Sławomir SAMOLEJ, Wojciech RZĄSA, Dariusz RZOŃCA,
Jan SADOLEWSKI, Bartosz JĘDRZEJEC

Wprowadzenie do informatyki I – architektura komputerów, algorytmika, paradygmaty i języki programowania

skrypt dla studentów
kierunków nieinformatycznych
na uczelniach technicznych

OFICYNA
WYDAWNICZA
POLITECHNIKI RZESZOWSKIEJ

Wydano za zgodą Rektora

O p i n i o d a w c a
dr hab. Marcin SZPYRKA, prof. AGH

R e d a k t o r
Marta JAGIEŁOWICZ

P r o j e k t o k ł a d k i
Joanna MIKUŁA

Matryce zostały przygotowane przez Autorów w programie LaTeX

Skrypt przeznaczony dla studentów kierunków nieinformatycznych
na uczelniach technicznych

architektura komputerów
podstawowe algorytmy i struktury danych
paradygmaty programowania
programowanie strukturalne i obiektowe

© Copyright by Oficyna Wydawnicza Politechniki Rzeszowskiej, Rzeszów 2014, 2016

p-ISBN 978-83-7199-944-5
e-ISBN 978-83-7934-754-4

Oficyna Wydawnicza Politechniki Rzeszowskiej
al. Powstańców Warszawy 12, 35-959 Rzeszów

Ark. wyd. 11,82. Ark. druk. 11,00.
Oddano do druku w grudniu 2016 r. Wydrukowano w grudniu 2016 r. Wydanie II.
Drukarnia Oficyny Wydawniczej, al. Powstańców Warszawy 12, 35-959 Rzeszów
Zam. nr 129/16

Spis treści

1. Wstęp **7**

2. Historia informatyki **9**

3. Wprowadzenie do architektury komputerów **13**
- 3.1. Wprowadzenie 13
- 3.2. Układy liczbowe, podstawowe kodowania 13
- 3.3. Podstawy arytmetyki i logiki komputerów 18
- 3.4. Jądro mikroprocesora AVR 26
- 3.5. Zadania 28

4. Elementy algorytmów i struktur danych **31**
- 4.1. Wprowadzenie 31
- 4.2. Podstawy algorytmiki 31
- 4.3. Elementarne struktury danych 40
- 4.4. Złożoność obliczeniowa 44
- 4.5. Zadania 47

5. Język Ruby i paradygmaty programowania **49**
- 5.1. Wprowadzenie 49
- 5.2. Instalacja i korzystanie z języka 50
- 5.3. Język Ruby – podstawy 51
  - 5.3.1. O języku 51
  - 5.3.2. Hello world! 52
  - 5.3.3. Ruby i język polski 53
  - 5.3.4. Zmienne 54
  - 5.3.5. Instrukcje warunkowe 56
  - 5.3.6. Iteracje, czyli powtarzanie 61
  - 5.3.7. Struktury danych 63
- 5.4. Programowanie strukturalne 65
- 5.5. Programowanie proceduralne 67

5.5.1. Motywacja 67
5.5.2. Podprogramy 67
5.5.3. Zmienne lokalne i globalne 70
5.6. Programowanie obiektowe 71
5.6.1. Rozwój koncepcji 71
5.6.2. Obiekty i klasy 71
5.6.3. Metody 72
5.6.4. Dostęp do metod i pól obiektu 73
5.6.5. Konstruowanie obiektu 74
5.6.6. Obsługa błędów 76
5.6.7. Dziedziczenie 77
5.7. Programowanie imperatywne i deklaratywne 80
5.8. Programowanie funkcjonalne 83
5.8.1. Wiadomości wstępne 83
5.8.2. Rekurencja 84
5.8.3. Skutki uboczne funkcji 86
5.8.4. Funkcje jako parametry funkcji 86
5.8.5. Definiowanie nowych funkcji 87
5.8.6. Polimorfizm 87
5.8.7. Stosowanie języków funkcyjnych 88
5.9. Programowanie zdarzeniowe 89
5.10. Zadania 89

6. Język C 93
6.1. Wprowadzenie 93
6.2. Podstawowe elementy języka C 94
6.2.1. Pierwszy program 94
6.2.2. Zestaw znaków dopuszczalnych 95
6.2.3. Nazwy 95
6.2.4. Słowa kluczowe 96
6.2.5. Typy danych 96
6.2.6. Literały i stałe symboliczne 96
6.2.7. Zmienne 98
6.2.8. Wyrażenia 99
6.3. Operacje wejścia-wyjścia 100
6.3.1. Operacje standardowego wyjścia 100
6.3.2. Operacje standardowego wejścia 102
6.4. Operatory 105
6.4.1. Grupy operatorów 105
6.4.2. Operatory arytmetyczne 106
6.4.3. Operatory porównania 107

6.4.4. Operatory logiczne 108
6.4.5. Operatory wskazywania 109
6.5. Instrukcje 109
6.5.1. Podział instrukcji 109
6.5.2. Instrukcja złożona 110
6.5.3. Instrukcja przypisania 110
6.5.4. Instrukcja warunkowa 111
6.5.5. Instrukcje pętli 113
6.5.6. Instrukcja zaniechania 119
6.5.7. Instrukcja wyboru 121
6.5.8. Instrukcja wywołania funkcji 122
6.5.9. Instrukcja powrotu 123
6.5.10. Instrukcja pusta 123
6.6. Funkcje 124
6.6.1. Zastosowania funkcji 124
6.6.2. Definicja funkcji 124
6.6.3. Wywołanie funkcji 125
6.6.4. Deklaracja funkcji – prototyp 126
6.7. Tablice 127
6.7.1. Definicja tablicy 127
6.7.2. Deklaracja tablic 129
6.7.3. Inicjowanie tablic 130
6.8. Wskaźniki 131
6.8.1. Deklaracja wskaźnika 131
6.8.2. Operacje na wskaźnikach 132
6.9. Pliki 132
6.9.1. Funkcje otwarcia i zamknięcia pliku 132
6.9.2. Funkcje odczytu i zapisu do plików 135
6.10. Zadania 137

7. Język C++ 139
7.1. Wprowadzenie 139
7.2. Struktury i klasy 140
7.3. Biblioteki wejścia wyjścia 142
7.4. Konstruktory, destruktory i przeciążanie metod 144
7.5. Enkapsulacja pól 149
7.6. Dziedziczenie i metody wirtualne 151
7.7. Dynamiczne tworzenie obiektów 156
7.8. Zadania 160

8. Podsumowanie 163

Spis rysunków **165**

Spis tablic **167**

Bibliografia **169**

Skorowidz **171**

Rozdział 1.

Wstęp

Koniec poprzedniego i początek obecnego wieku upływa pod znakiem „rewolucji informatycznej”. Komputery i technologie informatyczne zmieniły sposób wymiany i przetwarzania informacji oraz przemysłowego wytwarzania urządzeń. Inaczej wygląda współczesna bankowość, telekomunikacja, przemysł rozrywkowy oraz wiele innych gałęzi gospodarki i życia społecznego. Narodził się „przemysł informatyczny”, a więc źródło przychodu przedsiębiorstw oraz technik zunifikowanego wytwarzania oprogramowania i komputerów. W miarę rozwoju informatyki wyodrębniły się główne obszary wiedzy i techniki stymulujące jej rozwój: przemysł mikroprocesorowy, elektroniczny i elektromechaniczny, algorytmika, języki i paradygmaty programowania, systemy operacyjne, sieci komputerowe, bazy danych oraz sztuczna inteligencja.

Od wielu absolwentów wyższych uczelni oczekuje się nie tylko umiejętności obsługi wybranych programów lub systemów komputerowych, ale także pogłębionej praktycznej wiedzy na temat współczesnych obszarów informatyki. Niezbędną wiedzę można uzyskać z wielu publikacji wydanych w Polsce i za granicą, wymaga to jednak przeprowadzenia systematycznych studiów. Na rynku polskim jest dostępna bardzo bogata literatura omawiająca szczegóły różnych dziedzin informatyki, jednak niewiele jest książek stanowiących przewodnik wprowadzający w sposób zintegrowany najważniejsze z nich. Niniejszy skrypt jest próbą wypełnienia takiej luki wydawniczej. Zadaniem, jakie postawili sobie autorzy, jest przeprowadzenie systematycznego kursu wprowadzającego do najistotniejszych działów współczesnej informatyki na takim poziome, aby absolwent technicznych studiów nieinformatycznych był w stanie swobodnie nawiązywać relacje z przedstawicielami świata informatyki. W kursie przyjęto zasadę wyjaśniania poszczególnych zagadnień na przykładach, pomijając celowo omówienie bardziej ścisłych zasad matematycznych leżących u podstawy działania technologii komputerowych.

Materiał został podzielony na dwie części. W niniejszym skrypcie omówiono historię informatyki, architekturę komputerów, algorytmikę oraz wybrane języki i paradygmaty programowania. Kolejny skrypt będzie obejmować wiedzę doty-

czącą sieci komputerowych, baz danych, systemów operacyjnych oraz wybranych technik sztucznej inteligencji.

Dobór materiału zawartego w skrypcie jest wynikiem kilkunastoletnich doświadczeń autorów w prowadzeniu wykładów, ćwiczeń audytoryjnych i laboratoryjnych z przedmiotu „informatyka” dla nieinformatycznych kierunków studiów technicznych. Celem skryptu jest dostarczenie Czytelnikom kompendium wiedzy z głównych działów informatyki. Może to być publikacja, od której rozpoczyna się studiowanie zagadnień współczesnej informatyki.

Skrypt podzielono na 8 rozdziałów. Rozdział 2. jest poświęcony ideom, wynalazkom i ludziom, których osiągnięcia stanowiły kroki milowe w rozwoju informatyki. Jak się okazuje, idea usprawniania obliczeń i wykorzystania do tego maszyn i algorytmów sięga starożytności, a od siedemnastego wieku można dostrzec systematyczny ciąg wynalazków i teorii matematycznych prowadzących do powstania współczesnych komputerów. Rozdział 3. omawia matematyczne podstawy działania komputerów oraz typowe elementy systemu mikroprocesorowego. Pokazuje, jak wychodząc od algebry i logiki matematycznej, można sformułować zasady działania podstawowych komponentów współczesnego mikroprocesora. W rozdziale 4. wprowadza się pojęcia algorytmu, struktury danych i złożoności obliczeniowej. Algorytmika wskazuje zasady konstruowania programów komputerowych, a także metody analizy ich poprawności i czasu wykonywania. Rozdział 5. jest poświęcony technikom programowania w jednym z najnowocześniejszych języków programowania o nazwie Ruby. Wprowadza również tak zwane paradygmaty programowania wypracowane wraz z rozwojem technik i języków programowania. Są to w ogólności techniki tworzenia programów, w których w różny sposób spoglądamy na odwzorowanie rzeczywistości w programach komputerowych. W rozdziale 6. wprowadza się elementy programowania w języku C, który od ponad 40 lat służy do przemysłowego tworzenia oprogramowania podstawowych urządzeń mikroprocesorowych, a także do wytwarzania programów osadzanych na wszystkich popularnych systemach operacyjnych. Rozdział 7. zawiera podstawowe techniki programowania w języku C++, który bazuje na języku C i pozwala na tworzenie oprogramowania zorientowanego obiektowo. Rozdział 8. stanowi krótkie podsumowanie, w którym autorzy odsyłają Czytelnika do innych publikacji zawierających poszerzone wiadomości z zagadnień przedstawionych w skrypcie.

Rozdział 2.

Historia informatyki

Dariusz Rzońca

Komputer jest uważany za jeden z symboli dwudziestego wieku. Obecnie trudno byłoby wyobrazić sobie codzienne życie bez korzystania z udogodnień, które zawdzięczamy intensywnemu rozwojowi informatyki w ostatnich dziesięcioleciach. Komputer jest powszechnie wykorzystywanym urządzeniem i zazwyczaj, korzystając z niego, nie zastanawiamy się nad długą historią informatyki i komputerów. Wyjaśnienie genezy „maszyn liczących" będących praprzodkami komputera wymaga krótkiego opisu sięgającego czasów starożytnych.

Od wieków ludzie konstruowali urządzenia ułatwiające dokonywanie obliczeń. Około 440 lat p.n.e. w antycznej Grecji i Rzymie opracowano abakus (abak) będący pierwowzorem liczydła. Miał on formę płyty z wyżłobionymi lub narysowanymi rowkami, w których przesuwano żetony. Powszechne użycie różnego typu liczydeł obserwowano przez kolejne tysiąclecia, aż do czasów nowożytnych.

Niezależnie od rozwoju techniki i maszyn wspomagających obliczenia należy pamiętać, że istotny wkład w podstawy informatyki już w starożytności wnieśli genialni matematycy. Jeden z pierwszych algorytmów, pozwalający na obliczenie największego wspólnego dzielnika, zawdzięczamy greckiemu matematykowi Euklidesowi żyjącemu w IV wieku p.n.e. Algorytm ten stosowany do dziś przedstawiono także w rozdziale 4.

Jednym z kolejnych mechanizmów liczących były opracowane w siedemnastym wieku pałeczki Nepera. Traktat „Rabdolgia" napisany w 1617 roku przez angielskiego matematyka Johna Nepera szczegółowo opisywał sposób sprowadzenia mnożenia do serii dodawań oraz obliczania iloczynów cząstkowych na specjalnych pałeczkach obliczeniowych. Miały one formę prostopadłościanu z oznaczonymi na bocznych ścianach iloczynami poszczególnych cyfr przy mnożeniu przez kolejne cyfry. Mnożenie polegało na wybraniu właściwych pałeczek i sumowaniu odpowiednich cyfr z sąsiadujących pałeczek dla żądanego iloczynu. Rozwój pałeczek Nepera doprowadził do zastąpienia ich walcami (lub też tarczami) umieszczonymi na wspólnej podstawie, wówczas wykonanie mnożenia sprowa-

działo się do właściwego obrotu walców i obliczenia poszczególnych cyfr wyniku przez dodawanie odpowiednich cyfr iloczynów cząstkowych.

Wilhelm Shickard w 1623 roku opracował maszynę liczącą pozwalającą na automatyczne uwzględnianie przeniesień generowanych podczas dodawania i odejmowania. Mnożenia nadal były wykonywane ręcznie za pomocą walców Nepera. Niestety prototyp tego urządzenia wkrótce po jego zbudowaniu spłonął w pożarze [19]. Niezależnie od prac Shickarda we Francji Blaise Pascal około 1645 roku opracował własną maszynę liczącą, zwaną „pascaliną". Łącznie wytworzono kilkadziesiąt pascalin w różnych wersjach, wspomagających obliczenia geodetów, bankierów i poborców podatkowych. Maszyna ta działała nieco inaczej niż urządzenia Shickarda, lecz nadal spełniała rolę jedynie sumatora. Dopiero w 1694 roku Gottfried Wilhelm Leibniz zaprojektował maszynę pozwalającą także na automatyczne wykonywanie mnożeń.

W osiemnastym wieku udoskonalano projekty Schickarda, Pascala i Lebniza, konstruując różnego rodzaju mechaniczne kalkulatory zwane arytmometrami. Intensywny rozwój nauki i techniki, a w szczególności astronomii i marynistyki wymagał opracowania wielu dokładnych tablic matematycznych. Żmudne obliczenia wykonywano ręcznie, co skutkowało wieloma błędami. Dziewiętnastowieczny matematyk i astronom Charles Babbage był twórcą projektów pozwalających na zautomatyzowanie tej żmudnej pracy. Stopień skomplikowania „maszyny różnicowej" oraz „maszyny analitycznej" przewyższał jednak ówczesne możliwości technologiczne. Projekty te, jakkolwiek nie zostały w pełni zrealizowane, wniosły istotny wkład w rozwój współczesnej informatyki [33, 32]. Współpracująca z Babbagem Ada Lovelace jest uznawana za autorkę pierwszego programu. Opisała ona szczegółowo metodę obliczania liczb Bernoulliego przy użyciu nieistniejącej wówczas maszyny analitycznej.

W dziewiętnastym wieku godne odnotowania są także prace George Boole'a. Algebra Boole'a, szerzej omówiona w rozdziale 3., pozwoliła na zdefiniowanie podstawowych mechanizmów działania współczesnych komputerów. Istotny wkład w rozwój informatyki miały także dokonania dwudziestowiecznego angielskiego matematyka Alana Turinga. W 1936 roku opublikował on pracę „On Computable Numbers" („O liczbach obliczalnych") zawierającą teoretyczne rozważania na temat abstrakcyjnego modelu maszyny umożliwiającej wykonywanie algorytmów (maszyna Turinga). Należy pamiętać także o pracach Claude'a Shannona, jednego z twórców teorii informacji.

Druga wojna światowa to czas dalszego intensywnego rozwoju informatyki, w szczególności ukierunkowanej na zastosowania militarne. Ówczesna technika wojskowa wymagała dokładnych tablic balistycznych, jak również maszyn kryptograficznych. Zapotrzebowanie na moc obliczeniową przewyższało możliwości mechanicznych czy też elektromechanicznych kalkulatorów. Podczas wojny po-

wstały m.in. maszyny Konrada Zusego (Niemcy) oraz kalkulatory przekaźnikowe w laboratoriach firm Bell i IBM (Stany Zjednoczone). Prace Johna von Neumanna w projekcie Manhattan (prace nad bombą atomową) przyczyniły się do rozwoju metod numerycznych i opracowania koncepcji architektury komputerów (architektura von Neumanna), stosowanej z pewnymi zmianami także obecnie (rozdział 3.). W projekcie Manhattan brał udział także Stanisław Ulam. Był on nie tylko jednym z pierwszych naukowców wykorzystujących w swoich pracach komputer, ale też autorem metod numerycznych, np. metody Monte Carlo. Podkreślenia wymagają także prace polskich matematyków i kryptologów (np. Mariana Rejewskiego, Jerzego Różyckiego i Henryka Zygalskiego) skutkujące złamaniem szyfru niemieckiej Enigmy i pośrednio przyczyniające się do rozwoju angielskiej techniki komputerowej [10].

Znaczącym osiągnięciem umożliwiającym dalszy rozwój maszyn obliczeniowych było opracowanie lampy elektronowej, zastępującej w kolejnych konstrukcjach przekaźniki elektromagnetyczne. Jednym z pierwszych lampowych kalkulatorów był ENIAC opracowany w 1946 roku w Stanach Zjednoczonych. Jego elementy zajmowały kilka pomieszczeń i łącznie ważyły około 30 ton. ENIAC miał potężną jak na tamte czasy moc obliczeniową, choć znacznie mniejszą niż współczesny kalkulator. Niekiedy ENIAC jest błędnie uważany za pierwszy komputer, należy jednak pamiętać o opracowanych wcześniej w ramach ściśle tajnego projektu wojskowego angielskich maszynach Colossus [31]. Materiały na ich temat odtajniono dopiero w 1975 roku.

Intensywny rozwój elektroniki w kolejnych latach umożliwił zastąpienie lamp elektronowych tranzystorami, następnie tranzystorów układami scalonymi integrującymi tysiące tranzystorów wewnątrz jednego układu. Pozwoliło to na zwiększenie wydajności kolejnych komputerów przy jednoczesnej ich miniaturyzacji i zmniejszeniu poboru energii. W latach sześćdziesiątych ubiegłego wieku wprowadzono pojęcie generacji komputerów, dzieląc je na maszyny oparte na przekaźnikach elektromagnetycznych (generacja zerowa), następnie lampach elektronowych (pierwsza generacja), tranzystorach (druga generacja) oraz układach scalonych małej i średniej skali integracji (trzecia generacja) [35]. Współczesne komputery należą do komputerów czwartej generacji opartych na układach scalonych wielkiej skali integracji, a opracowywane projekty komputerów kwantowych zaliczamy do piątej generacji.

W latach siedemdziesiątych ubiegłego wieku opracowano mikroprocesor, cyfrowy układ wykonujący operacje na podstawie wbudowanej listy instrukcji. Uniwersalność mikroprocesorów pozwala na wykorzystanie ich w różnych zastosowaniach w zależności od programu, zastępując dedykowane układy scalone, co przez zwiększenie popytu wpłynęło na znaczne zmniejszenie kosztów produkcji. Pierwszymi cywilnymi ogólnodostępnymi mikroprocesorami był opra-

cowany w 1971 roku przez firmę Intel czterobitowy układ 4004 o architekturze typowej dla kalkulatorów [7], a kolejnym ośmiobitowy układ 8008 (Intel, 1972). W dalszych latach nastąpił dynamiczny rozwój techniki mikroprocesorowej, powiększeniu uległo słowo maszynowe, przestrzeń adresowa i częstotliwość zegara. Współczesne mikroprocesory mają moc obliczeniową dziesiątki tysięcy razy większą od swoich pierwowzorów. Układ scalony takiego mikroprocesora w swojej strukturze zawiera niekiedy nawet setki milionów tranzystorów.

Mikroprocesory umożliwiły budowę tanich, niewielkich komputerów. W latach osiemdziesiątych zeszłego stulecia opracowano wiele domowych mikrokomputerów opartych na różnych mikroprocesorach. Najczęściej rozpowszechnione były komputery oparte na architekturze PC opracowanej w firmie IBM dzięki braku patentów na tę konstrukcję, co umożliwiło wielu producentom budowę tzw. „klonów”, tańszych niż komputery markowe, korzystając z gotowych rozwiązań bez inwestycji w badania i rozwój [1]. Popularność tej architektury skutkowała dynamicznym rozwojem oprogramowania i dalszym rozpowszechnianiem. Dzisiejsze komputery domowe są także następcami architektury PC.

Rozdział 3.

Wprowadzenie do architektury komputerów

Dariusz Rzońca

3.1. Wprowadzenie

Opis architektury współczesnych komputerów zawarty w niniejszym rozdziale został poprzedzony wyjaśnieniem podstawowych teoretycznych zagadnień. Skrótowo omówiono pozycyjne systemy liczbowe, w szczególności układy binarny i heksadecymalny. Przedstawienie reprezentacji liczb w pamięci komputera wymaga krótkiego opisu kodowań, takich jak kod naturalny binarny, kodowanie znak-moduł, znak-uzupełnienie do 1, znak-uzupełnienie do 2 oraz kodowań zmiennoprzecinkowych. Omówiono także podstawowe funkcje arytmetyczne i logiczne. W celu wyjaśnienia mechanizmów wykonywania obliczeń arytmetycznych w układzie cyfrowym zaprezentowano zasady działania półsumatora, sumatora i jednostki arytmetyczno-logicznej. Po skrótowym opisie architektur von Neumanna i harwardzkiej przedstawiono budowę jądra współczesnych mikroprocesorów ośmiobitowych na przykładzie jądra mikrokontrolera AVR. Zamieszczony w niniejszym rozdziale pobieżny opis poszczególnych elementarnych zagadnień należy traktować jedynie jako krótkie wprowadzenie do tematu, a nie kompletne kompendium.

3.2. Układy liczbowe, podstawowe kodowania

Pierwszym zagadnieniem wymagającym omówienia, jako jeden z podstawowych aspektów związanych z architekturą komputerów, jest sposób reprezentacji liczb w pamięci komputera. Powszechnie stosowany przez nas *dziesiętny system liczbowy* ze względów technicznych niezbyt dobrze nadaje się do bezpośredniego zastosowania w komputerze, jakkolwiek niektóre z pierwszych maszyn liczących, np. ENIAC (wzmiankowany w rozdziale 2.), pracowały właśnie w systemie dzie-

siętnym [19]. W systemie tym używamy dziesięciu symboli, 0–9 do zapisu poszczególnych cyfr. Liczby są zapisywane jako ciągi cyfr oznaczających mnożniki kolejnych potęg dziesiątki (*system pozycyjny*). Przykładowo, liczba tysiąc dwieście trzydzieści cztery zapisywana jest jako 1234, co oznacza, że jest sumą jednego tysiąca, dwóch setek, trzech dziesiątek i czterech jedności.

$1234 = 1 * 10^3 + 2 * 10^2 + 3 * 10^1 + 4 * 10^0$

Analogicznie są zbudowane inne pozycyjne systemy liczbowe. Liczba występująca w nazwie systemu, nazywana *podstawą systemu*, określa zarówno liczbę symboli reprezentujących cyfry, jak również podstawę kolejnych potęg wartościujących poszczególne pozycje w liczbie. W architekturze komputerów szczególne znaczenie ma *system dwójkowy*, zwany także binarnym. W systemie tym potrzebujemy jedynie dwóch symboli do zapisu liczby, co ułatwia techniczną reprezentację w pamięci komputera. Taką najmniejszą porcję informacji przyjmującą dwa stany nazywamy *bitem*. Przykład przeliczania wartości liczby z systemu dwójkowego na dziesiętny przedstawia się następująco (podstawę systemu liczenia oznaczono w nawiasach w prawym dolnym rogu liczby, brak oznaczenia należy rozumieć jako liczbę w systemie dziesiętnym):

$100110_{(2)} = 1*2^5+0*2^4+0*2^3+1*2^2+1*2^1+0*2^0 = 32+4+2 = 38$

Aby przeliczyć liczbę z dziesiętnego systemu liczbowego na dwójkowy, należy postępować następująco:

- dzielimy liczbę przez dwa, reszta z dzielenia stanowi cyfrę jedności dla liczby po przeliczeniu na system binarny,
- otrzymany iloraz powtórnie dzielimy przez dwa, reszta z dzielenia stanowi kolejną cyfrę,
- proces ten powtarzamy, aż w wyniku dzielenia otrzymamy zero.

Przykład takiego przeliczenia dla liczby 38 pokazano na rys. 3.1. Trzydzieści osiem dzieli się przez dwa bez reszty, więc cyfra jedności przeliczonej liczby będzie wynosić zero. Iloraz jest równy dziewiętnaście, liczba ta będzie dzielną w kolejnym dzieleniu. Podczas następnego dzielenia w wyniku otrzymujemy dziewięć i resztę równą jednemu, a więc cyfrę dwójek przeliczonej liczby. Postępujemy analogicznie, aż do otrzymania końcowego wyniku.

Łatwo zauważyć, że zapis liczby w systemie dwójkowym zazwyczaj jest znacznie dłuższy od zapisu w systemie dziesiętnym oraz że przeliczanie pomiędzy systemem dwójkowym i dziesiętnym jest dość żmudne. Dla wygody często korzysta się z systemu szesnastkowego, zwanego także heksadecymalnym. W systemie tym występuje szesnaście symboli oznaczających cyfry, po cyfrze 9 jako dalszych symboli używamy kolejnych liter alfabetu łacińskiego A, B itd., aż do F. Porównanie wartości cyfr w systemie szesnastkowym z liczbami dziesiętnymi i dwójkowymi przedstawiono w tabl. 3.1.

38	
19	0
9	1
4	1
2	0
1	0
0	1

Rys. 3.1. Zamiana liczby
38 na system dwójkowy

Tablica 3.1. Porównanie systemów liczbowych

Liczba dziesiętna	Cyfra szesnastkowa	Czterobitowa liczba dwójkowa
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Przeliczanie pomiędzy systemem dziesiętnym a szesnastkowym jest analogiczne do pokazanych uprzednio przeliczeń pomiędzy systemem dziesiętnym a dwójkowym. Konwertując liczbę heksadecymalną na system dziesiętny, mnożymy kolejne cyfry przez odpowiednie potęgi szesnastki, a wyniki sumujemy. Chcąc zapisać liczbę dziesiętną w systemie szesnastkowym, wykonujemy serię dzieleń z resztą przez szesnaście. Otrzymane reszty stanowią kolejne cyfry heksadecymalne przekształconej liczby, począwszy od cyfry jedności, które zapisujemy według tabl. 3.1. Konwersja pomiędzy systemem binarnym a szesnastkowym jest jeszcze prostsza, gdyż nie wymaga pośredniego przejścia przez system dziesiętny. Aby przekształcić liczbę szesnastkową na dwójkową należy, pamiętając że każdej cyfrze szesnastkowej odpowiadają cztery bity, zapisać kolejne wartości tych bitów według tabl. 3.1. Podobnie, aby przekształcić liczbę dwójkową na szesnastkową, należy podzielić ją na grupy czterobitowe,

zaczynając od najmniej znaczącej cyfry i uzupełniając w razie potrzeby zerami od lewej strony, a następnie zapisać kolejne cyfry szesnastkowe odpowiadające poszczególnym grupom. Przykłady takiej konwersji pokazano na rys. 3.2.

$$100110_{(2)} = 26_{(16)}$$
$$\mathrm{A1F7}_{(16)} = 1010000111110111_{(2)}$$

Rys. 3.2. Konwersja między systemem heksadecymalnym a binarnym

Jak wspomniano, w pamięci komputera liczby dwójkowe są reprezentowane przez ciągi bitów. Operacje są zazwyczaj przeprowadzane na ciągach o długości ośmiu, szesnastu, trzydziestu dwóch bądź sześćdziesięciu czterech bitów. Taki ciąg ośmiu bitów nazwiemy *bajtem*, dwa bajty *słowem* (*słowem szesnastobitowym*), a cztery *podwójnym słowem* (*słowem trzydziestodwubitowym*). *Kilobajt* to 1024 bajty, *megabajt* to 1024 kilobajty, *gigabajt* to 1024 megabajty, dalsze wielokrotności tworzymy analogicznie, używając standardowych przedrostków. Wyjaśnienia wymaga użyty mnożnik (1024, a nie 1000) – wartość ta została wybrana jako potęga dwójki ($1024 = 2^{10}$). Przyjmując, że dany ciąg bitów reprezentuje nieujemną liczbę całkowitą, na jednym bajcie możemy zapisać liczby z zakresu 0–255, na dwóch bajtach 0–65535 itd. Takie kodowanie, gdzie pewien ciąg bitów reprezentuje nieujemną liczbę całkowitą zapisaną w systemie dwójkowym, nazwiemy *naturalnym kodem binarnym* (*NB*). Aby wprowadzić możliwość reprezentacji liczb ujemnych, konieczne jest rozważanie innego kodowania. Istnieją trzy podstawowe kodowania: *znak-moduł* (*ZM*), *znak-uzupełnienie do 1* (*ZU1*) oraz *znak-uzupełnienie do 2* (*ZU2*) [16]. W każdym z nich reprezentacja liczb dodatnich jest identyczna z tą dla kodu naturalnego binarnego przy wyzerowanym najbardziej znaczącym bicie (bicie znaku). Liczby ujemne są oznaczane ustawionym najbardziej znaczącym bitem. W kodowaniu znak-moduł po bicie znaku jest zapisywany moduł (wartość bezwzględna) danej liczby. W kodowaniu ZU1 dla liczb ujemnych po bicie znaku jest zapisywane *dopełnienie* modułu, tj. negacja każdego z bitów (zero jest zastępowane jedynką, jedynka zerem). W kodowaniu ZU2 dla liczb ujemnych po bicie znaku jest zapisywane dopełnienie modułu powiększone o 1. Porównanie tych kodowań dla wybranych wartości z zakresu jednego bajta przedstawiono w tabl. 3.2.

Jak widać, w kodowaniach ZM i ZU1 występuje podwójna reprezentacja zera. Wady tej jest pozbawione kodowanie ZU2, pozwala ono także na reprezentację liczb z nieco szerszego zakresu, jak również ułatwia przeprowadzanie działań arytmetycznych, dzięki czemu jest powszechnie stosowane.

Dotychczas przedstawione kodowania pozwalały na reprezentację liczb całkowitych. Ułamki wymagają nieco innego podejścia. Rozważając zapis ułamków dziesiętnych w systemie dziesiętnym, można zauważyć, że w zapisie matematycz-

Tablica 3.2. Porównanie kodowań ZM, ZU1, ZU2

Liczba dziesiętna	ZM	ZU1	ZU2
-128			10000000
-127	11111111	10000000	10000001
-126	11111110	10000001	10000010
-125	11111101	10000010	10000011
...	...	...	...
-1	10000001	11111110	11111111
-0	10000000	11111111	
0			00000000
+0	00000000	00000000	
1	00000001	00000001	00000001
...	...	...	...
126	01111110	01111110	01111110
127	01111111	01111111	01111111

nym pojawia się symbol separatora dziesiętnego, oddzielającego część całkowitą od ułamkowej. Separatorem tym jest przecinek (w Polsce) lub kropka (w krajach anglosaskich). Pierwsza cyfra po separatorze oznacza liczbę części dziesiętnych, druga setnych itd. Analogicznie można przedstawiać ułamki dwójkowe w systemie binarnym, pamiętając, że kolejne cyfry po separatorze będą reprezentować mnożniki kolejnych ujemnych potęg dwójki ($\frac{1}{2}$, $\frac{1}{4}$ itd.). Przykład takiego zapisu jest następujący:

$101{,}011_{(2)} = 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2} + 1 * 2^{-3} = 4 + 1 + \frac{1}{4} + \frac{1}{8} = 5{,}375$

Zauważmy, że niektóre ułamki dziesiętne zapisane w systemie dwójkowym mają nieskończone rozwinięcie dwójkowe. Przykładowo, jedna dziesiąta w zapisie dwójkowym jest ułamkiem okresowym.

$0{,}1 = 0{,}0(0011)_2$

Wykorzystując ułamki dwójkowe o skończonym rozwinięciu, możemy reprezentować liczby dziesiętne ze skończoną dokładnością. Kumulacja błędów numerycznych podczas obliczeń może prowadzić do uzyskania nieprawidłowego wyniku końcowego. Aby tego uniknąć, niekiedy stosuje się specyficzne zabiegi [29], których omówienie wykracza jednak poza ramy niniejszej pracy.

W zapisie matematycznym ułamka dwójkowego występuje dodatkowy symbol separatora. Reprezentując taki ułamek w pamięci komputera, konieczne jest przyjęcie pewnego kodowania pozwalającego na uniknięcie zapisu tego symbolu, tak aby pojedynczy bit oznaczał jedynie cyfrę dwójkową. Niekiedy w prostych zastosowaniach, jak np. systemach wbudowanych opartych na mikrokontrolerach jednoukładowych, wystarczające jest kodowanie stałoprzecinkowe, w którym ustalona liczba bitów reprezentuje część ułamkową. Częściej spotyka się reprezen-

tację zmiennoprzecinkową, w której liczba reprezentowana ma postać $\pm m * 2^p$, gdzie m jest *mantysą*, a p *cechą* (wykładnikiem) liczby. Aby uniknąć możliwości niejednoznacznego zapisu tej samej wartości, mantysa zazwyczaj jest znormalizowana, tj. należy do przedziału $[1; 2)$. Oznacza to, że składa się z pojedynczego bitu części całkowitej (zawsze równego jednemu) i pewnej liczby bitów o długości zależnej od przyjętego kodowania reprezentującej część ułamkową. Zazwyczaj w powszechnie przyjętych standardach kodowania liczb zmiennoprzecinkowych pierwszy bit mantysy jako ustalony pomija się w zapisie dla oszczędności miejsca. Wykładnik liczby jest zazwyczaj zapisywany w postaci spolaryzowanej, to jest zwiększonej o pewną stałą wartość, tak aby ułatwić reprezentację wykładników ujemnych. Przykładem takiego kodowania liczb zmiennoprzecinkowych jest standard IEEE-754 [12]. Liczby *pojedynczej precyzji* są w nim zapisywane na trzydziestu dwóch bitach. Najbardziej znaczący bit jest bitem znaku. Kolejne osiem bitów koduje wykładnik zwiększony o 127. Ostatnie dwadzieścia trzy bity reprezentują mantysę z pominiętym najstarszym bitem.

Rozważmy przykładowe przeliczenie 00111111010000000000000000000000 z kodowania IEEE-754 na wartość dziesiętną. Najstarszy bit (bit znaku) wynosi 0, co oznacza, że jest to liczba dodatnia. Kolejne osiem bitów ma postać 01111110. Liczba $01111110_{(2)}$ dwójkowo to 126 dziesiętnie. Aby otrzymać wykładnik, od wartości tej musimy odjąć 127, a więc wykładnik wynosi $p = -1$. Ostatnie dwadzieścia trzy bity reprezentują część ułamkową mantysy. Jak już wspomniano, część całkowita wynosi zawsze 1. Oznacza to, że w naszym przypadku mantysa (po usunięciu końcowych nieznaczących zer) wynosi $m = 1{,}1_{(2)}$ dwójkowo, czyli $1{,}5$ dziesiętnie. Ostatecznie więc wartość liczby wynosi $1{,}5 * 2^{-1} = 0{,}75$.

Standard IEEE-754 definiuje także szczególne przypadki, np. dla oznaczenia liczb zbyt dużych lub małych dla przyjętego kodowania ($\pm\infty$) i wyników obliczeń, które sygnalizują błąd (NaN – *Not a Number*). Szczegółowe omówienie tych niuansów wykracza poza zakres niniejszej pracy, dociekliwego Czytelnika odsyłamy do publikacji [12].

3.3. Podstawy arytmetyki i logiki komputerów

Działania arytmetyczne w systemie dwójkowym można wykonywać podobnie jak znane ze szkoły obliczenia pisemne w systemie dziesiętnym. Podczas dodawania sumujemy pary bitów na odpowiadających sobie pozycjach, począwszy od najmniej znaczącego bitu, uwzględniając ewentualne przeniesienie z poprzedniej pozycji. Przykład takiego sumowania dla liczb 38 i 12 zapisanych dwójkowo pokazano na rys. 3.3.

Podczas mnożenia wyznaczamy iloczyny częściowe mnożnej przez kolejne bity mnożnika, a następnie je sumujemy. Zauważmy, że poszczególne iloczyny

$$
\begin{array}{r}
100110 \\
+ \quad 1100 \\
\hline
110010
\end{array}
$$

Rys. 3.3. Sumowanie
liczb binarnych

częściowe będą wynosić zero, jeżeli dany bit mnożnika był zerem, będą zaś równe mnożnej, jeżeli dany bit mnożnika był jedynką. Oznacza to, że w praktyce mnożenie sprowadza się do serii przesunięć mnożnej w lewo i dodawań do dotychczasowego wyniku pośredniego. Przykład mnożenia liczb 38 i 5 zapisanych dwójkowo pokazano na rys. 3.4.

$$
\begin{array}{r}
100110 \\
* \quad 101 \\
\hline
100110 \\
100110 \\
\hline
10111110
\end{array}
$$

Rys. 3.4. Mnożenie
liczb binarnych

Dodawanie i odejmowanie liczb ze znakiem zapisanych w kodzie ZU2 wykonujemy na wszystkich bitach, włącznie z bitem znaku, tak jakby to były liczby zapisane w kodowaniu NB. Ewentualne przeniesienie z bitu znaku jest pomijane. Otrzymany wynik jest również zapisany w kodowaniu ZU2. Przykład dodawania liczb 38 i -72 zapisanych w kodzie ZU2 na jednym bajcie pokazano na rys. 3.5.

$$
\begin{array}{r}
00100110 \\
+ \quad 10111000 \\
\hline
11011110
\end{array}
$$

Rys. 3.5. Sumowanie
liczb w kodzie ZU2

Należy pamiętać, że wynik takich obliczeń może przekroczyć zakres wartości reprezentowanych przez kodowanie na danej liczbie bitów. Nadmiar wystąpi, gdy było przeniesienie z najbardziej znaczącego bitu modułu sumy na bit znaku i jednocześnie nie wystąpiło przeniesienie z bitu znaku lub gdy nie było przeniesienia z najbardziej znaczącego bitu modułu sumy na bit znaku i jednocześnie wystąpiło przeniesienie z bitu znaku. Obserwację tej sytuacji podczas dodawania dwóch liczb dodatnich (np. $125 + 10$) lub dwóch ujemnych (np. $-125 + (-10)$) pozostawiamy Czytelnikowi w ramach ćwiczeń.

Omówienie sposobu przeprowadzania innych działań w różnych kodowaniach wykracza poza zakres niniejszego skryptu. Szczegółowy opis można znaleźć np. w pracach [16, 18].

Drugą grupą operacji, poza działaniami arytmetycznymi, są funkcje logiczne. Ich omówienie należy poprzedzić wprowadzeniem pojęcia *algebry Boole'a*. Matematycznie algebra Boole'a jest krotką $(\mathbb{B}, \vee, \wedge, ', \mathbf{0}, \mathbf{1})$, gdzie $\mathbb{B}$ jest pewnym zbiorem zamkniętym ze względu na wymienione dalej działania, $\vee, \wedge$ – działaniami dwuargumentowymi, zwanymi odpowiednio sumą i iloczynem, $'$ – działaniem jednoargumentowym, zwanym dopełnieniem, a $\mathbf{0}$ i $\mathbf{1}$ – różnymi elementami spełniającymi następujące własności [24]:

$$x \vee y = y \vee x \tag{3.1}$$

$$x \wedge y = y \wedge x \tag{3.2}$$

$$(x \vee y) \vee z = x \vee (y \vee z) \tag{3.3}$$

$$(x \wedge y) \wedge z = x \wedge (y \wedge z) \tag{3.4}$$

$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z) \tag{3.5}$$

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z) \tag{3.6}$$

$$x \vee \mathbf{0} = x \tag{3.7}$$

$$x \wedge \mathbf{1} = x \tag{3.8}$$

$$x \vee x' = \mathbf{1} \tag{3.9}$$

$$x \wedge x' = \mathbf{0} \tag{3.10}$$

Równania (3.1), (3.2) nazywamy prawami przemienności, (3.3), (3.4) prawami łączności, (3.5), (3.6) prawami rozdzielności, (3.7), (3.8) prawami identyczności, a (3.9), (3.10) prawami dopełnienia. Zauważmy, że konstrukcja tych praw jest dualna, tj. jeśli w dowolnym z nich zamienimy ze sobą działania $\vee$ z $\wedge$ i jednocześnie elementy $\mathbf{0}$ z $\mathbf{1}$, to otrzymamy drugie z praw. Na tej podstawie można

sformułować *zasadę dualności* mówiącą o tym, że jeżeli $(\mathbb{B}, \vee, \wedge, ', \mathbf{0}, \mathbf{1})$ jest algebrą Boole'a, to $(\mathbb{B}, \wedge, \vee, ', \mathbf{1}, \mathbf{0})$ także jest algebrą Boole'a. Dowód tej własności pozostawiamy Czytelnikowi.

Opierając się na prawach (3.1)–(3.10), można sformułować i udowodnić szereg dalszych własności, takich jak np. prawa idempotentności, pochłaniania, De Morgana itd. Ich przedstawienie wykracza jednak poza ramy niniejszej pracy, wykazanie niektórych z nich zaproponowano jako zadania do samodzielnego rozwiązania przez Czytelnika, które zamieszczono na końcu rozdziału. Szczegółowy opis metod można znaleźć np. w publikacjach [16, 24].

Możemy rozważać różne algebry Boole'a. Przykładowo, dla dowolnego niepustego zbioru S krotka $\left(2^S, \cup, \cap, ', \emptyset, S\right)$, gdzie 2^S jest zbiorem wszystkich podzbiorów zbioru S; $\cup, \cap$ są odpowiednio operacjami sumy i przecięcia zbiorów, $'$ dopełnieniem zbioru, $\emptyset$ zaś zbiorem pustym, jest algebrą Boole'a. W szczególności dla zbioru jednoelementowego otrzymujemy następującą algebrę Boole'a: $(\{\emptyset, S\}, \cup, \cap, ', \emptyset, S)$. W dalszej części rozdziału, o ile nie zostanie zaznaczone inaczej, będziemy posługiwać się zerojedynkową algebrą Boole'a określoną jako $(\{0, 1\}, \mathrm{OR}, \mathrm{AND}, \mathrm{NOT}, 0, 1)$, z funkcjami logicznymi OR, AND i NOT zdefiniowanymi jak na rys. 3.6-3.8.

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

Rys. 3.6. Funkcja OR

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

Rys. 3.7. Funkcja AND

x	NOT x
0	1
1	0

Rys. 3.8. Funkcja NOT

W celu uproszczenia zapisu niekiedy funkcję OR oznacza się przez $+$, AND zaś przez $\cdot$ lub wręcz pomija się ten symbol, jeżeli nie prowadzi to do niejednoznaczności, natomiast negację NOT zapisuje się jako kreskę nad symbolem zmiennej, np. $\overline{x}$. Przedstawione na rys. 3.6-3.8 funkcje logiczne stanowią podstawowy *system funkcjonalnie pełny*, tj. przy ich użyciu można wyrazić dowolną inną funkcję logiczną. Przykładowo, funkcja XOR (oznaczana często symbolem $\oplus$), określona jak na rys. 3.9, może być zapisana jako $x \oplus y = x\overline{y} + \overline{x}y$.

