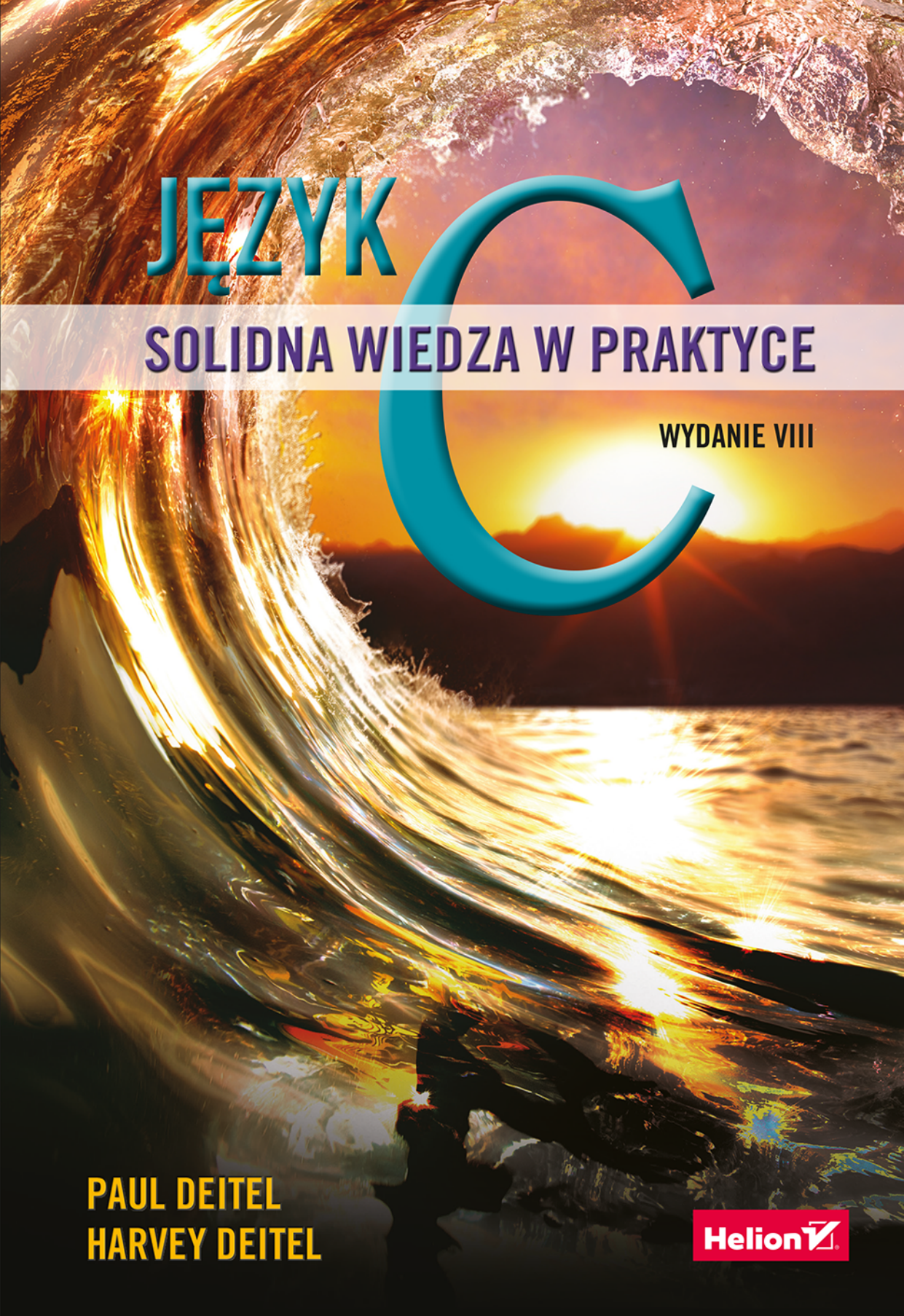




JĘZYK

SOLIDNA WIEDZA W PRAKTYCE

WYDANIE VIII

**PAUL DEITEL
HARVEY DEITEL**

Helion 

Tytuł oryginału: C How to Program (8th Edition)

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-6286-4

Authorized translation from the English language edition, entitled C HOW TO PROGRAM, 8th Edition by DEITEL, PAUL J.; DEITEL, HARVEY, published by Pearson Education, Inc, publishing as Pearson, Copyright © 2016, 2013, 2010 Pearson Education, Inc., Hoboken, NJ 07030.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

POLISH language edition published by Helion SA, Copyright © 2020.

Microsoft® and Windows® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries.

DEITEL, the double-thumbs-up bug and DIVE INTO are registered trademarks of Deitel and Associates, Inc.

Apple, Xcode, Swift, Objective-C, iOS and OS X are trademarks or registered trademarks of Apple, Inc. Java is a registered trademark of Oracle and/or its affiliates.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/jcsol8>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/jcsol8.zip>

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Wprowadzenie	19
Rozdział 1. Wprowadzenie do informatyki, internetu i sieci	31
1.1. Wprowadzenie	32
1.2. Hardware i software	32
1.2.1. Prawo Moore’a	32
1.2.2. Organizacja komputera	33
1.3. Hierarchia danych	35
1.4. Język maszynowy, język assemblera i język wysokiego poziomu	37
1.5. Język programowania C	38
1.6. Biblioteka standardowa C	40
1.7. C++ i inne oparte na C języki programowania	41
1.8. Technologia obiektowa	42
1.8.1. Samochód jako obiekt	42
1.8.2. Metoda i klasa	42
1.8.3. Tworzenie egzemplarza	42
1.8.4. Wielokrotne używanie klasy	43
1.8.5. Komunikat i wywołanie metody	43
1.8.6. Atrybut i zmienna egzemplarza	43
1.8.7. Hermetyzacja i ukrywanie informacji	43
1.8.8. Dziedziczenie	43
1.9. Typowe środowisko programowania w języku C	44
1.9.1. Faza 1. — tworzenie programu	45
1.9.2. Fazy 2. i 3. — przetwarzanie i kompilowanie programu	45
1.9.3. Faza 4. — linkowanie programu	45
1.9.4. Faza 5. — wczytywanie programu	46
1.9.5. Faza 6. — wykonywanie programu	46
1.9.6. Problemy, które mogą pojawić się podczas wykonywania programu	46
1.9.7. Standardowe strumienie wejścia, wyjścia i błędów	46
1.10. Przykładowa aplikacja w języku C utworzona na platformach Windows, Linux i macOS ...	46
1.10.1. Uruchomienie aplikacji napisanej w C w oknie wiersza poleceń systemu Windows	47
1.10.2. Uruchomienie aplikacji napisanej w C za pomocą GNU C w systemie Linux	50
1.10.3. Uruchomienie aplikacji napisanej w C za pomocą okna narzędzia Terminal w systemie macOS	52

1.11.	System operacyjny	54
1.11.1.	Windows — własnościowy system operacyjny	55
1.11.2.	Linux — otwarty system operacyjny	55
1.11.3.	Systemy firmy Apple — macOS dla komputerów Mac, iOS dla urządzeń mobilnych iPhone, iPad i iPod touch	55
1.11.4.	System Android firmy Google	56
1.12.	Internet i sieć WWW	56
1.12.1.	Internet — sieć sieci	57
1.12.2.	Sieć WWW — internet przyjazny użytkownikowi	57
1.12.3.	Usługa sieciowa	57
1.12.4.	AJAX	59
1.12.5.	Internet rzeczy	59
1.13.	Wybrana kluczowa terminologia związana z oprogramowaniem	59
1.14.	Na bieżąco z technologiami informatycznymi	62

Rozdział 2. Wprowadzenie do programowania w języku C67

2.1.	Wprowadzenie	68
2.2.	Prosty program w języku C — wyświetlenie wiersza tekstu	68
2.3.	Następny prosty program w języku C — dodawanie dwóch liczb całkowitych	72
2.4.	Koncepcje dotyczące pamięci	77
2.5.	Arytmetyka w języku C	78
2.6.	Podejmowanie decyzji — operatory równości i relacji	82
2.7.	Bezpieczne programowanie w języku C	86

Rozdział 3. Programowanie strukturalne w języku C99

3.1.	Wprowadzenie	100
3.2.	Algorytm	100
3.3.	Pseudokod	100
3.4.	Struktury kontrolne	101
3.5.	Polecenie wyboru if	102
3.6.	Polecenie wyboru if-else	103
3.7.	Polecenie iteracji while	107
3.8.	Studium przypadku tworzenia algorytmu 1 — iteracja kontrolowana przez licznik	108
3.9.	Studium przypadku tworzenia algorytmu 2 — iteracja kontrolowana przez wartownik	111
3.10.	Studium przypadku tworzenia algorytmu 3 — zagnieżdżone polecenia kontrolne	117
3.11.	Operatory przypisania	121
3.12.	Operatory inkrementacji i dekrementacji	121
3.13.	Bezpieczne programowanie w języku C	124

Rozdział 4. Struktury warunkowe programu w języku C143

4.1.	Wprowadzenie	144
4.2.	Podstawy iteracji	144
4.3.	Iteracja oparta na liczniku	144
4.4.	Konstrukcja for	146
4.5.	Konstrukcja — uwagi i obserwacje	149
4.6.	Przykłady użycia polecenia for	150
4.7.	Konstrukcja switch umożliwiającą wybór spośród wielu możliwości	153
4.8.	Konstrukcja do-while	159
4.9.	Polecenia break i continue	160

4.10.	Operatory logiczne	162
4.11.	Mylenie operatorów równości (==) i przypisania (=)	165
4.12.	Podsumowanie programowania strukturalnego	166
4.13.	Bezpieczne programowanie w języku C	171

Rozdział 5. Funkcje w języku C 187

5.1.	Wprowadzenie	188
5.2.	Modularyzacja programów w języku C	188
5.3.	Funkcje biblioteki matematycznej	189
5.4.	Funkcje	191
5.5.	Definicja funkcji	191
5.5.1.	Funkcja square()	191
5.5.2.	Funkcja maximum()	195
5.6.	Więcej o prototypie funkcji	196
5.7.	Stos wywołań funkcji i stos ramek	199
5.8.	Nagłówki	202
5.9.	Przekazywanie argumentów przez wartość i przez referencję	203
5.10.	Generowanie liczb losowych	204
5.11.	Przykład — gra hazardowa i wprowadzenie typu enum	208
5.12.	Klasy przechowywania	211
5.13.	Reguły dotyczące zasięgu	213
5.14.	Rekurencja	216
5.15.	Przykład użycia rekurencji — ciąg Fibonacciego	220
5.16.	Rekurencja kontra iteracja	223
5.17.	Bezpieczne programowanie w języku C	225

Rozdział 6. Tablice w języku C 245

6.1.	Wprowadzenie	246
6.2.	Tablica	246
6.3.	Definiowanie tablicy	247
6.4.	Przykłady tablic	248
6.4.1.	Definiowanie tablicy i stosowanie pętli do przypisania wartości elementom	248
6.4.2.	Inicjalizowanie tablicy w definicji za pomocą listy inicjalizacyjnej	249
6.4.3.	Określenie wielkości tablicy za pomocą stałej symbolicznej i inicjalizowanie elementów tablicy na podstawie wyniku obliczeń	250
6.4.4.	Sumowanie elementów tablicy	251
6.4.5.	Stosowanie tablicy do podsumowania wyników ankiety	252
6.4.6.	Wizualizacja wartości elementów tablicy za pomocą histogramu	254
6.4.7.	Rzut kością 60 000 000 razy i podsumowanie wyników za pomocą tablicy	255
6.5.	Stosowanie tablicy znaków do przechowywania ciągów tekstowych i operowania na nich	256
6.5.1.	Inicjalizacja tablicy znaków za pomocą ciągu tekstowego	256
6.5.2.	Inicjalizacja tablicy znaków za pomocą listy znaków	256
6.5.3.	Uzyskiwanie dostępu do znaków ciągu tekstowego	256
6.5.4.	Umieszczanie danych wejściowych w tablicy znaków	257
6.5.5.	Wyświetlanie tablicy znaków przedstawiającej ciąg tekstowy	257
6.5.6.	Przykład użycia tablicy znaków	257
6.6.	Statyczna i automatyczna tablica lokalna	258
6.7.	Przekazywanie funkcji argumentu w postaci tablicy	260
6.8.	Sortowanie tablicy	264

6.9.	Studium przypadku — obliczanie średniej, mediany i dominanty za pomocą tablic	266
6.10.	Wyszukiwanie elementów w tablicy	270
6.10.1.	Wyszukiwanie liniowe elementu w tablicy	271
6.10.2.	Wyszukiwanie binarne elementu w tablicy	272
6.11.	Tablica wielowymiarowa	275
6.11.1.	Prezentacja tablicy o dwóch indeksach	275
6.11.2.	Inicjalizacja tablicy dwuwymiarowej	276
6.11.3.	Definiowanie elementów w wierszu	278
6.11.4.	Sumowanie elementów tablicy dwuwymiarowej	278
6.11.5.	Operacje na tablicy dwuwymiarowej	278
6.12.	Tablica o zmiennej wielkości	281
6.13.	Bezpieczne programowanie w języku C	284

Rozdział 7. Wskaźniki w języku C305

7.1.	Wprowadzenie	306
7.2.	Definiowanie i inicjalizowanie zmiennej wskaźnika	306
7.3.	Operatory wskaźnika	307
7.4.	Przekazywanie argumentów do funkcji przez referencję	309
7.5.	Stosowanie kwalifikatora <code>const</code> ze wskaźnikiem	314
7.5.1.	Konwersja znaków ciągu tekstowego na wielkie za pomocą wskaźnika niebędącego stałą do danych niebędących stałą	315
7.5.2.	Wyświetlanie po jednym znaku ciągu tekstowego za pomocą wskaźnika niebędącego stałą do danych będących stałą	316
7.5.3.	Próba modyfikacji za pomocą wskaźnika będącego stałą do danych niebędących stałą	318
7.5.4.	Próba modyfikacji za pomocą wskaźnika będącego stałą do danych będących stałą	318
7.6.	Sortowanie bąbelkowe z użyciem przekazywania przez referencję	319
7.7.	Operator <code>sizeof</code>	322
7.8.	Wyrażenia i arytmetyka wskaźnika	325
7.8.1.	Dozwolone operatory dla arytmetyki wskaźnika	325
7.8.2.	Wskaźnik prowadzący do elementu tablicy	325
7.8.3.	Dodawanie liczby całkowitej do wskaźnika	325
7.8.4.	Odejmowanie liczby całkowitej od wskaźnika	326
7.8.5.	Inkrementacja i dekrementacja wskaźnika	326
7.8.6.	Odejmowanie wskaźników	327
7.8.7.	Przypisanie jednego wskaźnika innemu	327
7.8.8.	Wskaźnik do <code>void</code>	327
7.8.9.	Porównywanie wskaźników	327
7.9.	Związek między wskaźnikiem i tablicą	328
7.9.1.	Notacja wskaźnika i przesunięcia	328
7.9.2.	Notacja wskaźnika i indeksu	329
7.9.3.	Brak możliwości modyfikacji nazwy tablicy za pomocą arytmetyki wskaźnika	329
7.9.4.	Ustalanie indeksu i przesunięcia wskaźnika	329
7.9.5.	Kopiowanie ciągu tekstowego za pomocą tablicy i wskaźnika	330
7.10.	Tablica wskaźników	332
7.11.	Studium przypadku — symulacja tasowania i rozdawania kart	333
7.12.	Wskaźnik do funkcji	337
7.12.1.	Sortowanie w kolejności rosnącej lub malejącej	337
7.12.2.	Stosowanie wskaźnika do funkcji podczas tworzenia systemu opartego na menu	340
7.13.	Bezpieczne programowanie w języku C	342

