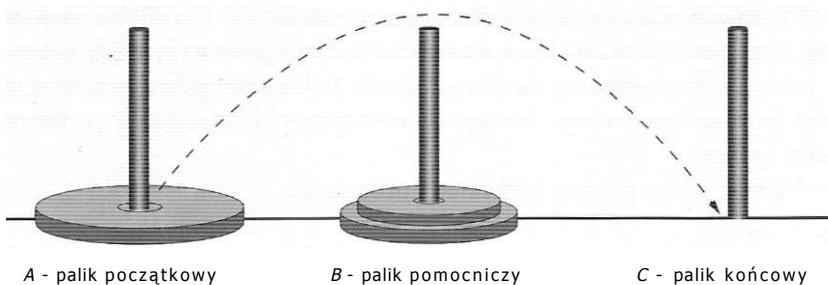
Proponujemy wycięcie z papieru trzech krążków o malejących długościach promienia i umieszczenie ich na kartce z narysowanymi trzema palikami. Krążki ustawić najpierw w ten sposób, aby przedstawiały sytuację początkową, czyli wszystkie trzy krążki umiejscowione od największego na dole do najmniejszego na górze na paliku początkowym A. Teraz przenieś krążki na palik końcowy, oczywiście z zachowaniem zasad, starając się wykonać jak najmniejszą liczbę ruchów. Porównaj swój sposób postępowania z naszą propozycją poniżej:

```
A->C
A->B
C->B
A->C
B->A
B->C
A->C
```

Podaną przez nas metodą wykonasz zadanie najmniejszą liczbą ruchów. Ogólnie metodę tę możemy opisać w następujący sposób:

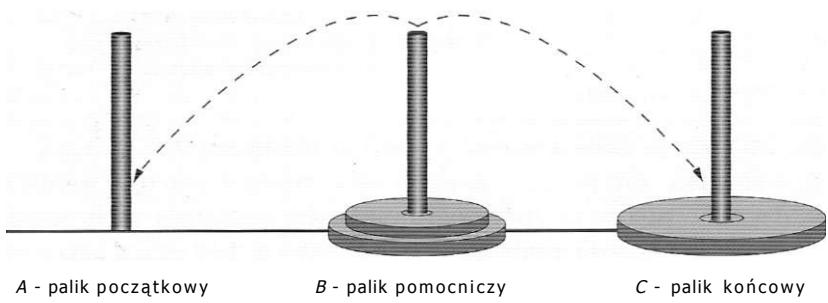
1. Przeniesienie dwóch krążków z palika początkowego na pomocniczy z użyciem palika końcowego (ryc. 6.4 na stronie obok).
2. Przeniesienie największego krążka z palika początkowego na końcowy (ryc. 6.4).

## 6.1. Funkcje rekurencyjne w informatyce

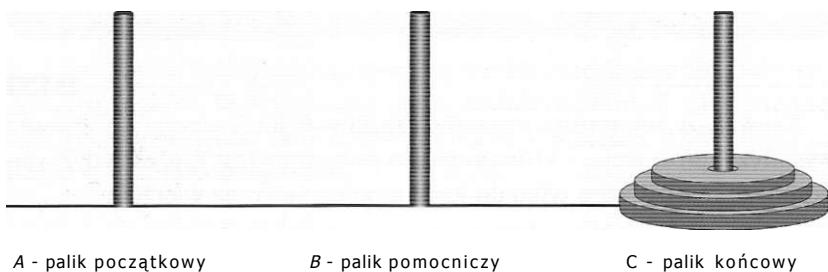


Ryc. 6.4. Sytuacja po przeniesieniu dwóch krążków na palik pomocniczy

3. Przeniesienie dwóch krążków z palika pomocniczego na końcowy, z użyciem palika początkowego (ryc. 6.5 i 6.6).



Ryc. 6.5. Sytuacja po przeniesieniu największego krążka na palik końcowy



Ryc. 6.6. Sytuacja po przeniesieniu dwóch krążków z palika pomocniczego na palik końcowy; koniec zadania

Taka zasada będzie też obowiązywać dla dowolnej liczby krążków:  
przenies  $n-1$  krazkow z palika A na B  
przenies ostatni z krazkow z A na C  
przenies  $n-1$  krazkow z B na C

## 6. Rekurencja

Przeniesienie  $n$  krążków oznacza przeniesienie  $n-1$  krążków i jednego, przeniesienie z kolejnych  $n-1$  krążków to przeniesienie  $n-2$  krążków i jednego, aż dojdziemy do przeniesienia tylko jednego krążka. To już jest prawie konstrukcja funkcji rekurencyjnej rozwiązującej problem wież Hanoi.

Umieszczamy poniżej kod całego programu:

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

void hanoi(int n, char a, char c, char b) // funkcja rekurencyjna
{
    if (n==1)
        cout << a << " -> " << c << endl;
    else
    {
        hanoi(n-1,a,b,c);
        cout << a << " -> " << c << endl;
        hanoi(n-1,b,c,a);
    }
}

int main()
{
    cout << "Ile krazkow jest na poczatkowym paliku? ";
    int ile;
    cin >> ile;
    hanoi(ile,'A','C','B');
    cin.ignore();
    getchar();
    return 0;
}
```

6.9

Zauważ, że nie musisz oznaczać krążków w jakiś szczególny sposób: wystarczy informacja, z którego palika pobierany jest krążek, gdyż wiadomo, że masz dostęp tylko do krążka położonego na wierzchu.

### 6.1.7. Zamiana liczby na postać dwójkową - rozwiązanie rekurencyjne

Jeszcze raz powróćmy do zadania zamiany liczby z postaci dziesiątkowej na liczbę w systemie o innej podstawie liczenia, na przykład o podstawie wynoszącej 2. W metodzie iteracyjnej, jak zapewne pamiętasz, musielibyśmy użyć tablicy do przechowywania poszczególnych cyfr rozwiązania. Rozmiar tej tablicy stanowił ograniczenie wielkości liczby, jaką mogliśmy przeliczyć. Przy rozwiązyaniu rekurencyjnym nie musimy zakładać żadnych ograniczeń wartości liczby, którą będziemy zamieniać na

system dwójkowy, ponieważ nie używamy tablicy. Jedyne ograniczenia wynikają z reprezentacji liczb w komputerze oraz rozmiaru pamięci operacyjnej,<sup>^</sup> konkretnie stosu.

```
void dwojkowa(int liczba)
{
    if (liczba>=2)
        dwojkowa(liczba/2);
    cout << liczba%2;
}
```

Jeśli chcemy, aby funkcja była uniwersalna i zamieniała liczbę na jej postać w systemie o dowolnej podstawie liczenia, musimy podać jeszcze jeden parametr - podstawę systemu liczenia:

```
void zamiana(int liczba, int podstawa)
{
    if (liczba>=podstawa)
        zamiana(liczba/podstawa,podstawa);
    cout << liczba%podstawa;
}
```

Uniwersalna funkcja rekurencyjnej zamiany liczb pomiędzy systemami

**6.10**

Zauważ, że w ten sposób te reszty z dzielenia, które są uzyskane jako ostatnie, zostaną wypisane jako pierwsze i na odwrót. Zapamiętaj tę konstrukcję algorytmu rekurencyjnego, warto ją wykorzystać, gdy na przykład trzeba będzie wypisać dane w odwrotnej kolejności.

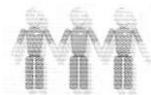
Przypominamy, że również na tym etapie, jeśli chcesz wykorzystać podstawę liczenia większą od 10, musisz wprowadzić dodatkowe warunki, które zamieniają 10 na literę A, 11 na literę B i tak dalej.

### 6.1.8. Rekurencyjne odwracanie wprowadzonego ciągu znaków

Zajmijmy się zadaniem wypisywania w odwrotnej kolejności pobranego ciągu znaków. Wykorzystamy do tego własność funkcji rekurencyjnej zawartą w algorytmie zamieniającym liczbę w systemie dziesiętnym na liczbę w systemie dwójkowym.

#### Przykład

Pobrany z klawiatury ciąg znaków o nieznanej początkowo długości wypiszemy w odwrotnej kolejności aniżeli kolejność wprowadzania znaków.



W celu rozwiązania tego zadania z zastosowaniem iteracji trzeba kolejne znaki wprowadzić do tablicy, a następnie odczytać tablicę od ostatniej użytej do zapisu komórki aż do komórki pierwszej. Problemem jest jednak zdefiniowanie potrzebnej tablicy. Nie wiemy, jakiej długości ciąg znaków zostanie wprowadzony, nie wiemy zatem, jakiej długości tablicę przygotować. Albo przygotujemy tablicę zbyt krótką i nie zmieścimy ciągu, albo zdecydowanie za długą, niepotrzebnie blo-

## 6. Rekurencja

kując pamięć. Nie jest to więc algorytm uniwersalny, gdyż nie obsługuje wszystkich przypadków.

Ten problem rozwiązuje metoda rekurencyjna. Funkcja, którą prezentujemy, jest na tyle prosta i czytelna, że zinterpretujesz ją samodzielnie:

```
void odwroc()
{
    char znak;
    cout << "Podaj znak: ";
    cin >> znak;
    if (znak != '.')
    {
        odwroc();
        cout << znak;
    }
}
```

6.11

Mechanizm działania funkcji rekurencyjnej opisaliśmy już dokładnie przy omawianiu silni, tu w podobny sposób zostają odkładane na stos kolejno wprowadzane znaki. Stąd też, przy spełnieniu warunku kończącego wywołania rekurencyjne, gdy wprowadzona zostanie kropka, wszystkie znaki zostaną wypisane, ale w kolejności od ostatniego do pierwszego.

### 6.2. Metoda „dziel i zwyciężaj”

„Dziel i zwyciężaj” to ogólna nazwa metod polegających na podziale danego problemu na skończoną liczbę mniejszych problemów (faza „dziel”), znalezieniu ich rozwiązań (faza „zwyciężaj”), a następnie połączeniu ich z powrotem w jedną całość (faza „połącz”). Zwykle mniejsze problemy wciąż reprezentują ten sam wyjściowy problem, ale są rozpatrywane dla danych o mniejszym rozmiarze, stąd rozwiązywanie ich jest prostsze. Ponieważ często mniejszy problem jest tożsamym z problemem wyjściowym, wiele algorytmów rozwiązuje się w sposób rekurencyjny w połączeniu z metodą „dziel i zwyciężaj”. Metoda ta charakteryzuje się dużą szybkością.

Opis metody  
"dziel i zwyciężaj"

Na ogół sposób postępowania metodą „dziel i zwyciężaj” składa się, jak to już wspominaliśmy, z trzech etapów:

**Dziel** - dzielimy początkowy problem na mniejsze podproblemy,

**Zwyciężaj** - rozwiązuje my podproblemy (zwykle rekurencyjnie),

**Połącz** - łączymy rozwiązania wszystkich podproblemów w jedną całość, aby uzyskać rozwiązanie całego problemu.

Algorytmy oparte na metodzie „dziel i zwyciężaj” zazwyczaj charakteryzują się niską złożonością obliczeniową. Są zadania, które mają bardziej optymalne rozwiązania oparte na innych metodach, ale są też ta-

kie, których inaczej rozwiązać się nie da. Korzystając z tej metody, można zaprojektować algorytm przeszukiwania ciągu w celu znalezienia wyróżnionego elementu, można też w szybki sposób posortować ciągi.

### 6.2.1. Przeszukiwanie binarne

Przeszukiwanie binarne, zwane inaczej połówkowym, jest metodą szukania wyróżnionego elementu w już posortowanym ciągu. Zamiast zaczynać od pierwszego elementu i przeglądać tablicę aż do znalezienia elementu szukanego lub dotarcia do jej końca, przeglądanie zaczyna się od środkowego elementu tablicy, który porównywany jest z szukanym elementem (nie zawsze będzie to faktycznie element leżący dokładnie pośrodku; jeśli liczba elementów w ciągu jest parzysta, możemy określić dwa elementy jako środkowe - wtedy założymy, że środkowym elementem jest element bliższy początkowi ciągu). Jeżeli środkowy element jest mu równy, przeszukiwanie zostaje zakończone. Jeżeli jest on większy niż szukany element, to odrzuca się górną połowę tablicy i ogranicza się poszukiwanie do elementów tablicy położonych poniżej środka. Jeżeli zaś środkowy element jest mniejszy od szukanego elementu, odrzuca się dolną połowę i kontynuuje poszukiwanie jedynie w górnej części tablicy.

Założymy, że szukanym elementem jest liczba 7, a ciąg, w którym szukamy, ma postać:

1 3 6 7 9 13 14 14 17 21 23 25 28 31 34 35 45 46 76 89

Sprawdzamy wartość środkowego elementu tablicy (tu ma wartość 21). Jest ona większa niż 7, a więc odrzucamy górną część tablicy i będziemy kontynuować poszukiwanie w nowym ciągu:

1 3 6 7 9 13 14 14 17

Teraz porównamy środek nowej tablicy (o wartości 9) z liczbą 7. Jest on od 7 większy, zatem zawężamy obszar poszukiwań do tablicy:

1 3 6 7

Porównujemy szukany element z liczbą 3, w wyniku czego zostaje nam do przeszukania tablica: 6 7.

Kolejne porównanie z liczbą 6 sprawia, że ostatecznie do przeszukania zostaje tablica jednoelementowa. Jedynym jej elementem jest 7, a zatem element został odnaleziony.

Zwróci uwagę, że może się odbyć tylko jedno porównanie elementu poszukiwanego ze środkowym elementem ciągu, gdy element ten znajduje się właśnie w środku tablicy. Tak byłoby na przykład, gdybyśmy szukali liczby 3 w ciągu: 1 2 3 4 5 6.

Poniżej prezentujemy funkcję rekurencyjną realizującą problem przeszukiwania binarnego. Jeśli element zostanie znaleziony, funkcja przy-

Opis metody  
na podstawie  
przykładu

## 6. Rekurencja

muje wartość numeru indeksu elementu, który jest pierwszym napotkanym szukanym elementem. W przeciwnym wypadku wynik działania funkcji wynosi -1.

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

int binarne(int poczatek, int koniec, int tab[], int szukany)
{
    if (poczatek<=koniec)
    {
        int srodek = (poczatek+koniec)/2;
        if (tab[srodek]==szukany)                                // element znaleziony
            return srodek;
        if (tab[srodek]>szukany)
            return binarne(poczatek,srodek,tab,szukany);
        else
            return binarne(srodek+1,koniec,tab,szukany);
    }
    return -1;                                              // element nie występuje w tablicy
}

int main()
{
    int tab[6] = {1,2,3,4,5,6};
    int szuk;
    cout << "Jakiego elementu szukasz? ";
    cin >> szuk;
    if (binarne(0,5,tab,szuk)==-1)
        cout << "Elementu nie ma w tablicy";
    else
        cout << "Element odnaleziono na pozycji " << binarne(0,5,tab,szuk)+1;
    cin.ignore();
    getchar();
    return 0;
}
```

6.12

Oczywiście, algorytm poszukiwania elementu przez połowienie przedziału można zrealizować również iteracyjnie, nie tylko metodą rekurencyjną.

### 6.2.2. Sortowanie tablicy przez scalanie

Na funkcjach rekurencyjnych i metodach „dziel i zwyciężaj” oparte są metody sortowania. Spośród nich najpowszechniej używane jest sortowanie szybkie i sortowanie przez scalanie (ang. *merge sort*), szczególnie dobrze nadające się do sortowania struktur, do których nie ma swobodnego dostępu, a więc na przykład plików.

Sortowanie przez scalanie polega na podzieleniu sortowanej struktury na dwie części, ich rekurencyjnym posortowaniu oraz scaleniu już posortowanych kawałków z powrotem w jedną całość. Algorytm ten został po raz pierwszy zaproponowany przez Johna von Neumanna w 1945 roku. Algorytm ma złożoność obliczeniową klasę  $\Theta(n \log n)$ . Wadą tego typu sortowania jest konieczność zapewnienia programowi dodatkowego obszaru pamięci o wielkości równej sortowanemu zbiorowi.

#### Czy wiesz, że...

John von Neumann (1903-1957), amerykański matematyk i informatyk pochodzenia węgierskiego, od wczesnych lat wykazywał zdumiewające zdolności pamięciowe: w wieku sześciu lat potrafił wykonać w pamięci dzielenie liczb ośmiocyfrowych. Wniósł znaczący wkład w rozwój logiki matematycznej, teorii liczb i teorii mnogości. Książką *Teoria gier i zachowania ekonomicznego* położył podwaliny pod rozwój teorii gier. Brał także udział w projektach budowy bomby atomowej i wodorowej. W 1946 roku rozpoczął projekowanie komputera, który wykorzystywał program przechowywany w pamięci. Dzisiaj niemal wszystkie popularne komputery klasy PC mają strukturę stworzoną przez tego uczonego, stąd noszą nazwę von Neumanna.



Przeanalizujmy po kolejny każdy krok prowadzący do posortowania nieuporządkowanej tablicy. Na początku dzielimy tablicę na dwie równe części (będziemy je nazywać podciągami) lub, jeśli liczba jej elementów jest nieparzysta, na dwie części, z których jedna ma o jeden element więcej niż druga. Aby obie części posortować, znów dzielimy każdą z nich na dwie części, które dalej będziemy dzielić, aż każda z części - podciągów będzie miała co najwyżej jeden element. Podciąg jednoelementowy jest posortowany. Teraz będziemy scalać posortowane podciągi. Scalanie dwóch podciągów polega na tym, że porównujemy ich pierwsze elementy i mniejszy z nich usuwamy z podciągu, a wstawiamy do nowego ciągu. Postępujemy tak, aż obydwa podciągi pozostaną puste.

Opis metody sortowania przez scalanie

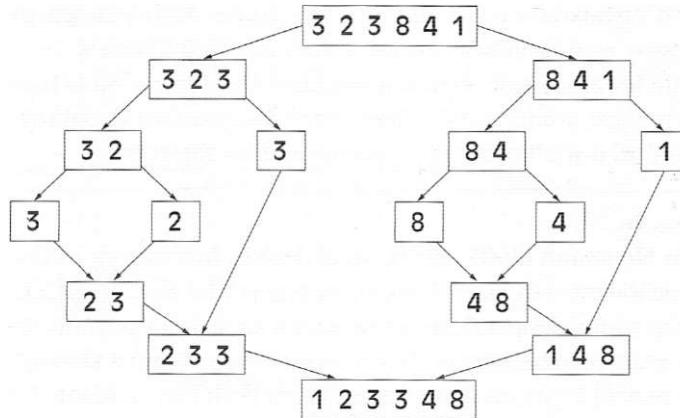
Jeśli oznaczymy ciąg do posortowania jako  $a_n = (a_{lewy}, a_{lewy+1}, \dots, a_{prawy})$ , to sposób postępowania przy sortowaniu przez scalanie możemy zapisać za pomocą listy kroków:

1. Jeśli  $lewy < prawy$ , przypisz:  $środek := (lewy + prawy) \ div 2$  i przejdź do kroku 2.
2. Zastosuj ten algorytm dla  $(a_n, lewy, środek)$  i przejdź do kroku 3.
3. Zastosuj ten algorytm dla  $(a_n, środek + 1, prawy)$  i przejdź do kroku 4.
4. Scal ciągi  $(a_{lewy}, \dots, a_{środek})$ ,  $(a_{środek + 1}, \dots, a_{prawy})$ .

## 6. Rekurencja

Przykładowe zastosowanie metody sortowania przez scalanie

W zrozumieniu tej metody powinien dopomóc rysunek obrazujący ideę sortowania przez scalanie:



Ryc. 6.7. Sortowanie przez scalanie

Zauważ, że na rysunku łatwo jest rozróżnić dwie fazy działania algorytmu: rozbijanie tablicy na podtablice, a następnie scalanie ich na powrót w jedną całość, już posortowaną. Musimy zatem napisać dwie funkcje: rekurencyjną sortowania przez scalanie oraz funkcję scalającą elementy tablicy.

Poniżej prezentujemy funkcję rekurencyjną sortowania przez scalanie:

```
void MergeSort(int tab[], int lewy, int prawy)
{
    int srodek;
    srodek = (lewy+prawy)/2;
    if (lewy<srodek)
        MergeSort(tab, lewy, srodek);
    if (srodek+1<prawy)
        MergeSort(tab, srodek+1, prawy);
    Merge(tab, lewy, prawy);
}
```

6.13

Lewy i prawy to indeksy wyrazów: początku i końca sortowanej podtablicy - przy pierwszym wywołaniu funkcji będzie to 0 i pomniejszona o 1 liczba elementów tablicy; srodek oznacza indeks wyrazu, który teraz będzie wskazywał ostatni element pierwszego podcięgu, a zwiększonego o 1 - pierwszy wyraz podcięgu drugiego. Dla podcięgów zostanie rekurencyjnie wywołana funkcja MergeSort.

Aby posortować na przykład dziesięcioelementową tablicę o nazwie tab, wywołamy funkcję rekurencyjną w następujący sposób: `MergeSort(tab, 0, 9)`.

Funkcja Merge (scal) ma za zadanie połączenie dwóch już posortowanych części tablicy. Porównywać więc będzie ich pierwsze elementy

**i** - zgodnie z wynikiem porównania - elementy mniejsze ustawi przed większymi aż do momentu, gdy pobierze wszystkie elementy jednej z tablic. Wtedy dopisane zostaną pozostałe elementy drugiej tablicy.

Funkcja **Merge** działa więc następująco:

- 1) dopóki żaden ze scalonych ciągów *a* i *b* nie jest pusty, porównuj ze sobą pierwsze elementy każdego z nich; w zbiorze wynikowym *c* umieszczaj mniejszy z elementów, usuwając go jednocześnie ze scalanego ciągu; jeśli obydwa porównywane elementy są sobie równe, do zbioru wynikowego wstaw ten drugi (w rzeczywistości nie ma znaczenia, który wstawisz);
- 2) do końca zbioru *c* dopisz zawartość tego scalonego ciągu *a* lub *b*, który zawiera jeszcze elementy.

Przykładowo, jeśli ciąg *a* = (1, 2, 4), ciąg *b* = (1, 4, 7), to ciąg *c* = (1, 1, 2, 4, 4, 7).

Oto pełny kod programu, w którym użyto funkcji scalającej **Merge** i rekurencyjnej funkcji **MergeSort**:

```
#include <iomanip>
#include <cstdlib>
#include <iostream>
#include <cstdio>
using namespace std;

const int N = 19;      // liczba elementów w sortowanym zbiorze

// *****
// Procedura scalania (Merge)
// *****

void Merge(int d[], int pocz, int kon)
{
    int i, i1, i2, sr;
    int temp[N];           // tablica pomocnicza, do niej przepiszymy elementy
                           // tablicy d
    for (i=0; i<N; i++)
        temp[i] = d[i];
    sr = (pocz+kon)/2;
    i = pocz;              // indeks "wędrujący" po tablicy d
    i1 = pocz;              // indeks "wędrujący" po pierwszej połówce tablicy temp
    i2 = sr+1;              // indeks "wędrujący" po drugiej połówce tablicy temp
    while (i1<=sr & i2<=kon)
    {
        if (temp[i1]<temp[i2])
        {
            d[i] = temp[i1];
            i1++;
        }
        else
        {
            d[i] = temp[i2];
            i2++;
        }
        i++;
    }
}
```

## 6. Rekurencja

```
}

if (il>sr)
    while (i2<=kon)
    {
        d[i] = temp[i2];
        i2++; i++;
    }
else
    while (il<=sr)
    {
        d[i] = temp[il];
        il++; i++;
    }
}

/******************
** Procedura sortowania przez scalanie **
******************/

void MergeSort(int d[], int pocz, int kon)
{
    int sr;
    sr = (pocz+kon)/2;
    if (pocz<sr)
        MergeSort(d,pocz,sr);
    if (sr+1<kon)
        MergeSort(d,sr+1,kon);
    Merge(d,pocz,kon);
}

/******************
** Program główny **
******************/

int main()
{
    int d[N],i;
    // generujemy zbiór do sortowania wypełniony
    // liczbami pseudolosowymi z zakresu od 0 do 99
    srand((unsigned)time(NULL));
    cout << "Przed sortowaniem:\n\n";
    for (i=0; i<N; i++)
    {
        d[i] = rand()%100;
        cout << setw(4) << d[i];
    }
    cout << endl << endl;
    // wywołujemy procedurę sortowania przez scalanie
    MergeSort(d,0,N-1);
    // zbiór został posortowany; wypisujemy wyniki
    cout << "Po sortowaniu:\n\n";
    for (i=0; i<N; i++)
        cout << setw(4) << d[i];
    cout << endl << endl;
    getchar();
    return 0;
}
```

6.14

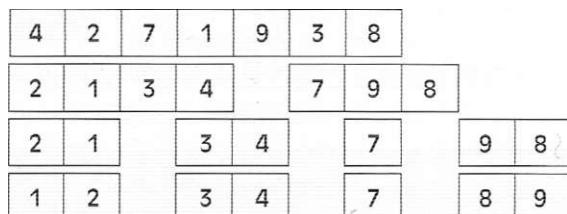
www.operon.pl

### 6.2.3. Sortowanie szybkie

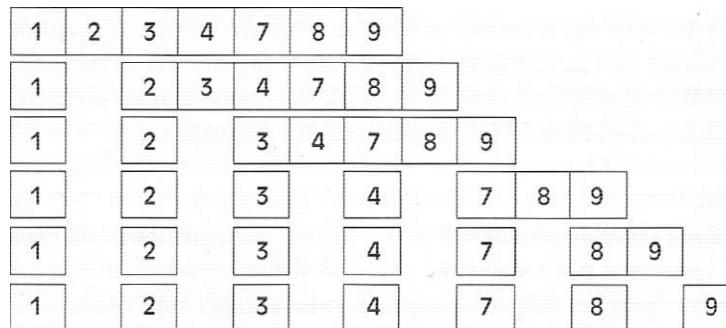
Jedną z najszybszych metod sortowania jest sortowanie szybkie (ang. *quicksort*) o logarytmiczno-liniowej złożoności obliczeniowej  $O(n \log n)$ , chociaż w pewnych niekorzystnych przypadkach (np. dane odwrotnie posortowane) algorytm ten może mieć złożoność klasy  $O(n^2)$ . W algorytmie tym nie ma etapu scalania posortowanych części ciągu, ponieważ działa on w miejscu, to znaczy nie potrzebuje do działania dodatkowej tablicy, jak to było w wypadku sortowania przez scalanie. Zasada sortowania szybkiego oparta jest na następującym mechanizmie postępowania:

- wybieramy element ciągu (najczęściej jest to pierwszy element ciągu, ale może to być również element wybrany losowo, leżący pośrodku ciągu lub wskazany inną, dowolną metodą);
- dzielimy ciąg na dwa podciągi: w lewym znajdują się elementy nie większe niż wybrany element, a w prawym - nie mniejsze;
- dopóki długość podciągu jest większa od 1, wywołujemy dla niego tę samą procedurę sortującą.