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

Rys. 3.9. Funkcja XOR

Przedstawiony podstawowy system funkcjonalnie pełny jest nadmiarowy ze względu na liczbę funkcji. Istnieją systemy zawierające mniej funkcji logicznych, za pomocą których można także wyrazić dowolną inną funkcję. W szczególności dwa najmniejsze systemy funkcjonalnie pełne zawierają jedynie pojedynczą funkcję logiczną: NOR (rys. 3.10) lub NAND (rys. 3.11). Wykazanie, że za pomocą dowolnej z tych funkcji można zrealizować funkcje AND, OR i NOT pozostawiamy Czytelnikowi.

x	y	x NOR y
0	0	1
0	1	0
1	0	0
1	1	0

Rys. 3.10. Funkcja NOR

x	y	x NAND y
0	0	1
0	1	1
1	0	1
1	1	0

Rys. 3.11. Funkcja NAND

Rozwój elektroniki cyfrowej zaowocował produkcją układów scalonych zawierających tzw. *bramki logiczne* realizujące elementarne funkcje. Korzystając

z właściwości systemu funkcjonalnie pełnego, możliwe było opracowanie układów obliczających dowolną funkcję logiczną. Konstrukcja układu obliczającego pewną funkcję arytmetyczną również może być potraktowana jako specyficzna funkcja logiczna. Przykładowo, operacja dodawania arytmetycznego (aby uniknąć pomyłki z funkcją OR, dalej oznaczana przez $\boxplus$) dla dwóch bitów $x \boxplus y$ może zostać opisana jako: $s = x \oplus y$, $c = xy$, gdzie s jest bitem wyniku, a c bitem przeniesienia (rys. 3.12). Układ realizujący taką operację nazwiemy *półsumatorem* [7].

x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Rys. 3.12. Tablica prawdy półsumatora

Zaprojektowanie układu realizującego operację dodawania bitów na odpowiadających sobie pozycjach dwóch liczb binarnych wymaga uwzględnienia przeniesienia z poprzedniej pozycji. Układ zwany *sumatorem* [7] ma trzy wejścia: x_i, y_i (na które są podawane bity z i-tej pozycji) i c_{i-1} (uwzględniające przeniesienie z poprzedniej pozycji) oraz dwa wyjścia: s_i – i-ty bit sumy i c_i – przeniesienie z i-tej pozycji. Stan na wyjściach jest określony następująco:

$s_i = x_i \oplus y_i \oplus c_{i-1}$

$c_i = x_i \cdot y_i + (x_i \oplus y_i) \cdot c_{i-1}$

Tablicę prawdy dla takiego układu pokazano na rys. 3.13.

x_i	y_i	c_{i-1}	c_i	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Rys. 3.13. Tablica prawdy sumatora

Kaskadowo łącząc opisane sumatory jednobitowe wejściami i wyjściami przeniesień, możemy utworzyć *sumator wielobitowy z przeniesieniem szeregowym*.

Propagacja przeniesień oznacza, że potrzebny jest dłuższy czas na ustalenie wyniku na najbardziej znaczących bitach. Zwiększenie szybkości obliczeń można uzyskać, stosując sumatory z *przeniesieniem równoległym*, w których są jednocześnie przewidywane przeniesienia na poszczególnych pozycjach na podstawie stanów wejść. Szczegółowy opis tego zagadnienia wykraczający poza ramy niniejszego skryptu przedstawiono np. w pracy [16].

W analogiczny sposób możliwe jest budowanie układów realizujących inne funkcje arytmetyczne. *Arytmometrem* albo jednostką arytmetyczno-logiczną ALU (*Arithmetic Logic Unit*) nazywamy układ realizujący pewne funkcje arytmetyczne i logiczne. Przykładem czterobitowego arytmometru jest układ scalony SN74181. W zależności od stanu wejść sterujących S_0–S_3 i wyboru trybu pracy M realizuje on wybraną funkcję logiczną lub arytmetyczną na dwóch liczbach czterobitowych podanych na wejścia A_0–A_3 i B_0–B_3, jak pokazano w tabl. 3.3 opracowanej na podstawie [30]. Przykładowo, dla $S = 1001$ i $M = 0$ arytmometr ten realizuje dodawanie arytmetyczne liczb A i B. Jednostki te można łączyć ze sobą i generatorami przeniesień SN74182 [30] w celu powiększenia rozmiaru słowa obliczeniowego.

Tablica 3.3. Funkcje arytmometru SN74181, na podstawie [30]

Wejścia sterujące				Logika ujemna na liniach danych		
				$M = H$	$M = L$ (operacje arytmetyczne)	
S_3	S_2	S_1	S_0	(funkcje logiczne)	$C_n = L$ (bez przeniesienia)	$C_n = H$ (z przeniesieniem)
L	L	L	L	$F = \overline{A}$	F = A MINUS 1	F = A
L	L	L	H	$F = \overline{AB}$	F = AB MINUS 1	F = AB
L	L	H	L	$F = \overline{A} + B$	$F = A\overline{B}$ MINUS 1	$F = A\overline{B}$
L	L	H	H	$F = 1$	F = MINUS 1 (kod ZU2)	F = ZERO
L	H	L	L	$F = \overline{A + B}$	$F = A$ PLUS $(A + \overline{B})$	$F = A$ PLUS $(A + \overline{B})$ PLUS 1
L	H	L	H	$F = \overline{B}$	$F = AB$ PLUS $(A + \overline{B})$	$F = AB$ PLUS $(A + \overline{B})$ PLUS 1
L	H	H	L	$F = \overline{A \oplus B}$	F = A MINUS B MINUS 1	F = A MINUS B
L	H	H	H	$F = A + \overline{B}$	$F = A + \overline{B}$	$F = (A + \overline{B})$ PLUS 1
H	L	L	L	$F = \overline{A}B$	$F = A$ PLUS $(A + B)$	$F = A$ PLUS $(A + B)$ PLUS 1
H	L	L	H	$F = A \oplus B$	F = A PLUS B	F = A PLUS B PLUS 1
H	L	H	L	$F = B$	$F = A\overline{B}$ PLUS $(A + B)$	$F = A\overline{B}$ PLUS $(A + B)$ PLUS 1
H	L	H	H	$F = A + B$	$F = (A + B)$	$F = (A + B)$ PLUS 1
H	H	L	L	$F = 0$	F = A PLUS A‡	F = A PLUS A PLUS 1
H	H	L	H	$F = A\overline{B}$	F = AB PLUS A	F = AB PLUS A PLUS 1
H	H	H	L	$F = AB$	$F = A\overline{B}$ PLUS A	$F = A\overline{B}$ PLUS A PLUS 1
H	H	H	H	$F = A$	F = A	F = A PLUS 1

‡ Każdy bit jest przesuwany na kolejną pozycję znaczącą.

Jednostka arytmetyczno-logiczna jest jedną z części składowych procesora. Oprócz niej można wyróżnić *jednostkę sterującą*, *pamięć* i *układy wejścia/wyjścia*. Jednostka sterująca sekwencyjnie realizuje kolejne kroki programu, pobierając z pamięci rozkaz do wykonania i dane wejściowe, podając je na wejścia

ALU i zapisując wynik. W *architekturze von Neumanna* wspólna pamięć zawiera zarówno program do wykonania, jak też dane. W *architekturze harwardzkiej* pamięci i magistrale programu i danych są rozdzielone. Obie te podstawowe architektury są stosowane we współczesnych mikroprocesorach.

Pamięć ze względu na sposób obsługi można podzielić na *pamięć o dostępie swobodnym* (RAM – *Random Access Memory*) i *pamięć tylko do odczytu* (ROM – *Read Only Memory*). Obecnie ze względów technologicznych pamięć ROM jest zazwyczaj wykonywana w technologii Flash EEPROM (*Electrically-Erasable Programmable Read-Only Memory*) pozwalającej na ograniczoną liczbę cykli przeprogramowania bezpośrednio w systemie komputerowym. W pamięci tej jest zapisany kompletny program do wykonania w przypadku niewielkich mikrokontrolerów jednoukładowych bądź też fragment programu ładującego, przepisujący właściwy program do pamięci RAM i uruchamiający go. Pamięć RAM pozwala na praktycznie nieograniczoną liczbę cykli zapisu i odczytu, ale jej zawartość jest tracona podczas zaniku zasilania.

Specyficznym rodzajem pamięci, do której mikroprocesor ma bezpośredni dostęp, są *rejestry*. Występują rejestry dedykowane, mające specyficzne przeznaczenie oraz uniwersalne wykorzystywane do różnych celów. Pojedynczy rejestr ma niewielką pojemność, w zależności od mikroprocesora od jednego do kilku bajtów, niekiedy rejestry mogą być łączone w słowa. Jądro mikroprocesora zazwyczaj zawiera od kilku do kilkudziesięciu rejestrów. W niektórych mikroprocesorach występuje wyróżniony rejestr zwany *akumulatorem*, na którym są wykonywane operacje logiczne i arytmetyczne. Innym specyficznym rejestrem jest *rejestr stanu* przechowujący znaczniki ostatniego wyniku obliczeń. Pozwala to na łatwe wykonanie skoku warunkowego w zależności od tego, czy ostatni wynik był dodatni, zerowy itp. *Licznik rozkazów* przechowuje adres kolejnego rozkazu do wykonania. Zazwyczaj jest on inkrementowany w trakcie wykonywania polecenia, tak by sekwencyjnie realizować kolejne rozkazy. Instrukcja skoku (bezwarunkowego lub warunkowego) modyfikuje licznik rozkazów, zmieniając przebieg wykonywania programu. Na podstawie licznika rozkazów jest pobierane kolejne polecenie do wykonania z pamięci programu do *rejestru rozkazów*. Wartość w rejestrze jest dekodowana przez *dekoder rozkazów* wytwarzający odpowiednie sygnały sterujące poszczególnymi blokami mikroprocesora i podłączonymi peryferiami.

W architekturze mikroprocesora można także wyróżnić magistrale służące do przesyłu poszczególnych sygnałów. Magistrala danych pozwala na przesyłanie informacji pomiędzy poszczególnymi blokami procesora, jak również łączy mikroprocesor z układami peryferyjnymi. Magistrala sterująca określa tryb pracy wskazanego układu. Magistrala adresowa pozwala na zaadresowanie konkretnej

komórki pamięci zewnętrznej lub urządzenia wejścia/wyjścia [25]. W kolejnym podrozdziale przedstawiono krótki opis przykładowego mikroprocesora.

3.4. Jądro mikroprocesora AVR

Przykładowymi współczesnymi prostymi mikroprocesorami ośmiobitowymi są mikroprocesory z rodziny Atmel AVR. Niniejsza praca zawiera jedynie skrótowe omówienie jądra procesora, Czytelnik zainteresowany szczegółami może sięgnąć do literatury, np. [8], oraz dokumentacji producenta.

Mikroprocesory AVR w porównaniu z innymi układami dostępnymi obecnie na rynku posiadają wiele zalet i interesujących rozwiązań. Przede wszystkim są one układami typu RISC (*Reduced Instruction Set Computer*) wykonującymi większość instrukcji w pojedynczym cyklu zegara. Pozwala to na osiągnięcie wysokiej wydajności, aż do 1 MIPS (*Million Instructions Per Second*) na każdy megaherc zegara. W niektórych procesorach (jak np. ATmega 162) częstotliwość wewnętrznego zegara może być programowo zmieniana, np. w celu obniżenia poboru energii, gdyż nie występuje tu sztywny wstępny podział częstotliwości jak choćby w układach zgodnych z MCS 51, a konfigurowalny preskaler umożliwia regulowany podział od 1 (brak podziału) aż do 256. Ponieważ ATmega 162 może pracować maksymalnie z zegarem wewnętrznym równym 16 MHz, pozwala to na osiągnięcie wydajności 16 MIPS, co jest dobrym wynikiem jak na procesor tej klasy. W wyższych modelach (jak właśnie ATmega 162) zaimplementowano także sprzętowy multiplikator umożliwiający szybkie sprzętowe mnożenie liczb ośmiobitowych. Kolejną zaletą pozwalającą na osiąganie wysokiej wydajności jest architektura harwardzka o rozdzielonej pamięci programu i danych, a także osobnych magistralach tych pamięci.

Należy podkreślić, że jądro procesorów AVR było projektowane w ścisłej współpracy z projektantami kompilatorów języków wysokiego poziomu, co zaowocowało dobrym dopasowaniem architektury procesora do wymagań stawianych przez kompilator. Dzięki temu programy pisane w wysokopoziomowych językach mogą być zoptymalizowane, a tworzenie efektywnego kodu jest prostsze [20].

Ważną zaletą wartą podkreślenia jest wysoka symetria procesora, czyli równouprawnienie rejestrów i ich dostępność dla poszczególnych funkcji, a także duża ortogonalność, czyli dostępność zaimplementowanych trybów adresowania w wielu instrukcjach i dla wielu operandów. Procesory AVR zawierają trzydzieści dwa ośmiobitowe rejestry ogólnego przeznaczenia. Większość instrukcji może całkowicie dowolnie korzystać z tych rejestrów, przy czym żaden z nich nie jest wyróżniony (nie występuje tzw. akumulator). Dzięki temu kod wynikowy jest bardziej zwarty, gdyż unika się zbędnych rozkazów transferu danych. Sześć ostatnich

rejestrów może być łączonych w pary, tworząc trzy szesnastobitowe rejestry adresowe. Zestaw instrukcji udostępnia wiele różnych trybów adresowania, w tym adresowanie względne z automatycznym zwiększaniem (lub zmniejszaniem) adresu po (lub przed) wykonaniu operacji bądź z dodaniem sześciobitowego przemieszczenia, co umożliwia np. łatwe adresowanie elementów tablicy, bądź stosu bez konieczności zmiany rejestru ciągle wskazującego na pierwszy element.

Warto zauważyć, że procesory AVR jako nieliczne posiadają aż trzy rejestry indeksowe umożliwiające adresowanie względne. Kompilatory języków wysokiego poziomu zazwyczaj zajmują jeden z takich rejestrów na potrzeby programowego stosu tworzonego w pamięci operacyjnej. Fakt pozostania jeszcze dwóch takich rejestrów umożliwia wygodne adresowanie jednocześnie dwóch różnych miejsc w pamięci (co często występuje w programie, np. podczas kopiowania zawartości pamięci z jednego miejsca w drugie) bez potrzeby ciągłego przeładowywania rejestrów. Możliwość wykorzystania automatycznej inkrementacji tych rejestrów jest dodatkowym udogodnieniem.

Kolejną zaletą jest bogactwo instrukcji warunkowych umożliwiających zarówno dowolny skok w przypadku spełnienia określonego warunku, jak też warunkowy skok o jedną instrukcję, będący w istocie warunkowym wykonaniem pojedynczego rozkazu. Rozkaz ten może być dowolny, w szczególności może to być instrukcja skoku bezwarunkowego, ale także np. ustawienie pojedynczego bitu czy też wywołanie procedury. Ułatwia to optymalizację kodu przez kompilator wyższego poziomu (jak np. C), jak też zwiększa czytelność programu pisanego w asemblerze poprzez znaczne zmniejszenie liczby etykiet.

Warto również wspomnieć o dużej liczbie instrukcji działających na poszczególnych bitach, w szczególności instrukcje te umożliwiają ustawienie bądź wyzerowanie dowolnego bitu w dowolnym rejestrze, a także warunkowe rozgałęzienie programu w zależności od wartości pojedynczego bitu. Ten sam efekt można oczywiście uzyskać także w klasyczny sposób, stosując normalne bajtowe operacje logiczne i maskowanie, przy czym wydłuża to zarówno rozmiar programu, jak też czas potrzebny na jego wykonanie.

Kolejną istotną innowacją występującą w procesorach AVR jest przenoszenie flagi zera. Podczas porównywania liczb o rozmiarze większym niż rozmiar pojedynczego rejestru, czyli dłuższych niż osiem bitów, konieczne jest wykonanie kilku kolejnych porównań. Przykładowo, gdyby flaga zera zależała jedynie od ostatniej instrukcji, wówczas w celu porównania dwóch liczb trzydziestodwubitowych należałoby wykonać cztery ośmiobitowe porównania, a po każdym z nich testować flagę zera, by sprawdzić, czy poszczególne części tych liczb były równe. Testowanie flagi jedynie po ostatnim porównaniu mijałoby się z celem, gdyż flaga ta zależałaby jedynie od ostatniej operacji i pokazywałaby równość ośmiu najbardziej znaczących bitów, nie zaś całych trzydziestodwubitowych liczb. Projektanci

procesorów AVR pomyśleli także o tym i w procesorach tych flaga zera jest przenoszona – zależy ona od wszystkich porównań składowych, nie zaś jedynie od ostatniego. Umożliwia to łatwe i optymalne porównywanie dużych liczb.

Wszystkie wymienione elementy bezpośrednio przekładają się na łatwość tworzenia optymalnego i efektywnego kodu w językach wysokiego poziomu, takich jak język C. Według testów polegających na kompilowaniu tych samych programów na różne procesory, procesor AVR zawsze mieścił się w czołówce, jeśli chodzi o zwartość kodu wynikowego, po podsumowaniu zaś zwyciężył [2]. Kod wynikowy na procesor AVR zajmował średnio prawie dwukrotnie mniej niż na procesor zgodny z MCS 51. Należy podkreślić, że programy te nie były w żaden sposób optymalizowane na konkretny procesor, przypuszczalnie optymalizacja pozwoliłaby na dalsze zmniejszenie kodu wynikowego.

3.5. Zadania

1. Przelicz na układ dziesiętny:
   - $101_{(2)}$,
   - $11010111_{(2)}$,
   - $101_{(16)}$,
   - $\mathrm{BACD}_{(16)}$.
2. Przelicz na układ dwójkowy:
   - $101_{(10)}$,
   - $637_{(10)}$,
   - $101_{(16)}$,
   - $\mathrm{BACD}_{(16)}$.
3. Przelicz na układ szesnastkowy:
   - $101_{(2)}$,
   - $11010111_{(2)}$,
   - $101_{(10)}$,
   - $637_{(10)}$.
4. Zapisz w kodowaniach ZM, ZU1, ZU2 na jednym bajcie każdą z liczb:
   - -3,
   - -101,

- -38.

5. Wykonaj działania w układzie dwójkowym:
   - $1100101 + 1101$,
   - $1100101 * 1101$.
6. Zapisz liczby w kodowaniu ZU2 i oblicz. Wynik przelicz na układ dziesiętny:
   - $-37 + 48$,
   - $-37 + (-48)$,
   - $56 - 60$,
   - $-56 + 60$.
7. Wykaż, że w dowolnej algebrze Boole'a zachodzą prawa idempotentności:
   - $x \vee x = x$,
   - $x \wedge x = x$.
8. Wykaż, że w dowolnej algebrze Boole'a zachodzą prawa pochłaniania:
   - $(x \wedge y) \vee x = x$,
   - $(x \vee y) \wedge x = x$.
9. Wykaż, że w dowolnej algebrze Boole'a zachodzą prawa De Morgana:
   - $(x \vee y)' = x' \wedge y'$,
   - $(x \wedge y)' = x' \vee y'$.
10. Wykaż, że $(\{0, 1\}, \mathrm{AND}, \mathrm{OR}, \mathrm{NOT}, 1, 0)$ jest algebrą Boole'a.
11. Wykaż, że funkcja XOR może być wyrażona w następujący sposób: $x \oplus y = x\overline{y} + \overline{x}y$.
12. Zapisz funkcje AND, OR, XOR i NOT jedynie przy użyciu funkcji:
    - NOR,
    - NAND.

Rozdział 4.

Elementy algorytmów i struktur danych

Dariusz Rzońca

4.1. Wprowadzenie

Niniejszy rozdział zawiera skrótowy opis najistotniejszych zagadnień związanych z algorytmiką oraz wprowadzenie do podstawowych struktur danych. Po wyjaśnieniu pojęcia algorytmu przedstawiono typowe sposoby jego zapisu. Omówiono semantykę schematów blokowych i pseudojęzyka. Skrótowo opisano podstawowe elementy programów, instrukcje warunkowe, pętle, przedstawiono techniki programowania iteracyjnego i rekurencyjnego. Zaprezentowano kilka prostych algorytmów, jak np. algorytm Euklidesa. W dalszej części rozdziału zawarto podstawowe informacje związane z elementarnymi strukturami danych, takimi jak tablica, lista, stos i kolejka. Na przykładzie wybranych algorytmów sortowania omówiono elementarne pojęcia związane ze złożonością obliczeniową algorytmów.

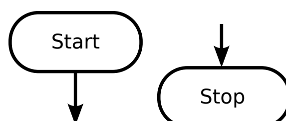
4.2. Podstawy algorytmiki

Nieformalnie *algorytm* definiuje się zazwyczaj jako przepis czy też metodę postępowania prowadzącą do rozwiązania pewnego problemu. Przepis taki musi spełniać szereg warunków. Powinien składać się z ciągu kolejno wykonywanych poleceń, przy czym każde z nich musi być *jednoznacznie określone*, tj. zdefiniowane w sposób niebudzący wątpliwości interpretacyjnych. Wymaga się także, by każda z operacji była *wykonalna*, tj. możliwa do zrealizowania. Istotną cechą jest *skończoność* algorytmu, tj. wymóg zakończenia jego działania po wykonaniu skończonej liczby operacji. Zazwyczaj algorytm działa na pewnych *danych wejściowych*, a w wyniku ich przetwarzania uzyskujemy *dane wyjściowe*.

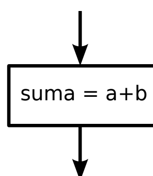
Zestaw poleceń możliwych do wykorzystania rozpatrywany jest w kontekście pewnego *modelu obliczeń*. W literaturze omówiono wiele takich abstrakcyjnych modeli, jak np. maszyna Turinga czy też maszyna ze swobodnym dostępem do pa-

mięci RAM (*Random Access Machine*) [28]. W maszynach tych program stanowi lista elementarnych rozkazów, podobnych do niskopoziomowego asemblera lub też kodu maszynowego. Aby ułatwić analizę algorytmów, można zapisywać je w postaci opisu słownego tworzącego program w wysokopoziomowym uproszczonym pseudojęzyku (pseudokodzie) lub też przedstawiać graficznie w formie schematów blokowych. W niniejszym rozdziale będą wykorzystywane oba te podejścia jako wzajemnie się uzupełniające.

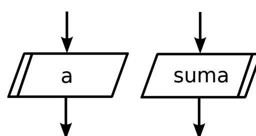
Schemat blokowy składa się z opisanych dalej figur geometrycznych reprezentujących poszczególne operacje, połączonych strzałkami ilustrującymi kolejność ich wykonywania. Początek (pierwszą operację do wykonania) i koniec programu (zatrzymanie działania) przedstawiamy w owalach z jedną strzałką, odpowiednio wychodzącą bądź wchodzącą jak na rys. 4.1. Polecenia do wykonania, jak np. przypisanie wartości do zmiennych, czy też obliczenia są reprezentowane przez prostokąty (rys. 4.2). Wczytywanie i wyświetlenie wartości zmiennych uwidaczniamy w równoległobokach z podwójną linią odpowiednio z lewej bądź prawej strony, tak jak to przedstawiono na rys. 4.3. Bardzo istotną rolę pełni blok warunkowy (decyzyjny) pozwalający na sterowanie przebiegiem wykonania programu. W zależności od wartości logicznej wyrażenia w bloku warunkowym jako kolejna jest wykonywana operacja wskazana przez strzałkę wychodzącą oznaczoną symbolem *T* (tak, prawda) bądź *N* (nie, fałsz) (rys. 4.4).



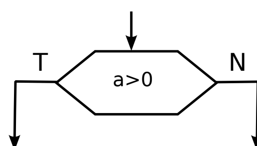
Rys. 4.1. Bloki początkowy i końcowy



Rys. 4.2. Blok operacyjny

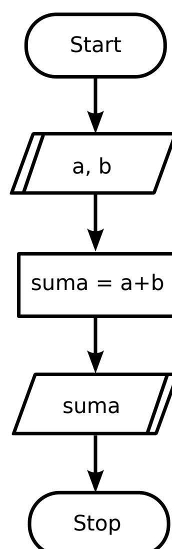


Rys. 4.3. Bloki wejściowy i wyjściowy



Rys. 4.4. Blok warunkowy

Opisane elementy łączy się na diagramie, tworząc schemat blokowy. Na rysunku 4.5 przedstawiono przykładowy schemat blokowy programu wczytującego dwie liczby i wyświetlającego ich sumę. Program po uruchomieniu wczytuje dwie liczby do zmiennych a i b w pojedynczym bloku wejściowym, następnie oblicza zmienną $suma$ w bloku operacyjnym, wypisuje jej wartość w bloku wyjściowym i kończy działanie.



Rys. 4.5. Program sumujący dwie liczby – schemat blokowy

Semantyka schematów blokowych jest stosunkowo ujednolicona. Pseudokod jest językiem abstrakcyjnym opartym na języku naturalnym, pozwalającym na sporą dowolność w zapisie, przez co różni autorzy zazwyczaj stosują własne specyficzne dialekty. Składnia stosowanego w niniejszym rozdziale pseudokodu jest podobna do opisanej w pracy [28], jedynie do operacji przypisania będzie stosowany symbol $=$ a nie $\leftarrow$. Należy zauważyć, że rozróżnienie znaczenia symbolu $=$ jako przypisanie bądź porównanie będzie się odbywać na podstawie kontekstu danego fragmentu programu, a jedynie tam, gdzie mogłoby to budzić wątpliwości

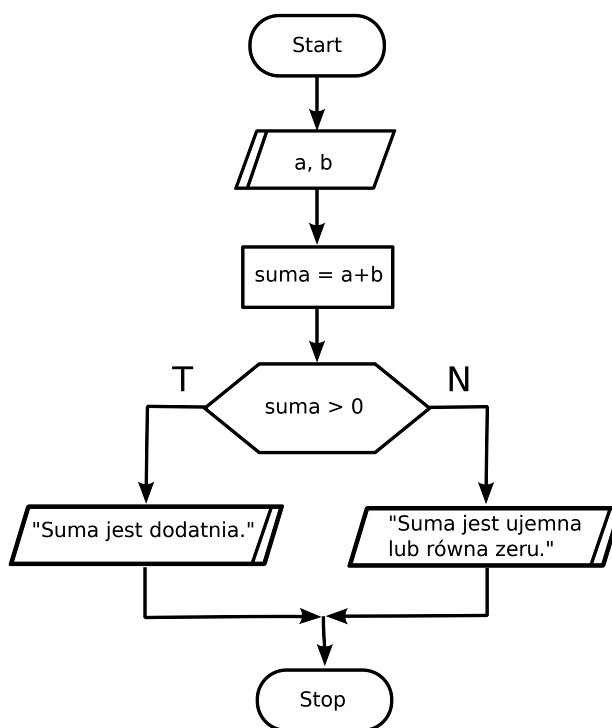
interpretacyjne pojawi się dodatkowy komentarz. Nieco inną semantykę pseudojęzyka można znaleźć w publikacji [6].

W pseudokodzie odpowiednikami bloków wejściowych i wyjściowych są instrukcje `czytaj` i `pisz`. Przykład programu sumującego dwie liczby, odpowiadający schematowi z rys. 4.5, przedstawiono na listingu 4.1.

Listing 4.1

```
SUMA
1  czytaj a
2  czytaj b
3  suma = a + b
4  pisz suma
```

Blok warunkowy powoduje rozgałęzienie programu. Można go użyć, aby wykonać odpowiedni fragment programu w zależności od warunku logicznego. Przykładowo, na rys. 4.6 przedstawiono zmodyfikowany program z rys. 4.5, w którym wykorzystano blok warunkowy, aby wyświetlić odpowiedni komunikat w zależności od znaku sumy.



Rys. 4.6. Program wyświetlający komunikat uzależniony od znaku sumy

W pseudokodzie odpowiednikiem bloku warunkowego jest instrukcja warunkowa. Ma ona postać `jeśli` *warunek* `to` *instrukcja* `inaczej` *instrukcja*. Przy prawdziwym warunku wykonywana jest instrukcja umieszczona po słowie kluczowym `to`, w przeciwnym razie instrukcja umieszczona po słowie `inaczej`. Człon `inaczej` i następująca po nim instrukcja są opcjonalne. W każdym z członów zamiast pojedynczej instrukcji do wykonania może wystąpić ciąg instrukcji ujęty w nawiasy klamrowe `{ }`. Celowe jest także stosowanie odpowiednich wcięć polepszających czytelność programu. Program w pseudokodzie realizujący to samo zadanie, co schemat z rys. 4.6 przedstawiono na listingu 4.2.

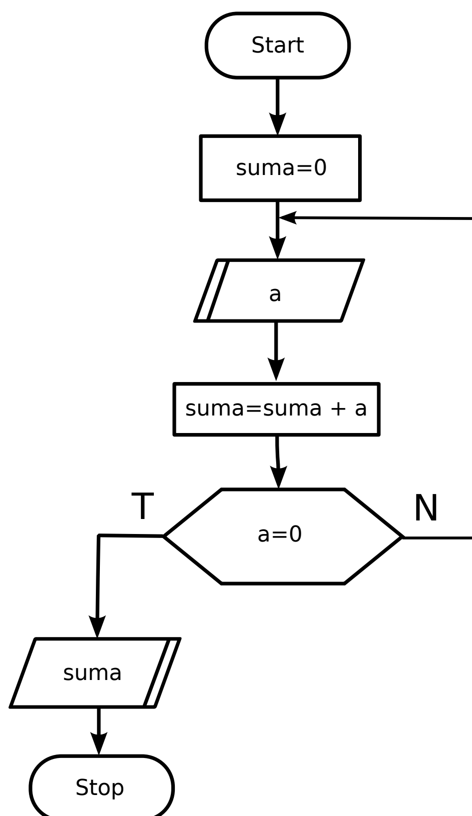
Listing 4.2

SUMA

```
1  czytaj a
2  czytaj b
3  suma = a + b
4  jeśli suma > 0
5  to pisz "Suma jest dodatnia."
6  inaczej pisz "Suma jest ujemna lub równa zeru."
```

Wykorzystanie na schemacie bloku warunkowego może często prowadzić do powstania tak zwanej *pętli*, w ramach której wielokrotnie są wykonywane te same operacje. Przykład programu zawierającego pętlę przedstawiono na rys. 4.7. Program ten sumuje kolejno wczytywane liczby, aż do momentu wczytania zera. Po wczytaniu zera wyświetlana jest obliczona suma i program kończy działanie. Zapis $suma = suma + a$ nie jest równaniem, należy traktować go jako polecenie zwiększenia obecnej wartości sumy (zmienna $suma$ z prawej strony wyrażenia) o wartość a i przypisania wyniku do zmiennej $suma$ (lewa strona wyrażenia).

Analizując przykład przedstawiony na rys. 4.7, można wyróżnić operacje inicjalizacyjne wykonywane jednokrotnie przed wejściem do pętli ($suma = 0$), operacje wielokrotnie ponawiane w pętli i warunek zakończenia pętli testowany w każdym obiegu. Części tego typu zazwyczaj można wyróżnić w każdej pętli, niekiedy poszczególne z nich mogą być puste. Należy podkreślić znaczenie nadania właściwych wartości początkowych poszczególnym zmiennym, gdyż niekiedy pominięcie tego kroku może prowadzić do błędów. W opisywanym przykładzie w ciele pętli bieżąca wartość zmiennej $suma$ każdorazowo jest zwiększana o a, więc końcowy wynik jest sumą nie tylko wczytanych liczb, ale także początkowej wartości tej zmiennej. Aby uniknąć przekłamania, konieczna jest więc jej inicjalizacja liczbą zero, czyli elementem neutralnym dodawania. Brak jawnej specyfikacji wartości początkowej w algorytmie może skutkować inicjalizacją przypadkową wartością w implementacji w formie programu komputerowego, zależnie od docelowego języka.



Rys. 4.7. Program sumujący wiele liczb

Pętle są bardzo często wykorzystywane w rzeczywistych programach. Większość języków programowania definiuje specjalne konstrukcje i słowa kluczowe wprowadzające różne rodzaje pętli, specyficzne dla danego języka i wspieranych przez niego paradygmatów programowania. W niektórych językach możliwa jest także „ręczna” implementacja pętli z wykorzystaniem instrukcji skoku warunkowego w sposób bezpośrednio odpowiadający zapisowi na schemacie blokowym. Postępowania takiego należy jednak zdecydowanie unikać, gdyż prowadzi do powstawania nieczytelnych programów, jak to szerzej opisano przy wyjaśnianiu paradygmatów programowania w rozdziale 5. Podobnie, w używanym w tym rozdziale pseudokodzie pętle będziemy zapisywać, stosując specyficzne konstrukcje, a nie instrukcje skoku.

Pierwszy operator pętli wykorzystany podczas tworzenia programu w pseudokodzie odpowiadającego schematowi z rys. 4.7 to operator `powtarzaj`. Pętle tworzone przy zastosowaniu tego operatora mają konstrukcję `powtarzaj` *instrukcja* `aż_do` *warunek*. Instrukcja (bądź też blok instrukcji ujęty w nawiasy

klamrowe) jest wielokrotnie powtarzany przy niespełnionym warunku opuszczenia pętli. Przykład programu w pseudokodzie analogicznego do schematu z rys. 4.7 pokazano na listingu 4.3.

Listing 4.3

```
SUMA
1 suma = 0
2 powtarzaj
3    {
4    czytaj a
5    suma = suma + a
6    }
7 aż_do a = 0
8 pisz suma
```

Warto zauważyć, że w programach z rys. 4.7 i listingu 4.3 zero wczytane jako symbol końcowy będzie także dodane do wartości sumy. Oczywiście nie zmieni to jej wartości, gdyż zero jest elementem neutralnym dodawania. Jeżeli w specyfikacji programu przyjęto by jednak inny symbol terminalny pozbawiony tej własności, to konstrukcja programu powinna być inna. Obecnie porównanie wartości zmiennej a z zerem $a = 0$ następuje na końcu pętli. Oznacza to w szczególności, że instrukcje w pętli wykonają się przynajmniej raz. Alternatywnie można rozważać taką konstrukcję programu, w której sprawdzenie warunku następuje przed każdym obiegiem pętli, a nie po nim. Wówczas w szczególnym przypadku instrukcje z ciała pętli mogą nigdy nie być wykonane. W pseudokodzie konstrukcję taką reprezentuje pętla postaci `dopóki` *warunek* `wykonuj` *instrukcja*. Odpowiedni program wykorzystujący taką konstrukcję pętli przedstawiono na listingu 4.4. Przeanalizowanie jego działania i zaprojektowanie odpowiadającego mu schematu blokowego pozostawiono Czytelnikowi w ramach ćwiczeń do samodzielnego wykonania.

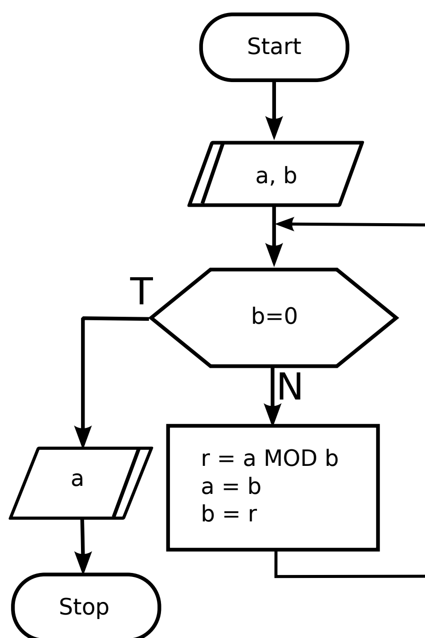
Listing 4.4

```
SUMA
1 suma = 0
2 czytaj a
3 dopóki a ≠ 0
4 wykonuj
5    {
6    suma = suma + a
7    czytaj a
8    }
9 pisz suma
```

Trzecim operatorem pętli występującym w pseudokodzie jest operator `dla`. Konstrukcja pętli ma postać `dla` *zmienna = wartość początkowa* `do` *wartość koń-*

cowa `wykonuj` *instrukcja*. Przed wejściem do pętli jest wykonywane przypisanie wartości początkowej do zmiennej sterującej. Po każdym obiegu pętli wartość zmiennej sterującej jest zwiększana o jeden. Po przekroczeniu wartości końcowej pętla jest opuszczana. Oczywiście pętle tego typu można zrealizować, stosując także poznane poprzednio konstrukcje i dodatkową inkrementację zmiennej sterującej w ciele pętli, co pozostawiono jako ćwiczenie Czytelnikowi.

Kolejnym przykładem prostego algorytmu, który zostanie przeanalizowany, jest *algorytm Euklidesa* wyznaczania największego wspólnego dzielnika liczb naturalnych. Należy on do najstarszych algorytmów spośród współcześnie używanych, opisał go grecki matematyk Euklides około trzysta lat przed naszą erą. Oparty jest on na twierdzeniu, że największy wspólny dzielnik dowolnych liczb naturalnych a i b jest równy największemu wspólnemu dzielnikowi liczby b i reszty z dzielenia liczby a przez b, co zapisujemy $\mathrm{NWD}(a, b) = \mathrm{NWD}(b, a \,\mathrm{MOD}\, b)$. Dowód tego twierdzenia można znaleźć np. w pracach [6, 18]. Przykładowy schemat blokowy programu wyznaczającego największy wspólny dzielnik liczb naturalnych według algorytmu Euklidesa pokazano na rys. 4.8. Napisanie odpowiadającego mu programu w pseudokodzie pozostawiono Czytelnikowi.



Rys. 4.8. Algorytm Euklidesa

Działanie przedstawionego programu zostanie prześledzone na przykładzie obliczania największego wspólnego dzielnika liczb 7514 i 5610. Wartości zmiennych w poszczególnych obiegach pętli pokazano na rys. 4.9. Po sześciu obiegach

pętli wartość b równa zeru wskazuje, że znaleziono największy wspólny dzielnik i należy opuścić pętlę. Wartość największego wspólnego dzielnika jest zapisana w zmiennej a i wynosi 34.

r	a	b
	7514	5610
1904	5610	1904
1802	1904	1802
102	1802	102
68	102	68
34	68	34
0	**34**	0

Rys. 4.9. Przykład działania algorytmu Euklidesa

Dotychczasowe przykłady były na tyle proste, że nie wymagały podziału na mniejsze części. W bardziej złożonych programach zazwyczaj wyodrębnia się fragmenty (podprogramy) realizujące pewne funkcjonalności. Pozwala to na uproszczenie konstrukcji programu i zwiększenie jego czytelności. W zależności od użytego języka programowania niekiedy podprogramy dzielimy na funkcje i procedury, w zależności od tego, czy zwracają jakąś wartość czy też nie. Paradygmat programowania proceduralnego szerzej opisano w rozdziale 5. W pseudokodzie podprogramy będziemy definiowali w ten sam sposób, co program. Nagłówek programu (podprogramu), dotychczas zawierający jedynie nazwę, można rozszerzyć o opcjonalną listę zmiennych, którym zostaną nadane wartości podczas wywołania. Listę taką podajemy w nawiasach po nazwie programu, oddzielając przecinkami poszczególne zmienne. Wywołanie podprogramu następuje przez podanie jego nazwy i określenie wartości zmiennych przekazywanych jako parametry. Po wywołaniu rozpoczyna się wykonywanie podprogramu. Kończąc podprogram, jest możliwe określenie zwracanej wartości za pomocą słowa kluczowego `zwróć`.

Ostatnie przykłady przedstawiały programy zawierające pętle. Programy tego typu nazywamy *iteracyjnymi*. Odmiennym sposobem konstrukcji programów wielokrotnie powtarzających te same operacje jest wykorzystanie *rekurencji*. Procedurę (podprogram) nazywamy rekurencyjną, gdy pośrednio bądź bezpośrednio wywołuje samą siebie [28]. Rekurencja niekiedy upraszcza budowę programów i sprawia, że są bardziej czytelne, aczkolwiek problemy rozwiązywane rekurencyjnie można także rozwiązać iteracyjnie.

Sposób tworzenia programów rekurencyjnych zostanie przedstawiony na przykładzie programu obliczającego silnię. Iteracyjnie silnią pewnej liczby całkowitej dodatniej n nazywamy iloczyn wszystkich liczb całkowitych dodatnich nie większych niż n. Definiując silnię w sposób rekurencyjny, wystarczy zauważyć, że dla dowolnej liczby całkowitej dodatniej n jej silnia jest iloczynem tej liczby z silnią z liczby $n - 1$. Silnia z zera ma wartość jeden. Opierając się na tych spostrzeżeniach, można sformułować następujący wzór rekurencyjny:

$$n! = \begin{cases} 1 & \text{dla } n = 0 \\ n(n-1)! & \text{dla } n \in \mathbb{N}_+ \end{cases} \tag{4.1}$$

Opierając się na wzorze (4.1), zostanie napisana rekurencyjna procedura obliczająca silnię z liczby przekazanej w wywołaniu. Podstawowa konstrukcja najprostszych procedur rekurencyjnych opiera się na pojedynczej instrukcji warunkowej `jeśli`. W warunku sprawdzamy, czy należy zakończyć wywołania rekurencyjne, jeżeli tak, to zwracamy konkretną liczbę, w przeciwnym wypadku zwracamy wartość uzależnioną od rekurencyjnego wywołania tej samej procedury z innym parametrem. Podobnie wygląda procedura przedstawiona na listingu 4.5.

Listing 4.5

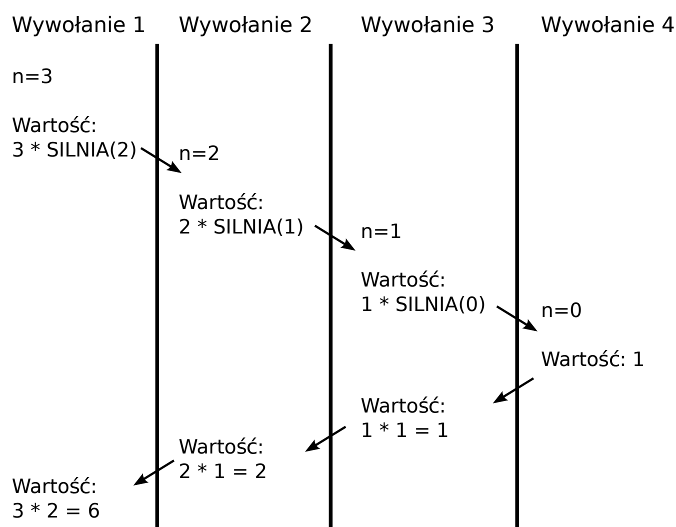
```
SILNIA(n)
1  jeśli n = 0
2  to zwróć 1
3  inaczej zwróć n * SILNIA(n-1)
```

Analiza działania przedstawionej procedury zostanie przeprowadzona na przykładzie. Rysunek 4.10 ukazuje wartości zmiennych w kolejnych wywołaniach rekurencyjnych podczas obliczania silni z liczby 3.

Oczywiście rekurencyjna konstrukcja procedury obliczającej silnię jest tylko jedną z możliwości. Czytelnikowi do samodzielnego napisania pozostawiono iteracyjną wersję tej procedury.

4.3. Elementarne struktury danych

W przedstawionych poprzednio algorytmach do przechowywania niewielkiej liczby danych wykorzystywano pojedyncze zmienne. Często jednak zachodzi konieczność skorzystania ze struktur pozwalających przechowywać dane w dogodny dla rozwiązania danego problemu sposób. Taką podstawową strukturą jest *tablica* występująca w prawie wszystkich językach programowania. Jednowymiarowa k-elementowa tablica składa się z k komórek, w których są przechowywane pojedyncze zmienne. Dostęp do poszczególnych komórek odbywa się poprzez podanie indeksu (numeru) żądanej komórki, na ogół w nawiasach kwadratowych



Rys. 4.10. Przykład działania procedury rekurencyjnej

po nazwie tablicy. W zależności od użytego języka programowania tablice są indeksowane od zera, jedności bądź wskazanej liczby. Tablice wielowymiarowe można traktować jako tablice tablic, tj. elementami tablicy n-wymiarowej są tablice $(n-1)$-wymiarowe.

Przykład graficznej reprezentacji jednowymiarowej siedmioelementowej tablicy zawierającej kolejne liczby nieparzyste pokazano na rys. 4.11. Przyjęto, że tablica ta jest indeksowana od jedynki, a indeksy poszczególnych komórek uwidoczniono pod nimi.