Rozdział 8. Znaki i ciągi tekstowe w języku C 363

8.1.	Wprowadzenie	364
8.2.	Podstawy dotyczące znaków i ciągów tekstowych	364
8.3.	Biblioteka obsługi znaków	366
8.3.1.	Funkcje isdigit(), isalpha(), isalnum() i isxdigit()	367
8.3.2.	Funkcje islower(), isupper(), tolower() i toupper()	368
8.3.3.	Funkcje isspace(), iscntrl(), ispunct(), isprint() i isgraph()	370
8.4.	Funkcje konwersji ciągu tekstowego	371
8.4.1.	Funkcja strtod()	371
8.4.2.	Funkcja strtol()	372
8.4.3.	Funkcja strtoul()	373
8.5.	Funkcje biblioteki standardowej wejścia-wyjścia	374
8.5.1.	Funkcje fgets() i putchar()	374
8.5.2.	Funkcja getchar()	376
8.5.3.	Funkcja sprintf()	376
8.5.4.	Funkcja sscanf()	377
8.6.	Funkcje biblioteki przeznaczonej do operacji na ciągach tekstowych	378
8.6.1.	Funkcje strcp() i strncpy()	378
8.6.2.	Funkcje strcat() i strncat()	379
8.7.	Funkcje porównania zdefiniowane w bibliotece przeznaczonej do obsługi ciągów tekstowych	380
8.8.	Funkcje wyszukiwania zdefiniowane w bibliotece przeznaczonej do obsługi ciągów tekstowych	382
8.8.1.	Funkcja strchr()	383
8.8.2.	Funkcja strcspn()	383
8.8.3.	Funkcja strpbrk()	384
8.8.4.	Funkcja strrchr()	384
8.8.5.	Funkcja strspn()	385
8.8.6.	Funkcja strstr()	386
8.8.7.	Funkcja strtok()	386
8.9.	Funkcje dotyczące pamięci zdefiniowane w bibliotece przeznaczonej do obsługi ciągów tekstowych	387
8.9.1.	Funkcja memcpy()	388
8.9.2.	Funkcja memmove()	389
8.9.3.	Funkcja memcmp()	390
8.9.4.	Funkcja memchr()	390
8.9.5.	Funkcja memset()	391
8.10.	Pozostałe funkcje w bibliotece przeznaczonej do obsługi ciągów tekstowych	391
8.10.1.	Funkcja strerror()	392
8.10.2.	Funkcja strlen()	392
8.11.	Bezpieczne programowanie w języku C	393

Rozdział 9. Formatowanie danych wejściowych i wyjściowych w języku C 407

9.1.	Wprowadzenie	408
9.2.	Strumienie	408
9.3.	Formatowanie danych wyjściowych za pomocą funkcji printf()	408
9.4.	Wyświetlanie liczb całkowitych	409
9.5.	Liczby zmiennoprzecinkowe	410
9.5.1.	Specyfikatory konwersji e, E i F	410
9.5.2.	Specyfikatory konwersji g i G	411
9.5.3.	Prezentacja specyfikatorów konwersji liczb zmiennoprzecinkowych	411

9.6.	Wyświetlanie ciągów tekstowych i znaków	412
9.7.	Inne specyfikatory konwersji	413
9.8.	Określanie szerokości pola i dokładności podczas wyświetlania danych	414
9.8.1.	Określanie szerokości pola dla wyświetlanej liczby całkowitej	414
9.8.2.	Określanie dokładności dla liczb całkowitych, zmiennoprzecinkowych i ciągów tekstowych ...	415
9.8.3.	Określanie szerokości pola i dokładności wartości	416
9.9.	Stosowanie opcji w ciągu tekstowym formatowania funkcji printf()	416
9.9.1.	Wyrównanie do prawej i lewej strony	416
9.9.2.	Wyświetlanie liczb dodatnich i ujemnych z opcją + i bez niej	417
9.9.3.	Stosowanie opcji w postaci spacji	418
9.9.4.	Stosowanie opcji #	418
9.9.5.	Stosowanie opcji 0	419
9.10.	Wyświetlanie literałów i sekwencje sterujące	419
9.11.	Pobieranie za pomocą funkcji scanf() sformatowanych danych wejściowych	420
9.11.1.	Składnia funkcji scanf()	420
9.11.2.	Specyfikatory konwersji funkcji scanf()	420
9.11.3.	Pobieranie liczb całkowitych za pomocą funkcji scanf()	420
9.11.4.	Pobieranie liczb zmiennoprzecinkowych za pomocą funkcji scanf()	422
9.11.5.	Pobieranie znaków i ciągów tekstowych za pomocą funkcji scanf()	423
9.11.6.	Stosowanie zbioru w funkcji scanf()	423
9.11.7.	Stosowanie szerokości pola podczas pracy z funkcją scanf()	424
9.11.8.	Pomijanie znaków w strumieniu danych wejściowych	425
9.12.	Bezpieczne programowanie w języku C	426
Rozdział 10. Struktury, unie, operacje na bitach i wyliczenia w języku C		435
10.1.	Wprowadzenie	436
10.2.	Definicja struktury	436
10.2.1.	Struktura odwołująca się do samej siebie	437
10.2.2.	Definiowanie zmiennej typu struktury	437
10.2.3.	Nazwy tagów struktury	438
10.2.4.	Operacje możliwe do przeprowadzania na strukturze	438
10.3.	Inicjalizacja struktury	439
10.4.	Uzyskanie dostępu do elementu struktury za pomocą operatorów . i ->	439
10.5.	Stosowanie struktur wraz z funkcjami	441
10.6.	Definicja typedef	441
10.7.	Przykład — wysoko wydajna symulacja tasowania i rozdawania kart	442
10.8.	Unia	445
10.8.1.	Deklaracja unii	445
10.8.2.	Operacje możliwe do przeprowadzania na unii	445
10.8.3.	Inicjalizacja unii w deklaracji	446
10.8.4.	Przykład użycia unii	446
10.9.	Operatory bitowe	447
10.9.1.	Wyświetlenie za pomocą bitów liczby całkowitej bez znaku	448
10.9.2.	Uogólnienie funkcji displayBits() i zapewnienie jej przenośności	450
10.9.3.	Stosowanie operatorów bitowych i, lub, wyłączającego lub i dopełnienia	450
10.9.4.	Stosowanie operatorów bitowych przesunięcia w lewo i przesunięcia w prawo	453
10.9.5.	Operatory bitowe przypisania	454
10.10.	Pole bitowe	455
10.10.1.	Definiowanie pola bitowego	455
10.10.2.	Zastosowanie pola bitowego do przedstawienia figury i koloru karty	456
10.10.3.	Nienazwane pole bitowe	458

10.11. Stała wyliczenia	459
10.12. Anonimowe struktury i unie	460
10.13. Bezpieczne programowanie w języku C	460
Rozdział 11. Przetwarzanie plików w języku C	473
11.1. Wprowadzenie	474
11.2. Plik i strumień	474
11.3. Tworzenie pliku o dostępie sekwencyjnym	475
11.3.1. Wskaźnik do FILE	476
11.3.2. Stosowanie funkcji fopen() do otwierania pliku	476
11.3.3. Stosowanie funkcji feof() do sprawdzania pod kątem znacznika końca pliku	477
11.3.4. Stosowanie funkcji fprintf() do zapisywania danych w pliku	477
11.3.5. Stosowanie funkcji fclose() do zamykania pliku	477
11.3.6. Tryb otwarcia pliku	479
11.4. Odczytywanie danych z pliku o dostępie sekwencyjnym	480
11.4.1. Zerowanie wskaźnika położenia w pliku	481
11.4.2. Program pobierający informacje o saldach klientów	481
11.5. Plik o dostępie swobodnym	484
11.6. Tworzenie pliku o dostępie swobodnym	485
11.7. Losowy zapis danych w pliku o dostępie swobodnym	487
11.7.1. Stosowanie funkcji fseek() do umieszczania wskaźnika położenia w pliku	488
11.7.2. Sprawdzanie pod kątem błędów	489
11.8. Odczytywanie danych z pliku o dostępie swobodnym	489
11.9. Studium przypadku — program przetwarzający transakcje	490
11.10. Bezpieczne programowanie w języku C	496
Rozdział 12. Struktury danych w języku C	507
12.1. Wprowadzenie	508
12.2. Struktura odwołująca się do samej siebie	508
12.3. Dynamiczna alokacja pamięci	509
12.4. Lista jednokierunkowa	510
12.4.1. Funkcja insert()	516
12.4.2. Funkcja delete()	516
12.4.3. Funkcja printList()	518
12.5. Stos	518
12.5.1. Funkcja push()	522
12.5.2. Funkcja pop()	523
12.5.3. Zastosowanie stosu	523
12.6. Kolejka	524
12.6.1. Funkcja enqueue()	528
12.6.2. Funkcja dequeue()	529
12.7. Drzewo	529
12.7.1. Funkcja insertNode()	533
12.7.2. Funkcje nawigacji po drzewie: inOrder(), preOrder() i postOrder()	533
12.7.3. Eliminowanie duplikatów	534
12.7.4. Binarne drzewo wyszukiwania	534
12.7.5. Inne operacje przeprowadzane na drzewie binarnym	534
12.8. Bezpieczne programowanie w języku C	534

Rozdział 13. Preprocesor w języku C	547
13.1. Wprowadzenie	548
13.2. Dyrektywa preprocesora #include	548
13.3. Dyrektywa preprocesora #define — stałe symboliczne	548
13.4. Dyrektywa preprocesora #define — makra	549
13.4.1. Makro z jednym argumentem	550
13.4.2. Makro z dwoma argumentami	551
13.4.3. Znak kontynuowania makra	551
13.4.4. Dyrektywa preprocesora #undef	551
13.4.5. Makra i funkcje biblioteki standardowej	551
13.4.6. Nie umieszczaj w makrze wyrażenia powodującego efekty uboczne	551
13.5. Kompilacja warunkowa	552
13.5.1. Dyrektywa preprocesora #if...#endif	552
13.5.2. Komentowanie bloków kodu za pomocą #if...#endif	552
13.5.3. Warunkowa kompilacja kodu używanego podczas debugowania	552
13.6. Dyrektywy preprocesora #error i #pragma	553
13.7. Operatory # i ##	553
13.8. Numery wierszy	554
13.9. Predefiniowane stałe symboliczne	554
13.10. Asercje	554
13.11. Bezpieczne programowanie w języku C	555
 Rozdział 14. Inne zagadnienia związane z językiem C	 561
14.1. Wprowadzenie	562
14.2. Przekierowanie operacji wejścia-wyjścia	562
14.2.1. Przekierowanie wejścia za pomocą znaku <	562
14.2.2. Przekierowanie wejścia za pomocą znaku 	562
14.2.3. Przekierowanie wyjścia	563
14.3. Zmiennej długości lista argumentów	563
14.4. Stosowanie argumentów powłoki	565
14.5. Kompilowanie programu składającego się z wielu plików kodu źródłowego	566
14.5.1. Deklaracje extern dla zmiennych globalnych w innych plikach	566
14.5.2. Prototypy funkcji	567
14.5.3. Ograniczenie zasięgu za pomocą słowa kluczowego static	568
14.5.4. Plik Makefile	568
14.6. Zakończenie działania programu za pomocą exit() i atexit()	568
14.7. Sufiksy dla literałów liczb całkowitych i zmiennoprzecinkowych	569
14.8. Obsługa sygnałów	570
14.9. Dynamiczna alokacja pamięci — funkcje calloc() i realloc()	572
14.10. Bezwarunkowe odgałęzienie za pomocą goto	573
 Rozdział 15. C++ jako lepsza wersja C — wprowadzenie do technologii obiektowej	 579
15.1. Wprowadzenie	580
15.2. C++	580
15.3. Prosty program dodający dwie liczby	580
15.3.1. Program w C++ dodający dwie liczby	580
15.3.2. Nagłówek <iostream>	581
15.3.3. Funkcja main()	581
15.3.4. Deklaracje zmiennych	581
15.3.5. Obiekty standardowych strumieni wejścia i wyjścia	581

15.3.6.	Manipulator strumienia <code>std::endl</code>	582
15.3.7.	Wyjaśnienie prefiksu <code>std::</code>	582
15.3.8.	Połączone strumienie wyjścia	582
15.3.9.	Polecenie <code>return</code> nie jest wymagane w funkcji <code>main()</code>	582
15.3.10.	Przeciążanie operatorów	583
15.4.	Biblioteka standardowa języka C++	583
15.5.	Pliki nagłówkowe	584
15.6.	Funkcja typu <code>inline</code>	585
15.7.	Słowa kluczowe języka C++	587
15.8.	Referencja i parametry przekazywane przez referencję	587
15.8.1.	Parametry referencyjne	588
15.8.2.	Przekazywanie argumentów przez wartość i przez referencję	589
15.8.3.	Referencja jako alias w funkcji	591
15.8.4.	Zwrot referencji przez funkcję	592
15.8.5.	Komunikat błędu do niezainicjalizowanej referencji	593
15.9.	Pusta lista parametrów	593
15.10.	Argumenty domyślne	593
15.11.	Jednoargumentowy operator ustalenia zasięgu	595
15.12.	Przeciążanie funkcji	596
15.13.	Szablony funkcji	599
15.13.1.	Definiowanie szablonu funkcji	599
15.13.2.	Stosowanie szablonu funkcji	600
15.14.	Wprowadzenie do technologii obiektowej i UML	602
15.14.1.	Podstawowe koncepcje technologii obiektowej	602
15.14.2.	Klasy, elementy składowe danych i funkcji	603
15.14.3.	Projekt i analiza zorientowane obiektowo	604
15.14.4.	Zunifikowany język modelowania	604
15.15.	Wprowadzenie do szablonu klasy <code>vector</code> biblioteki standardowej języka C++	605
15.15.1.	Problemy związane z opartą na wskaźnikach tablicą w stylu języka C	605
15.15.2.	Stosowanie szablonu klasy <code>vector</code>	605
15.15.3.	Obsługa wyjątków — przetwarzanie indeksu spoza zakresu	610
15.16.	Zakończenie rozdziału	611
Rozdział 16. Wprowadzenie do klas, obiektów i ciągów tekstowych		619
16.1.	Wprowadzenie	620
16.2.	Definiowanie klasy z funkcją składową	620
16.3.	Definiowanie funkcji składowej z parametrem	623
16.4.	Dane składowe, funkcje składowe <code>set</code> i <code>get</code>	625
16.5.	Inicjalizacja obiektu za pomocą konstruktora	631
16.6.	Umieszczenie klasy w oddzielnym pliku, aby umożliwić jej wielokrotne użycie	634
16.7.	Oddzielenie interfejsu od implementacji	638
16.8.	Weryfikacja danych wejściowych za pomocą funkcji <code>set</code>	643
16.9.	Zakończenie rozdziału	647
Rozdział 17. Klasy — zgłaszanie wyjątków		655
17.1.	Wprowadzenie	656
17.2.	Studium przypadku — klasa <code>Time</code>	656
17.3.	Zasięg klasy i dostęp do jej elementów składowych	663
17.4.	Funkcje dostępu i funkcje narzędziowe	664
17.5.	Studium przypadku — konstruktor z argumentami domyślnymi	664

17.6. Destruktory	670
17.7. Kiedy są wywoływane konstruktor i destruktor?	671
17.8. Studium przypadku — pułapka podczas zwrotu odwołania lub wskaźnika do prywatnych danych składowych	674
17.9. Domyślne przypisywanie danych składowych	677
17.10. Obiekty i funkcje składowe typu const	678
17.11. Kompozycja — obiekt jako element składowy klasy	681
17.12. Funkcje i klasy zaprzyjaźnione	686
17.13. Stosowanie wskaźnika this	688
17.14. Statyczne elementy składowe klasy	694
17.15. Zakończenie rozdziału	698

Rozdział 18. Przeciążanie operatorów i klasa string711

18.1. Wprowadzenie	712
18.2. Stosowanie przeciążonych operatorów klasy string biblioteki standardowej	712
18.3. Podstawy przeciążania operatorów	715
18.4. Przeciążanie operatorów dwuargumentowych	717
18.5. Przeciążanie dwuargumentowych operatorów wstawiania danych do strumienia i pobierania danych ze strumienia	717
18.6. Przeciążanie operatorów jednoargumentowych	721
18.7. Przeciążanie jednoargumentowych operatorów prefiks i postfiks ++ i --	722
18.8. Studium przypadku — klasa Date	723
18.9. Dynamiczne zarządzanie pamięcią	728
18.10. Studium przypadku — klasa Array	730
18.10.1. Stosowanie klasy Array	731
18.10.2. Definicja klasy Array	735
18.11. Operator w postaci funkcji składowej kontra operator w postaci funkcji nieskładowej	742
18.12. Konwersja między typami	743
18.13. Konstruktor typu explicit i operatory konwersji	744
18.14. Przeciążanie funkcji operator()	747
18.15. Zakończenie rozdziału	747