Efektywność i szybkość działania tej metody zależy od długości podciągów, na które podzielony został ciąg w każdym z kolejnych wywołań funkcji rekurencyjnej. Jeśli podciągi są mniej więcej równej długości, sortowanie odbywa się szybko. Efektywność metody będzie mała, gdy zastosujemy ją do ciągu już uporządkowanego. Zauważ bowiem, jak wygląda podział ciągu o elementach losowych na kolejne podciągi (ryc. 6.8), a jak wygląda w wypadku, gdy chcemy posortować ciąg już posortowany (ryc. 6.9).



Ryc. 6.8. Podział losowych elementów ciągu względem pierwszego elementu



Ryc. 6.9. Podział posortowanego ciągu względem pierwszego elementu - widoczna skrajna asymetria tego rodzaju podziału

Opis metody sortowania szybkiego

Przykładowe zastosowania sortowania szybkiego

## 6. Rekurencja

Jak widzisz, liczba wykonywanych operacji w ciągu już posortowanym wyraźnie wzrosła. Ta różnica i jej wpływ na szybkość działania algorytmu zwiększa się wraz ze wzrostem liczby elementów sortowanego ciągu.

Podobnie jak w metodzie sortowania przez scalanie funkcja sortowania szybkiego pobiera jako argumenty tablicę do posortowania oraz indeksy jej pierwszego i ostatniego elementu.

```
void quicksort(int tablica[], int x, int y)
{
    int i, j, temp, v;
    i = x;
    j = y;
    v = tablica[x] ;
    do
    {
        while (v>tablica[i]) i++;           // szukam większego lub równego
        while (v<tablica[j]) j--;           // szukam mniejszego lub równego
        if (i<=j)
        {
            temp = tablica[i];
            tablica[i] = tablica[j];
            tablica[j] = temp;
            i++;
            j--;
        }
    }
    while (i<=j);
    if (x<j) quicksort(tablica,x,j);
    if (i<y) quicksort(tablica,i,y);
}
```

6.15

Pieny kod programu umieściliśmy na płycie CD dołączonej do podręcznika **6.16**

### 6.3. Problem skoczka szachowego I problem ośmiu hetmanów - algorytmy z powrotami

Oba problemy - skoczek szachowy i ośmioletni - realizuje się za pomocą tak zwanych algorytmów z powrotem (zwanych również algorytmami z nawrotami). Są one często wykorzystywane w problemach kombinatorycznych. Omówmy zarówno same zadania, jak i algorytmy służące do ich realizacji, a łatwo zrozumiesz, czym jest rekurencja z powrotem.



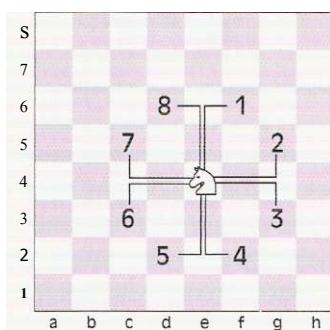
#### Przykład

Mamy daną szachownicę kwadratową, której rozmiar dowolnie ustalamy. Na jednym z pól ustawiamy skoczkę szachowego. Badamy, czy skoczek może przejść całą szachownicę, odwiedzając tylko jeden raz każde z pól.

### 6.3. Problem skoczka szachowego i problem ośmiu hetmanów - algorytmy z powrotami

Checemy skonstruować algorytm, który w przypadku, gdy jest to możliwe, poda nam rozkład ruchów skoczka, a w przypadku, gdy jest to niemożliwe, wypisze odpowiedni komunikat. Przypomnijmy najpierw sposób poruszania się skoczka. Figura ta wykonuje ruchy, które składają się z sumy przesunięć: dwa pola po prostej, na której jest umieszczony, oraz jedno pole w kierunku prostopadłym do prostej, na której nastąpiło pierwsze z przesunięć. A zatem maksymalnie skoczek ma osiem możliwości wykonania ruchu, zgodnie ze schematem przedstawionym na rycinie 6.10.

Opis metody  
poruszania się  
skoczka po planszy



Ryc. 6.10. Możliwe ruchy skoczka

Załóżmy, że realizujesz to zadanie, siedząc nad rozłożoną szachownicą. Zapewne najpierw stawiasz skoczka na jakiś wstępnie wybrany polu, a potem, ustaliwszy kolejność próbnych ruchów, przechodzisz na kolejne pola, zaznaczając te, które skoczek odwiedził. W wypadku, gdy któryś z ruchów nie daje już możliwości wykonania następnego, wracasz do ruchu poprzedniego, anulując poprzedni ruch, i wykonujesz inny możliwy dla skoczka ruch. W ten sposób badasz, czy kolejne ruchy doprowadzą do obejścia całej planszy. Czasami okazuje się, że musisz się cofnąć nie o jeden, ale o więcej ruchów. W ten sposób albo uda ci się przejść całą planszę, albo po wykorzystaniu wszelkich możliwości uznasz, że zadania nie da się zrealizować.

Zauważ, że dla całego postępowania kluczowe jest wielokrotne wycofywanie się z błędного ruchu - właśnie owe „powroty” będą charakterystyczne dla algorytmów z powrotami. Zapiszmy więc algorytm, który rozwiąże nasz przykład. W celu realizacji zadania znalezienia drogi przejścia skoczka przez szachownicę ustalamy kolejność prób wykonywania ruchów jak na rycinie 6.10. A zatem skoczek wykonuje kolejne ruchy, począwszy od zdefiniowanego liczba 1, przez następne, jeśli okaże się, że/dany ruch jest niemożliwy do wykonania, gdyż skoczek znajdzie się na polu, które już odwiedził.

Przedstawmy kilka pierwszych ruchów skoczka na szachownicy o wymiarach 4x4. Skoczek postępuje według zasady: jeśli może wykonać ruch

## 6. Rekurencja

zdefiniowany jako 1, wykonuje go, po czym znów chce wykonać ruch 1. Jeśli się da, to go wykonuje, jeśli nie, próbuje wykonać ruch zdefiniowany jako 2. W ten sposób skoczek wykonał zaledwie trzy ruchy, co pokazano na rycinie 6.11. Z ostatniego, trzeciego pola nie da się już jednak wykonać żadnego ruchu, a nie jest prawdą, że skoczek obszedł całą planszę.



Ryc. 6.11. Przykładowa kolejność ruchów skoczka umieszczonego na szachownicy  $4 \times 4$ . Reszty pól nie da się obejść.

Czy to oznacza, że zadanie jest niewykonalne? Niekoniecznie. Może gdyby wcześniej skoczek wykonał jakiś inny ruch, niezgodny z zasadą wykonywania ruchów od pierwszego zdefiniowanego, to mógłby przejść całą planszę. W tym miejscu wykonamy zatem krok powrotu algorytmu do poprzedniego pola, z którego skoczek miał jeszcze inną możliwość wykonania ruchu. Wracamy na pole oznaczone na rycinie 6.11 numerem 2 i kontynuujemy przechodzenie planszy (ryc. 6.12). W ten sposób przechodzimy na pole trzecie, a z niego możemy wykonać następny ruch (poprzednio trzeci wykonany ruch był ostatni z możliwych do wykonania ruchów). Teraz pozostały już tylko dwa pola, których skoczek nie odwiedził. Ponieważ z pola oznaczonego trzynastką - numerem ostatnio wykonanego ruchu - nie jesteśmy w stanie wykonać następnego ruchu, musimy się zatem cofnąć do takiego położenia, z którego można wykonać ruch dotychczas niewykonywany. W ten sposób będziemy kontynuować próbę poszukiwania takiej drogi, która przeprowadzi skoczka przez wszystkie pola planszy.



Ryc. 6.12. Dzięki innej kolejności ruchów skoczek nie odwiedził tylko dwóch pól.

I tak dla szachownicy o wymiarach  $8 \times 8$  oraz przy początkowym położeniu skoczka w lewym górnym rogu szachownicy droga przejścia planszy wygląda następująco:

1	38	43	34	3	36	19	22
44	59	2	37	20	23	4	17
39	42	33	60	35	18	21	10
58	45	40	53	24	11	16	5
41	32	57	46	61	26	9	12
50	47	52	25	54	15	6	27
31	56	49	62	29	8	13	64
48	51	30	55	14	63	28	?

Liczbą 1 oznaczono początkowe ustawienie skoczka, a kolejną liczbą każdy następny ruch. W tym przypadku przejście planszy było możliwe.

Zajmijmy się teraz samym algorytmem od strony kodu. Ustalamy tablicę reprezentującą planszę:

```
int plansza [8] [8];
```

Następnie definiujemy funkcję określającą współrzędne pola planszy, na które przeniesie się skoczek po wykonaniu ruchu:

```
void przemieszczenie ()
```

```
nowe_wsp[0][0] = -2;  nowe_wsp[0][1] =  1
nowe_wsp[1][0] = -1;  nowe_wsp[1][1] =  2
nowe_wsp[2][0] =  1;  nowe_wsp[2][1] =  2
nowe_wsp[3][0] =  2;  nowe_wsp[3][1] =  1
nowe_wsp[4][0] =  2;  nowe_wsp[4][1] = -1
nowe_wsp[5][0] =  1;  nowe_wsp[5][1] = -2
nowe_wsp[6][0] = -1;  nowe_wsp[6][1] = -2
nowe_wsp[7][0] = -2;  nowe_wsp[7][1] = -1
```

Kodowanie metody  
przechodzenia  
planszy przez skoczka

Dla przykładu: jeśli skoczek znajdował się na polu [3][4], to po wykonaniu mchu zdefiniowanego w funkcji jako pierwszy przemieści się na pole [1][5]. Teraz już możemy napisać rekurencyjną funkcję przechodzenia planszy:

```
int Ruch(int i, int wsp1, int wsp2) // i - numer ruchu
                                         // wsp1, wsp2 - aktualne współrzędne skoczka
{
    int nr, wspanst1, wspanst2;
                                         // nr - kolejny ruch spośród ośmiu możliwych
                                         // wspanst1, wspanst2 - współrzędne następnego pola
    nr = 0;
    while (nr!=7)                      // dopóki są możliwe do wykonania ruchy
    {
        wspanst1 = wsp1+nowe_wsp[nr][0];
        wspanst2 = wsp2+nowe_wsp[nr][1];
        if (wspanst1>=0 && wspanst1<n && wspanst2>=0 && wspanst2<n)
                                         // jeśli ruch nie wyprowadzi skoczka poza planszę
            if (plansza[wspanst1][wspanst2]==0)
                {
                    // zaznaczenie ustawienia następnego ruchu
                    plansza[wspanst1][wspanst2] = i;
                    if (i<n*n)
                        {
                            // rekurencyjne wywołanie funkcji dla następnego ruchu
                            if (!Ruch(i+1,wspanst1,wspanst2))
                                // usunięcie zaznaczenia ruchu
                                plansza[wspanst1][wspanst2] = 0;
                            else
                                return 1;
                        }
                    else
                        return 1;
                }
        nr = nr+1;
    }
    return 0;
}
```

6.17

## 6. Rekurencja

### Opis problemu ośmiu hetmanów

Pełny kod programu znajdziesz na płycie CD (6.18).

Podobny w założeniach i sposobie realizacji jest problem ośmiu hetmanów. Zadanie polega na ustawieniu na kwadratowej planszy (o liczbie pól  $n^2$ ) 8 figur hetmanów tak, aby się wzajemnie nie szachowały. Figury te szachują się, gdy stoją:

- w tym samym wierszu,
- w tej samej kolumnie,
- na tej samej linii po skosie.

Postawimy więc hetmana na którymś z pól pierwszej linii planszy, następnie hetmana na drugiej linii planszy, tak aby nie był szachowany przez pierwszego hetmana, a następnie kolejnego hetmana na następnej linii, przy czym postępuwać tak będziemy dotąd, aż ustawiemy wszystkie figury lub dla kolejnego hetmana nie będzie już wolnego pola nieszachowanego przez poprzednio ustalone figury. Wówczas wycofamy się z ustawienia ostatniego hetmana (powrót algorytmu) i zastąpimy go innym. Jeśli żadne z ustawień nie będzie spełniało założeń zadania, zmienimy również ustawienie jeszcze wcześniejszego z ustawionych hetmanów. Jak widzisz, jest to również zadanie do realizacji metodą prób i błędów (z których wycofujemy się, realizując powrót algorytmu). Na klasycznej szachownicy żądana ustawienie ośmiu hetmanów jest możliwe, jednak na przykład na szachownicy cztero- lub dziewięciopolowej nie da się go wykonać. Spróbuj samodzielnie napisać program, który będzie ustalał ustawienia hetmanów; w razie trudności przeanalizuj kod programu, który umieściliśmy na płycie CD dołączonej do podręcznika (6.19).



### Pytania kontrolne

1. Opisz sposób rekurencyjnego definiowania problemów.
2. Jakie warunki muszą być spełnione, aby funkcja rekurencyjna była poprawna?
3. Na czym polegają metody „dziel i zwyciężaj”? Skąd wzięła się ich nazwa?
4. Jakiej metody użyjesz do obliczenia wyrazów ciągu Fibonacciego: rekurencyjnej czy iteracyjnej? Odpowiedź uzasadnij.
5. Porównaj metodę iteracyjną obliczania silni liczby naturalnej z metodą rekurencyjną.
6. Wyjaśnij rolę stosu w rekurencji.
7. Czym najczęściej grozi niepoprawny warunek zakończenia wywołań funkcji rekurencyjnych?

## Ćwiczenia

1. Napisz algorytm, który będzie obliczał wartość dowolnego wyrazu ciągu wskazanego wzorem rekurencyjnym:

$$\begin{cases} a_1 = 1 \\ a_2 = 2 \\ a_n = a_{n-1} + 2n + a_{n-2} \end{cases}$$

2. Napisz program, który rekurencyjnie znajduje miejsce zerowe wielomianu, jeśli dla podanych z zewnątrz argumentów wielomian przyjmuje wartości o przeciwnych znakach.
3. Napisz program, który rekurencyjnie szuka w ciągu maksymalnego i minimalnego elementu równocześnie.
4. Napisz rekurencyjną funkcję wypisującą na ekran zawartość tablicy.

5. Poniżej podana jest funkcja rekurencyjna:

```
int funkcja_rek(int i)
{
    if (i==1) return 2;
    else return funkcja_rek(i+1)+i;
}
```

Określ, jakie ograniczenia należy nałożyć na wartość argumentu  $i$ , aby funkcja była poprawna. Odpowiedź uzasadnij.

6. Poniżej zdefiniowana jest rekurencyjnie funkcja wyznaczająca wartość dowolnego, «-tego wyrazu ciągu:

```
int ciag (int i)
{
    if (i==1 || i==2) return 3;
    else return ciag(i-1)+3*ciag(i-2);
}
```

Na jej podstawie oblicz wartość czwartego wyrazu ciągu.

7. Napisz funkcję rekurencyjną obliczającą wartość ra-tego wyrazu ciągu, którego wyrazy są kwadratami kolejnych liczb naturalnych.

8. Napisz funkcję rekurencyjną, wyznaczającą największy wspólny dzielnik liczb całkowitych<sup>A</sup> i B.

Wskazówka: Wykorzystaj algorytm Euklidesa.

9. Napisz program, który dla zbioru n-elementowego podanego z zewnątrz wypisze wszystkie jego permutacje (permutacją zbioru nazywamy każdy ciąg utworzony ze wszystkich jego elementów).

Wskazówka: Dla zbioru  $Z = \{1, 2, 3, 4\}$  zbiór wszystkich permutacji został przedstawiony poniżej. Elementy zbioru umieszczone zostały w tablicy czteroelementowej.

Funkcji rekurencyjnej przekazujemy jako argumenty tablicę oraz indeks elementu: `void permutacje (int T[4], int index)`.

## 6. Rekurencja

W odwołaniu rekurencyjnym funkcja wywołuje się z indeksem o 1 większym.

**1234  
1243  
1324  
1342  
1432  
1423  
2134  
2143  
2314  
2341  
2431  
2413  
3214  
3241  
3124  
3142  
3412  
3421  
4231  
4213  
4321  
4312  
4132  
4123**

## 7. Struktury – typ definiowany przez użytkownika

W tym rozdziale pokażemy ci, jak grupować dane różnych typów, umieszczając je w pojedynczej zmiennej. Nauczysz się tworzyć nowe, zdefiniowane przez siebie typy złożone, zwane strukturami. Zastosujesz zdefiniowane przez siebie struktury w swoich programach. Utworzysz z nami małą, prostą bazę danych, w której wykorzystasz zdefiniowane przez siebie struktury.

### 7.1. Definicja struktury

Struktura (typ strukturalny) jest złożonym typem danych służącym do grupowania informacji opisujących jakiś obiekt. Dane te mogą być (i najczęściej są) różnych typów. Poszczególne dane zgrupowane w strukturze nazywamy polami lub składowymi struktury. Zaczniemy od przykładu: aby zapamiętać dane pewnej osoby, takie jak imię, nazwisko, wiek i plec (zatem dane różnego typu), umieszczaliśmy je w kilku zadeklarowanych zmiennych. Chcąc wyświetlić informacje o tej osobie, musielibyśmy wyświetlić wartości poszczególnych zmiennych. Należało więc zapamiętać, że wszystkie te zmienne dotyczą tej samej opisywanej wielkości. Można jednak zgrupować wszystkie informacje i umieścić w jednej zmiennej. Typ tej zmiennej to właśnie struktura. Aby korzystać z tego typu, musisz go najpierw zdefiniować. Potem możesz tworzyć zmienne zdefiniowanego przez siebie typu.

Definicja struktury ma postać

```
struct nazwa_typu      // nazwa typu podlega tym samym
                        // regułom co nazwy zmiennych
{
    typ_pola_1 nazwa_pola_1;
    typ_pola_2 nazwa_pola_2;
    .
    .
    .
    typ_pola_n nazwa_pola_n;
};                      // definicja typu musi kończyć się średnikiem
```

Definicja typu strukturalnego w kodzie programu

Aby pokazać, jak informacje o osobie umieścić w jednej zmiennej, możemy przykładowo zdefiniować strukturę o nazwie człowiek:

```
struct człowiek
{
    char imie[14];
    char nazwisko[20];
    int wiek;
    char piec[10];
};
```

## 7. Struktury - typ definiowany przez użytkownika

Mając tak zdefiniowany typ, możemy z niego korzystać, deklarując zmienne strukturalne. **Zmienne strukturalne są to zmienne, które są typu struktury przez nas zdefiniowanej.**

Dla typu **człowiek** mogą być to zmienne **mama, babcia, stryjek**.

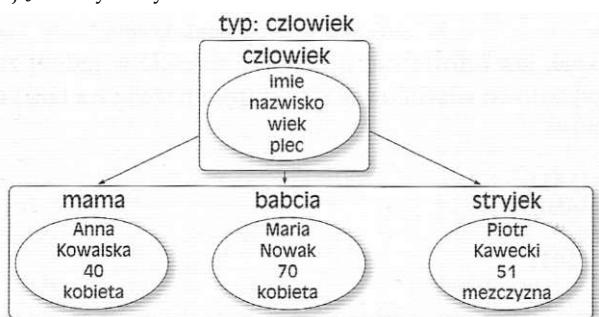
Ich deklaracja wygląda następująco:  
człowiek mama, babcia, stryjek;

Każdej zmiennej przypiszemy wartość imienia, nazwiska, wieku i pici przez odwołanie się do pól zdefiniowanej struktury. Odniesienie się do poszczególnych pól struktury realizujemy za pomocą operatora ozaczonego kropką, zwanego też operatorem wyłuskania.

```
mama.imie = "Anna";
mama.nazwisko = "Kowalska";
mama.wiek = 40;
mama.plec = "kobieta";
babcia.imie = "Maria";
babcia.nazwisko = "Nowak";
babcia.wiek = 70;
babcia.plec = "kobieta";
stryjek.imie = "Piotr";
stryjek.nazwisko = "Kawecki";
stryjek.wiek = 51;
stryjek.plec = "mezczyzna";
```

Każdej zmiennej przypisaliśmy wartości pól zdefiniowanej struktury, na przykład zmiennej mama nadaliśmy wartości: Anna Kowalska, 40, kobieta. Są to zatem zmienne różnych typów (wiek jest liczbą całkowitą, a imię, nazwisko i plec są ciągami znaków o różnych dozwolonych długościach). Zatem - w przeciwieństwie do tablicy, która będąc pojedynczą zmienną, przechowuje kilka, kilkanaście bądź więcej wartości tego samego typu - **pola struktury mogą być różnych typów**.

Na rycinie 7.1 przedstawiamy ilustrację zmiennych: mama, babcia, stryjek - są one zmiennymi typu człowiek. Zauważ, że samo zdefiniowanie typu człowiek nie opisuje jeszcze żadnej osoby: jest dopiero modelem, mówiącym o tym, jakie cechy przypiszemy konkretnej osobie. Dopiero zmienne strukturalne z nadanymi wartościami poszczególnych pól opisują rzeczywistych ludzi.



Ryc. 7.1. Graficzna ilustracja typu o nazwie człowiek oraz zmiennych tego typu: mama, babcia, stryjek

Pola zmiennej strukturalnej możemy oczywiście inicjować wartościami podanymi przez użytkownika, jak w przykładzie poniżej:

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

struct osoba
{
    char imie[20];
    char nazwisko[20];
    int wiek;
};

int main()
{
    osoba uczen;           // definicja zmiennej typu osoba
    cout << "podaj imie ucznia ";
    cin >> uczen.imie;
    cout << "podaj nazwisko ucznia ";
    cin >> uczen.nazwisko;
    cout << "podaj wiek ucznia ";
    cin >> uczen.wiek;
    cout << "Informacje o uczniu: \n";
    cout << uczen.imie << " " << uczen.nazwisko << " " << uczen.wiek << " lat";
    cin.ignore();
    getchar();
    return 0;
}
```

7.1

Na rycinie 7.2, obok użytego już przykładu uczeń, przedstawiliśmy jeszcze dwie inne przykładowe wielkości, które warto definiować za pomocą struktur:



Ryc.7.2. Struktury: uczeń, samochód i punkt wraz z przykładowymi polami

Jeśli chcemy zdefiniować strukturę, która opisuje cechy samochodów zgodnie z atrybutami zaznaczonymi na rysunku, to prawidłowa definicja wygląda następująco:

```
struct samochod
{
    char marka [30];           // jeśli uznamy, że jest to
                                // wystarczająca długość
    int pojemnosc_silnika;
    int rok_produkcji;
};                           // przypominamy o średniku!
```

Dwie przykładowe zmienne typu strukturalnego zadeklarujemy:  
`samochod moje_auto, auto_sasiada;`  
 a wartości nadamy im podobnie jak w poprzednim przykładzie.

## 7. Struktury - typ definiowany przez użytkownika

Definicja struktury punkt w kartezjańskim układzie współrzędnych miałyby postać:

```
struct punkt
{
    float wsp_x;
    float wsp_y;
};
```

Poniżej przedstawiamy fragment kodu programu, w którym jest wykorzystana zdefiniowana struktura:

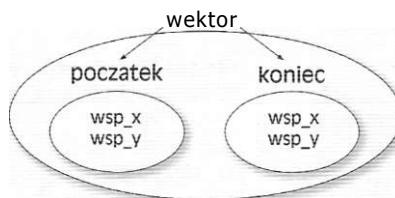
```
int main()
{
    punkt punkt1, punkt2; // deklaracja zmiennych typu punkt
    punkt1.wsp_x = 3;
    punkt1.wsp_y = 5;
    punkt2.wsp_x = 3;
    punkt2.wsp_y = 7.4;
    if (punkt1.wsp_x==punkt2.wsp_x)
        cout << "Punkty wyznaczają prostą równoległą do osi OY";
    getch();
    return 0;
}
```

Po zdefiniowaniu struktury punkt przejdźmy do kolejnej definicji matematycznej - tym razem dotyczącej wektorów. Przypomnijmy, że wektorem jest uporządkowana para dwóch punktów. Pierwszy z nich nazywamy początkiem wektora, drugi zaś jego końcem. Aby opisać wektory w kartezjańskim układzie współrzędnych, możemy zdefiniować strukturę do tego przeznaczoną:

```
struct wektor
{
    float poczatek_wsp_x; // odcięta początku wektora
    float poczatek_wsp_y; // rzędna początku wektora
    float koniec_wsp_x; // odcięta końca wektora
    float koniec_wsp_y; // rzędna końca wektora
};
```

Na podstawie definiowanej struktury punkt zdefiniujemy strukturę wektor (ryc. 7.3):

```
struct wektor
{
    punkt poczatek;
    punkt koniec;
};
```



Ryc. 7.3. Struktura wektor, której pola poczatek i koniec są również strukturami o polach wsp\_x i wsp\_y.

Teraz, w celu odwołania się do poszczególnych pól zmiennej wektor, zastosujemy dwukrotnie operator odniesienia do pola struktury. Dla przykładu: aby nadać wartości zmiennej moj\_wektor należącej do typu wektor, wykonamy kolejno instrukcje:

```
moj_wektor.początek.wsp_x = 4.5;
```

```
moj_wektor.początek.wsp_y = -6.1;
moj_wektor.koniec.wsp_x = 2.8;
moj_wektor.koniec.wsp_y = 4.9;
```

Pamiętaj, że definicja struktury punkt musi się znajdować przed definicją struktury wektor, gdyż druga z nich wykorzystuje pierwszą. Przeanalizuj krótki program, wykorzystujący obie struktury; jego zadaniem jest wyznaczenie współrzędnych środka wektora, którego krańce podajemy z zewnątrz:

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

struct punkt           // definicja struktury punkt musi poprzedzać definicję
{                      // struktury wektor
    float wsp_x;
    float wsp_y;
};                     // pamiętaj o średniku kończącym definicję struktury

struct wektor
{
    punkt poczatek;
    punkt koniec;
};                   // pamiętaj o średniku kończącym definicję struktury

int main()
{
    punkt p;           // deklaracja zmiennej strukturalnej typu punkt
    wektor w;           // deklaracja zmiennej strukturalnej typu wektor
    cout << "Podaj odcięta poczatku wektora: ";
    cin >> w.poczatek.wsp_x;
    cout << "Podaj rzedna poczatku wektora: ";
    cin >> w.poczatek.wsp_y;
    cout << "Podaj odcięta konca wektora: ";
    cin >> w.koniec.wsp_x;
    cout << "Podaj rzedna konca wektora: ";
    cin >> w.koniec.wsp_y;
    p.wsp_x = 0.5*(w.poczatek.wsp_x+w.koniec.wsp_x);
    p.wsp_y = 0.5*(w.poczatek.wsp_y+w.koniec.wsp_y);
    cout << "Oto wspolrzedne srodka wektora: " << p.wsp_x << ", " << p.wsp_y;
    cin.ignore();
    getchar();
    return 0;
}
```

7.2

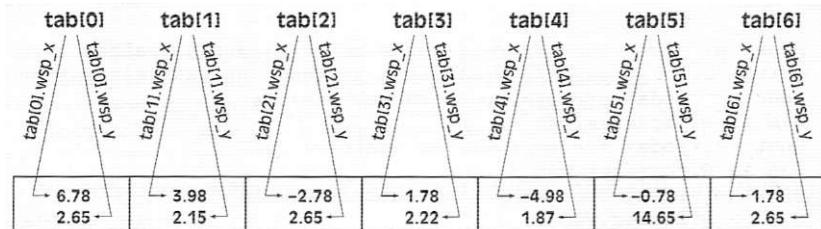
Już w tym krótkim programie można zauważyc, że dużo łatwiej jest odnosić się do pojedynczego wektora za pomocą jednej zmiennej, niż przeznaczać na jego opis aż cztery zmienne (dwie współrzędne początku i dwie współrzędne końca).