1	3	5	7	9	11	13
1	2	3	4	5	6	7

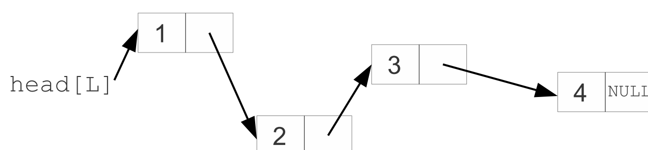
Rys. 4.11. Przykład graficznej reprezentacji tablicy

Inną często stosowaną strukturą danych jest *lista*. W wielu współczesnych językach programowania lista jest jedną z wbudowanych struktur, w innych zaś listę można zaimplementować w sposób programowy, na przykład przy użyciu wskaźników, czyli zmiennych określających lokalizację innych elementów w pamięci. W drugim przypadku typowo pojedynczy element listy składa się z pola (bądź pól) klucza służącego do przechowywania pewnych danych oraz pola wskaźnikowego zawierającego adres kolejnego elementu listy (lub wartość `NULL` w ostatnim elemencie). Opcjonalnie może występować także drugie pole wskaźnikowe pokazujące na poprzedni element listy (*lista dwukierunkowa*). Konieczne jest także

zapamiętanie wskaźnika na początkowy element listy (atrybut `head`). Zauważmy, że w strukturze takiej dostęp do poszczególnych elementów w odróżnieniu od tablicy odbywa się sekwencyjnie. Pierwszy element listy zawiera wskaźnik na drugi element, w drugim znajdziemy wskaźnik na trzeci itd. Znalezienie ostatniego elementu wymaga przejścia przez całą listę. Wygodne jest jednak wstawianie nowego elementu w dowolnym miejscu, jak również usuwanie poszczególnych elementów. W przypadku takich operacji na liście wystarcza modyfikacja kilku pól wskaźnikowych w elemencie docelowym oraz elementach sąsiednich, nie jest konieczne przesuwanie pozostałych elementów, tak jak w przypadku tablicy.

Listę można zdefiniować także w sposób rekurencyjny jako strukturę określoną na pewnym skończonym zbiorze elementów, która nie zawiera żadnych elementów (lista pusta) lub stanowi połączenie elementu i listy [28].

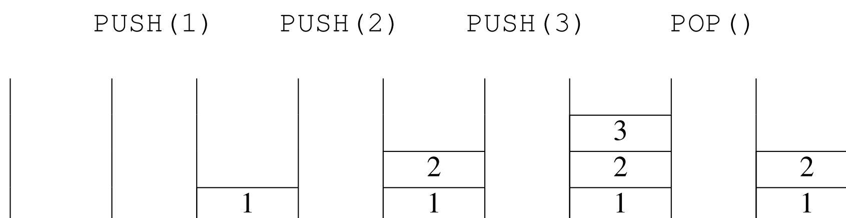
Na rysunku 4.12 pokazano przykład graficznej ilustracji czteroelementowej listy jednokierunkowej zawierającej kolejne liczby całkowite dodatnie.



Rys. 4.12. Przykład listy jednokierunkowej

W pewnych zastosowaniach korzystne jest użycie struktur pozwalających w naturalny sposób na chwilowe zapamiętanie napływających danych, a następnie ich obsłużenie w odpowiednim porządku. Przykładem takiej struktury jest *stos* buforujący dane w celu obsłużenia ich w porządku LIFO (*Last In First Out*), tj. jako pierwszy będzie obsłużony element, który został jako ostatni odłożony na stos. Stos jest często wykorzystywany w informatyce. Wiele mikroprocesorów ma sprzętowy stos umożliwiający np. tymczasowe przechowanie wartości rejestrów podczas wywołania procedury, często stos jest także wewnętrznie implementowany programowo przez kompilatory języków wysokiego poziomu. Intuicyjnie stos w sensie programistycznym można porównać do stosu przedmiotów ułożonych jeden na drugim. W danej chwili mamy dostęp jedynie do przedmiotu znajdującego się na górze (wierzchołek stosu), dostęp do kolejnych elementów wymaga uprzedniego zdjęcia przedmiotów poprzednich. Oznacza to, że na stosie możemy wykonać jedynie operację odłożenia nowego elementu na górę stosu (polecenie `PUSH`) bądź zdjęcia elementu z wierzchołka (`POP`). Przykłady graficznych ilustracji stosu, na który zostały odłożone liczby 1, 2, 3 poprzez kolejne instrukcje `PUSH`, a następnie zdjęta ostatnia liczba operacją `POP`, pokazano na rys. 4.13.

Jak wspomniano, stos pełni rolę bufora typu LIFO. Niekiedy zachodzi konieczność skorzystania z bufora typu FIFO (*First In First Out*) pozwalającego na obsłużenie buforowanych danych w kolejności ich napływania. Taki bufor na-



Rys. 4.13. Przykładowe operacje na stosie

zywamy *kolejką FIFO*. Nowe dane są umieszczane na końcu kolejki (operacja `ENQUEUE`), a dane do przetworzenia pobierane z końca kolejki (`DEQUEUE`).

Opisane struktury danych można implementować w różny sposób, także za pomocą innych struktur, co opisano np. w pracy [28]. Przykładowo, stos może być łatwo implementowany jako tablica zawierająca w kolejnych komórkach kolejne elementy stosu. W zmiennej pomocniczej jest przechowywany indeks ostatniego położonego elementu (wierzchołek stosu). Dodanie elementu (`PUSH`) wymaga inkrementacji tej zmiennej i umieszczenia wstawianego elementu w komórce tablicy indeksowanej przez wierzchołek. Podczas pobierania elementu (`POP`) sprawdzamy, czy stos nie jest pusty, a jeżeli nie, to zapamiętujemy do zwrócenia ostatni element i dekrementujemy indeks wierzchołka. Napisanie takiej implementacji w pseudokodzie pozostawiono Czytelnikowi do samodzielnego wykonania.

Stos można zaimplementować także za pomocą listy. W przypadku reprezentacji listowej nowe elementy najwygodniej jest wstawiać na początek listy. W takim rozwiązaniu zachowanie porządku LIFO pociąga za sobą zdejmowanie elementów również od pierwszego na liście. Alternatywnie można rozważać reprezentację, w której nowe elementy są umieszczane na końcu listy, ale wymaga to czasochłonnego przechodzenia przez całą listę do ostatniego elementu, zarówno podczas operacji `PUSH`, jak i `POP`.

Kolejka również może być wygodnie implementowana za pomocą tablicy lub listy. Należy pamiętać, że reprezentacja listowa kolejki w odróżnieniu od stosu wymaga, aby miejsca wstawiania i pobierania danych były po przeciwnych stronach listy, tak by zachować porządek FIFO. Oznacza to, że jeżeli przyjmiemy, że operacja `ENQUEUE` umieszcza nowy element na początku listy, to w implementacji procedury `DEQUEUE` należy przejść do ostatniego elementu i ten element pobrać. Oczywiście alternatywnie `ENQUEUE` może wstawiać elementy na końcu listy, i wówczas `DEQUEUE` pobiera pierwszy element na liście.

4.4. Złożoność obliczeniowa

Niejednokrotnie zadanie można rozwiązać na różne sposoby, wykorzystując odmienne algorytmy. W celu porównania efektywności algorytmów celowe jest określanie ich *złożoności obliczeniowej*. Zaawansowane aspekty złożoności algorytmów, takie jak formalna definicja notacji asymptotycznej, równomierne i logarytmiczne kryterium wagowe, jak też różne metody szacowania złożoności wykraczają poza zakres niniejszego skryptu i nie będą tu omawiane. Informacje te można znaleźć np. w publikacjach [3, 28]. W pracy zostaną przedstawione jedynie najistotniejsze kwestie i nieformalne definicje związane z zagadnieniem złożoności obliczeniowej.

Złożoność algorytmu analizujemy w odniesieniu do pewnego rozmiaru danych wejściowych, zwanego *rozmiarem zadania*. W zależności od algorytmu rozmiarem zadania może być np. liczba wierzchołków grafu w algorytmach działających na grafach czy też rozmiar tablicy wejściowej w algorytmach sortowania. *Złożonością czasową* algorytmu nazywamy czas potrzebny na jego wykonanie jako funkcję rozmiaru zadania [28]. Analogicznie, rozpatrując *złożoność pamięciową* algorytmu, analizujemy, jak zmienia się zapotrzebowanie na pamięć wraz ze wzrostem rozmiaru zadania. Rozważając charakter funkcji złożoności przy dążeniu do wartości granicznej, określamy *asymptotyczną złożoność*, odpowiednio czasową bądź pamięciową.

Przeanalizujmy wybrane algorytmy sortowania. Formalna definicja sortowania wymagałaby wprowadzenia *relacji częściowego porządku* i pojęcia *porządku liniowego*. Omówienie tych zagadnień wykracza poza zakres niniejszego opracowania, można je znaleźć np. w [24, 28]. W niniejszej pracy sortowanie będziemy rozumieć intuicyjnie jako zmianę kolejności elementów pewnego ciągu, tak aby ułożyć je niemalejąco bądź nierosnąco, czyli aby każdy element za wyjątkiem ostatniego był odpowiednio mniejszy bądź równy albo większy bądź równy kolejnemu.

Jako pierwszy zostanie przeanalizowany algorytm sortowania przez wybieranie (selekcję). W algorytmie tym, sortując rosnąco, znajdujemy najmniejszy element w tablicy i umieszczamy go na pierwszej pozycji, zamieniając z tym, który się tam aktualnie znajduje. W kolejnym kroku znajdujemy drugi najmniejszy element spośród pozostałych i zamieniamy go z elementem z drugiej pozycji. Postępujemy analogicznie dla wszystkich kolejnych elementów tablicy. Przykładowy program sortujący według tego algorytmu zamieszczono na listingu 4.6. Przyjęto tu, że tablica A, zawierająca elementy do posortowania, jest przekazywana jako parametr, atrybut `length[A]` oznacza zaś rozmiar tablicy.

Przypuśćmy, że sortowaniu podlega tablica A zawierająca elementy $A = (7, 5, 1, 4, 2, 3, 8, 6)$. Kolejne kroki sortowania pokazano na rys. 4.14.

Listing 4.6

SORTOWANIE_SELEKCJA(A)
```
 1  dla i = 1 do length[A] - 1 wykonuj
 2     {
 3     min = i
 4     dla j = i + 1 do length[A] wykonuj
 5        {
 6        jeśli A[min] > A[j] to min = j
 7        }
 8     jeśli min ≠ i to
 9        {
10        temp = A[i]
11        A[i] = A[min]
12        A[min] = temp
13        }
14     }
```

Numer kroku	Tablica							
	7	5	1	4	2	3	8	6
1	**1**	5	**7**	4	2	3	8	6
2	1	**2**	7	4	**5**	3	8	6
3	1	2	**3**	4	5	**7**	8	6
4	1	2	3	**4**	5	7	8	6
5	1	2	3	4	**5**	7	8	6
6	1	2	3	4	5	**6**	8	**7**
7	1	2	3	4	5	6	**7**	**8**

Rys. 4.14. Przykład działania algorytmu sortowania przez selekcję

Rozważmy liczbę porównań koniecznych do wykonania przy sortowaniu n-elementowej tablicy jako najczęściej wykonywanych operacji w algorytmie (*operacja dominująca*). W i-tym kroku znajdujemy najmniejszy element w podtablicy liczącej $n - i + 1$ elementów. Operacja taka wymaga $n - i$ porównań w każdym kroku, musimy wykonać $n - 1$ kroków. W każdym kolejnym kroku wykonujemy o jedno porównanie mniej niż w kroku poprzednim, liczba porównań w poszczególnych krokach tworzy więc ciąg arytmetyczny. Ostatecznie liczba porównań do wykonania wyraża się wzorem (4.2).

$$\sum_{i=1}^{n-1}(n-i) = \frac{1}{2}n^2 - \frac{1}{2}n \qquad (4.2)$$

Przyjmując, że wykonanie każdego z porównań trwa równie długo i pomijając czas potrzebny na wykonanie pozostałych operacji w programie, można przyjąć, że czas sortowania tablicy przedstawionym algorytmem można z góry ograniczyć pewną funkcją kwadratową uzależnioną od rozmiaru zadania. Taki charakter złożoności zapisujemy jako $\mathcal{O}(n^2)$. Formalna definicja notacji $\mathcal{O}$ jest następująca [21]:

$$f(n) = \mathcal{O}\left(g(n)\right) \overset{def.}{\Longleftrightarrow} \bigvee_{\substack{c>0 \\ n_0>0}} \bigwedge_{n \geqslant n_0} f(n) \leqslant c \cdot g(n) \tag{4.3}$$

Analogicznie do notacji $\mathcal{O}$ służącej do wskazania funkcji będącej ograniczeniem górnym można zdefiniować notację Ω, która określa ograniczenie dolne.

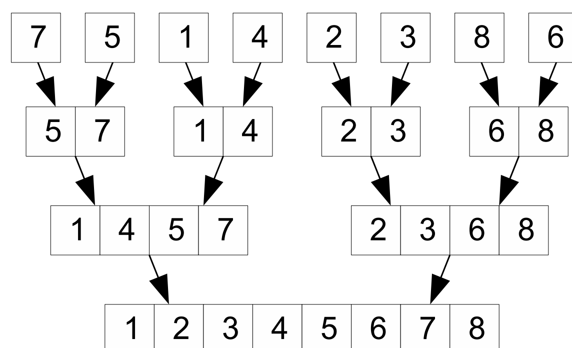
$$f(n) = \Omega\left(g(n)\right) \overset{def.}{\Longleftrightarrow} \bigvee_{\substack{c>0 \\ n_0>0}} \bigwedge_{n \geqslant n_0} c \cdot g(n) \leqslant f(n) \tag{4.4}$$

Połączeniem notacji $\mathcal{O}$ i Ω jest notacja Θ. Jeżeli funkcję $f(n)$ możemy ograniczyć od pewnego n_0 z góry i z dołu za pomocą tej samej funkcji $g(n)$, ale przemnożonej przez różne stałe c_1 i c_2, to mówimy, że „$f(n)$ jest theta od $g(n)$”. Formalnie definiujemy to następująco:

$$f(n) = \Theta\left(g(n)\right) \overset{def.}{\Longleftrightarrow} \bigvee_{\substack{c_1>0 \\ c_2>0 \\ n_0>0}} \bigwedge_{n \geqslant n_0} c_1 \cdot g(n) \leqslant f(n) \leqslant c_2 \cdot g(n) \tag{4.5}$$

Jako kolejny algorytm sortowania rozważmy sortowanie przez scalanie. W algorytmie tym tablicę do posortowania dzielimy początkowo na pojedyncze elementy, które mogą być traktowane jako jednoelementowe podtablice. W kolejnych krokach scalamy sąsiednie podtablice, tak aby nadal były posortowane. Przebieg sortowania dla użytej w poprzednim przykładzie tablicy zawierającej elementy 7, 5, 1, 4, 2, 3, 8, 6 pokazano na rys. 4.15. Przykładową implementację tego algorytmu w pseudokodzie można znaleźć np. w pracy [28].

W pierwszym kroku każdy element tablicy traktujemy jako osobną, jednoelementową podtablicę. Oczywiste jest, że jednoelementowa tablica jest posortowana. W drugim kroku podtablice te łączymy w tablice dwuelementowe. W kolejnych krokach kontynuujemy to postępowanie, scalając podtablice w tablice cztero-, następnie ośmioelementowe, aż do otrzymania tablicy wynikowej. Zauważmy, że dzięki posortowaniu podtablic przy każdym scaleniu wystarcza nam pojedyncze porównanie, aby określić położenie docelowe jednego elementu. Przykładowo, w ostatnim kroku w przykładzie z rys. 4.15 porównanie wartości 1 z pierwszej podtablicy z wartością 2 z drugiej wystarcza, aby stwierdzić, że



Rys. 4.15. Kolejne kroki sortowania przez scalanie

jedynka jest najmniejszą wartością z obu podtablic i powinna trafić na pierwsze miejsce w tablicy wynikowej. Następnie porównamy liczbę 4 z pierwszej tablicy z liczbą 2 z drugiej i na tej podstawie wartość 2 trafi do drugiej komórki w tablicy docelowej. Kolejne porównanie nastąpi między elementami 4 a 3. Łatwo zauważyć, że scalając dwie podtablice o łącznej liczbie n elementów, musimy wykonać co najwyżej $n - 1$ porównań. Z kolei liczba kroków potrzebna do posortowania całej tablicy jest proporcjonalna do logarytmu o podstawie dwóch z liczby elementów (w pierwszym kroku scalamy tablice jednoelementowe, w drugim dwuelementowe, w trzecim czteroelementowe, w n-tym 2^{n-1}-elementowe). Można więc stwierdzić, że ograniczenie górne sumarycznej liczby porównań potrzebnej do posortowania n-elementowej tablicy przez scalanie wynosi $\mathcal{O}(n \lg n)$.

4.5. Zadania

1. Zaprojektuj program rozwiązujący równania kwadratowe o współczynnikach podawanych z klawiatury. Rozpatrz wszystkie przypadki. Przedstaw go w postaci:

   - schematu blokowego,
   - pseudokodu.

2. Zaprojektuj program wczytujący liczby do momentu wczytania wartości 7. Po wczytaniu symbolu 7 program powinien wyświetlić sumę wczytanych liczb i zakończyć pracę. Liczba 7 jest tu traktowana jedynie jako symbol końcowy i nie powinna być ujęta w końcowej sumie. Przedstaw program w postaci:

   - schematu blokowego,

- pseudokodu.

3. Zaprojektuj program wczytujący liczby do momentu wczytania zera. Po wczytaniu tego symbolu terminalnego program powinien wyświetlić najmniejszą z wczytanych liczb (nie uwzględniając końcowego zera) i zakończyć działanie. Przedstaw program w postaci:
   - schematu blokowego,
   - pseudokodu.
4. Zaprojektuj program wczytujący liczby do momentu wczytania zera. Po wczytaniu tego symbolu końcowego program powinien wyświetlić największą z wczytanych liczb ujemnych i zakończyć działanie. Analizując działanie programu, rozważ także przypadek niewczytania żadnej liczby ujemnej. Przedstaw program w postaci:
   - schematu blokowego,
   - pseudokodu.
5. Napisz w pseudokodzie program znajdujący największy wspólny dzielnik dwóch liczb naturalnych za pomocą algorytmu Euklidesa.
6. Napisz w pseudokodzie iteracyjną procedurę obliczającą silnię z liczby naturalnej przekazanej jako parametr.
7. Wiedząc, że trójkąt nie ma przekątnych, a każdy n-kąt ($n \geqslant 4$) ma o $n - 2$ przekątnych więcej niż ($n - 1$)-kąt, napisz:
   - wzór rekurencyjny na liczbę przekątnych wielokąta,
   - rekurencyjny program w pseudokodzie obliczający liczbę przekątnych wielokąta.
8. Zapisz w pseudokodzie procedury `PUSH` i `POP` implementujące stos za pomocą tablicy.
9. Przedstaw kolejne kroki sortowania dla tablicy zawierającej elementy 15, 7, 10, 1, 8, 13, 3, 6 według algorytmu:
   - sortowania przez selekcję,
   - sortowania przez scalanie.

Rozdział 5.

Język Ruby i paradygmaty programowania

Wojciech Rząsa

5.1. Wprowadzenie

Paradygmaty programowania definiują sposób, w jaki tworzony jest program komputerowy. Zasadniczo dotyczą one języków programowania wyższego poziomu, w których programy nie są ciągiem prostych instrukcji wykonywanych jedna za drugą, ale mają określoną strukturę. Programiści tych języków powinni stosować się do pewnych reguł pisania programów, które mają być wydajne, przejrzyste, nie będą wymagać nadmiaru pracy. Reguły te mogą być do pewnego stopnia wymuszane przez używany język programowania, ale przeważnie nie są. Często też wybór zestawu przyjętych reguł jest decyzją programisty bądź zespołu, w którym pracuje i zestaw taki jest dostosowany do specyficznych wymagań tworzonego projektu.

Przekonanie o tym, jakie paradygmaty programowania są właściwe i jakie powinny być stosowane w różnych przypadkach zmieniało się z czasem. Istotny wpływ na to miał rozwój języków programowania, które wspierały bądź nie różne paradygmaty. Zdaniem B. Stroustupa, twórcy języka C++, można mówić, że język programowania *wspiera* pewien paradygmat programowania, jeśli dostarcza mechanizmów, które pozwalają w wygodny sposób tworzyć programy zgodnie z danym paradygmatem. Można też mówić, że język *umożliwia* (ale nie wspiera) stosowanie określonego paradygmatu, jeśli pisanie w tym języku programów zgodnie z tym paradygmatem jest możliwe, ale niewdzięczne, trudne, czy też proces programowania jest podatny na błędy (por. [27]).

Celem tego rozdziału jest opisanie najistotniejszych paradygmatów programowania, pokazanie ich cech charakterystycznych, wad i zalet w sposób możliwie zrozumiały. Przedstawienie konkretnych przykładów dobrego i złego stylu programowania, wykorzystania różnych paradygmatów, a także różnic pomiędzy nimi wymaga zastosowania języka bądź języków programowania. Nie jest to ła-

twe zadanie. Z jednej strony nie jest możliwe pokazanie przykładów wszystkich paradygmatów programowania za pomocą jednego tylko języka, gdyż żaden język nie wspiera, a nawet nie umożliwia stosowania wszystkich paradygmatów. Z drugiej jednak strony trudno wymagać od Czytelnika zapoznania się z nawet tylko kilkoma językami sztandarowymi dla określonych paradygmatów tylko po to, by pokazać przykłady i różnice. Kompromisem może być zaprezentowanie przykładów w tzw. pseudokodzie – języku, który tylko przypomina język programowania. Zdaniem autora podejście to ma jednak wszystkie wady poprzednich rozwiązań, a dodatkowo nie daje Czytelnikowi nawet jednego użytecznego narzędzia programowania, nie pozwalając mu także sprawdzić w praktyce, jak działają przedstawione przykłady.

W rozdziale tym zdecydowano się więc wybrać jeden rzeczywisty język programowania, który wspiera możliwie wiele paradygmatów. Do tego celu wybrano język Ruby (czyt. [ru:bi], po ang. rubin), którego podstawy i kolejne elementy opisano w dalszej części rozdziału. Zasadniczą część przykładów pokazano więc w języku Ruby i może być przez Czytelnika sprawdzona za pomocą interpretera tego języka. W nielicznych przypadkach, tam gdzie będzie to konieczne, przykłady będą zawierały elementy, których nie ma w języku Ruby, a które są dodane, aby zilustrować pewne paradygmaty programowania bądź różnice pomiędzy nimi. Fakt „dodania" czegoś do języka zostanie wyraźnie zaznaczony. W niewielu przypadkach zostaną zamieszczone także proste przykłady w innych językach. Będą one szczegółowo opisane, tak aby nie utrudniały zrozumienia istoty.

Język Ruby rozwija się bardzo dynamicznie. W tym opracowaniu korzystano z aktualnie zalecanej wersji tego języka – 2.0.0.

5.2. Instalacja i korzystanie z języka

Interpreter języka Ruby jest dostępny w najbardziej znanych dystrybucjach Linuksa. Można go także pobrać ze strony domowej[1], jednak zalecanym i bardzo wygodnym sposobem instalacji jest wykorzystanie jednego z narzędzi `rbenv`[2] bądź `rvm`[3]. Instalator interpretera języka Ruby przeznaczony dla Windows nazywa się `RubyInstaller`[4].

Program należy zapisać w pliku z rozszerzeniem `.rb` i uruchomić komendą `ruby nazwa_pliku.rb`. Poza tym w każdej instalacji jest dostępny także interaktywny interpreter języka – `irb`, pozwalający wykonywać na bieżąco wpisy-

[1] http://www.ruby-lang.org/pl/downloads/
[2] https://github.com/sstephenson/rbenv
[3] https://rvm.io/
[4] http://rubyinstaller.org/

wane polecenia. Jest to bardzo wygodne narzędzie podczas nauki, do testowania prostych wyrażeń czy bardzo prostych programów.

Zakładamy, że Czytelnik będzie na bieżąco korzystał z zainstalowanego interpretera języka, aby sprawdzić, jak działają programy przedstawione na listingach oraz ich własne modyfikacje. Tylko w ten sposób jest możliwe przyswojenie wiedzy i umiejętności przekazywanych w tym rozdziale.

5.3. Język Ruby – podstawy

5.3.1. O języku

Podrozdział ten opisujący podstawy języka Ruby jest bazą wykorzystywaną w kolejnej części rozdziału. Opis ten nie jest kompletny, a kolejne elementy języka będą wprowadzane w miarę potrzeby w następnych podrozdziałach wraz z opisem poszczególnych paradygmatów programowania.

Język Ruby powstał w 1995 r. Stworzył go Yukihiro Matsumoto znany pod pseudonimem Matz. Na temat swojej motywacji twórca języka pisze: *Przed powstaniem języka Ruby znałem wiele języków, ale żaden z nich nie satysfakcjonował mnie w pełni. [...] Postanowiłem stworzyć język programowania, który zaspokoiłby moje potrzeby programistyczne. [...] Ku mojemu zaskoczeniu okazało się, że wielu programistów na cały świecie ma odczucia podobne do moich*[5]. W efekcie powstał język, który *większość programistów uważa, za elegancki i łatwy w użyciu, a pisanie w nim programów to sama przyjemność*[6]. Ponieważ składnia języka Ruby nie jest skomplikowana, nie zawiera zbędnych elementów język ten doskonale nadaje się do nauki programowania, a także do ilustrowania przykładowych algorytmów czy technik programistycznych. Spośród różnych możliwych do wybrania języków programowania Ruby powinien sprawić Czytelnikowi najmniej kłopotów, stanowiąc jednocześnie wygodne narzędzie, możliwe do użycia w przyszłości.

Pomimo swojej zewnętrznej prostoty język ten oferuje bardzo wiele także zaawansowanym programistom. Strona domowa języka Ruby cytuje następującą wypowiedź Matza: *Ruby jest prosty z wyglądu, ale bardzo skomplikowany w środku, tak jak ciało ludzkie*[7]. Nie jest to więc język mający jedynie wartość dydaktyczną, który po nauczeniu się programowania należy porzucić, ale narzędzie dające bardzo duże możliwości zaawansowanym programistom.

Poza wspomnianą już, wydaną w języku polskim książką o programowaniu w języku Ruby, której współautorem jest Matz [9], doskonały opis tego języka

[5] Falangan D., Matsumoto Y., *Ruby. Programowanie*. Helion, Gliwice 2009.

[6] Tamże.

[7] `http://www.ruby-lang.org`

prezentuje także pozycja [5]. Warto również zajrzeć do książki *Learn Ruby The Hard Way*[8] przeznaczonej dla tych, którzy nie mają doświadczenia w programowaniu w innych językach. Te ostatnie pozycje nie są jednak obecnie dostępne w języku polskim.

Można korzystać także z serwisów internetowych, jak RubyMonk[9], CodeSchool[10] czy CodeAcademy[11], pozwalających na naukę języka Ruby za pomocą interaktywnego kursu, bez konieczności instalowania jakichkolwiek narzędzi. Warto także zajrzeć do serwisu iwanttolearnruby.com[12], w którym można odnaleźć zebrane informacje o serwisach pomagających w nauce tego języka.

5.3.2. Hello world!

Ponieważ tradycyjnie już opisy języków programowania rozpoczyna się od przykładu programu, który wypisuje na ekranie tekst *Hello world*, także w tym opracowaniu rozpoczniemy od prezentacji tego przykładu. *Hello world* w języku Ruby jest bardzo prostym programem, co prezentuje listing 5.1.

Listing 5.1

```
1  puts "Hello world"
```

Polecenie `puts` służy do wypisywania tekstu na ekranie. Tekst w języku Ruby można zawrzeć w pojedynczych albo podwójnych cudzysłowach: `"przykładowy tekst"` albo `'przykładowy tekst'`.

W języku Ruby tekst na ekraine można wypisywać także za pomocą polecenia `print`. Różnica polega na tym, że polecenie `puts` w przeciwieństwie do `print` dodaje na końcu wypisywanego tekstu znak nowej linii (jeśli takiego znaku tam nie było). Przykładowy program może więc wyglądać też tak, jak to przedstawiono na listingu 5.2. Wtedy jednak po zakończeniu programu kursor zatrzyma się zaraz za słowem `world`, a nie w kolejnej linii.

Listing 5.2

```
1  print "Hello world"
```

Znaki nowej linii (i inne tzw. znaki specjalne) można wstawiać w języku Ruby w dowolnym miejscu tekstu za pomocą dwóch znaków: znaku \ czyli *backslash* (czyt. [bekslesz]) oraz dodatkowego znaku określającego, jaki znak specjalny ma być użyty. Przykładowo, aby w danym miejscu tekstu wstawić znak nowej linii,

[8] Książka jest dostępna za darmo do czytania on-line pod adresem: `http://ruby.learncodethehardway.org/book/`
[9] `http://rubymonk.com/`
[10] `http://www.codeschool.com/paths/ruby`
[11] `http://www.codecademy.com/tracks/ruby`
[12] `http://iwanttolearnruby.com/`

trzeba w tym miejscu umieścić sekwencję `\n`. Aby program napisany z użyciem polecenia `print` działał tak samo, jak ten napisany za pomocą `puts`, należy go poprawić tak jak na listingu 5.3.

Listing 5.3

```
1  print "Hello world\n"
```

Jeśli natomiast chcielibyśmy, żeby każde ze słów znajdowało się w osobnej linii, można napisać to w sposób przedstawiony na listingu 5.4 albo 5.5.

Listing 5.4

```
1  print "Hello\nworld\n"
```

Listing 5.5

```
1  puts "Hello\nworld"
```

W języku Ruby, jak w każdym języku programowania, do programu można dodawać komentarze – wyjaśnienia dla czytającego kod, co dany fragment robi i co miał na myśli programista, gdy go pisał. Komentarz w języku Ruby zaczyna się od znaku `#` – wszystko co zostanie napisane po tym znaku aż do końca linii jest ignorowane przez interpreter języka. Komentarz w przykładowym programie mógłby wyglądać tak, jak przedstawia to listing 5.6.

Listing 5.6

```
1  # wypisze tekst na ekranie
2  puts "Hello world" # to jest wypisywanie
```

Wieloliniowy komentarz wykonuje się za pomocą `=begin` i `=end`, co prezentuje listing 5.7.

Listing 5.7

```
1  =begin
2  To jest komentarz w języku Ruby,
3  zajmujący więcej niż jedną linię
4  =end
5  puts "Hello world" # to jest wypisywanie
```

5.3.3. Ruby i język polski

Język Ruby obsługuje kodowanie znaków w językach narodowych, zarówno w przetwarzanych w programach tekstach, jak i w samym kodzie źródłowym. W wersji 2.0 interpreter języka zakłada, że program będzie kodowany w standardzie UTF-8 i poprawnie obsługuje programy zawierające znaki narodowe kodowane w tym standardzie. Program wypisujący w języku polskim wraz ze znakami narodowymi komunikat z rozdziału 5.3.2 może wyglądać tak jak na listingu 5.8.

Listing 5.8
```
1  puts "Witaj świecie!"
```

5.3.4. Zmienne

Programy komputerowe służą do przetwarzania danych. Aby było to możliwe, dane te muszą (przynajmniej przez jakiś czas) być przechowywane w pamięci. Do zapisania danych w pamięci albo ich odczytania służą zmienne. W języku Ruby zmienne stanowią etykiety wskazujące miejsce, gdzie wcześniej w pamięci została zapisana jakaś wartość.

W języku Ruby zmiennych nie trzeba deklarować. Pojawiają się one w programie, gdy po raz pierwszy zostanie do nich przypisana wartość. Zmienne nie mają też określonych typów. Można stwierdzić, że w danym momencie do pewnej zmiennej jest przypisana *wartość* określonego typu (np. liczba albo tekst). Później jednak do tej samej zmiennej można przypisać wartość innego typu. Jest to wygodna cecha języka pozwalająca na dużą elastyczność podczas programowania, wymaga jednak pewnej uwagi, ponieważ może być przyczyną błędów w programach.

Listing 5.9 przedstawia program, w którym tekst powitania został najpierw przypisany do zmiennej `powitanie`, a dopiero potem wyświetlony.

Czasami zachodzi konieczność zaznaczenia, że zmienna nie zawiera żadnej wartości: *nic* nie przekazano, *nic* nie należy zapamiętywać; *nic* w języku Ruby jest oznaczane poprzez słowo `nil`. Jeśli więc zmienna nie powinna zawierać jakiejkolwiek wartości, należy jej przypisać wartość `nil`, np. `srednia = nil`.

Listing 5.9
```
1  powitanie = "Cześć, jak Ci się to podoba?"
2
3  puts powitanie
```

Interpolacja w tekście

Zmienne można umieszczać w tekście (np. aby wypisać je na ekranie). Operacja taka nazywa się *interpolacją* zmiennej w tekście. W języku Ruby interpolacja wymaga zastosowania specjalnej sekwencji znaków: `#{zmienna}`. Działa ona tylko dla tekstów zapisanych w podwójnym cudzysłowie (`"`), dla tekstów zaś w pojedynczym cudzysłowie (`'`) interpolacja nie jest wykonywana. Przykład wykorzystania interpolacji pokazuje listing 5.10. Zmienna `imie` została wstawiona do tekstu, który został następnie umieszczony w zmiennej `powitanie`.

Interpolację można także wykonać wprost w tekście podawanym do wypisania poleceniu `puts` (listing 5.11).

Listing 5.10

```
1 imie = "Janek"
2 powitanie = "Cześć #{imie}, jak Ci się to podoba?"
3
4 puts powitanie
```

Listing 5.11

```
1 imie = "Janek"
2 puts "Cześć #{imie}, jak Ci się to podoba?"
```

Wczytywanie do zmiennych

Dane, na których operuje program, mogą być oczywiście wczytywane „z zewnątrz” przez ten program, np. mogą być podawane przez użytkownika z klawiatury. Wczytywanie danych z klawiatury wykonuje się w języku Ruby za pomocą polecenia `gets`. Aby najpierw wczytać imię, a potem użyć go w wypisywanym tekście, trzeba użyć polecenia `gets`; to co „zwróci” polecenie należy przypisać do jakiejś zmiennej (zapamiętać do późniejszego użycia). Potem wartości tej zmiennej można użyć w tekście (np. tak jak to przedstawiają listingi 5.10 czy 5.11). Przykład programu wczytującego imię z klawiatury przedstawia listing 5.12.

Listing 5.12

```
1 puts "Jak masz na imię?"
2 imie = gets
3 puts "Cześć #{imie}, jak Ci się to podoba?"
```

W programie tym znajduje się jednak jeden błąd. Mianowicie do zmiennej zostanie wczytane wszystko to, co poda użytkownik, aż do naciśnięcia klawisza Enter. Znak nowej linii (skutek naciśnięcia Enter) również znajdzie się na końcu tekstu zapisanego w zmiennej. Wynik działania programu będzie więc następujący (zakładając, że użytkownik zapytany o imię wpisze Wojtek).

```
Jak masz na imię?
Wojtek
Cześć Wojtek
, jak Ci się to podoba?
```

Jak widać, po imieniu został wypisany znak przejścia do nowej linii. Aby tego uniknąć, z wczytanego tekstu trzeba usunąć ten znak za pomocą polecenia `chomp` wywołanego na poleceniu `gets`, czyli na tym, co zwróci to polecenie (zob. listing 5.13). Po poprawce program wypisze zgodnie z oczekiwaniami:

```
Jak masz na imię?
Wojtek
Cześć Wojtek, jak Ci się to podoba?
```

Listing 5.13
```
1 puts "Jak masz na imię?"
2 imie = gets.chomp # <-- tutaj usuwam Enter poleceniem chomp
3 puts "Cześć #{imie}, jak Ci się to podoba?"
```

Wczytywanie liczb

Wartości podawane z klawiatury są wczytywane jako tekst. Jeśli zachodzi potrzeba wczytania liczb, wczytany tekst trzeba zamienić na liczbę. W języku Ruby, jak w wielu językach programowania, inaczej są traktowane liczby całkowite (zapisywane za pomocą typu o nazwie `Fixnum`), a inaczej liczby niecałkowite (zmiennoprzecinkowe, przechowywane za pomocą typu o nazwie `Float`). O typach i sposobie przechowywania liczb można przeczytać w rozdziale 3.

Jeśli wczytany tekst ma zostać zamieniony na liczbę całkowitą, należy użyć polecenia `to_i` (od ang. *to integer*). Aby wczytany tekst zamienić na liczbę zmiennoprzecinkową, należy użyć polecenia `to_f` (ang. *to float*). Dopiero po zamianie na wczytanych wartościach będzie można wykonać operacje liczbowe. Przykład użycia obu opisanych poleceń przedstawia listing 5.14.

Listing 5.14
```
1  puts "Na którym jesteś roku studiów?"
2  rok = gets.to_i # rok studiów, to liczba całkowita
3
4  do_konca = 5 - rok
5
6  puts "A jaką masz średnią?"
7  srednia = gets.to_f # średnia rzadko jest liczbą całkowitą
8
9  puts "Czy wiesz, że do końca studiów już tylko #{do_konca} lat?"
10 puts "I Twoja średnia to tylko #{srednia}!?"
```

Oczywiście, aby konwersja tekstu na liczby zadziałała poprawnie, konwertowany tekst musi zawierać poprawny zapis liczby. Należy także pamiętać, że w językach programowania do oddzielenia części całkowitej liczby od jej części ułamkowej stosuje się kropkę zamiast przecinka.

5.3.5. Instrukcje warunkowe

Pisanie nieco bardziej złożonych i bardziej użytecznych programów wymaga możliwości sprawdzania warunków i na ich podstawie podejmowania decyzji, które operacje powinny zostać przez program wykonane, a które pominięte. Służą do tego tzw. *instrukcje warunkowe*.

Warunkiem może być każde wyrażenie, którego wynik ma wartość logiczną prawdy bądź fałszu. Można sprawdzać wzajemne relacje pomiędzy wartościami

za pomocą operatorów ==, <, >, <=, =>. Operatory logiczne są zapisywane za pomocą określeń angielskich (pisanych małymi literami): `or`, `and`, `not`. W wyrażeniach logicznych można oczywiście używać nawiasów, jeśli są one potrzebne.

Składnię podstawowej wersji instrukcji warunkowej przedstawia listing 5.15. Jeśli `warunek` będzie spełniony, to zostaną wykonane instrukcje pomiędzy linią 2 a linią 4, jeśli zaś warunek nie będzie spełniony, instrukcje te zostaną pominięte.

Listing 5.15

```
1 if warunek
2   # tutaj jakieś instrukcje
3   # które zostaną wykonane
4   # gdy warunek będzie spełniony
5 end
```

Aby wykonać jeden z dwóch możliwych do wyboru fragmentów kodu (wybrać jedną z dwóch ścieżek), należy użyć nieco bardziej rozbudowanej wersji instrukcji przedstawionej na listingu 5.16. W tym wypadku, podobnie jak poprzednio, instrukcje pomiędzy linią 2 a 4 zostaną wykonane, jeśli warunek będzie spełniony. Jeśli natomiast warunek nie będzie spełniony, zostaną wykonane instrukcje pomiędzy linią 6 a 8.

Listing 5.16

```
1 if warunek
2   # tutaj jakieś instrukcje
3   # które zostaną wykonane
4   # gdy warunek będzie spełniony
5 else
6   # tutaj jakieś instrukcje
7   # które zostaną wykonane
8   # gdy warunek NIE będzie spełniony
9 end
```

Za pomocą instrukcji warunkowej można trochę rozbudować program z listingu 5.14, wypisując różnie komunikaty, zależnie od wysokości średniej (zob. listing 5.17). Oczywiście w razie potrzeby pomiędzy instrukcją `if` a `else` oraz `else` a `end` można umieścić więcej niż jedną instrukcję.

Język Ruby pozwala pisać jeszcze bardziej złożone instrukcje warunkowe za pomocą instrukcji `elsif` pokazanej na listingu 5.18. Polecenia `elsif` można używać wielokrotnie, testując wiele różnych warunków. Przykład zastosowania takiej instrukcji przedstawia listing 5.19.

Instrukcje warunkowe można zagnieżdżać jedną w drugiej, np. listing 5.20. W tym przykładzie uwagi na temat zbyt niskiej czy wysokiej średniej zostaną wypisane tylko w przypadku, gdy do końca studiów pozostanie mniej niż 3 lata.

Listing 5.17

```
1  puts "Na którym jesteś roku studiów?"
2  rok = gets.to_i # rok studiów, to liczba całkowita
3
4  do_konca = 5 - rok
5
6  puts "A jaką masz średnią?"
7  srednia = gets.to_f # średnia rzadko jest liczbą całkowitą
8
9  puts "Czy wiesz, że do końca studiów już tylko #{do_konca} lat?"
10
11 if srednia > 3.5
12   puts "A Twoja średnia to aż #{srednia}!"
13 else
14   puts "I Twoja średnia to tylko #{srednia}!?"
15 end
```

Listing 5.18

```
1  if warunek_pierwszy
2    # tutaj jakieś instrukcje
3    # które zostaną wykonane
4    # gdy warunek_pierwszy będzie spełniony
5  elsif warunek_drugi
6    # tutaj jakieś instrukcje
7    # które zostaną wykonane
8    # gdy warunek_pierwszy NIE będzie spełniony
9    # ale będzie spełniony warunek_drugi
10 elsif warunek_trzeci
11   # tutaj jakieś instrukcje
12   # które zostaną wykonane
13   # gdy warunek_pierwszy NIE będzie spełniony,
14   # warunek_drugi NIE będzie spełniony
15   # ale będzie spełniony warunek_trzeci
16 else
17   # tutaj jakieś instrukcje
18   # które zostaną wykonane
19   # gdy żaden z warunków NIE będzie spełniony
20 end
```

Warto zwrócić uwagę, że – tak jak to pokazują przykłady przedstawione na listingach 5.17, 5.19, 5.20 – instrukcje znajdujące się wewnątrz bloków `if`, `else` czy `elsif` należy pisać z wcięciem. Dzięki temu na pierwszy rzut oka widać, gdzie zaczyna się, a gdzie kończy fragment kodu, który zostanie wykonany przy spełnionym bądź niespełnionym warunku. Wcięcia należy robić od razu w trakcie pisania poszczególnych linii, tak aby uniknąć błędów.

Listing 5.19
```
1  puts "Na którym jesteś roku studiów?"
2  rok = gets.to_i # rok studiów, to liczba całkowita
3
4  do_konca = 5 - rok
5
6  puts "A jaką masz średnią?"
7  srednia = gets.to_f # średnia rzadko jest liczbą całkowitą
8
9  puts "Czy wiesz, że do końca studiów już tylko #{do_konca} lat?"
10
11 if srednia > 3.5 and srednia < 4.5
12   puts "A Twoja średnia #{srednia} tak jakoś pośrodku..."
13 elsif srednia > 4.5
14   puts "A Twoja średnia to aż #{srednia}!"
15 else
16   puts "I Twoja średnia to tylko #{srednia}!?"
17 end
```

Listing 5.20
```
1  puts "Na którym jesteś roku studiów?"
2  rok = gets.to_i # rok studiów, to liczba całkowita
3
4  do_konca = 5 - rok
5
6  puts "A jaką masz średnią?"
7  srednia = gets.to_f # średnia rzadko jest liczbą całkowitą
8
9
10 if do_konca < 3
11   puts "Wiesz, że do końca studiów już tylko #{do_konca} lat?"
12   if srednia > 3.5
13     puts "A Twoja średnia to aż #{srednia}!"
14   else
15     puts "I Twoja średnia to tylko #{srednia}!?"
16   end
17 else
18   puts "Masz jeszcze czas poprawić średnią #{srednia}."
19 end
```

W języku Ruby istnieje dodatkowa instrukcja warunkowa `unless`. Różni się ona od instrukcji `if` tym, że warunek jest interpretowany odwrotnie, tak jak to ilustruje listing 5.21. Wraz z instrukcją `unless` można w razie potrzeby użyć instrukcji `else`, nie można natomiast użyć `elsif`. Czasami użycie `unless` zamiast `if not warunek` poprawia czytelność programu.