Rozdział 19. Programowanie zorientowane obiektowo — dziedziczenie759

19.1. Wprowadzenie	760
19.2. Klasa bazowa i klasa pochodna	760
19.3. Relacje między klasą bazową i klasą pochodną	763
19.3.1. Utworzenie i użycie klasy CommissionEmployee	763
19.3.2. Utworzenie klasy BasePlusCommissionEmployee bez wykorzystania dziedziczenia	767
19.3.3. Tworzenie hierarchii dziedziczenia CommissionEmployee- BasePlusCommissionEmployee	772
19.3.4. Hierarchia dziedziczenia CommissionEmployee-BasePlusCommissionEmployee wykorzystująca dane chronione	776
19.3.5. Hierarchia dziedziczenia CommissionEmployee-BasePlusCommissionEmployee wykorzystująca dane prywatne	779
19.4. Konstruktor i destruktor w klasie pochodnej	784
19.5. Dziedziczenie publiczne, chronione i prywatne	785
19.6. Stosowanie dziedziczenia w tworzeniu oprogramowania	786
19.7. Zakończenie rozdziału	787

Rozdział 20. Programowanie zorientowane obiektowo — polimorfizm 793

20.1.	Wprowadzenie	794
20.2.	Wprowadzenie do polimorfizmu — polimorficzna gra wideo	794
20.3.	Relacje między obiektami w hierarchii dziedziczenia	795
20.3.1.	Wywołanie funkcji klasy bazowej z poziomu obiektu klasy pochodnej	796
20.3.2.	Skierowanie wskaźnika funkcji pochodnej na obiekt klasy bazowej	798
20.3.3.	Wywołanie funkcji składowej klasy pochodnej za pomocą wskaźnika klasy bazowej	799
20.3.4.	Funkcje wirtualne i destruktory wirtualne	801
20.4.	Typ pola i konstrukcja switch	808
20.5.	Klasa abstrakcyjna i czysta funkcja wirtualna	808
20.6.	Studium przypadku — system kadrowo-płacowy oparty na polimorfizmie	810
20.6.1.	Tworzenie abstrakcyjnej klasy bazowej Employee	811
20.6.2.	Tworzenie konkretnej klasy pochodnej SalariedEmployee	814
20.6.3.	Tworzenie konkretnej klasy pochodnej CommissionEmployee	817
20.6.4.	Tworzenie pośredniej konkretnej klasy pochodnej BasePlusCommissionEmployee	818
20.6.5.	Przykład przetwarzania polimorficznego	820
20.7.	(Opcjonalnie) Polimorfizm, funkcje wirtualne i wiązanie dynamiczne „pod maską”	823
20.8.	Studium przypadku — system kadrowo-płacowy oparty na polimorfizmie, mechanizmie RTTI, rzutowaniu w dół, operatorach dynamic_cast i typeid oraz klasie type_info	827
20.9.	Zakończenie rozdziału	830

Rozdział 21. Dokładniejsza analiza strumieni wejścia i wyjścia 837

21.1.	Wprowadzenie	838
21.2.	Strumienie	838
21.2.1.	Strumienie klasyczne kontra strumienie standardowe	839
21.2.2.	Nagłówki biblioteki iostream	839
21.2.3.	Obiekty i klasy strumieni wejścia-wyjścia	840
21.3.	Strumień wyjścia	842
21.3.1.	Dane wyjściowe zmiennych typu char *	842
21.3.2.	Wyświetlanie znaków za pomocą funkcji składowej put()	843
21.4.	Strumień wejścia	843
21.4.1.	Funkcje składowe get() i getline()	844
21.4.2.	Funkcje składowe peek(), putback() i ignore() obiektu istream	846
21.4.3.	Operacje wejścia-wyjścia zapewniające bezpieczeństwo typu	846
21.5.	Niesformatowane operacje wejścia-wyjścia przeprowadzane za pomocą funkcji read(), write() i gcount()	847
21.6.	Wprowadzenie do manipulatorów strumienia	848
21.6.1.	Podstawa liczb całkowitych w strumieniu — manipulatory dec, oct, hex i setbase	848
21.6.2.	Dokładność liczb zmiennoprzecinkowych	849
21.6.3.	Szerokość pola — width() i setw()	850
21.6.4.	Manipulatory strumienia wyjścia zdefiniowane przez użytkownika	851
21.7.	Stany formatu strumienia i manipulatorów strumienia	852
21.7.1.	Zera na końcu wartości i przecinek (manipulator showpoint)	853
21.7.2.	Wyrównanie wartości (manipulatory left, right i internal)	854
21.7.3.	Dopełnienie (funkcja fill() i manipulator setfill)	855
21.7.4.	Wewnętrzna podstawa strumienia (manipulatory dec, oct, hex, showbase)	857
21.7.5.	Liczby zmiennoprzecinkowe — notacja naukowa oraz z określoną liczbą cyfr po przecinku (manipulatory scientific i fixed)	857
21.7.6.	Kontrolowanie wielkości liter (manipulator uppercase)	858

21.7.7. Stosowanie formatu boolowskiego (manipulator boolalpha)	859
21.7.8. Ustawianie i zerowanie stanu formatu za pomocą funkcji składowej flags()	860
21.8. Stany błędu strumienia	861
21.9. Powiązanie strumieni wyjścia i wejścia	863
21.10. Zakończenie rozdziału	863
Rozdział 22. Dokładniejsza analiza obsługi wyjątków	875
22.1. Wprowadzenie	876
22.2. Przykład — obsługa próby dzielenia przez zero	876
22.3. Ponowne zgłoszenie wyjątku	882
22.4. Rozwinięcie stosu	883
22.5. Kiedy używać obsługi wyjątków?	885
22.6. Konstruktor, destruktor i obsługa wyjątków	886
22.7. Wyjątki i dziedziczenie	887
22.8. Przetwarzanie niepowodzenia w wyniku działania operatora new	887
22.9. Klasa unique_ptr i dynamiczna alokacja pamięci	890
22.10. Hierarchia wyjątków biblioteki standardowej	893
22.11. Zakończenie rozdziału	894
Rozdział 23. Wprowadzenie do szablonów niestandardowych	901
23.1. Wprowadzenie	902
23.2. Szablony klas	902
23.3. Szablon funkcji przeznaczonej do przeprowadzania operacji na obiekcie specjalizacji szablonu klasy	906
23.4. Parametry pozbawione typu	908
23.5. Argumenty domyślne parametrów typu szablonu	908
23.6. Przeciążanie szablonu funkcji	909
23.7. Zakończenie rozdziału	909
Dodatek A. Pierwszeństwo operatorów w językach C i C++	913
Dodatek B. Kodowanie znaków ASCII	917
Dodatek C. Systemy liczbowe	919
C.1. Wprowadzenie	920
C.2. Skracanie liczb dwójkowych do ósemkowych i szesnastkowych	922
C.3. Konwersja liczb ósemkowych i szesnastkowych na dwójkowe	923
C.4. Konwersja liczb dwójkowych, ósemkowych i szesnastkowych na dziesiętne	924
C.5. Konwersja liczb dziesiętnych na dwójkowe, ósemkowe i szesnastkowe	925
C.6. Ujemne liczby dwójkowe — notacja dopełnienia do dwóch	926
Dodatek D. Sortowanie — informacje szczegółowe	931
D.1. Wprowadzenie	932
D.2. Notacja dużego O	932
D.3. Sortowanie przez wybieranie	933
D.4. Sortowanie przez wstawianie	936
D.5. Sortowanie przez scalanie	939

Dodatek E. Wielowątkowość oraz inne zagadnienia związane ze standardami C99 i C11	949
E.1. Wprowadzenie	950
E.2. Nowe nagłówki w C99	951
E.3. Wyznaczone metody inicjalizacyjne i złożone literały	951
E.4. Typ bool	954
E.5. Niejawne użycie typu int w deklaracji funkcji	955
E.6. Liczby zespolone	956
E.7. Usprawnienia w preprocesorze	957
E.8. Inne funkcje standardu C99	958
E.8.1. Minimalne ograniczenia zasobów kompilatora	959
E.8.2. Słowo kluczowe restrict	959
E.8.3. Niezawodne kopiowanie liczb całkowitych	959
E.8.4. Element składowy w postaci elastycznej tablicy	960
E.8.5. Złagodzone ograniczenia dotyczące inicjalizacji agregowanych elementów	960
E.8.6. Operacje matematyczne dla typów generycznych	960
E.8.7. Funkcja typu inline	961
E.8.8. Zakończenie działania funkcji bez wyrażenia	961
E.8.9. Predefiniowany identyfikator <code>__func__</code>	961
E.8.10. Makro <code>va_copy</code>	961
E.9. Nowe funkcje w standardzie C11	962
E.9.1. Nowe nagłówki C11	962
E.9.2. Obsługa wielowątkowości	962
E.9.3. Funkcja <code>quick_exit()</code>	969
E.9.4. Obsługa Unicode	969
E.9.5. Specyfikator funkcji <code>_Noreturn</code>	970
E.9.6. Wyrażenia typów generycznych	970
E.9.7. Dodatek L — możliwości w zakresie analizy i niezdefiniowane zachowanie	970
E.9.8. Kontrola wyrównania do granicy	971
E.9.9. Asercje statyczne	971
E.9.10. Typy zmiennoprzecinkowe	971
E.10. Zasoby dostępne w internecie	971
Dodatki w internecie	975
Dodatek F. Użycie debugera Visual Studio	FTP
Dodatek G. Użycie debugera GNU	FTP

Programowanie strukturalne w języku C



Zagadnienia

W tym rozdziale poruszymy następujące zagadnienia:

- podstawowe techniki rozwiązywania problemów
- proces tworzenia algorytmu od początku do końca i jego stopniowego dopracowywania
- konstrukcja `if` i `if-else` do wyboru akcji
- pętla `while` do wielokrotnego wykonywania poleceń w programie
- iteracja oparta na liczniku i iteracja oparta na wartowniku
- programowanie strukturalne
- inkrementacja i dekrementacja oraz operatory przypisania

3.1. Wprowadzenie

Zanim przystąpisz do tworzenia programu rozwiązującego konkretny problem, musisz dokładnie poznać ten problem i starannie zaplanować sposób jego rozwiązania. W rozdziale tym i następnym omówimy techniki umożliwiające tworzenie strukturalnych programów komputerowych. W podrozdziale 4.12 przedstawimy podsumowanie technik programowania strukturalnego zaprezentowanych w tym i następnym rozdziale.

3.2. Algorytm

Rozwiązanie dowolnego problemu komputerowego oznacza wykonanie ciągu akcji w określonej kolejności. **Procedura** rozwiązywania problemu w kategoriach:

1. **akcji** przeznaczonych do wykonania
2. i **kolejności**, w jakiej te akcje mają zostać wykonane,

nosi nazwę **algorytmu**. Na przykładzie pokażemy teraz, jak dużą wagę ma poprawne określenie kolejności, w której powinny być wykonane akcje.

Przeanalizujmy algorytm „pobudki” umożliwiający pewnemu menedżerowi przygotowanie się do wyjścia do pracy: (1) wstanie z łóżka, (2) zdjęcie piżamy, (3) wzięcie prysznic, (4) ubranie się, (5) zjedzenie śniadania, (6) udanie się do biura. Ta procedura pozwala menedżerowi na doskonałe przygotowanie się do pracy i podejmowania właściwych decyzji. Przyjmijmy teraz, że te *same* kroki zostaną wykonane w nieco innej kolejności: (1) wstanie z łóżka, (2) zdjęcie piżamy, (3) ubranie się, (4) wzięcie prysznic, (5) zjedzenie śniadania, (6) udanie się do biura. W takim przypadku menedżer pojawi się w biurze mokry. Określenie kolejności wykonywania poleceń w programie komputerowym nosi nazwę **kontroli przepływu działania programu**. W rozdziałach tym i następnym poznasz możliwości języka C w zakresie kontroli programu.

3.3. Pseudokod

Pseudokod to sztuczny i nieformalny język, który pomaga w opracowaniu algorytmu. Przedstawiony tutaj pseudokod jest szczególnie użyteczny podczas tworzenia algorytmu, który później zostanie skonwertowany na strukturalny program w C. Pseudokod jest podobny do *języka, którym posługujesz się na co dzień*: jest wygodny i przyjazny, ale *nie* jest rzeczywistym językiem programowania.

Program zapisany w pseudokodzie *nie* jest wykonywany w komputerze. Pomaga natomiast w przemyśleniu programu, zanim zaczniesz go tworzyć w języku programowania takim jak C.

Pseudokod składa się jedynie ze znaków, więc możesz go wygodnie napisać za pomocą dowolnego edytora tekstu. Starannie przygotowany pseudokod można łatwo skonwertować na kod w języku C, który następnie zostanie uruchomiony. W wielu przypadkach wystarcza po prostu zastąpienie poleceń pseudokodu odpowiadającymi im poleceniami w języku C.

Pseudokod składa się tylko z poleceń *akcji* i *decyzji* — są one wykonywane po przeprowadzeniu konwersji z postaci pseudokodu na polecenia w języku C, które następnie są wykonywane. Definicje *nie* są poleceniami wykonywalnymi — to jedynie komunikaty dla kompilatora. Na przykład następująca definicja:

```
int i;
```

wskazuje kompilatorowi typ zmiennej *i* oraz nakazuje mu zarezerwowanie pamięci dla tej zmiennej. Jednak definicja *nie* powoduje przeprowadzenia w programie żadnej akcji — pobrania danych wejściowych, wygenerowania danych wyjściowych, obliczenia wartości, porównania elementów itd. Część programistów decyduje się na umieszczenie listy wszystkich zmiennych na początku pseudokodu i krótkie omówienie ich przeznaczenia.

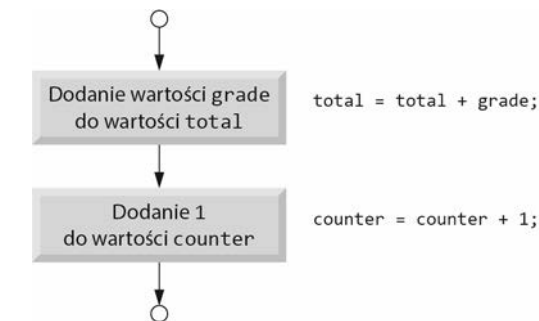
3.4. Struktury kontrolne

Standardowo polecenia w programie są wykonywane jedno po drugim w kolejności, w której zostały zapisane. To nosi nazwę **wykonywania sekwencyjnego**. Wkrótce poznasz różne polecenia języka C pozwalające określić, że kolejne wykonywane polecenie ma być *inne* niż następne w sekwencji. To nosi nazwę **przekazania kontroli**.

W latach 60. ubiegłego wieku stało się jasne, że bezkrytyczne stosowanie mechanizmu przekazywania kontroli było przyczyną największych trudności napotykanych przez doświadczonych twórców oprogramowania. Jako winowajcę wskazywano przede wszystkim **polecenie goto**, które pozwala przekazać kontrolę do jednego z wielu możliwych miejsc w programie. Notacja tzw. programowania strukturalnego stała się praktycznie synonimem **eliminacji goto**.

Programiści Bohm i Jacopini¹ pokazali, że istnieje możliwość tworzenia programów *bez* poleceń goto. W owym czasie wyzwaniem dla programistów była zmiana stylu tworzenia kodu na pozbawiony poleceń goto. Jednak dopiero w latach 70. programowanie strukturalne zaczęło być traktowane poważnie. Wyniki były imponujące, a zespołom pracującym nad oprogramowaniem udało się skrócić czas powstawania aplikacji, zwiększyć częstotliwość dostarczania systemów oraz realizować projekty programistyczne w ramach ustalonych budżetów. Programy powstałe z wykorzystaniem technik strukturalnych stały się bardziej przejrzyste, łatwiejsze do debugowania i modyfikowania, a także znacznie rzadziej popełniano w nich błędy².

Praca Bohma i Jacopiniego pokazała, że każdy program mógłby zostać utworzony z wykorzystaniem jedynie trzech **struktur kontrolnych: sekwencji, wyboru i iteracji**. Struktura sekwencji jest prosta — o ile nie zostanie wskazane inaczej, komputer wykonuje polecenia C jedno po drugim, w kolejności, w której zostały zapisane. **Schemat blokowy** na rysunku 3.1 pokazuje strukturę sekwencji stosowaną w C.



RYСУNEK 3.1. Schemat blokowy struktury sekwencji w języku C

Schemat blokowy

Schemat blokowy to *graficzne* przedstawienie algorytmu lub jego fragmentu. Przygotowuje się go z użyciem specjalnych symboli połączonych strzałkami określanymi mianem **linii schematu blokowego**.