## 7.2. Tablice o elementach typu strukturalnego

Wykorzystanie struktur do definiowania pojedynczych zmiennych nie jest jedynym ich zastosowaniem. Zwykle grupujemy większą liczbę danych zdefiniowanych za pomocą tej samej struktury, przechowując je w tablicach. Do przechowywania siedmiu punktów bez użycia struktur można przygotować dwie tablice: jedną dla wartości odciętych, drugą dla wartości rzędnych. Natomiast typ strukturalny traktuje wszystkie poła jako jedną spójną całość, a zatem można utworzyć tablicę, w której zgrupujemy zmienne typu wcześniej zdefiniowanej struktury. Jeśli więc zdefiniowaliśmy już strukturę, na przykład `punkt`, to możemy zdefiniować tablicę punktów:

```
punkt tab[7];
i odpowiednio wypełnić ją wartościami:
for (int i=0; i<7; i++)
{
    cout << "podaj odcieta punktu " << i+1 << "-ego ";
    cin >> tab[i].wsp_x;
    cout << "podaj rzedna punktu " << i+1 << "-ego ";
    cin >> tab[i].wsp_y;
}
```

Zauważ, że najpierw odwołujemy się do elementu tablicy; element ten jest zmienną strukturalną typu `punkt`. Następnie odwołujemy się do pola struktury. Tablicę taką przedstawia rycina 7.4.



Ryc. 7A. Ilustracja odwołania do pól struktur będących elementami tablicy

Jeśli chcemy odnieść się do elementu tablicy o indeksie `i`, piszemy `tab[i]`. Ponieważ element ten jest typu strukturalnego, zatem aby odnieść się do pola elementu, stawiamy kropkę, a dopiero po niej nazwę pola, do którego się odwołujemy. Gdyby pole to również było typu strukturalnego, postawilibyśmy kolejną kropkę, a po niej nazwę pola zagnieżdzonego. Przykładowo, jeśli zadeklarowaliśmy tablicę wektor `vec[20];`, to odciętej pierwszego z wektorów nadalibyśmy wartość w sposób następujący:  
`vec[0].poczatek.wsp_x = 9.6;`

Za wypełnienie całej tablicy wektorów odpowiadały fragment kodu:

```
for (int i=0; i<20; i++)
{
    cout << "podaj odcieta poczatku " << i+1 << "-go wektora ";
    cin >> tab[i].poczatek.wsp_x;
    cout << "podaj rzedna poczatku " << i+1 << "-go wektora ";
    cin >> tab[i].poczatek.wsp_y;
    cout << "podaj odcieta konca " << i+1 << "-go wektora ";
    cin >> tab[i].koniec.wsp_x;
    cout << "podaj rzedna konca " << i+1 << "-go wektora ";
    cin >> tab[i].koniec.wsp_y;
}
```

W programach, w których ma sens grupowanie informacji różnych typów w jednej złożonej zmiennej, warto stosować struktury.

Na koniec przeanalizuj prosty program:

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

struct uczen           // definicja struktury
{
    char imie[20];
    char nazwisko[20];
    long numer_akt;
};                      // pamiętaj o średniku!
int main()
{
    uczen grupa[10];
    cout << "Wypelniamy tablice z danymi uczniów z naszej grupy: ";
    for (int i=0; i<10; i++)
    {
        cout << "\nPodaj imię ucznia: ";
        cin >> grupa[i].imie;
        cout << "Podaj nazwisko ucznia: ";
        cin >> grupa[i].nazwisko;
        cout << "Podaj numer akt ucznia: ";
        cin >> grupa[i].numer_akt;
    }
    cin.ignore();
    getchar();
    return 0;
}
```

7.3

Kod tego programu można jeszcze uzupełnić o fragment, który odpowiadały za wypisanie wprowadzonych do tablicy danych dotyczących uczniów, jest to jednak na tyle proste, że element ten pominęliśmy.

W wypadku tablic, których elementy są typu prostego, obsługę wy pełniania tablic, wypisywania ich, a także dokonywania wszelkich operacji na ich elementach powinniśmy umieścić w uprzednio zdefiniowanych funkcjach. Zagadnieniu temu jest poświęcony następny podrozdział.

## 7. Struktury - typ definiowany przez użytkownika

### Baza danych z wykorzystaniem tablicy struktur

Znając już pojęcie struktury oraz podstawowe czynności obsługi zmiennych typu strukturalnego, możemy przystąpić do budowania małej bazy danych. Bazę danych zapiszemy jako tablicę, musimy zatem założyć na wstępnie, ile maksymalnie elementów będzie zawierała.

Przyjmijmy więc, że tworzymy listę uczniów, którzy chcą uczestniczyć w wyjeździe na narty do Zwardonia. Zakładamy przy tym, że nie będzie ich więcej niż dwudziestu, gdyż tyle jest miejsc w autokarze. Zaczniemy od zdefiniowania struktury uczestnika wycieczki. Struktura uczestnika będzie się różnić od poprzednio zdefiniowanej struktury ucznia, ponieważ oprócz imienia i nazwiska chcemy zapisać datę urodzenia ucznia jako dokładniejszą informację aniżeli wiek. Zamiat uzyć w strukturze ucznia trzech pól związanych z rokiem, miesiącem i dniem urodzenia, wcześniej zdefiniujemy strukturę definiującą datę.

```
struct data
{
    int rok;
    int miesiąc;
    int dzień;
};
```

Teraz możemy strukturę `data` wykorzystać przy definiowaniu kolejnej struktury:

```
struct uczestnik
{
    char imie[20];
    char nazwisko[20];
    data data_urodzenia; // to pole jest typu struktury zdefiniowanej
                         // wcześniej
```

### Sposób działania tworzonej bazy

Najprostszy program obsługujący naszą bazę realizuje następujące funkcje:

- tworzy bazę, dopisując kolejno uczestników wycieczki;
- wyświetla listę wszystkich uczestników.

Jeśli wśród osób, które wpisały się na listę, jest opiekun wycieczki będący najstarszym jej uczestnikiem, to zastosujemy również funkcję, która wyświetla dane opiekuna wycieczki. Definicje funkcji użytych w programie umieścimy bezpośrednio po definicji struktur.

Zacznijmy od zdefiniowania funkcji, której zadaniem jest wypełnienie przygotowanej tablicy uczestników wycieczki. W wyniku jej działania otrzymujemy informacje o liczbie osób wpisanych do bazy. Funkcja ta będzie nam potrzebna choćby przy wypisywaniu uczestników: wypisana zostanie tylko zapełniona część tablicy, nie zaś cała tablica.

```

int wypelnij(uczestnik osoby[20])      // argumentem funkcji
                                         // jest tablica zmiennych
                                         // strukturalnych
{
    int t = 0;
    char odpowiedz = 'T';
    do
    {
        cout << "Podaj imie uczestnika ";
        cin >> osoby[t].imie;
        cout << "Podaj nazwisko uczestnika ";
        cin >> osoby[t].nazwisko;
        cout << "Podaj rok urodzenia uczestnika ";
        cin >> osoby[t].data_urodzenia.rok;
        cout << "Podaj miesiac urodzenia uczestnika ";
        cin >> osoby[t].data_urodzenia.miesiac;
        cout << "Podaj dzien urodzenia uczestnika ";
        cin >> osoby[t].data_urodzenia.dzien;
        cout << "Czy ktos jeszcze chce sie zapisac? T/N ";
        cin >> odpowiedz;
        t++;
    }
    while (t<20 && odpowiedz!='N' && odpowiedz!='n');
    return t;                                // funkcja przyjmuje wartość równą
                                                // liczbie zajętych komórek tablicy
}

```

7.4

Warto zauważyć, że argumentem funkcji jest tablica zmiennych typu `uczestnik`. Skoro tablica może być argumentem funkcji, to nic nie stoi na przeszkodzie, aby była to tablica typu przez nas zdefiniowanego.

Skonstruujemy teraz funkcję `wypisz`, której zadaniem jest wypisanie kolejno wszystkich uczestników. Jest to funkcja dwuargumentowa - pierwszym argumentem jest tablica, do której wprowadziliśmy kolejno dane uczestników (oczywiście jest to tablica przechowująca elementy typu `uczestnik`). Drugi argument informuje o liczbie wpisanych osób, dlatego nie zostanie wypisana cała tablica, lecz jej część zapełniona danymi uczestników.

```

void wypisz(uczestnik t[20], int liczba)
{
    for (int i=0; i<liczba; i++)
    {
        cout << t[i].imie << " " << t[i].nazwisko;
        cout << " " << t[i].data_urodzenia.rok;
        cout << " " << t[i].data_urodzenia.miesiac;
        cout << " " << t[i].data_urodzenia.dzien << endl;
    }
}

```

7.5

## 7. Struktury - typ definiowany przez użytkownika

Kolejna funkcja pobiera również dwa argumenty - tablicę uczestników wraz z informacją o liczbie wpisanych do niej osób. W wyniku jej działania otrzymujemy dane o najstarszej wpisanej do bazy osobie; według naszych założeń będzie to opiekun wycieczki. Zauważ, że wynik funkcji jest strukturą; otrzymujemy więc komplet informacji o opiekunie: jego imię, nazwisko, datę urodzenia. Nie oznacza to, że funkcja przyjmuje kilka wartości - wciąż jest to jedna i tylko jedna wartość, ale składająca się z kilku pól. Gdybyśmy informacji o uczestnikach nie grupowali w zmiennych typu strukturalnego, nie byłoby możliwe uzyskanie jako wartości funkcji równocześnie kilku informacji o opiekunie. Oto treść tej funkcji:

```
uczestnik znajdz_opiekuna(uczestnik t[20], int liczba)
{
    uczestnik opiekun;
    opiekun = t[0];
    for (int i=1; i<liczba; i++)
        if (opiekun.data_urodzenia.rok > t[i].data_urodzenia.rok)
            opiekun = t[i];
    return opiekun;
}
```

7.6

Jest to poszukiwanie elementu minimalnego w tablicy, tyle tylko, że szukamy tego elementu w konkretnym polu struktury, jakim jest zagnieżdżone pole o nazwie rok. Aby zbytnio nie komplikować kodu programu, przyjęliśmy, że opiekun jest zdecydowanie starszy od uczniów, dlatego porównujemy wyłącznie wartość roku urodzenia, a nie miesiąca czy dnia urodzenia. Na koniec potrzebujemy jeszcze funkcji, która pobiera jako argument zmienną typu strukturalnego, a jej zadaniem jest wypisanie wszystkich pól zmiennej:

```
void wypisz(uczestnik ktos)
{
    cout << ktos.imie << " " << ktos.nazwisko;
    cout << " " << ktos.data_urodzenia.rok;
    cout << " " << ktos.data_urodzenia.miesiac;
    cout << " " << ktos.data_urodzenia.dzien << endl;
}
```

7.7

Teraz już możemy napisać funkcję główną, która wykorzysta zarówno zdefiniowane struktury, jak i funkcje je przetwarzające. Funkcja wypełnij wpisującą uczestników wycieczki do bazy danych wykorzystuje pętle do ... while, przy czym warunkiem zakończenia pętli jest negatywna odpowiedź na pytanie, czy ktoś jeszcze chce się dopisać do bazy. Uczestników wpisywać będziemy do tablicy, którą nazwiemy osoby. Na stronie obok prezentujemy funkcję main wykorzystującą wszystkie wcześniej przedstawione funkcje obsługujące bazę. Całość programu znajdziesz na płycie CD dołączonej do podręcznika (7.8)

```

int main()
{
    uczestnik osoby[20];
    int t = wypelnij(osoby);           // wykona się funkcja wypelnij, a jej wartość
    cout << endl << endl;             // zostanie przypisana zmiennej t
    wypisz(osoby,t);
    cout << "*****Opiekun wycieczki:*****" << endl;
    wypisz(znajdz_opiekuna(osoby,t));
    cin.ignore();
    getchar();
    return 0;
}

```

7.9

W ten sposób powstała bardzo prosta w zamyśle i nietrudna do utworzenia baza. Dzięki użyciu struktur kod programu jest czytelny i przejrzysty. Teraz możesz się pobawić w rozbudowę tej bazy, wzbogacając ją o nowe funkcje. Warto dodać na przykład funkcję usuwającą elementy bazy, bowiem osoba, która wpisała się na listę wyjazdową może się rozmyślić i zrezygnować z wyjazdu. Wówczas element zbędny w tablicy należy zapełnić następnym wypełnionym, a następny kolejnym, tak aby zlikwidować powstałą lukę w tablicy. Również samo wpisywanie uczestnika wycieczki można umieścić w funkcji i korzystać z niej w funkcji głównej main. Inwencję pozostawiamy tobie, pamiętaj jednak, aby tworzyć funkcje, które operują na strukturach. Przypomnijmy, że:

- argumentami funkcji mogą być zarówno struktury, jak i tablice struktur;
- funkcja może przyjąć wartość, która jest typu strukturalnego.

### Pytania kontrolne



1. Scharakteryzuj złożony typ danych, jakim jest struktura.
2. Jakie typy mogą mieć pola w strukturze?
3. W jaki sposób w definiowanej strukturze należy umieścić pole, które jest inną strukturą?
4. W którym miejscu programu definiuje się struktury?
5. Na jakie sposoby można przekazać zmienną strukturalną do funkcji?
6. Jak można się odwołać do pól struktur umieszczonych w tablicy?
7. Jaka jest podstawowa różnica pomiędzy strukturą a tablicą?
8. Co to jest pole struktury?
9. W jaki sposób odwołujemy się do pola struktury?

## Ćwiczenia

1. Znajdź i popraw błędy w poniższym fragmencie kodu programu:

```
struct kwiat
{
    int ilosc_platkow;
    int osiagana_wysokosc;
}
int main()
{
    kwiat lilia;
    ilosc_platkow.lilia = 4;
    osiagana_wysokosc.lilia = 40;
    return 0;
}
```
2. Na podstawie poniższej funkcji zapisz najprostszą definicję struktury w niej wykorzystanej:

```
void wyswietl_wartosci(osoba pracownik)
{
    cout << "Staz pracy pana " << pracownik.nazwisko;
    cout << "," << pracownik.staz_pracy;
    cout << endl << "Zarabia on: " << pracownik.zarobek;
}
```
3. Zdefiniuj strukturę opisującą film na płycie DVD. Film powinien być opisany przez tytuł, nazwisko reżysera, rok produkcji i czas trwania.
4. Zdefiniuj typ strukturalny opisujący kartę do gry oraz funkcję, która porównuje, czy dwie karty są tego samego koloru.

## 8. Typ wskaźnikowy – zastosowania

W tym rozdziale omówimy podstawy pracy ze wskaźnikami. Nauczysz się deklarować wskaźniki, nadawać im wartości i korzystać z nich przy odwoływaniu się do wskazanych obszarów pamięci. Wyjaśnimy też, dla czego funkcja, która pobiera jako argument tablicę, działa bezpośrednio na niej, a nie na utworzonej kopii tablicy.

### 8.1. Deklaracja zmiennej wskaźnikowej i podstawowe operacje na wskaźnikach

Każda zmienna przechowywana w pamięci komputera ma przydzielony obszar, z którym jest związany jej adres. Aby odwołać się do tej zmiennej, wystarczy ten adres znać. Do tej pory zmienne, które definiowaliśmy, przechowywały różne wartości, zależnie od typu zmiennej. Teraz zajmujemy się zmienną, której wartością jest adres.

#### Definicja

**Wskaźnikami** nazywamy zmienne, których wartością jest adres pewnego obszaru pamięci, a zadaniem – wskazywanie na inne zmienne.

Deklaracja zmiennej wskaźnikowej wygląda następująco: typ `*nazwa`; gdzie typ oznacza typ wbudowany lub zdefiniowany przez nas (może to być np. struktura, którą zdefiniowaliśmy). Gwiazdka przed nazwą zmiennej informuje o tym, że jest ona wskaźnikiem. W ten sposób zadeklarowaliśmy wskaźnik o nazwie `nazwa`. Wskaźnik będzie mógł wskazywać na zmienne typu, dla którego został zdefiniowany.

Deklaracja wskaźnika

Rozważania na temat wskaźników zaczniemy od prostego programu, który pozwoli ci zrozumieć sposób działania wskaźnika:

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

int main()
{
    int i = 13;
    int *wsk;           // deklaracja wskaźnika (zauważ gwiazdkę przed nazwą)
    wsk = &i;            // od tej chwili wskaźnik będzie wskazywał na adres,
                        // pod którym znajduje się zmieniona i
    cout << "Wskaźnik typu integer wskazuje na 'i', której wartość jest: " << *wsk << endl;
```

## 8. Typ wskaźnikowy - zastosowania

```
cout << "Wartosc wskaznika: " << wsk;
getchar();
return 0;
}
```

8.1

Skierowanie  
wskaźnika na  
adres zmiennej

W naszym krótkim programie zdefiniowaliśmy zmienną *i*, która jest typu całkowitego, i nadaliśmy jej wartość początkową 13. W drugiej linijce zadeklarowaliśmy wskaźnik o nazwie *wsk*. Dopóki nie zwiążemy go z adresem jakiejś zmiennej, pozostaje bezużyteczny, wskazując przypadkowe miejsce w pamięci. Zanim użyjemy wskaźnika po raz pierwszy, należy go powiązać z adresem jakiejś zmiennej, inaczej moglibyśmy w sposób niezamierzony modyfikować wartości znajdujące się pod adresem, który aktualnie jest wartością wskaźnika.

W kolejnej linii kodu pojawiła się instrukcja:

```
wsk = &i;
```

Znak **&** występujący przed nazwą zmiennej *i* oznacza adres obszaru pamięci, w której ta zmienna jest przechowywana.

Od tego momentu wskaźnik *wsk* wskazuje zmienną *i*, mając wartość adresu, pod którym jest przechowywana zmienna *i*. W kolejnej linijce:

```
cout << "Wskaźnik typu integer wskazuje na 'i', której wartoscia  
jest: " << *wsk << endl;
```

zostaje odczytana wartość zmiennej, którą wskaźnik *wsk* wskazuje.

Z kolei w następnej linii:

```
cout << "Wartość wskaźnika: " << wsk;
```

odwołujemy się do wartości wskaźnika, czyli wyprowadzamy na ekran monitora adres pierwszej z komórek, w których jest przechowywana zmienna *i* (pierwszej z komórek, bo oczywiście zmienna w zależności od typu zajmować może więcej niż jedną komórkę).

### Zapamiętaj

**\*wsk** oznacza wartość, która znajduje się pod adresem będącym wartością wskaźnika *wsk*.

**wsk** oznacza adres, czyli wartość wskaźnika o nazwie *wsk*.

**&a** oznacza adres komórki, w której znajduje się zmienna *a*.

A zatem jeśli zadeklarujemy wskaźnik *wsk*:

```
int *wsk;
```

a następnie wykonamy przypisanie:

```
wsk = &a;
```

to od tego momentu wartość wskaźnika jest adresem zmiennej *a*, która jest typu **integer**.

**Wskaźnik służący do wskazywania zmiennych jednego typu nie może służyć do wskazywania zmiennych Innych typów.**

Popatrzmy, jak wygląda przykładowy ekran po wykonaniu programu:

```
Wskaźnik typu integer wskazuje na 'i', której wartość jest: 13
Wartość wskaznika: 0x241ff5c
```

Jak widzisz, wartość wskaźnika to właśnie wartość adresu. Adresy są zapisane w komputerze w postaci szesnastkowej, stąd obok cyfr w wartości wskaźnika pojawiły się również litery.

Jeśli początkowo wartością wskaźnika jest adres konkretnej zmiennej, to nie oznacza wcale, że wskaźnik ten musi być cały czas do niej przypisany. W każdej chwili działania programu możemy zmienić wartość wskaźnika i przypisać mu adres innej zmiennej, byle typu zgodnego ze zdefiniowanym<sup>1</sup> wskaźnikiem. Oto krótki program demonstrujący zmianę wartości wskaźnika:

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

int main()
{
    int i = 13, j = 16;
    int *wsk;
    wsk = &i;           // wskaźnik wskazuje na zmienną i
    cout << "Wskaźnik 'wsk' wskazuje na 'i', która ma wartość: " << *wsk << endl;
    cout << "Wartość wskaznika: " << wsk << endl;
    cout << "Za chwilę przekierujemy wskaźnik na adres zmiennej 'j'" << endl;
    wsk = &j;           // wskaźnik wskazuje na zmienną j
    cout << "Teraz wskaźnik wskazuje na 'j', która ma wartość: " << *wsk << endl;
    cout << "Wartość wskaznika: " << wsk;
    getchar();
    return 0;
}
```

8.2

Po wykonaniu programu na ekranie możemy zobaczyć:

```
Wskaźnik 'wsk' wskazuje na 'i', która ma wartość: 13
Wartość wskaznika: 0x241ff5c
Za chwilę przekierujemy wskaźnik na adres zmiennej 'j'
Teraz wskaźnik wskazuje na 'j', która ma wartość: 16
Wartość wskaznika: 0x241ff58
```

Zwróć uwagę na wartości adresów komórek, w których są przechowywane zmienne *i, j*. Adresy te różnią się o 4 bajty, czyli dokładnie tyle, ile zajmuje zmienna typu **integer**. Wynika stąd, że zmiennym zostały przyporządkowane komórki sąsiadujące ze sobą w pamięci, choć oczywiście tak być nie musiało.

Zadaniem wskaźnika jest nie tylko wskazywanie zmiennych i odczytywanie ich wartości. Za pomocą wskaźników otrzymujemy również pełny dostęp do wskazywanych zmiennych. Na przykład instrukcję:

```
*wsk = 19;
```

odczytamy jako: w miejsce, na które wskazuje *wsk*, wpisz 19. Jeśli więc wskaźnik wskazywałby na zmienną *i*, to jej wartość zostanie zmieniona

## 8. Typ wskaźnikowy - zastosowania

na 19, jeśli zaś wsk wskazywałby na zmienną, to w ten sposób zmieniona zostałaby wartość zmiennej j.

Przeanalizujmy kolejny program:

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

int main()
{
    int i = 13, j = 20;
    int *wsk1, *wsk2;
    int a = 0;
    wsk1 = &i;           // wskaźnik wsk1 wskazuje na zmienną i
    wsk2 = &j;           // wskaźnik wsk2 wskazuje na zmienną j
    cout << "Zmienna 'a' ma wartosc: " << a << endl;
    a = *wsk1 + *wsk2;      // modyfikacja wartości zmiennej a (1)
    cout << "Teraz 'a' ma wartosc: " << a << endl;
    cout << "Zmienna 'i' ma wartosc: " << i << endl;
    *wsk1 = *wsk1+3;      // modyfikacja wartości zmiennej i (2)
    cout << "Teraz zmienna 'i' ma wartosc: " << i << endl;
    cout << "Zmienna 'j' ma wartosc: " << j << endl;
    *wsk2 = *wsk1;        // modyfikacja wartości zmiennej j (3)
    cout << "Teraz zmienna 'j' ma wartosc: " << j << endl;
    getchar();
    return 0;
}
```

8.3

W programie zostały zadeklarowane dwa wskaźniki, jeden z nich wskazuje na zmienną *i* o wartości 13, drugi na zmienną *j*, której wartość wynosi 20. Zmienna *a* początkowo ma wartość 0. Modyfikujemy ją z wykorzystaniem wskaźników. Linię, w której się to odbywa, opatrzoną komentarzem (1), odczytamy: „niech od teraz zmienna *a* ma wartość będącą sumą wartości zmiennych, na które wskazują wskaźniki wsk1 i wsk2” (czyli od teraz będzie miała wartość sumy wartości zmiennych *i, j*). A zatem możemy wykonywać operacje arytmetyczne na zmiennych za pomocą wskaźników z nimi związanych. Kolejnych modyfikacji dokonaliśmy na wartościach zmiennych *i, j* w linii (2).