Listing 5.21
```
unless warunek
  # tutaj jakieś instrukcje
  # które zostaną wykonane
  # gdy warunek NIE będzie spełniony
else
  # tutaj jakieś instrukcje
  # które zostaną wykonane
  # gdy warunek będzie spełniony
end
```

Jeśli w przypadku spełnienia warunku należy wykonać tylko jedną instrukcję i nie ma potrzeby użycia `else`, w języku Ruby można wykorzystać składnię instrukcji `if`, pokazaną na listingu 5.22.

Listing 5.22
```
instrukcja if warunek
```

Przykład użycia takiej uproszczonej składni instrukcji warunkowej przedstawia linia 11 listingu 5.23. Należy zauważyć, że składnia ta odpowiada językowi naturalnemu, w którym często posługuje się zdaniami, gdzie część warunkowa jest umieszczona na końcu, np. *kup jabłka, jeśli będą świeże*. W podobny sposób można użyć instrukcji `unless`.

Listing 5.23
```
puts "Na którym jesteś roku studiów?"
rok = gets.to_i # rok studiów, to liczba całkowita

do_konca = 5 - rok

puts "A jaką masz średnią?"
srednia = gets.to_f # średnia rzadko jest liczbą całkowitą

puts "Czy wiesz, że do końca studiów już tylko #{do_konca} lat?"

puts "Nie zaliczyłeś niczego!?" if srednia == 2

if srednia > 3.5
  puts "A Twoja średnia to aż #{srednia}!"
else
  puts "I Twoja średnia to tylko #{srednia}!?"
end
```

5.3.6. Iteracje, czyli powtarzanie

W programowaniu wielokrotne wykonywanie tego samego fragmentu kodu nazywa się *iterowaniem*, a pojedyncze powtórzenie nazywa się jedną *iteracją*. W języku Ruby istnieje wiele sposobów na wielokrotne wykonywanie tego samego fragmentu kodu. Tutaj zostaną omówione niektóre z nich, dalej przy opisie struktur danych zostaną przedstawione dodatkowe.

Sposób najbardziej podobny do używanego w innych językach przedstawia listing 5.24. Instrukcje będą powtarzane tak długo, jak długo spełniony będzie warunek.

Listing 5.24
```
1 while warunek
2   # tutaj instrukcje, które
3   # należy powtarzać tak długo
4   # jak długo spełniony jest warunek
5 end
```

Listing 5.25
```
1 i = 0
2 while i < 10
3   puts i
4   i = i + 1 # zwiększ i o jeden
5 end
```

Pętli `while` można użyć np. do wypisania liczb, zaczynając od 0, a kończąc na 9, tak jak to przedstawiono na listingu 5.25. Nie jest to jednak sposób polecany. Pętli `while` w języku Ruby używa się, jeśli inne sposoby iterowania nie pozwalają osiągnąć celu. Wygodniejszy sposób wykonania fragmentu kodu ustaloną liczbę razy przedstawia listing 5.26 oraz działający tak samo 5.27[13].

Listing 5.26
```
1 10.times { |i| puts i }
```

Listing 5.27
```
1 10.times do |i|
2   puts i
3 end
```

Instrukcja `puts i` zostanie wykonana 10 razy; za pierwszym razem zmienna `i` (ta umieszczona pomiędzy znakami `|`) zostanie ustawiona na wartość 0, za drugim – na wartość 1 itd. Jak widać, jest to prostszy sposób osiągnięcia tego samego celu, co w przypadku listingu 5.25.

[13] W języku Ruby słów `do` i `end` można w takim kontekście używać zamiennie z `{` i `}`.

Inne sposoby iterowania przedstawia listng 5.28. Jeśli pętla nie powinna się zaczynać od wartości 1, można skorzystać z jednego ze sposobów przedstawionych w linii 1 i 2. Aby zacząć pętlę od wartości większej i kolejne iteracje wykonywać dla wartości coraz mniejszych, należy użyć sposobu z linii 3. Aby „przejść" jedynie po z góry ustalonych wartościach, można użyć ich listy umieszczonej w kwadratowych nawiasach, jak to pokazano w linii 4. Wtedy iteracje zostaną wykonane jedynie dla wartości znajdujących się na liście, w takiej kolejności, w jakiej umieszczono je na liście.

Listing 5.28
```
1 (5..12).each { |i| puts i }
2 5.upto(12) { |i| puts i }
3 12.downto(5) { |i| puts i }
4 [3, 12, 10, 1].each { |i| puts i }
```

Innym sposobem iterowania po wartościach z listy jest użycie pętli `for`, tak jak to prezentują listingi 5.29 i 5.30. Jest to jednak sposób rzadko spotykany w programach napisanych w języku Ruby. Jak łatwo stwierdzić użycie `each`, jak na listingu 5.28, jest bardziej zwartym rozwiązaniem.

Listing 5.29
```
1 for i in (5..12) do
2   puts i
3 end
```

Listing 5.30
```
1 for i in [3, 12, 10, 1] do
2   puts i
3 end
```

W niektórych przypadkach konieczne jest wcześniejsze zakończenie działania pętli, zanim pozwoli na to warunek (jak w pętli `while`) czy koniec kolekcji (jak dla `for` czy `each`). W tym celu można użyć polecenia `break`, które spowoduje natychmiastowe zakończenie pętli. Program z listingu 5.31 wypisze jedynie liczby od 1 do 3, gdyż dla liczby 100 zostanie wykonana instrukcja `break` i pętla zostanie przerwana.

Listing 5.31
```
1 [1, 2, 3, 100, 5, 6, 7].each do |i|
2   break if i > 10
3   puts i
4 end
```

Drugą instrukcją zaburzającą zwykły przebieg pętli jest `next`. Powoduje ona natychmiastowe przejście do kolejnej iteracji. Program z listingu 5.32 podczas

wypisywania pominie liczbę 100, gdyż dla tej liczby zostanie wykonane polecenie `next`, a w związku z tym nie wykona się polecenie `puts` wypisujące liczbę.

Listing 5.32
```
1 [1, 2, 3, 100, 5, 6, 7].each do |i|
2   next if i > 10
3   puts i
4 end
```

Podobnie jak w przypadku instrukcji warunkowych, pętle można w sobie zagnieżdżać. Można także umieszczać pętle wewnątrz instrukcji warunkowych oraz instrukcje warunkowe wewnątrz pętli. Zawsze jednak należy pamiętać o wykonaniu odpowiednich wcięć, tak aby bardziej zagnieżdżone instrukcje były pisane z większym wcięciem, inaczej zrozumienie napisanego kodu będzie bardzo trudne.

5.3.7. Struktury danych

Przechowywanie bardziej złożonych informacji, niż np. pojedyncze liczby, wymaga użycia bardziej złożonych struktur danych. Struktury danych szerzej omawia rozdział 4. Tutaj zostaną przedstawione dwie struktury danych wbudowane w język Ruby, a więc dostępne bez potrzeby wykonywania jakichkolwiek dodatkowych operacji.

Tablice (listy)

Pierwszą z często używanych struktur danych jest tablica, która w języku Ruby może być też nazywana listą. Tablica przechowuje wiele elementów w kolejnych ponumerowanych polach. Numery pól często nazywa się *indeksami* i mówi się, że *tablica jest indeksowana* liczbami całkowitymi. Do poszczególnych elementów tablicy należy się więc odwoływać, podając numer elementu. W języku Ruby tablice są domyślnie numerowane od 0.

Tablice w języku Ruby mają kilka cech, które nie są spotykane w innych językach, np. w C, C++ czy Javie. Pierwszą z nich jest to, że w języku Ruby tablice mogą przechowywać elementy różnych typów. Oznacza to, że w jednej tablicy można umieścić liczby całkowite, liczby zmiennoprzecinkowe, tekst, inne tablice. Drugą istotną cechą jest to, że rozmiar tablicy może się zmieniać, gdy są do niej dodawane elementy. W wielu językach rozmiar tablicy jest określany podczas jej tworzenia i potem nie można go już zmienić. W języku Ruby nie ma takiego ograniczenia.

Najprostszy sposób tworzenia tablicy przedstawia linia 1 listingu 5.33. Linia 2 tego listingu pokazuje sposób tworzenia tablicy, w której od razu zostaną umieszczone elementy (jak widać różnych typów). Pierwszy z tak dodanych elementów

będzie miał numer 0, kolejny 1 itd. Linia 3 przestawia sposób umieszczania elementów w już istniejącej tablicy – należy wykonać przypisanie jak do zwykłej zmiennej, ale poza nazwą zmiennej należy podać także numer elementu w nawiasach kwadratowych. Podobnie, aby użyć któregoś elementu tablicy, należy podać jego numer, jak to pokazują linie 4-5. Bardzo wygodna jest także możliwość dopisywania elementów na końcu istniejącej tablicy. Linia 6 pokazuje, jak na końcu tablicy `t` można dopisać tekst `"Adam"`.

Użycie tablic jest często wygodne i uzasadnione w połączeniu z pętlami, które pozwalają wygodnie wykonywać operacje na wszystkich bądź wybranych elementach tablicy. Listing 5.28 z sekcji 5.3.6 pokazuje sposób iterowania po tablicy za pomocą `each`. Podobnie jest wypisywana zawartość tablicy w liniach 8 do 10 listingu 5.33. Zmienna `element` jest ustawiana na kolejne elementy zapisane w tablicy, zaczynając od elementu pierwszego, i dla każdego z tych elementów są wykonywane polecenia pomiędzy słowami `do` i `end` (w tym wypadku jedno polecenie `puts`).

Listing 5.33

```
1  t = []
2  a = [1, 5, "Ola"]
3  t[3] = 23.5
4  a[0] = a[1] + 2
5  puts a[2]
6  t << "Adam"
7
8  a.each do |element|
9    puts element
10 end
```

Tablice asocjacyjne

Tablice asocjacyjne (ang. *Hash tables*) są podobne do zwykłych tablic. Zasadnicza różnica polega na tym, że elementy zapisane w tablicach asocjacyjnych nie są numerowane, ale nazywane w inny sposób. Tablice te czasami nazywa się słownikami, ponieważ – podobnie jak do haseł w słowniku są przyporządkowane wyjaśnienia – w tablicach asocjacyjnych do pewnych kluczy są przyporządkowane wartości. Tablice asocjacyjne (w przeciwieństwie do zwykłych) nie muszą więc być indeksowane liczbami całkowitymi. Są one bardzo użyteczną strukturą danych, ale ich implementacja napotyka na pewne trudności. Dlatego też nie jest wcale oczywiste, że tak jak w przypadku języka Ruby są one wbudowane w język.

W języku Ruby tablice asocjacyjne mogą być indeksowane dowolnymi obiektami: liczbami, tekstami. Bardzo często używa się do tego tzw. *symboli*. Symbol w języku Ruby jest bardzo podobny do łańcucha tekstowego, ale pozwala bardziej

efektywnie zarządzać pamięcią, stąd jest polecany do indeksowania tablic zamiast tekstów. Symbole zaczynają się od znaku dwukropka, po którym występuje łańcuch tekstowy bez cudzysłowu, np. `:imie`, `:ocena`.

Najprostszy sposób tworzenia pustej tablicy asocjacyjnej pokazuje linia 1 listingu 5.34. Tablica asocjacyjna opisująca dane studenta mogłaby mieć postać taką, jak na listingu 5.34 w linii 2. To co znajduje się po lewej stronie sekwencji znaków => stanowi *klucz*, natomiast to co po prawej jest *wartością* zapisaną w tablicy pod danym kluczem. Ponieważ tablice asocjacyjne indeksowane symbolami są bardzo często używane w języku Ruby, nieco uproszczono sposób ich tworzenia. Linia 3 spowoduje utworzenie dokładnie takiej samej tablicy asocjacyjnej, jak linia 2, ale jest znacznie krótsza. Takie użycie symboli jest jednak możliwe tylko podczas tworzenia nowej tablicy asocjacyjnej. Linia 4 pokazuje, jak można odwołać się do wartości zapisanych w tablicy. Wartości te można oczywiście zmieniać, tak jak to zaprezentowano w linii 5.

Iterowanie po tablicach asocjacyjnych wykonuje się podobnie jak dla zwykłych tablic. Różnica polega na tym, że w przypadku tablic asocjacyjnych wewnątrz pętli jest dostępny zarówno klucz, jak i zapisana pod nim w tablicy wartość. Przykład pokazano w liniach od 7 do 9.

Listing 5.34

```
1 a = {}
2 stud = { :imie => "Jan", :nazwisko => "Nowak", :srednia => 4.3 }
3 stud = { imie: "Jan", nazwisko: "Nowak", srednia: 4.3 }
4 puts stud[:imie]
5 stud[:srednia] = 4.73
6
7 stud.each do |klucz, wartosc|
8   puts "#{klucz}: #{wartosc}"
9 end
```

5.4. Programowanie strukturalne

Programowanie strukturalne jest stylem programowania od dawna uważanym za właściwy i niemal wszystkie współczesne imperatywne języki wspierają ten styl. Początkowo, gdy programy były pisane w języku maszynowym, czy w języku Assemblera, program składał się z ciągu sekwencyjnie wykonywanych instrukcji. Porządek wykonania zakłócały jedynie *skoki* (warunkowe bądź bezwarunkowe), które pozwalały przeskoczyć pewną liczbę instrukcji wprzód bądź wstecz. Większe programy pisane w ten sposób bardzo szybko stawały się nieczytelne. Skoki wykonywane z różnych miejsce w różne inne miejsca powodowały, że zrozumienie za co odpowiadał dany fragment kodu, z czym

był powiązany, od czego zależało jego działanie i jakie skutki powodował było bardzo trudne. Skutkiem były poważne trudności w utrzymaniu kodu oraz w poszukiwaniu i usuwaniu błędów.

Koncepcja programowania strukturalnego pozwala uniknąć tego typu problemów. Zakłada ona, że kod programu jest podzielony na hierarchicznie ułożone *bloki*, z których każdy realizuje konkretne funkcje. Każdy z bloków może być pominięty bądź wykonany jeden lub więcej razy. Zasadniczo jednak bloki kodu sa wykonywane w całości: nie ma możliwości „wskoczenia" w środek bloku (wykonywanie bloku musi się odbywać od początku). Nie „wyskakuje" się też przed zakończeniem bloku. Układ taki pozwala nie tylko łatwiej zrozumieć za co odpowiada każdy z fragmentów programu, ale też lepiej zoptymalizować kod podczas tłumaczenia na język maszynowy.

Programowanie niestrukturalne w języku Ruby jest w zasadzie niewykonalne. Jak to można zobaczyć w sekcjach 5.3.5 i 5.3.6, instrukcje warunkowe i pętle odpowiadają za to, ile razy zostanie wykonany blok kodu (oznaczany w języku Ruby znakami `{` i `}` bądź słowami `do`[14] i `end`). W języku Ruby nie ma instrukcji skoku (`goto`), która w wielu językach pozwala skoczyć z dowolnego do innego miejsca w programie. Instrukcja ta nawet dla języków doskonale wspierających tworzenie programów strukturalnych (jak język C) pozwalała pisać niestrukturalne, zawiłe i niezrozumiałe programy.

Przykład strukturalnie napisanego programu prezentuje listing 5.35. Nie trzeba wielkiej wprawy, żeby szybko stwierdzić, że dla liczb $<= 0$ program ten wypisze komunikat błędu, a dla pozostałych komunikat o tym, czy podana liczba jest parzysta. W języku Ruby na szczęście nie ma instrukcji `goto`, więc program z listingu 5.36 nie będzie działał. Jest to jednak przykład tego, jak może wyglądać program niestrukturalny. Gdyby instrukcja `goto` działała w języku Ruby, powodując przeskok do podanej etykiety, program ten działałbym tak samo, jak ten z listingu 5.35.

Programowanie niestrukturalne jest możliwe w wielu współcześnie używanych językach programowania. Należy go jednak unikać zarówno ze względu na czytelność programu, jak i na jego efektywność. Obecnie nie ma wątpliwości, że strukturalne podejście do pisania programów jest dobrą praktyką.

[14] W niektórych wypadkach, np. dla instrukcji `if`, instrukcję `do` można pominąć, jak to prezentują przykłady, gdyż wiadomo, że instrukcja `if` rozpoczyna blok kodu.

Listing 5.35
```
1  puts "Podaj liczbę całkowitą, większą od zera: "
2  liczba = gets.to_i
3
4  if liczba > 0            # czy dodatnia?
5    if liczba % 2 == 0     # czy parzysta?
6      puts "Liczba #{liczba} jest parzysta"
7    else
8      puts "Liczba #{liczba} jest nieparzysta"
9    end
10 else  # jeśli nie jest dodatnia
11   puts "Prosiłem o liczbę większą od zera!"
12 end
```

Listing 5.36
```
1  # Ten program nie będzie działał. W Rubym nie ma instrukcji
2  # goto. To tylko ilustracja programowania niestrukturalnego.
3  puts "Podaj liczbę całkowitą, większą od zera: "
4  liczba = gets.to_i
5
6  if not liczba > 0 goto blad
7  if liczba % 2 == 0 goto parzysta
8  puts "Liczba #{liczba} jest nieparzysta"
9  goto koniec
10 parzysta: puts "Liczba #{liczba} jest parzysta"
11 goto koniec
12 blad: puts "Prosiłem o liczbę większą od zera!"
13 koniec:
```

5.5. Programowanie proceduralne

5.5.1. Motywacja

Pisanie dobrych, łatwych w utrzymaniu programów wymaga poza programowaniem strukturalnym zastosowania dodatkowych technik. Taką techniką, doskonale łączącą się z programowaniem strukturalnym, jest programowanie proceduralne.

5.5.2. Podprogramy

Podczas programowania bardzo szybko okazuje się, że samo wykonywanie fragmentu kodu zero lub więcej razy nie wystarcza. Często jeden fragment kodu jest wielokrotnie potrzebny w różnych miejscach programu. Przykładowo, w programie obsługującym dziekanat uczelni obliczanie średniej ocen może być wykonywane wielokrotnie dla różnych danych (średnia ocen studenta z danego se-

mestru, średnia ocen grupy studentów, średnia ocen całego roku itp). Sposobem na uniknięcie konieczności wielokrotnego pisania (czy kopiowania) tego samego fragmentu kodu w różnych miejscach są podprogramy. Podprogram może otrzymywać listę parametrów (np. listę ocen, z których należy policzyć średnią), może wykonywać pewne operacje i zwracać wynik (np. obliczoną średnią). Zaletą podprogramu jest to, że można go wywoływać wielokrotnie z różnych miejsc w programie, wystarczy więc napisać go raz i dokładnie przetestować.

W różnych językach programowania podprogramy są różnie nazywane. Czasami są określane jako procedury, czasami jako funkcje. Dla języka Ruby wybrano pojęcie *funkcja*, którym będziemy się dalej posługiwać. Funkcja w języku Ruby pobiera zero lub więcej parametrów, wykonuje pewne operacje i zwraca jakąś wartość. Definiuje się ją za pomocą słowa kluczowego `def`, tak jak pokazuje to listing 5.37.

Listing 5.37

```
1 def nazwa_funkcji(lista, parametrow)
2   # instrukcje wykonujące
3   # pewne operacje
4 end
```

W języku Ruby są dwa sposoby zwracania wyniku z funkcji. W najprostszym przypadku jako wynik funkcji zwracana jest wartość ostatniego wyrażenia, które zostało wykonane wewnątrz funkcji. Ten sposób ilustruje przykład z listingu 5.38. Funkcja `dodaj` zawiera dwa polecenia: pierwsze wypisuje komunikat na ekranie, drugie dodaje do siebie liczby podane jako parametry funkcji. Wynik dodawania nie jest zapamiętywany w żadnej zmiennej. Ponieważ jednak jest to ostatnia operacja wykonana wewnątrz funkcji, jej wynik będzie wynikiem funkcji.

Listing 5.38

```
1 def dodaj(a, b)
2   puts "Dodaję #{a} do #{b}, zwracam wynik"
3   a + b
4 end
```

Drugim sposobem zwracania wartości z funkcji jest użycie słowa kluczowego `return`. Słowo to powoduje, że wykonanie funkcji kończy się natychmiast (żadne dalsze polecenia nie zostaną wykonane), a wartość występująca po słowie return staje się wartością funkcji. Przykład użycia `return` przedstawia linia 5 listingu 5.39 – jeśli jako parametr podano pustą listę (`oceny.empty?`), zwrócona zostanie wartość `nil` i funkcja zostanie zakończona. Jeśli parametrem nie będzie pusta lista, to zostanie obliczona wartość średniej. Wartość ta zostanie zwrócona pierwszym opisanym sposobem: iloraz `suma / ilosc` jest ostatnim wyrażeniem wewnątrz funkcji i jego wartość będzie wartością funkcji.

Listing 5.39
```
1  # funkcja przyjmuje jako parametr listę ocen
2  # zwraca wartość średniej na tej liście
3  # albo nil jeśli lista jest pusta
4  def srednia(oceny)
5    return nil if oceny.empty?
6
7    suma = 0.0
8    ilosc = 0
9    oceny.each do |o|
10     suma = suma + o
11     ilosc = ilosc + 1
12   end
13   suma / ilosc
14 end
```

Jak widać na listingu 5.39, programowanie proceduralne doskonale współgra z programowaniem strukturalnym. Ciało (treść) funkcji (linie 5-13), która może być wielokrotnie wywoływana, stanowi blok kodu. Wewnątrz niego poza prostymi instrukcjami, jak `suma = 0.0` w linii 7, znajduje się pętla (linie 9-12) obejmująca blok kodu zawierającego dwie instrukcje (linie 10-11). Połączenie programowania strukturalnego i proceduralnego pozwala tworzyć programy, w których nie tylko bloki kodu są ułożone hierarchicznie w przejrzysty sposób, ale także jeden blok kodu może być wielokrotnie wykorzystywany.

Listing 5.40
```
1  x = dodaj(12.3, 7.0)
2  y = dodaj(x, 5)
3
4  # lista ocen:
5  o = [3.5, 4.0, 5, 3, 3]
6  s = srednia(o)
7
8  s2 = srednia([5.0, 4.5, 4.5, 4.0])
```

Wywołanie funkcji w języku Ruby jest bardzo prostą operacją. Przykłady wywołania funkcji `dodaj` i `srednia` znajdują się na listingu 5.40. Wartości zwracane przez funkcje są zapamiętywane w zmiennych. W języku Ruby w wielu przypadkach nawiasy przy wywołaniu funkcji mogą być pominięte[15]. Dlatego wywołania funkcji mogą być wykonane tak, jak to przedstawia listing 5.41.

Zastosowanie programowania proceduralnego pozwala nie tylko pisać przejrzyste programy bez konieczności wielokrotnego powtarzania tych samych fragmentów kodu, ale także na wprowadzenie do programu pewnej abstrakcji i posłu-

[15] Jeśli jednoznacznie można określić, które parametry dotyczą której z wywoływanych funkcji.

Listing 5.41
```
1 x = dodaj 12.3, 7.0
2 y = dodaj x, 5
3
4 # lista ocen:
5 o = [3.5, 4.0, 5, 3, 3]
6 s = srednia o
7
8 s2 = srednia [5.0, 4.5, 4.5, 4.0]
```

giwanie się nią. Funkcja zamyka wewnątrz szczegóły dotyczące sposobu realizacji określonego zadania, dzięki czemu programista nie musi się nad nimi więcej zastanawiać, wystarczy że wywoła odpowiednią funkcję. Pozwala to a pisanie bardziej złożonych programów.

5.5.3. Zmienne lokalne i globalne

Funkcje stanowią autonomiczne fragmenty kodu; wywoływane z różnych miejsc mają działać zawsze tak samo, zawsze poprawnie, niezależnie od miejsca w programie, z którego zostały wywołane. Aby było to możliwe, każda funkcja może posługiwać się swoim zestawem zmiennych, niezależnych od zmiennych w innych miejscach programu. Takie zmienne nazywają się *zmiennymi lokalnymi*. Jeśli zmienna, taka jakiej używano do tej pory, zostanie utworzona wewnątrz funkcji (poprzez przypisanie do niej jakiejś wartości), to będzie zmienną lokalną tej funkcji. Funkcja `srednia` z listingu 5.39 zawiera zmienne lokalne `suma` i `ilosc`. Zmienne te nie są widziane spoza funkcji. Podobnie funkcja `srednia` nie ma dostępu do zmiennych lokalnych na zewnątrz. Dane wejściowe do funkcji należy przekazywać za pośrednictwem jej parametrów w momencie wywołania. Wyniki swoich działań funkcja przekazuje, zwracając wartości tak jak funkcja `srednia` zwraca obliczoną średnią ocen.

Tak jak w wielu innych językach, w języku Ruby istnieją *zmienne globalne*, czyli zmienne widoczne w całym programie. W języku Ruby nazwy zmiennych globalnych muszą zaczynać się od znaku `$`, np. `$zmienna`. Dostęp do takich zmiennych jest możliwy ze wszystkich funkcji w całym programie. Należy jednak unikać używania tego rodzaju zmiennych, ponieważ powodują one dużo zamieszania, utrudniają poszukiwanie błędów i sprawiają trudności podczas rozwijania oprogramowania.

5.6. Programowanie obiektowe

5.6.1. Rozwój koncepcji

Paradygmat programowania obiektowego był odpowiedzią na potrzebę tworzenia coraz większych programów i w związku z tym na konieczność dzielenia kodu programu na zwarte, zamknięte fragmenty, które mogły być niezależnie testowane i używane przez inne części oprogramowania. Koncepcja ta jest nieco bardziej złożona niż programowanie strukturalne czy proceduralne. Jest ona jednak, podobnie jak te dwa wcześniej omówione paradygmaty, powszechnie stosowana i wspierana (a czasami wręcz wymuszana) przez większość nowoczesnych języków programowania.

5.6.2. Obiekty i klasy

Koncepcja programowania obiektowego mówi, że świat składa się z różnego rodzaju *obiektów*, które wchodzą ze sobą w interakcje. Obiekty te są charakteryzowane przez swoje cechy. Program obiektowy składa się więc z zestawu obiektów, które mają swoje właściwości (cechy), które potrafią wykonywać pewne operacje oraz w określony sposób wysyłać komunikaty do innych obiektów. Informacje, które obiekt przechowuje (jego cechy), mogą się zmieniać w czasie działania programu na skutek wykonywanych przez obiekt operacji (np. cechą samochodu może być ilość paliwa w baku zmniejszająca się w czasie jazdy, a zwiększająca podczas tankowania). O tej sytuacji często się mówi, że obiekt przechowuje informacje o swoim *stanie*.

W świecie rzeczywistym można zauważyć, że składające się na ten świat obiekty można pogrupować w typy. Do jednego typu można zaliczyć obiekty, które reprezentują taki sam zestaw cech, potrafią wykonywać taki sam zestaw operacji i rozumieją takie same komunikaty. Innym rodzajem obiektów są np. samochody (dla których możemy podać pojemność silnika i z którymi komunikujemy się za pomocą pedałów i kierownicy), a innym osoby (dla których podaje się imię i nazwisko i z którymi można się komunikować za pomocą mowy). Z tego powodu w programowaniu obiektowym obiekty zostały pogrupowane w tzw. *klasy*. Klasa określa, jaki zestaw cech opisuje obiekty tej klasy, jaki zestaw operacji mogą one wykonywać i jakie komunikaty rozumieją.

W programie napisanym w języku Ruby wszystko jest obiektem. Programista może oczywiście tworzyć także swoje obiekty w miarę potrzeb, a żeby to ułatwić, może wykorzystać klasy. Nazwy klas w języku Ruby zaczynają się wielką literą. Cechy obiektów są przechowywane w specjalnych zmiennych, nazywanych *polami*. W języku Ruby pola muszą mieć nazwy zaczynające się od znaku `@`, np. `@kolor`. Operacje, które obiekt „potrafi” wykonać, są zaprogramowane w postaci

funkcji (takich jak opisane w podrozdziale 5.5) zawartych wewnątrz tego obiektu. Takie funkcje nazywa się *metodami*. *Metody* mają dostęp do *pól* obiektu, mogą pobierać z nich wartości i mogą je zmieniać. *Metody* mogą więc zmieniać *stan* obiektu.

Listing 5.42
```
1 class Samochod
2   def pomaluj(kolor)
3     @kolor = kolor
4   end
5
6   def kolor
7     @kolor
8   end
9 end
```

Listing 5.42 pokazuje przykładową klasę o nazwie `Samochod`. Klasa ta ma dwie metody: `pomaluj` i `kolor`. Metoda `pomaluj` pobiera jako parametr kolor, na który ma być pomalowany samochód i zapisuje go w polu obiektu o nazwie `@kolor`. W ten sposób nowy kolor samochodu zostanie zapamiętany. Metoda `pomaluj` zmienia więc stan obiektu. Druga metoda `kolor` zwraca wartość zapisaną w polu `@kolor`, pozwalając sprawdzić, jaką wartość ma ta cecha samochodu.

Klasa `Samochod` pozwala wygodnie tworzyć obiekty reprezentujące samochody. Przykład tworzenia obiektów pokazuje listing 5.43. Linie 1-3 spowodują utworzenie trzech obiektów klasy `Samochod`, z których każdy z osobna i niezależnie od innych jest charakteryzowany przez wartość koloru.

Listing 5.43
```
1 moj_samochod = Samochod.new
2 samochod_zony = Samochod.new
3 samochod_corki = Samochod.new
```

5.6.3. Metody

W programie obiektowym obiekty mogą wysyłać do siebie komunikaty, podobnie jak to się odbywa w świecie rzeczywistym. Pozwala to jednym obiektom przekazywać innym informację o tym, co powinny zrobić. W języku Ruby obiekt odbierający komunikat o określonej nazwie wywołuje swoją metodę, nazywającą się tak jak komunikat. Jeśli więc do jednego z obiektów klasy `Samochod` zostanie wysłany komunikat `kolor`, obiekt ten wywoła swoją metodę o nazwie `kolor`[16]. W wielu językach fakt wysyłania komunikatów do obiektu został pominięty już

[16] To domyślne zachowanie w języku Ruby można zmienić tak, że wysłanie komunikatu będzie skutkowało wywołaniem innej metody.

w fazie projektowania języka i mówi się o *wywołaniu metody na obiekcie*. W kontekście języka Ruby takie stwierdzenie jest uproszczeniem, ale często używa się go jako skrótu myślowego. Wysyłanie komunikatu do obiektu zapisuje się w programie za pomocą kropki. Przykładowo, wysłanie komunikatu `kolor` do obiektu `moj_samochod` zapisuje się tak, jak to przedstawiono na listingu 5.44.

Listing 5.44

```
1 moj_samochod.kolor
```

Używając komunikatów, można więc zażądać pomalowania obiektów samochodów utworzonych wcześniej (listing 5.43) oraz sprawdzać i np. wypisywać na ekranie ich kolory (zob. listing 5.45)[17].

Listing 5.45

```
1 moj_samochod.pomaluj(:czarny)
2 samochod_zony.pomaluj(:czerwony)
3 samochod_corki.pomaluj(:rozowy)
4
5 puts "Mój samochód ma kolor: #{moj_samochod.kolor}"
```

5.6.4. Dostęp do metod i pól obiektu

Na ogół w językach obiektowych istnieje możliwość ograniczenia dostępu do metod. Niektóre mogą być wywoływane tylko przez inne metody zdefiniowane w tym samym obiekcie (w niektórych językach także w innych obiektach tej samej klasy) – takie metody nazywa się *prywatnymi*. Inne metody można wywoływać z zewnątrz (z innych obiektów, z obiektów innych klas) – te metody nazywa się *publicznymi*. Dostęp do pól obiektu może być także ograniczony. W niektórych językach są to takie same ograniczenia, jak dla metod. W języku Ruby wszystkie pola obiektu są prywatne, co oznacza, że nie można się do nich „dostać” spoza tego obiektu. Można oczywiście udostępnić dane zapisane w polach bądź umożliwić ich zmianę, ale tylko za pomocą odpowiednich metod. Tak jak w zdefiniowanej wcześniej klasie `Samochod` (listing 5.42), zapewniono dostęp do wartości zapisanej w polu `@kolor`.

Domyślnie metody w języku Ruby są publiczne, a więc dostępne dla wszystkich. Aby umieścić w klasie metody prywatne, w definicji klasy należy użyć słowa `private`. Wszystkie metody zdefiniowane poniżej tego słowa będą prywatne. Listing 5.46 przedstawia zmodyfikowaną definicję klasy `Samochod` zawierającą prywatną metodę `znowu_malowanie`, zapamiętującą poprzedni kolor, na który

[17] W tym przykładzie użyto symboli do reprezentowania kolorów. Można oczywiście zastosować inną reprezentację, np. łańcuchy tekstowe czy liczby.

był pomalowany samochód. Metoda ta nie może być wywołana spoza obiektu[18], jest natomiast automatycznie wywoływana w przypadku malowania samochodu przez metodę `pomaluj`, a więc z „wnętrza” obiektu.

Listing 5.46

```
1  class Samochod
2    def pomaluj(kolor)
3      znowu_malowanie    # wywołanie metody prywatnej
4      @kolor = kolor
5    end
6
7    def kolor
8      @kolor
9    end
10
11   private             # metody zdefiniowane poniżej
12                       # tego miejsca będą prywatne
13   def znowu_malowanie
14     @stary_kolor = @kolor
15   end
16 end
```

5.6.5. Konstruowanie obiektu

Podczas konstruowania nowego obiektu często konieczne jest ustawienie jego początkowego *stanu* – początkowej wartości jego pól. Niektóre z tych wartości mogą być podawane podczas tworzenia obiektu, a niektóre mogą być ustawiane na stałe według określonego algorytmu. Przykładowo, jeśli klasa `Samochod` miałaby reprezentować także ilość paliwa zwiększaną przy tankowaniu, a zmniejszaną podczas jazdy, to początkowo należałoby ustawić tę ilość na 0. Kolor natomiast można ustawić według życzenia klienta.

Ustawienia początkowego stanu obiektów wykonuje się za pomocą tzw. *konstruktorów*. *Konstruktor* jest metodą, która ma dwie cechy różniące ją od innych metod. Po pierwsze jest on wywoływany automatycznie podczas tworzenia nowego obiektu. Po drugie nie może on zwrócić żadnej użytecznej wartości, nie ma bowiem możliwości odzyskania tej wartości.

Konstruktor w języku Ruby musi się nazywać `initialize`. Mając możliwość użycia konstruktora, do klasy `Samochod` można dodać funkcjonalność reprezentującą zużycie paliwa, np. tak jak to przedstawiono na listingu 5.47.

Konstruktor ustawia kolor samochodu na podaną wartość oraz ilość paliwa na 0 (linie 3-6). Tankowanie samochodu poprzez wywołanie metody `tankuj`

[18] Jeśli do obiektu klasy `Samochod` zostanie wysłany komunikat `znowu_malowanie`, obiekt odmówi wykonania metody.

Listing 5.47

```
class Samochod

  def initialize(kolor)
    @kolor = kolor
    @paliwo = 0
  end

  def jazda(ile_km)
    ile_paliwa = spalanie ile_km
    @paliwo = @paliwo - ile_paliwa
  end

  def tankuj(ile)
    @paliwo = @paliwo + ile
  end

  def pomaluj(kolor)
    znowu_malowanie     # wywołanie metody prywatnej
    @kolor = kolor
  end

  def kolor
    @kolor
  end

  private            # metody zdefiniowane poniżej
                     # tego miejsca będą prywatne
  def znowu_malowanie
    @stary_kolor = @kolor
  end

  def spalanie(km)   # oblicz ile spali samochod
    km * 6.0 / 100
  end
end
```

z podaniem liczby litrów powoduje zwiększenie ilości paliwa (linie 13-15). Jazda samochodem poprzez wywołanie metody `jazda` z podaniem liczby kilometrów powoduje zmniejszenie ilości paliwa w baku (linie 8-11). Ilość spalanego paliwa jest obliczana przez prywatną metodę `spalanie` (linie 32-34).

Sposób użycia klasy `Samochod` pokazuje listing 5.48. Na początku jest tworzony obiekt reprezentujący czerwony samochód. Następnie samochód jest tankowany 40 l paliwa, po czym przejeżdża 120 km. Ilość paliwa w baku samochodu

po ostatniej operacji zostanie automatycznie zmniejszona odpowiednio do przejechanego dystansu.

Listing 5.48
```
1 auto = Samochod.new :czerwony # Nowy czerwony samochód
2                               #                z pustym bakiem
3 auto.tankuj 40                # Tankuj 40 litrów
4 auto.jazda 120                # Jedź 120 km, il. paliwa
5                               # zmieniejszy się automatycznie
```

5.6.6. Obsługa błędów

Klasa z listingu 5.47 zawiera jedną poważną usterkę. Jeśli ilość paliwa w baku nie będzie wystarczająca do przejechania żądanej liczby kilometrów, obiekt klasy `Samochod` ustawi ilość paliwa na wartość mniejszą od zera (linie 9-10). Nie jest to sytuacja odpowiadająca rzeczywistości i tego typu błąd powinien zostać obsłużony.

Obsługa błędów w programowaniu obiektowym jest wykonywana za pomocą tzw. *wyjątków* (ang. *exceptions*). Program w momencie wystąpienia błędu *rzuca* (*generuje* bądź *zgłasza*) *wyjątek* (ang. *throws exception* bądź *raises exception*) zawierający opis sytuacji. Wyjątek taki może być w innym miejscu przechwycony (ang. *catch*) i obsłużony (operację tę można określić angielskim słowem *rescue*), np. poprzez wyświetlenie okna z komunikatem błędu. Jeśli jakiś wyjątek nie zostanie przechwycony, to spowoduje on zatrzymanie programu. Ponieważ wyjątki mają swoje typy (są obiektami różnych klas), łatwo jest selektywnie je obsługiwać w różnych miejscach w programie. Dzięki temu obsługa błędów za pomocą wyjątków jest elastyczna i wygodna.

Generowanie wyjątków w języku Ruby nie jest skomplikowane. Aby zgłosić wyjątek typu `RuntimeError` wystarczy napisać `raise "Komunikat błędu"`. Poprawiona wersja klasy `Samochod` zawierałaby więc metodę `jazda` z listingu 5.49.

Listing 5.49
```
1   def jazda(ile_km)
2     ile_paliwa = spalanie ile_km
3     if ile_paliwa > @paliwo
4       raise "Zbyt mało paliwa, żeby przejechać #{ile_km} km!"
5     end
6     @paliwo = @paliwo - ile_paliwa
7   end
```

Próba przejechania samochodem z pustym bakiem 100 km spowoduje błąd

```
RuntimeError: Zbyt mało paliwa, żeby przejechać 100 km!
from /tmp/c_samochod_paliwo.rb:12:in `jazda'
```

i program zostanie zakończony.

Listing 5.50

```
1 begin
2   s.jazda 100
3 rescue RuntimeError => e
4   puts "Nic z tego: #{e}"
5 end
```

Obsługę wyjątku wykonuje się, używając bloku `begin-rescue` wokół fragmentu kodu, w którym może się pojawić wyjątek (np. wywołania metody, która może zgłosić wyjątek, bądź metody wywołującej metodę, która może zgłosić wyjątek itd.). Program z listingu 5.50 przechwytuje zgłoszony wyjątek typu `RuntimeError` i zapamiętuje go w zmiennej `e`. Następnie zmienna ta jest wykorzystywana podczas wypisywania komunikatu o błędzie: w linii 4 program wypisuje swój komunikat, dołączając do niego komunikat zawarty w wyjątku. Po przechwyceniu i obsłużeniu wyjątku działanie programu jest kontynuowane. Jeśli wyjątek nie zostanie zgłoszony, blok po słowie `rescue` nie będzie wykonany. Pomiędzy słowem `begin` a `rescue` może się znajdować dowolny fragment programu, zawierający wiele instrukcji, wiele wywołań różnych metod. Jeśli w trakcie wykonania takiego fragmentu zostanie wygenerowany wyjątek, zostanie on obsłużony przez odpowiedni blok `rescue`.

Listing 5.51 zawiera kompletną definicję klasy `Samochod` wraz z obsługą błędów. Przykład wykorzystania tej klasy znajduje się na listingu 5.52. Program po zatankowaniu 40 l paliwa do samochodu zadaje użytkownikowi pytanie o liczbę kilometrów, które chce przejechać, następnie wypisuje informację o przejechaniu zadanego dystansu. Jeśli okazałoby się, że zatankowana ilość paliwa jest zbyt mała, program zamiast informacji o przejechaniu zadanego dystansu wypisze odpowiedni komunikat błędu.

5.6.7. Dziedziczenie

Jedną z najistotniejszych koncepcji związanych z programowaniem obiektowym jest dziedziczenie. Klasy mogą dziedziczyć pola i metody po innych klasach, dodając swoje, modyfikując niektóre, i w ten sposób tworząc strukturę bardziej ogólnych lub bardziej specjalizowanych klas.

Dalszy rozwój oprogramowania zawierającego klasę `Samochod` może doprowadzić do sytuacji, w której osobno powinny być reprezentowane samochody ciężarowe, a osobno osobowe. Wszystkie jednak będą miały cechy wspólne, dotychczas zdefiniowane dla samochodu. W tej sytuacji można utworzyć dwie

Listing 5.51

```
1  class Samochod
2
3    def initialize(kolor)
4      @kolor = kolor
5      @paliwo = 0
6    end
7
8    def jazda(ile_km)
9      ile_paliwa = spalanie ile_km
10     if ile_paliwa > @paliwo
11       raise "Zbyt mało paliwa, żeby przejechać #{ile_km} km!"
12     end
13     @paliwo = @paliwo - ile_paliwa
14   end
15
16   def tankuj(ile)
17     @paliwo = @paliwo + ile
18   end
19
20   def pomaluj(kolor)
21     znowu_malowanie    # wywołanie metody prywatnej
22     @kolor = kolor
23   end
24
25   def kolor
26     @kolor
27   end
28
29   private
30
31   def znowu_malowanie
32     @stary_kolor = @kolor
33   end
34
35   def spalanie(km)   # oblicz ile spali samochod
36     km * 6.0 / 100
37   end
38 end
```

klasy `Ciezarowy` i `Osobowy` dziedziczące (własności) po klasie `Samochod`. Listing 5.53 pokazuje, jak należy zapisać informację, że nowo tworzona klasa `Ciezarowy` (jak dotąd bez żadnych modyfikacji) ma dziedziczyć po klasie `Samochod`. W takim układzie klasę `Ciezarowy` nazywa się *klasą pochodną*, a klasę `Samochod` *klasą bazową*.

Listing 5.52
```ruby
auto = Samochod.new :czerwony # Nowy czerwony samochów
                              #          z pustym bakiem
auto.tankuj 40                # Tankuj 40 litrów
puts "Ile km mam przejechać? "
odleglosc = gets.to_i

begin
  auto.jazda odleglosc
  puts "Przejechano #{odleglosc} km."
rescue RuntimeError => e
  puts "Nic z tego: #{e}"
end
```

Listing 5.53
```ruby
class Ciezarowy < Samochod
end
```

Klasa opisująca samochód ciężarowy może udostępniać możliwość załadowania towaru za pomocą metody `zaladuj` z listingu 5.54. Ilość paliwa spalanego przez samochód ciężarowy jest większa niż dla samochodu osobowego i dodatkowo znacząco zależna od ładunku. Klasa `Ciezarowy` może więc zmodyfikować odziedziczoną po klasie `Samochod` metodę `spalanie`, jak to przedstawiono na listingu 5.55.

Listing 5.54
```ruby
  def zaladuj(ile)
    @ladunek = ile
  end
```

Listing 5.55
```ruby
  def spalanie(km)
    km * (20.0 + 0.5 * @ladunek) / 100
  end
```

W języku Ruby, jeśli nie zdefiniowano nowego konstruktora, zostanie wykonana metoda odziedziczona z klasy bazowej. Podczas tworzenia nowego obiektu samochodu ciężarowego konieczne jest jednak ustawienie jego aktualnego ładunku na wartość 0. Można to zrobić za pomocą nowego konstruktora (listing 5.56).

Konstruktor wykorzystuje słowo `super`, żeby wywołać konstruktor odziedziczony z klasy `Samochod`, dzięki czemu nie ma konieczności przepisywania po raz kolejny tego samego kodu. Taką samą operację można wykonać, definiując na nowo inne odziedziczone metody. Kompletna definicja klasy `Ciezarowy` będzie więc wyglądała tak jak na listingu 5.57.