Podobnie jak w przypadku pseudokodu, także schemat blokowy jest użyteczny podczas opracowywania i prezentowania algorytmu, choć to pseudokod jest preferowany przez większość programistów. Schemat blokowy wyraźnie pokazuje sposób działania struktury kontrolnej i dlatego zdecydowaliśmy się na jego użycie w tym miejscu.

Spójrz raz jeszcze na schemat blokowy struktury sekwencji (rysunek 3.1). **Symbol prostokąta**, nazywany **symbolem akcji**, wskazuje na typ akcji odpowiedzialnej za dołączenie wyniku obliczenia lub za operację wejścia-wyjścia. Linie schematu blokowego na tym rysunku pokazują *kierunek* wykonywania tych akcji — najpierw wartość `grade` jest dodawana do wartości `total`, a dopiero później następuje dodanie 1 do wartości `counter`. Język C pozwala zdefiniować dowolną liczbę akcji

¹ C. Bohm i G. Jacopini, *Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules*, Communications of the ACM, Vol. 9, No. 5, May 1966, str. 336 – 371.

² Jak się dowiesz z podrozdziału 14.14, istnieją sytuacje, w których polecenie `goto` okazuje się użyteczne.

w strukturze sekwencji. Jak się wkrótce przekonasz, w *każdym miejscu sekwencji umożliwiającym umieszczenie pojedynczej akcji można zamiast niej umieścić wiele akcji*.

Podczas rysowania schematu blokowego przedstawiającego pełny algorytm pierwszym stosowanym symbolem jest **zaokrąglony prostokąt** ze słowem *Początek*, ostatnim zaś zaokrąglony prostokąt ze słowem *Koniec*. Gdy rysujesz tylko *fragment* algorytmu (zobacz rysunku 3.1), wówczas pomijasz zaokrąglone prostokąty na rzecz małych symboli okręgu nazywanych **konektorami**.

Bodaj najważniejszym symbolem w schemacie blokowym jest **romb**, będący **symbolem decyzji**, który wskazuje *decyzję* do podjęcia. Więcej informacji na ten temat znajdziesz w następnej sekcji.

Polecenia wyboru w języku C

Język C oferuje trzy rodzaje struktur wyboru w postaci pewnych konstrukcji programistycznych. Konstrukcja *if* (zobacz podrozdział 3.5) umożliwia *wybór* (wykonanie) akcji, gdy warunek jest *spełniony*, lub *pominięcie* akcji w przypadku *niespełnienia* warunku. Konstrukcja *if-else* (zobacz podrozdział 3.6) powoduje wykonanie akcji, jeśli warunek jest *spełniony*, i wykonanie innej akcji w przypadku *niespełnienia* warunku. Natomiast konstrukcja *switch* (omówiona w rozdziale 4.) wykonuje jedną z *wielu* różnych akcji, w zależności od wartości wyrażenia. Konstrukcja *if* jest nazywana również **poleceniem pojedynczego wyboru**, ponieważ wybiera lub ignoruje jedną akcję. Konstrukcja *if-else* jest nazywana także **poleceniem podwójnego wyboru**, gdyż wybiera między dwiema różnymi akcjami. Z kolei konstrukcja *switch* bywa nazywana **poleceniem wielokrotnego wyboru**, wybiera bowiem jedną akcję z wielu dostępnych.

Polecenia iteracji w języku C

Język C oferuje trzy rodzaje *struktur iteracji* w postaci poleceń *while* (zobacz podrozdział 3.7), *do-while* i *for* (oba będą omówione w rozdziale 4.).

I na tym koniec. C ma tylko siedem poleceń kontrolnych: sekwencja oraz po trzy rodzaje poleceń wyboru i iteracji. Każdy program w języku C jest tworzony w użyciu połączenia dowolnej liczby tych poleceń kontrolnych, które są niezbędne dla algorytmu implementowanego w programie. Podobnie jak w przypadku pokazanej na rysunku 3.1 struktury sekwencji, także w kolejnych schematach blokowych innych struktur kontrolnych zobaczysz dwa małe symbole okręgów przedstawiających *punkt wejścia* i *punkt wyjścia*. Te **polecenia kontrolne pojedynczego wejścia i pojedynczego wyjścia** pozwalają na łatwe tworzenie przejrzystych programów. Istnieje możliwość łączenia ze sobą segmentów schematów blokowych poleceń kontrolnych przez połączenie punktu wejścia jednej struktury z punktem wejścia innej. To przypomina zabawę z dzieciństwa, gdy dziecko buduje stopy z klocków, i dlatego takie podejście nosi nazwę **łączenia w stos struktur kontrolnych**. Poznasz jeszcze jeden sposób łączenia poleceń kontrolnych — to **zagnieżdżanie**. Dlatego każdy program w języku C będzie skonstruowany tylko z *siedmiu* różnych typów poleceń kontrolnych łączonych ze sobą na tylko dwa sposoby. Można to nazwać kwintesencją prostoty.

3.5. Polecenie wyboru *if*

Polecenie wyboru jest stosowane do wybrania jednego z alternatywnych przebiegów akcji. Załóżmy na przykład, że aby zdać egzamin, trzeba zdobyć minimum 60 punktów. Polecenie w postaci następującego pseudokodu:

```
jeżeli wynik ucznia jest większy lub równy 60
    wyświetl "Zdałeś"
```

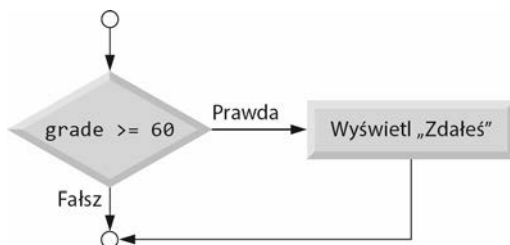
wskazuje, czy warunek „wynik ucznia jest większy lub równy wartości 60” został spełniony, czy nie. Jeżeli warunek jest prawdziwy, wtedy następuje wyświetlenie komunikatu „Zdałeś” i wykonanie następnego w kolejności polecenia pseudokodu (pamiętaj, że pseudokod nie jest rzeczywistym językiem programowania). Natomiast jeśli warunek jest fałszywy, polecenie wyświetlające komunikat zostanie zignorowane i będzie wykonane następne w kolejności polecenie pseudokodu.

Omówione tutaj polecenie *jeżeli* (ang. *if*) pseudokodu można w następujący sposób zapisać w C:

```
if ( grade >= 60 ) {  
    puts( "Zdałeś" );  
} // Koniec konstrukcji if
```

Zwróć uwagę na to, że kod w C dość ściśle odpowiada pseudokodowi (oczywiście konieczne będzie zadeklarowanie zmiennej `grade` typu `int`). Jest to jedna z cech pseudokodu, dzięki której jest on aż tak użyteczny jako narzędzie stosowane podczas tworzenia oprogramowania. Wiersz 2. w tej konstrukcji `if` został wcięty, co jest opcjonalne, choć jednocześnie gorąco zalecane, ponieważ wcięcia pomagają w przedstawieniu struktury programu. Kompilator C ignoruje **białe znaki** — takie jak spacje, tabulatory i znaki nowego wiersza — stosowane we wcięciach i do rozmieszczania kodu w pionie.

Schemat blokowy pokazany na rysunku 3.2 przedstawia polecenie `if` pojedynczego wyboru i zawiera chyba jeden z najważniejszych symboli używanych w schematach blokowych — romb, czyli symbol decyzji, który wskazuje na podjęcie decyzji. Symbol decyzji zawiera wyrażenie, np. warunek, które może być prawdziwe lub fałszywe. Z symbolu decyzji wychodzą *dwie* linie schematu blokowego: jedna wskazuje kierunek wybierany, gdy wyrażenie w symbolu jest prawdziwe, a druga — gdy to wyrażenie jest nieprawdziwe. Decyzja może być oparta na warunku zawierającym operator relacji lub równości. Tak naprawdę może zostać podjęta na podstawie *dowolnego* wyrażenia — jeżeli przyjmie ono wartość *zero*, będzie traktowane jako fałszywe, wartość *niezerowa* zaś powoduje potraktowanie wyrażenia jako prawdziwego.



RYСУNEK 3.2. Schemat blokowy polecenia `if` pojedynczego wyboru

Konstrukcja `if` również jest poleceniem kontrolnym *pojedynczego wejścia i pojedynczego wyjścia*. Wkrótce dowiesz się, że schematy blokowe pozostałych struktur kontrolnych także mogą zawierać (poza małymi okręgami i liniami) tylko prostokąty wskazujące akcje do wykonania oraz romby wskazujące na podejmowanie decyzji. Jest to *oparty na akcjach i decyzjach model programowania*, na który będziemy kłaść nacisk.

Można wyobrazić sobie siedem pojemników, z których każdy będzie zawierał jedynie schematy blokowe polecenia kontrolnego jednego z siedmiu typów. Segmenty tych schematów blokowych są puste — nic nie zostało umieszczone w prostokątach i rombach. Następnie Twoim zadaniem jest przygotowanie programu za pomocą dowolnej wymaganej przez algorytm liczby poleceń kontrolnych każdego typu, łącząc je *tylko* na dwa możliwe sposoby (*stos* lub *zagnieżdżenie*), oraz wypełnienie *akcji i decyzji* w sposób odpowiedni dla algorytmu. Przedstawimy teraz różne sposoby zapisywania akcji i decyzji.

3.6. Polecenie wyboru `if-else`

Polecenie wyboru `if` wykonuje pewną akcję tylko wtedy, gdy warunek jest prawdziwy. W przeciwnym razie akcja zostanie pominięta. Z kolei polecenie wyboru `if-else` pozwala zdefiniować *różne* akcje do wykonania na podstawie prawdziwości i nieprawdziwości warunku. Spójrz na następujący fragment pseudokodu:

```
jeżeli wynik ucznia jest większy lub równy 60
    wyświetl "Zdałeś"
w przeciwnym razie
    wyświetl "Nie zdałeś"
```

Jeżeli wynik uzyskany przez ucznia jest większy lub równy 60, zostanie wyświetlony komunikat *Zdałeś*. W przeciwnym razie będzie wyświetlony komunikat *Nie zdałeś*. W obu przypadkach po wyświetleniu komunikatu zostanie wykonane następne polecenie znajdujące się w sekwencji. Blok polecenia *w przeciwnym razie* również został wcięty.



Dobra praktyka programistyczna 3.1

Stosuj wcięcia dla bloków poleceń w konstrukcji *if-else* zarówno w pseudokodzie, jak i w języku C.



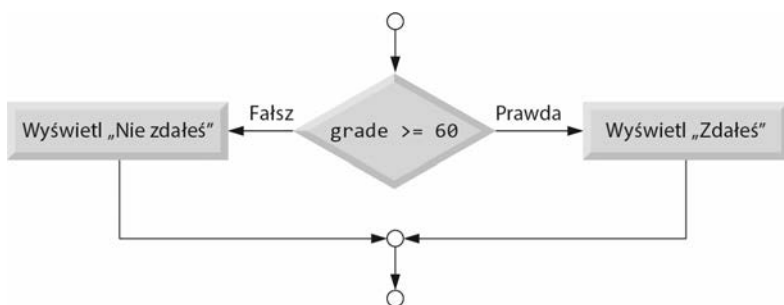
Dobra praktyka programistyczna 3.2

Jeżeli istnieje wiele poziomów wcięć, każdy z nich powinien mieć tę samą ilość dodatkowych spacji.

Omówiony pseudokod można w następujący sposób zapisać w języku C:

```
if ( grade >= 60 ) {
    puts( "Zdałeś" );
} // Koniec konstrukcji if
else {
    puts( "Nie zdałeś" );
} // Koniec bloku else.
```

Schemat blokowy z rysunku 3.3 pokazuje przepływ kontroli działania programu podczas wykonywania konstrukcji *if-else*. Także w tym przypadku poza małymi okręgami i strzałkami jedyne symbole w schemacie blokowym to prostokąty oznaczające akcje i romb oznaczający decyzję.



RYСУNEK 3.3. Schemat blokowy polecenia *if-else* podwójnego wyboru

Język C oferuje **operator warunkowy** (**?:**), bardzo ściśle związany z konstrukcją *if-else*. Ten operator to jedyny w języku C operator *trójargumentowy* — pobiera *trzy* operandy. W połączeniu z operatorem warunkowym tworzą one **wyrażenie warunkowe**. Pierwszym operandem jest *warunek*. Drugim operandem jest wartość całego wyrażenia warunkowego, gdy warunek *został* spełniony. Trzeci operand to wartość całego wyrażenia warunkowego, gdy warunek *nie został* spełniony. Na przykład przedstawione tutaj wywołanie funkcji `puts()`:

```
puts( grade >= 60 ? "Zdałeś" : "Nie zdałeś" );
```

zawiera argument w postaci wyrażenia warunkowego, które w zależności od spełnienia lub niespełnienia warunku `grade >= 60` generuje odpowiedni ciąg tekstowy (Zdałeś lub Nie zdałeś). To wywołanie `puts()` działa praktycznie tak samo jak omówiona wcześniej konstrukcja `if-else`.

Operandy drugi i trzeci w wyrażeniu warunkowym również mogą być akcjami przeznaczonymi do wykonania. Na przykład wyrażenie warunkowe:

```
grade >= 60 ? puts( "Zdałeś" ) : puts( "Nie zdałeś" );
```

należy zinterpretować następująco: „jeżeli wartość `grade` jest większa lub równa 60, to należy wywołać funkcję `puts("Zdałeś")`, natomiast w przeciwnym razie ma być wywołana funkcja `puts("Nie zdałeś")`”. To wyrażenie również jest porównywalne z przedstawioną wcześniej konstrukcją `if-else`. Operator warunkowy może być używany tam, gdzie nie można zastosować konstrukcji `if-else`, np. w wyrażeniach i z argumentami funkcji takiej jak `printf()`.



Wskazówka pomagająca uniknąć błędów 3.1

Dla drugiego i trzeciego operandu operatora warunkowego używaj wyrażen tego samego typu, aby uniknąć trudnych do wykrycia błędów.

Zagnieżdżone konstrukcje `if-else`

Zagnieżdżone konstrukcje `if-else` pozwalają sprawdzić wiele przypadków dzięki umieszczeniu konstrukcji `if-else` *wewnątrz* innych. Na przykład przedstawiony tutaj pseudokod spowoduje wyświetlenie oceny 5 w przypadku otrzymania na egzaminie wyniku większego lub równego 90, oceny 4 dla wyniku większego lub równego 80 (ale mniejszego niż 90), oceny 3 dla wyniku większego lub równego 70 (ale mniejszego niż 80), oceny 2 dla wyniku większego lub równego 60 (ale mniejszego niż 70) i oceny 1 dla wszystkich pozostałych wyników:

```
jeżeli wynik ucznia jest większy lub równy 90
    wyświetl "5"
w przeciwnym razie
    jeżeli wynik ucznia jest większy lub równy 80
        wyświetl "4"
    w przeciwnym razie
        jeżeli wynik ucznia jest większy lub równy 70
            wyświetl "3"
        w przeciwnym razie
            jeżeli wynik ucznia jest większy lub równy 60
                wyświetl "2"
            w przeciwnym razie
                wyświetl "1"
```

Ten pseudokod można w następującej postaci zapisać w języku C:

```
if ( grade >= 90 ) {
    puts( "5" );
} // Koniec konstrukcji if
else {
    if ( grade >= 80 ) {
        puts( "4" );
    } // Koniec konstrukcji if
    else {
        if ( grade >= 70 ) {
            puts( "3" );
        } // Koniec konstrukcji if
        else {
            if ( grade >= 60 ) {
                puts( "2" );
            }
        }
    }
}
```

```

    } // Koniec konstrukcji if
    else {
        puts( "1" );
    } // Koniec bloku else
    } // Koniec bloku else
    } // Koniec bloku else
    } // Koniec bloku else

```

Jeżeli wartość zmiennej `grade` jest większa lub równa 90, wówczas wszystkie cztery warunki zostają spełnione, ale zostanie wykonane wywołanie `puts()` po pierwszym warunku. Po wykonaniu tej funkcji blok `else „zewnętrznej”` konstrukcji `if-else` zostanie pominięty.