Instrukcja *\*wsk1 = \*wsk1+3;* jest równoważna instrukcji *i = i+3*, ponieważ wskaźnik wsk wskazuje na zmienną *i*. W linii (3) wykonana została instrukcja:

```
*wsk2 = *wsk1;
```

Skoro wskaźnik wsk2 wskazuje obecnie na zmienną *a* wsk1 na zmienną *i*, to powyższą instrukcję odczytamy: „zmiennej *j* przypisz wartość zmiennej *i*”. Uzyskamy ten sam efekt jak przy zastosowaniu instrukcji: *j = i;*

## 8.1. Deklaracja zmiennej wskaźnikowej i podstawowe operacje na wskaźnikach

Stąd spodziewany efekt działania powyższego programu:

```
Zmienna 'a' ma wartość: 0
Teraz 'a' ma wartość: 33
Zmienna 'i' ma wartość: 13
Teraz zmienna 'i' ma wartość: 16
Zmienna 'j' ma wartość: 20
Teraz zmienna 'j' ma wartość: 16
```

A co się stanie, jeśli zmienimy wartość wskaźnika? Przeanalizujmy poniższy program:

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

int main()
{
    int i = 5, j = 9;
    int *wsk1, *wsk2;
    wsk1 = &i;           // (1)
    wsk2 = &j;           // (2)
    cout << "wskaźnik wsk1 wskazuje na zmienną o wartości: " << *wsk1 << endl;
    cout << "wskaźnik wsk2 wskazuje na zmienną o wartości: " << *wsk2 << endl << endl;
    wsk1 = wsk2;         // zmiana wartości wskaźnika (3)
    cout << "wskaźnik wsk1 wskazuje na zmienną o wartości: " << *wsk1 << endl;
    cout << "wskaźnik wsk2 wskazuje na zmienną o wartości: " << *wsk2 << endl;
    getchar();
    return 0;
}
```

8.4

Zdefiniowaliśmy dwa wskaźniki wskazujące na zmienne typu całkowitego. Po wykonaniu instrukcji przypisania wartości wskaźników w linii (1) i (2) pierwszy z nich wskazuje na zmienną *i* (teraz ma ona wartość 5), drugi na zmienną *j*, która ma wartość 9. W linii (3) wykonana została instrukcja: *wsk1 = wsk2*; która oznacza: „wskaźnikowi *wsk1* przypisz wartość wskaźnika *wsk2*”. Skoro teraz *wsk1* ma tę samą wartość, co *wsk2*, wskazuje więc to samo, co wskaźnik *wsk2*, czyli zmienną *j*:

```
wskaźnik wsk1 wskazuje na zmienną o wartości: 5
wskaźnik wsk2 wskazuje na zmienną o wartości: 9

wskaźnik wsk1 wskazuje na zmienną o wartości: 9
wskaźnik wsk2 wskazuje na zmienną o wartości: 9
```

Widzisz więc, że możemy modyfikować nie tylko wartości zmiennych, na które wskaźniki wskazują, ale i wartości samych wskaźników, jak zrobiliśmy to w powyższym przykładzie. Ważnymi i przydatnymi operacjami wykonywanymi na wskaźnikach są inkrementacja i dekrementacja. Začnijmy od przykładu. Jeśli zdefiniujemy wskaźnik:

```
int *wsk;
```

a następnie przypiszemy mu adres zmiennej całkowitej *k* za pomocą instrukcji:

```
wsk = &k;
```

to instrukcja *wsk++*; zwiększy wartość wskaźnika *wsk* o 4, gdyż tyle bajtów zajmuje obiekt, dla którego wskaźnik jest zdefiniowany.

## 8. Typ wskaźnikowy - zastosowania

Gdyby wskaźnik miał wskazywać na zmienne typu `double`, byłby zadeklarowany:

```
double *wsk;
```

Wówczas ta sama instrukcja `wsk++`; zwiększyłaby jego wartość o 8, ponieważ tyle bajtów zajmuje zmienna typu `double`.

### Zapamiętaj

Instrukcja `*wsk++`; modyfikuje wartość zmiennej, na którą wskazuje wskaźnik.

Instrukcja `wsk++`; modyfikuje wartość wskaźnika, czyli adresu, zwiększając tę wartość o tyle bajtów, ile zajmuje zmienna, na którą wskaźnik może wskazywać.

## 8.2. Przekazywanie argumentów funkcji przez wskaźnik

Pamiętasz, że zmienne przekazujemy do funkcji przez wartość lub przez referencję. W pierwszym wypadku funkcja działa na zmiennych lokalnych, których wartości są kopiami argumentów funkcji, w drugim zaś na oryginalnych zmiennych, czyli zmiany zostają dokonane na komórkach w pamięci, które zmienne zajmują. Dlatego zmiany dokonane przez funkcję na zmiennych są zachowane po wyjściu z funkcji tylko w tym drugim wypadku. Teraz pokażemy, jak przekazać funkcji zmienną za pomocą wskaźników.

Zacznijmy od przykładu wraz z pokazaniem efektu jego działania:

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

void zmien(int *wsk)           // definicja funkcji
{
    *wsk = *wsk+10;
}

int main()
{
    int a = 3;
    int *wsk;
    wsk = &a;
    cout << "Zmienna 'a' ma wartosc: " << a << endl;
    zmien(wsk);                  // wywołanie funkcji
    cout << "Teraz zmienna 'a' ma wartosc: " << a;
    getchar();
    return 0;
}
```

8.5

```
Zmienna 'a' ma wartosc: 3
Teraz zmienna 'a' ma wartosc: 13
```

Przyjrzyjmy się definicji funkcji: skoro funkcja pobiera jako argument wskaźnik na zmienną, to faktycznie pobiera adres zmiennej. A jeśli pobrany został adres zmiennej, funkcja działa bezpośrednio na zmiennej, a nie na jej kopii. Nasza zdefiniowana funkcja działa więc tak samo jak funkcja w postaci:

```
void zmien(int &k) // definicja funkcji
{
    k = k+10;
}
```

Zalecamy jednak przekazywanie funkcji argumentów będących wskaźnikami, ponieważ mamy wtedy pewność, że taka funkcja będzie działała na oryginalnych zmiennych. (Pamiętaj o naszym ostrzeżeniu przed definiowaniem funkcji, którym argumenty przekazuje się przez referencję, gdyż funkcje te przy wywołaniu nie różnią się zapisem od funkcji, którym argumenty przekazuje się przez wartość).

### 8.3. Zastosowanie wskaźników w tablicach

Przejdzmy teraz do wykorzystania wskaźników przy obsłudze tablic. Zaczniemy od programu, dzięki któremu zobaczymy, jak ułożone są w pamięci kolejne elementy tablicy:

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

int main()
{
    int tab[5];
    cout << "Kolejne elementy tablicy znajdują się w komórkach: " << endl;
    for (int i=0; i<5; i++)
        cout << &tab[i] << " ";
    getchar();
    return 0;
}
```

8.6

Zauważ, że w pętli nie wyświetlamy wartości elementów, lecz adresy komórek, w których te wartości będą przechowywane. Przykładowy ekran po wykonaniu programu będzie wyglądał następująco:

```
Kolejne elementy tablicy znajdują się w komórkach:
8x8812ff7c 8x8812ff88 8x8812ff84 Bx8812ff88
```

Widać stąd, że jeśli znamy adres pierwszego elementu tablicy i wiemy, ile bajtów zajmuje zmienna tego samego typu, co elementy tablicy, to łatwo obliczymy adres komórki, pod którym znajdziemy zapisaną wartość

## 8. Typ wskaźnikowy - zastosowania

dowolnego n-tego elementu tablicy. Jeśli więc ustawimy wskaźnik na pierwszy element tablicy, to inkrementując go, będziemy się przesuwać do kolejnych jej elementów. Aby znaleźć adres pierwszego elementu tablicy, wystarczy odwołać się do nazwy tablicy.

### Zapamiętaj

W języku C++ nazwa tablicy jest jednocześnie wskaźnikiem do jej pierwszego elementu.

Oznacza to, że `tab` jest tym samym, co `&tab[0]`.

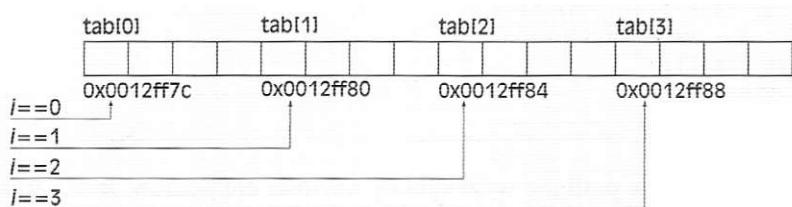
Wynika stąd, że instrukcje `cout << tab;` i `cout << &tab[0];` są równoważne. Dlatego gdy przekazywaliśmy wywoływanej funkcji nazwę tablicy, to przekazywaliśmy wskaźnik do jej pierwszego elementu, czyli adres w pamięci, pod którym znajdowała się tablica. Funkcja działała więc na oryginalnej tablicy, a nie na jej kopii.

Najpierw pokażemy ci, jak jest obsługiwana tablica w tradycyjny sposób. Założmy, że wypełniamy tablicę czteroelementową w sposób iteracyjny, odnosząc się do poszczególnych jej elementów za pomocą indeksów. To zadanie realizuje instrukcja iteracyjna, którą już doskonale znasz:

```
for (int i=0; i<4; i++) tab[i] = rand()%11;
```

Zastanówmy się jednak, jakie jest jej działanie, jeśli chodzi o sposób realizacji zadania pod względem dokonywanych operacji.

Zmiennej `i` zostaje przypisana wartość 0 i wyszukiwany zostaje element `tab[0]`. Po znalezieniu komórki w pamięci zajętej przez ten element zostaje mu przypisana wartość. Aby przypisać wartość dowolnemu `i`-temu elementowi tablicy, każdorazowo najpierw zostaje znaleziony początek tablicy, a dopiero potem wyliczona wartość adresu obszaru pamięci, w którym znajduje się `i`-ty element (ryc. 8.1). Losowo wygenerowana liczba zostaje wpisana w ten obszar pamięci.



Ryc. 8.1. Fragment obszaru pamięci z tablicą `tab`, w której każdy element zajmuje 4 komórki pamięci, wraz z ilustracją odwoływania się do komórek tablicy przez indeks `i`.

Analogicznie wygląda odczytywanie i modyfikowanie elementów tablicy. Najpierw zostaje rozpoznany adres pierwszego elementu tablicy, a dopiero później wyliczony adres elementu, do którego chcemy się odwołać.

A jak wygląda wypełnianie tablicy za pomocą wskaźnika? Zdefiniujemy wskaźnik na zmienną będącą liczbą całkowitą (bo takiego typu są elementy tablicy w naszym przykładzie), ustawiemy go na pierwszym elemencie tablicy, a następnie będziemy go tylko przesuwać wzduż tablicy, bez potrzeby odnajdywania za każdym razem początku tablicy. Napiszmy od razu cały program, w którym wypełnimy tablicę za pomocą wskaźnika i odczytamy ją również za pomocą wskaźnika.

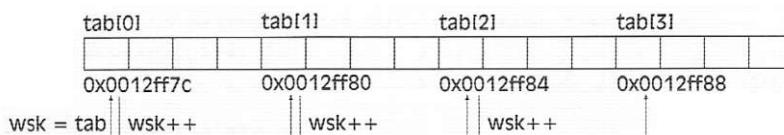
Poruszanie się po tablicy za pomocą wskaźnika

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

int main()
{
    int tab[4];
    int *wsk; // definicja wskaźnika na zmienne typu integer
    wsk = tab; // wskaźnik wskazuje na pierwszy element tablicy
    srand(time(NULL)); // uruchomienie generatora liczb pseudolosowych
    for (int i=0; i<4; i++)
    {
        *wsk = rand()%10; // nadajemy wartość elementowi, na który wskazuje
        wsk++; // wskaźnik, i przesuwamy go na następny element
    }
    cout << "Kolejne elementy tablicy: " << endl;
    wsk = tab; // znów ustaviamy wskaźnik na pierwszy element
    for (int i=0; i<4; i++)
    {
        cout << *wsk << " "; // odczytujemy wartość elementu, na który wskazuje
        wsk++; // wskaźnik, i przesuwamy się do następnego elementu
    }
    wsk = NULL; // ustaviamy wskaźnik na adres pusty
    getchar();
    return 0;
}
```

8.7

Po zakończeniu pracy ze wskaźnikiem nadaliśmy mu wartość **NULL** - jest to tak zwany adres pusty.



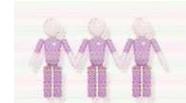
Ryc. 8.2. Fragment obszaru pamięci z tablicą tab, w której każdy element zajmuje 4 komórki pamięci, wraz z ilustracją inkrementacji wskaźnika wsk.

Jeśli na przykład nadajemy wartość /-temu elementowi tablicy, to aby wypełnić kolejny element, inkrementujemy wskaźnik bez potrzeby odnajdywania adresu pierwszego elementu tablicy (ryc. 8.2), jak to robiliśmy przy obsłudze tablicy bez użycia wskaźnika.

Zacznijmy od przykładu:

### Przykład

W programie utworzymy listę trzech najlepszych uczniów w klasie, a następnie wyświetlimy ją na ekranie monitora.



Dla potrzeb programu zdefiniujemy trzypolową strukturę uczniów. Zadeklarujemy trzyelementową tablicę oraz wskaźnik, za pomocą którego wypełnimy zawartość tablicy, a następnie ją odczytamy.

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

struct uczen
{
    char imie[10];
    char nazwisko[15];
    float srednia_ocen;
};

int main()
{
    uczen tab[3];
    uczen *wsk;           // wsk - wskaźnik na zmienne typu uczen
    wsk = tab;             // ustawiamy wskaźnik na początek tablicy
    for (int i=0; i<3; i++)
    {
        cout << "Podaj imię ucznia: ";
        // wypełnimy tablicę, odwołując się do pól struktury za pomocą wskaźnika
        cin >> wsk->imie;
        cout << "Podaj nazwisko ucznia: ";
        cin >> wsk->nazwisko;
        cout << "Podaj średnią ocen ucznia: ";
        cin >> wsk->srednia_ocen;
        wsk++;
    }
    wsk = tab;             // ponownie ustawiamy wskaźnik na początek tablicy
    cout << "Oto najlepsi uczniowie w klasie: " << endl;
    for (int i=0; i<3; i++)
    {
        // odczytamy tablicę, odwołując się do pól struktury za pomocą wskaźnika
        cout << wsk->imie << " ";
        cout << wsk->nazwisko << " ";
        cout << wsk->srednia_ocen << endl;
        wsk++;
    }
    cin.ignore();
    getchar();
    return 0;
}
```

8.9

## 8. Typ wskaźnikowy - zastosowania

Zauważ, że użycie wskaźników wcale nie utrudniło nam zadania, wręcz przeciwnie, przyspieszyliśmy poruszanie się pomiędzy elementami tablicy.



### Pytania kontrolne

1. Podaj sposób deklarowania wskaźnika.
2. W jaki sposób nadaje się wskaźnikowi wartość początkową? Podaj przykład.
3. W jaki sposób można zmienić zawartość obszaru pamięci, na jaki wskazuje zmienna wskaźnikowa? Podaj przykład.
4. Na jaki adres wskazuje nowo zadeklarowany wskaźnik?
5. Czy wskaźnik na zmienne typu całkowitego może zostać użyty do wskazania na zmienną typu rzeczywistego? Odpowiedź uzasadnij.

### Ćwiczenia

1. Przeanalizuj poniższy fragment kodu i podaj, jakie wartości zostaną wyświetcone na ekranie monitora:

```
int a, b;
int *wsk1, *wsk2;
wsk2 = &a;

a = 5;
b = 6;
wsk1 = wsk2;
b = *wsk1+3;
wsk1 = wsk2+2;
cout << wsk1;
cout << wsk2;
```
2. Wypełnij tablicę dziesięcioletową losowymi wartościami z przedziału (5, 11). Z wykorzystaniem zdefiniowanego wskaźnika, znając rozmiar tablicy, odczytaj jej elementy w odwrotnej kolejności, aniżeli zostały zapisane.
3. Dwie dwudziestoelementowe tablice wypełnij wartościami losowymi z przedziału (0, 7). Za pomocą dwóch zdefiniowanych wskaźników wyświetl elementy obydwu tablic oraz zbadaj, ile jest par elementów, które znajdują się na tej samej pozycji w tablicach i mają tę samą wartość.

## 9. Zmienne i struktury dynamiczne

W tym rozdziale przedstawimy, czym są zmienne i struktury dynamiczne. Zapoznamy cię ze sposobem definiowania tablic, których rozmiar jest ustalany dopiero w trakcie działania programu. Potem omówimy listy jedno- i dwukierunkową, wraz z ich podstawową obsługą. Prezentując zastosowanie zmiennych i struktur dynamicznych w omawianych przykładach, wskażemy w jakich przypadkach warto po nie siegnać.

Zmienne dynamiczne są to zmienne, które tworzymy w trakcie działania programu za pomocą operatora `new`. Usuwa się je operatorem `delete`. Czas ich występowania w programie jest uzależniony od potrzeb programu, zajmują więc miejsce w pamięci tylko wówczas, gdy są wykorzystywane - w przeciwnieństwie do zmiennych statycznych, które istnieją od początku do końca trwania programu.

Zmienna dynamiczna nie ma własnej nazwy, dlatego dostęp do niej jest możliwy wyłącznie przez adres obszaru pamięci, który ona zajmuje. Adres ten jest przechowywany we wskaźniku do tej zmiennej. Jeśli zadeklarujemy wskaźnik do zmiennej dynamicznej, to najlepiej jest nadać mu wartość początkową `NULL`, czyli przypisać-mu adres zerowy. W ten sposób można się zabezpieczyć przed wykonaniem operacji na obszarze pamięci, którego zmiana wartości nie była zamierzona.

Operatory `new`  
oraz `delete`

### 9.1. Tablica dynamiczna jednowymiarowa

Analizowaliśmy już wiele programów, które wykorzystują tablice. Pewnym utrudnieniem była wówczas konieczność określenia rozmiaru tablicy jeszcze przed komplikacją. Często rozmiar tablicy musiał zostać ustalony, zanim wiadomo było, jak dłuża tablica będzie potrzebna. Stąd niejednokrotnie okazywała się ona za dłuża lub za krótka. Ponadto raz zdefiniowana tablica statyczna (czyli taka, jaką wykorzystywaliśmy do tej pory) zajmuje przydzielony jej obszar pamięci aż do końca działania programu, choć często zdarza się, że po zastosowaniu tablicy jest ona niepotrzebna podczas dalszej pracy programu. Przy pisaniu krótkich programów, niewymagających zbyt wiele dodatkowej pamięci, nie ma to wielkiego znaczenia. Jednak dla dużych projektów i programów warto mieć możliwość usunięcia tablicy, z której program już nie korzysta. "Warunki te spełnia tablica dynamiczna.

Tablice, których  
ilość elementów  
ustala się w trakcie  
działania programu

## Definicja

**Tablica dynamiczna** to taka tablica, która zostaje utworzona w trakcie działania programu i może być usunięta przed jego zakończeniem.

W przeciwieństwie do zwykłych tablic statycznych, które istnieją przez cały czas działania programu, tablice dynamiczne są definiowane w trakcie jego działania, w zależności od potrzeb użytkownika. Powinno się je usuwać przed zakończeniem działania programu, w celu zwolnienia obszarów pamięci przez nie zajmowanej.

Podobnie jak w wypadku tablic statycznych, tablice dynamiczne mogą przechowywać elementy jednego typu. Aby móc w programie zadeklarować tablicę, musimy najpierw zadeklarować wskaźnik zdolny wskazywać element tablicy, a następnie za jego pomocą dynamicznie **alokować pamięć**, czyli rezerwować obszar przeznaczony na przechowywanie tego elementu. Na żądanie napisanego przez nas programu system operacyjny przyznaje nam obszar pamięci, w którym będą przechowywane dane, w takiej ilości, w jakiej jest nam aktualnie potrzebny. Należy jednak pamiętać, aby uzyskane w ten sposób dynamicznie alokowane obszary pamięci zwolnić przed zakończeniem działania programu.

Deklaracja jednowymiarowej tablicy dynamicznej

Deklaracja dynamicznej tablicy elementów typu `float`:

```
float *tab = NULL; // na razie wskaźnik na nic nie wskazuje  
tab = new float [k];
```

Obydwie instrukcje często łączy się w jedną:

```
float *tab = new float [k];
```

gdzie: `tab` to wskaźnik do obiektu typu `float` (oczywiście nazwa dowolna), `new` jest operatorem służącym do alokacji pamięci (słowo kluczowe), a `k` - rozmiarem tablicy (może być liczbą wpisaną z klawiatury). Wskaźnik `tab` wskazuje pierwszy element tablicy (element o indeksie 0).



### Przykład

Program pyta użytkownika, ile liczb chce wprowadzić do tablicy, następnie wypełnia tablicę podanymi liczbami i wyświetla ją.

Wiemy już, jak utworzyć tablicę dynamiczną. Korzystamy z niej zaś tak, jak z tablic statycznych. Choć tablica jako zmienna dynamiczna nie ma właściwej nazwy, to wskaźnikiem, który posłużył do jej utworzenia, możesz się posługiwać jak nazwą tablicy statycznej (według zasady obowiązującej w C++: nazwa tablicy jest wskaźnikiem do jej elementu o indeksie 0). Poprawna jest na przykład instrukcja: `tab[3] = 12;`, która oznacza: elementowi tablicy o indeksie 3 (czwarty element tablicy dynamicznej) przypisz wartość 12. W poniższym kodzie omawianego programu przeanalizuj sposób poruszania się po tablicy dynamicznej za pomocą wskaźnika. Wykorzystamy go do wy-

pełnienia i wyświetlenia tablicy tab. Pojawia się tu wyrażenie `tab+i`, gdzie `tab` jest wskaźnikiem, a `i` liczbą typu `integer`. Zapis taki oznacza, że do wskaźnika `tab` dodajemy `i` razy rozmiar typu, na jaki on wskazuje. W naszym przypadku „przesuwamy” się o `i` miejsc w tablicy `tab`.