Listing 5.56

```
1    def initialize(kolor)
2      super(kolor)   # wywołanie odziedziczonego konstruktora
3      @ladunek = 0   # ustawienie dodatkowych atrybutów
4    end
```

Listing 5.57

```
1  class Ciezarowy < Samochod
2
3    def initialize(kolor)
4      super(kolor)   # wywołanie odziedziczonego konstruktora
5      @ladunek = 0   # ustawienie dodatkowych atrybutów
6    end
7
8    def zaladuj(ile)
9      @ladunek = ile
10   end
11
12   private
13
14   def spalanie(km)
15     km * (20.0 + 0.5 * @ladunek) / 100
16   end
17 end
```

Za pomocą stosunkowo niewielkich modyfikacji i bez potrzeby powtarzania tego samego kodu uzyskano nową klasę reprezentującą samochód ciężarowy, mający cechy innych samochodów, ale różniący się od nich możliwością załadowania towaru oraz sposobem, w jaki należy obliczyć spalanie paliwa.

Hierarchia dziedziczenia może być bardziej rozbudowana. Po klasie `Samochod` może dziedziczyć więcej klas, po klasie `Ciezarowy` może dziedziczyć kolejna, reprezentująca bardziej specyficzny rodzaj samochodu itd. W języku Ruby klasa może dziedziczyć tylko z jednej klasy bazowej. W niektórych językach, np. w C++, możliwe jest także wielokrotne dziedziczenie.

5.7. Programowanie imperatywne i deklaratywne

Wszystkie opisane dotąd rozwiązania stanowią przykłady *programowania imperatywnego*. Polega ono na opisywaniu kolejnych kroków prowadzących od danych wejściowych do żądanego rozwiązania. Opisywane jest więc nie tyle **co** należy osiągnąć, ale **jak** należy to zrobić. Nie jest definiowane jak ma wyglądać rezultat, a tylko jak należy do niego dojść. Przykładem może być pieczenie ciasta na podstawie przepisu, bez jakiejkolwiek informacji o rodzaju tego ciasta. Do-

świadczony kucharz będzie wprawdzie potrafił przewidzieć efekt, ale osoba bez odpowiedniego doświadczenia dopiero po wykonaniu wszystkich kroków zobaczy czemu służył przepis. Funkcja z listingu 5.39 definiuje poszczególne kroki, które listę ocen zamieniają na pojedynczą liczbę. Analizując te kroki, można oczywiście dojść do wniosku, że liczba ta będzie średnią ocen i że będzie to średnia arytmetyczna. Jednakże nigdzie ten fakt nie został zdefiniowany[19].

Programowanie deklaratywne polega na opisywaniu żądanych rezultatów bez definiowania sposobu ich osiągnięcia. Opisywane jest więc **co** należy osiągnąć, a nie **jak** należy to zrealizować. Realizujący zadanie (w przypadku programu jest to komputer) musi na podstawie opisu żądanych wyników odnaleźć sposób na ich dostarczenie. W niektórych przypadkach jest to stosunkowo proste, w innych wymaga zastosowania bardzo złożonych algorytmów.

Język Ruby jest językiem imperatywnym, jednakże w wielu przypadkach możliwa jest realizacja zadań w sposób deklaratywny. Prostym przykładem jest umożliwienie dostępu do pól obiektu. Jak opisano w sekcji 5.6.4, wszystkie pola obiektów są prywatne, a dostęp do nich jest możliwy za pomocą przeznaczonych do tego metod. Jeśli metody te mają wprost zwracać wartość zapisaną w danym polu albo ustawiać tę wartość, mogą wyglądać tak jak na listingu 5.58. Listing ten zawiera definicję klasy `Student` z dwoma polami: `@imie` oraz `@nazwisko`, każde z tych pól jest dostępne do odczytu za pomocą metod `imie` i `nazwisko` oraz do zapisu za pomocą metod `imie=` oraz `nazwisko=`[20]. Metody te służące do odczytywania i ustawiania wartości pól obiektu są często nazywane odpowiednio *getter*ami i *setter*ami.

Definiowanie w ten sposób metod dla każdej właściwości jest jednakże uciążliwe. Język Ruby jednak pozwala wykonać to w sposób deklaratywny, tzn. zadeklarować, że wybrane właściwości mają być dostępne do odczytu, do zapisu bądź do odczytu i do zapisu. Listing 5.59 przedstawia klasę, w której zadeklarowano, że dla pól `imie` i `nazwisko` potrzebne są metody do odczytu i do zapisu, dla pól `srednia` potrzebna jest metoda do odczytu, a dla pola `ocena` tylko metoda do zapisu. Jak widać, zapis deklaratywny jest znacznie bardziej zwarty, a realizacja zadania w tym wypadku bardzo prosta – interpreter języka Ruby zadba po prostu o wygenerowanie odpowiednich metod, analogicznych do tych z listingu 5.58.

Poza językami, które jak Ruby pozwalają umieszczać elementy deklaratywne w programie imperatywnym, istnieją także języki umożliwiające jedynie progra-

[19] Może poza nazwą funkcji, która może być jednak myląca i formalnie rzecz biorąc nie musi odpowiadać temu co rzeczywiście realizuje dana funkcja. Nazwa stanowi więc w rzeczywistości tylko pewną wskazówkę czego można się spodziewać, analizując poszczególne kroki algorytmu.

[20] Ruby pozwala używać na końcu nazw metod znaków, takich jak `=?!`. Znak `=` oznacza, że metoda może być użyta do przypisania wartości, np. `stud.imie="Józef"` zamiast `stud.imie=("Józef")`, czy jak w innych językach `stud.setImie("Józef")`.

Listing 5.58

```
1  class Student
2    def imie
3      @imie
4    end
5
6    def imie=(imie)
7      @imie = imie
8    end
9
10   def nazwisko
11     @nazwisko
12   end
13
14   def nazwisko=(nazwisko)
15     @nazwisko = nazwisko
16   end
17 end
```

Listing 5.59

```
1 class Student
2   attr_accessor :imie, :nazwisko  # do odczytu i zapisu
3   attr_reader :srednia            # tylko do odczytu
4   attr_writer :ocena              # tylko do zapisu
5 end
```

mowanie deklaratywne. Przykładem takiego języka szeroko stosowanego w praktyce jest SQL – język zapytań do baz danych, który bardziej szczegółowo zostanie opisany w rozdziale dotyczącym baz danych. Język ten służy do wyspecyfikowania, jakie dane w jakiej formie (ewentualnie jak przetworzone) należy zwrócić. Serwer bazy danych odpowiada za realizację zapytania i wygenerowanie danych w odpowiedniej formie.

Przykładem prostego zapytania SQL może być: `SELECT * FROM studenci WHERE srednia > 4.0`. Takie zapytanie zwróci wszystkie dane o studentach, którzy mają średnią powyżej 4.0. Należy zauważyć, że zapytanie nie definiuje, że należy przejść po całej tabeli zawierającej dane studentów, zaczynając od pierwszego wiersza, kończąc na ostatnim i z każdego wiersza wypisując wszystkie zawarte w nim dane. Polecenie definiuje jedynie żądany efekt. SQL jest przykładem języka, dla którego realizacja deklaratywnie opisanych zadań jest złożona i wymaga skomplikowanych algorytmów. Opis deklaratywny jest jednak wygodnym i zwartym sposobem uzyskiwania danych, stąd jego popularność.

Programowanie deklaratywne jest bardziej zwarte od imperatywnego, oddziela cel programu od sposobu realizacji tego celu. Jest ono jednakże trudniejsze do zrealizowania, często wymaga złożonych algorytmów realizujących zadeklarowane

cele. Często też programowanie imperatywne wydaje się łatwiejsze do przyswojenia – programista ma kontrolę nad każdym wykonywanym krokiem, nie musi się zastanawiać, czy dostatecznie precyzyjnie wyspecyfikował żądany efekt.

Na zakończenie warto zauważyć, że matematycy najczęściej posługują się opisem deklaratywnym. Przykładowo definicja

$$\{k \in \mathbb{Z}\colon k > 10 \wedge 2 \mid k\} \tag{5.1}$$

opisuje zbiór wszystkich parzystych liczb całkowitych większych od 10. Z oczywistych względów nie opisuje jednak, jak taki zbiór uzyskać, co nie przeszkadza dalszym rozważaniom na jego temat. Jak więc widać i jak pokazali przez stulecia matematycy, opis deklaratywny pozwala operować na abstrakcjach, których uzyskanie w kolejnych krokach postępowania imperatywnego nie jest możliwe.

5.8. Programowanie funkcjonalne

5.8.1. Wiadomości wstępne

Programowanie funkcjonalne albo *funkcyjne* (ang. *functional programming*) jest paradygmatem zasadniczo różnym od tego, jak najczęściej przedstawia się istotę programowania. Spośród opisanych w tym rozdziale, najbliższym programowaniu funkcjonalnemu przykładem jest język SQL. Można o nim przeczytać jako o języku „prawie funkcjonalnym" (ang. *nearly functional*)[21].

Paradgmat pogramowania funkcjonalnego jest oparty na *rachunku lambda*, który jest formalnym systemem matematycznym opisującym obliczenia. Wykonanie programu funkcjonalnego polega na obliczaniu wartości (jak dla wyrażenia matematycznego), a nie na wykonywaniu kolejnych kroków. W tym sensie podejście funkcyjne jest bardziej zbliżone do podejścia deklaratywnego niż imperatywnego.

Program napisany w języku funkcyjnym jest wyrażeniem, którego elementami są między innymi funkcje, rozumiane tak jak w matematyce, tj. zwracające wartości dla zadanych parametrów. Parametrami funkcji mogą być nie tyko wartości, ale także inne funkcje. Jest wiele odmian języków funkcyjnych i wiele podejść do tego paradygmatu, ale często stosowanymi rozwiązaniami są *leniwa ewaluacja* (obliczanie tylko tych części wyrażenia, które są konieczne do obliczenia jego wartości), brak skutków ubocznych obliczenia funkcji (jedynym skutkiem jest wartość funkcji), automatyczne zarządzanie pamięcią (wprowadzone później do takich języków, jak Java, Python czy Ruby), *przezroczystość odniesienia* (ang. *referential transparency*) pozwalająca zastępować fragmenty wyrażeń obliczonymi wcześniej wartościami.

[21] `http://www.haskell.org/haskellwiki/Introduction`

Przykładem podejścia funkcjonalnego jest używanie arkusza kalkulacyjnego, gdzie nie opisuje się kolejnych kroków obliczeń, które należy wykonać, ale zależności pomiędzy wartościami poszczególnych komórek. W codziennych, nieskomplikowanych zastosowaniach zależności te są proste, ale wprawni użytkownicy arkuszy kalkulacyjnych potrafią za ich pomocą wykonywać bardzo skomplikowane obliczenia. Wykonanie programu funkcyjnego można więc porównać do obliczania wartości (ewaluowania) skomplikowanego wyrażenia algebraicznego, w którym oblicza się wartości kolejnych podwyrażeń, wykonuje na nich następne operacje, aż do otrzymania wyniku. Oczywiście program funkcyjny jest znacznie bardziej skomplikowanym wyrażeniem niż wyrażenia algebraiczne i są w nim wykonywane znacznie bardziej zaawansowane operacje niż operacje algebraiczne. Przykładami języków funkcyjnych są Haskell, ML, OCaml, Lisp, Erlang.

Programowanie funkcjonalne, jakkolwiek zasadniczo różne od podejścia imperatywnego, jest bardzo użytecznym sposobem tworzenia programów. Pozwala na wysokopoziomowe podejście do programowania, a współczesne kompilatory generują bardzo efektywny kod. Programowanie funkcjonalne jest używane bardzo często w zastosowaniach, w których poprawność programu ma szczególnie istotne znaczenie, paradygmat ten ułatwia bowiem formalną weryfikację tej poprawności. Ze względu na to, że program ściśle funkcjonalny nie przechowuje stanu bardzo łatwo jest zrównoleglać takie programy, co we współczesnych zastosowaniach staje się coraz bardziej istotne.

Elementy programowania funkcjonalnego są obecne w wielu językach imperatywnych (np. w językach Python, Ruby, JavaScript) i sprawiają, że programowanie w nich jest wygodniejsze, bardziej zwarte i niezawodne. Wiele koncepcji realizowanych w językach funkcyjnych trafiło do programowania imperatywnego jako uznane tzw. dobre praktyki programowania. Przykłady niektórych elementów funkcyjnych obecnych w języku Ruby przedstawione są w kolejnych podrozdziałach.

5.8.2. Rekurencja

Rekurencja (rekursja) jest sposobem na powtarzanie operacji w językach funkcyjnych. Funkcja rekurencyjna to taka, która wywołuje samą siebie. Oczywiście wywołanie takie nie może być bezwarunkowe, gdyż doprowadziłoby do *nieskończonej rekurencji*, czyli zapętlenia programu. W językach funkcyjnych nie występuje iteracja, taka jak opisana w podrozdziale 5.3.6, a powtarzanie operacji wykonuje się właśnie za pomocą rekurencji. Operacja ta jest także możliwa w językach imperatywnych.

Rekurencja jest często stosowana w matematyce. Typowym przykładem funkcji definiowanej rekurencyjnie jest silnia (4.1). Aby obliczyć wartość $n!$, trzeba

obliczyć wartość $(n-1)!$. W przypadku silni rekurencja nie jest nieskończona, a warunkiem ją kończącym jest przypadek dla $n = 0$.

Rekurencyjną implementację funkcji silnia przedstawia listing 5.60. Jak widać, funkcja silnia wywołana z parametrem innym niż zero wywołuje samą siebie. Oczywiście funkcję tę można także zaimplementować bez użycia rekurencji (iteracyjnie): 5.61[22].

Listing 5.60

```
1 def silnia(n)
2   if n == 0
3     1
4   else
5     n * silnia(n-1)
6   end
7 end
```

Listing 5.61

```
1 def silnia(n)
2   w = 1
3   1.upto n do |i|
4     w = w * i
5   end
6   w
7 end
```

Silnia jest bardzo prostym przykładem funkcji rekurencyjnej i na ogół jest implementowana iteracyjnie, co w jej przypadku często przekłada się na efektywność. W programowaniu używa się rekurencji do zwartego i wygodnego implementowania znacznie bardziej skomplikowanych algorytmów, dla których implementacja iteracyjna jakkolwiek możliwa jest znacznie bardziej złożona i mało przejrzysta. Przykładem takiego algorytmu jest szybkie sortowanie, tzw. *quick sort*, czy też sortowanie przez scalanie (zob. rozdział 4.). Rekurencja dostarcza także wygodnego sposobu operowania na tzw. rekurencyjnych strukturach danych, takich jak drzewa czy listy (zob. rozdział 4.).

Większość współczesnych języków programowania, które umożliwiają tworzenie podprogramów (funkcji, procedur czy metod) pozwala stosować rekurencję.

[22] Każda z tych implementacji powinna dodatkowo sprawdzać, czy parametr n nie jest mniejszy od zera.

5.8.3. Skutki uboczne funkcji

Jedną z cech języków czysto funkcyjnych jest brak tzw. *efektów ubocznych* działania funkcji. Oznacza to, że jedyny skutkiem działania funkcji jest obliczony i zwrócony przez nią wynik. Inne języki pozwalają na efekty uboczne, pozwalając na przykład, aby funkcja zmieniała wartość jednej ze zmiennych globalnych, zapisywała coś do pliku, bazy danych itp.

Języki czysto funkcyjne wprowadzają inne od efektów ubocznych sposoby wczytywania danych do programu i zapisywania wyników. W językach imperatywnych byłaby jednak niemożliwa całkowita rezygnacja z efektów ubocznych. Niemniej jednak tam, gdzie to tylko możliwe, jest ona uważana za dobrą praktykę programowania. Oczywiście tworzy się funkcje, które piszą na ekran, do pliku czy zapisują dane do bazy danych. Jednak zdecydowanie odradza się tworzenie funkcji, które zmieniałyby wartości zmiennych globalnych, zaleca się także unikanie stosowania zmiennych globalnych tam, gdzie to tylko możliwe. Dzięki temu programy są łatwiejsze do zrozumienia, utrzymania i rozwijania, a funkcje są rzeczywiście autonomicznymi fragmentami programu, realizującymi swoje dobrze zdefiniowane zadania, niezależnie od miejsca, z którego zostały wywołane.

5.8.4. Funkcje jako parametry funkcji

W językach funkcyjnych parametrami funkcji mogą być nie tylko wartości czy zmienne, ale także inne funkcje. Funkcja może także zostać zwrócona jako wynik działania innej funkcji. Tę cechę języków funkcyjnych wprowadzono w języku Ruby. Funkcja (czy metoda) może pobierać jako parametr inną funkcję bądź fragment kodu (tzw. *blok kodu*) i wywoływać go podczas działania dowolną liczbę razy, podając wybrane przez siebie parametry.

Przykładem takiego wykorzystania funkcji jest zalecany w języku Ruby sposób wykonywania iteracji opisany w podrozdziale 5.3.6. W przytaczanym już wcześniej przykładzie z listingu 5.62 `times` jest to metoda wywoływana na obiekcie `10`[23]. Parametrem tej metody jest blok kodu `{ |i| puts i }`, który wypisuje na ekranie liczbę podaną mu jako parametr. Metoda `times` wywołuje ten blok kodu 10 razy, za każdym razem podając jako parametr do bloku kolejną liczbę naturalną.

Listing 5.62
```
1  10.times { |i| puts i }
```

Podobnie jest realizowana, także omówiona wcześniej, iteracja po tablicy (listing 5.63). Tablica również jest obiektem. Na tym obiekcie wywoływana jest

[23] Jak wspomniano wcześniej, w języku Ruby wszystko jest obiektem, stałe liczbowe także, można więc dla nich definiować metody i je wywoływać.

metoda `each`, parametrem metody `each` jest blok kodu `{ |a| puts a }`. Blok ten, podobnie jak ten z poprzedniego przykładu, powoduje wypisanie na ekranie wartości podanej w parametrze podczas wywołania. Metoda `each` wywołana na obiekcie tablicy odnajduje pierwszy element tej tablicy i wywołuje dla tego elementu podany jej w parametrze blok kodu, następnie odnajduje kolejny element tablicy i ponownie dla tego drugiego elementu wywołuje blok kodu itd.

Listing 5.63
```
1 t = [10, 30, 12, 15]
2
3 t.each { |a| puts a }
```

Listing 5.64 przedstawia przykład metody `upto` wywoływanej na obiekcie 5 i pobierającej zarówno argument liczbowy 12 (a więc zwykłą wartość), jak i blok kodu `{ |i| puts i }`. Metoda `upto` wywołuje podany blok kodu najpierw dla wartości 5, potem dla kolejnych, aż do wartości równej swojemu parametrowi, w tym wypadku 12. Podobnie działa metoda `downto` przedstawiona na listingu 5.28.

Listing 5.64
```
1 5.upto(12) { |i| puts i }
```

Język Ruby pozwala oczywiście samodzielnie tworzyć metody czy funkcje, których parametrami są bloki kodu bądź inne funkcje. Pozwala to na wygodne tworzenie zwartych programów.

5.8.5. Definiowanie nowych funkcji

Funkcje w językach funkcyjnych mogą w trakcie swojego działania tworzyć i wywoływać nowe funkcje. W rzeczywistości więc program w czasie wykonywania sam się modyfikuje, dzięki czemu może być bardziej zwarty, a przez to prostszy. Języki posiadające tę cechę są nazywane *dynamicznymi*. Ruby jest językiem dynamicznym – w czasie działania programu możliwe jest tworzenie kodu programu w języku Ruby, a następnie wykonywanie tego kodu. Takiej cechy nie mają języki zwane *statycznymi*, jak C czy Java. Tworzenie i uruchamianie kodu w czasie działania programu daje bardzo duże możliwości, wymaga jednak sporo uwagi, gdyż może być przyczyną trudnych do odszukania błędów. Jest to jednak cecha języka programowania dająca mu dużą siłę wyrazu i pozwalająca oszczędzić wiele pracy.

5.8.6. Polimorfizm

Podczas tworzenia programów występuje wiele sytuacji, w których tak samo nazywane operacje są wykonywane na wartościach różnych typów. Przykładowo,

na ekran można wypisywać zarówno liczby jak, i łańcuchy tekstowe. Podobnie, sortować można liczby całkowite, zmiennoprzecinkowe czy łańcuchy tekstowe. Wypisywanie liczb jest realizowane inaczej niż łańcuchów tekstowych. Wygodnie jednak, jeśli można te operacje nazwać tak samo i tak samo wywoływać, niezależnie od parametrów, których dotyczą. Jeśli język programowania pozwala na definiowanie tak samo nazwanych operacji dla różnych typów danych, to mówimy że pozwala realizować *polimorfizm*.

Łatwo zauważyć, że to, czy można wykonać operację wypisania na ekranie zależy od tego, czy daną wartość da się przedstawić w postaci tekstu. Podobnie, możliwość posortowania pewnych wartości zależy jedynie od tego, czy da się je porównywać tak, aby operacja porównania ustanawiała porządek wśród tych wartości. Można więc napisać jedną funkcję czy metodę wypisującą wartość oraz jedną funkcję sortującą wartości. Funkcję wypisującą można wywołać dla każdej wartości, która potrafi zamienić siebie na tekst, funkcję sortującą można wywoływać dla każdej listy wartości, które można pomiędzy sobą porównywać. Nieistotne jest, jaki dokładnie typ mają parametry podane wymienionym funkcjom, ważne jest to, żeby dało się na nich zrealizować wymagane operacje (zamiany na tekst bądź porównania).

Polimorfizm jest charakterystyczną cechą języków funkcyjnych, ale pojawia się nie tylko w tych językach. W różnym stopniu i w różny sposób polimorfizm jest wspierany np. przez C++ czy język Java. Język Ruby wspiera polimorfizm w dość oczywisty sposób: ponieważ zmienne nie mają określonych typów, można im przypisywać wartości dowolnego typu. Funkcje czy metody można więc wywoływać dla dowolnego typu, a sama funkcja w razie potrzeby może sprawdzić, czy wartość, którą dostała w parametrze, potrafi zrealizować wymagane działania (czy odpowiada na żądany komunikat, czyli czy definiuje odpowiednią metodę).

5.8.7. Stosowanie języków funkcyjnych

Języki funkcyjne, choć nie tak popularne, jak najbardziej znane C, C++ czy Java, są jednak szeroko stosowane. Rozwój tych języków ma też ogromny wpływ na postać tych najbardziej popularnych języków imperatywnych. Wiele z rozwiązań wprowadzonych do języków imperatywnych jako nowości było od dawna znanych w językach funkcyjnych. W wielu zastosowaniach języki funkcyjne są nie do zastąpienia, bardzo często łączy się je też z językami imperatywnymi. Bardzo pożyteczną, zwłaszcza w ostatnim czasie, cechą języków funkcyjnych jest fakt, że napisane w nich programy można bez dodatkowego narzutu programistycznego w sposób zupełnie naturalny uruchamiać na architekturach równoległych. Jest to możliwe, ponieważ stosunkowo łatwo jest stwierdzić, które wyrażenia można ewaluować niezależnie od siebie, a więc w razie potrzeby równocześnie na różnych procesorach. Programy funkcyjne w przeciwieństwie do impera-

tywnych nie przechowują zmieniającego się ciągle stanu (wartości zmiennych), nie pozwalają na efekty uboczne, a więc kolejność ewaluacji wyrażeń nie ma znaczenia. Nie ma zatem potrzeby tak kosztownego w programach imperatywnych synchronizowania poszczególnych, równolegle wykonywanych zadań.

Programowanie funkcyjne jest jednak różne od najczęściej uczonego i przedstawianego podejścia do tworzenia oprogramowania, a więc wymaga zmiany sposobu myślenia o programowaniu. Z tego powodu wielu programistów uważa je za niezbyt zachęcające. Wielu programistów mówi jednak, że nauczenie się programowania funkcyjnego powoduje poprawę jakości kodu tworzonego w językach imperatywnych.

5.9. Programowanie zdarzeniowe

Klasyczne podejście imperatywne zakłada, że istnieją kolejne instrukcje wykonywane sekwencyjne przez program. Przepływ programu może się rozgałęziać, np. dzięki instrukcjom warunkowym, może się zmieniać zależnie od danych wejściowych. Zasadniczo jednak to program „decyduje", jakie operacje są wykonywane i w jakich momentach należy odczytać dane użytkownika.

W *programowaniu zdarzeniowym* sytuacja jest odwrotna – to zdarzenia decydują o tym, jakie instrukcje i w jakiej kolejności są wykonywane. W przypadku wystąpienia zdarzenia program realizuje kod zdefiniowany jako obsługa tego zdarzenia. Takie podejście stosuje się na przykład przy programowaniu graficznego interfejsu użytkownika (GUI). Po uruchomieniu program obsługujący GUI nie wykonuje operacji, a jedynie oczekuje na zdarzenie. Takim zdarzeniem może być kliknięcie w przycisk, pojawienie się kursora myszy nad określoną ikoną, wybranie opcji z menu itp. Każdy z elementów GUI ma przypisany fragment programu obsługujący zdarzenia związane z tym elementem, kolejność realizacji tych fragmentów programu zależy od tego, w jakiej kolejności użytkownik wykorzystuje elementy interfejsu graficznego. Podobnie realizowane są *aplikacje www* oczekujące na zdarzenie polegające na otwarciu strony www znajdującej się pod określonym adresem.

Programowanie zdarzeniowe jest możliwe w wielu językach programowania i jego zastosowanie zależy w głównej mierze od tego, czy odpowiada ono charakterystyce realizowanego zadania.

5.10. Zadania

1. Napisać program wczytujący z klawiatury podstawę i wysokość trójkąta oraz obliczający jego pole.

2. Napisać program wczytujący z klawiatury liczbę całkowitą i wypisujący informację o tym, czy wczytana liczba jest parzysta czy nieparzysta.

3. Napisać program wczytujący z klawiatury długości odcinków sprawdzający i wypisujący, czy z odcinków o podanych długościach można zbudować trójkąt.

4. Napisać program rozwiązujący równanie kwadratowe.

5. Napisać program wczytujący 10 liczb i wypisujący ich średnią arytmetyczną.

6. Napisać program wczytujący liczby dopóki nie zostanie podane 0 i wypisujący ich średnią arytmetyczną. Średnia nie może uwzględniać podanego na końcu zera.

7. Napisać program wczytujący liczbę całkowitą do zmiennej `ilosc`, następnie wczytujący `ilosc` liczb do tablicy i wypisujący największą, najmniejszą z nich oraz ich średnią arytmetyczną.

8. Napisać program wczytujący z klawiatury do tablicy asocjacyjnej nazwę przedmiotu jako klucz oraz ocenę z tego przedmiotu jako wartość i wypisujący przedmioty z najlepszą i najgorszą oceną. Wczytywanie należy zakończyć, gdy użytkownik poda pustą nazwę przedmiotu.

9. Napisać funkcję `wczytaj` pytającą użytkownika, ile ocen chce wczytać, następnie wczytującą zadaną liczbę ocen do tablicy i zwracającą tę tablicę.

10. Napisać funkcję `srednia` pobierającą tablicę ocen (np. utworzoną przez funkcję z zadania 9.), obliczającą i zwracającą średnią ocen z tablicy.

11. Napisać funkcję wczytującą do tablicy asocjacyjnej dane studenta: imię, nazwisko, numer indeksu, listę ocen (w postaci tablicy).

12. Napisać funkcję, która wywołuje funkcję z zadania 11., następnie pyta, czy wczytać dane kolejnego studenta i w razie potrzeby ponownie wywołuje funkcję. Wczytane dane mają być zwracane przez funkcję w postaci tablicy.

13. Napisać funkcję pobierającą tablicę zwróconą przez funkcję z zadania 12. i zwracającą tablicę asocjacyjną opisującą studenta z najwyższą średnią.

14. Napisać definicję klasy `Osoba` zawierającej pola `@imie` oraz `@nazwisko` oraz metody pozwalające odczytywać i modyfikować te pola (*gettery* i *settery*) oraz zawierającą metodę `to_s`, która zwraca łańcuch tekstowy składający się z imienia i nazwiska osoby. Sprawdzić, jak zadziała wywołanie `puts o`, jeśli `o` będzie obiektem klasy `Osoba`.

15. Do klasy z zadania 14. dodać pole `@data_urodzenia`, metodę, która pozwala odczytywać to pole oraz konstruktor pozwalający ustawić wszystkie atrybuty nowego obiektu na żądane wartości.

16. Zdefiniować klasę `Student` dziedziczącą z klasy `Osoba` i dodającą pole `@numer_indeksu` oraz *getter* i *setter* tego pola oraz konstruktor pozwalający ustawić także to pole i wywołujący konstruktor klasy `Osoba`. Sprawdzić działanie odziedziczonej z klasy bazowej metody `to_s`.

17. Zdefiniować klasę `Pracownik` dziedziczącą po klasie `Osoba` z zadań 14. i 15., dodającą pole `@tytul_naukowy` wraz z `getterem` i `setterem` oraz konstruktor pozwalający ustawić także to pole i wywołujący konstruktor klasy `Osoba`. Zdefiniować na nowo metodę `to_s`, tak aby zwracała tekst zawierający na początku tytuł naukowny, a następnie tekst zwracany przez metodę `to_s` odziedziczoną z klasy bazowej.

18. Do klasy `Pracownik` z zadania 17. dodać atrybut `@placa` ze standardowym *getterem* oraz metodę `placa=` ustawiającą wartość tego atrybutu. Metoda ta ma generować wyjątek o treści *Pracownik nie może zarabiać mniej niż 0 zł* w przypadku próby przypisania wartości mniejszej od zera. Przetestować działanie metody.

19. Napisać program wykorzystujący klasę `Pracownik` z zadania 18. Program ma wczytywać z klawiatury kolejne dane pracownika i tworzyć nowy obiekt. Program musi przechwytywać wyjątek generowany przez kod z zadania 18. i wyświetlać komunikat błędu.

20. Ciąg Fibonacciego definiuje równanie (5.2). Napisać rekurencyjną implementację funkcji obliczającej `n`-ty wyraz tego ciągu:

$$F_n = \begin{cases} 0 & \text{dla } n = 0 \\ 1 & \text{dla } n = 1 \\ F_{n-1} + F_{n-2} & \text{dla } n > 1 \end{cases} \tag{5.2}$$

21. Napisać iteracyjną implementację funkcji obliczającej `n`-ty wyraz ciągu Fibonacciego. Zastanowić się, która z implementacji jest bardziej efektywna.

22. Napisać rekurencyjną implementację algorytmu *wyszukiwania binarnego*.

23. Napisać iteracyjną implementację algorytmu *wyszukiwania binarnego*. Która z implementacji jest prostsza i bardziej przejrzysta?

Rozdział 6.

Język C

Bartosz Jędrzejec, Jan Sadolewski, Sławomir Samolej

6.1. Wprowadzenie

Autorem języka C był Dennis Ritchie (1972 r.). Język C został zaprojektowany w celu zwiększenia przenośności kodu źródłowego pomiędzy różnymi platformami. Jego elastyczność pozwala na tworzenie w nim zarówno oprogramowania użytkowego, jak i systemów operacyjnych. Ze względu na swoją uniwersalność i przenaszalność język C zyskał ogromną popularność, co zaowocowało powstaniem wielu kompilatorów, które różnie interpretowały oryginalną koncepcję autora. Dlatego w 1989 roku Amerykański Narodowy Instytut Standaryzacji (ANSI) wydał standard ANSI C [17], w którym zawarł wytyczne dotyczące dopuszczalnych konstrukcji w języku oraz ich znaczenie (semantykę). W 1990 roku międzynarodowa organizacja ISO, która zajmuje się standaryzacją, ratyfikowała wersję ANSI C pod nazwą *ISO/IEC 9899:1990. Programming languages – C* [13]. Od tej pory standard ISO uznaje się za wzorcowy, jednakże z różnych względów producenci kompilatorów lekko odbiegają od tego standardu, dotyczy to zwłaszcza tworzenia oprogramowania pod specjalizowane mikrokontrolery. Drugą cechą charakterystyczną języka C jest możliwość wykonania prostego tłumaczenia na kod asemblera, ponieważ standardowe operacje i typy danych bezpośrednio nawiązują do operacji procesora dostępnych na danej platformie sprzętowej. Jeżeli taka bezpośrednia instrukcja procesora nie istnieje, to można ją zastąpić grupą innych instrukcji, których efekt działania byłby identyczny. Składnia języka C jest jednak podatna na błędy zapisane przez programistów, co jest jedną z jego wad. Oznacza to, że w języku C kompilator kompiluje dokładnie to, co jest poprawne składniowo i napisane przez programistę, nawet w przypadku gdy pierwotny zapis wydaje się bezsensowny. Stąd nawet nieuważnie postawiony średnik lub brakujące nawiasy w wywołaniu funkcji mogą się kompilować bez żadnych błędów, mogą być jednak przyczyną nieprawidłowego działania programu.

W rozdziale przedstawiono skrócony kurs języka C. Obejmuje on typowe zagadnienia dla początkujących programistów języka C, takie jak pierwszy program, wypisywanie i pobieranie danych z konsoli, podstawowe operacje i instrukcje. W dalszej części przedstawiono przydatne konstrukcje wykorzystujące tablice oraz pliki. Rozdział jest skróconą wersją części „Podstawy języka C" z pracy [15].

6.2. Podstawowe elementy języka C

6.2.1. Pierwszy program

W podrozdziale omówiono elementarne składniki języka C, takie jak: alfabet, zasady tworzenia nazw, nazwy zarezerwowane, typy danych, literały, stałe symboliczne, zmienne oraz wyrażenia. Na podstawie tych pojęć są budowane kolejne składniki języka.

Większość podręczników dotyczących podstaw programowania rozpoczyna się od programu, którego zadaniem jest wypisanie przykładowego tekstu *Hello World*. W niniejszej publikacji przykładowy program tego typu zamieszczono na listingu 6.1.

Listing 6.1

```
1 #include <stdio.h>
2
3 /*Mój pierwszy program*/
4
5 void main()
6 {
7   printf("Witaj Przyjacielu\n");
8 }
```

Efekt wykonania programu:

```
Witaj Przyjacielu
```

W programie wykorzystano standardową funkcję wyprowadzania danych na ekran o nazwie `printf`, zdefiniowaną w bibliotece `stdio.h`. Bibliotekę dołączono w linii 1 za pomocą dyrektywy `#include`. W linii 3 wstawiono komentarz rozpoczynający się znakami `/*` i zakończony układem znaków w odwrotnej kolejności `*/`. Komentarze można wstawiać w dowolnym miejscu programu, są one ignorowane przez kompilator. Każdy program w języku C musi mieć zdefiniowaną dokładnie jedną funkcję główną o nazwie `main` (nazwa ta jest zarezerwowana tylko dla tej funkcji). Jej definicja składa się z nagłówka (linia 5) oraz bloku ograniczonego klamrami (linie 6 i 8). W programie wywołano funkcję standardową `printf` (linia 7), która powoduje wyprowadzenie wskazanych argumentów na standardowe wyjście – ekran komputera.

6.2.2. Zestaw znaków dopuszczalnych

Każdy język programowania posługuje się szeroko rozumianym alfabetem, czyli zestawem znaków dopuszczalnych. W języku C zestaw ten obejmuje następujące elementy:

- duże litery alfabetu łacińskiego (od `A` do `Z`),
- małe litery alfabetu łacińskiego (od `a` do `z`),
- cyfry (od `0` do `9`),
- znaki specjalne `! * + \ "< # ( = | { > % ) ~ ; } / ^ - [ : , ? & _ ] ` . ␣` (*spacja*).

6.2.3. Nazwy

Elementy języka C, takie jak: stałe, zmienne, funkcje itp., są identyfikowane poprzez swoją nazwę. Nazwa (identyfikator) jest to ciąg literowo-cyfrowy zaczynający się od litery lub znaku podkreślenia. Nazwy wieloczłonowe możemy łączyć za pomocą znaku _ (podkreślenie) lub możemy stosować tzw. notację „wielbłądzią” (np. `MojaZmienna`). Język C w nazwach rozróżnia duże i małe litery, a długość nazw jest praktycznie nieograniczona, choć liczba rozróżnialnych znaków zależy od użytego kompilatora i zazwyczaj wynosi od 32 (starsze wersje) do 256 znaków (nowsze wersje).

Przykłady poprawnych nazw:
`pole`, `objetosc_walca`, `i`, `X1`, `delta`

Przykłady niepoprawnych nazw:
`1X`, `pole-powierzchni`, `$a`, `obwód_koła`

Tablica 6.1. Słowa kluczowe języka C

Słowa kluczowe języka C			
auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

6.2.4. Słowa kluczowe

Słowem kluczowym jest słowo zarezerwowane, którego znaczenie i użycie jest jednoznacznie określone semantyką języka. Zestaw słów kluczowych standardu ANSI C przedstawiono w tabl. 6.1. Znaczenie i sposób użycia przedstawionych słów kluczowych będzie omawiane w kolejnych podrozdziałach publikacji.

6.2.5. Typy danych

Ze względu na sposób przedstawienia danych w pamięci komputera język C ma wbudowane 4 podstawowe typy danych:

- `int` – reprezentuje liczbę całkowitą,
- `char` – reprezentuje pojedynczy znak (np. literę alfabetu),
- `float` – reprezentuje liczbę rzeczywistą (pojedynczej precyzji),
- `double` – reprezentuje liczbę rzeczywistą (podwójnej precyzji).

W języku C istnieją jeszcze dodatkowe typy danych:

- `void` – oznacza brak wartości zwracanej przez funkcję oraz brak parametrów funkcji,
- `enum` – opisuje typ wyliczeniowy,
- wskaźnikowy `*` – reprezentuje zmienne typu wskaźnikowego[1].

6.2.6. Literały i stałe symboliczne

Literał jest to jednostka leksykalna reprezentująca stałą ustaloną wartość wpisaną bezpośrednio w kodzie programu. Język C ma bogaty zestaw literałów zależnych od reprezentowanych typów danych. Można je podzielić w następujący sposób:

1. Literały całkowitoliczbowe
    - dziesiętne
      zestaw znaków: `0 1 2 3 4 5 6 7 8 9 + -`,
      przykłady: `125, -89, 0, +23`,

[1] Typ wskaźnikowy omówiono w podrozdziale 6.8.

- ósemkowe
  zestaw znaków: `0 1 2 3 4 5 6 7 + -` (pierwszym znakiem jest zero),
  przykłady: `0125, 0, 072, -012,`
- szesnastkowe
  zestaw znaków: `0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F + -` (pierwszymi znakami są zero i małe lub duże `x` – `0x`, `0X`),
  przykłady: `0x125, 0xaa, 0X12C, -0xc1b`.

2. Literały rzeczywiste
   zestaw znaków: `0 1 2 3 4 5 6 7 8 9 . + - e E`,
   przykłady: `1.2, 0., 0.5e3, -125E-2`.

3. Literały znakowe
   pojedynczy znak ujęty w apostrof,
   zestaw znaków:

   - cały repertuar „widocznych" znaków ASCII,
     przykłady: `'A', 'b', '+', '0'`,
   - znaki specjalne (tabl. 6.2)
     przykłady: `'\n', '\t', '\\'`,

   Każdy znak może być podany jako znak specjalny zawierający wartość kodu ASCII danego znaku w postaci ósemkowej lub szesnastkowej. Interpretacja jest przerywana po trzech znakach dla trybu ósemkowego lub po napotkaniu znaku niepasującego do ciągu ósemkowego albo szesnastkowego,
   przykłady: `'\113', '\x4b'`.

4. Literały łańcuchowe
   ciąg dowolnych znaków ujętych w cudzysłowy,
   przykłady: `"To jest lancuch"`, `"X + Y = Z"`, `"\n tekst"`.

5. Stałe symboliczne
   identyfikator zastępujący ciąg znaków definiowany w dyrektywie preprocesora:
   `#define` *`NAZWA tekst`*
   Przykłady definicji stałych symbolicznych pokazano na listingu 6.2.

Listing 6.2

```
1 #define PI 3.1415926
2 #define WZOR 4*a*b*b
```

Tablica 6.2. Najczęściej spotykane znaki specjalne

Sekwencja znaków	Wartość ASCII	Znaczenie
\a	7	sygnał dźwiękowy (BEL)
\b	8	cofnięcie o jeden znak (BS)
\t	9	tabulacja pozioma (HT)
\v	11	tabulacja pionowa (VT)
\n	10	nowa linia (LF)
\f	12	nowa strona (FF)
\r	13	powrót karetki (CR)
\"	34	cudzysłów
\'	39	apostrof
\?	63	znak zapytania
\\	92	kreska ukośna (backslash)
\0	0	znak pusty (null)

Listing 6.3

```
1  /*Stałe symboliczne*/
2  #include <stdio.h>
3  #define PI  3.1415926
4  #define PROMIEN  2
5  #define WYSOKOSC 4.4
6  void main()
7  {
8    float promien, wysokosc, objetosc;
9    promien = PROMIEN;
10   wysokosc = WYSOKOSC;
11   objetosc = PI * promien * promien * wysokosc;
12   printf("Objętość walca =");
13   printf("%f",objetosc);
14 }
```

Na listingu 6.3 zdefiniowano 3 stałe symboliczne (linie 2-4), które następnie użyto w programie głównym. Główną zaletą stosowania wyrażeń stałych jest możliwość ich zmian w definicjach, które są przekazywane do wszystkich miejsc programu korzystających z tych stałych. Dodatkowo nazwy przyjętych stałych są kojarzone z tradycyjnie używanymi w innych dziedzinach (np. stała `PI` - linia 2).

6.2.7. Zmienne

Zmienna jest to nazwa, której jest przypisany typ danych. W języku C każda zmienna użyta w kodzie programu musi zostać zadeklarowana. Deklaracje zmien-

nych są umieszczane bezpośrednio po nawiasie klamrowym ({) rozpoczynającym instrukcję złożoną[2].

Listing 6.4

```
1  /*Deklaracje zmiennych*/
2  #include <stdio.h>
3
4  void main()
5  {
6    char znak;
7    int tab[10], a;
8    float x = 2.5;
9    int tab[5] = {1, 5, 10, 2, 0};
10   char tekst[] = "To jest łańcuch";
11 }
```

Na listingu 6.4 pokazano deklaracje zmiennych różnego typu. Zmiennej przed pierwszym użyciem należy przydzielić wartość za pomocą instrukcji przypisania[3], lub wczytując wartość za pomocą funkcji standardowego wejścia[4]. Język C dopuszcza nadanie wartości początkowych w deklaracji zmiennych, co zostało zrealizowane w liniach 8-10. Kompilator języka C nie sygnalizuje przekroczenia zakresu wartości typu danej zmiennej, o prawidłowe przydzielenie typu musi zadbać programista.

6.2.8. Wyrażenia

Wyrażeniem jest literał, zmienna lub połączenie wymienionych elementów za pomocą operatorów (i nawiasów). Przykłady wyrażeń pokazano na listingu 6.5, w których wykorzystano m.in. operatory arytmetyczne, logiczne, porównania, przypisania oraz operator warunkowy. Użyte operatory szerzej będą opisane w podrozdziale 6.4.

Listing 6.5

```
1  x = 2 * a * b
2  a >= b
3  a > b ? a : b
4  a > b && a > 0
5  a == b || a != c
```

[2] Instrukcję złożoną omówiono w podrozdziale 6.5.2.
[3] Instrukcję przypisania omówiono w podrozdziale 6.5.3.
[4] Funkcje standardowego wejścia/wyjścia omówiono w podrozdziałach 6.3.1 i 6.3.2.

6.3. Operacje wejścia-wyjścia

6.3.1. Operacje standardowego wyjścia

W języku C standardowe operacje wejścia-wyjścia dostępne są w postaci funkcji zdefiniowanych w module `stdio.h` – podstawowe z nich będą omówione w kolejnych podrozdziałach.

Podstawowe funkcje standardowego wyjścia:

- `putchar`,
- `puts`,
- `printf`.

Funkcja `putchar` wyprowadza pojedynczy znak podany jako parametr. Prototyp funkcji[5] jest następujący:

```
int putchar(int character);
```

Parametrem funkcji `putchar(character)` jest wyrażenie całkowite. W przypadku prawidłowego zakończenia działania funkcja zwraca wartość kodu ASCII wyprowadzonego znaku, błędne wykonanie powoduje, że funkcja zwraca wartość `EOF`[6].

Na listingu 6.6 pokazano przykłady użycia różnych rodzajów parametrów aktualnych funkcji[7] `putchar`. W liniach 9 i 11 jest to zmienna, a w liniach 10 i 12 stała lub literał.