Tę konstrukcję `if` można zapisać w jeszcze innej postaci:

```

if ( grade >= 90 ) {
    puts( "5" );
} // Koniec konstrukcji if
else if ( grade >= 80 ) {
    puts( "4" );
} // Koniec bloku else. if
else if ( grade >= 70 ) {
    puts( "3" );
} // Koniec bloku else. if
else if ( grade >= 60 ) {
    puts( "2" );
} // Koniec bloku else. if
else {
    puts( "1" );
} // Koniec bloku else

```

Dla kompilatora C obie postaci są odpowiednikami. Druga z przedstawionych jest popularna, ponieważ pozwala uniknąć zbyt dużego poziomu wcięcia kodu w prawą stronę. Takie wcięcia powodują często zmniejszenie ilości miejsca w wierszu i tym samym zmuszają do podziału wierszy, co z kolei negatywnie wpływa na czytelność programu.

Konstrukcja `if` oczekuje tylko jednego polecenia w jej definicji — w takim przypadku można pominąć nawias klamrowy. Jeżeli w definicji konstrukcji `if` chcesz umieścić więcej niż jedno polecenie, muszą one znajdować się w nawiasie klamrowym. Zestaw poleceń umieszczonych w nawiasie klamrowym jest nazywany **poleceniem złożonym** lub **blokiem**.



Obserwacja dotycząca tworzenia oprogramowania 3.1

Polecenie złożone może być umieszczone w programie wszędzie tam, gdzie dozwolone jest stosowanie pojedynczego polecenia.

W kolejnym fragmencie kodu przedstawiamy polecenie złożone w bloku `else` konstrukcji `if-else`.

```

if ( grade >= 60 ) {
    puts( "Zdałeś." );
} // Koniec konstrukcji if
else {
    puts( "Nie zdałeś." );
    puts( "Musisz ponownie uczęszczać na zajęcia." );
} // Koniec bloku else

```

W omawianym przykładzie jeśli wartość zmiennej `grade` jest mniejsza niż lub równa 60, program wykona *oba* polecenia `puts()` zdefiniowane w bloku `else`, a skutkiem będzie wyświetlenie następujących komunikatów:

```

Nie zdałeś.
Musisz ponownie uczęszczać na zajęcia.

```

Nawias klamrowy w bloku `else` jest bardzo ważny. Gdyby go nie było, polecenie:

```
puts( "Musisz ponownie uczęszczać na zajęcia." );
```

znajdowałoby się *na zewnątrz* bloku `else` konstrukcji `if` i byłoby wykonywane niezależnie od spełnienia warunku. Dlatego nawet uczniowi, który zdał egzamin, zostałby wyświetlony komunikat o konieczności ponownego uczęszczania na zajęcia.



Wskazówka pomagająca uniknąć błędu 3.2

Definicje poleceń kontrolnych zawsze umieszczaj w nawiasie klamrowym, nawet jeśli blok składa się z tylko jednego polecenia. To rozwiązuje problem *zapomnianego bloku `else`* (przedstawiony w ćwiczeniach 3.30 i 3.31).

Błąd składni jest wychwytywany przez kompilator. Z kolei *błąd logiczny* pojawia się w trakcie wykonywania programu. *Błąd logiczny o znaczeniu krytycznym* powoduje awarię programu i jego przedwczesne zakończenie. *Błąd logiczny, który nie ma znaczenia krytycznego*, pozwala kontynuować działanie programu, choć wygenerowane przez niego dane mogą być nieprawidłowe.

Podobnie jak polecenie złożone można umieścić wszędzie tam, gdzie dozwolone jest użycie pojedynczego polecenia, tak samo może w ogóle nie być polecenia. Mamy wówczas tzw. puste polecenie przedstawione za pomocą średnika w miejscu, w którym powinno znaleźć się polecenie.



Często popełniany błąd w trakcie programowania 3.1

Umieszczenie średnika po warunku w konstrukcji `if`, np. `if (grade >= 60);`, prowadzi do powstania błędu logicznego w poleceniu `if` pojedynczego wyboru i błędu składni w poleceniu `if` podwójnego wyboru.



Wskazówka pomagająca uniknąć błędu 3.3

Wpisanie początkowego i końcowego nawiasu klamrowego dla polecenia złożonego jeszcze przed rozpoczęciem dodawania tworzących je poleceń pomaga uniknąć pominięcia niezbędnego nawiasu klamrowego oraz chroni przed błędami składni i logicznymi (początkowy i końcowy nawias klamrowy są konieczne).

3.7. Polecenie iteracji `while`

Polecenie iteracji (nazywane również **poleceniem powtarzającym** lub **pętlą**) pozwala określić akcję, która ma być powtarzana, gdy warunek został spełniony. Ten fragment pseudokodu:

```
dopóki lista zakupów zawiera elementy
    kup następny element i skreśl go z listy
```

opisuje iterację przeprowadzaną podczas zakupów. Warunek „lista zakupów zawiera elementy” może być prawdziwy lub fałszywy. Jeżeli jest prawdziwy, wówczas zostanie wykonana akcja „kup następny element i skreśl go z listy”. Ta akcja będzie wykonywana *ciągle*, dopóki warunek pozostaje spełniony. Polecenie lub polecenia umieszczone w pętli *dopóki* (ang. *while*) tworzą jej definicję.

W pewnym momencie warunek nie zostanie spełniony (w omawianym przykładzie po kupieniu ostatniej rzeczy i skreśleniu jej z listy). Wówczas nastąpi zakończenie iteracji i wykonanie pierwszego polecenia pseudokodu znajdującego się *po* tej strukturze iteracji.



Często popełniany błąd w trakcie programowania 3.2

Jeżeli w definicji pętli *dopóki* brakuje akcji powodującej, że warunek w pewnym momencie nie zostanie spełniony, struktura iteracji nigdy nie zostanie przerwana — jest to błąd nazywany pętlą działającą w nieskończoność.



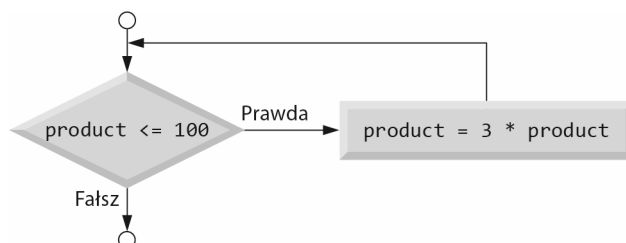
Często popełniany błąd w trakcie programowania 3.3

Zapisanie słowa kluczowego, np. `while` lub `if`, z użyciem pierwszej wielkiej litery, np. `While` lub `If`, powoduje wygenerowanie błędu podczas kompilacji. Pamiętaj, że w języku C wielkość liter ma znaczenie, a słowa kluczowe należy zapisywać małymi literami.

Jako przykład polecenia *dopóki* zobacz segment programu odpowiedzialny za odszukanie pierwszej potęgi liczby 3 większej niż 100. Zmienna `product` typu `int` została zainicjalizowana wartością 3. Po zakończeniu wykonywania przedstawionego tutaj fragmentu kodu zmienna `product` będzie zawierała szukaną liczbę.

```
product = 3;
while ( product <= 100 ) {
    product = 3 * product;
}
```

Schemat blokowy na rysunku 3.4 pokazuje przepływ kontroli działania programu w przedstawionym wcześniej poleceniu iteracji `while`. Zwróć uwagę, że także na tym schemacie poza małymi okręgami i strzałkami znajdują się tylko prostokąt i romb. Ten schemat blokowy wyraźnie pokazuje iterację. Strzałka wychodząca z prostokąta jest skierowana do rombu oznaczającego decyzję, której warunek jest sprawdzany w trakcie każdej iteracji aż do chwili, gdy nie zostanie spełniony. Wówczas nastąpi zakończenie wykonywania polecenia `while` i przejście do następnego polecenia w programie.



RYСУNEK 3.4. Schemat blokowy polecenia iteracji `while`

Po rozpoczęciu wykonywania polecenia `while` wartość zmiennej `product` wynosi 3. Jest ona ciągle mnożona przez 3 i przyjmuje kolejno wartości 9, 27 i 81. Gdy wartością zmiennej `product` będzie 243, warunek w pętli `while` (`product <= 100`) nie zostanie spełniony. W efekcie nastąpi zakończenie iteracji, a ostateczną wartością zmiennej `product` będzie 243. Działanie programu będzie kontynuowane od następnego polecenia znajdującego się po pętli `while`.

3.8. Studium przypadku tworzenia algorytmu 1 — iteracja kontrolowana przez licznik

Aby zilustrować sposoby tworzenia algorytmów, zajmiesz się rozwiązaniem wielu wariantów problemu polegającego na obliczeniu średniego wyniku w klasie. Spójrz na opis problemu:

W klasie liczącej dziesięcioro uczniów został przeprowadzony sprawdzian. Do dyspozycji masz wyniki (liczby całkowite z przedziału od 0 do 100) uzyskane przez uczniów. Twoje zadanie polega na obliczeniu średniego w klasie wyniku ze sprawdzianu.

Średni wynik w klasie jest równy sumie poszczególnych wyników podzielonej przez liczbę uczniów. Algorytm przeznaczony do rozwiązania tego problemu w komputerze musi pobrać wszystkie wyniki, obliczyć średnią i wyświetlić tę wartość.

Pracę zaczniesz od przygotowania pseudokodu wymieniającego wszystkie akcje konieczne do wykonania i wskazującego ich kolejność. **Iteracja oparta na liczniku** zostanie zastosowana do pojedynczego pobrania wyników. Ta technika używa zmiennej nazywanej **licznikiem** do określenia, ile razy ma zostać wykonany zestaw poleceń. W omawianym przykładzie zakończenie iteracji nastąpi, gdy wartość licznika przekroczy 10. Teraz przedstawimy algorytm w postaci pseudokodu (zobacz listing 3.1) i odpowiadający mu kod programu w C (zobacz listing 3.2). Natomiast w następnym przykładzie zobaczysz, jak wygląda *opracowywanie* algorytmu pseudokodu. Iteracja oparta na liczniku jest często nazywana **iteracją skończoną**, ponieważ liczba iteracji jest znana jeszcze *przed* rozpoczęciem wykonywania pętli.

LISTING 3.1. Algorytm pseudokodu używający opartej na liczniku iteracji do rozwiązania problemu polegającego na obliczeniu średniego w klasie wyniku ze sprawdzianu

```
1  zmiennej total przypisz wartość zero
2  zmiennej licznika grade przypisz wartość jeden
3
4  dopóki zmienna licznika grade ma wartość mniejszą lub równą dziesięć
5      pobierz następny wynik
6      dodaj ten wynik do zmiennej total
7      dodaj jeden do zmiennej licznika grade
8
9  zmiennej average przypisz wartość całkowitą podzieloną przez dziesięć
10 wyświetl średni wynik w klasie
```

LISTING 3.2. Wykorzystanie opartej na liczniku iteracji do obliczenia średniego w klasie wyniku ze sprawdzianu

```
1 // Plik: fig03_06.c
2 // Program obliczający średni w klasie wynik ze sprawdzianu za pomocą iteracji opartej na liczniku
3 #include <stdio.h>
4
5 // Wykonywanie programu rozpoczyna się od funkcji main()
6 int main( void )
7 {
8     unsigned int counter; // Numer wyniku, który zostanie teraz wpisany
9     int grade; // Wynik ze sprawdzianu
10    int total; // Suma wyników wpisanych przez użytkownika
11    int average; // Średni wynik ze sprawdzianu
12
13    // Faza inicjalizacji
14    total = 0; // Inicjalizacja wartości całkowitej
15    counter = 1; // Inicjalizacja licznika pętli
16
17    // Faza przetwarzania
18    while ( counter <= 10 ) { // Iteracja ma się odbyć 10 razy
19        printf( "%s", "Podaj wynik: " ); // Prośba o podanie danych wejściowych
20        scanf( "%d", &grade ); // Odczytanie wyniku wpisanego przez użytkownika
21        total = total + grade; // Dodanie wyniku do wartości całkowitej
22        counter = counter + 1; // Inkrementacja licznika
23    } // Koniec pętli while
```

```

24
25 // Faza zakończenia
26 average = total / 10; // Dzielenie liczb całkowitych
27
28 printf( "Średni wynik w klasie wynosi %d\n", average ); // Wyświetlenie wyniku
29 } // Koniec funkcji main()

```

Dane wyjściowe:

```

Podaj wynik: 98
Podaj wynik: 76
Podaj wynik: 71
Podaj wynik: 87
Podaj wynik: 83
Podaj wynik: 90
Podaj wynik: 57
Podaj wynik: 79
Podaj wynik: 82
Podaj wynik: 94
Średni wynik w klasie wynosi 81

```

W algorytmie wspomnieliśmy o wartości całkowitej i liczniku. **Wartość całkowita** to zmienna stosowana do akumulacji sumy ciągu wartości. Z kolei licznik to zmienna (wiersz 8.) użyta do *zliczania* — w omawianym przykładzie zliczana jest liczba podanych wyników. Ponieważ w tym programie zmienna licznika została użyta do liczenia od 1 do 10 (tylko wartości dodatnie), więc została zadeklarowana jako typu `unsigned int`. Zmienna tego typu może przechowywać jedynie wartości nieujemne, czyli od zera w górę. Zmienna przeznaczona do przechowywania wartości całkowitej powinna zostać zainicjalizowana jako 0 jeszcze *przed* jej użyciem w programie. W przeciwnym razie suma będzie uwzględniała także poprzednią wartość przechowywaną w pamięci zarezerwowanej dla zmiennej wartości całkowitej. Zmienna licznika jest zwykle inicjalizowana z wartością 0 lub 1, w zależności od sposobu użycia (w książce przedstawimy przykłady obu wariantów). Niezainicjalizowana zmienna zawiera **wartość „śmieciową”** — ostatnią wartość przechowywaną w pamięci zarezerwowanej dla danej zmiennej.



Często popełniany błąd w trakcie programowania 3.4

Jeżeli zmienna licznika lub wartości całkowitej nie zostanie zainicjalizowana, wynik wygenerowany przez program prawdopodobnie będzie nieprawidłowy. Jest to typowy przykład błędu logicznego.



Wskazówka pomagająca uniknąć błędu 3.4

Inicjalizuj wszystkie zmienne liczników i wartości całkowitych.

W omawianym programie obliczona średnia wyników ze sprawdzianu wynosi 81. Suma wszystkich wyników to 817, a po podzieleniu tej liczby przez 10 otrzymujemy wynik 81,7 — liczbę zawierającą *przecinek dziesiętny*. W następnym podrozdziale zobaczysz, jak wygląda praca z takimi liczbami (nazywa się je *liczbami zmiennoprzecinkowymi*).

Ważna informacja o miejscu umieszczania definicji zmiennej

W rozdziale 2. wspomnieliśmy, że standard C pozwala umieszczać definicję zmiennej w *dowolnym* miejscu w funkcji `main()`, ale jeszcze przed użyciem zmiennej w kodzie. W tym rozdziale będziemy kontynuować grupowanie definicji zmiennych na początku funkcji `main()`, aby tym samym podkreślić fazy inicjalizacji, przetwarzania i zakończenia prostych programów. Jednak począwszy od

rozdziału 4., definicja każdej zmiennej będzie umieszczona tuż przed miejscem jej pierwszego użycia w kodzie. W rozdziale 5. — w omówieniu *zasięgu* zmiennej — zobaczysz, jak ta praktyka pomaga w eliminowaniu błędów.

3.9. Studium przypadku tworzenia algorytmu 2 — iteracja kontrolowana przez wartownik

Teraz uogólnimy problem polegający na obliczeniu średniego wyniku ze sprawdzianu. Przyjrzyj się następującemu problemowi:

Opracuj obliczający średni w klasie wynik ze sprawdzianu program, który przetworzy dowolną liczbę wyników po każdym jego uruchomieniu.

W poprzednim przykładzie programu obliczającego średni wynik ze sprawdzianu liczba tych wyników (10) była wcześniej znana. Natomiast w tym przypadku nie mamy żadnej wskazówki dotyczącej liczby wyników, które zostaną podane. Program musi mieć możliwość przetwarzania *dowolnej* liczby wyników. Jak program może ustalić, kiedy należy zakończyć pobieranie danych wejściowych (tutaj w postaci wyników)? Skąd program będzie wiedział, kiedy obliczyć i wyświetlić średni wynik ze sprawdzianu?

Jednym z rozwiązań problemu jest użycie wartości specjalnej o nazwie **wartość wartownika** (inne jej nazwy to **wartość sygnału**, **wartość fikcyjna** i **wartość flagi**) wskazującej na zakończenie podawania danych. W omawianym programie użytkownik będzie wpisywał wyniki dopóty, dopóki nie poda wszystkich *niezbędnych* wyników. Następnie użytkownik wprowadzi wartość wartownika oznaczającą „ostatni wynik został już podany”. Iteracja kontrolowana przez wartość wartownika jest często określana mianem **iteracji w nieskończoność**, ponieważ liczba iteracji jest *nieznana* przed rozpoczęciem wykonywania pętli.