```
#include <iostream>
#include <iomanip>
#include <cstdio>
#include <new>           // dodajczamy bibliotekę, w której jest
using namespace std;    // zdefiniowany sposób postępowania w razie braku
                        // miejsca na utworzenie tablicy

int main()
{
    float *tab = NULL; // deklaracja wskaźnika i ustawienie jego
                        // wartości na adres zerowy, czyli NULL
    cout << "Ile liczb wpisziesz do tablicy? ";
    int ile;
    float liczba;
    cin >> ile;
    try
    {
        tab = new float [ile]; // próbujemy utworzyć nową tablicę
                               // za pomocą operatora new
    }
    catch(bad_alloc)
    {
        cout << "Brak miejsca na utworzenie tablicy";
        cin.ignore();
        getchar();
        return -1;
    }
    for (int i=0; i<ile; i++) // jeśli utworzenie powiodło się
    {
        cout << "Podaj liczbę: ";
        cin >> liczba;
        *(tab+i) = liczba; // wypełniamy komórkę tablicy
    }
    cout << endl << "Wypisuje zawartość tablicy:" << endl;
    for (int i=0; i<ile; i++)
        cout << setw(6) << *(tab+i);
    delete [] tab; // usuwamy tablicę z pamięci
    cin.ignore(); // za pomocą operatora delete
    getchar();
    return 0;
}
```

Jeśli w programie, korzystając z zapasu pamięci, zamierzamy utworzyć bardzo dużą tablicę, wówczas możemy nie znaleźć wystarczająco dużego, spójnego obszaru na jej przechowanie. W takim wypadku wskaźnik `tab` zamiast wskazywać na początek tablicy, wskazywał będzie na adres `NULL`. Należy sprawdzać, czy operacja utworzenia tablicy się udała. W tym celu dołączliśmy bibliotekę `new`. Zdefiniowana jest tam funkcja zgłaszająca błąd alokacji pamięci (`bad_alloc`). W bloku `try` (czyli: próbuj) umieszczałyśmy funkcje, które chcemy śledzić pod ka-

9.1

Biblioteka  
<new>

tem wystąpienia błędów. W naszym programie jest to błąd związany z nieudaną alokacją pamięci za pomocą operatora new. Instrukcja catch przechwytuje przypadki, kiedy funkcja wewnątrz bloku try się nie powiodła. Instrukcję catch (bad\_alloc) można interpretować jako „jeśli wykryłeś błąd alokacji pamięci, to wykonaj instrukcję w klamrze”.

A oto efekt działania programu:

```
Ile liczb wpiszesz do tablicy? 4
Podaj liczbe: 2.4
Podaj liczbe: 1
Podaj liczbe: -4
Podaj liczbe: 4.6

Wypisuje zawartosc tablicy:
2.4 1 -4 4.6
```

Zauważysz w kodzie programu, że choć wskaźnik tab posłużył nam do zapełnienia tablicy i jej wypisania, to jego wartość w całym programie nie uległa zmianie. Nie przesuwaliśmy wskaźnika tab, a jedynie w instrukcji \*(tab+i) = liczba; umieszczonej w pętli wpisywaliśmy liczby do komórek odpowiednio przesuniętych względem komórki, na którą wskazuje wskaźnik tab:

**Jeśli przypadkową instrukcją przypisania zmienisz wartość wskaźnika wskazującego na tablicę dynamiczną, możesz stracić możliwość korzystania z tej tablicy.**

#### Usuwanie tablicy z pamięci

Załóżmy, że po wyprowadzeniu wartości elementów tablicy na ekran monitora nie będziemy już korzystać z naszej dynamicznej tablicy. Chcemy więc zwolnić obszar pamięci, który wykorzystywaliśmy, a w dłuższym programie niż nasz przykładowy moglibyśmy użyć tego obszaru ponownie. Zwolnienia pamięci dokonuje się za pomocą operatora delete:

```
delete [] tab;
```

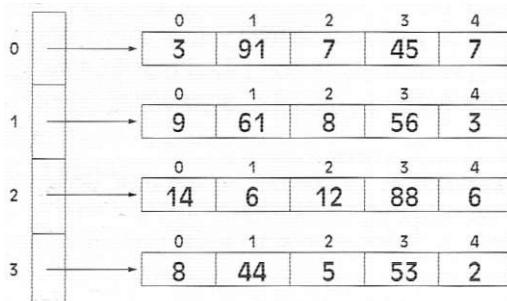
Powyższa linia zwalnia pamięć zajętą przez tablicę obsługiwana za pomocą wskaźnika tab. Po takiej instrukcji wskaźnik tab oczywiście nadal istnieje, ponieważ wskaźnik jest zmienną statyczną, której nie da się usunąć z programu.

## 9.2. Tablica dynamiczna dwuwymiarowa

Zagadnienie tablicy dwuwymiarowej wymaga umiejętności zadeklarowania wskaźnika do wskaźnika. Tworzyć będziemy tablicę tablic, co oznacza, że w naszej interpretacji dwuwymiarowa tablica dynamiczna

## 9.2. Tablica dynamiczna dwuwymiarowa

będzie jednowymiarową tablicą dynamiczną, przechowującą wskaźniki do tablic dynamicznych jednowymiarowych, w których umieszczone zostaną dane. Graficznie sytuację tę przedstawia rycina 9.1. Obsługujemy taką tablicę analogicznie do tablicy statycznej, z tym że nazwy używamy nazwy wskaźnika (z pierwszej dynamicznej tablicy jednowymiarowej).



Ryc. 9.1. Graficzna interpretacja tablicy dynamicznej dwuwymiarowej

Deklaracja dwuwymiarowej tablicy dynamicznej [w] [k] liczb całkowitych wygląda następująco:

```
int **tab;           // tab jest wskaźnikiem do wskaźnika na zmienne
                     // typu int

tab = new int *[w]; // w - liczba wierszy fz klawiatury), tu jest
                     // tablica jednowymiarowa wskaźników do zmennych
                     // typu int

for (i=0; i<w; i++) // dla każdego wskaźnika z poprzedniej
    tab[i] = new int [k]; // tworzymy tablicę liczb całkowitych
                          // k - liczba kolumn
```

Zauważ, że najpierw tworzona jest jednowymiarowa tablica wskaźników, dopiero potem powstają tablice, na które te wskaźniki będą wskaazywały. Zatem usuwanie z pamięci będzie się odbywać w odwrotnej kolejności. Najpierw zwolnimy pamięć zajmowaną przez tablice przechowujące liczby, następnie tablicę wskaźników, które na te tablice wskazywały.

### Przykład

Napiszemy program, który utworzy dwuwymiarową tablicę dynamiczną o żądanach wymiarach (podanych z klawiatury) i wypełni ją losowymi liczbami z przedziału (0, 100).

Deklaracja  
dwuwymiarowej  
tablicy  
dynamicznej



Oto kod programu:

```
#include <iostream>
#include <iomanip>
#include <cstdio>
#include <cstdlib>
#include <new>
using namespace std;

int main()
{
    int wie, kol, i, j;
    cout << "Podaj liczbę wierszy i kolumn tablicy:\n";
    cin >> wie >> kol;
    srand(time(NULL));           // inicjacja generatora liczb
    int **tab;                  // deklaracja wskaźnika do wskaźnika na zmienne typu int
    try                         // próba tworzenia tablicy wskaźników
    {
        tab = new int *[wie];
    }
    catch(bad_alloc)
    {
        cout << "Brak miejsca na utworzenie tablicy. Konczy program";
        return -1;
    }
    for (i=0; i<wie; i++)         // próba tworzenia dynamicznych tablic liczb
    {
        try
        {
            tab[i] = new int [kol];
        }
        catch(bad_alloc)
        {
            cout << "Brak miejsca na utworzenie tablicy. Konczy program";
            return -1;
        }
    }
    for (i=0; i<wie; i++)
        for (j=0; j<kol; j++)
    {
        tab[i][j] = rand()%101;   // wypełnianie tablic liczbami
        cout << setw(4) << tab[i][j]; // wyświetlanie elementów
    }
    cout << endl;
    // teraz usuniemy tablice z obszaru zajmowanej pamięci
    for (i=0; i<wie; i++)
        delete [] tab[i];       // usuwamy kolejne tablice z liczbami
    delete [] tab;              // usuwamy tablicę wskaźników
    cin.ignore();
    getchar();
    return 0;
}
```

9.2

W powyższym programie tworzymy tablicę wskaźników do tablic jednowymiarowych. Jeśli zabraknie miejsca w pamięci na utworzenie któ-

## 9.2. Tablica dynamiczna dwuwymiarowa

rejkolwiek z nich, wówczas program zakończy działanie, zgodnie z tokiem postępowania omówionym wcześniej (tu również sprawdzaliśmy dostępność obszaru pamięci na zapis tablicy).

Wiemy, że tablica dwuwymiarowa jest tablicą wskaźników do jednowymiarowych tablic przechowujących dane. Poszczególne tablice z danymi nie muszą jednak mieć takiego samego rozmiaru. W przykładzie pierwszy wskaźnik będzie wskazywał na tablicę jednoelementową, następny na tablicę dwuelementową itd. Dopiero ostatni wskaźnik wskazuje na tablicę o takiej liczbie elementów, jaką ma tablica wskaźników.

Przyjrzyjmy się kolejnemu przykładowi.

### Przykład

Napiszemy program, który utworzy tabelę odległości pomiędzy miastami. Nazwy miast oraz odległości pomiędzy nimi podaje użytkownik.



Zapewne niejednokrotnie, chcąc poznać odległość pomiędzy dwoma miastami w Polsce, korzystałeś z tabeli podanej w atlasie samochodowym. Przykładowo dla czterech miast tabela taka może mieć następujący wygląd:

	Kraków	Warszawa	Zakopane	Katowice
Kraków	0	295	100	75
Warszawa	295	0	402	297
Zakopane	100	402	0	156
Katowice	75	297	156	0

Tab. 9.1. Tabela odległości pomiędzy miastami

Zwykle jednak część pól odległości pozostaje niewypełniona, gdyż na przykład odległość z Krakowa do Warszawy jest taka sama jak z Warszawy do Krakowa:

	Kraków	Warszawa	Zakopane	Katowice
Kraków	0			
Warszawa	295	0		
Zakopane	100	402	0	
Katowice	75	297	156	0

Tab. 9.2. zmodyfikowana tabela odległości pomiędzy miastami

W naszym programie oprócz tablicy zamieszczającej odległości zdefiniujemy również tablicę dynamiczną, do której wpiszemy nazwy miast w liczbie podanej przez użytkownika. Dla przejrzystości odczytu w odpowiednich miejscach wyświetcone zostaną nazwy miast.

## 9. Zmienne i struktury dynamiczne

```
#include <iostream>
#include <iomanip>
#include <cstdio>
#include <new>
using namespace std;

int main()
{
    int liczba, i, j;           // liczba - zmienna przechowująca liczbę miast
    cout << "Podaj liczbę miast: ";
    cin >> liczba;
    char **miasta;             // deklarujemy tablicę dwuwymiarową o nazwie miasta,
    miasta = new char *[liczba]; // w której poszczególne wiersze zawierają
    for (i=0; i<liczba; i++)   // nazwy miast (będące tablicami znaków)
        miasta[i] = new char[20];
    for (i=0; i<liczba; i++)
    {
        cout << "Podaj nazwę miasta: ";
        cin >> miasta[i];
    }
    int **tab;      // deklarujemy tablicę odległości między miastami
    tab = new int *[liczba];
    for (i=0; i<liczba; i++)
        try                    // próba utworzenia tablicy
    {
        tab[i] = new int[i+1];
    }
    catch(bad_alloc)          // jeśli brak miejsca na tablicę,
    {
        return -1;              // to zakończ działanie programu
    }
    for (i=0; i<liczba; i++)
        for (j=0; j<i+1; j++)
            if (i==j)
                tab[i][j]=0;
            else
            {
                cout << "Podaj odległość z miasta " << miasta[i];
                cout << " do miasta " << miasta[j] << ": ";
                cin >> tab[i][j];
            }
    cout << "";
    for (i=0; i<liczba; i++)
        cout << setw(10) << miasta[i];
    cout << endl;
    for (i=0; i<liczba; i++)
    {
        for (j=0; j<i+1; j++)
        {
            if (j==0)
                cout << setw(10) << miasta[i];
            cout << setw(10) << tab[i][j];
        }
        cout << endl;
    }
}
```

### 9.3. Lista jednokierunkowa - tworzenie listy i wprowadzanie do niej elementów

```
for (i=0; i<liczba; i++)
    delete [] tab[i];
delete [] tab;
delete [] miasta;
cin.ignore();
getchar();
return 0;
}
```

9.3

Poniżej przedstawiamy przykładowy efekt działania programu:

```
Podaj liczbe miast: 4
Podaj nazwe miasta: Krakow
Podaj nazwe miasta: Warszawa
Podaj nazwe miasta: Zakopane
Podaj nazwe miasta: Katowice
Podaj odleglosc z miasta Warszawa do miasta Krakow: 295
Podaj odleglosc z miasta Zakopane do miasta Krakow: 100
Podaj odleglosc z miasta Zakopane do miasta Warszawa: 402
Podaj odleglosc z miasta Katowice do miasta Krakow: 75
Podaj odleglosc z miasta Katowice do miasta Warszawa: 297
Podaj odleglosc z miasta Katowice do miasta Zakopane: 156
```

	Krakow	Warszawa	Zakopane	Katowice
Krakow	0			
Warszawa	295	0		
Zakopane	100	402	0	
Katowice	75	297	156	0

Tak przygotowaną tabelicę możesz wykorzystać na przykład do znalezienia miast, pomiędzy którymi odległość jest najmniejsza (największa). Jako ćwiczenie dodatkowe polecamy napisanie funkcji wyświetlającej nazwy miast położonych najbliżej siebie (przydatne wskazówki znajdziesz w rozdziale poświęconym poszukiwaniu minimum liczb).

Poznaliśmy już typ zmiennej, jaką jest tablica dynamiczna - kolejna dynamiczna struktury danych.

### 9.3. Lista jednokierunkowa - tworzenie listy i wprowadzanie do niej elementów

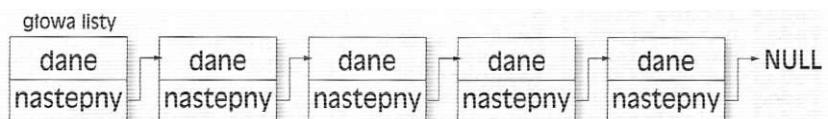
Dynamiczna struktura danych jest ciągiem powiązanych ze sobą zmiennych dynamicznych, których liczba nie jest znana w czasie komplikacji, a nawet podczas uruchamiania programu. Lista jest złożoną dynamiczną strukturą danych, która przechowuje dane wraz z zapamiętaniem ich kolejności. W przeciwieństwie do tablicy elementy listy nie muszą być położone w sąsiednich komórkach pamięci. Cechą ta sprawia, że dla rozbudowanych projektów lista jest dobrym rozwiązaniem, gdy chcemy na przykład zapamiętać dużą liczbę elementów, a nie dysponujemy wystarczająco dużym spójnym obszarem na utworzenie tablicy dynamicznej. Jednak główną i najważniejszą cechą listy jest możliwość dołączania elementów bez zdefiniowania uprzednio ich liczby. Nawet w przypadku tablicy dynamicznej po podaniu rozmiaru nie możemy już jej powiększyć.

Zalety użycia  
w programie listy  
dynamicznej

## 9. Zmienne i struktury dynamiczne

### Opis budowania jednokierunkowej listy dynamicznej

Tworzenie listy zaczynamy od zdefiniowania jej pierwszego elementu, czyli głowy (ang. *head*). Głowa jest jednocześnie identyfikatorem listy, dlatego w naszych programach również wskaźnik do pierwszego elementu listy będziemy nazywać głową. Do głowy dołączamy następne elementy. Dla każdego kolejnego elementu listy będzie znajdowany wolny obszar pamięci, do którego zostaną wpisane dane z nim związane. W wypadku listy jednokierunkowej każdy element zna adres elementu następnego, natomiast nie zna adresu elementu poprzedniego (ryc. 9.2).



Ryc. 9.2. Pięcioelementowa lista jednokierunkowa

Elementami listy są struktury. Każda ze struktur wchodzących w skład listy musi mieć, oprócz pól z danymi, dodatkowe pole, w którym jest przechowywany wskaźnik do następnego elementu listy. Ostatni element listy wskazuje na adres zerowy, niezwiązany z żadnym obiektem, czyli na **NULL**. Taki adres jest informacją, że nie ma już w liście więcej elementów.

Lista jest strukturą o dostępie sekwencyjnym, co oznacza, że aby dostać się na przykład do czwartego elementu listy, należy „przejść” wszystkie wcześniejsze elementy, począwszy od głowy. Natomiast w tablicy mamy do czynienia z dostępem swobodnym, czyli wystarczy podać indeks elementu, aby uzyskać do niego dostęp.

Zacznijmy od zadeklarowania prostej struktury, która posłuży nam do zbudowania listy:

```
struct element
{
    int liczba;
    element *nastepny;
};
```

Struktura ta ma tylko jedno pole na przechowywanie danych i nazywa się ono **liczba**. Drugie pole jest wskaźnikiem do zmiennej typu **element** - wskaźnik ten posłuży nam do zbudowania listy. Każdy element listy (struktury) tworzy się za pomocą znanego ci już operatora new.

W listach mamy dwa zasadnicze sposoby dołączania kolejnych elementów: **dołączanie do początku listy** i **dołączanie do końca listy**.

Przedstawimy teraz pierwszy ze sposobów za pomocą przykładu.

### Przykład

Napiszmy program, który zadaje pytanie, czy chcemy dodać element do listy. Dopóki otrzymuje pozytywną odpowiedź, pobiera wpisaną na klawiaturze liczbę całkowitą i dołącza ją do listy. Na koniec wyświetla na ekranie monitora całą zawartość listy.

### 9.3. Lista jednokierunkowa - tworzenie listy i wprowadzanie do niej elementów

Zaczniemy od zdefiniowania struktury elementu listy, która będzie przechowywać wprowadzane liczby. Strukturę tę nazwaliśmy **element\_listy**. Oto jej definicja:

```
struct element_listy
{
    int liczba;
    element_listy *nastepny;
};
```

Musimy w naszym programie zadeklarować wskaźnik głowa, który będzie wskazywał na początek listy:

```
element_listy *glowa;
```

1. Dopóki w naszej liście nie ma jeszcze żadnego elementu, głowa powinna wskazywać na NULL:

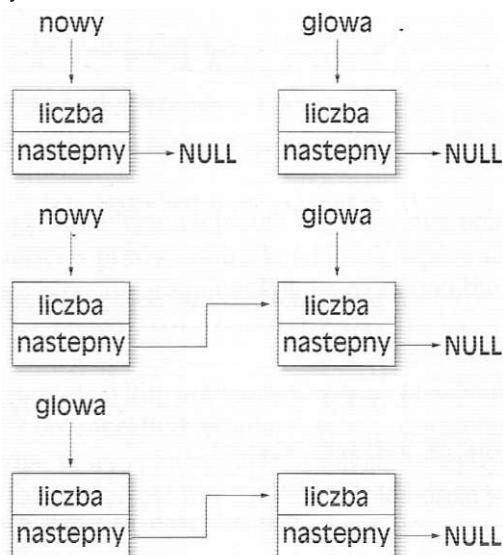
```
glowa = NULL;
```

Taki zapis oznacza, że lista jest pusta: **glowa = NULL**

2. Teraz możemy już dodawać kolejne elementy listy według następujących instrukcji:

- a) operatorem **new** alokujemy pamięć na nowy element listy,
- b) wprowadzamy dane do przydzielonego obszaru pamięci,
- c) wskaźnik następny nowego elementu ustawiamy na element wskazywany przez wskaźnik głowa,
- d) wskaźnik głowa ustawiamy, aby wskazywał nowo utworzony element listy.

Dołączanie kolejnych elementów do początku listy



Ryc. 9.3. Dołączanie elementu do początku listy

Na rycinie 9.3 przedstawiliśmy dołączanie nowego elementu do istniejącej listy jednoelementowej. Identyczne kroki należy wykonać, gdy

## 9. Zmienne i struktury dynamiczne

lista jest pusta lub ma więcej elementów. Kroki te możemy zapisać w postaci kodu w C++ (zakładamy, że mamy zadeklarowany wskaźnik nowy):

```
nowy = new element_listy; // alokacja pamięci
cout << "Podaj liczbę: ";
cin >> nowy->liczba; // wprowadzenie pola danych
nowy->nastepny = głowa; // nowy element będzie wskazywał na
                         // ten sam adres co głowa
głowa = nowy;
```

W powyższych instrukcjach pojawił się operator odwołania (wyłuskania) do pola struktury, jest nim `->` (do pól struktury, na którą wskazuje wskaźnik, odwołujemy się przez operator `->`, a nie przez operator kropkę, przydatny w sytuacji, kiedy struktura miała swoją nazwę).

Podamy teraz całość programu tworzącego listę z dopisywaniem elementów do początku i wypisującego utworzoną listę:

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

struct element_listy
{
    int liczba;
    element_listy *nastepny;
};

int main()
{
    element_listy *glowa = NULL; // tworzymy listę pustą
    element_listy *nowy;
    char odp;
    int d;
    cout << "Czy chcesz podać element listy? t/n: ";
    cin >> odp; // dodajemy kolejne elementy
    while (odp != 'n')
    {
        cout << "Podaj dane: ";
        cin >> d;
        try // próba dołączenia nowego elementu
        {
            nowy = new element_listy;
        }
        catch(bad_alloc) // jeśli nie ma miejsca na nowy element
        {
            cout << "Nie ma już miejsca na następny element";
            break;
        }
        nowy->nastepny = głowa; // jeśli jest miejsce na nowy element
        nowy->liczba = d;
        głowa = nowy;
        cout << endl << "Chcesz podać kolejny element listy? t/n: ";
        cin >> odp;
    }
    element_listy *temp = głowa; // początek wypisywania elementów listy
    cout << endl << "Oto utworzona przez ciebie lista:" << endl;
```

### 9.3. Lista jednokierunkowa - tworzenie listy i wprowadzanie do niej elementów

```
while (temp!=NULL)
{
    cout << temp->liczba << " ";
    temp = temp->nastepny;
}
cin.ignore();
getchar();
return 0;
}
```

9.4

Przy dołączaniu kolejnych elementów do listy musimy pamiętać, że obszar pamięci jest obszarem ograniczonym, może więc zdarzyć się sytuacja, że nie ma już miejsca na umieszczenie kolejnego elementu listy. W tej sytuacji w naszym programie zostanie wyprowadzona na ekran monitora informacja o braku miejsca oraz wyświetlona zostanie cała dotychczas wy pełniona lista. Można ten problem rozwiązać na wiele innych sposobów, przy czym odpowiednią instrukcję należy umieścić w klamrach bloku, występującego po instrukcji `catch (bad_alloc)`. W naszym przykładzie przerywamy działanie pętli instrukcją `break` (tu jest jedno z uzasadnionych zastosowań tej instrukcji) oraz wypisujemy aktualną zawartość listy.

Obstuga  
przypadku braku  
pamięci na  
dołaczanie  
elementu do listy

Oto przykładowy wynik działania programu:

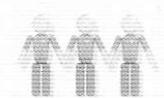
```
Czy chcesz podac element listy? t/n: t
Podaj dane : 4
Chcesz podac kolejny element listy? t/n: t
Podaj dane : 12
Chcesz podac kolejny element listy? t/n: t
Podaj dane : 8
Chcesz podac kolejny element listy? t/n: n
Oto utworzona przez ciebie lista:
8 12 4
```

Zwróć uwagę, że kolejne elementy listy są w niej umieszczone w odwrotnej kolejności w porównaniu z kolejnością wprowadzania.

Zajmijmy się teraz dołączaniem kolejnych elementów do końca listy.

#### Przykład

Napiszmy program, który pobiera wpisane na klawiaturze liczby całkowite i po każdej pobranej liczbie zadaje pytanie, czy zostanie wprowadzona kolejna liczba. W wypadku odpowiedzi negatywnej wyświetla na ekranie monitora całą zawartość listy zgodnie z kolejnością wprowadzania.



Lista jest strukturą danych o dostępie sekwencyjnym, zatem aby dołączyć nowy element do końca listy, należy przejść od głowy przez całą listę, odnaleźć jej koniec i do niego podłączyć kolejny element. Sposób ten jest niezmiernie powolny, zwłaszcza dla długich list. Istnieje jednak metoda prostsza: wystarczy na koniec listy skierować dodatkowy wskaź-

## 9. Zmienne i struktury dynamiczne

nik, który zawsze będzie wskazywał na ostatni element. Wskaźnik ten nazwiemy: **ogon** (ang. *tail*) .

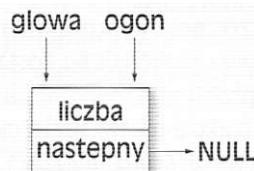
Proces budowy i dołączania kolejnych elementów tej listy rozbijemy na poszczególne kroki:

1. W **pustej liście na NULL skierowane są dwa wskaźniki: głowa i ogon:**

**głowa = NULL**

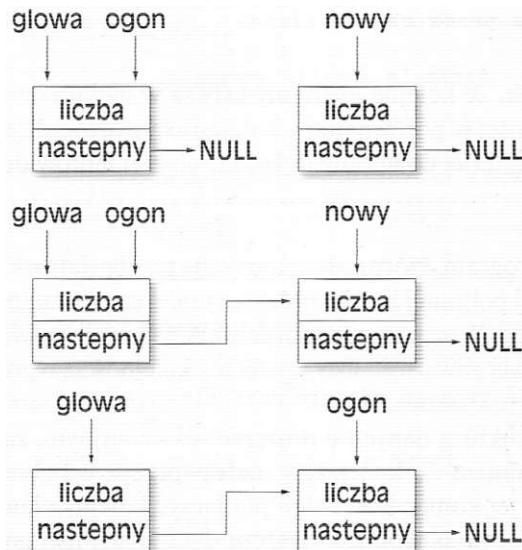
**ogon = NULL**

2. Na pierwszy element listy wskazuje zarówno wskaźnik głowa, jak i ogon - pierwszy element jest też ostatnim elementem (ryc. 9.4).



Ryc. 9.4. Usta jednoelementowa ze wskaźnikami głowa i ogon

3. Drugi i każdy kolejny element listy dodajemy według schematu:
  - a) operatorem new alokujemy pamięć na nowy element listy,
  - b) wprowadzamy dane do przydzielonego obszaru pamięci,
  - c) wskaźnik następny nowego elementu ustawiamy na **NULL**,
  - d) wskaźnik następny z elementu wskazywanego przez ogon ustawiamy na nowy element,
  - e) wskaźnik ogon ustawiamy, aby wskazywał nowo utworzony element listy.



Ryc. 9.5. Dołączanie kolejnych elementów do końca listy wskazywanego przez ogon

Na rycinie 9.5 zilustrowaliśmy dodanie kolejnego elementu do listy jednoelementowej. Dodawanie dalszych elementów będzie się odbywać według identycznego schematu; oczywiście, nie będzie już takiej sytuacji, że wskaźniki głowa i ogon wskazują ten sam element listy (jak to jest w liście jednoelementowej).