Efekt wykonania programu:

```
A B
```

Funkcja `puts` wyprowadza łańcuch znaków podany jako parametr. Prototyp funkcji jest następujący:

```
int puts(const char *str);
```

Parametrem funkcji `puts` jest stała lub zmienna łańcuchowa. W przypadku prawidłowego zakończenia działania funkcja zwraca nieujemną wartość całkowitą, natomiast w przypadku błędu zwraca `EOF`. Po wyprowadzeniu wskazanego łańcucha funkcja dodaje znak nowej linii (`'\n'`).

---

[5] Definicję prototypu funkcji podano w podrozdziale 6.6.

[6] `EOF` – stała zdefiniowana w module `stdio.h`, oznaczająca m.in. nieprawidłowy znak lub osiągnięty koniec pliku.

[7] Definicję parametrów aktualnych funkcji podano w podrozdziale 6.6

Listing 6.6
```
1  /* Przykłady zastosowania funkcji putchar */
2  #include <stdio.h>
3  #define SPACJA 32
4
5  int main()
6  {
7    char c = 'B';
8    int znak = 65;
9    putchar(znak);
10   putchar(SPACJA);
11   putchar(c);
12   putchar('\n');
13   return 0;
14 }
```

Na listingu 6.7 pokazano przykłady zastosowania funkcji `puts`. Wynik działania programu:

```
linia1
linia2
linia3
To praktyczna funkcja
```

Listing 6.7
```
1  /* Przykłady zastosowania funkcji puts */
2  #include <stdio.h>
3
4  void main()
5  {
6    char lancuch [] = "To praktyczna funkcja";
7    puts ("linia1\nlinia2\nlinia3");
8    puts (lancuch);
9  }
```

W linii 7 programu funkcja `puts` otrzymała jako parametr literał łańcuchowy zawierający znak specjalny ('\n'), w linii 8 parametrem była zaś zmienna łańcuchowa, która została zainicjowana w deklaracji (linia 6).

Funkcja `printf` służy do wyprowadzenia wartości numerycznych, znakowych i łańcuchowych. Prototyp funkcji:

```
int printf(const char * format, ...);
```

Pełny repertuar posługiwania się funkcją `printf()` zaprezentowano w pracy [15]. Na potrzebę niniejszego opracowania zdecydowano się na pokazanie wy-

branych jej możliwości. Po pierwsze funkcję stosuje się do wyprowadzania tekstów na konsolę. Po drugie pozwala ona na sformatowane wypisywanie zawartości zmiennych. Listing 6.8 pokazuje jej najpopularniejsze zastosowania.

Listing 6.8

```
1  /* Przykład zastosowania funkcji printf */
2  #include <stdio.h>
3
4  void main()
5  {
6    int k = 21101;
7    float f = 3.15;
8    printf("Zmienna całkowita: ");
9    printf("%d",k); printf("\n");
10   printf("Zmienna zmiennoprzecinkowa: %f\n",f);
11 }
```

Wynik działania programu:

```
Zmienna całkowita: 21101
Zmienna zmiennoprzecinkowa: 3.150000
```

Za pomocą funkcji na konsolę (standardowe wyjście) można wyprowadzać dowolny tekst, jak to pokazano w linii 8 listingu 6.8. Jeśli w cudzysłowie znajduje się sekwencja znaków `%d`, to w tym miejscu na standardowym wyjściu zostanie wypisana w sposób dziesiętny zawartość zmiennej typu całkowitego, którą wskazano po przecinku w funkcji `printf`. W podobny sposób można wypisać stan zmiennej typu zmiennopozycyjnego, wskazując ten fakt sekwencją znaków `%f` (por. linie 9 i 10). W linii 10 pokazano, jak wypisanie kolejnych tekstów, stanu zawartości zmiennych oraz znaku specjalnego można zawrzeć w jednym wywołaniu funkcji `printf`.

6.3.2. Operacje standardowego wejścia

Podstawowe funkcje standardowego wejścia:

- `getchar`,
- `gets`,
- `scanf`.

Funkcja `getchar` oznacza wczytywanie pojedynczego znaku zwracanego jako wynik działania funkcji. Prototyp funkcji jest następujący:

```
int getchar(void);
```

Funkcja zwraca kod wczytanego znaku po akceptacji klawiszem Enter. Na listingu 6.9 pokazano przykład wprowadzenia z klawiatury pojedynczego znaku oraz sprawdzenie poprawności działania funkcji getchar poprzez wyświetlenie wczytanej wartości za pomocą funkcji printf.

Listing 6.9
```
1  /* Przykłady zastosowania funkcji getchar */
2  #include <stdio.h>
3
4  void main()
5  {
6    char znak;
7    printf("Podaj znak do wyświetlenia: ");
8    znak = getchar();
9    printf("Podałeś znak ");
10   putchar(znak);
11   printf(" o kodzie ASCII %d\n",znak);
12 }
```

Wynik działania programu:

```
Podaj znak do wyświetlenia: A
Podałeś znak A o kodzie ASCII 65
```

Kolejną funkcją standardowego wejścia jest funkcja gets – wczytywanie jednej linii tekstu zakończonej znakiem Enter. Prototyp funkcji jest następujący:

```
char * gets(char * str);
```

Funkcja zapisuje wczytany łańcuch do zmiennej łańcuchowej str oraz

Listing 6.10
```
1  /* Przykłady zastosowania funkcji gets */
2  #include <stdio.h>
3
4  void main()
5  {
6    char tekst[256];
7    printf("Podaj łańcuch znaków:\n");
8    gets(tekst);
9    printf("Podałeś następujacy tekst:");
10   puts(tekst);
11 }
```

zwraca go jako wynik swojego działania i dołącza automatycznie znak końca

łańcucha (długość wczytywanego łańcucha musi być więc przynajmniej o 1 znak mniejsza niż zadeklarowana). Funkcja **nie sprawdza**, czy długość wczytywanego łańcucha nie przekracza zadeklarowanego rozmiaru, co może spowodować przepełnienie zarezerwowanego bufora i zakończenie programu z błędem. Zmusza to programistę do wnikliwej uwagi, zarówno jeśli chodzi o deklarację rozmiaru łańcucha, jak i wprowadzane dane. Na listingu 6.10 pokazano przykład użycia funkcji `gets`.

Wynik działania programu:

```
Podaj łańcuch znaków:
Mój tekst
Podałeś następujący tekst: Mój tekst
```

Program wczytuje łańcuch znaków do zmiennej `tekst`, której rozmiar został podany w deklaracji. Wczytany łańcuch znaków został następnie wypisany na ekranie z zastosowaniem funkcji `puts`.

Ostatnią z omawianych funkcji standardowego wejścia jest `scanf`. Funkcja ta umożliwia wprowadzanie danych różnego typu, a jej prototyp ma następującą postać:

```
int scanf(const char * format, ...);
```

Funkcja wczytuje dane ze standardowego wejścia zgodnie ze wskazanym `formatem` i przechowuje pod adresami zmiennych, podanymi jako parametry „`...`”. Funkcja zwraca liczbę poprawnie wprowadzonych wartości, w przypadku błędu (niedopasowania formatu do wczytanej wartości) natychmiast kończy działanie. Funkcja ta, podobnie jak funkcja `printf`, dysponuje bogatym repertuarem

Listing 6.11

```
1  /* Przykłady zastosowania funkcji scanf */
2  #include <stdio.h>
3
4  void main()
5  {
6    int a;
7    float x;
8    printf("Podaj jedną wartość całkowitą: ");
9    scanf("%d",&a);
10   printf("Odczytano liczbę: %d\n",a);
11   printf("Podaj jedną wartość rzeczywistą: ");
12   scanf("%f",&x);
13   printf("Odczytano liczbę: %f\n",x);
14 }
```

wywołań, który został omówiony między innymi w pracy [15]. Na listingu 6.11 pokazano przykład użycia funkcji `scanf` dla danych różnego typu.

Wynik działania programu jest następujący:

```
Podaj jedną wartość całkowitą: 3
Odczytano liczbę: 3
Podaj jedną wartość rzeczywistą: 2.4
Odczytano liczbę: 2.400000
```

Funkcja „oczekuje”, że użytkownik w odpowiednim momencie działania programu wprowadzi na standardowe wejście liczbę w odpowiednim formacie. Jeśli pierwszym parametrem funkcji jest łańcuch `%d`, to funkcja „zakłada”, że użytkownik będzie wprowadzał liczbę całkowitą (por. linię 9 listingu 6.11). Podana przez użytkownika liczba zostanie wprowadzona do zmiennej wskazanej jako drugi parametr funkcji. Aby funkcja zadziałała poprawnie, należy przed nazwą zmiennej koniecznie wpisać znak `&`. W podobny sposób (por. linię 12 listingu 6.11) wskazuje się funkcji, że ma ona pobrać dane do zmiennej typu zmiennoprzecinkowego.

6.4. Operatory

6.4.1. Grupy operatorów

Operator określa działanie, jakie należy wykonać na obiektach (stałe lub zmienne) podanych jako argumenty. Ze względu na liczbę argumentów operatory dzielimy na jedno-, dwu- lub trójargumentowe. Operatory można również pogrupować według tzw. cech funkcjonalnych, wyróżniając:

- operatory arytmetyczne,
- operatory porównania,
- operatory logiczne,
- operatory bitowe,
- operatory przypisania,
- operatory unarne,
- operator rozmiaru,
- operator konwersji,
- operator warunkowy,

- operator przecinkowy,
- operatory wskazywania.

Wybrane grupy operatorów będą omówione w kolejnych podrozdziałach.

6.4.2. Operatory arytmetyczne

W języku C występują następujące operatory arytmetyczne:

`+` dodawanie,

`-` odejmowanie,

`*` mnożenie,

`/` dzielenie,

`%` reszta z dzielenia liczb całkowitych.

Operatory arytmetyczne należą do grupy operatorów dwuargumentowych, a typ wyniku ich działania zależy od typów argumentów. Należy zwrócić uwagę, że w przypadku dzielenia dwóch argumentów całkowitych dokonuje się dzielenie całkowite. Na listingu 6.12 pokazano przykłady użycia operatorów arytmetycznych.

Listing 6.12

```
1  /* Przykłady zastosowania operatorów arytmetycznych*/
2  #include <stdio.h>
3
4  void main ()
5  {
6    int a = 10, b = 3, c;
7    float x = 5.5, y = 1.5, z;
8    printf("a = %d,",a); printf(" b = %d\n",b);
9    c = a * b;
10   printf("a * b = %d\n",c);
11   c = a / b;
12   printf("a / b = %d\n",c);
13   c = a % b;
14   printf(" a %% b = %d\n",c);
15   printf("x = %f, y = %f\n",x,y);
16   z = x / y + y;
17   printf("x / y + y = %f\n",z);
18  }
```

Wynik działania programu:

```
a = 10, b = 3
```

```
a * b = 30
a / b = 3
a % b = 1
x = 5.500000, y = 1.500000
x / y + y = 5.166667
```

W liniach 9, 11 i 13 pokazano działanie operatorów arytmetycznych w połączeniu z argumentami całkowitymi, w linii 16 zaś działanie operatora dzielenia dla argumentów rzeczywistych.

6.4.3. Operatory porównania

Język C dysponuje następującymi sześcioma operatorami porównania:

- `==` równe,
- `!=` różne,
- `<` mniejsze,
- `<=` mniejsze lub równe,
- `>` większe,
- `>=` większe lub równe.

Operatory te należą do grupy operatorów dwuargumentowych i są podstawowymi składnikami wyrażeń logicznych oraz warunkowych. Wynikiem ich działania jest zawsze wartość całkowita: 1 (prawda) lub 0 (fałsz). Na listingu 6.13 pokazano przykłady użycia operatorów porównania i tworzenia wyrażeń warunkowych.

Listing 6.13

```
1  /* Przykłady zastosowania operatorów porównania*/
2  #include <stdio.h>
3
4  void main()
5  {
6    int a = 10, b = 15, c = 20;
7    printf("a > 0             wynik: %d\n",a > 0);
8    printf("a == b            wynik: %d\n",a == b);
9    printf("(a + b) > c       wynik: %d\n",(a + b) > c);
10   printf("(a - b) != (c - a) wynik: %d\n",(a - b) != (c - a));
11   printf("c <= b            wynik: %d\n",c <= b);
12 }
```

Wynik działania programu jest następujący:

```
a > 0             wynik: 1
```

```
a == b              wynik: 0
(a + b) > c         wynik: 1
(a - b) != (c - a) wynik: 1
c <= b              wynik: 0
```

Przykład 6.13 demonstruje wynik działania operatorów porównania. Jak można zauważyć, operandami tych operatorów mogą być zarówno literały, zmienne, jak również wyrażenia.

6.4.4. Operatory logiczne

Operatory logiczne pozwalają wraz z operatorami porównania na tworzenie rozbudowanych wyrażeń logicznych. Do dyspozycji programistów języka C dostępne są dwa operatory dwuargumentowe (AND i OR) oraz jeden jednoargumentowy (NOT):

- `&&` logiczne AND (i) /koniunkcja/,
- `||` logiczne OR (lub) /alternatywa/,
- `!` logiczne NOT (nie) /negacja/.

Listing 6.14

```
1  /* Przykłady zastosowania operatorów logicznych*/
2  #include <stdio.h>
3
4  int main ()
5  {
6   int a = 10, b = 15, c = 20;
7   printf("((a + b) >= c) && (b == 1)      wynik:");
8   printf(" %d\n",((a + b) >= c) && (b == 1));
9   printf("((c - a) < b) || (2 * a > c)    wynik:");
10  printf(" %d\n",((c - a) < b) || (2 * a > c));
11  printf("!((a + b) > c)                  wynik:");
12  printf(" %d\n",! ((a + b) > c));
13  printf("((a + b) && c) || ! (b == 1)    wynik:");
14  printf(" %d\n",((a + b) && c) || ! (b == 1));
15  printf("! (a + c - 2 * b)               wynik:");
16  printf(" %d\n",! (a + c - 2 * b));
17 }
```

Podobnie jak w przypadku operatorów porównania, wynikiem działania operatorów logicznych jest wartość całkowita (0 lub 1). Operator NOT zamienia wartość wyrażenia różną od 0 na 0 (fałsz) oraz 0 na 1 (prawda). Podczas obliczania wartości wyrażeń z udziałem operatorów logicznych AND i OR kompilator wykorzystuje ich własności matematyczne, tzn.:

- przy operatorze AND – jeżeli lewy operand ma wartość 0, prawy nie jest obliczany,
- przy operatorze OR – jeżeli lewy operand ma wartość 1, prawy nie jest obliczany.

Ten sposób działania rzutuje na to, że wszelkie zmiany, które miałyby być wykonane w prawym operandzie (zgodnie z zapisem programu) zostaną pominięte. Program 6.14 przedstawia przykłady rozbudowanych wyrażeń logicznych.

Wynik działania programu:

```
((a + b) >= c) && (b == 1)    wynik: 0
((c - a) < b) || (2 * a > c)  wynik: 1
! ((a + b) > c)               wynik: 0
((a + b) && c) || ! (b == 1)  wynik: 1
! (a + c - 2 * b)             wynik: 1
```

6.4.5. Operatory wskazywania

Do grupy tej należą cztery następujące operatory:

`&` adresu,

`*` adresowania pośredniego,

`.` składowej,

`->` wskaźnikowy składowej.

Operator adresu oraz adresowania bezpośredniego są operatorami jednoargumentowymi, natomiast operator składowej i wskaźnikowy składowej są operatorami dwuargumentowymi. Zostaną one omówione w podrozdziale dotyczącym wskaźników (por. podrozdział 6.8).

6.5. Instrukcje

6.5.1. Podział instrukcji

Instrukcją nazywamy składniowo poprawny ciąg znaków zakończony średnikiem, którą można interpretować jako polecenie przekazane do wykonania dla kompilatora. Instrukcje dzielimy na:

- instrukcję złożoną,
- instrukcję przypisania,

- instrukcję warunkową (`if`),
- instrukcję pętli (`while`, `do-while`, `for`),
- instrukcję zaniechania (`break`),
- instrukcję kontynuacji (`continue`),
- instrukcję wyboru (`switch`),
- instrukcję wywołania funkcji,
- instrukcję powrotu (`return`),
- instrukcję skoku (`goto`),
- instrukcję pustą.

W kolejnych podrozdziałach zostaną omówione tyko niektóre instrukcje, kluczowe z punktu widzenia konstruowania podstawowych programów.

6.5.2. Instrukcja złożona

Instrukcją złożoną, nazywaną instrukcją bloku lub grupującą, jest ciąg deklaracji oraz instrukcji ograniczony nawiasami klamrowymi `{` i `}`. Instrukcję złożoną stosujemy wszędzie tam, gdzie składnia języka C wymaga pojedynczej instrukcji, a algorytm rozwiązywanego problemu kilku działań. Ogólny zapis instrukcji złożonej jest następujący:

```
{
 ciąg_deklaracji
 ciąg_instrukcji
}
```

6.5.3. Instrukcja przypisania

Instrukcja przypisania nadaje zmiennej stojącej po lewej stronie operatora przypisania wartość wyrażenia po prawej stronie. Ma ona następującą postać:

```
zmienna operator_przypisania wyrażenie;
```

Język C dopuszcza wielokrotną instrukcję przypisania, która jest wykonywana zgodnie z łącznością operatorów przypisania od strony prawej do lewej. Przykład użycia operatora przypisania pokazano na listingu 6.15. Wynik działania programu jest następujący:

```
a = 5
```

```
b = 5
c = 5
```

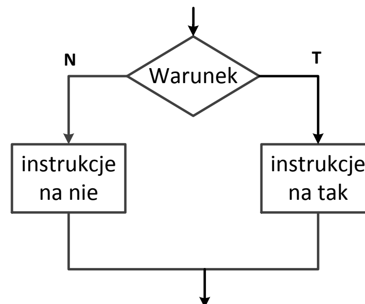
Listing 6.15

```
1  /* Przykład instrukcji przypisania*/
2  #include <stdio.h>
3
4  void main()
5  {
6    int a, b, c;
7    c = 5; //pojedyncza instrukcja przypisania
8    a = b = c; //wielokrotna instrukcja przypisania
9    printf("a = %d\n",a);
10   printf("b = %d\n",b);
11   printf("c = %d\n",c);
12 }
```

W programie zmiennej `c` nadano wartość 5 w pojedynczej instrukcji przypisania (linia 7). Następnie zainicjowano zmienne `a` i `b` w wielokrotnej instrukcji przypisania (linia 8) wartością zmiennej `c`.

6.5.4. Instrukcja warunkowa

Instrukcja warunkowa jest zaliczana do instrukcji decyzyjnych, które po ob-



Rys. 6.1. Schemat blokowy instrukcji warunkowej

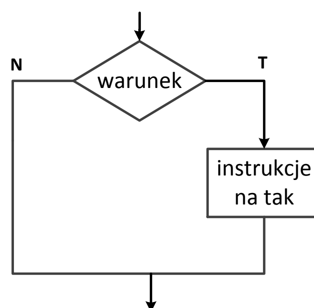
liczeniu warunku rozstrzygają, jaka grupa instrukcji będzie wykonywana. Pełny zapis instrukcji warunkowej `if` jest następujący:

```
if (warunek)
{
 instrukcje na tak
}
else
{
 instrukcje na nie
}
```

Schemat blokowy odpowiadający tej instrukcji pokazano na rys. 6.1. Gdy nie ma działania po członie `else`, to można się posłużyć skróconą wersją polecenia `if`:

```
if (warunek)
{
 instrukcje na tak
}
```

Schemat blokowy tej wersji instrukcji przedstawiono na rys. 6.2 Przykład



Rys. 6.2. Schemat blokowy
skróconej instrukcji warunkowej

działania instrukcji warunkowej `if` pokazano na listingu 6.16. Program demonstruje rozwiązanie równania kwadratowego w zakresie liczb rzeczywistych. W programie zastosowano zagnieżdżone instrukcje warunkowe. Wyniki działania programu są następujące:

Przykład 1.

```
Podaj współczynniki równania kwadratowego a, b, c:
4 4 1
x1 = x2 = -0.50
```

Przykład 2.

```
Podaj współczynniki równania kwadratowego a, b, c:
```

```
1 4 -12
x1 = -6.00 x2 = 2.00
```

Przykład 3.

```
Podaj współczynniki równania kwadratowego a, b, c:
5 7 20
Brak rozwiązania w dziedzinie liczb rzeczywistych
```

Listing 6.16

```
1  /* Przykład instrukcji warunkowej*/
2  #include <stdio.h>
3  #include <math.h>
4
5  int main ()
6  {
7    float a, b, c, delta, x1, x2;
8    printf("Podaj współczynniki równania kwadratowego a, b, c:\n");
9    scanf("%f%f%f",&a,&b,&c);
10   delta = b * b - 4 * a * c;
11   if(delta > 0)
12   {
13     x1 = (-b -sqrt(delta)) / (2 * a);
14     x2 = (-b + sqrt(delta)) / (2 * a);
15     printf("x1 = %f\t",x1);
16     printf("x2 = %f\n",x2);
17   }
18   else if(delta == 0)
19        {
20          x1 = x2 = -b/(2*a);
21          printf("x1 = %f\n",x1);
22        }
23        else
24        {
25          printf("Brak rozwiązania w dziedzinie liczb rzeczywi\
26 stych\n");
27        }
28 }
```

6.5.5. Instrukcje pętli

Instrukcja pętli w językach programowania umożliwia wielokrotne wykonanie instrukcji do momentu zmiany wartości wyrażenia sterującego pętlą. W języku C występują trzy instrukcje pętli:

- pętla `while`,

- pętla `do-while`,
- pętla `for`.

Pętla `while`

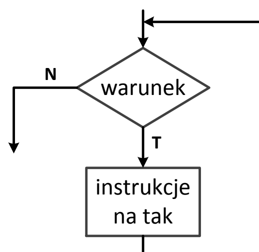
W pętli `while` lista instrukcji na „tak” wykonywana jest dopóki warunek jest prawdziwy (różny od 0). W szczególnym przypadku (warunek podczas pierwszej iteracji równa się 0) lista instrukcji nie wykona się ani razu. Ogólna postać pętli `while` jest następująca:

```
while (warunek)
{
 instrukcje na tak
}
```

Schemat blokowy instrukcji pętli `while` pokazano na rysunku 6.3. Najpierw jest sprawdzany warunek działania pętli, a następnie jest wykonywana lista instrukcji na „tak”. Na listingu 6.17 pokazano przykłady zastosowania pętli `while`. Wynik działania programu jest następujący:

```
znak    kod ASCII
-------------
  a        97
  b        98
  c        99
  d       100
```

W programie zastosowano dwie pętle `while`. W pierwszej pętli jest drukowanych 20 znaków '`-`' tworzących linię pokreślenia, w drugiej zaś jest wypisywany znak wraz z jego kodem ASCII, do momentu wciśnięcia klawisza enter.



Rys. 6.3. Schemat blokowy pętli `while`

Listing 6.17

```
1  /* Przykłady instrukcji pętli while*/
2  #include <stdio.h>
3  #include <conio.h>
4
5  void main()
6  {
7    int k = 1;
8    char zn;
9    printf("\nznak\tkod ASCII\n");
10   k=1;
11   while(k <= 20)
12   {
13     putchar('-');
14     k++;
15   }
16   while((zn = getch()) != '\r')
17   {
18     putchar('\n'); putchar(zn);
19     printf(" %d",zn);
20   }
21 }
```

Pętla `do-while`

Lista instrukcji znajdująca się pod pętlą `do-while` jest wykonywana dopóki warunek występujący po słowie `while` jest prawdziwy. Lista ta wykona się w odróżnieniu od pętli `while` przynajmniej jeden raz. Ogólna postać pętli `do-while` jest następująca:

```
do
{
 instrukcje
}
while (warunek);
```

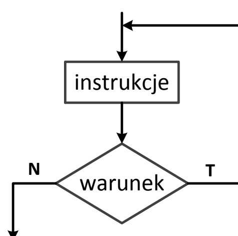
Schemat blokowy instrukcji pętli `do-while` pokazano na rys. 6.4. Najpierw wykonuje się lista instrukcji, a następnie jest sprawdzany warunek. Na listingu 6.18 przedstawiono przykłady zastosowania pętli `do-while`. Wynik działania programu jest następujący:

```
Podaj liczbę wyrazów ciągu > 0:-1
Podaj liczbę wyrazów ciągu > 0:4
Suma szeregu wynosi: 2.083333
```

Listing 6.18

```
1  void main()
2  {
3    int i,n;
4    double suma = 0;
5    do
6    {
7      printf("Podaj liczbę wyrazow ciągu > 0: ");
8      scanf("%d",&n);
9    }
10   while(n <= 0);
11   i = 1;
12   do
13   {
14     suma = suma + (1.0 / i);
15     i++;
16   }
17   while(i <= n);
18   printf("Suma szeregu wynosi: %f\n",suma);
19 }
```

Program 6.18 liczy sumę szeregu harmonicznego $1+\frac{1}{2}+\frac{1}{3}+...+\frac{1}{n}$. W pierwszej pętli `do-while` jest wczytywana liczba wyrazów szeregu (musi być ona dodatnia), w drugiej zaś jest obliczana suma szeregu.



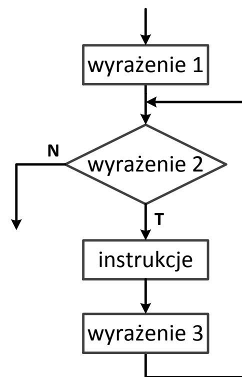
Rys. 6.4. Schemat blokowy pętli `do-while`

Pętla `for`

Pętla `for` w języku C jest trzecią instrukcją organizacji cyklu. Ogólna postać pętli `for` jest następująca:

```
for(wyrażenie1; wyrażenie2; wyrażenie3)
{
 instrukcje
}
```

Schemat blokowy instrukcji pętli `for` pokazano na rys. 6.5. W pierwszej kolej-



Rys. 6.5. Schemat blokowy pętli `for`

ności jest obliczana wartość *wyrażenia 1*, następnie jest sprawdzany warunek (*wyrażenie 2*). Jeżeli warunek jest prawdziwy, wykonywana jest lista instrukcji, a potem obliczane *wyrażenie 3*. Podobnie jak w przypadku pętli `while`, *instrukcje* i *wyrażenie 3* mogą nie wykonać się ani razu. W pętli *wyrażenie 1*, *wyrażenie 2* lub *wyrażenie 3* mogą być puste, ale znaki średnika muszą wystąpić. Brak *wyrażenia 2* w pętli `for` oznacza wartość zawsze prawdziwą.

Zapis instrukcji `for` można przedstawić w sposób równoważny, korzystając z pętli `while`:

```
wyrażenie1;
while (wyrażenie2)
{
 instrukcje
 wyrażenie3;
}
```

Na listingach 6.19 i 6.20 pokazano przykłady zastosowania pętli `for`. Wynik działania programu 6.19 jest następujący:

```
Podaj n>0: 4
Proszę podać 4 liczb
```

```
10 20 30 40
Średnia arytmetyczna wynosi 25
```

Listing 6.19

```
1  /* Przykład instrukcji pętli for*/
2  #include <stdio.h>
3
4  int main ()
5  {
6    int i,n;
7    float suma,a;
8    printf("Podaj n>0: ");
9    scanf("%d",&n);
10   if(n > 0)
11   {
12     printf("Proszę podać %d liczb\n",n);
13     for(i = 1,suma = 0; i <= n; i++)
14     {
15       scanf("%f",&a);
16       suma += a;
17     }
18     printf("Średnia arytmetyczna wynosi: %f\n",suma / n);
19   }
20   else
21   {
22     printf("Błędne n\n");
23   }
24 }
```

Program oblicza średnią arytmetyczną dla n liczb wprowadzanych z klawiatury. W przypadku podania błędnego n (<1) program kończy działanie, wyświetlając odpowiedni komunikat.

Drugi z przykładów zastosowania pętli `for` (listing 6.20) pokazuje możliwość ich zagnieżdżania, co oznacza, że wśród listy instrukcji w pętli zewnętrznej występuje ponownie pętla `for`. Wynik działania programu jest następujący:

```
Podaj liczbę wierszy: 5
Podaj liczbę kolumn: 10
Podaj znak: *

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

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

Listing 6.20

```
1  /* Przykład instrukcji pętli for*/
2  #include <stdio.h>
3
4  void main()
5  {
6    int i, j, n, m;
7    char znak;
8    printf("Podaj znak: ");
9    znak = getchar();
10   printf("Podaj liczbę wierszy: ");
11   scanf("%d",&n);
12   printf("Podaj liczbę kolumn: ");
13   scanf("%d",&m);
14   if(n > 0 && m > 0 && znak > 32 && znak < 127)
15   {
16     for(i = 1; i <= n; i++)
17     {
18       for(j = 1; j <= m; j++)
19         putchar(znak);
20       putchar('\n');
21     }
22   }
23   else
24   {
25     printf("Błędne dane\n");
26   }
27 }
```

Program wyświetla prostokąt o n wierszach i m kolumnach wypełniony znakiem podanym z klawiatury. Po wczytywaniu wartości wierszy i kolumn program sprawdza prawidłowe ich wprowadzenie (wartości większe od 0) oraz dodatkowo, czy zmienna znak jest znakiem widocznym na ekranie (linia 15).

6.5.6. Instrukcja zaniechania

Instrukcja zaniechania `break` umożliwia wcześniejsze opuszczenie pętli przed zakończeniem pełnego jej cyklu. Zapis instrukcji jest następujący:

```
break;
```

Schemat blokowy użycia instrukcji `break` pokazano na rys. 6.6. Na listingu 6.21 pokazano przykład działania instrukcji `break`. Wynik działania

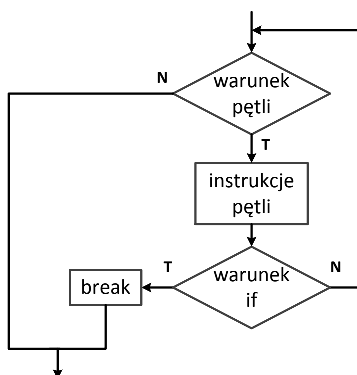
programu jest następujący:

```
a=10
a=-5
a=8
a=0
Suma podanych liczb wynosi: 13
```

Program liczy sumę liczb wprowadzanych z klawiatury zakończonych zerem.

Listing 6.21

```
1  /* Przykład instrukcji break*/
2  #include <stdio.h>
3
4  void main()
5  {
6    int a,suma;
7    suma = 0;
8    while(1)
9    {
10     printf("a=");
11     scanf("%d",&a);
12     if(a == 0)
13       break;
14     suma += a;
15   }
16   printf("Suma podanych liczb wynosi: %d\n",suma);
17 }
```



Rys. 6.6. Schemat blokowy użycia instrukcji zaniechania `break`

Pętla `while` ma ustawiony warunek na 1 (prawda), zakończenie pętli umożliwia instrukcja `break`.

6.5.7. Instrukcja wyboru

Instrukcja wyboru umożliwia wybór ścieżki działania programu w przypadku sytuacji wielowariantowych. Ogólna postać instrukcji `switch-case` jest następująca:

```
switch(wyrażenie)
{
 case wyrażenie_stałe_1:
      lista instrukcji 1
      break;
 case wyrażenie_stałe_2:
      lista instrukcji 2
      break;
      ...
 case wyrażenie_stałe_n:
      lista instrukcji n
      break;
 default:
      awaryjna lista instrukcji
      break;
}
```

Wyrażenie po komendzie `switch`, jak również wyrażenia stałe po `case` są wartościami całkowitymi. Każda z grup instrukcji może kończyć się instrukcją zaniechania `break`. Jeżeli instrukcja ta zostanie pominięta, to sterowanie programu przejdzie do następnych instrukcji umieszczonych poniżej następnego `case`. Schemat blokowy instrukcji wyboru `switch` pokazano na rys. 6.7. W pierwszej kolejności wyliczana jest wartość wyrażenia występującego po słowie `switch`, a następnie sterowanie jest przeniesione do listy instrukcji członu `case`, po którym występujące wyrażenie stałe równe obliczonemu. W przypadku gdy żadne wyrażenie stałe nie jest równe obliczonemu sterowanie przekazane jest do awaryjnej listy instrukcji.

Na listingu 6.22 pokazano przykład działania instrukcji `switch`. Wynik działania programu jest następujący:

```
Podaj liczbę:
2
Jedynka albo dwójka
```

Listing 6.22

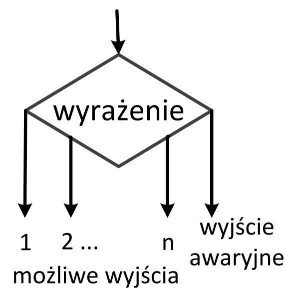
```
1  /* Przykład instrukcji switch*/
2  #include <stdio.h>
3
4  void main()
5  {
6    int a=1;
7    while(a!=0)
8    {
9       printf("Podaj liczbę:\n");
10      scanf("%d",&a);
11      switch(a)
12      {
13         case 1:
14         case 2: printf("\tJedynka albo dwójka\n");
15                 break;
16         case 3:
17         case 4: printf("\tTrójka albo czwórka\n");
18                 break;
19         default:  printf("\tNierozpoznana liczba\n");
20      }
21   }
22 }
```

```
Podaj liczbę:
4
Trójka albo czwórka
Podaj liczbę:
0
Nierozpoznana liczba
```

Program rozpoznaje cztery liczby: 1, 2, 3, 4, dla których wyświetla komunikat: „Jedynka albo dwójka”, jeśli użytkownik wprowadził na wejście 1 lub 2, „Trójka albo czwórka”, jeśli użytkownik wprowadził na wejście 3 lub 4. Odczyt, rozpoznawanie liczb i wypisywanie powtarzane jest w pętli. Kompilator po znalezieniu odpowiedniego członu `case` wykonuje instrukcje do momentu napotkania instrukcji `break` lub w przypadku jej braku do klamry zamykającej instrukcję `switch`. Program kończy działanie, gdy zostanie podana wartość 0.

6.5.8. Instrukcja wywołania funkcji

Instrukcja wywołania funkcji umożliwia wykonanie zapisanego uprzednio algorytmu dla bieżących parametrów. Są używane następujące postacie wywołania tej instrukcji:



Rys. 6.7. Schemat blokowy
instrukcji `switch`

- w instrukcji przypisania (np. `znak = getchar();`),
- samodzielnie (np. `puts("Tekst");`).

Techniki wywoływania i tworzenia funkcji szerzej omówiono w podrozdziale 6.6.

6.5.9. Instrukcja powrotu

Instrukcja powrotu kończy wykonanie funkcji. Są używane następujące dwie postacie wywołania:

- `return;`
- `return wyrażenie;`

Pierwsza postać nie udostępnia rezultatu funkcji (jedynie powoduje jej zakończenie), postać druga natomiast udostępnia rezultat w postaci wyniku wyrażenia. Zastosowania instrukcji omówiono w podrozdziale 6.6.

6.5.10. Instrukcja pusta

Instrukcja pusta nie powoduje żadnych skutków w wykonaniu programu i jest stosowana w celu poprawienia czytelności programu. Ogólna postać instrukcji jest następująca:

```
;
```

Instrukcja `;` jest wykorzystywana, gdy kompilator wymaga instrukcji, a algorytm nie wykonuje żadnego działania. Należy ją stosować ostrożnie, gdyż

może być przyczyną trudnych do wykrycia błędów. Przykłady błędnego użycia przedstawiają się następująco:
`while(1);` – pusta pętla nieskończona,
`for(i=1;i<n;i++);` – pętla zwiększająca tylko `i` od `1` do `n`,
`if(n<0);` – wykonuje się instrukcja pusta, gdy zostanie spełniony warunek.

6.6. Funkcje

6.6.1. Zastosowania funkcji

Funkcja jest to wydzielona część programu wykonująca określony algorytm. Zastosowanie funkcji umożliwia skrócenie programu głównego (funkcji `main`) i poprawienie czytelności kodu. Przetestowane funkcje mogą być dołączane do wielu programów, co skraca również czas potrzebny do zaprogramowania danego zadania. Stosowanie funkcji umożliwia współpracę wielu programistów przy wspólnym projekcie.

W programach w języku C można wyróżnić trzy podstawowe elementy związane z funkcjami:

- definicję funkcji,
- wywołanie funkcji,
- deklarację funkcji.

Elementy te zostaną omówione w kolejnych podrozdziałach.

6.6.2. Definicja funkcji

Definicja funkcji (kod funkcji) w języku C składa się z nagłówka oraz instrukcji bloku, a jej ogólna postać jest następująca:

```
typ_danych nazwa_funkcji(lista parametrów formalnych)
{
 lista zmiennych lokalnych
 lista instrukcji
}
```

Nagłówek funkcji składa się z typu danych zwracanego wyniku, nazwy funkcji oraz listy parametrów formalnych podanej w nawiasach okrągłych. **Nagłówek nie jest zakończony średnikiem, ponieważ zaraz po nim występuje blok funkcji.**

Typ danych wyniku zwracanego przez funkcję należy do zbioru typów dozwolonych w języku C. Jeżeli funkcja nie zwraca wartości, jako typ danych

stosuje się `void`. Nazwa funkcji jest typowym identyfikatorem w rozumieniu języka C. Lista parametrów formalnych określa zestaw par w postaci:

typ_parametru nazwa_parametru_formalnego

oddzielonych przecinkami.

W bloku funkcji deklarujemy zmienne lokalne oraz instrukcje niezbędne do realizacji algorytmu. Wśród instrukcji może wystąpić instrukcja `return`, która pokazuje miejsce zakończenia działania funkcji i przekazania sterowania do funkcji wywołującej. Jeżeli instrukcja `return` nie występuje, to funkcja kończy działanie na nawiasie klamrowym zamykającym blok funkcji. W treści funkcji może wystąpić kilka instrukcji `return`, choć wykona się tylko jedna. Instrukcja ta umożliwia zwrot wartości przez funkcję, którego typ musi być zgodny z określonym w nagłówku funkcji. Przykład definicji funkcji pokazano na listingu 6.23.

Listing 6.23

```
1 void podkreslenie(char znak, int n)
2 {
3    int i;
4    putchar('\n');
5    for(i = 1; i <= n; i++)
6      putchar(znak);
7    putchar('\n');
8 }
```

Zadaniem funkcji `podkreslenie` jest wyprowadzenie jednej linii zawierającej `n znaków`; `n` i `znak` są parametrami formalnymi funkcji. Funkcja nie zwraca żadnej wartości, dlatego jej typ wyniku zdefiniowano jako `void`.

Definicja funkcji jest określana dla parametrów ogólnych zwanych parametrami formalnymi. Parametry te przekazywane są do wnętrza funkcji tylko przez wartość. Możliwe jest także przekazanie wartości wskaźnika[8]. Przekazanie parametru przez wartość oznacza, że funkcja otrzymuje kopie tych parametrów i działa na nich jak na zmiennych lokalnych zainicjowanych wartościami początkowymi. W przypadku przekazania parametru za pomocą wskaźnika funkcja ma możliwość zmiany zawartości pod zapamiętanym adresem.

6.6.3. Wywołanie funkcji

W celu skorzystania z algorytmu zapisanego w definicji funkcji trzeba się do niej odwołać poprzez jej nazwę i podanie parametrów aktualnych. Ogólna postać wywołania funkcji jest następująca:

[8] Wskaźniki omówiono w podrozdziale 6.8.