Nie ulega wątpliwości, że wartość wartownika musi zostać wybrana w taki sposób, aby *nie* było możliwe jej pomylenie z akceptowaną wartością danych wejściowych. Skoro wynik jest *nieujemną* liczbą całkowitą, w omawianym przypadku -1 będzie odpowiednią wartością wartownika. Po uruchomieniu programu będzie on przetwarzał strumień danych wejściowych, np. w postaci 95, 96, 75, 74, 89 i -1 . Program obliczy i wyświetli średni wynik dla podanych wartości 95, 96, 75, 74 i 89, a -1 uzna za wartość wartownika, która *nie* powinna być uwzględniona do średniej.

Od początku do końca, stopniowe udoskonalanie

W programie przeznaczonym do obliczania średniego wyniku wykorzystasz technikę *od początku do końca, stopniowe udoskonalanie*, która ma duże znaczenie podczas tworzenia programów o doskonałej strukturze. Pracę rozpoczyna się od przygotowania pseudokodu przedstawiającego **początek**:

określenie średniego w klasie wyniku ze sprawdzianu

Początek to pojedyncze polecenie wyrażające ogólny sposób funkcjonowania programu, a zatem praktycznie w *pełni* przedstawia przeznaczenie programu. Niestety rzadko zawiera wystarczającą ilość szczegółów związanych z tworzeniem programu w C. Dlatego trzeba przystąpić do procesu *udoskonalania*. Początek należy więc podzielić na szereg małych zadań wymienionych w kolejności, w której powinny być wykonane. Efektem jest **pierwszy etap udoskonalania**:

*inicjalizacja zmiennych
dane wejściowe, suma i liczba wyników ze sprawdzianu
obliczenie i wyświetlenie średniego wyniku w klasie*

W tym miejscu została użyta tylko *struktura sekwencji* — wymienione kroki są wykonywane w podanej kolejności, jeden po drugim.



Obserwacja dotycząca tworzenia oprogramowania 3.2

Każdy etap udoskonalania, podobnie jak sam początek, to pełna specyfikacja algorytmu. Inny jest tylko poziom szczegółów.

Drugie udoskonalenie

Przechodzimy teraz na następny poziom udoskonalania, np. **drugie udoskonalenie**, i zajmiemy się określeniem zmiennych. Konieczne jest obliczenie wartości całkowitej liczb, zliczenie liczby wartości do przetworzenia, przygotowanie zmiennej dla każdego podawanego wyniku (dane wejściowe) i zmiennej dla wartości przechowującej obliczoną średnią. Dlatego to polecenie pseudokodu:

```
inicjalizacja zmiennych
```

można zmodyfikować na następujące:

```
inicjalizacja zmiennej total z wartością zero  
inicjalizacja zmiennej counter z wartością zero
```

Konieczne jest zainicjalizowanie tylko zmiennych przechowujących wartość całkowitą (total) i licznika (counter). Z kolei zmienne dla średniej (average) i wyniku (grade) — odpowiednio dla obliczonej średniej i danych wejściowych użytkownika — nie muszą być zainicjalizowane, ponieważ wartość pierwszej zostanie obliczona, a drugiej podana przez użytkownika. Polecenie pseudokodu:

```
dane wejściowe, suma i liczba wyników ze sprawdzianu
```

wymaga użycia *struktury iteracji* pobierającej poszczególne wyniki. Ponieważ wcześniej nie jest znana liczba wyników do przetworzenia, wykorzystana będzie iteracja kontrolowana przez wartownik. Użytkownik będzie podawał niezbędne wyniki pojedynczo. Po wpisaniu ostatniego konieczne będzie podanie wartości wartownika. Po pobraniu każdego wyniku program sprawdzi, czy została podana wartość wartownika, i jeśli tak, zakończy działanie pętli. Udoskonalona wersja wymienionego wcześniej polecenia pseudokodu przedstawia się następująco:

```
pobierz pierwszy wynik (to może być wartość wartownika)  
dopóki użytkownik nie wprowadzi wartości wartownika  
    dodaj ten wynik do wartości całkowitej  
    dodaj jeden do wartości licznika  
pobierz następny wynik (to może być wartość wartownika)
```

Zwróć uwagę na *brak* w pseudokodzie nawiasów klamrowych obejmujących zestaw poleceń tworzących definicję pętli *dopóki* (while). Wszystkie polecenia w tej pętli zostały po prostu wcięte, co wyraźnie pokazuje, że do niej należą. Warto w tym miejscu przypomnieć, że pseudokod to nieformalna pomoc dla programisty.

Polecenie pseudokodu:

```
obliczenie i wyświetlenie średniego w klasie wyniku ze sprawdzianu
```

można udoskonalic na postać:

```
jeżeli licznik ma wartość inną niż zero  
    średnia to wartość całkowita podzielona przez wartość licznika  
    wyświetl średnią  
w przeciwnym razie  
    wyświetl "Nie podano żadnego wyniku"
```

Zachowaj ostrożność i sprawdź pod kątem *dzielenia przez zero* — jest to **błąd krytyczny**, który jeśli nie zostanie wykryty i ujawni się dopiero w uruchomionym programie, doprowadzi do jego **awarii**. Pełny pseudokod po drugiej fazie udoskonalania przedstawiamy na listingu 3.3.

LISTING 3.3. Zapisany w pseudokodzie algorytm używający iteracji na podstawie wartownika do rozwiązania problemu obliczenia średniego w klasie wyniku ze sprawdzianu

```
1  inicjalizacja zmiennej total z wartością zero
2  inicjalizacja zmiennej counter z wartością zero
3
4  pobierz pierwszy wynik (to może być wartość wartownika)
5  dopóki użytkownik nie wprowadzi wartości wartownika
6      dodaj ten wynik do wartości całkowitej
7      dodaj jeden do wartości licznika
8      pobierz następny wynik (to może być wartość wartownika)
9
10 jeżeli licznik ma wartość inną niż zero
11     średnia to wartość całkowita podzielona przez wartość licznika
12     wyświetl średnią
13 w przeciwnym razie
14     wyświetl "Nie podano żadnego wyniku"
```



Często popełniany błąd w trakcie programowania 3.5

Próba dzielenia przez zero prowadzi do błędu krytycznego.



Dobra praktyka programistyczna 3.3

Podczas dzielenia przez wyrażenie, którego wartością może być zero, należy wyraźnie przeprowadzić sprawdzenie pod tym kątem i zapewnić odpowiednią obsługę takiej próby w programie (np. wyświetlenie komunikatu błędu) zamiast pozwolić na wygenerowanie błędu krytycznego.

Na listingach 3.2 i 3.3 pseudokod zawiera po kilka pustych wierszy, które poprawiają jego czytelność. Te wiersze ponadto dzielą program na oddzielne fazy.



Obserwacja dotycząca tworzenia oprogramowania 3.3

Wiele programów można logicznie podzielić na trzy fazy: *inicjalizacji* (odpowiedzialną za inicjalizację zmiennych programu), *przetwarzania* (zajmującą się odpowiednią pracą z wartościami danych wejściowych i zmiennych programu) oraz *zakończenia* (w której są przeprowadzane ostateczne obliczenia i wyświetlenie ich wyników).

Przedstawiony na listingu 3.3 algorytm pseudokodu zawiera ogólne rozwiązanie problemu obliczenia średniego w klasie wyniku ze sprawdzianu. Ten algorytm udało się opracować po zaledwie dwóch udoskonaleniach. Jednak czasami potrzeba więcej poziomów usprawniania.



Obserwacja dotycząca tworzenia oprogramowania 3.4

Proces programowania od początku do końca i stopniowego udoskonalania można zakończyć, gdy zapisany w pseudokodzie algorytm jest na tyle szczegółowy, że można go skonwertować na kod w języku C. Implementowanie programu w C jest wówczas dość proste.

Program zaimplementowany w języku C i przykładowe dane wygenerowane po jego uruchomieniu przedstawia listing 3.4. Wprawdzie podany został tylko jeden wynik, obliczona średnia będzie prawdopodobnie liczbą z przecinkiem dziesiętnym. Typ `int` nie może przedstawiać takiej liczby. W programie pojawił się więc typ `float`, który umożliwia obsługę liczb zawierających przecinek dziesiętny (czyli **liczb zmiennoprzecinkowych**), i operator specjalny nazywany operatorem rzutowania, który obsługuje obliczanie średniej. Zanim omówimy funkcjonalność programu, przeanalizujemy jego kod źródłowy.

LISTING 3.4. Wykorzystujący iterację opartą na wartowniku program obliczający średni w klasie wynik ze sprawdzianu

```
1 // Plik: fig03_08.c
2 // Wykorzystujący iterację opartą na wartowniku program obliczający średni w klasie wynik ze sprawdzianu
3 #include <stdio.h>
4
5 // Wykonywanie programu rozpoczyna się od funkcji main()
6 int main( void )
7 {
8     unsigned int counter; // Liczba wprowadzonych wyników
9     int grade; // Wynik ze sprawdzianu
10    int total; // Suma wyników wpisanych przez użytkownika
11
12    float average; // Średnia ma postać liczby zmiennoprzecinkowej
13
14    // Faza inicjalizacji
15    total = 0; // Inicjalizacja wartości całkowitej
16    counter = 0; // Inicjalizacja licznika pętli
17
18    // Faza przetwarzania
19    // Pobranie pierwszego wyniku od użytkownika
20    printf( "%s", "Podaj wynik, -1 kończy program: " ); // Prośba o podanie danych wejściowych
21    scanf( "%d", &grade ); // Odczytanie wyniku wpisanego przez użytkownika
22
23    // Pętla będzie wykonywana, dopóki użytkownik nie poda wartości wartownika
24    while ( grade != -1 ) {
25        total = total + grade; // Dodanie wyniku do wartości całkowitej
26        counter = counter + 1; // Inkrementacja licznika
27
28        // Pobranie następnego wyniku od użytkownika
29        printf( "%s", "Podaj wynik, -1 kończy program: " ); // Prośba o podanie danych wejściowych
30        scanf( "%d", &grade ); // Odczytanie następnego wyniku
31    } // Koniec pętli while
32
33    // Faza zakończenia
34    // Jeżeli użytkownik podał co najmniej jeden wynik
35    if ( counter != 0 ) {
36
37        // Obliczenie średniej wszystkich podanych wyników
38        average = ( float ) total / counter; // Uniknięcie skrócenia wyniku
39
40        // Wyświetlenie wartości średniej z dokładnością do dwóch cyfr po przecinku
41        printf( "Średni wynik w klasie wynosi %.2f \n", average );
42    } // Koniec konstrukcji if
43    else { // Jeżeli nie został podany żaden wynik, należy wyświetlić odpowiedni komunikat
44        puts( "Nie podano żadnego wyniku." );
45    } // Koniec bloku else
46 } // Koniec funkcji main()
```

Dane wejściowe:

Podaj wynik, -1 kończy program: 75
Podaj wynik, -1 kończy program: 94
Podaj wynik, -1 kończy program: 97
Podaj wynik, -1 kończy program: 88
Podaj wynik, -1 kończy program: 70
Podaj wynik, -1 kończy program: 64
Podaj wynik, -1 kończy program: 83
Podaj wynik, -1 kończy program: 89
Podaj wynik, -1 kończy program: -1
Średni wynik w klasie wynosi 82.50

Podaj wynik, -1 kończy program: -1
Nie podano żadnego wyniku.

Zwróć uwagę na polecenie złożone w pętli while (wiersz 24.) na listingu 3.4. Nawias klamrowy jest *niezbędny* i gwarantuje wykonanie wszystkich czterech poleceń w pętli. Bez tego nawiasu trzy ostatnie polecenia w definicji pętli znalazłyby się *poza* pętlą, co z kolei spowodowałoby nieprawidłową interpretację tego kodu.

```
while ( grade != -1 )  
    total = total + grade; // Dodanie wyniku do wartości całkowitej  
    counter = counter + 1; // Inkrementacja licznika  
    printf( "%s", "Podaj wynik, -1 kończy program: " ); // Prośba o podanie danych wejściowych  
    scanf( "%d", &grade ); // Odczytanie następnego wyniku
```

Skutkiem działania tego fragmentu kodu jest *pętla działająca w nieskończoność*, o ile użytkownik nie wpisze -1 jako pierwszego wyniku.



Wskazówka pomagająca uniknąć błędu 3.5

W pętli kontrolowanej przez wartość wartownika należy wyraźnie podać tę wartość użytkownikowi w komunikacie zawierającym prośbę o podanie danych wejściowych.

Jawna i niejawna konwersja między typami

Średnia nie zawsze będzie wartością w postaci liczby całkowitej. Bardzo często średnią może być wartość typu 7,2 lub -93,5. Taka wartość jest określana mianem liczby zmiennoprzecinkowej i może być przedstawiona za pomocą typu danych float. W wierszu 12. zmienna average została zdefiniowana jako typu float, aby mogła zawierać także część ułamkową wyniku obliczeń. Jednak wynikiem operacji `total / counter` jest liczba całkowita, ponieważ wartości *obu* użytych w niej zmiennych są w postaci liczb całkowitych. Dzielenie dwóch liczb całkowitych powoduje przeprowadzenie tzw. **dzielenia całkowitego**, w trakcie którego część ułamkowa wyniku zostaje **skrócona** (utracona). Ponieważ *najpierw* jest przeprowadzane obliczenie, część ułamkowa zostaje utracona *przed* przypisaniem wartości zmiennej average. Jeżeli chcesz przeprowadzić operację zmiennoprzecinkową z użyciem liczb całkowitych, konieczne jest utworzenie wartości tymczasowych w postaci liczb zmiennoprzecinkowych. Język C oferuje jednoargumentowy **operator rzutowania** pozwalający na wykonanie takiego zadania, jak pokazaliśmy w wierszu 38. omawianego programu:

```
average = ( float ) total / counter;
```

Tutaj operator rzutowania tworzy *tymczasową* kopię zmiennoprzecinkową operandu `total`. Wartość przechowywana w zmiennej `total` wciąż jest liczbą całkowitą. Wykorzystanie operatora rzutowania w taki sposób jest nazywane **jawną konwersją**. Operacja jest teraz przeprowadzana na liczbie zmiennoprzecinkowej (tymczasowa wartość `total` typu float), która jest dzielona przez wartość

w postaci liczby całkowitej typu `unsigned int` przechowywanej w zmiennej `counter`. C przeprowadza obliczenie wyrażenia arytmetycznego tylko wtedy, gdy typy danych operandów są *identyczne*. Aby zagwarantować, że operandy są *tego samego* typu, kompilator wykonuje operację nazywaną **niejawną konwersją** wybranych operandów. Na przykład jeśli wyrażenie zawiera typy danych `unsigned int` i `float`, następuje utworzenie kopii operandu typu `unsigned int` i jego konwersja na typ `float`. W omawianym przykładzie po utworzeniu kopii zmiennej `counter` i jej konwersji na typ `float` przeprowadzane jest obliczenie, a wynik w postaci liczby zmiennoprzecinkowej zostaje przypisany zmiennej `average`. Język C oferuje zestaw reguł przeznaczonych do konwersji operandów na różne typy. Więcej informacji na ten temat znajdziesz w rozdziale 5.

Operatory rzutowania są dostępne dla *większości* typów danych — do ich utworzenia używa się nawiasów, w które jest ujęta nazwa typu danych. Każdy operator rzutowania jest **operatorem jednoargumentowym**, czyli pobiera tylko jeden operand. W rozdziale 2. była mowa o dwuargumentowych operatorach arytmetycznych. C obsługuje również jednoargumentowe wersje operatorów `+` i `-`, więc można tworzyć wyrażenia w postaci takiej jak `-7` i `+5`. Kolejność wykonywania operatorów rzutowania to od prawej do lewej strony, a ich pierwszeństwo jest takie samo jak innych operatorów jednoargumentowych takich jak `+` i `-`. To pierwszeństwo jest o jeden poziom wyżej niż **operatorów wielokrotności**, np. `*`, `/` i `%`.