Zapiszmy zatem w języku C++ kroki prowadzące do zbudowania listy z kolejnych elementów dołączanych do końca listy:

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

struct element_listy
{
    int liczba;
    element_listy *nastepny;
};

int main()
{
    element_listy *glowa = NULL;
    element_listy *ogon = NULL;
    element_listy *nowy;
    char odp;
    int d;
    cout << "Czy chcesz podać element listy? t/n: ";
    cin >> odp;
    while (odp != 'n')
    {
        cout << "Podaj dane: ";
        cin >> d;
        try
        {
            nowy = new element_listy;
        }
        catch(bad_alloc)
        {
            cout << endl << "Brak pamięci na ostatni element";
            break;
        }
        nowy->nastepny = NULL;
        nowy->liczba = d;
        if (glowa == NULL)
        {
            glowa = nowy;
            ogon = glowa;
        }
        else
        {
            ogon->nastepny = nowy;
            ogon = nowy;
        }
        cout << endl << "Czy chcesz podać kolejny element listy? t/n: ";
        cin >> odp;
    }
    element_listy *temp = glowa; // początek wypisywania elementów listy
    cout << endl << "Oto utworzona przez ciebie lista:" << endl;
```

## 9. Zmienne i struktury dynamiczne

```
while (temp!=NULL)
{
    cout << temp->liczba << " ";
    temp = temp->nastepny;
}
cin.ignore();
getchar();
return 0;
}
```

9.5

Dla czytelności kodu pominęliśmy obsługę wyjątkowej sytuacji, kiedy nie można utworzyć nowego elementu ze względu na brak miejsca w pamięci.

Wynik działania tego programu jest następujący:

```
Czy chcesz podać element listy? t/n: t
Podaj dane : 5
Czy chcesz podać kolejny element listy? t/n: t
Podaj dane : 8
Czy chcesz podać kolejny element listy? t/n: t
Podaj dane : 3
Czy chcesz podać kolejny element listy? t/n: n
Oto utworzona przez ciebie lista:
5 8 3
```

Jak widzisz, przy dołączaniu elementów do końca listy zachowana jest kolejność wpisywania danych.

### 9.4. Lista jednokierunkowa - usuwanie listy z pamięci

Zajmijmy się teraz problemem usuwania listy z pamięci komputera. Lista jako struktura dynamiczna może zostać usunięta z pamięci komputera, gdy przestaje już być potrzebna. Listę usuwa się element po elemencie; można to realizować na różne sposoby, ale w liście jednokierunkowej najlepiej zacząć od głowy. Przedstawmy kolejne czynności w postaci listy kroków:

1. Jeśli wskaźnik **głowa** wskazuje **NULL**, to zakończ.
2. Ustaw pomocniczy wskaźnik **temp** na adres, który wskazuje **głowa**.
3. Wskaźnik **głowa** ustaw do następnego elementu listy.
4. Usuń strukturę, na którą wskazuje wskaźnik **temp** i przejdź do kroku 1.

Oto odpowiedni fragment programu realizujący usuwanie listy z pamięci:

```
while (glowa!=NULL)
{
    temp = glowa;
    glowa = glowa->nastepny;
    delete temp;
}
```

Usuwanie z listy  
elementu  
spełniającego  
warunek

Zwolniony obszar pamięci można teraz wykorzystać ponownie.

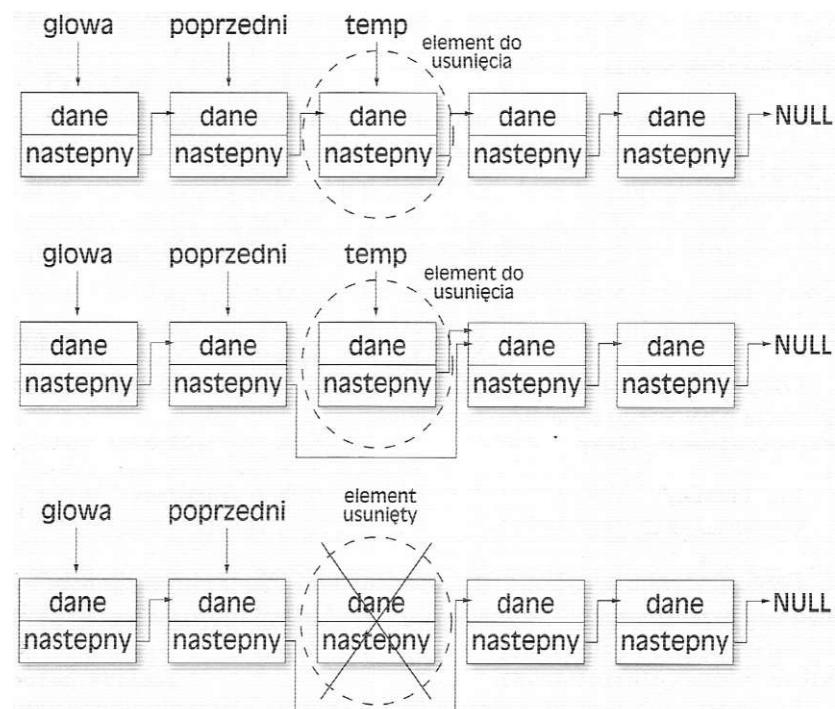
Problemem trudniejszym od usunięcia całej listy jest usunięcie jej do- wolnego elementu. Wyróżniamy tutaj dwa odrębne przypadki i w każ-

#### 9.4. Lista jednokierunkowa - usuwanie listy z pamięci

dym z nich postępujemy odmiennie. W obu sytuacjach przyjmujemy, że wskaźnik temp wskazuje na element do usunięcia, natomiast wskaźnik poprzedni wskazuje na poprzedni element listy - jeśli taki istnieje.

Przypadek 1. Jeśli szukanym elementem jest głowa, należy przesunąć wskaźnik głowa do następnego elementu listy i usunąć element, na który wskazuje wskaźnik temp.

Przypadek 2. Jeśli szukany element znajduje się wewnętrz lub na końcu listy, wtedy należy postąpić zgodnie ze schematem przedstawionym na rycinie 9.6.



Ryc. 3.6. Usuwanie wyróżnionego elementu z listy jednokierunkowej

Do usunięcia elementu, którego pole danych spełnia jakiś warunek, będziemy potrzebowali - jak widać z ryciną - dwóch pomocniczych wskaźników: temp i poprzedni.

Funkcja usun\_elem służy do usunięcia elementów listy o zadanej wartości. Wynikiem działania funkcji jest wskaźnik do listy (po usunięciu z niej elementów), natomiast jako argumenty funkcja pobiera wskaźnik do głowy listy i wartość elementu do usunięcia:

```
element_listy* usun_elem(element_listy *glowa, int wartosc)
{
    element_listy *temp = glowa;
    element_listy *poprzedni = glowa;
```

## 9. Zmienne i struktury dynamiczne

```
while (temp!=NULL)
{
    if (temp->liczba==wartosc)
        if (temp == glowa)
        {
            glowa = glowa->nastepny;
            delete temp;
            temp = glowa ;
            poprzedni = glowa;
        }
        else
        {
            poprzedni->nastepny = temp->nastepny;
            delete temp;
            temp = poprzedni->nastepny;
        }
    else
    {
        poprzedni = temp;
        temp = temp->nastepny;
    }
}
return glowa;
}
```

9.6

Oczywiście przed definicją tej funkcji musi się pojawić globalna deklaracja używanego typu strukturalnego:

```
struct element_listy
{
    int liczba;
    element_listy *nastepny;
};
```

Przy założeniu, że głowa jest wskaźnikiem przypisanym do listy istniejącej w programie głównym, wywołanie funkcji `usun_elem` w funkcji main będzie miało postać:  
`glowa = usun_elem(glowa, k)`

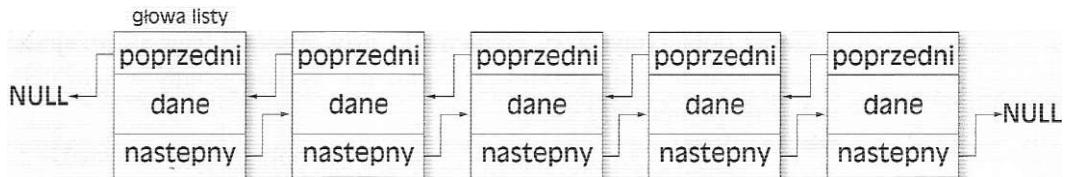
Przedstawiona funkcja usuwa z listy wszystkie elementy, które mają wartość równą wartości poszukiwanej. W podobny sposób dołączamy element do listy w dowolnym jej miejscu. Napisz jako ćwiczenie program dołączający element do listy po elemencie o zadanej wartości.

## 9.5. Lista dwukierunkowa

Charakterystyka  
listy  
dwukierunkowej

Różnica pomiędzy listą jednokierunkową a dwukierunkową polega na tym, że w liście dwukierunkowej mamy bezpośrednio dostęp za pomocą wskaźnika zarówno do elementu leżącego za danym elementem, jak i do elementu znajdującego się bezpośrednio przed nim. Operacje dokonywane na liście dwukierunkowej są analogiczne do tych, które wykonywaliśmy na liście jednokierunkowej, wzbogacone jednak o obsługę

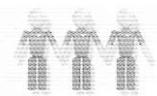
wskaźnika skierowanego na element poprzedni. Dzięki dodatkowemu wskaźnikowi (**poprzedni**) praca z listą dwukierunkową jest znacznie prostsza niż z listą jednokierunkową. Na rycinie 9.7 przedstawiliśmy schemat budowy listy dwukierunkowej:



Ryc. 9.7. Lista dwukierunkowa

Przy kła

Ustawiamy uczniów w rzędzie, notując ich imię i wzrost. Po ustawniu każdego następnego ucznia użytkownik programu odpowiada na pytanie, czy do rzędu dołączy kolejny uczeń. Jeśli nie, program wyświetla wszystkie dane, następnie wypisuje imiona uczniów, którzy są wyżsi od obydwu sąsiadów.



Analogicznie jak przy obsłudze listy jednokierunkowej zaczniemy od zdefiniowania struktury stanowiącej elementy listy:

```
struct uczeń
{
    char imie[20];
    int wzrost;
    uczeń *następny;
    uczeń *poprzedni;
};
```

W naszym przykładzie struktura jest czteropolowa: dwa pierwsze pola są polami danych, dwa pozostałe to wskaźniki skierowane na sąsiadujące elementy. Teraz utworzymy listę:

```
uczeń *głowa;
głową = new uczeń; // alokujemy dynamicznie pamięć na głowę listy
głową->następny = NULL;
głową->poprzedni = NULL;
```

Możemy zacząć dopisywanie kolejnych uczniów. Podobnie jak w liście jednokierunkowej dopisywanie elementów do końca listy jest bardzo uproszczone, gdy zadeklarujemy wskaźnik **ogon**, który stale wskazuje na ostatni element listy. Można też dodawać elementy listy do jej końca bez wskaźnika **ogon**. Aby dopisać dowolny element listy, podobnie jak w liście jednokierunkowej, posłużymy się zmienną pomocniczą **temp**, za pomocą której odnajdziemy adres ostatniego elementu w liście. Wskaźnik **następny** tego elementu, skierowany na **NULL**, przekierujemy na nowo utworzony element, a wskaźnik **poprzedni** nowo utworzonego elementu skierujemy na element, który do tej pory był elementem ostatnim:

Dopisywanie  
elementów  
do końca listy  
dwukierunkowej  
bez użycia  
wskaźnika  
na **ogon**

## 9. Zmienne i struktury dynamiczne

```
nowy = new uczen;           // alokacja pamięci na nowy element
cout << "Podaj imię ucznia: ";
cin >> nowy->imie;        // wprowadzenie pola danych: imię
cout << "Podaj wzrost ucznia: ";
cin >> nowy->wzrost;      // wprowadzenie pola danych: wzrost
nowy->nastepny = NULL;
```

Teraz dołączymy nowy element do listy, musimy więc ją przejrzeć w celu odnalezienia ostatniego elementu. Wykorzystamy w tym celu wskaźnik pomocniczy temp:

```
temp = głowa;
while (temp->nastepny!=NULL) temp = temp->nastepny;
```

Kiedy opuszczamy pętlę, temp->nastepny wskazuje na dotychczasowy ostatni element. Dołączymy do niego nowy element, a nowy element wskaźnikiem poprzedni skierujemy na element ostatni:

```
temp->nastepny = nowy;
nowy->poprzedni = temp;
```

Oczywiście w każdej chwili możemy wyświetlić elementy listy:

```
temp = głowa;
if (temp->nastepny!=NULL)
    do
    {
        temp = temp->nastepny;
        cout << temp->imię << " " << temp->wzrost << endl;
    }
    while ((temp->nastepny)!=NULL);
```

Teraz napiszemy funkcję, której zadaniem jest wprowadzenie na ekran imion osób umiejscowionych na liście pomiędzy dwoma osobami od nich niższymi. Zmienną pomocniczą temp skierujemy od razu na drugi element listy, ponieważ pierwszy element ma tylko jednego sąsiada, a więc nie może spełniać warunków zadania. Dla każdego elementu będziemy sprawdzać warunek dotyczący wzrostu aż do elementu, który jako ostatni znów będzie miał tylko jednego sąsiada:

```
uczen temp;
temp = głowa->nastepny;
while (temp->nastepny!=NULL)
{
    if (temp->poprzedni->wzrost<temp->wzrost && temp->nastepny
        ->wzrost<temp->wzrost)
        cout << temp->imię << endl;
    temp = temp->nastepny;
}
```

>  
Usuwanie środkówowych elementów z listy dwukierunkowej odbywa się łatwiej niż usuwanie z listy jednokierunkowej, ponieważ zawsze mamy dostęp zarówno do elementu poprzedniego, jak i następnego. Natomiast elementy krańcowe wymagają odrębnego potraktowania. Aby usunąć taki element z listy, musimy skierować na niego pomocniczy wskaźnik temp i odpowiednio przekierować wskaźniki następny i poprzedni,

Usuwanie  
elementu z listy  
dwukierunkowej

a w wypadku, gdy usuwanym elementem listy jest pierwszy element, należy przesunąć wskaźnik głowa. Zatem przy usuwaniu wskazanego elementu z listy dwukierunkowej musimy rozpatrzyć trzy przypadki (zakładamy, że wskaźnik temp pokazuje element do usunięcia):

1. Usuwany element jest pierwszy w liście:

```
glowa = glowa->nastepny;
glowa->poprzedni = NULL;
delete temp;
temp = glowa;
```

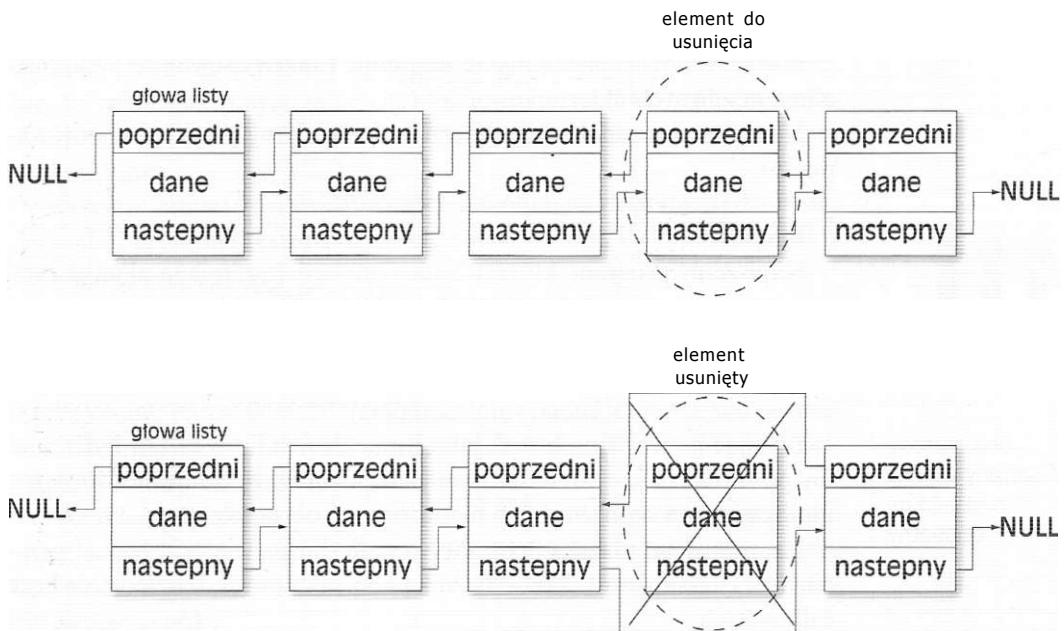
2. Usuwamy ostatni element z listy:

```
temp->poprzedni->nastepny = NULL;
delete temp;
temp = NULL;
```

3. Usuwany jest element wewnętrzny:

```
temp->poprzedni->nastepny = temp->nastepny;
temp->nastepny->poprzedni = temp->poprzedni;
usun = temp;
temp = temp->nastepny;
delete usun;
```

Usuwanie elementu wewnętrznego przedstawiono na rycinie 9.8.



Ryc. 9.8. Usuwanie wyróżnionego elementu z listy dwukierunkowej

Teraz zapoznasz się ze sposobem usuwania z pamięci całej listy dwukierunkowej. Usuwając listę jednokierunkową, zaczynaliśmy od głowy listy. W listach dwukierunkowych masz większą swobodę działania, możesz usuwać elementy listy, zaczynając od jej końca. Tylko raz przejdziemy li-

usuwanie listy dwukierunkowej

## 9. Zmienne i struktury dynamiczne

stę w celu odnalezienia ostatniego elementu (można zdefiniować wskaźnik ogon, aby oszczędzić sobie tego kroku). Potem, korzystając jedynie ze wskaźników do elementów poprzednich, będziemy usuwać elementy listy od końca, aż do usunięcia głowy. Oto fragment kodu, który realizuje to zadanie:

```
temp = glowa;
while (temp->nastepny!=NULL)
    temp = temp->nastepny;           // jesteśmy na końcu listy,
    while (temp->poprzedni!=NULL)   // dopóki nie wróćmy
        // na początek listy
    {
        cout << "usuwam z listy ucznia o imieniu: " << temp->imie << endl;
        temp = temp->poprzedni;       // temp przedstawiamy na element
                                       // znajdujący się przed elementem
                                       // ostatnim
        delete temp->nastepny;       // usuwamy następny (czyli
                                       // aktualnie ostatni)
        temp->nastepny = NULL;        // przedostatni jest teraz ostatni
    }
    // wyszliśmy z pętli, więc temp wskazuje na głowę listy
cout << "usuwam głowę listy ";
delete temp;                      // usuwamy głowę listy
```

Pozostawiamy ci jako zadanie napisanie funkcji usuwającej wskazany element z listy dwukierunkowej.

Na koniec przeanalizujmy jeszcze jeden ciekawy przykład zastosowania list.



Opis metody zamiany ułamka zwykłego na dziesiętny

### Przykład

Napiszemy program, którego zadaniem będzie zamiana ułamka zwykłego na ułamek w postaci dziesiętnej. Ułamek zwykły powinny tworzyć dwie liczby całkowite, przy czym licznik ma być większy lub równy 0, natomiast mianownik ma być większy od 0.

Podając na wejściu ułamek jako iloraz dwóch liczb całkowitych, mamy pewność, że jest to liczba wymierna, a zatem rozwinięcie dziesiętne ułamka będzie skończone lub nieskończone okresowe.

Zacznijmy od ułamka  $3/16$ . Pierwszą liczbę rozwinięcia (przed przecinkiem) uzyskujemy, dzieląc licznik przez mianownik w dziedzinie liczb całkowitych:

$$3 : 16 = 0, \text{ reszta } 3$$

Kolejne cyfry rozwinięcia (po przecinku) uzyskujemy, dzieląc przez mianownik uzyskaną resztę z dzielenia pomnożoną przez 10:

$$30 : 16 = 1, \text{ reszta } 14$$

$$140 : 16 = 8, \text{ reszta } 12$$

$$120 : 16 = 7, \text{ reszta } 8$$

$$80 : 16 = 5, \text{ reszta } 0$$

Widzimy, że reszta z jednego z kolejnych dzielen jest równa 0, co oznacza, że ułamek ma skończone rozwinięcie dziesiętne. Pozostaje tylko wypisanie wyniku dzielenia, w naszym przykładzie jest to 0,1875.

Prześledźmy teraz, co się dzieje w przypadku ułamka 4/7:

$$4 : 7 = 0, \text{ reszta } 4$$

$$40 : 7 = 5, \text{ reszta } 5$$

$$50 : 7 = 7, \text{ reszta } 1$$

$$10 : 7 = 1, \text{ reszta } 3$$

$$30 : 7 = 4, \text{ reszta } 2$$

$$20 : 7 = 2, \text{ reszta } 6$$

$$60 : 7 = 8, \text{ reszta } 4$$

Postępujemy podobnie jak poprzednio, ale reszta równa 0 nie pojawiła się, natomiast wyniki zaczęły się powtarzać. Oznacza to, że mamy do czynienia z ułamkiem, który ma rozwinięcie nieskończone okresowe. W takim wypadku kończymy obliczenia, kiedy reszta z dzielenia osiągnie wartość, która już wcześniej wystąpiła. Gdybyśmy kontynuowali dzielenie, to cyklicznie powtarzałby się blok zaznaczony kolorową czcionką. A zatem ciąg cyfr 571428 należy ująć w okrągłe nawiasy, jest to bowiem okres ułamka.

Musimy zatem każdorazowo po wykonanym kolejnym dzieleniu sprawdzić uzyskaną resztę z dzielenia ze wszystkimi, które otrzymaliśmy od rozpoczęcia operacji zamiany.

W programie wykonującym zadanie zamiany ułamka na postać dziesiętną utworzymy listę (wystarczy lista jednokierunkowa), której elementy będą przechowywały wyniki kolejnych dzielen oraz otrzymywane z nich reszty. Po każdym dzieleniu będziemy badać, czy otrzymana reszta z dzielenia bądź jego wynik jest zerem. Spełnienie jednego z tych warunków kończy działanie algorytmu.

Wynikiem działania programu powinien być ciąg cyfr rozwinięcia (przepisanych z utworzonej listy). W odpowiednich miejscach należy oczywiście wyświetlić jeszcze przecinek i nawiasy (w wypadku rozwinięcia okresowego).

Spróbuj samodzielnie napisać program zamieniający ułamek na postać dziesiętną; w razie niepowodzenia przeanalizuj kod programu, który umieściliśmy na płycie CD dołączonej do podręcznika [9.7]



### Pytania kontrolne

1. Co to jest zmienna dynamiczna?
2. Jaki operator służy do alokacji pamięci dla zmiennej dynamicznej?
3. Jaki związek istnieje pomiędzy typem deklarowanej zmiennej dynamicznej a ilością alokowanej pamięci?
4. Kiedy i w jaki sposób zwalnia się pamięć zajmowaną przez zmienną dynamiczną?
5. W jaki sposób można sprawdzić, czy istnieje wystarczająca ilość pamięci na utworzenie zmiennej dynamicznej?
6. W jaki sposób można porównać wartości dwóch zmiennych dynamicznych?
7. W jaki sposób można podać rozmiar tablicy dynamicznej?
8. Czy liczbę elementów tablicy dynamicznej można zmienić w czasie działania programu? Uzasadnij odpowiedź.
9. Czym różni się tablica dynamiczna od listy? Podaj przykłady wykorzystania tablicy oraz listy.
10. Jak usuwa się listę z pamięci?
11. W jaki sposób odwołujemy się do pola struktury, na którą wskazuje wskaźnik?

### Ćwiczenia

1. Napisz program, który utworzy n-elementową tablicę dynamiczną, gdzie n jest wartością podaną przez użytkownika, wypełni ją liczbami rzeczywistymi, również podanymi przez użytkownika, a następnie skopiuje do nowo utworzonej tablicy dynamicznej tylko te liczby, które są większe od 0.
2. Napisz program, który utworzy dwie kwadratowe tablice dynamiczne 0 rozmiarze podanym przez użytkownika, wypełni je losowymi liczbami z przedziału (1, 9), a następnie wyświetli informację o liczbie elementów, które stoją na tych samych pozycjach w obu tablicach 1 mają te same wartości.
3. Zadeklaruj listę uczniów w klasie. Uwzględnij pola: nazwisko, wiek, wzrost. Napisz funkcje umożliwiające:
  - a) utworzenie listy,
  - b) dodanie danych ucznia do listy,
  - c) usunięcie z listy wskazanego ucznia,
  - d) posortowanie uczniów według wzrostu,
  - e) usunięcie listy.

# 10. Zapis do plików i odczyt z plików

W tym rozdziale omówimy sposób zapisu danych do pliku oraz ich odczytywania z pliku. Przedstawimy na przykładach, kiedy najlepiej odczytywać z pliku pojedyncze znaki, a kiedy wygodniej odczytać całe wyrazy lub wiersze. Poznasz metody obsługi błędów zapisu i odczytu. Zaprezentujemy również kilka prostych funkcji ułatwiających analizę zawartości plików.

Wiesz już, jak w języku C++ wyprowadzić informację na ekran lub wprowadzić z klawiatury wartości zmiennych do programu. Instrukcja: `cout << "łubie informatykę";` była interpretowana jako strumień danych płynący na standardowe wyjście, czyli na ekran monitora. Instrukcja: `cin >> z;` oznacza zaś strumień danych płynący ze standardowego wejścia do programu (w tym wypadku do zmiennej `z`) .

Bardzo często używamy plików jako źródła danych i miejsca przeznaczenia wyników. Aby móc stosować pliki w programie, musimy się nauczyć definiować strumienie płynące z pliku lub do pliku. Obsługę plików rozpoczęmy od napisania odpowiedniej dyrektywy preprocesora: `#include <fstream>`. Dyrektywa ta dołącza do programu bibliotekę, która zawiera funkcje obsługi plików.

Biblioteka do obsługi plików

## 10.1. zapis do pliku

Kiedy mamy już dołączoną bibliotekę `fstream`, możemy bez przeszkodek zdefiniować obiekt, na przykład `zapis`, który pozwoli nam zapisać dane do pliku `wyjscie.txt`:

```
ofstream zapis("wyjscie.txt");
```

W istocie powyższa instrukcja jest deklaracją identyfikatora strumienia wyjściowego o nazwie `zapis`, będącego obiektem klasy `ostream`. Obiekty i klasy opisaliśmy w następnym rozdziale, w tej chwili obiekt możesz kojarzyć ze zmienną, a klasę z typem tej zmiennej. Deklaracja identyfikatora strumienia jest tu jednocześnie połączona z informacją, że dane będą skierowane do pliku o nazwie `wyjscie.txt`.

Plik wyjściowy może mieć dowolne rozszerzenie lub nie mieć go wcale. My daliśmy rozszerzenie `.txt`, aby ułatwić otwieranie pliku wyjściowego w edytorze tekstu.

W celu stworzenia nowego pliku o nazwie `wyjscie.txt` i zapisu do niego dwóch liczb całkowitych, na przykład 3 i -20, oddzielonych spacją, wystarczy napisać:

Tworzenie nowego pliku

## 10. Zapis do plików i odczyt z plików

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

int main()
{
    ofstream zapis("wyjscie.txt");
    zapis << 3 << " " << -20;
    zapis.close();
    return 0;
}
```

10.1

### Zamykanie pliku

Plik **wyjscie.txt** pojawi się w folderze, w którym został uruchomiony twój program. Funkcje umożliwiające obsługę plików uruchamia się przez podanie nazwy strumienia i - po kropce - nazwy funkcji, którą chcemy użyć. Tak więc instrukcja **zapis.close()** umieszczona w ostatniej linijce kodu oznacza, że funkcja **close** zapisze na dysku i zamknie plik powiązany z obiektem (zmienną) **zapis**. Obiekt **zapis** klasy **ofstream** po użyciu funkcji **close** nadal funkcjonuje w naszym programie i może zostać powtórnie użyty do utworzenia dowolnego innego pliku.