```
nazwa_funkcji(lista parametrów aktualnych)
```

Listing 6.24
```
1 /* Przykład wywołania funkcji */
2 #include <stdio.h>
3
4 void main()
5 {
6   char znak = '*';
7   podkreslenie(znak, 10);
8 }
```

Parametrami aktualnymi są wyrażenia oddzielone przecinkami, dla których program wywołujący funkcję dokonuje obliczeń. Pomiędzy parametrami formalnymi (dla których pisana jest definicja funkcji) a aktualnymi (dla których następuje wywołanie funkcji) zachodzą następujące relacje:

- liczba parametrów formalnych i aktualnych jest równa,
- typ parametrów formalnych i aktualnych musi być zgodny – typ parametru aktualnego nie może być szerszy od typu parametru formalnego,
- kolejność parametrów formalnych i aktualnych musi być taka sama.

Przykład wywołania funkcji `podkreslenie` (listing 6.23) prezentuje linia 7 programu 6.24.

6.6.4. Deklaracja funkcji – prototyp

Deklaracja (prototyp) funkcji określa, do jakiego typu należy funkcja. Ogólna postać prototypu jest następująca:

```
typ_danych nazwa_funkcji(lista typów parametrów);
```

Elementy występujące w prototypie mają identyczne znaczenie z odpowiadającymi im w nagłówku definicji funkcji. Prototyp funkcji `podkreslenie` omówionej w podrozdziale 6.6.2 może mieć jedną z dwóch następujących postaci:

```
void podkreslenie(char znak, int n);
```
lub
```
void podkreslenie(char, int);
```

Prototyp nie wymaga podania nazw parametrów formalnych (musi być podany tylko typ). Opuszczenie tych nazw nie jest jednak celowe, gdyż mogą zawierać informacje istotne przy wywołaniu funkcji, co przedstawia następujący przykład:

```
void data(int dzien, int miesiac, int rok);
```
lub
```
void data(int, int, int);
```

Nazwy parametrów pokazują dane, które należy kolejno podać w wywołaniu bez konieczności odszukania definicji funkcji.

Prototyp funkcji jest niezbędny, jeżeli wywołanie funkcji występuje przed jej definicją. Zazwyczaj prototypy wszystkich funkcji tworzonych przez użytkownika umieszcza się przed pierwszą definicją funkcji. Pełny zapis programu z omawianą funkcją `podkreslenie` pokazano na listingu 6.25. Przykład 6.25 wskazuje zalecany układ programu, tj.:

- dołączone biblioteki,
- stałe symboliczne,
- prototypy funkcji,
- funkcja `main`,
- definicje funkcji.

Wynik działania programu jest następujący:

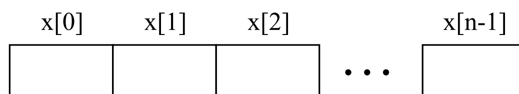
```
**********
```

6.7. Tablice

6.7.1. Definicja tablicy

Tablica w języku C jest to jednorodna struktura danych składająca się z elementów tego samego typu. Tablice mogą być jedno- lub wielowymiarowe, język C nie nakłada ograniczenia na liczbę wymiarów tablic. Elementy tablicy zajmują spójny obszar pamięci, którego początek określa adres przydzielony nazwie tablicy. Na rysunku 6.8 przedstawiono graficznie tablicę n-elementową jednowymiarową o nazwie `x`. Indeksowanie tablic w języku C zaczyna się od 0, a odwołanie do elementu tablicy odbywa się w następującej formie:

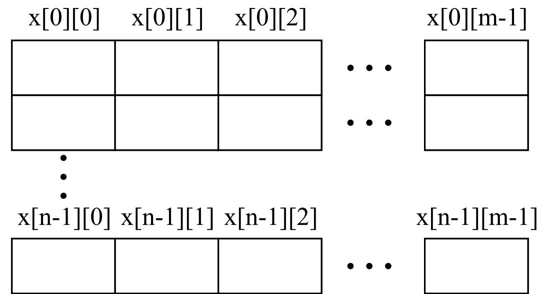
Listing 6.25
```
/* Przykłady funkcji */
#include <stdio.h>

#define N 10

//prototyp funkcji
void podkreslenie(char znak, int n);

void main()
{
  char znak = '*';
  podkreslenie(znak, N);
}

void podkreslenie(char znak, int n)
{
   int i;
   printf("\n");
   for(i = 1; i <= n; i++)
     putchar(znak);
   printf("\n");
}
```



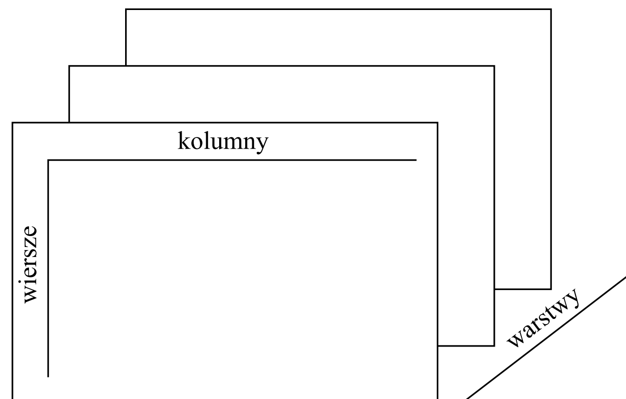
Rys. 6.8. Tablica jednowymiarowa

```
nazwa_tablicy[indeks1];
```

Dostęp do elementów tablicy jest dostępem swobodnym, czyli można odwołać się do dowolnego elementu tablicy. Czas tej operacji jest stały niezależnie od jego indeksów. Tablicę dwuwymiarową można przedstawić jako zestaw tablic jednowymiarowych (rys. 6.9). Elementy tablicy dwuwymiarowej są składowane w spójnym obszarze pamięci w następującej kolejności: wiersz 0, wiersz 1, ..., wiersz n-1. Tablicę trójwymiarową można traktować jako zbiór tablic dwuwymiarowych, gdzie podaje się 3 indeksy oznaczające: wiersz, kolumnę i warstwę (rys. 6.10). Ogólnie można powiedzieć, że tablica n-wymiarowa jest zestawem tablic n-1-wymiarowych.



Rys. 6.9. Tablica dwuwymiarowa



Rys. 6.10. Tablica trójwymiarowa

6.7.2. Deklaracja tablic

Deklaracja tablicy jest to informacja dla kompilatora o liczbie i typie elementów w celu przydzielenia odpowiedniej wielkości pamięci operacyjnej. Tablicę w języku C deklaruje się w następującej konstrukcji:

```
typ_danych nazwa_tablicy[lista_rozmiarów]={lista_wartości};
```

gdzie:
`typ_danych` – typ podstawowy lub zdefiniowany,
`nazwa_tablicy` – identyfikator tablicy,
`lista_rozmiarów` – określenie liczby wymiarów i zakresu ich indeksowania (opcjonalna w przypadku podania listy wartości – kompilator automatycznie obliczy rozmiar tablicy),
`lista_wartości` – zestaw danych początkowych tablicy (opcjonalny).
Na listingu 6.26 pokazano deklarację trzech tablic; podanie wielkości wymiarów było niezbędne.

Listing 6.26
```
1 int tab[10];
2 float x[10][10];
3 char lancuch[255];
```

6.7.3. Inicjowanie tablic

Tablice mogą być zainicjowane poprzez podanie wszystkich wartości lub tylko początkowych fragmentów. Listing 6.27 prezentuje przykłady deklaracji i inicjowania tablic.

Listing 6.27
```
1 float x[10] = {1.2, 0., -3.7};
2 int a[] = {0, 1, 2, 3, 4};
3 char color[] = {'b', 'i', 'a', 'l', 'y'};
4 char barwa[] = "szary";
5 char tekst[100] = "";
6 int b[2][3] = {{100, 101, 102},{200, 201, 202}};
7 int b[2][3] = {100, 101, 102, 200, 201, 202};
8 int c[2][3] = {{100},{200, 201}};
9 char owoce[][8] = {"Banan", "Cytryna", "Melon"};
```

W linii 1 zainicjowano 3 pierwsze elementy tablicy `x`, pozostałe zeruje kompilator. W linii 2 kompilator dobiera rozmiar tablicy `a` i ustala go zgodnie z liczbą zainicjowanych elementów. Tablice znakowe mogą być inicjowane na dwa sposoby. W linii 3 zainicjowano tablicę `color` za pomocą listy 5 elementów typu znakowego, w przypadku zaś tablicy `barwa` (linia 4) została ona zainicjowana łańcuchem znaków, dlatego kompilator zarezerwował 6 elementów i dodał na końcu łańcucha znak \0. Wyzerowanie łańcucha znaków w deklaracji pokazano w linii 5. Deklaracje w liniach 6 i 7 są równoważne, gdyż w przypadku inicjowania wszystkich elementów tablic dwuwymiarowych nie ma konieczności zaznaczania klamrami poszczególnych wierszy. W linii 8 została zainicjowana tablica `c` jednym elementem w wierszu zerowym (niezbędne nawiasy) i dwoma elementami w wierszu pierwszym, pozostałe elementy są zerowane. Przykład inicjowania dwuwymiarowej tablicy znakowej `owoce` pokazano w linii 9, kompilator dobiera wymiar pierwszy tablicy ze względu na liczbę podanych łańcuchów. Drugi wymiar jest niezbędny do wyliczenia adresu kolejnego wiersza przez kompilator.

6.8. Wskaźniki

6.8.1. Deklaracja wskaźnika

Wskaźnik w języku C jest to zmienna, której wartością jest adres wskazujący na pewien obszar pamięci mogący należeć do innej zmiennej. Z operacjami na wskaźnikach są związane dwa operatory z grupy wskazywania:

- operator pobrania adresu (`&`),
- operator adresowania pośredniego (`*`), zwany również operatorem wyłuskania lub dereferencji.

Wskaźniki w języku C deklaruje się w następującej postaci:

*typ_danych * nazwa_zmiennej_wskaźnikowej*

typ_danych – typ podstawowy lub zdefiniowany,
nazwa_zmiennej_wskaźnikowej – identyfikator wskaźnika.

Gwiazdka (`*`) poprzedzająca nazwę zmiennej wskaźnikowej w deklaracji oznacza, że zmienna jest wskaźnikiem. Przykłady deklaracji stosujących wskaźniki pokazano na listingu 6.28.

Listing 6.28

```
1 int *p;
2 int **p;
3 int *tab[8];
4 char (*p)[8];
5 char *f(void);
6 int (*f)(void);
7 float v;
8 float *pv=&v;
9 int *pu=NULL;
```

Linia 1 – `p` jest wskaźnikiem do zmiennej typu `int`.
Linia 2 – `p` jest wskaźnikiem do wskaźnika na zmienną typu `int`.
Linia 3 – `tab` jest 8-elementową tablicą wskaźników typu `int`.
Linia 4 – `p` jest wskaźnikiem do 8-elementowej tablicy typu `char`.
Linia 5 – `f` jest bezparametrową funkcją zwracającą wskaźnik do obiektu typu `char`.
Linia 6 – `f` jest wskaźnikiem do bezparametrowej funkcji zwracającej wartość typu `int`.
Linia 8 – jest to deklaracja wskaźnika `pv` i zainicjowanie adresem zmiennej

v zadeklarowanej w linii 7.
Linia 9 – jest to wyzerowanie wskaźnika pu.

Wskaźniki można zainicjować w deklaracji adresem innej zmiennej lub wartością zerową NULL, która jest stałą symboliczną zdefiniowaną w module stdio.h.

6.8.2. Operacje na wskaźnikach

W języku C można przeprowadzać następujące operacje na wskaźnikach:

- nadania wartości,
- uzyskania wartości przechowywanej pod adresem równym wskaźnikowi,
- porównania wskaźników,
- zmiany wartości wskaźnika o liczbę całkowitą.

Na listingu 6.29 pokazano sposób nadania wartości wskaźnikowi w instrukcji przypisania (linia 2) oraz wyłuskania zawartości pod wskaźnikiem (linia 3).

Listing 6.29

```
1 int u = 13, v, *pu;
2 pu = &u;
3 v = *pu;
```

6.9. Pliki

6.9.1. Funkcje otwarcia i zamknięcia pliku

Plik jest to ciąg danych (zestaw liczb, linii tekstów, struktur) tego samego typu składowanych w pamięci zewnętrznej komputera. W języku C w pliku nagłówkowym stdio.h został predefiniowany typ danych o nazwie FILE, który umożliwia deklarowanie zmiennych plikowych. Deklaracja zmiennej plikowej jest następująca:

```
FILE *plik;
```

Wskaźnik plik wskazuje na strukturę danych zawierającą informację o pliku.

W celu skorzystania z funkcji obsługujących pliki w pierwszej kolejności musi być użyta funkcja fopen, której prototyp ma postać:

```
FILE* fopen(const char *path, const char *mode);
```

Pierwszym parametrem (path) funkcji fopen jest ścieżka dostępu do pliku zakończona jego nazwą. Drugi parametr mode określa tryb otwarcia pliku, który może przyjąć następujące wartości:

r – otwiera istniejący plik do odczytu; wskaźnik pozycji odczytu będzie ustawiony na jego początek,

w – tworzy nowy plik w trybie do zapisu; jeżeli plik wcześniej istniał, to kasuje jego zawartość i ustawia wskaźnik pozycji zapisu na początek pliku,

a – otwiera istniejący plik do dopisywania, dane będą zawsze dołączane do końca pliku, rozszerzając go; próby zmiany wskaźnika pozycji pliku będą ignorowane,

r+ – otwiera istniejący plik do czytania i pisania (aktualizacji); wskaźnik pozycji zapisu i odczytu będzie ustawiony na początek pliku,

w+ – tworzy nowy plik w trybie do pisania i czytania; jeżeli plik wcześniej istniał, to zawartość zostanie skasowana, a wskaźnik zostanie ustawiony na początek pliku,

a+ – otwiera istniejący plik do dopisywania i czytania; dane będą dopisywane do końca pliku, rozszerzając go, a próby zmiany położenia wskaźnika pozycji w pliku utrzymają się tylko do wywołania pierwszej operacji zapisu.

Dodatkowo do parametru mode warto podać wartość specyfikującą interpretację pliku. Dopuszczalne są dwie wartości:

t – plik będzie otwarty w trybie tekstowym, stąd separatorem wiersza będzie jeden znak lub znaki właściwe dla aktualnego systemu operacyjnego[9],

b – plik będzie otwarty w trybie binarnym, czyli żaden znak przesłany do pliku lub odczytany z pliku nie będzie poddany modyfikacji.

Po pomyślnym otwarciu pliku funkcja fopen zwraca wskaźnik plikowy, który służy jako identyfikator pliku dla innych funkcji wejścia-wyjścia. Gdy funkcja open nie może otworzyć pliku, zwraca wskaźnik zerowy (NULL), program powinien zakończyć działanie lub ponowić otwarcie pliku. Wśród najczęściej spotykanych powodów niemożności otwarcia pliku można wskazać: nieprawidłową nazwę pliku, nieistniejącą ścieżkę lub zabroniony dostęp do pliku.

[9] Pozwala to na ujednolicenie sekwencji '\n' oznaczającej znak nowej linii dla wszystkich systemów operacyjnych.

Kolejną funkcją niezbędną do obsługi plików jest `fclose` o następującym prototypie:

```
int fclose(FILE *plik);
```

Jej zadaniem jest zamknięcie pliku podanego jako parametr funkcji. Gdy wskaźnik do pliku jest równy `NULL`, to zachowanie funkcji jest niezdefiniowane. Funkcja zwraca wartość `0` w przypadku prawidłowego wykonania lub `EOF` w przypadku wystąpienia błędów. Stosowanie funkcji `fclose` jest niezbędne, ponieważ opróżnia bufor danych (wysyła niezapisane dane z buforów wyjściowych do plików) i zwalnia wskaźnik oraz zasoby systemowe, które mogą być skojarzone z innym plikiem, a także odblokowuje dostęp do pliku dla innych programów.

Prawidłową kolejność wywoływania funkcji obsługujących plik pokazano na listingu 6.30. Dobrą praktyką zabezpieczenia wykonywania programu jest sprawdzenie, czy dostęp do pliku został utworzony. Wykonuje się to instrukcją `if` i porównaniem otrzymanego wskaźnika do pliku z wartością `NULL`.

Listing 6.30

```
1  #include <stdio.h>//dołączenie biblioteki systemowej
2  ...
3  FILE *plik;//deklaracja zmiennej plikowej
4  ...
5  plik = fopen("dane.txt","r+");//otwarcie pliku
6  if(plik == NULL)
7  {
8    printf("Błąd otwarcia pliku\n");
9    getchar();
10   exit(-1);
11 }
12 ...
13 //funkcje dostępu do pliku
14 ...
15 fclose(plik);//zamknięcie pliku
16 ...
```

Współczesne systemy operacyjne zwykle automatycznie zamykają pozostawione otwarte pliki przy zakończeniu aplikacji. Pozostawienie otwartego pliku blokuje do niego dostęp innym programom, jeżeli w systemie operacyjnym nie zastosowano automatycznych mechanizmów współdzielenia plików.

6.9.2. Funkcje odczytu i zapisu do plików

Język C umożliwia dwa sposoby odczytu i zapisu do plików: tekstowy i binarny. W przypadku zapisu tekstowego następuje konwersja danych do łańcuchów tekstowych, które następnie są zapisywane do plików; wartości numeryczne mogą stracić dokładność, ponieważ zostaną obcięte. Natomiast dla zapisu binarnego postać wartości numerycznych jest identyczna z ich układem bitów w pamięci operacyjnej, a co za tym idzie są dokładnie takie, na jakich operuje program. Z wielu funkcji dostępu do plików tekstowych i binarnych zdefiniowanych w bibliotece standardowej `stdio.h` wybrano najczęściej używane, które przedstawiono w tabl. 6.3.

Listing 6.31

```
1  /* Przykłady zastosowania funkcji fputc, fgetc */
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  void main()
6  {
7    FILE *plik_we;
8    char nazwa[80];
9    int znak;
10   printf("Wyświetlanie zawartości pliku");
11   printf("\nPodaj nazwę pliku: ");
12   gets(nazwa);
13   plik_we = fopen(nazwa,"r+");
14   if(plik_we == NULL)
15   {
16     printf("\nPlik %s nie istnieje",nazwa);
17     getchar();
18     exit(1); //zamknięcie wszystkich plików i wyjście
19   }
20   else
21   {
22     while((znak = fgetc(plik_we)) != EOF)
23       fputc(znak,stdout);
24     fclose(plik_we);
25   }
26 }
```

Na listingu 6.31 pokazano użycie standardowych funkcji `fgetc` i `fputc` zapisujących i odczytujących pojedyncze znaki. Wynik działania programu jest następujący:

```
Wyświetlanie zawartości pliku
Podaj nazwe pliku: plik.txt
```

Tablica 6.3. Wybrane funkcje odczytu i zapisu do plików

Nagłówek funkcji	**Przeznaczenie**
`int fputc(int c, FILE *plik)`	wysyła `c` do pliku, zwraca `c` lub `EOF`
`int fgetc(FILE *plik)`	odczytuje znak z pliku, zwraca wartość znaku lub `EOF`
`int fputs(const char *s, FILE *plik)`	Wysyła łańcuch `s` do pliku, zwraca `EOF` w przypadku błędu.
`char *fgets(char *s, int n, FILE *plik)`	odczytuje znaki z pliku i umieszcza w łańcuchu `s`, odczyt zostaje przerwany po odczytaniu `n-1` znaków lub znaku końca wiersza; umieszcza na końcu łańcucha `s` znak końca łańcucha; zwraca wczytany łańcuch lub `NULL`; jeżeli wczytana liczba znaków jest mniejsza niż `n-1`, wstawia przed końcem łańcucha znak nowej linii
`int fprintf(FILE *plik, format, ...);`	sformatowany zapis do pliku, zwraca wartość `int` równą liczbie wyprowadzanych znaków lub `EOF` w przypadku błędu, `format` – format zapisywanych wartości (taki jak dla funkcji `printf`), . . . – lista zapisywanych zmiennych
`int fscanf(FILE *plik, format, ...)`	sformatowany odczyt z pliku, zwraca wartość `int` równą liczbie prawidłowo odczytanych danych lub `EOF` jeżeli osiągnięto koniec zbioru, `format` – format odczytywanych wartości (taki jak dla funkcji `scanf`), . . . – lista odczytywanych zmiennych.

```
Plik testowy.
Linia 1
Linia 2
```

Użytkownik podaje nazwę pliku do odczytu. W przypadku podania nieprawidłowej nazwy funkcja `fopen` zwróci `NULL` i nastąpi wyjście z programu z równoczesnym zamknięciem wszystkich otwartych plików (funkcja `exit`). Podanie poprawnej nazwy spowoduje przejście do odczytu zawartości pliku tekstowego znak po znaku za pomocą funkcji `fgetc` (linia 22) i wyprowadzenie jej na standardowy strumień wyjścia `stdout` – ekran monitora (linia 23).

6.10. Zadania

1. Przetestować działanie operatorów arytmetycznych: `+`, `-`, `*`, `/`, `%` dla dwóch liczb całkowitych poprzez wpisanie następującego fragmentu kodu w funkcji `main`.

   ```c
   int a = 10, b = 4;
   printf("a = %d\nb = %d\nop = %d\n", a, b, a + b);
   ```

2. Napisać program do obliczania pola trójkąta o podstawie a i wysokości h dla wartości podawanych z klawiatury.
3. Napisać program do rozwiązywania równań liniowych postaci $y = a \cdot x + b$, gdzie $a \neq 0$. Rozwiązaniem takiego równania jest $x = \frac{-b}{a}$. Po wczytaniu współczynników a i b należy sprawdzić warunek obliczeń i wyświetlić stosowny wynik.
4. Napisać program wyświetlający największą i najmniejszą liczbę wprowadzaną z klawiatury, aż do napotkania zera.
5. Napisać program, który wczytuje do tablicy elementów typu `float` pięć liczb podanych z klawiatury z wykorzystaniem dowolnej pętli, np. `for`.
6. Napisać funkcje do wyszukiwania maksymalnej i minimalnej liczby w tablicy. Dostęp do elementów tablicy powinien odbywać się za pomocą indeksów.
7. Napisać funkcję do wyświetlenia znaków będących cyframi we wczytanej tablicy, operując na wskaźniku „wędrującym” po tablicy.
8. Napisać funkcję obliczającą statystykę występowania każdego znaku w tablicy. Jako argumenty funkcja przyjmuje dowolną tablicę znaków oraz tablicę liczb całkowitych o rozmiarze 26 (alfabet łaciński a – z). Literze 'a'

tej tablicy odpowiada indeks 0, ’b’ odpowiada 1 itd. Po zakończeniu funkcji w tablicy liczb całkowitych powinny znajdować się wartości informujące o tym, ile razy litera odpowiadająca danemu indeksowi występowała w łańcuchu wejściowym.

9. Napisać program, który tworzy plik tekstowy, a w nim 10 linii rozpoczynających się ich numerami przy użyciu pętli oraz funkcji `fprintf`.

10. Napisać program, który wyświetli zawartość podanego pliku tekstowego oraz podsumuje liczbę linii w nim występujących.

Rozdział 7.

Język C++

Jan Sadolewski

7.1. Wprowadzenie

Język C++ został zaprojektowany przez Bjarne Stroustrupa w latach 80. XX wieku początkowo jako obiektowe rozszerzenie języka C. W tym czasie powstawało inne obiektowe rozszerzenie języka C pod nazwą Objective C. Jednakże to C++ zdobył pozycję jednego z najpopularniejszych języków programowania ogólnego przeznaczenia, czyli takiego, w którym można pisać zarówno systemy operacyjne, jak i programy użytkowe. Główną wadą Objective C jest nadmiarowa składnia, która nie poprawia czytelności kodu, ale wymaga większej pracy ze strony programisty. Zaletą C++ były początkowo niewielkie zmiany w składni w stosunku do „zwykłego C”. Z czasem zmiany te rozrosły się do rangi nowego języka programowania. Pierwsze kompilatory języka C++ były translatorami przekształcającymi kod (ang. *front-end*) na język C do czasu wydania (na początku lat 90.) kompilatora GNU C++ należącego do pakietu GNU C[1].

Język C++ jest określony przez standard ISO/IEC 14882:1998, w którym wprowadzono zmiany w 2003 i w 2011 roku [14]. Ze względu na znaczne możliwości tego języka (cały standard z 2011 roku ma 1338 stron) zostaną przedstawione tylko podstawowe jego funkcjonalności, do których należą definicje klas i dziedziczenie, biblioteki wejścia-wyjścia, konstruktory i destruktory z dynamiczną alokacja pamięci oraz metody wirtualne z wykrywaniem typu obiektu w trakcie pracy aplikacji.

Cechą języka C++ jest możliwość kompilacji kodu zapisanego w standardzie ANSI C z nielicznymi wyjątkami, co pozwala na wykorzystanie dotychczasowych umiejętności nabytych podczas nauki języka C do zastosowania w C++. Należy pamiętać, że C++ nie jest językiem łatwiejszym do nauki niż sam język C, ale jego rozpowszechnienie wymaga nawet niewielkiej wiedzy na temat jego możliwości.

[1] Składnik standardowych dystrybucji Linux, dostępny również dla systemu Windows na `http://sourceforge.net/projects/mingw/files/Installer/mingw-get`.

7.2. Struktury i klasy

W języku C jest dostępny mechanizm agregacji danych zwany strukturami. Przykładowo, chcąc zapamiętać imię, nazwisko i wiek dziesięciu osób, konieczne jest zadeklarowanie trzech tablic o rozmiarze 10 elementów, po jednej tablicy na każdy atrybut osoby. Ponadto, aby zapamiętać nazwisko i imię, konieczne są tablice tablic, których indeksowanie jest kłopotliwe. Rozwiązaniem jest zastosowanie struktury danych, które agregują atrybuty w jeden spójny zbiór.

Strukturą (ang. *structure*) nazywamy zwięzły obszar, w którym są przechowywane nazwane pola zawierające inne typy. Na listingu 7.1 przedstawiono deklarację struktury `Osoba` zawierającej pola określające atrybuty `Imie`, `Nazwisko` i `Wiek`. Należy zwrócić uwagę, że deklarację struktury rozpoczyna się od słowa kluczowego `struct`, następnie nazwy struktury i definicji pól objętych nawiasami klamrowymi. Deklarację kończy znak średnika, który często jest zapominany przez początkujących programistów.

Listing 7.1
```
1 struct Osoba
2 {
3   char Nazwisko[20];
4   char Imie[20];
5   int  Wiek;
6 };
```

Definicję zmiennej wykorzystującej typ strukturalny przedstawiono w linii 3 na listingu 7.2. W języku C wymagane jest podanie słowa kluczowego `struct` przed nazwą typu strukturalnego podczas deklaracji zmiennej, a w C++ możliwe jest jego pominięcie. Jak można zauważyć, dostęp do pól struktury jest realizowany za pomocą operatora kropki (`.`). Operator ten ma wysoki priorytet, co powoduje, że konstrukcja `czlowiek.Nazwisko` może być postrzegana przez programistę jako identyfikator.

Język C++ pozwala na definicję klas. Poprzez klasę rozumie się pewną strukturę danych, do której można dodać funkcje operujące na tychże danych. Funkcje te nazywa się metodami, a dane składowe polami. Cechą, która odróżnia klasę od struktury[2], jest domyślny modyfikator zakresu widoczności. W C++ istnieją trzy modyfikatory zakresu:

- `public` – oznacza, że elementy (pola i metody) są wszędzie widoczne,
- `protected` – oznacza, że elementy będą widoczne (poza klasą, w której są zadeklarowane) tylko w klasach z niej dziedziczących,

[2] W języku C++ dopuszcza się również deklarowanie metod w strukturach.

Listing 7.2

```
1  void main()
2  {
3      struct Osoba czlowiek;
4      printf("Podaj imię:");
5      fgets(czlowiek.Imie, 20, stdin);
6      czlowiek.Imie[strlen(czlowiek.Imie)-1] = '\0'; //kasowanie
7         // znaku '\n' kończącego łańcuch zwrócony przez fgets
8      printf("Podaj nazwisko:");
9      fgets(czlowiek.Nazwisko, 20, stdin);
10     czlowiek.Nazwisko[strlen(czlowiek.Nazwisko)-1] = '\0';
11     printf("Podaj wiek:");
12     scanf("%d", &czlowiek.Wiek);
13     //wypisanie wczytanych danych
14     printf("Podana osoba>> Imie: %s, Nazwisko: %s, Wiek: %d",
15        czlowiek.Imie, czlowiek.Nazwisko, czlowiek.Wiek);
16 }
```

Listing 7.3

```
1  class Osoba
2  {
3   public:
4      char Nazwisko[20];
5      char Imie[20];
6      int  Wiek;
7  };
```

- `private` – oznacza, że elementy będą widoczne tylko w tej klasie, w której są zadeklarowane[3].

Sama deklaracja klasy jest zbliżona do struktury. Odpowiednik struktury `Osoba` z listingu 7.1 przedstawiono w postaci klasy na listingu 7.3. Jak można zauważyć, deklaracja klasy zastąpiła słowo kluczowe `struct` na `class` i dodała modyfikator widoczności `public` w trzeciej linii. Wykorzystując przykład z listingu 7.2, można zmodyfikować deklarację zmiennej `czlowiek` poprzez usunięcie słowa `struct` przed `Osoba czlowiek` w linii 3. Uzyskany w ten sposób obiekt `czlowiek` klasy `Osoba` nazywa się instancją. W celu przetestowania poprawności naszej deklaracji możemy posłużyć się tym samym kodem z języka C, co tylko potwierdza możliwość migracji kodu źródłowego pomiędzy C a C++.

W dalszej części rozdziału będą przedstawione cechy charakteryzujące klasy, przy czym najpierw konieczne jest omówienie nowego standardu wyświetlania informacji na konsoli.

[3] W C++ istnieją jeszcze zaprzyjaźnienia, które pozwalają metodom innych klas na dostęp do prywatnych elementów, jednak zagadnienie to wykracza poza ramy tej książki.

7.3. Biblioteki wejścia wyjścia

Typowym błędem programistów piszących w C była pomyłka w argumentach dla funkcji `printf`. Wraz z rozwojem języka C++ powstała nowa koncepcja wyświetlania informacji na ekranie poprzez wykorzystanie przeciążenia operatorów. Zagadnienie przeciążania operatorów nie zostanie omówione, ponieważ przekracza ramy tej książki. Jego efektem jest uzyskanie innej funkcjonalności operatora niż w pierwotnym przypadku. Ma to na celu uproszczenie zapisu kodu, lecz niestety czyni go też bardziej niezrozumiałym przez osobę, która nie uczestniczyła w procesie jego powstawania.

Przykładowy sposób wyświetlania danych przewidziany dla języka C++ przedstawiono na listingu 7.4. Jak można zauważyć, zamiast pliku nagłówkowego `stdio.h` stosuje się `iostream` (bez rozszerzenia .h) oraz deklarację `using namespace std;`, która definiuje używaną w tym pliku przestrzeń nazw. Przykłady nie wykorzystują innej przestrzeni nazw niż standardowa, dlatego zastosowano identyfikator `std`. Przestrzenie nazw wykraczają poza ramy przewidziane w niniejszej książce; są one przeznaczone dla bardziej zaawansowanych programistów. Początkujący programista C++ nie powinien koncentrować się na przestrzeniach nazw, tylko pamiętać o dopisaniu tej deklaracji. Gdyby jednak o tym zapomniał, to kompilator zgłosiłby błąd mówiący o nieznalezionych identyfikatorach `cout` i `endl`.

Listing 7.4

```
1  #include <iostream>
2  using namespace std;
3
4  void main()
5  {
6      cout << "Witaj świecie" << endl;
7  }
```

Identyfikator `cout` oznacza standardowy strumień wyjściowy, odpowiednik `stdout` z języka C, tylko zapisany w postaci klasy `ostream`. Strumień wyjściowy ma przeciążony operator << (pierwotnie przesunięcia bitowego w lewo), którego nowym zadaniem jest wyświetlenie argumentu znajdującego się z prawej strony i zwrócenie referencji na ten strumień. Powoduje to możliwość tworzenia konstrukcji kaskadowej, tak jak w przykładzie, pozwalającej na dopisanie kolejnego operatora << oraz znaku końca linii z wymieceniem bufora `endl`. Jak można zauważyć, z prawej strony operatora << występuje literał tekstowy. Może również wystąpić zmienna dowolnego typu prostego, przez co nie ma konieczności pamiętania modyfikatora formatowania łańcucha dla konkretnego typu. W przypadku konieczności wyświetlenia wartości zmiennej w innym formacie niż domyślny należy wykorzystać manipulatory strumieniowe, takie jak: `hex`, `oct`, `dec`

oraz zdefiniowane w pliku `iomanip` manipulatory parametryzowane: `setbase`, `setprecision`, `setfill`, `setw` itp., których przykład zastosowania przedstawiono na listingu 7.5. Pełne omówienie tych manipulatorów wykracza jednak poza ramy tej książki.

Listing 7.5

```
1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4
5  void main()
6  {
7    int x = 32;
8    cout << "Test manipulatorów: " << hex << x << " float: "
9      << setprecision(10) << 3.14f << endl;
10 }
```

Strumień `cout` służy do wyświetlania informacji, natomiast strumień `cin` do pobierania informacji z konsoli. W strumieniu `cin` przeciążono operator >> (pierwotnie przesunięcia bitowego w prawo), którego nowym zadaniem jest pobieranie danych z konsoli i zapisywanie ich pod zmienne typu prostego. Przykład użycia operatora >> przedstawiono na listingu 7.6. Pobieranie wartości zmiennych w języku C++ jest nieco prostsze niż w C, ponieważ nie podaje się znaków ampersand (`&`) przed nazwą zmiennej. Kompilator sam pobiera adres (referencję) do zmiennej, pod którą ma zapisać wynik. Ponadto błędny zapis ~~`cin >> 40`~~ nie skompiluje się, a semantycznie poprawny odpowiednik w języku C ~~`scanf("%d", 40)`~~ może być fatalny w skutkach, nie wspominając już o możliwości pomyłki w za-

Listing 7.6

```
1  #include <iostream>
2  using namespace std;
3
4  void main()
5  {
6      char imie[20];
7      int wiek;
8      cout << "Podaj imię: ";
9      cin >> imie;
10     cout << "Podaj wiek: ";
11     cin >> wiek;
12
13     cout << imie << " ma teraz " << wiek << " lat";
14 }
```

stosowanych modyfikatorach funkcji `scanf` w stosunku do przekazanego typu argumentu.

7.4. Konstruktory, destruktory i przeciążanie metod

Konstruktorem (ang. *constructor*) nazywa się specjalną metodę, której zadaniem jest zainicjowanie zmiennych w instancji obiektu. Celem konstruktora jest ustawienie pewnych wartości dla pól występujących w klasie. Jeżeli w deklaracji klasy nie podano konstruktora (jak w przykładzie przedstawionym na listingu 7.3), to kompilator przyjmuje domyślny bezparametrowy konstruktor. Cechami charakterystycznymi konstruktora w C++ są nazwa, która musi być identyczna z nazwą klasy, oraz brak podanego typu zwracanego. Uniemożliwia to deklarowanie funkcji zgodnie ze standardem ANSI C, w którym domyślnym typem zwracanym przez funkcje był `int`. Konstruktorów może być kilka poprzez wykorzystanie mechanizmu przeciążania metod. Przykład deklaracji konstruktora dla klasy `Osoba` przedstawiono w linii 8 listingu 7.7.

Listing 7.7

```
1  class Osoba
2  {
3   public:
4      char Nazwisko[20];
5      char Imie[20];
6      int  Wiek;
7
8      Osoba();
9      ~Osoba();
10 };
```

Destruktorem (ang. *destructor*) nazywa się specjalną metodę, której zadaniem jest „zniszczenie” obiektu poprzez zwolnienie dodatkowej zaalokowanej pamięci (jeżeli taką przydzielono w trakcie „życia” instancji). Destruktor może być tylko bezparametrowy. Jeżeli w deklaracji nie podano destruktora, to kompilator przyjmuje, że jest domyślny. Nazwa destruktora jest równa nazwie klasy poprzedzonej znakiem tyldy (~). W każdej klasie może być tylko jeden destruktor. Przykład jego deklaracji przedstawiono w linii 9 listingu 7.7.

Deklaracje konstruktorów, destruktorów oraz innych metod wymagają późniejszej implementacji. Przykładową implementację konstruktora i destruktora będącego kontynuacją listingu 7.7 przedstawiono na listingu 7.8. Jak można zauważyć, implementacja konstruktora składa się z nazwy klasy, operatora zakresu podwójnego dwukropka (`::`) oraz nazwy konstruktora i jego parametrów. Później następuje implementacja konstruktora, czyli kod podobny jak dla funkcji, z jedną

Listing 7.8
```
1  Osoba::Osoba()
2  {
3    memset(Nazwisko, 0, 20);
4    memset(Imie, 0, 20);
5    Wiek = 0;
6  }
7
8  Osoba::~Osoba()
9  {
10 }
```

uwagą dotyczącą widoczności zmiennych. Zmienne `Nazwisko`, `Imie`, `Wiek` są polami w klasie, dlatego nie wymagają deklaracji, ponadto wartości pól w każdej instancji klasy `Osoba` mogą być inne. Podczas tworzenia nowej instancji warto wyczyścić zawartość tablic, dlatego użyto funkcję `memset`, która wypełnia tablicę `Nazwisko` wartością `0`. Podobne działania wykonuje się w przypadku tablicy `Imie`. Pole `Wiek` ma ustawioną wartość na `0` zwyczajnie poprzez operator podstawienia.

Konstruktor służy do ustawienia wartości początkowych tych zmiennych. Jeżeli wartość tej zmiennej jest specyficzna dla każdej instancji i musi być ustalona już w trakcie jej tworzenia, to należy wykorzystać konstruktor z parametrami. Przykład takiego konstruktora, który mógłby być umieszczony w deklaracji klasy `Osoba` (np. po linii 8 listingu 7.7), jest następujący: `Osoba(const char* Imie, const char* nazwisko, const int wiek);`. Pierwszy parametr tego konstruktora ma celowo taką samą nazwę `Imie`, jak nazwa pola, aby przedstawić możliwość przysłonięcia pól przez parametry, co jest dopuszczalne i akceptowalne w języku C++. Implementację tego konstruktora przedstawiono na listingu 7.9. Aby uzyskać dostęp do przysłoniętego pola, należy wykorzystać słowo kluczowe `this` oraz operator strzałki (`->`) wydobywający pole z podanego wskaźnika. Mechanizm ten wykorzystano w funkcji `strncpy` kopiującej do tablicy z pierwszego argumentu łańcuch tekstowy przekazany jako drugi argument o długości nie większej niż wartość trzeciego argumentu. Wyjaśnienia wymaga jeszcze modyfikator `const` użyty przed wskaźnikiem `char*`, którego zada-

Listing 7.9
```
1  Osoba::Osoba(const char* Imie, const char* nazwisko, const int
   wiek)
2  {
3    strncpy(this->Imie, Imie, 20);
4    strncpy(Nazwisko, nazwisko, 20);
5    this->Wiek = wiek;
6  }
```

Listing 7.10

```
1  class Osoba
2  {
3   public:
4    char Nazwisko[20];
5    char Imie[20];
6    int  Wiek;
7
8    Osoba();
9    Osoba(const char* Imie, const char* nazwisko, const int wiek);
10   ~Osoba();
11   void WczytajOsobe();
12   void WypiszOsobe();
13 };
```

Listing 7.11

```
1  void Osoba::WczytajOsobe()
2  {
3    cout << "Podaj imię: ";
4    cin >> Imie;
5    cout << "Podaj nazwisko: ";
6    cin >> Nazwisko;
7    cout << "Podaj wiek: ";
8    cin >> Wiek;
9  }
10
11 void Osoba::WypiszOsobe()
12 {
13   cout << Imie << " " << Nazwisko << " ma teraz " << Wiek <<
14     " lat";
15 }
```

niem jest poinformowanie kompilatora, że wartość parametru `Imie` nie może ulec zmianie. Poprzez słowo kluczowe `this` można się dostać również do pola nieprzysłoniętego przez argument, wówczas zapis w linii 5 omawianego listingu jest równoważny z następującym `Wiek = wiek;`. Znaczenie słowa `const` przed parametrem `wiek` można również sprawdzić, dodając hipotetyczną instrukcję `wiek = 20;`, której wystąpienie w implementacji konstruktora spowoduje błąd kompilacji z komunikatem, że nie można zmienić wartości stałego obiektu.

Jak wspomniano na początku rozdziału, metodą w klasie jest funkcja operująca na polach. Dodatkowo konstruktory i destruktor są specjalnymi metodami służącymi do inicjacji i destrukcji nieautomatycznych pól w instancji[4]. Metody służą

[4] Dotychczas używane pola (tablice i zmienna całkowita) są niszczone automatycznie przez kompilator.

Listing 7.12
```
1  void Osoba::WczytajOsobe(bool pIm, bool pNazw, bool pWiek)
2  {
3      if(pIm)
4      {
5          cout << "Podaj imię: ";
6          cin >> Imie;
7      }
8      if(pNazw)
9      {
10         cout << "Podaj nazwisko: ";
11         cin >> Nazwisko;
12     }
13     if(pWiek)
14     {
15         cout << "Podaj wiek: ";
16         cin >> Wiek;
17     }
18 }
19
20 void Osoba::WczytajOsobe()
21 {
22     WczytajOsobe(true, true, true);
23 }
```

przede wszystkim do wykonania typowych operacji na obiekcie, jak na przykład pobranie danych czy prezentacja wyników, ewentualnie wykonanie pewnych obliczeń. Przykład deklaracji metod `WczytajOsobe` i `WypiszOsobe` przedstawiono w liniach 11 i 12 na listingu 7.10. Metody te nie przyjmują parametrów, a ich deklaracje są identyczne z prototypami funkcji omówionymi w podrozdziale 6.6, z tą różnicą, że znajdują się wewnątrz klasy. Implementację metod zawarto na listingu 7.11, a ich działanie nie wymaga dodatkowego komentarza.

Poprzez przeciążanie metod rozumie się możliwość deklaracji dwóch lub więcej metod o tej samej nazwie, tylko z innymi argumentami. Wybranie właściwej metody do wywołania powierza się kompilatorowi na etapie kompilacji. Przykładowo, do zadeklarowanej w poprzednim akapicie metody `WczytajOsobe` zostanie dodana druga metoda o nazwie `WczytajOsobe`, ale z trzema parametrami typu `bool`[5]. Deklaracja tej metody wyraża się następująco `void WczytajOsobe(bool pIm, bool pNazw, bool pWiek);`, jej implementację przedstawiono zaś na listingu 7.12. Równocześnie można zmniejszyć ilość

[5] Typ `bool` został wprowadzony do języka C++ w celu odróżnienia go od innych typów całkowitych, takich jak `int` czy `char`. Wartościami dla typu `bool` są stałe `true` i `false`, które w C++ zostały zaliczone w poczet słów kluczowych tego języka.

Listing 7.13

```
1  void main()
2  {
3    Osoba os1("Jan", "Nowak", 50);
4    Osoba os2;
5    os2.WczytajOsobe();
6
7    os1.WypiszOsobe();
8    cout << endl;
9    os2.WypiszOsobe();
10   cout << "\nOsoba os2 ma podane nazwisko: " << os2.Nazwisko <<
11      endl;
12 }
```

kodu potrzebnego do realizacji bezparametrowej metody `WczytajOsobe` poprzez wywołanie sparametryzowanej metody `WczytajOsobe`, tak jak w liniach 20–23 listingu 7.12.

Ostatnim etapem prac nad klasą `Osoba` jest wykorzystanie metod do łatwiejszego zarządzania obiektami. Przykład użycia metod oraz dostępu do pól klasy przedstawiono na listingu 7.13. W linii 3 został wykorzystany konstruktor z parametrami, w którym inicjuje się instancję `os1` danymi wpisanymi bezpośrednio do kodu programu. W linii 4 wykorzystano konstruktor bez parametrów (należy

Listing 7.14