Formatowanie liczby zmiennoprzecinkowej

W wierszu 41. na listingu 3.4, podczas wyświetlania średniej został użyty specyfikator konwersji `%.2f`. Litera `f` w tym specyfikatorze oznacza, że wyświetlana jest wartość zmiennoprzecinkowa. Z kolei `.2` określa **dokładność**, z którą wartość jest wyświetlana — tutaj są to dwie cyfry po przecinku. W przypadku użycia specyfikatora konwersji `%f`, czyli bez podanej dokładności, zostanie wykorzystana **dokładność domyślna** wynosząca 6, dokładnie jak po zastosowaniu specyfikatora `%.6f`. Podczas wyświetlania liczb zmiennoprzecinkowych są one **zaokrąglane** do podanej liczby cyfr po przecinku. Wartość przechowywana w pamięci pozostaje natomiast niezmienniona. Po wykonaniu przedstawionych tutaj poleceń wyświetlone będą wartości 3.45 i 3.4:

```
printf( "%.2f\n", 3.446 ); // Dane wyjściowe to 3.45
printf( "%.1f\n", 3.446 ); // Dane wyjściowe to 3.4
```



Często popełniany błąd w trakcie programowania 3.6

Podanie dokładności w specyfikatorze konwersji ciągu tekstowego formatowania w funkcji `scanf()` spowoduje błąd. Ustawienie dokładności jest używane jedynie w specyfikatorze konwersji w funkcji `printf()`.

Uwagi dotyczące liczb zmiennoprzecinkowych

Wprawdzie liczby zmiennoprzecinkowe nie zawsze są w stu procentach dokładne, ale znajdują zastosowanie w wielu aplikacjach. Na przykład gdy mówimy o normalnej temperaturze ciała, 36,6 stopnia Celsjusza, nie trzeba stosować zbyt dużej dokładności. Jeżeli odczytujesz ją z termometru elektronicznego, faktyczna temperatura może wynosić 36,6999473210643. Chodzi tutaj o to, że liczba 36,6 będzie wystarczająca w większości aplikacji. Do tego tematu jeszcze powrócimy w dalszej części książki.

Kolejna kwestia dotycząca liczb zmiennoprzecinkowych wiąże się z dzieleniem. Jeżeli podzielimy 10 przez 3, otrzymasz wynik 3,333333... z nieskończoną sekwencją cyfr 3. W pamięci komputer alokuje *konkretną* ilość miejsca do przechowywania takiej wartości, więc przechowywana wartość zmiennoprzecinkowa jest jedynie *przybliżona*.



Często popełniany błąd w trakcie programowania 3.7

Używanie liczb zmiennoprzecinkowych w sposób zakładający ich dużą dokładność może prowadzić do błędów. W większości komputerów liczby zmiennoprzecinkowe są przedstawiane jedynie w przybliżeniu.



Wskazówka pomagająca uniknąć błędu 3.6

Nie używaj liczb zmiennoprzecinkowych do sprawdzania równości.

3.10. Studium przypadku tworzenia algorytmu 3 — zagnieżdżone polecenia kontrolne

Zdefiniujemy teraz kolejny problem do rozwiązania. Raz jeszcze przygotujesz algorytm za pomocą pseudokodu i techniki „od początku do końca, stopniowe udoskonalanie”, a następnie utworzysz odpowiadający mu program w języku C. Wcześniej dowiedziałeś się, że struktury kontrolne mogą być *układane* na sobie (w sekwencji), podobnie jak dziecko buduje konstrukcje z klocków. W omawianym przykładzie zostanie użyty drugi z możliwych sposobów łączenia poleceń kontrolnych w C — **zagnieżdżanie** jednego *wewnątrz* drugiego. Zapoznaj się z opisem problemu:

Uczelnia oferuje kurs przygotowujący studentów do egzaminu państwowego pozwalającego uzyskać licencję brokera nieruchomości. W poprzednim roku 10 uczestników tego kursu przystąpiło do egzaminu. Oczywiście uczelnia chce wiedzieć, jak im poszło na tym egzaminie. Poproszono Cię o utworzenie programu podsumowującego wyniki. Masz listę 10 studentów. Obok każdego z nich znajduje się cyfra 1 oznaczająca zdanie egzaminu lub 2 oznaczająca niezdanie egzaminu.

Utworzony program powinien analizować wyniki w następujący sposób:

1. Wprowadzenie danych wejściowych wyniku testu, 1 lub 2. Wyświetlenie komunikatu „Podaj wynik egzaminu” za każdym razem, gdy program pobiera kolejny wynik.
2. Zliczenie liczby obu typów wyniku.
3. Wyświetlenie podsumowania wyników egzaminu podające liczbę studentów, którzy zdali egzamin, i liczbę tych, którym się nie powiodło.
4. Jeżeli egzamin zdało więcej niż 8 studentów, powinien zostać jeszcze wyświetlony komunikat „Nagroda dla instruktora!”.

Po dokładnym przeczytaniu opisu problemu można poczynić następujące obserwacje:

1. Program musi przetworzyć 10 wyników egzaminu. Użyta zostanie pętla kontrolowana przez licznik.
2. Każdy wynik to liczba 1 lub 2. W trakcie odczytu wyniku egzaminu program musi ustalić tę liczbę. W omawianym tutaj algorytmie przeprowadzane jest sprawdzenie pod kątem liczby 1. Jeżeli liczbą nie jest 1, przyjmuje się założenie, że liczba to 2. (W ćwiczeniu 3.27 Twoim zadaniem będzie zagwarantowanie, że każdym wynikiem egzaminu jest liczba 1 lub 2).
3. Używane są dwa liczniki — pierwszy do zliczenia studentów, którzy zdali egzamin, i drugi do zliczenia tych, którym się nie udało zdać egzaminu.
4. Gdy program przetworzy wszystkie wyniki, musi określić, czy więcej niż 8 studentów zdało egzamin.

Podczas opracowywania algorytmu wykorzystasz technikę „od początku do końca, stopniowe udoskonalanie”. Pracę rozpoczynasz od pseudokodu przedstawiającego początek:

analiza wyników egzaminu i ustalenie czy instruktor powinien otrzymać nagrodę

Trzeba w tym miejscu ponownie podkreślić, że początek to *pełne* przedstawienie sposobu działania programu, choć niewątpliwie konieczne jest kilka usprawnień, zanim ten pseudokod będzie można zamienić na program w języku C. Po pierwszym udoskonaleniu pseudokod ma następującą postać:

```
inicjalizacja zmiennych
pobranie danych wejściowych egzaminu oraz zliczenie zdanych i niezdanych
wyświetlenie podsumowania wyników i określenie czy instruktor powinien otrzymać nagrodę
```

Tutaj również mamy *pełne* przedstawienie sposobu działania programu i konieczne jest dalsze udoskonalenie pseudokodu. Przechodzimy do konkretnych zmiennych. Liczniki są potrzebne do zliczania zdanych i niezdanych egzaminów. Ponadto licznik kontroluje działanie pętli, a do przechowywania danych wejściowych użytkownika potrzebna jest zmienna. Dlatego następujące polecenie pseudokodu:

```
inicjalizacja zmiennych
```

można zamienić na trzy nowe:

```
inicjalizacja zmiennej passes z wartością zero
inicjalizacja zmiennej failures z wartością zero
inicjalizacja zmiennej student z wartością zero
```

Zwróć uwagę na zainicjalizowanie jedynie licznika i zmiennej dla wartości całkowitej. Polecenie pseudokodu:

```
pobranie danych wejściowych egzaminu oraz zliczenie zdanych i niezdanych
```

wymaga pętli pozwalającej na pobranie kolejno wyniku każdego egzaminu. Liczba danych wejściowych jest z góry znana — to dokładnie dziesięć wyników egzaminu — więc odpowiednie jest użycie pętli kontrolowanej przez licznik. Wewnątrz pętli (np. **zagnieżdżenie** w pętli) podwójnego wyboru polecenie ustali wynik egzaminu i przeprowadzi inkrementację odpowiedniego licznika. Po udoskonaleniu polecenie pseudokodu ma następującą postać:

```
dopóki licznik studentów ma wartość mniejszą lub równą dziesięć
  pobierz wynik następnego egzaminu

  jeżeli student zdał
    dodaj jeden do zmiennej pass
  w przeciwnym razie
    dodaj jeden do zmiennej failures

  dodaj jeden do licznika student
```

Zwróć uwagę na użycie pustych wierszy, co znacznie poprawia czytelność kodu. Polecenie pseudokodu:

```
wyświetlenie podsumowania wyników i określenie czy instruktor powinien otrzymać nagrodę
```

może po udoskonaleniu mieć następującą postać:

```
wyświetl liczbę zdanych egzaminów
wyświetl liczbę niezdanych egzaminów
jeżeli egzamin zdało więcej niż ośmioro studentów
  wyświetl "Nagroda dla instruktora"
```

Pełny pseudokod po udoskonaleniach możesz zobaczyć na listingu 3.5. Puste wiersze w pętli *dopóki* mają na celu zwiększenie czytelności programu.

LISTING 3.5. Pseudokod algorytmu przeznaczonego do analizy wyników egzaminu

```
1  inicjalizacja zmiennej passes z wartością zero
2  inicjalizacja zmiennej failures z wartością zero
3  inicjalizacja zmiennej student z wartością zero
4
5  dopóki licznik studentów ma wartość mniejszą lub równą dziesięć
6      pobierz wynik następnego egzaminu
7
8      jeżeli student zdał
9          dodaj jeden do zmiennej pass
10     w przeciwnym razie
11         dodaj jeden do zmiennej failures
12
13     dodaj jeden do licznika student
14
15 wyświetl liczbę zdanych egzaminów
16 wyświetl liczbę niezdaných egzaminów
17 jeżeli egzamin zdało więcej niż ośmioro studentów
18     wyświetl "Nagroda dla instruktora"
```

Ten pseudokod jest wystarczający, aby można było przeprowadzić jego konwersję na język C. Kod programu w języku C i dwa przykładowe jego uruchomienia przedstawia listingu3.6. Wykorzystana została cecha języka C umożliwiająca inicjalizację zmiennej podczas jej definiowania (wiersze od 9. do 11.). Taka inicjalizacja jest przeprowadzana w trakcie kompilacji. Zwróć także uwagę na to, że podczas wyświetlania na ekranie wartości typu unsigned int został użyty specyfikator konwersji %u (wiersze 33. i 34.).

LISTING 3.6. Program przeprowadzający analizę wyników egzaminu

```
1  // Plik: fig03_10.c
2  // Analiza wyników egzaminu
3  #include <stdio.h>
4
5  // Wykonywanie programu rozpoczyna się od funkcji main()
6  int main( void )
7  {
8      // Inicjalizacja zmiennych w ich definicjach
9      unsigned int passes = 0; // Liczba zdanych egzaminów
10     unsigned int failures = 0; // Liczba niezdaných egzaminów
11     unsigned int student = 1; // Licznik studentów
12     int result; // Jeden wynik egzaminu
13
14     // Przetworzenie danych 10 studentów za pomocą pętli kontrolowanej przez licznik
15     while ( student <= 10 ) {
16
17         // Pobieranie danych wejściowych od użytkownika
18         printf( "%s", "Podaj wynik egzaminu (1=zdany,2=niezdany): " );
19         scanf( "%d", &result );
20
21         // Jeżeli wynikiem jest 1, należy inkrementować wartość zmiennej passes
22         if ( result == 1 ) {
23             passes = passes + 1;
24         } // Koniec konstrukcji if
25         else { // W przeciwnym razie należy inkrementować wartość zmiennej failures
26             failures = failures + 1;
27         } // Koniec bloku else
28
29         student = student + 1; // Inkrementacja licznika studentów
```

```

30 } // Koniec pętli while
31
32 // Faza zakończenia oraz wyświetlenie liczby zdanych i niezdanych egzaminów
33 printf( "Zdane: %u\n", passes );
34 printf( "Niezdane: %u\n", failures );
35
36 // Jeżeli egzamin zdało więcej niż 8 studentów, należy wyświetlić komunikat "Nagroda dla instruktora!"
37 if ( passes > 8 ) {
38     puts( "Nagroda dla instruktora!" );
39 } // Koniec konstrukcji if
40 } // Koniec funkcji main()

```

Dane wyjściowe:

```

Podaj wynik egzaminu (1=zdany,2=niezdany): 1
Podaj wynik egzaminu (1=zdany,2=niezdany): 2
Podaj wynik egzaminu (1=zdany,2=niezdany): 2
Podaj wynik egzaminu (1=zdany,2=niezdany): 1
Podaj wynik egzaminu (1=zdany,2=niezdany): 1
Podaj wynik egzaminu (1=zdany,2=niezdany): 1
Podaj wynik egzaminu (1=zdany,2=niezdany): 2
Podaj wynik egzaminu (1=zdany,2=niezdany): 1
Podaj wynik egzaminu (1=zdany,2=niezdany): 1
Podaj wynik egzaminu (1=zdany,2=niezdany): 2
Zdane: 6
Niezdane: 4

```

```

Podaj wynik egzaminu (1=zdany,2=niezdany): 1
Podaj wynik egzaminu (1=zdany,2=niezdany): 1
Podaj wynik egzaminu (1=zdany,2=niezdany): 1
Podaj wynik egzaminu (1=zdany,2=niezdany): 2
Podaj wynik egzaminu (1=zdany,2=niezdany): 1
Podaj wynik egzaminu (1=zdany,2=niezdany): 1
Podaj wynik egzaminu (1=zdany,2=niezdany): 1
Podaj wynik egzaminu (1=zdany,2=niezdany): 1
Podaj wynik egzaminu (1=zdany,2=niezdany): 1
Podaj wynik egzaminu (1=zdany,2=niezdany): 1
Zdane: 9
Niezdane: 1
Nagroda dla instruktora!

```



Obserwacja dotycząca tworzenia oprogramowania 3.5

Doświadczenie pokazuje, że najtrudniejszym etapem rozwiązywania problemu jest opracowanie algorytmu. Po przygotowaniu prawidłowego algorytmu proces tworzenia działającego programu w języku C okazuje się całkiem prosty.



Obserwacja dotycząca tworzenia oprogramowania 3.6

Wielu programistów tworzy programy, nie używając nigdy żadnych narzędzi wspomagających programowanie, np. pseudokodu. Są przekonani, że tworzenie pseudokodu jedynie opóźnia wygenerowanie ostatecznych danych wyjściowych.

3.11. Operatory przypisania

Język C oferuje wiele operatorów przypisania umożliwiających skrócenie wyrażeń przypisania. Na przykład następujące polecenie:

```
c = c + 3;
```

może zostać skrócone za pomocą **operatora dodawania z przypisaniem** (**+=**):

```
c += 3;
```

Operator += wartość wyrażenia znajdującego się po *prawej* stronie operatora dodaje do wartości zmiennej po *lewej* stronie operatora, a wynik przypisuje zmiennej po *lewej* stronie operatora. Każde polecenie o postaci:

```
zmienna = zmienna operator wyrażenie;
```

w którym *operator* to jeden z operatorów dwuargumentowych +, -, *, / lub % (albo inny spośród omówionych w rozdziale 10.), może zostać zapisane jako:

```
zmienna operator= wyrażenie;
```

Dlatego przypisanie `c += 3` powoduje dodanie 3 do wartości zmiennej `c`. Tabela 3.1 zawiera arytmetyczne operatory przypisania oraz wykorzystujące je przykładowe wyrażenia.

TABELA 3.1. Arytmetyczne operatory przypisania

Operator przypisania	Przykładowe wyrażenie	Wyjaśnienie	Przypisanie
Założenie: <code>int c = 3, d = 5, e = 4, f = 6, g = 12;</code>			
<code>+=</code>	<code>c += 7</code>	<code>c = c + 7</code>	10 do <code>c</code>
<code>-=</code>	<code>d -= 4</code>	<code>d = d - 4</code>	1 do <code>d</code>
<code>*=</code>	<code>e *= 5</code>	<code>e = e * 5</code>	20 do <code>e</code>
<code>/=</code>	<code>f /= 3</code>	<code>f = f / 3</code>	2 do <code>f</code>
<code>%=</code>	<code>g %= 9</code>	<code>g = g % 9</code>	3 do <code>g</code>

3.12. Operatory inkrementacji i dekrementacji

Język C oferuje jednoargumentowy **operator inkrementacji** (**++**) i jednoargumentowy **operator dekrementacji** (**--**), o których więcej dowiesz się z tabeli 3.2. Jeżeli zmienna `c` ma być inkrementowana o 1, wówczas można skorzystać z operatora `++` zamiast wyrażenia `c = c + 1` lub `c += 1`. Jeżeli operatory inkrementacji lub dekrementacji zostały umieszczone *przed* zmienną (*prefiks*), noszą nazwę **operatorów** odpowiednio **preinkrementacji** i **predekrementacji**. Natomiast umieszczenie operatorów inkrementacji lub dekrementacji *po* zmiennej (*postfiks*) powoduje, że są one określane mianem **operatorów** odpowiednio **postinkrementacji** i **postdekrementacji**. Preinkrementacja (predekrementacja) zmiennej powoduje jej inkrementację (dekrementację) o 1, a *następnie* użycie nowej wartości w wyrażeniu, w którym ta zmienna występuje. Postinkrementacja (postdekrementacja) zmiennej powoduje użycie w wyrażeniu wartości bieżącej, a *następnie* inkrementację (dekrementację) wartości tej zmiennej o 1.