Aby powtórnie użyć już istniejącego obiektu **zapis** do obsługi strumienia skierowanego, na przykład do pliku **obliczenia.txt**, należy wykorzystać funkcję **open**, która otwiera plik już istniejący. Realizujemy to w następujący sposób:

```
zapis.open("obliczenia.txt");
```

Musisz jednak pamiętać, że można tak zrobić dopiero po uprzednim zamknięciu funkcją **close** pliku, z którym wcześniej był powiązany obiekt **zapis**. Omówimy to zagadnienie na przykładzie:

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

int main()
{
    int a=4;
    float b=8.5;
    ofstream wyj("ala.txt");           // utworzenie obiektu wyj (1)
    wyj << a << endl << b;           // zapis do pliku (2)
    wyj.close();                      // zamknięcie pliku
    wyj.open("ala.txt");              // ponowne otwarcie pliku
    wyj.close();                      // ponowne zamknięcie pliku
    wyj.open("zosia.txt");
    wyj << endl << endl << 99;
    wyj.close();
    return 0;
}
```

10.2

W linii opatrzonej komentarzem (1) tworzymy obiekt o nazwie **wyj** klasy **ofstream** powiązany z plikiem **ala.txt**. Obiekt ten będzie nam slu-

żył do utworzenia strumieni zapisujących dane w pliku. W linii z komentarzem (2) wpisujemy do pierwszego wiersza pliku **ala.txt** wartość zmiennej **a**, do drugiego zaś wartość zmiennej **b** (zauważ, że zmienne te nie muszą być tego samego typu). Gdyby program zakończył się po linii zamykającej plik **ala.txt**, to efektem jego działania byłby zapis do pliku liczb 4 oraz 8,5. W kolejnej linii plik **ala.txt** został jednak ponownie otwarty funkcją **open** i powiązany z obiektem **wyj** służącym do zapisu danych. W kolejnej linii zamykamy ponownie plik **ala.txt** (jest on w tej chwili pusty), a w następnej linii obiekt **wyj** otwiera nieistniejący dotąd plik **zosia.txt**. Ostatecznie wynikiem działania tego programu są dwa pliki: **ala.txt** - pusty i **zosia.txt** - w którym w trzecim wierszu jest wpisana liczba 99.

**Nieuchronną konsekwencją ponownego otwarcia pliku za pomocą obiektu klasy `ofstream` jest bezpowrotnie skasowanie całej dotychczasowej zawartości pliku.**

Do tej pory nazwy obsługiwanych przez nas plików wprowadzaliśmy już w momencie pisania kodu programu. Nazwy te można również pobierać w czasie pracy programu. Przeanalizujmy przykład:

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

int main()
{
    char nazwa[50];           // tablica znaków z ograniczeniem do 50 znaków
    cout << "Podaj nazwę pliku, jaki chcesz utworzyć ";
    cin >> nazwa;            // pobranie nazwy dla tworzonego pliku
    ofstream wyniki(nazwa);   // utworzenie pliku (1)
    wyniki << 's';          // do pliku wpiszemy literę s
    wyniki.close();
    return 0;
}
```

Podawanie nazw plików z klawiatury

**10.3**

W linii oznaczonej komentarzem (1) opisano główną zmianę: zamiast nazwy podanej wprost w cudzysłowie w nawiasie umieszczono nazwę zmiennej, będącej ciągiem znaków podanych przez użytkownika programu (jest to po prostu tablica znaków). W miejsce nazwy pliku można wpisać również całą ścieżkę bezwzględną pliku lub względną - jeśli pliki mają zostać zapisane w innym folderze niż folder programu, na przykład:

```
ofstream wyniki("c:/wyniki/rezultat.txt"); // ścieżka bezwzględna
ofstream wyniki("../rezultat.txt");        // ścieżka względna
```

Foldery wpisane w ścieżce muszą już istnieć na dysku.

## 10. Zapis do plików i odczyt z plików

Obsługa błędu  
otwarcia pliku  
do zapisu

Podczas próby zapisu do pliku lub odczytu z pliku mogą się pojawić błędy. Jeśli tworzymy plik, to w programie powinniśmy zamieścić instrukcję sprawdzającą, czy powiodło się otworzenie pliku do zapisu. Może się bowiem zdarzyć, że na dysku docelowym pozostało zbyt mało miejsca lub jest on chroniony przed zapisem. Chcemy, aby w takim wypadku wyświetlił się komunikat: „Pliku nie można otworzyć”.

Napiszmy program, który tylko otworzy istniejący plik lub utworzy nowy i go zamknie. Jeśli zaś operacja ta okaże się niemożliwa do wykonania, wypisze komunikat „Pliku nie można otworzyć” i zakończy swoją pracę, wysyłając do systemu operacyjnego informację o błędzie wykonania.

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

int main()
{
    ofstream wyniki("out.txt");
    if (!wyniki)           // początek obsługi błędów
    {
        cout << "Pliku nie można otworzyć";
        getchar();          // umożliwia ci przeczytanie komunikatu błędu
        return 1;
    }                      // koniec obsługi błędów
    wyniki.close();
    return 0;
}
```

10.4

Obsługa błędów jest w powyższym programie realizowana w instrukcji if rozpoczynającej się w linii opatrzonej odpowiednim komentarzem. Jeśli próba powiązania obiektu wyniki z plikiem out.txt się nie powiedzie, to strumień wyniki osiągnie wartość „fałsz”, a fakt ten jest wykorzystany do utworzenia warunku. Instrukcja return 1 kończy działanie programu i przekazuje do systemu operacyjnego wartość 1 (niektóre systemy operacyjne wykorzystują informację, czy uruchomiony program zakończył się powodzeniem czy nie - domyślnie program zakończony poprawnie przyjmuje wartość 0).

### 10.2. Odczyt danych z pliku

Aby odczytać dane z pliku, musimy - podobnie jak przy zapisie - zdefiniować strumień kojarzony z plikiem, z którego chcemy czytać. Do definiowania strumieni wejściowych służy klasa ifstream, która jest zdefiniowana (podobnie jak klasa ofstream) w bibliotece fstream.

Załóżmy, że mamy na dysku plik o nazwie prostokat.txt, w którego pierwszym wierszu zapisana jest liczba 8, a w drugim wierszu - liczba 10.

### Przykład

Napiszemy program, który odczyta dwie liczby zapisane w pliku i obliczy pole prostokąta o długościach boków równych tym liczbom. Napiszemy też obsługę błędów, gdyby nie powiodła się próba odczytu z pliku.



Obstuga błędów  
otwarcia pliku

Oto kod programu realizujący powyższe zadanie:

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

int main()
{
    int a, b;
    ifstream wejscie("prostokat.txt"); // powiązanie strumienia z plikiem (1)
    if (!wejscie) // badanie poprawności otwarcia (2)
    {
        cout << "Nie mozna otworzyc pliku";
        getchar(); // umożliwia przeczytanie komunikatu błędu (3)
        return 1;
    }
    wejscie >> a >> b; // odczyt z pliku (4)
    wejscie.close();
    cout << "Pole prostokata wynosi: " << a*b;
    getchar();
    return 0;
}
```

**10.5**

W linii z komentarzem (1) zadeklarowaliśmy obiekt klasy ifstream, nadaliśmy mu nazwę wejście i powiązaliśmy go z plikiem o nazwie **prostokat.txt**. Kolejna linia zawiera warunek badający poprawność otwarcia pliku, a następująca po warunku instrukcja jest wykonywana w razie wystąpienia błędu odczytu. Do najczęstszych błędów odczytu należy próba otwarcia nieistniejącego pliku. Jeśli plik został otwarty do odczytu, to w kolejnej linii jest przedstawiony sposób odczytu danych z pliku wejściowego (zauważ pełną analogię do sposobu zapisu do pliku). W linii z komentarzem (4) odczytane z pliku wartości zostają przypisane zmiennej **a** i **b**.

Podobnie jak przy zapisie do pliku raz zadeklarowany identyfikator może służyć do odczytania treści różnych plików, wystarczy wykorzystać funkcję open i jako jej argumentu użyć nazwy nowego pliku, na przykład: **wejście.open("innny.txt");**



### Przykład

Obliczymy sumę liczb typu float zapisanych w pliku tekstowym. Plik, z którego czytamy, ma dość nieuporządkowaną strukturę, ponieważ liczby są od siebie oddzielone różną ilością spacji i znaków końca wiersza. Dane znajdują się w pliku **liczby.txt**. Wydruk tego pliku wygląda następująco:

```
2.8 3 11  
0 -6.4  
  
5.2  
-2 10 8
```

Oto realizacja przykładu:

```
#include <iostream>  
#include <cstdio>  
#include <fstream>  
using namespace std;  
  
int main()  
{  
    float a, suma = 0;  
    char nazwa [100];  
    cout << "Podaj nazwe pliku" << endl;  
    cin >> nazwa;  
    ifstream we(nazwa);  
    if (!we)  
    {  
        cout << "Nie mozna otworzyc pliku";  
        cin.ignore();  
        getchar();  
        return 1;  
    }  
    while (!we.eof()) // dopoki nieprawda, ze natrafiliśmy na koniec pliku  
    {  
        we >> a; // odczyt z pliku  
        if (we) // jeśli strumień wejściowy nie jest pusty  
            suma = suma+a;  
    }  
    we.close();  
    cout << "Suma liczb w pliku " << nazwa << " wynosi: " << suma;  
    cin.ignore();  
    getchar();  
    return 0;  
}
```

10.6

Wynik działania tego programu przy poprawnej nazwie pliku:

```
Podaj nazwe pliku  
liczby.txt  
Suma liczb w pliku liczby.txt wynosi: 31.6
```

Jeśli podamy nazwę pliku, który nie istnieje, otrzymamy:

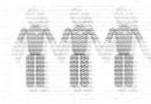
```
Podaj nazwe pliku  
liczby  
Kie można otworzyć pliku
```

Do wykrycia końca pliku wykorzystaliśmy w tym programie funkcję `eof()`, która przyjmuje wartość logiczną „fałsz”, jeśli jeszcze nie ma końca pliku lub wartość logiczną „prawda”, jeśli osiągnięty został koniec pliku. Zastosowaliśmy tę funkcję w pętli `while` w linii opatrzonej pierwszym komentarzem. Operator wejścia `>>` pomija wszystkie białe znaki i odczytuje w pętli tylko kolejne liczby z pliku, przypisując je zmiennej `a`.

Sprawdzenie końca pliku

### Przykład

Napiszemy program, który przepisze zawartość pliku `liczby.txt` z poprzedniego przykładu do pliku `liczby2.txt` w taki sposób, żeby każda kolejna liczba była w osobnym wierszu pliku wynikowego.



W tym programie będziemy obsługiwać przypadki, gdy nie da się otworzyć pliku, z którego chcemy czytać, lub pliku, do którego chcemy zapisywać.

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

int main()
{
    float a;
    ifstream we("liczby.txt");
    if (!we)
    {
        cout << "Nie mozna otworzyc pliku do odczytu";
        getchar();
        return 1;
    }
    ofstream wy("liczby2.txt");
    if (!wy)
    {
        cout << "Nie mozna otworzyc pliku do zapisu";
        getchar();
        return 1;
    }
    while (!we.eof())
    {
        we >> a;
        if (we) // jeśli strumień wejściowy nie jest pusty
            wy << a << endl;
    }
    we.close();
    wy.close();
}
```

10.7

Jeśli program nie napotka problemów przy odczycie lub zapisie, wynikiem jego działania będzie utworzenie pliku **liczby2.txt** o zawartości:

```
2.8  
3  
11  
0  
-6.4  
5.2  
-2  
10  
8
```

Znasz już prosty sposób odczytywania z pliku i zapisywania do pliku liczb. Pokażemy ci teraz, jakich konstrukcji najlepiej użyć do analizy pojedynczych znaków w pliku tekstowym.



### Przykład

Napiszemy program, który zlicza liczbę znaków występujących w pliku tekst.txt. W tym programie wykorzystamy funkcję `get()`, która wczytuje (pobiera) pojedynczy znak. Zawartość pliku tekst.txt (po każdym wierszu jest znak końca wiersza, również po ostatnim):

```
To be  
or not  
to be
```

Bazując na poprzednich przykładach, przeanalizuj kod programu:

```
#include <iostream>  
#include <cstdio>  
#include <fstream>  
using namespace std;  
  
int main()  
{  
    int n = 0;      // licznik znaków w pliku  
    char c;  
    ifstream we("tekst.txt");  
    if (!we)  
    {  
        cout << "Nie mozna otworzyc pliku do odczytu";  
        getchar();  
        return 1;  
    }  
    while (!we.eof())  
    {  
        we.get(c);    // pojedynczy znak pobrany z pliku tekst.txt  
                    // zostaje przypisany zmiennej c  
        if (we)        // jeśli strumień wejściowy nie jest pusty  
            n = n+1;   // zwiększamy wartość licznika znaków  
    }  
    we.close();  
    cout << n;  
    getchar();  
    return 0;  
}
```

10.8

[www.operon.pl](http://www.operon.pl)

W wyniku działania tego programu zostanie wyświetlona liczba 19. Samych widocznych znaków i spacji jest 16, ale program doliczył również 3 znaki końca wiersza.

Jeśli chcielibyśmy zliczyć tylko znaki alfanumeryczne (to znaczy wszystkie z pominięciem białych znaków, wszelkich znaków interpunkcyjnych i znaków sterujących, takich jak koniec wiersza czy koniec pliku), to w pętli while z poprzedniego przykładu wystarczy dopisać warunek:

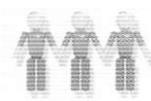
```
if (isalnum(c))
    n = n+1;
```

Funkcja `isalnum()` przyjmuje wartość „prawda”, jeśli jej argument podany w nawiasie jest literą lub cyfrą. Funkcjami podobnymi w swoim działaniu do `isalnum` są `isdigit()` - przyjmuje wartość „prawda”, gdy jej argument jest cyfrą, oraz funkcja `isalpha()` - przyjmuje wartość „prawda”, gdy jej argument jest literą.

Funkcje badające rodzaj znaku

### Przykład

Litery (bez białych znaków) z pliku z poprzedniego przykładu wpiszemy do nowego pliku o nazwie **Literki.txt**. Wyświetlimy również liczbę liter w pliku.



Oto realizacja tego przykładu:

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

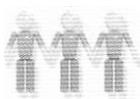
int main()
{
    int n = 0; // licznik liter
    char c;
    ifstream we("tekst.txt");
    ofstream wy("literki.txt");
    while (!we.eof())
    {
        we.get(c);
        if (we && isalpha(c)) // jeśli strumień wejściowy nie jest pusty
        { // i jeśli pobrany znak jest literą,
            n = n+1; // to zwiększamy wartość licznika liter
            wy.put(c); // i literę zapisujemy do pliku literki.txt
        }
    }
    we.close();
    wy.close();
    cout << n;
    getchar();
    return 0;
}
```

10.9

## 10. Zapis do plików i odczyt z plików

Wynikiem działania tego programu jest wypisanie na ekranie monitora liczby 13 oraz utworzenie pliku **iiterki.txt** o treści „Tobeornottobe”. Dla skrócenia kodu usunęliśmy obsługę błędów zapisu i odczytu. W programie wykorzystaliśmy funkcję **put( )**, która wpisuje znak „c” do pliku powiązanego ze strumieniem **wy**.

Skoro umiemy już zliczyć znaki zapisane w pliku, możemy się zająć zliczaniem słów, które w nim wystąpiły. Taki program może się przydać na przykład w sytuacji, gdy otrzymasz w pliku tekstowym nazwiska uczestników wycieczki, a nie wiesz, ile osób jest wpisanych na listę ( zakładamy, że wszyscy mają tylko jedno nazwisko, to znaczy nie ma osób o nazwisku typu: Wyrwa Krzyżański). Zliczając słowa w pliku, otrzymasz odpowiedź na swoje pytanie.



### Przykład

Napiszemy program, który zliczy liczbę słów w pliku. Przez „słowo” rozumiemy ciąg liter zakończony dowolnym białym znakiem (wszystkie słowa wpisane do pliku muszą kończyć się co najmniej jednym białym znakiem).

Odczyt i zapis  
ciągów  
znakowych

Na początku zadeklarujemy dwie tablice znaków: jedną do wpisania nazwy pliku, z którego będziemy czytać, drugą do przechowania słowa odczytanego z pliku. Do tej pory odczytywaliśmy z pliku lub zapisywaliśmy do niego liczby całkowite bądź rzeczywiste oraz znaki. Można również odczytywać lub zapisywać całe ciągi znaków. W wierszu opatrzonym komentarzem (2) do tablicy **słowo** będziemy wstawiać kolejne słowa pobrane z odczytywanego pliku.

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

int main()
{
    int n = 0;
    char plik[100], słowo[100]; // deklaracja tablic znakowych (1)
    cout << "Podaj nazwe pliku" << endl;
    cin >> plik;
    ifstream we(plik);
    if (!we)
    {
        cout << "Nie mozna otworzyc pliku do czytania";
        cin.ignore();
        getchar();
        return 1;
    }
    while (!we.eof())
    {
        we >> słowo;           // wprowadzenie słowa z pliku wejściowego (2)
```

```

        if (we)
            n = n+1;
    }
    we.close();
    cout << n;
    cin.ignore();
    getchar();
    return 0;
}

```

10.10

Bardzo użyteczna i często używana jest możliwość odczytywania z pliku całych wierszy. Do odczytania całego wiersza z pliku posłużymy się funkcją `getline`.

#### Przykład

Napiszemy fragment programu, w którym są odczytywane kolejne wiersze z pliku i obliczana jest ich liczba.



Poniżej prezentujemy program realizujący przykład:

```

char wiersz[200];
ifstream we("tekst.txt");
while (iwe.eof())
{
    we.getline(wiersz,sizeof(wiersz));
    n = n+1;
}
cout << n;

```

Zmienna `wiersz` jest tablicą znaków, w której będziemy zapisywać kolejne wiersze z pliku. W funkcji:

```
we.getline(wiersz,sizeof(wiersz))
```

pierwszym argumentem jest nazwa tablicy znaków, a drugim argumentem - liczba całkowita, która oznacza maksymalną liczbę znaków, jaka ma zostać wpisana do tablicy. W naszym przypadku zastosowaliśmy funkcję `sizeof()`, której wartością jest rozmiar tablicy - można w tym miejscu wpisać wprost jakąś liczbę. Funkcja `getline` ma jeszcze trzeci parametr, który jeśli nie jest wpisany, przyjmuje wartość domniemaną `\n` (tak właśnie jest w naszym programie). Ostatni parametr jest ogranicznikiem, który określa, kiedy przerwać wczytywanie. Kiedy znak ten zostanie napotkany w strumieniu wejściowym, to wczytywanie zostanie przerwane, nawet jeśli nie osiągnęliśmy jeszcze ilości określonej w poprzednim parametrze. Użycie funkcji `getline` może wyglądać na przykład tak:

```
getline(wiersz,10,'k')
```

W tym wypadku wczytywanie zostanie przerwane po pobraniu dziesięciu znaków lub wcześniej, jeśli została napotkana litera „k”.



### Pytania kontrolne

1. Jaką bibliotekę należy dodać do programu, aby mieć możliwość zapisu do pliku danych i odczytu danych z pliku?
2. Czym różni się plik od tablicy?
3. Jaka rolę pełni zazwyczaj plik w programie?
4. W jaki sposób można utworzyć na dysku nowy plik?
5. Jak zapisujemy i jak odczytujemy liczby w pliku?
6. Dlaczego przy tworzeniu strumieni plikowych wejściowych i wyjściowych należy sprawdzać poprawność ich utworzenia?
7. Jak skonstruować pętle, która służy do odczytania całego pliku?

### Ćwiczenia

1. Napisz program, który wpisze do pliku tekst pobrany z klawiatury i umieści każde zdanie w osobnej linii. Przez zdanie rozumiemy ciąg znaków zakończony kropką.  
Wskazówka: Nowe zdanie (oprócz pierwszego) rozpoznajemy po tym, że po kropce i spacjach występuje duża litera.
2. Napisz program, który odczyta tekst z pliku, wypisze go na ekranie oraz poda informację, ile w tym pliku było zdań. Przez zdanie rozumiemy ciąg znaków zakończony kropką.  
Wskazówka: Nowe zdanie (oprócz pierwszego) rozpoznajemy po tym, że po kropce i spacjach występuje duża litera.
3. Napisz program, który odczyta plik tekstowy i na ekran monitora wprowadzi informację o ilości liczb występujących w pliku. Zakładamy, że w odczytywanym pliku nie ma liczb zmiennoprzecinkowych, to znaczy liczb z kropką dziesiętną.  
Wskazówka: Jeśli cyfry w ciągu występują po sobie, to tworzą liczbę. Jeśli zaś ciągi cyfr przedzielone są spacjami, to liczby są dwie.
4. Napisz program, który w pliku tekstowym zamienia wszystkie pierwsze litery wyrazów na wielkie litery.
5. Napisz program, który będzie służył do usuwania komentarzy z tekstu programu napisanego w C++ (pamiętaj, że są dwa rodzaje komentarzy: jednoliniowy, zaczynający się od //, i blokowy, zawierający się pomiędzy znakami /\* i \*/).

## 11. Wstęp do programowania obiektowego

W tym rozdziale przedstawimy podstawowe wiadomości i umiejętności związane z programowaniem obiektowym. Dowiesz się, co to jest obiekt i klasa, a także do czego służy hermetyzacja (zwana też kapsuikowaniem) oraz konstruktor i destruktor. Powiemy ci, jak sprawić, aby funkcja była zaprzyjaźniona z klasą, i co z tej przyjaźni wynika.

### 11.1. Obiekt a klasa

Programy możemy tworzyć również z użyciem klas i obiektów, jest to wówczas programowanie obiektowe. Obiektem w tym programowaniu nazywamy podstawowy element programu łączący opis stanu pewnej części rzeczywistości (jaką opisuje program) z jej zachowaniem, czyli funkcjami, inaczej zwanyimi metodami obiektu. Na przykład obiektem może być samochód, który jest opisany przez takie parametry jak: marka, moc silnika, aktualna prędkość, waga itp. Samochód ma również przyporządkowane funkcje (akcje), które może wykonać, na przykład: jazda z przyspieszeniem, tankowanie albo sposób zachowania się na zakręcie. Oczywiście, funkcje wykonywane przez samochód mogą, ale nie muszą wpływać na wartości parametrów opisujących jego stan, na przykład przyspieszenie ma wpływ na aktualną prędkość, lecz nie na markę.

Podejście obiektowe (łączące w jedną całość dane i funkcje wykonywane na tych danych) znacznie się różni od omawianego do tej pory programowania proceduralnego, w którym dane oraz funkcje na nich operujące są traktowane rozłącznie i programista musi pamiętać o odpowiednim użyciu funkcji do odpowiednich danych. Obiektowe podejście do programowania jest zgodne ze sposobem percepcji otoczenia przez mózg ludzki - zawsze kojarzymy cechy obiektów z ich zachowaniem. Dodatkowo silna jest w człowieku potrzeba klasyfikowania obiektów podobnych - co również ma swoje odbicie w programowaniu obiektowym. Wszystkie obiekty są mianowicie elementami klas, które reprezentują typ obiektów o takim samym zachowaniu. Programowanie obiektowe polega na opisie wzajemnych interakcji obiektów, w czym jest analogiczne do tego, co obserwujemy w realnym świecie. Programowanie obiektowe jest najbardziej naturalnym sposobem opisu rzeczywistości w programach komputerowych, dzięki czemu programy takie są łatwe w pisaniu i późniejszej analizie.

Klasa i obiekt -  
opis pojęć

**Definicja klasy i obiektu**

Budowanie programów obiektowych umożliwiają wspomniane wcześniej narzędzia. Oto ich formalna definicja:

**D e f i n i c j a**

^ ^

Klasa jest to złożony typ będący opisem (definicją) pól i metod obiektów do niej należących.

**Definicja**

Obiekt jest konkretyzacją danej klasy i wypełnieniem wzorca (jakim jest klasa) określonymi danymi.

W uproszczeniu można powiedzieć, że klasa to struktura wzbogacana o definicje funkcji. Podobnie zatem jak struktura, klasa ma zdefiniowane pola, w których są przechowywane dane obiektów. Liczba i typy pól są definiowane przez programistę. W klasie zdefiniowane są również metody, czyli funkcje, które może wykonywać obiekt danej klasy. Ogólnie pola i metody klasy nazywamy składnikami klasy.

**Definicja klasy nie wiąże się z utworzeniem obiektu, jest to jedynie zdefiniowanie nowego typu obiektów (zmiennych). Klasa jest typem obiektu, a nie samym obiektem.**

**Definicja klasy Ułamek**

Sposób definiowania klasy w programie pokażemy na przykładzie klasy ułamków zwykłych:

```
class Ułamek
{
public:
    int licznik;
    int mianownik;
};
```

**Klasyfikacja składników klasy**

Definicja klasy rozpoczyna się od słowa kluczowego **class**. Za klamrą zamykającą definicję klasy wymagany jest średnik, tak jak przy definiowaniu struktury. We wnętrzu klasy pojawiło się nowe słowo: **public**. Oznacza ono, że składniki po nim wymienione (po dwukropku) są tak zwanyimi **składnikami publicznymi klasy**, czyli mamy do nich dostęp zarówno z wnętrza klasy, jak i spoza niej. Tak napisana klasa funkcjonalnie niczym się nie różni od podobnej struktury. Różnice te pojawią się, kiedy użyjemy składników o innym zakresie dostępu niż **public**. W klasie mogą wystąpić składniki opatrzone etykietą **private** - są one **prywatnymi składnikami klasy**, a dostęp do nich jest możliwy wyłącznie z wnętrza klasy. W wypadku danych oznacza to, że tylko funkcje będące składnikami tej klasy mogą do nich zapisywać lub

z nich czytać, w wypadku zaś funkcji - tylko inne funkcje składowe tej klasy mogą je wywołać. Składniki klasy mogą być również typu `protected`, są one wówczas dostępne zarówno z wnętrza klasy, jak i dla klas jej pochodnych (opisanych na końcu tego rozdziału). Domyślnie - jeśli nie ma podanego innego zakresu dostępu, dane i funkcje zadeklarowane wewnątrz klasy są prywatne. Etykiety `public`, `private`, `protected` zwane są inaczej specyfikatorami dostępu.

Deklaracja obiektu `ull` klasy **Ułamek** zdefiniowanej wcześniej może wyglądać następująco:

`Ułamek ull;`

Dostęp do poszczególnych składników obiektów uzyskujemy w taki sam sposób, jak do składowych struktur, czyli za pomocą operatora wyłuskania (kropki):

`ull.licznik;`  
`ull.mianownik;`

Deklaracja obiektu zdefiniowanej klasy

## 11.2. Hermetyzacja - rola metod publicznych

Wymienione specyfikatory dostępu ściśle łączą się z jedną z najważniejszych cech programowania obiektowego, jaką jest hermetyzacja.

Definicja

**Hermetyzacja** pozwala na ukrycie pewnych danych i funkcji obiektu przed bezpośrednim dostępem z zewnątrz.

Dzięki temu mechanizmowi otrzymujemy obiekt, do którego ukrytych (hermetycznych) pól i metod mamy dostęp tylko przez udostępnione metody publiczne. Jest to bardzo cenna cecha programowania obiektowego, ponieważ znacznie ogranicza możliwość popełnienia błędu. Wyobraź sobie samochód, w którym masz dostęp nie tylko do pedału gazu, sprzęgła i hamulca, ale możesz także dowolnie ustawić napięcie prądnicy, fazę zapłonu, skład mieszanki paliwowej, ciśnienie oleju - łatwo sobie wyobrazić, że taki samochód byłby bardzo narażony na usterki.

Napiszemy zatem naszą klasę **Ułamek** w taki sposób, aby dostęp do danych: **licznik** i **mianownik** był możliwy tylko przez metody, które „wiedzą”, jak postępować z licznikiem i mianownikiem. W klasie **Ułamek** niedopuszczalna powinna być operacja przypisania mianownikowi wartości 0.

Wzbogaćmy zdefiniowaną klasę o metody związane z obsługą ułamków. Będą to publiczne składowe klasy. Na początek zdefiniujemy funkcję **zapisz()**, wypełniającą obiekty klasy **Ułamek**, i funkcję **wypisz()**, wyświetlającą te obiekty:

Deklaracja i definicja metody klasy

```

class Ulamek
{
    int licznik;
    int mianownik;
public:
    void zapisz(int l, int m);           // deklaracja metody klasy
    void wypisz()
    {
        cout << licznik << "/" << mianownik;
    }
};                                     // koniec definicji klasy

void Ulamek::zapisz(int l, int m)    // definicja metody klasy
{
    licznik = l;
    if (m!=0)
        mianownik = m;
    else
    {
        cout << "Mianownik nie moze miec wartosci 0 ";
        getchar();
        exit(1);
    }
}

```

**11.1**Operator  
zasięgu

Teraz klasa **Ulamek** ma cztery składowe: dwie prywatne - **licznik** i **mianownik**, oraz dwie publiczne - zapisującą dane do obiektu **Ulamek** i wypisującą wartość ułamka. Definicje metod klasy umieściliśmy na dwa dopuszczalne sposoby. Metoda **wypisz** jako bardzo krótka została umieszczona wewnętrz definiacji klasy. Natomiast metoda **zapisz** została tylko zadeklarowana wewnątrz klasy, a jej definicję umieściliśmy poza definicją klasy, dlatego jej nazwę uzupełniliśmy nazwą klasy, do której metoda należy. W tym celu posłużyliśmy się tak zwanym **operatorem zasięgu**, oznaczanym **::** (podwójnym dwukropkiem).

Zauważ, że zmieniliśmy specyfikator dostępu przy polach **licznik** i **mianownik**, ponieważ usunęliśmy specyfikator **public**. Jest to równoznaczne z uzyskaniem przez obydwa pola domyślnego statusu **private**. Do pól **licznik** i **mianownik** zadeklarowanej w ten sposób klasy nie można się już odwołać w programie (spoza klasy) przez operator wyłuskania. Do pól prywatnych mają dostęp jedynie metody klasy. Tu właśnie mamy do czynienia z mechanizmem hermetyzacji. Wartości licznika i mianownika obiektu możemy zmienić tylko przy użyciu funkcji **zapisz**. Dzięki takiemu rozwiązaniu zabezpieczamy się przed przypisaniem mianownikowi niedozwolonej wartości 0. Jeśli spróbujemy to zrobić, metoda **zapisz** zareaguje wypisaniem odpowiedniego komunikatu

i zakończeniem programu (funkcja `exit(1)` kończy cały program, przekazując do systemu operacyjnego wartość 1 sygnalizującą błąd).

Jeśli zdefiniowaliśmy klasę, możemy już utworzyć obiekty tej klasy.

Spójrz na prosty program, który wykorzystuje klasę zdefiniowaną powyżej:

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

class Ulamek
{
    int licznik;
    int mianownik;
public:
    void zapisz(int l, int m);
    void wypisz()
    {cout << licznik << "/" << mianownik;}
};

void Ulamek::zapisz(int l, int m) // definicja metody klasy
{
    licznik = l;
    if (m!=0)
        mianownik = m;
    else
    {
        cout << "Mianownik nie moze miec wartosci 0 ";
        getchar();
        exit(1);
    }
}
int main()
{
    Ulamek ull, ul2;
    ull.zapisz(4,5);
    ul2.zapisz(1,7);
    cout << "Pierwszy ulamek: ";
    ull.wypisz();
    cout << endl << "Drugi ulamek: ";
    ul2.wypisz();
    getchar();
    return 0;
}
```

**11.2**

Jako efekt działania tego krótkiego programu na ekranie monitora zobaczymy:

Pierwszy ulamek: 4/5  
 Drugi ulamek: 1/7

### 11.3. Rola konstruktora i destruktora

Przyjrzyj się sposobowi deklarowania obiektów klasy **Ulamek** z poprzedniego przykładu i nadawaniu im wartości:

```
Ulamek uli;
uli.zapisz(4,5);
```

Najpierw deklarujemy obiekt, a następnie uruchamiamy odpowiednią funkcję (dostęp do metody, podobnie jak dostęp do pól danych klasy, uzyskujemy przez operator kropki). Jeśli tworzymy klasy, chcielibyśmy móc się nimi posługiwać tak samo wygodnie jak każdym innym typem. Jak sobie zapewne przypominasz, deklarując zmienną typu `int`, możemy nadać jej wartość w jednej instrukcji:

Konstruktor

W podobny sposób możemy również zadeklarować i nadać początkowe wartości obiektem klasy **Ulamek**. W tym celu użyjemy specjalnej metody zwanej **konstruktorem**.

**Konstruktorem** nazywamy specjalną metodę automatycznie uruchamianą w trakcie definiowania (tworzenia) obiektu pozwalającą na nadanie początkowych wartości danym obiektu.

Omówmy sposób deklaracji konstruktora w C++. Metoda ta nie przyjmuje żadnej wartości (nawet `void!`), a jej nazwa jest taka jak nazwa zawierającej ją klasy. Przeróbmy zatem metodę `zapisz` na konstruktor:

```
class Ulamek
{
    int licznik;
    int mianownik;
public:
    Ulamek (int l, int m);           // deklaracja konstruktora
    void wypisz()
    {
        cout << licznik << "/" << mianownik;
    }
};                                // koniec definicji klasy

Ulamek::Ulamek(int l, int m)      // definicja konstruktora
{
    licznik = l;
    if (m!=0)
        mianownik = m;
    else
    {
        cout << "Mianownik nie moze miec wartosci 0 ";
        getchar();
        exit(1);
    }
}
```

11.3

Obiekt zdefiniowanej w ten sposób klasy możemy zadeklarować następująco:

```
Ulamek liczba(3,5);
```

Przy tak zbudowanym konstruktorze straciliśmy jednak możliwość deklarowania obiektu bez nadawania mu początkowych wartości (co mogliśmy zrobić w poprzednim programie). Istnieje rozwiązanie tego problemu. Klasa może mieć kilka różnych konstruktorów różniących się liczbą argumentów - jest to tak zwane **przeciążenie konstruktora**. Aby móc deklarować obiekty typu **Ulamek** bez konieczności podawania ich wartości początkowych, możemy napisać bezparametrywy konstruktor, który sam nadaje początkowe wartości obiektom, na przykład:

```
class Ulamek
{
    int licznik;
    int mianownik;
public:
    Ulamek() // definicja pierwszego konstruktora (bezparametrowego)
    {
        licznik = 1; // licznikowi ułamka przypisujemy wartość 1
        mianownik = 1; // mianownikowi ułamka przypisujemy wartość 1
    }
    Ulamek (int l, int m); // deklaracja drugiego konstruktora (z parametrami)
    void wypisz()
    {
        cout << licznik << "/" << mianownik; // koniec definicji klasy
    };
};

Ulamek::Ulamek(int l, int m) // definicja konstruktora z parametrami
{
    licznik = ul;
    if (m!=0)
        mianownik = m;
    else
    {
        cout << "Mianownik nie może mieć wartości 0 ";
        getchar();
        exit(1);
    }
}
```

11.4

Oprócz konstruktora w klasie powinien być również zadeklarowany **destruktor**.

### Definicja

**Destruktorem** nazywamy specjalną metodę bezparametryową, która jest wywoływana zawsze w momencie usuwania obiektu.

Destruktor ma nazwę taką jak nazwa klasy, ale poprzedzoną węzykiem na przykład destruktor klasy **Rakieta** będzie miał nazwę **-Rakieta**. Dla naszej klasy **Ulamek** destruktor może wyglądać następująco:

## 11. Wstęp do programowania obiektowego

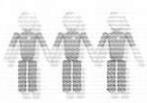
```
~Ulamek()  
{cout << "Usuwam obiekt ";}
```

Destruktor ten poza usunięciem obiektu wypisuje jeszcze odpowiedni komunikat - dzięki niemu wiemy, kiedy jest uruchamiany.

**Jeśli w swojej klasie nie zdefiniujesz destruktora, zostanie on wygenerowany przez kompilator.**

Destruktor jest metodą, która uruchamia się automatycznie zwykłe tuż przed zakończeniem programu. Sami możemy spowodować wywołanie destruktora dla obiektów znajdujących się w pamięci dynamicznej.

Jak się zapewne domyślasz, obiekty zdefiniowanej klasy mogą być argumentami funkcji, a funkcja może przyjmować wartość obiektu zdefiniowanej klasy.



Funkcja  
zaprzyjaźniona

### Przykład

Napiszemy program, który mnoży dwa ułamki. Wynik zostaje wyświetlony na ekranie monitora.

W tym celu zdefiniujemy funkcję, która pobiera dwa obiekty klasy **Ulamek**, mnoży je i przyjmuje wartość obiektu tej samej klasy będący iloczynem pobranych ułamków. Funkcja mnożąca ułamki nie będzie metodą klasy - działa jedynie na obiektach zdefiniowanej klasy. Pojawia się tu jednak problem, ponieważ do składowych prywatnych klasy mają dostęp tylko metody klasy. W naszym wypadku wystąpi zatem problem z dostępem funkcji zewnętrznej do licznika i mianownika, gdyż są to składowe prywatne obiektu typu **Ulamek**. Aby funkcja niebędąca elementem klasy miała dostęp do jej składników prywatnych, musimy w klasie zapisać **deklarację przyjaźni** (zaprzyjaźnienia się) z tą funkcją.

### Definicja

**Funkcja zaprzyjaźniona** to taka funkcja zewnętrzna (niebędąca składnikiem klasy), dla której w definicji klasy umieszczona jest deklaracja przyjaźni za pomocą słowa kluczowego **friend**.

Klasa może być też zaprzyjaźniona z inną klasą, ale to zagadnienie wykracza poza rama tego podręcznika.

W wypadku naszej klasy **Ulamek** po wpisaniu:

```
friend Ulamek pomnoz (Ulamek, Ulamek);
```

**funkcja pomnoz przyjmująca wartości typu Ulamek będzie zaprzyjaźniona z klasą Ulamek.**

W programie dodaliśmy jeszcze jedną pozytyczną metodę prywatną **skracaj**, służącą do skracania ułamka. Metoda ta nie będzie widocz-

na na zewnątrz klasy (jako prywatna), ale będzie z niej korzystała metoda wypisz. Przed każdym wypisaniem ułamek zostaje skrócony.

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

class Ulamek
{
private:
    int licznik;
    int mianownik;
    void skracaj();

public:
    Ulamek() // definicja pierwszego konstruktora
    {
        licznik = 1;
        mianownik = 1;
    }
    Ulamek(int l, int m); // deklaracja drugiego konstruktora
    void wypisz()
    {
        skracaj();
        cout << licznik << "/" << mianownik;
    }
    friend Ulamek pomnoz(Ulamek, Ulamek); // deklaracja przyjazni
}; // koniec definicji klasy

void Ulamek::skracaj()
{
    int a = abs(licznik);
    int b = abs(mianownik);
    while (a!=b)
        if (a>b)
            a = a-b;
        else
            b = b-a;
    licznik = licznik/a;
    mianownik = mianownik/a;
}

Ulamek::Ulamek(int l, int m)
{
    licznik = l;
    if (m!=0)
        mianownik = m;
    else
    {
        cout << "Mianownik nie moze miec wartosci zero ";
        getchar();
        exit(1);
    }
}

Ulamek pomnoz(Ulamek ul, Ulamek u2) // funkcja mnozaca ulamki
{
```

## 11. Wstęp do programowania obiektowego

```
Ulamek wynik;
wynik.licznik = ul.licznik*u2.licznik;
wynik.mianownik = ul.mianownik*u2.mianownik;
return wynik; // wynikiem funkcji jest obiekt klasy Ulamek
}
int main()
{
    int l, m;
    cout << "Podaj licznik i mianownik pierwszego ułamka" << endl;
    cin >> l >> m;
    Ulamek a(l,m);
    cout << "Podaj licznik i mianownik drugiego ułamka" << endl;
    cin >> l >> m;
    Ulamek b(l,m);
    cout << "Pierwszy ułamek: ";
    a.wypisz();
    cout << endl << "Drugi ułamek: ";
    b.wypisz();
    cout << endl << "Iloczyn ";
    a.wypisz();
    cout << " i ";
    b.wypisz();
    cout << " wynosi ";
    pomnoz(a,b).wypisz();
    cin.ignore();
    getchar();
    return 0;
}
```

11.5

Oto efekt działania programu:

```
Podaj licznik i mianownik pierwszego ułamka
4 6
Podaj licznik i mianownik drugiego ułamka
Pierwszy ułamek: 2/3
Drugi ułamek: 1/2
Iloczyn 2/3 i 1/2 wynosi 1/3
```

W metodzie skracającej użyliśmy algorytmu Euklidesa. Klasę **Ułamek** można w zależności od potrzeb wzbogacić o dodatkowe metody, takie jak: dodawanie, odejmowanie, dzielenie i inne.

Dziedziczenie, czyli definiowanie klas pochodnych

Umiejętność tworzenia klas w programowaniu zaawansowanym jest bardzo przydatna, gdyż na bazie zdefiniowanej klasy, na przykład: **Pojazd**, łatwo zbudować klasy: **Samochód**, **Motor**, **Traktor**, opierając się na tak zwanym dziedziczeniu. Mówimy wówczas, że klasa **Motor** jest klasą pochodną klasy **Pojazd**. Programista może również zbudować klasę opisującą obiekt większy, ale częściowo składający się z innych, mniejszych, już wcześniej zdefiniowanych obiektów, albo przejmujący część ich cech. Przykładowo: jeśli chcesz uszyć powłóczkę na poduszkę, to nie zainteresuje cię produkcja guzików, nici i materiału, ale skorzystasz z gotowych produktów. Temat dziedziczenia jest bardzo rozległy, zachęcamy jednak do zgłębiania go za pomocą dodatkowej lektury.



### Pytania kontrolne

1. Czym się różni klasa od struktury?
2. W jaki sposób deklarujemy obiekty klasy?
3. Jakie są sposoby deklarowania metod w klasie?
4. Jak odwołujemy się do składowych klasy?
5. Jakie są rodzaje składników klasy ze względu na dostęp do nich?
6. Co to jest konstruktor i do czego służy?
7. Co to jest funkcja zaprzyjaźniona i jakie ma właściwości?

### Ćwiczenia

1. Napisz klasę wektor, która będzie miała metodę:
  - a) pobierającą współrzędne wektora,
  - b) wyświetlającą współrzędne wektora w postaci na przykład [x;y],
  - c) obliczającą długość wektoraoraz funkcję zaprzyjaźnioną:
  - a) sumującą dwa wektory,
  - b) obliczającą iloczyn wektora przez liczbę (skalar).

# Indeks

## A

- algorytm 7
  - etapy konstruowania 8
  - Euklidesa 95
  - liniowy 10
  - numeryczny 8
  - , poprawność 23
  - rozgałęziony 16
    - typowe złożoności 27-28
  - złożoność obliczeniowa 24, 26
    - czasowa 26
    - kwadratowa 27
    - liniowa 26, 27
    - pamięciowa 26
    - pesymistyczna 27
  - i - wielomianowa 27
- alokacja pamięci 220
- ASCII 155-159
  - , kod 155
- asemblacja 39

## B

- bajt 36
- biblioteka 40
- bisekcja 106
- bit 35
  - najbardziej znaczący 35
  - najmniej znaczący 35
- błąd syntaktyczny 49
- break 57, 64-65
- Browna ruchy 113

## C

- case 57
- Cezara szyfr 156-159
- cin 48
- continue 64
- cout 43
- cstdio 43
- cstdlib 112

## D

- dane
  - przejściowe 12

- wejściowe 8
- wyjściowe 8
- debuger 40
- default 57
- deklaracja
  - przyjazni 262
  - zmiennej 47
- dekrementacja 60
- destruktor 261
- deasemblacja 39
- do ... while 62
- drzewo
  - postępowania 18
  - wywołań rekurencyjnych 173
- dyrektywa preprocesora 43
- dziedziczenie 264
- „dzieli i zwyciężaj” metoda 180

## E

- Eratostenesa sito 140

## F

- Fibonacciego ciąg 172
- for 59
- friend 262
- fstream 243

## G

- Gaussa metoda eliminacji 159-160
- getchar() 43
- getline() 153
- GIMPS 92

## H

- Hanoi wieża 175-178

- hermetyzacja 257

- Herona wzór 29

## I

- if... else 50
- inicjacja zmiennej 48
- inkrementacja 21, 60
- instrukcja warunkowa 16
  - pusta 16
  - wewnętrzna 16

- , zagnieżdżenie 54
- instrukcja ziożona 51
- iomanip 125
- ignore() 49
- iostream 43
- iteracja 21
- #include 43
- K**
- klasa 256
  - funkcji 27
  - , metody 256
  - pochodna 264
  - , pole 256
  - , składniki 256
  - , - prywatne 256
  - , - publiczne 256
  - , złożoności algorytmu 27
- klucz 157
- kod maszynowy 38
- kod źródłowy 39
- komentarz 45
- kompilacja 40
- kompilator 40
- konserwacja programu 38
- konstruktor 260
  - , przeciążenie 261
- L**
- liczba
  - pierwsza 91, 140
  - złożona 91
  - % 109
- linker 40
- linkowanie 40
- lista 227
  - dwukierunkowa 236–241
  - jednokierunkowa 234–236
  - kroków 9
  - usuwanie 234–241
- M**
- macierz 122
- main() 43
- współczynników 162
- manipulator 44, 125
- mnemonik 39
- Monte Carlo metody 109
- N**
- new 221
- Newtona-Raphsona metoda 99
- Neumann John von 38, 183
- NULL 112
- NWD 95
- NWW98
- O**
- obiekt 255, 256
- , metoda 255
- obsługa błędu 246
- odczyt danych z pliku 246–253
- odsiewanie liczb 140
- O „O duże”, notacja 27
- operacja dominująca 26
- operandy 13
- operator
  - arytmetyczny 14
  - logiczny 15
  - przypisania 14
  - relacji 14.
  - zasięgu 258
- ośmiu hetmanów, problem 192
- P**
- pętla 21
  - zagnieżdzona 123
- połowieienie przedziałów 106
- preprocesor 43
- programowanie obiektowe 255
- przeszukiwanie
  - binarne 181
  - liniowe 128
- pseudojęzyk 9
- R**
- rand() 112
- rekurencja 167
  - , funkcja 167, 168
  - , metoda 167
- return 43
- Riemanna całka oznaczona 102
- rzutowanie 157

**S**

schemat  
- blokowy 10  
- , - , podstawowe elementy 10  
-Homera 174-175  
silnia 167  
składowe struktury 195  
skoczek szachowy, problem 188-191  
sortowanie 142  
- bąbelkowe 142  
- przez scalanie 182-183  
- przez wstawianie 145  
- przez wybór 148-149  
- szybkie 187  
specyfikacja problemu  
algorytmicznego 11  
specyfikatory dostępu 257  
sprawdzanie końca pliku 249  
stała tekstowa (string) 43  
stała 12  
standardowe  
- wejście 47  
- wyjście 43  
stos 169  
strandQ 112  
struktura 195  
-, pole 195  
switch 56  
system  
binarny 35  
- liczbowy, zamiana 134  
- pozycyjny 35

**T**

tablica 117  
- , argument funkcji 120  
- dwuwymiarowa 117  
-, deklaracja 123  
- dynamiczna 219-220  
-jednowymiarowa 117,118

- tekstowa 152  
-, - deklaracja 152  
- kwadratowa 122  
-, deklaracja 123  
- , przekroczenie zakresu 119  
- , przeszukiwanie liniowe 128  
- , wypełnianie 118, 123  
time() 112  
treść funkcji 70  
tryb krokowy 49  
typ zmiennej 12, 41

**U**

układ równań  
- , postać macierzowa 162  
- , postać trójkątna 161

**W**

wartownik 130  
wektor  
- niewiadomych 162  
- wyrazów wolnych 162  
while 61  
wnętrze funkcji 70  
wskaźnik 207  
-, deklaracja 207  
-, typ 207  
wyrażenia logiczne 14

**V**

void 75

**Z**

zmienna 12, 47  
- globalna 72  
- lokalna 72  
- pomocnicza 12  
-, typ 41-42  
znacznik końca tekstu 155  
znak nowej linii 44

## **Literatura pomocnicza**

- T. H. Cormen, Ch. E. Leiserson, R. L. Rivest, *Wprowadzenie do algorytmów*, Warszawa 1998.
- A. Drozdek, *C++ Algorytmy i struktury danych*, Gliwice 2004.
- B. Eckel, C. Allison, *Thinking in C++*, Gliwice 2003.
- J. Grębosz, *Symfonia C++*, t. I-III, Kraków 1999.
- K. Jakubczyk, *Turbo Pascal i Borland C++ przykłady*, Gliwice 2002.
- R. Kent, *C++, Algorytmy i struktury danych*, Gliwice 2004.
- J. Kniat, *Programowanie w języku C++*, Poznań 1998.
- K. Koleśnik, *Wstęp do programowania z przykładami w Turbo Pascalu*, Gliwice 1999.
- S. B. Lippman, *Istota języka C++ . Zwięzły opis*, Warszawa 2004.
- S. B. Lippman, J. Lajoie, *Podstawy języka C++*, Warszawa 2002.
- R. Neapolitan, K. Naimipour, *Podstawy algorytmów z przykładami w C++*, Gliwice 2004.
- H. Schildt, *C++ programowanie*, Warszawa 2002.
- R. Sedgewick, *Algorytmy w C++*, Warszawa 2003.
- B. Stroustrup, *Język C++*, Warszawa 2004.
- A. Struzińska-Walczak, K. Walczak, *Nauka programowania dla początkujących C++*, Warszawa 2002.
- M. M. Sysło, *Algorytmy*, Warszawa 2002.
- N. Wirth, *Algorytmy + struktury danych = programy*, Warszawa 2001.
- P. Wróblewski, *Algorytmy, struktury danych i techniki programowania*, Gliwice 2003.

Notatki

270'

## Informacje o płycie CD dołączonej do podręcznika

Na płycie znajdują się pliki z kodami źródłowymi oraz kompilatory języka C++ (Borland Builder Personal 6.0 wymaga darmowej rejestracji).

Po umieszczeniu płyty w napędzie nastąpi automatyczne uruchomienie strony głównej. Jeśli strona się nie pojawi, należy otworzyć plik START.exe, znajdujący się w głównym katalogu płyty CD.

Płyta jest uzupełnieniem podręcznika *Informatyka, część I. Podręcznik dla liceum ogólnokształcącego* autorstwa Piotra Brody i Danuty Smołuchy.

Wykorzystywanie utworów w innych celach, w szczególności ich powielanie lub wypożyczanie, bez zezwolenia zabronione. Wszystkie prawa autorskie do utworów zastrzeżone.

Nazwy firm i produktów zamieszczonych na płycie są znakami towarowymi należącymi do odpowiednich właścicieli.

Płyta CD została poddana wszechstronnym testom antywirusowym.

Wydawnictwo nie ponosi odpowiedzialności za ewentualne szkody wynikłe z korzystania z oprogramowania zawartego na płycie CD.

Dziękujemy twórcom oprogramowania za wyrażenie zgody na umieszczenie ich materiałów na płycie CD dołączonej do niniejszej publikacji.

Copyright by Wydawnictwo Pedagogiczne OPERON Sp. z o.o.

Prezentacja powstała przy użyciu programu Flash Universal Launcher autorstwa Herberta Janssena ([herbert.janssen@web.de](mailto:herbert.janssen@web.de)).