```
1  class Osoba
2  {
3  protected:
4    char Nazwisko[20];
5    char Imie[20];
6    int  Wiek;
7  public:
8    Osoba();
9    Osoba(const char* Imie, const char* nazwisko, const int wiek);
10   ~Osoba();
11   void WczytajOsobe();
12   void WczytajOsobe(bool pIm, bool pNazw, bool pWiek);
13   void WypiszOsobe();
14   void SetNazwisko(const char* NoweNazwisko);
15   string GetNazwisko();
16   void SetImie(const char* NoweImie);
17   string GetImie();
18   void SetWiek(const int NowyWiek);
19   int GetWiek();
20 };
```

zwrócić uwagę, że nie może on mieć nawiasów), a podanie danych do zainicjowania pozostawiono na później (tymczasowo konstruktor bezparametrowy wyzeruje pamięć przydzieloną dla instancji `os2`). Pobranie danych do zainicjowania jest wykonywane przez bezparametrowy wariant metody `WczytajOsobe` (linia 5), która pośrednio wywołuje metodę `WczytajOsobę` z trzema parametrami typu `bool`. W linii 7 na konsoli wyświetla się dane osoby zapamiętanej w instancji `os1`, a instrukcja z linii 8 przechodzi do następnego wiersza konsoli. Podobnie w linii 9 wyświetla się dane osoby zapamiętanej w instancji `os2`. W linii 10 zaprezentowano bezpośredni dostęp do pól klasy `Osoba` (ponieważ dostęp do nich nie został ograniczony zakresem widoczności). Powoduje to, że pole `Nazwisko` z instancji `os2` może być wyświetlone bezpośrednio na ekranie.

7.5. Enkapsulacja pól

Jedną z cech programowania obiektowego jest enkapsulacja pól. Polega ona na ograniczeniu możliwości wykorzystania bezpośredniego dostępu do pól obiektu na rzecz używania metod. Podejście to pozwala na ławiejsze wprowadzenie późniejszych zmian w tych fragmentach kodu, które wykorzystują metody do pozyskiwania informacji o tych wartościach. Bezpośredni dostęp do pól jest szybszy z punktu widzenia czasu wykonywania niż angażowanie metod, ale tworząc początkowy kod, programista skupia się nad zwiększeniem podatności na zmiany, a nie nad szybkością działania.

Kontynuując przykład dotyczący klasy `Osoba` z poprzedniego podrozdziału, procesowi enkapsulacji należy poddać pola `Nazwisko`, `Imie`, `Wiek`. Wykonuje się ją poprzez zmodyfikowanie zakresu widoczności pól z `public` na `private` lub `protected`. Decyzję o wyborze ograniczonego zakresu widoczności pozostawia się autorowi klasy. Użycie węższego zakresu widoczności (`private`) nie pozwoli na dostęp do pól nawet dla klas pochodnych, o których będzie mowa w podrozdziale 7.6. Początkującym programistom można zasugerować wykorzystanie zakresu widoczności `protected`. Modyfikując linię 3 listingu 7.10 na `protected` oraz dopisując w linii 7 zakres widoczności `public` dla pozostałych metod (jak na listingu 7.14), uniemożliwia się skompilowanie kodu z listingu 7.13. Pole `os2.Nazwisko` jest użyte w miejscu poza zakresem jego widoczności. W celu przywrócenia pierwotnej funkcjonalności trzeba zadeklarować dodatkowe metody enkapsulujące pola klasy. Zwyczajowo nazwy tych metod rozpoczynają się od słów `Get` i `Set`[6]. Metoda `Set` przyjmuje jeden parametr będący nową wartością dla pola, którego nazwa stanowi dalszy człon metody, a typem zwracanym jest zazwyczaj `void`. Gdy metoda `Set` jest napisana profesjonalnie z walidacją

[6] Różne podejścia programistyczne mogą stosować swoje przedrostki (np. `get`, `set`, `Get_`, `Set_`).

Listing 7.15
```
1  void Osoba::SetNazwisko(const char* NoweNazwisko)
2  {
3    strncpy(Nazwisko, NoweNazwisko, 20);
4  }
5
6  string Osoba::GetNazwisko()
7  {
8    return Nazwisko;
9  }
10
11 void Osoba::SetImie(const char* NoweImie)
12 {
13   strncpy(Imie, NoweImie, 20);
14 }
15
16 string Osoba::GetImie()
17 {
18   return Imie;
19 }
20
21 void Osoba::SetWiek(const int NowyWiek)
22 {
23   Wiek = NowyWiek;
24 }
25
26 int Osoba::GetWiek()
27 {
28   return Wiek;
29 }
```

wartości wejściowej, to jako typy zwracane stosuje się `bool` lub `int`, które informują o poprawności nowej wartości i pomyślnej aktualizacji tego pola w obiekcie. Metoda `Get` nie przyjmuje parametrów[7], a typ zwracany jest zgodny z typem enkapsulowanego pola. Metoda `Get` zwraca kopię wartości pola, jej modyfikacje nie wpływają więc na instancję klasy. Takie podejście czyni obiekt hermetycznym, czyli odpornym na przypadkowe modyfikacje wykonane w kodzie poza klasą przez programistę. Jest to zabezpieczenie tylko na poziomie kompilacji, ponieważ nigdy nie ma pewności, że jakiś „wiszący wskaźnik" nie zmodyfikuje tego obszaru pamięci, w którym jest przechowywana nasza klasa.

W klasie `Osoba` wykorzystano typ `string` do zwracania kopii wartości zapamiętanych w polach `Nazwisko` i `Imie`. Realizacja tego „kopiowania" jest

[7] Profesjonalne implementacje metody `Get` zwracające typy klasowe wykorzystują referencje przekazywane jako parametry, które zostały pominięte w tej książce.

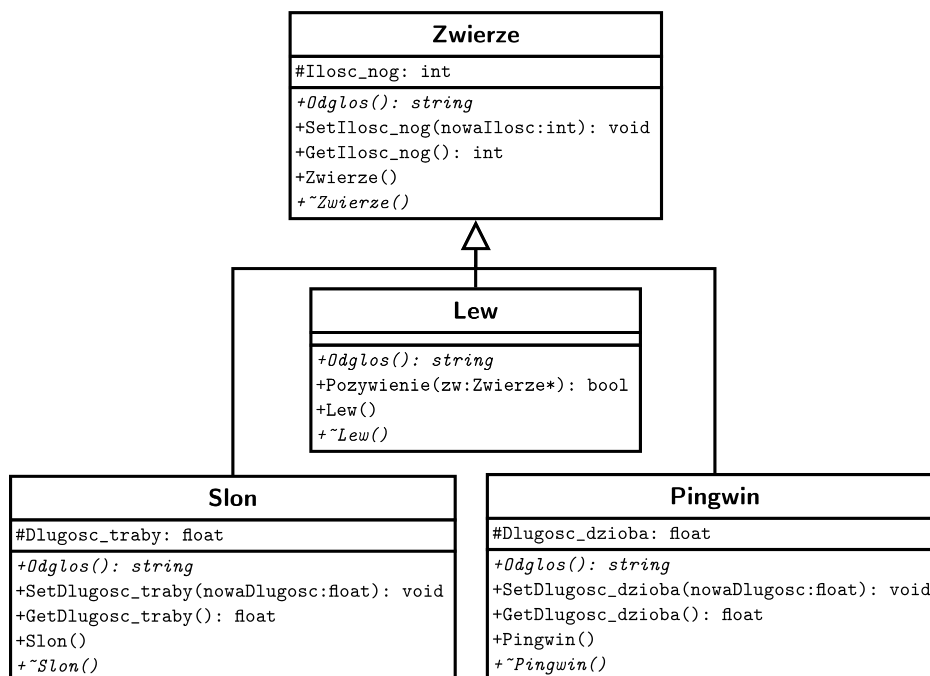
Listing 7.16
```
1 cout << "\nOsoba os2 ma podane nazwisko: " <<
2         os2.GetNazwisko().c_str() << endl;
```

łatwa, ponieważ string jest klasą w C++, dla której zaimplementowano mechanizmy pozwalające na pobranie wartości z tablicy znaków. Na listingu 7.15 przedstawiono implementację metod wykonujących enkapsulację pól dla klasy Osoba. Należy zwrócić uwagę, że deklarowanie klas jako typ zwracany przez metodę lub funkcję może (pomimo poprawnej kompilacji) nie działać poprawnie. Uwaga ta dotyczy zwłaszcza starszych kompilatorów, które przydzielały dla lokalnych obiektów miejsce na stosie. Współczesne popularne kompilatory są odporne na tego typu sytuację, ale w celu zapewnienia maksymalnej zgodności z kompilatorami warto rozważyć zastosowanie referencji, które wykraczają poza ramy tej książki. Aby przywrócić możliwość skompilowania kodu z listingu 7.13, pole os2.Nazwisko należy zastąpić wywołaniem metod GetNazwisko i c_str zwracających kopię łańcucha i następnie wskaźnik na tablicę znaków do wypisania w tym łańcuchu. Wówczas linia 10 listingu 7.13 przyjmie postać przedstawioną na listingu 7.16.

7.6. Dziedziczenie i metody wirtualne

Dotychczas rozważany przykład z klasą Osoba nawiązywał do struktury zawierającej dane osoby. Oznacza to, że program ten łatwo można przekształcić do postaci strukturalnej, wprowadzając odpowiednie funkcje operujące na wskaźnikach do struktury. Cechą programowania obiektowego jest dziedziczenie, które pozwala na współużytkowanie danych i kodu przez różne klasy. W niniejszym podrozdziale będzie to przedstawione na przykładzie klas dotyczących zwierząt. Ilustracja hierarchii klas zostanie wyrażona za pomocą diagramu w graficznym języku UML [4].

Na rysunku 7.1 przedstawiono schemat klas dla pewnych obiektów wywodzących się od klasy Zwierze. Każda klasa jest przedstawiona w postaci prostokąta podzielonego na trzy części. W pierwszej znajduje się nazwa klasy, w drugiej wprowadzane pola, a w trzeciej metody. Zakres widoczności danego pola lub metody określa pierwszy znak w wierszu należącym do drugiej lub trzeciej części prostokąta. Znakiem plus (+) oznacza się pola lub metody należące do zakresu public, znakiem hash (#) – zakres widoczności protected, a znakiem minus (–) – zakres widoczności private. Mechanizm dziedziczenia w C++ polega na przekazywaniu zadeklarowanych pól i metod klasom, które wywodzą się z klasy podstawowej. Przykładowo, na rys. 7.1 klasa Zwierze jest klasą podstawową, a dziedziczą z niej (oznaczane przez symbol strzałki z grotem skiero-



Rys. 7.1. Schemat dziedziczenia klas w języku UML

wanym ku klasie podstawowej) klasy `Slon`, `Lew` i `Pingwin`. Naturalne wydaje się, że słoń, lew i pingwin są uszczegółowieniem ogólnego pojęcia zwierzę. Pole `Ilosc_nog` zostało zadeklarowane w klasie bazowej, ponieważ każdemu zwierzęciu możemy przypisać liczbę nóg. Ponadto każde ze zwierząt może wydawać odgłos, dlatego utworzono metodę wirtualną (oznaczaną przez czcionkę pochyłą) o nazwie `Odglos`, która zwraca łańcuch typu `string` z opisem dźwięku wydawanego przez to zwierzę. Stąd każdy obiekt klasy `Slon`, `Lew` czy `Pingwin` będzie miał pole `Ilosc_nog` oraz metodę `Odglos`, którą będzie mógł „nadpisać”, o czym będzie mowa w dalszej części tego podrozdziału.

Nie każde zwierzę ma trąbę, dlatego klasa `Slon` wprowadza pole o nazwie `Dlugosc_traby` typu `float`, które będzie przechowywać tę „właściwość” słonia. Podobnie będzie w przypadku klasy `Pingwin`, która wprowadza pole `Dlugosc_dzioba` również typu `float`. Warto zauważyć, że w języku UML po nazwach metod w nawiasach okrągłych umieszcza się parametry w postaci *`nazwa`*`:`*`typ`*, a po nawiasie okrągłym zamykającym parametry znak dwukropka (`:`) i *`typ_zwracany`* przez funkcję. Klasa `Lew` wprowadza metodę `Pozywienie`, której zadaniem jest sprawdzenie, czy obiekt danej klasy jest lubianym pożywieniem przez lwa. Metoda ma jeden parametr o nazwie `zw` i typie wskaźnik na `Zwierze`.

Listing 7.17
```
1  class Zwierze
2  {
3    protected:
4      int Ilosc_nog;
5    public:
6      virtual string Odglos();
7      void SetIlosc_nog(int nowaIlosc);
8      int GetIlosc_nog();
9      Zwierze();
10     virtual ~Zwierze();
11 };
```

Listing 7.18
```
1  class Slon : public Zwierze
2  {
3    protected:
4      float Dlugosc_traby;
5    public:
6      virtual string Odglos();
7      void SetDlugosc_traby(float nowaDlugosc);
8      float GetDlugosc_traby();
9      Slon();
10     virtual ~Slon();
11 };
```

Tworząc deklarację klasy w C++ na podstawie diagramu klas UML stosuje się następujące podejście: rozpoczyna się od klas podstawowych, czyli od tych, od których nie ma strzałek wychodzących. Dalej należy utworzyć początek deklaracji klasy o nazwie `Zwierze` przedstawiony w linii 1 na listingu 7.17. Następnie wypisuje się pola o widoczności `private`, potem `protected` i `public`. Zazwyczaj jednym modyfikatorem widoczności możemy pokryć wszystkie deklaracje pól i metod należące do tego samego zakresu. Nie ma przeszkód do przeplatania modyfikatorów widoczności w deklaracjach. W przykładzie klasa `Zwierze` ma jedno pole `Ilosc_nog` typu `int`, stąd powstała linia 4 listingu. Po deklaracji pól należy zapisać metody danej klasy przedstawione w trzeciej części prostokąta. Ponieważ metoda `Odglos` jest wirtualna, przed jej deklaracją należy użyć słowa kluczowego `virtual`, tak jak w linii 6 listingu. Linie 7 i 8 zawierają metody enkapsulujące pole `Ilosc_nog`, a linie 9 i 10 deklarują konstruktor i destruktor (omawiane w poprzednich podrozdziałach nie wymagają więc dodatkowego komentarza). Warto też zaznaczyć, że destruktor powinien być wirtualny, ponieważ ułatwia to zwalnianie obiektów z pamięci w sytuacjach, gdy klasa jest bazą przewidzianą do dalszego dziedziczenia.

Kontynuując podejście konwersji diagramu UML do postaci kodu w C++, przechodzi się do tych klas pochodnych, które już mają zadeklarowane wszystkie klasy bazowe. Konwersja prostokąta odpowiadającego za klasę `Slon` przebiega podobnie jak w przypadku klasy `Zwierze`. Jedyna różnica polega na zaznaczeniu dziedziczenia, które wykonuje się poprzez znak dwukropka (`:`), modyfikatora widoczności dziedziczenia `public` oraz nazwy klasy bazowej `Zwierze`. Przedstawiono to w linii 1 listingu 7.18.

Pozostałe deklaracje nie powinny stanowić większych trudności dla Czytelnika. Celem sprawdzenia poprawności samodzielnego utworzenia pozostałe klasy `Lew` i `Pingwin` przedstawiono na listingu 7.19. Częstą pomyłką wśród początkujących programistów jest zamiana pozycji w deklaracji typu parametru z nazwą parametru. Na diagramie UML umieszcza się najpierw nazwę, a później dwukropek i typ parametru, natomiast w C i C++ – najpierw typ, a później nazwę parametru. Tak przygotowane deklaracje wymagają implementacji w kodzie źródłowym. Implementację metod umieszczono na listingu 7.20. W celu zaoszczędzenia miejsca pominięto puste deklaracje destruktorów oraz prostych metod enkapsulujących pola. Metoda `Lew::Pozywienie` będzie zaimplementowana w dalszej części tego rozdziału.

Listing 7.19

```
1  class Lew : public Zwierze
2  {
3    public:
4      virtual string Odglos();
5      bool Pozywienie(Zwierze* zw);
6      Lew();
7      virtual ~Lew();
8  };
9
10 class Pingwin : public Zwierze
11 {
12   protected:
13     float Dlugosc_dzioba;
14   public:
15     virtual string Odglos();
16     void SetDlugosc_dzioba(float nowaDlugosc);
17     float GetDlugosc_dzioba();
18     Pingwin();
19     virtual ~Pingwin();
20 };
```

Listing 7.20

```
Zwierze::Zwierze()
{
  Ilosc_nog = 0;
}

string Zwierze::Odglos()
{
  return "To zwierze nie wydaje odgłosu";
}

Slon::Slon()
{
  Ilosc_nog = 4;
  Dlugosc_traby = 2.5;
}

string Slon::Odglos()
{
  return "Słoń chrząka lub trąbi";
}

Lew::Lew()
{
  Ilosc_nog = 4;
}

string Lew::Odglos()
{
  return "Lew ryczy";
}

bool Lew::Pozywienie(Zwierze* zw){//będzie później zaimpl.
  return false;
}

Pingwin::Pingwin()
{
  Ilosc_nog = 2;
  Dlugosc_dzioba = 0.375;
}

string Pingwin::Odglos()
{
  return "Pingwin kwili";
}
```

Listing 7.21
```
1  void main()
2  {
3    Zwierze* zoo[10];
4    Slon* s;
5    s = new Slon();
6    s->SetDlugosc_traby(5.0);
7    zoo[0] = s;
8    cout << "Zwierze w klatce ma " << zoo[0]->GetIlosc_nog() <<
9            " nóg\n";
10   cout << "Długość trąby słonia " <<
11           ((Slon*)zoo[0])->GetDlugosc_traby();
12   delete s;
13 }
```

7.7. Dynamiczne tworzenie obiektów

W niniejszym podrozdziale zostaną omówione operatory `new` oraz `delete` należące do języka C++. Ich zadaniem jest utworzenie nowej instancji klasy oraz jej likwidacja, gdy nie będzie już potrzebna. Dotychczas wykorzystywane były deklaracje lokalne, dla których wywoływanie konstruktorów i destruktorów wykonywał samoczynnie kompilator. Dzięki dynamicznemu tworzeniu obiektów można zobaczyć różnice między działaniem zwykłej metody a metody wirtualnej.

Deklarując tablicę wskaźników do klasy `Zwierze`, uzyskuje się możliwość przechowania wskazania do instancji obiektów dowolnej klasy dziedziczącej z klasy podstawowej `Zwierze`. Przykład deklaracji takiej tablicy przedstawiono w linii 3 listingu 7.21 W linii 4 zadeklarowano wskaźnik `s` do klasy `Slon`, a w linii 5 został on zainicjowany instancją klasy `Slon` poprzez operator `new`. Operator `new` wywołał bezparametrowy konstruktor klasy zdefiniowany na listingu 7.20. Ponieważ konstruktor ten ustala początkową długość trąby słonia na wartość `2.5`, w celu zmiany tej wartości posłużono się konstrukcją z linii 6 listingu 7.21, w której wykorzystano operator strzałki `->` wybierający pole lub metodę z zadanego wskaźnika (operator ten był wcześniej używany w rozdziale przy dostępie do pola poprzez wartość `this`). Wskaźnik do klasy `Slon` może być wykorzystany w miejscu, gdzie wskaźnik do klasy `Zwierze` jest spodziewany. Dopuszcza to zapis użyty w linii 7, gdzie pod zerowym indeksem tablicy `zoo` został zapisany (skopiowany) wskaźnik do klasy `Slon`. Wskaźnik do klasy `Zwierze` pozwala tylko na dostęp do pól i metod zapisanych w klasie `Zwierze` oraz wszystkich klas bazowych z których dziedziczy `Zwierze`, pod warunkiem że zakres widoczności pola lub metody umożliwia dostęp do niej. Przykład wykorzystania wspólnego pola `Ilosc_nog` przedstawiono w linii 8 omawianego listingu. Dostęp do pól

i metod z klasy nadrzędnej[8] może być zrealizowany jednym z dwóch możliwych sposobów. Pierwszy może być zastosowany, jeżeli programista ma całkowitą pewność, że pod danym wskaźnikiem jest obiekt wymaganego przez nas typu, wówczas można wykorzystać konwersję wskaźników użytą w linii 11 listingu 7.21, w której za pomocą konstrukcji `((Slon*)zoo[0])` można wykonać konwersję wskaźnika na `Zwierze` do postaci wskaźnika na `Slon`. Drugi sposób wykorzystuje operator `dynamic_cast`, który sprawdza, czy taka konwersja wskaźników klas jest możliwa, jednakże wymaga to włączenia w kompilatorze mechanizmu RTTI (ang. *Run-Time Type Information*). Więcej na temat tego operatora będzie przedstawione podczas implementacji metody `Lew::Pozywienie`.

Ponieważ obiekt klasy `Slon` został utworzony za pomocą operatora `new`, należy go zwolnić za pomocą operatora `delete`. Przykład wywołania tego operatora został przedstawiony w linii 12 na listingu 7.21. Podczas zwalniania pamięci przeznaczonej dla tego obiektu należy pamiętać, że późniejsze (błędne) odwołanie się do niej może skutkować niepoprawną pracą programu. Oznacza to, że wykorzystanie obecnej wartości zmiennej `s` czy odwołanie się do jej kopii w tablicy `zoo` pod indeksem `0` może skutkować nawet przerwaniem pracy programu. Niezwolnienie pamięci przydzielonej dla obiektu i pozostawienie go, gdy już nie jest on potrzebny, powoduje bezpowrotną jej utratę (w tej aplikacji), zwaną „wyciekiem pamięci” (ang. *memory leak*). Dostęp do zwolnionych lub niezainicjowanych obiektów czy szukanie wycieków pamięci to typowe problemy programistów piszących w języku C++. W związku z tym pisanie profesjonalnych aplikacji w tym języku powinno powierzać się doświadczonym specjalistom.

Następnie zostanie omówione wykorzystanie metod wirtualnych. Inicjując kolejne elementy tablicy instancjami różnych klas, jak to przedstawiono w liniach 6–13 na listingu 7.22, można uzyskać różne wartości zwrócone przez metodę `Odglos`. Ponieważ elementy tablicy `zoo` nie zostały zainicjowane przez kompilator, w pętli `for` znajdującej się w liniach 4 i 5 ustalono je na wartość `NULL`. W ten sposób będzie można sprawdzić, czy dany element tablicy został już „obsadzony” jakimś zwierzęciem. Aby sprawdzić, czy element w tablicy jest zainicjowany, wystarczy porównać go z wartością `NULL`, jak to wykonano w linii 16 omawianego listingu. Jeżeli jest to wartość różna od `NULL`, to możemy wywołać metodę `Odglos`, która w wyniku zwróci łańcuch tekstowy właściwy dla odgłosu danego zwierzęcia będącego w tej klatce. Omawiany program wydrukuje na ekranie następujące wyniki:

```
Odgłos z klatki 0: Słoń chrząka lub trąbi i ma 4 nogi
Odgłos z klatki 1: Lew ryczy i ma 4 nogi
Odgłos z klatki 2: Słoń chrząka lub trąbi i ma 4 nogi
Odgłos z klatki 3: Pingwin kwili i ma 2 nogi
```

[8] Nie dotyczy to metod wirtualnych.

Listing 7.22

```
1  void main()
2  {
3    Zwierze* zoo[10];
4    for(int i = 0; i < 10; i++)
5      zoo[i] = NULL;
6    zoo[0] = new Slon();
7    zoo[1] = new Lew();
8    zoo[2] = new Slon();
9    zoo[3] = new Pingwin();
10   zoo[4] = new Pingwin();
11   zoo[5] = new Lew();
12   zoo[6] = new Slon();
13   zoo[8] = new Pingwin();
14   for(int j = 0; j < 10; j++)
15   {
16     if(zoo[j] == NULL)
17         cout << "Klatka " << j << " jest pusta\n";
18     else
19     cout << "Odgłos z klatki " << j << ": " <<
20       zoo[j]->Odglos().c_str() << " i ma " <<
21       zoo[j]->GetIlosc_nog() << " nogi\n";
22   }
23   for(int k = 0; k < 10; k++)
24   {
25     if(zoo[k] != NULL)
26       delete zoo[k];
27   }
28 }
```

```
Odgłos z klatki 4: Pingwin kwili i ma 2 nogi
Odgłos z klatki 5: Lew ryczy i ma 4 nogi
Odgłos z klatki 6: Słoń chrząka lub trąbi i ma 4 nogi
Klatka 7 jest pusta
Odgłos z klatki 8: Pingwin kwili i ma 2 nogi
Klatka 9 jest pusta
```

Świadczą one o tym, że jest uruchamiana ta metoda wirtualna `Odglos`, która została zdefiniowana (bliższym stwierdzeniem byłoby użycie słowa „nadpisana”) przez daną klasę, niezależnie od wskaźnika typu. Mechanizm ten wykorzystuje się często w sytuacji, gdy jest wiele podobnych klas, a każda z nich ma realizować pewną ogólną operację, ale w nieco odmienny sposób.

Przy założeniu, że każda klasa definiuje pewną metodę wirtualną, która zwracałaby unikatowe wartości, możliwe jest jednoznaczne rozróżnienie klasy, do której należy ten obiekt. Ponieważ takie założenie może być trudne do realiza-

Listing 7.23
```
1  bool Lew::Pozywienie(Zwierze* zw)
2  {
3    Slon* s;
4    s = dynamic_cast<Slon*>(zw);
5    if(s != NULL)
6      return true;
7    Pingwin* p;
8    p = dynamic_cast<Pingwin*>(zw);
9    if(p != NULL)
10     return true;
11   return false;
12 }
```

cji, zwłaszcza w przypadku gdy jest dużo klas, a niemożliwe gdy otrzymujemy napisane już klasy bez kodu źródłowego, to właściwym rozwiązaniem jest użycie wspomnianego wcześniej operatora `dynamic_cast` z mechanizmem RTTI. Przykład takiego rozwiązania został zaprezentowany w implementacji metody `Pozywienie`, którą przedstawiono na listingu 7.23. Operator `dynamic_cast` wymaga dwóch operandów: pierwszy z nich jest podawany w nawiasach trójkątnych < i >, a drugi w nawiasach okrągłych (i). W pierwszym operandzie zapisuje się wymagany typ konwersji, a w drugim wyrażenie do konwersji. W przypadku

Listing 7.24
```
1  void main()
2  {
3    Zwierze* zoo[10];
4    Lew* l = new Lew();
5    //jak linie 4 - 13 z listingu 7.22
6    for(int j = 0; j < 10; j++)
7    {
8      if(zoo[j] != NULL)
9      {
10       if(l->Pozywienie(zoo[j]))
11         cout << "Lew żywi się zwierzęciem " <<
12           zoo[j]->Odglos().c_str() << "\n";
13       else
14         cout << "Lew nie lubi " << zoo[j]->Odglos().c_str() <<
15           "\n";
16     }
17   }
18   //jak linie 23 - 27 z listingu 7.22
19   delete l;
20 }
```

gdy wyrażenie nie może być przekształcone do wymaganego typu, to operator `dynamic_cast` zwraca `NULL`. Prezentowany na listingu 7.23 kod sprawdza, czy klasą przekazaną jako parametr jest `Slon` lub `Pingwin`. Jeżeli tak, to zwracana jest wartość `true`, w przeciwnym razie zwracana jest wartość `false`.

Mechanizm RTTI jest domyślnie włączony dla programów pisanych w C++ kompilowanych przez Visual Studio od wersji 2010. We wcześniejszych wersjach bez włączonej opcji RTTI zastosowanie operatora `dynamic_cast` powodowało zgłoszenie ostrzeżenia na etapie kompilacji z informacją dotyczącą sugerowanej zmiany opcji kompilacji. Sposób na włączenie właściwej opcji do obsługi RTTI należy znaleźć w dokumentacji używanego kompilatora. Zazwyczaj jest to przełącznik w jednym z okien parametrów kompilacji (kompilatory z IDE) albo dodatkowy parametr w linii poleceń (kompilatory uruchamiane z linii poleceń).

Na listingu 7.24 przedstawiono kod funkcji `main`, w którym – podobnie jak w listingu 7.22 – dla każdego elementu w tablicy `zoo` sprawdza się, czy zwierzę to stanowi pożywienie dla lwa. Jak można zauważyć, w linii 10 operator strzałki `->` może być również wykorzystywany do wywoływania metod, gdy lewa jego strona jest wskaźnikiem.

Warto podkreślić, że niniejsze wprowadzenie do języka C++ miało charakter informacyjny. Pełny opis dostępnych metod dla standardowych klas (takich jak `string`) można znaleźć w literaturze dotyczącej tego języka [11] czy też na stronach internetowych, takich jak `http://www.cplusplus.com/`. Kompilatory wyposażone w IDE bardzo często udostępniają podpowiedź kontekstową do wpisywanego kodu, umożliwiającą „odkrycie" nowych funkcji standardowych klas, których dokumentację szybo można odnaleźć w Internecie.

7.8. Zadania

1. Do przeliczenia roku kalendarza gregoriańskiego na kalendarz muzułmański (z dokładnością do jednego roku) stosuje się przedstawiony wzór. Napisz program w języku C++, który dla podanego roku gregoriańskiego z przedziału lat 2000–2099 oznaczonego przez R obliczy rok muzułmański H.

$$H = R - 622 + \frac{R - 621}{32}$$

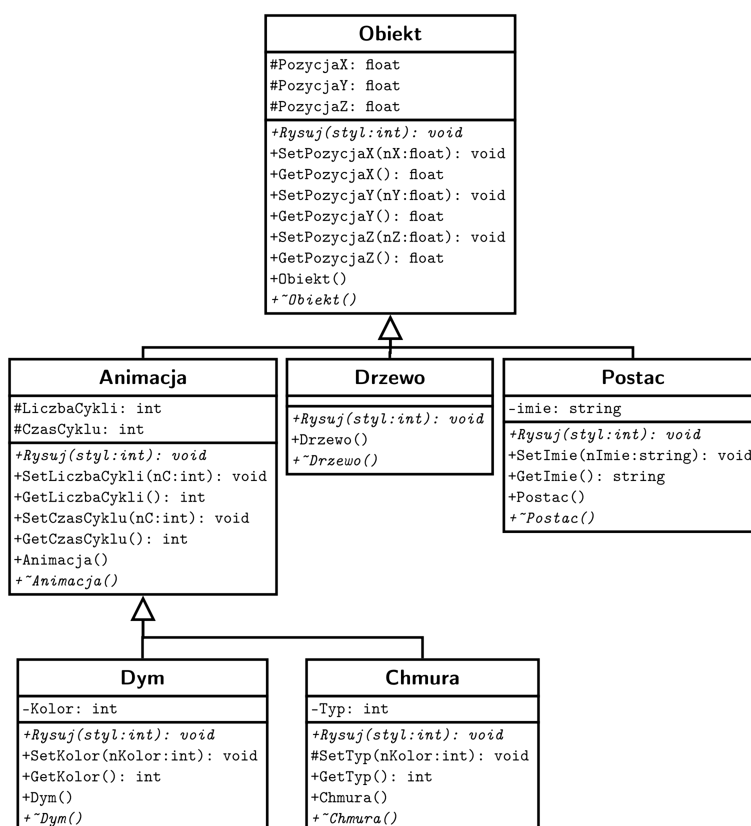
2. Do wyznaczenia napięcia U na kondensatorze w stanie nieustalonym służy podany wzór. Napisz program w języku C++, który obliczy napięcie na kondensatorze dla podanej z klawiatury wartości czasu t, gdy $t > 0$. Zakłada się, że obwód zawiera rezystancję $R = 22k\,\Omega$ i kondensator o pojemności $C = 10\,\mu F$ podłączone do źródła napięcia $U_0 = 7\,V$.

$$U = U_0 \cdot e^{-\frac{t}{R \cdot C}}$$

3. Napisz program w języku C++ do wyliczenia kapitału końcowego FV przy miesięcznych wpłatach w wysokości $kwota$ przez liczbę miesięcy wyrażoną w zmiennej $mies$ według podanego wzoru. W obliczeniach można przyjąć, że stopa procentowa $rate$ jest równa $0,0024$.

$$FV = \sum_{t=1}^{mies} kwota \cdot (1 + rate)^{mies-t}$$

4. Napisz deklarację klas, pól oraz metod w języku C++ zgodnie z koncepcją wyrażoną w postaci prezentowanego diagramu UML.



5. Zaimplementuj konstruktor klasy `Dym` z poprzedniego zadania, ustawiając wartości pól zgodnie z tabelą.

Nazwa	Typ	Wartość
`PozycjaX`	`float`	`24.5`
`PozycjaY`	`float`	`82.15`
`PozycjaZ`	`float`	`-12.781`
`LiczbaCykli`	`int`	`2`
`CzasCyklu`	`int`	`20000`
`Kolor`	`int`	`0x00C0C0C0`

6. Utwórz deklarację metody `CzyAnimowany` dla klasy `Obiekt` zwracającej typ `bool` dla podanego argumentu będącego wskaźnikiem do klasy `Obiekt`.

7. Utwórz implementację metody `CzyAnimowany` z poprzedniego zadania sprawdzającej, czy obiekt dziedziczy z klasy `Animacja`. Można wykorzystać operator `dynamic_cast`.

Rozdział 8.

Podsumowanie

Celem publikacji było systematyczne wprowadzenie do zagadnień arytmetyki i architektury komputerów, algorytmów i struktur danych, wybranych języków (C, C++ i Ruby) i paradygmatów programowania. Ze względu na ograniczoną objętość publikacji pewne zagadnienia zostały potraktowane w sposób selektywny. Autorzy mają nadzieję, że przedłożona praca stanowi dobre wyjście do podjęcia samodzielnych studiów nad wybranymi działami informatyki, które zainteresowały Czytelnika.

Kontynuację studiów z zagadnień architektury i arytmetyki komputerów można rozpocząć od prac [16, 23]. Przystępną pozycją wprowadzającą do zagadnień algorytmiki jest [34]. Studia nad tym działem informatyki można kontynuować w oparciu o publikacje [28, 3, 6]. Swoje zainteresowania dotyczące programowania w języku Ruby warto poszerzać o opracowanie [9]. Lista publikacji dotycząca programowania w języku C jest szczególnie pokaźna. Autorzy sugerują podjęcie dalszej nauki programowania w tym języku, rozpoczynając od pracy [26] i kontynuując na podstawie [17] oraz [22]. Naukę programowania w języku C++ warto kontynuować opierają się na opracowaniu [11]. Bardziej sformalizowane podejście do opisu tego języka prezentuje praca [27].

Autorzy zachęcają również do sięgnięcia po kontynuację niniejszego skryptu obejmującą wprowadzenie do sieci komputerowych, baz danych, systemów operacyjnych i technik sztucznej inteligencji.

Spis rysunków

3.1. Zamiana liczby 38 na system dwójkowy 15
3.2. Konwersja między systemem heksadecymalnym a binarnym . . . 16
3.3. Sumowanie liczb binarnych 19
3.4. Mnożenie liczb binarnych 19
3.5. Sumowanie liczb w kodzie ZU2 19
3.6. Funkcja OR 21
3.7. Funkcja AND 21
3.8. Funkcja NOT 21
3.9. Funkcja XOR 22
3.10. Funkcja NOR 22
3.11. Funkcja NAND 22
3.12. Tablica prawdy półsumatora 23
3.13. Tablica prawdy sumatora 23

4.1. Bloki początkowy i końcowy 32
4.2. Blok operacyjny 32
4.3. Bloki wejściowy i wyjściowy 32
4.4. Blok warunkowy 33
4.5. Program sumujący dwie liczby – schemat blokowy 33
4.6. Program wyświetlający komunikat uzależniony od znaku sumy . . 34
4.7. Program sumujący wiele liczb 36
4.8. Algorytm Euklidesa 38
4.9. Przykład działania algorytmu Euklidesa 39
4.10. Przykład działania procedury rekurencyjnej 41
4.11. Przykład graficznej reprezentacji tablicy 41
4.12. Przykład listy jednokierunkowej 42
4.13. Przykładowe operacje na stosie 43
4.14. Przykład działania algorytmu sortowania przez selekcję 45
4.15. Kolejne kroki sortowania przez scalanie 47

6.1. Schemat blokowy instrukcji warunkowej 111
6.2. Schemat blokowy skróconej instrukcji warunkowej 112
6.3. Schemat blokowy pętli `while` 114
6.4. Schemat blokowy pętli `do-while` 116
6.5. Schemat blokowy pętli `for` 117
6.6. Schemat blokowy użycia instrukcji zaniechania `break` 120
6.7. Schemat blokowy instrukcji `switch` 123
6.8. Tablica jednowymiarowa 128
6.9. Tablica dwuwymiarowa 129
6.10. Tablica trójwymiarowa 129

7.1. Schemat dziedziczenia klas w języku UML 152

Spis tablic

3.1. Porównanie systemów liczbowych 15
3.2. Porównanie kodowań ZM, ZU1, ZU2 17
3.3. Funkcje arytmometru SN74181, na podstawie [30] 24

6.1. Słowa kluczowe języka C 95
6.2. Najczęściej spotykane znaki specjalne 98
6.3. Wybrane funkcje odczytu i zapisu do plików 136

Bibliografia

[1] Antosiewicz W., Kilen N., Komputer dla mas. *Mówią wieki*, 7:28–32, 2001.

[2] Atmel. AVR Enhanced RISC C Code Benchmark, 1998, `www.atmel.com/Images/doc1292.pdf`.

[3] Banachowski L., Diks K., Rytter W., *Algorytmy i struktury danych*. WNT, Warszawa 2006.

[4] Bennet S., McRobb S., Farmer R., *Object-Oriented Systems Analysis and Design Using UML*. McGraw-Hill Education, Maidenhead 2006.

[5] Black. D. A., *The Well Grounded Rubbyist*. Manning Publications Co., Greenwich 2009.

[6] Cormen T. H., Leiserson C. E., Rivest R. L., Stein C., *Wprowadzenie do algorytmów*. PWN, Warszawa 2012.

[7] Ćwirko R., Rusek M., Marciniak W., *Układy scalone w pytaniach i odpowiedziach*. WNT, Warszawa 1987.

[8] Doliński J., *Mikrokontrolery AVR w praktyce*. Wydawnictwo BTC, Warszawa 2003.

[9] Falangan D., Matsumoto Y., *Ruby. Programowanie*. Helion, Gliwice 2009.

[10] Grajek M., *Enigma. Bliżej prawdy*. Dom Wydawniczy Rebis, Poznań 2007.

[11] Grębosz J., *Symfonia C++ Standard*. Edition 2000, Kraków 2008.

[12] IEEE Computer Society. *IEEE-754. IEEE Standard for Floating-Point Arithmetic*, 2008.

[13] ISO/IEC 9899:1990. Programming languages – C, 1990.

[14] ISO/IEC 14882:1998. Information technology – Programming languages – C++, 1998.

[15] Jędrzejec B., Sadolewski J., *Programowanie w języku C i C++, Skrypt dla informatyków i automayków (w przygotowaniu)*. Oficyna Wydawnicza Politechniki Rzeszowskiej, Rzeszów 2014.

[16] Kalisz J., *Podstawy elektroniki cyfrowej*. WKŁ, Warszawa 2007.

[17] Kernighan B. W., Ritchie D. M., *Język ANSI C*. Wydawnictwa Naukowo-Techniczne, Warszawa 2000.

[18] Knuth D. E., *Sztuka programowania*. WNT, Warszawa 2002.

[19] Ligonnière R., *Prehistoria i historia komputerów*. Zakład Narodowy im. Ossolińskich, Wrocław 1992.

[20] Myklebust G., The AVR microcontroller and C compiler co–design. *3rd Euro. Microprocessor and Microcontroller Seminar*, str. 164–170, Heathrow, U.K., Nov. 1996.

[21] Papadimitriou C. H., *Złożoność obliczeniowa*. WNT, Warszawa 2002.

[22] Petzold C., *Programowanie Windows*. Wydawnictwo RM 1999.

[23] Petzold C., *Kod. Ukryty język sprzętu komputerowego i oprogramowania*. Wydawnictwa Naukowo-Techniczne 2002.

[24] Ross K., Wright C., *Matematyka dyskretna*. PWN, Warszawa 2000.

[25] Sacha K., Rydzewski A., *Mikroprocesor w pytaniach i odpowiedziach*. WNT, Warszawa 1985.

[26] Schildt H., *Programowanie C*. Wydawnictwo RM 2002.

[27] Stroustrup B., *Język C++*. Wydawnictwa Naukowo-Techniczne, Warszawa 2002.

[28] Świder K., *Wykłady z algorytmów i struktur danych z zadaniami*. Oficyna Wydawnicza Politechniki Rzeszowskiej, Rzeszów 2004.

[29] Świder Z., *Realizacje cyfrowe algorytmów sterowania i filtracji. Błędy zmiennoprzecinkowe i zaokrąglenia współczynników*. Oficyna Wydawnicza Politechniki Rzeszowskiej, Rzeszów 2003.

[30] Texas Instruments. *The TTL Logic Data Book*, 1999.

[31] Waszkiewicz J., Z prehistorii i historii komputerów – uwagi na marginesie pewnej lektury. *Matematyka*, 5/1994:259–278, 1994.

[32] Waszkiewicz J., Kto skonstruował pierwszy komputer? (cz. 2). *Matematyka*, 6/1999:323–329, 1999.

[33] Waszkiewicz J., Kto skonstruował pierwszy komputer? *Matematyka*, 5/1999:259–268, 1999.

[34] Wirth N., *Algorytmy + struktury danych = programy*. Wydawnictwa Naukowo-Techniczne 2000.

[35] Zwoliński A., *Obraz w relacjach społecznych*. Wydawnictwo WAM, Kraków 2004.

Skorowidz

A
akumulator 25
algebra Boole'a 20
algorytm 31
  Euklidesa 38
ALU *zob.* arytmometr
architektura
  harwardzka 25
  von Neumanna 25
arytmometr 24
asocjacyjna, tablica *zob.* tablica asocjacyjna
`attr_reader` 81
`attr_writer` 81

B
bajt 16
`begin-rescue` 77
bit 14
blok
  końcowy 32
  operacyjny 32
  początkowy 32
  warunkowy 32, 34
  wejściowy 32
  wyjściowy 32
blok kodu 66, 86
`bool` 147
bramka logiczna 22
`break` 62, 119

C
C 93
C++ 139
cecha (liczby zmiennoprzec.) 18
`char` 96
`chomp` 55
`cin` 143
`const` 145
`cout` 142

D
`def` 68
dekoder rozkazów 25
`DEQUEUE` 43
destruktor 144, 156
`do` 66
dopełnienie 16, 20
`double` 96
`do-while` 115
dziedziczenie 77, 151

E
`each` 62
`else` 57, 111
`elsif` 57
`empty?` 68
`end` 66
enkapsulacja
  metody enkaps. 149, 153
  pól 149
`ENQUEUE` 43

`enum` 96
exception 76

F
`fclose` 134
`fgetc` 136
`fgets` 136
`Fixnum` 56
`Float` 56
`float` 96
`fopen` 132
`for` 62, 116
`fprintf` 136
`fputc` 136
`fputs` 136
`fscanf` 136
funkcja 68, 124
    jako parametr funkcji 86
    jako wynik funkcji 86
    skutki uboczne 86
funkcja logiczna
    AND 21
    NAND 22
    NOR 22
    NOT 21
    OR 21
    XOR 22

G
generacja komputera 11
`getchar` 102
`gets` 55, 102
`getter` 81
gigabajt 16

H
hash table *zob.* tablica asocjacyjna
Hello world 52
hermetyzacja obiektu 150

I
`if` 57, 111
`initialize` 74
instancja 141, 144, 145, 156
instrukcja warunkowa 35, 56
instrukcje zagnieżdżone 57
`int` 96
interpolacja 54
`iostream` 142
iteracja 61

J
jednostka
    arytmetyczno-logiczna *zob.* arytmometr
    sterująca 24
język narodowy 53
język polski 53

K
kilobajt 16
klasa 71, 140
    bazowa 78, 152
    deklaracja 141, 153
    pochodna 78
kodowanie
    IEEE-754 18
    naturalne binarne 16
    NB *zob.* kodowanie, naturalne binarne
    stałoprzecinkowe 17
    ZM *zob.* kodowanie, znak-moduł
    zmiennoprzecinkowe 17
    znak-moduł 16
    znak-uzupełnienie do 1 16
    znak-uzupełnienie do 2 16
      działania
    ZU1 *zob.* kodowanie, znak-uzupełnienie do 1
    ZU2 *zob.* kodowanie, znak-uzupełnienie do 2
kolejka FIFO 43

- komentarz 53
- komunikat 71–73
- konstruktor 74, 144, 156
  - dziedziczenie 79

L

- liczby
  - całkowite 56
  - zmiennoprzecinkowe 56 *zob.* kodowanie, zmiennoprzecinkowe
- licznik rozkazów 25
- lista 41, 63
  - definicja rekurencyjna 42
  - dwukierunkowa 41

M

- magistrala
  - adresowa 25
  - danych 25
  - sterująca 25
- manipulator strumieniowy 142
- mantysa 18
- megabajt 16
- metoda 72, 140, 146, 149, 151
  - ograniczenie dostępu 73
  - prywatna 73
  - publiczna 73
  - wirtualna 152, 158
- mikroprocesor 11, 24–26
  - AVR 26–28
- model obliczeń 31

N

- `namespace` 142
- `new` 72
- `next` 62
- `nil` 54
- notacja
  - $\mathcal{O}$ 46
  - Ω 46
  - Θ 46

O

- obiekt 71, 73, 141, 144
- Objective C 139
- obsługa błędów 76
- operacja dominująca 45
- operator C++
  - `.` (kropki) 140
  - `::` (zakresu) 144
  - `<<` 142
  - `->` 145, 160
  - `>>` 143
  - `delete` 156
  - `dynamic_cast` 157, 159
  - `new` 156
- operator C/C++
  - `!` 108
  - `&` 109
  - `&&` 108
  - `*` 109
  - `.` 109
  - `<` 107
  - `<=` 107
  - `!=` 107
  - `==` 107
  - `>` 107
  - `->` 109
  - `>=` 107
  - `||` 108
- operator Ruby
  - `<` 57
  - `<=` 57
  - `==` 57
  - `=>` 57
  - `>` 57
  - `and` 57
  - `not` 57
  - `or` 57
- `ostream` 142

P

- pamięć 24

EEPROM 25
o dostępie swobodnym 25
RAM *zob.* pamięć, o dostępie swobodnym
ROM *zob.* pamięć, tylko do odczytu
tylko do odczytu 25
paradygmaty programowania 49
pętla 35–38, 61
podprogram 39, 67
pole klasy 71, 140, 145, 146, 151
dostęp 73
polimorfizm 87
`POP` 42
porządek
FIFO 42
LIFO 42
liniowy 44
półsumator 23
prawo
De Morgana 29
dopełnienia 20
idempotentności 29
identyczności 20
łączności 20
pochłaniania 29
przemienności 20
rozdzielności 20
`print` 52
`printf` 100, 103
`private` 73, 141
procedura *zob.* podprogram
iteracyjna 39
rekurencyjna 39
programowanie
deklaratywne 80
funkcjonalne 83
funkcyjne *zob.* programowanie, funkcjonalne
imperatywne 80
obiektowe 71
proceduralne 67
strukturalne 65
zdarzeniowe 89
`protected` 140
przeciążanie
metod 144, 147
operatorów 142
przestrzeń nazw 142
przysłonięcie pola 145
pseudojęzyk *zob.* pseudokod
pseudokod 33
`public` 140
`PUSH` 42
`putchar` 100
`puts` 52, 100

R

`raise` 76
referencja 143
rejestr 25
rozkazów 25
stanu 25
rekurencja 39, 84 *zob.* rekurencja
rekursja *zob.* rekurencja
relacja częściowego porządku 44
`rescue` 77
`return` 68, 123
RISC 26
rozmiar zadania 44
RTTI 157, 159
Ruby 50
instalacja 50
interpreter 50
uruchomienie programu 50
Run-Time Type Information zob. RTTI
`RuntimeError` 76

S

`scanf` 102

schemat blokowy 32
`setter` 81
silnia 40
słowo 16
  podwójne 16
  szesnastobitowe *zob.* słowo
  trzydziestodwubitowe *zob.* słowo, podwójne
sortowanie 44
  przez scalanie 46
  przez wybieranie 44
SQL 82
stan obiektu 71, 72
`std` 142
`stdout` 142
stos 42
`string` 150
struktura 140
struktury danych 40–43, 63
strumień 142
sumator 23
  z przeniesieniem równoległym 24
  z przeniesieniem szeregowym 23
`super` 79
`switch` 121
symbol 64
system funkcjonalnie pełny 22
system liczbowy
  binarny *zob.* system liczbowy, dwójkowy
  dwójkowy 14
    działania 18, 19
  dziesiętny 13
  heksadecymalny *zob.* system liczbowy, szesnastkowy
  podstawa 14
  pozycyjny 14
  szesnastkowy 14
  zamiana 14–16

T
tablica 40, 63, 127
  indeks 63
tablica asocjacyjna 64
tekst, wypisywanie 52
`this` 145
`times` 61
`to_f` 56
`to_i` 56

U
układy wejścia/wyjścia 24
UML 151
`unless` 59
UTF-8 53

V
`virtual` *zob.* metoda, wirtualna
`void` 96

W
`while` 61, 114
właściwość 71
wyciek pamięci 157
wyjątek 76
  generowanie (zgłaszanie) 76
  obsługa (przechwycenie) 77

Z
zakres widoczności 140, 149, 153
zaprzyjaźnienia klas 141
zasada dualności 21
złożoność
  asymptotyczna 44
  czasowa 44
  obliczeniowa 44
  pamięciowa 44
zmienna 54
  globalna 70
  lokalna 70