TABELA 3.2. Operatory inkrementacji i dekrementacji

Operator	Przykładowe wyrażenie	Wyjaśnienie
++	++a	Zwiększenie wartości a o 1, następnie użycie nowej wartości a w wyrażeniu, w którym ona występuje
++	a++	Użycie bieżącej wartości a w wyrażeniu, w którym występuje, następnie zwiększenie jej o 1
--	--b	Zmniejszanie wartości b o 1, następnie użycie nowej wartości b w wyrażeniu, w którym ona występuje
--	b--	Użycie bieżącej wartości b w wyrażeniu, w którym występuje, następnie zmniejszenie jej o 1

W programie na listingu 3.7 zobaczysz różnice w działaniu między wersjami preinkrementacji i postinkrementacji operatora ++. Postinkrementacja zmiennej c powoduje jej inkrementację już *po* użyciu w poleceniu printf(). Z kolei preinkrementacja zmiennej c powoduje jej inkrementację *jeszcze przed* użyciem w poleceniu printf().

LISTING 3.7. Przykłady preinkrementacji i postinkrementacji

```
1 // Plik: fig03_13.c
2 // Preinkrementacja i postinkrementacja
3 #include <stdio.h>
4
5 // Wykonywanie programu rozpoczyna się od funkcji main()
6 int main( void )
7 {
8     int c; // Zdefiniowanie zmiennych
9
10    // Przykład postinkrementacji
11    c = 5; // Przypisanie 5 wartości zmiennej c
12    printf( "%d\n", c ); // Wyświetlenie wartości 5
13    printf( "%d\n", c++ ); // Wyświetlenie wartości 5, a następnie postinkrementacja
14    printf( "%d\n", c ); // Wyświetlenie wartości 6
15
16    // Przykład preinkrementacji
17    c = 5; // Przypisanie 5 wartości zmiennej c
18    printf( "%d\n", c ); // Wyświetlenie wartości 5
19    printf( "%d\n", ++c ); // Preinkrementacja, a następnie wyświetlenie wartości 6
20    printf( "%d\n", c ); // Wyświetlenie wartości 6
21 } // Koniec funkcji main()
```

Dane wyjściowe:

```
5
5
6

5
6
6
```

Program wyświetla wartość zmiennej c przed użyciem operatora ++ i po jego użyciu. Operator dekrementacji (--) działa podobnie.



Dobra praktyka programistyczna 3.4

Operatory jednoargumentowe należy umieszczać przy operandach, bez spacji między operandem i operatorem.

Trzy operatory przypisania użyte w programie z listingu 3.6:

```
passes = passes + 1;  
failures = failures + 1;  
student = student + 1;
```

można zapisać w nieco zwięźlejszej postaci z zastosowaniem *operatora przypisania*:

```
passes += 1;  
failures += 1;  
student += 1;
```

lub *operatora preinkrementacji*:

```
++passes;  
++failures;  
++student;
```

bądź też *operatora postinkrementacji*:

```
passes++;  
failures++;  
student++;
```

Trzeba w tym miejscu wyraźnie podkreślić, że podczas inkrementacji lub dekrementacji zmiennej o samą siebie w poleceniu formy preinkrementacji i postinkrementacji mają *taki sam* efekt. Tylko wtedy, gdy zmienna pojawia się w kontekście większego wyrażenia, preinkrementacja i postinkrementacja mają *odmienny* efekt (podobnie rzecz się ma z predekrementacją i postdekrementacją). Z przedstawionych dotąd wyrażań tylko proste nazwy zmiennych mogą być używane jako operandy operatora inkrementacji lub dekrementacji.



Często popełniany błąd w trakcie programowania 3.8

Próba użycia operatora inkrementacji lub dekrementacji w wyrażeniu innym niż prosta nazwa zmiennej spowoduje wygenerowanie błędu składni, np. jeśli spróbujesz zapisać `++(x + 1)`.



Wskazówka pomagająca uniknąć błędu 3.7

Ogólnie rzecz biorąc, język C nie określa kolejności, w której będą obliczane operandy operatora (w rozdziale 4. poznasz wyjątki od tej reguły dla kilku operatorów). Dlatego operatorów inkrementacji lub dekrementacji należy używać jedynie w poleceniach, w których zmienna jest inkrementowana lub dekrementowana przez samą siebie.

Tabela 3.3 zawiera pierwszeństwo i kolejność wykonywania działań dotychczas omówionych operatorów. W tabeli pierwszeństwo operatorów jest coraz mniejsze dla każdego kolejnego wiersza zawierającego operatory. Druga kolumna wskazuje kolejność na poszczególnych poziomach pierwszeństwa. Zwróć uwagę, że operator warunkowy (:?), jednoargumentowe operatory inkrementacji (++), dekrementacji (--), plus (+), minus (-) i rzutowania oraz operatory przypisania (=, +=, -=, *=, /= i %=) mają kolejność od prawej do lewej strony. W trzeciej kolumnie znajdziesz nazwy poszczególnych grup operatorów. Wszystkie pozostałe operatory wymienione w tabeli 3.3 mają kolejność wykonywania działań od lewej do prawej.

TABELA 3.3. Pierwszeństwo i kolejność dotychczas omówionych operatorów

Operatory	Kolejność wykonywania działań	Typ
++ (postfiks) -- (postfiks)	Od prawej do lewej	Postfiks
+ - (typ) ++ (prefiks) -- (prefiks)	Od prawej do lewej	Jednoargumentowy
* / %	Od lewej do prawej	Mnożenia i dzielenia
+ -	Od lewej do prawej	Dodawania i odejmowania
< <= > >=	Od lewej do prawej	Relacji
== !=	Od lewej do prawej	Równości
?:	Od prawej do lewej	Warunku
= += -= *= /= %=	Od prawej do lewej	Przypisania

3.13. Bezpieczne programowanie w języku C

Przepełnienie arytmetyczne

W przedstawionym na listingu 2.4 programie obliczającym sumę dwóch wartości typu `int` znalazło się następujące polecenie (wiersz 18.):

```
sum = integer1 + integer2; // Obliczenie sumy
```

Nawet takie proste polecenie potencjalnie sprawia problem — dodawanie liczb całkowitych może doprowadzić do tego, że wartość będzie *zbyt duża*, aby mogła być przechowywana w typie `int`. Ten problem jest znany pod nazwą **przepełnienia arytmetycznego** i może doprowadzić do niezdefiniowanego zachowania, potencjalnie umożliwiając przeprowadzenie ataku na system.

Wartości minimalna i maksymalna, które w konkretnym systemie mogą być przechowywane w zmiennej typu `int`, są przedstawiane przez zmienne odpowiednio `INT_MAX` i `INT_MIN` zdefiniowane w pliku nagłówkowym `limits.h`. Podobne stałe istnieją także dla innych typów liczb całkowitych i poznasz je w rozdziale 4. Wartości stałych dla stosowanej platformy możesz poznać po wyświetleniu wymienionego pliku nagłówkowego w edytorze tekstu.

Za dobrą praktykę programistyczną uznaje się zagwarantowanie jeszcze *przed* przeprowadzeniem obliczeń arytmetycznych takich jak w wierszu 18. w programie przedstawionym na listingu 2.4, że *nie* spowodują one przepełnienia. Przeznaczony do tego celu kod znajdziesz w witrynie internetowej CERT (<https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards>) — w polu wyszukiwania wpisz po prostu `INT32-C`. Wspomniany kod używa operatorów `&&` (logiczne i) oraz `||` (logiczne lub), które poznasz w rozdziale 4. W solidnym kodzie produkcyjnym takie operacje sprawdzenia powinny być przeprowadzane dla *wszystkich* obliczeń, które mogą spowodować przepełnienie lub niedomiar. W dalszych rozdziałach zaprezentujemy jeszcze inne techniki programistyczne przeznaczone do obsługi błędów przepełnienia arytmetycznego.

Liczba całkowita bez znaku

W wierszu 8. programu z listingu 3.2 zadeklarowana jest zmienna `counter` typu `unsigned int`, ponieważ służy do zliczania *jedynie wartości nieujemnych*. Ogólnie rzecz biorąc, licznik przechowujący tylko wartości nieujemne powinien zostać zadeklarowany wraz z użyciem `unsigned` przed nazwą typu liczby całkowitej. Zmienna typu `unsigned` może przedstawiać wartości od 0 do mniej więcej dwukrotnie większego zakresu dodatniego dla danego typu liczb całkowitych. Maksymalną wartość `unsigned int` dla używanej platformy możesz odczytać z wartości zmiennej `UINT_MAX` w pliku nagłówkowym `limits.h`.

Przeznaczony do obliczania średniego wyniku ze sprawdzianu program z listingu 3.2 mógłby mieć zadeklarowane zmienne `grade`, `total` i `average` jako typu `unsigned int`. Wynik uzyskany na sprawdzianie to wartość z przedziału od 0 do 100, więc zmienne `total` i `average` powinny mieć wartości większe lub równe zero. Jednak te zmienne zostały zadeklarowane jako `int`, ponieważ nie można kontrolować danych wejściowych, które faktycznie zostaną wprowadzone przez użytkownika — mógłby on podać np. *wartość ujemną*. Co gorsze, użytkownik może podać wartość niebędącą nawet liczbą. (W dalszej części książki dowiesz się, jak radzić sobie z takimi danymi wejściowymi).

Czasami pętla kontrolowana przez wartownik używa nieprawidłowej wartości do zakończenia działania pętli. Na przykład w przeznaczonym do obliczania średniego wyniku ze sprawdzianu programie z listingu 3.4 zakończenie pętli następuje po podaniu wartości -1 (nieprawidłowy wynik). Dlatego zadeklarowanie zmiennej `grade` jako typu `unsigned int` byłoby nieprawidłowe. Jak się przekonasz, wskaźnik EOF (ang. *End of File*) oznaczający koniec pliku (zostanie omówiony w następnym rozdziale i jest często używany do zakończenia pętli kontrolowanej przez wartownik) to również liczba ujemna. Więcej informacji na ten temat znajdziesz w rozdziale 5. książki *Secure Coding in C and C++, 2/e* napisanej przez Roberta Seacorda.

Funkcje `scanf_s()` i `printf_s()`

Dodatek K standardu C11 wprowadza znacznie bezpieczniejsze wersje funkcji `printf()` i `scanf()`, o nazwach odpowiednio `printf_s()` i `scanf_s()` — ich omówienie i związane z nimi kwestie bezpieczeństwa znajdziesz w podrozdziałach 6.13 i 7.13. Dodatek K standardu został oznaczony jako *opcjonalny*, więc nie każdy twórca kompilatora C będzie go implementował. Firma Microsoft zaimplementowała własne wersje funkcji `printf_s()` i `scanf_s()` jeszcze przed ich opublikowaniem w standardzie C11. Dlatego kompilator Microsoftu generuje komunikat ostrzeżenia dla każdego wywołania `scanf()`. W tym komunikacie programista jest informowany, że funkcja `scanf()` jest *przestarzała* (nie powinna być już stosowana) i zamiast niej powinien rozważyć użycie funkcji `scanf_s()`.

Wiele organizacji stosuje standardy tworzenia kodu źródłowego wymagające, aby kompilacja była przeprowadzana bez *komunikatów ostrzeżeń*. Istnieją dwa sposoby na wyeliminowanie dotyczących `scanf()` komunikatów ostrzeżeń w kompilatorze Microsoft Visual C++, a mianowicie zastosowanie funkcji `scanf_s()` zamiast `scanf()` lub wyłączenie komunikatów ostrzeżeń. W przypadku przedstawionych dotychczas poleceń pobierania danych wejściowych użytkownik Visual C++ może po prostu zastąpić `scanf()` funkcją `scanf_s()`. Jeżeli chcesz wyłączyć wyświetlanie komunikatów ostrzeżeń w Visual C++, skorzystaj z następującej procedury:

1. Naciśnij klawisze **Alt+F7**. To spowoduje wyświetlenie okna dialogowego *Property Pages* dla projektu.
2. W lewej kolumnie rozwiń *Configuration Properties*, a później *C/C++* i zaznacz *Preprocessor*.
3. W prawej kolumnie, na końcu wartości dla *Preprocessor Definitions*, wstaw `:_CRT_SECURE_NO_WARNINGS`.
4. Kliknij przycisk OK, aby zapisać zmiany.

Nie będziesz już otrzymywać ostrzeżeń dotyczących funkcji `scanf()` oraz innych, które firma Microsoft z podobnych powodów uznała za przestarzałe. W przypadku tworzenia solidnego kodu produkcyjnego nie zaleca się wyłączania komunikatów ostrzeżeń generowanych przez kompilator. Więcej informacji na temat używania funkcji `scanf_s()` i `printf_s()` znajdziesz w sekcjach o bezpiecznym programowaniu w C w dalszej części książki.

Podsumowanie

3.1. Wprowadzenie

- Zanim przystąpisz do tworzenia programu rozwiązującego konkretny problem, musisz dokładnie poznać ten problem i starannie zaplanować sposób jego rozwiązania.

3.2. Algorytm

- Rozwiązanie dowolnego problemu komputerowego oznacza wykonanie ciągu **akcji** w określonej **kolejności** (str. 100).
- **Procedura** (str. 100) rozwiązywania problemu w kategoriach akcji (str. 100) do wykonania i kolejności, w jakiej te akcje mają zostać wykonane, nosi nazwę **algorytmu** (str. 100).
- Kolejność wykonywania akcji jest ważna.

3.3. Pseudokod

- Pseudokod (str. 100) to sztuczny i nieformalny język pomocny w opracowaniu algorytmu.
- Pseudokod jest podobny do języka, którym się posługujesz na co dzień, i nie jest rzeczywistym językiem programowania.
- Pseudokod pomaga w przemyśleniu programu, zanim zaczniesz jego tworzenie w języku programowania.
- Pseudokod składa się jedynie ze znaków, więc możesz go wygodnie napisać za pomocą dowolnego edytora tekstu.
- Starannie przygotowany pseudokod można łatwo skonwertować na kod w języku C.
- Pseudokod składa się jedynie z poleceń akcji.

3.4. Struktury kontrolne

- Standardowo polecenia w programie są wykonywane jedno po drugim w kolejności, w której zostały zapisane. To nosi nazwę **wykonywania sekwencyjnego** (str. 101).
- Różne polecenia języka C pozwalają określić, że kolejne wykonywane polecenie ma być inne niż następne w sekwencji. To nosi nazwę **przekazania kontroli** (str. 101).
- Notacja tzw. **programowania strukturalnego** stała się praktycznie synonimem dla **eliminacji goto** (str. 101).
- Programy powstałe z wykorzystaniem technik strukturalnych stały się bardziej przejrzyste, łatwiejsze do debugowania i modyfikowania, a także znacznie rzadziej popełniano w nich błędy.
- Każdy program mógłby zostać utworzony z wykorzystaniem jedynie trzech **struktur kontrolnych: sekwencji, wyboru i iteracji** (str. 101).
- Struktura sekwencji jest prosta — o ile nie zostanie wskazane inaczej, komputer wykonuje polecenia C jedno po drugim, w kolejności, w której zostały zapisane.
- **Schemat blokowy** (str. 101) to graficzne przedstawienie algorytmu lub jego fragmentu. Zostaje przygotowany z użyciem **prostokątów, rombów, zaokrąglonych prostokątów i małych okręgów** połączonych strzałkami określanymi mianem **linii schematu blokowego** (str. 101).
- Symbol **prostokąta** to symbol **akcji** (str. 101), który wskazuje typ akcji odpowiedzialnej za dołączenie wyniku obliczenia lub za operację wejścia-wyjścia.
- **Linie schematu blokowego** pokazują kierunek wykonywania akcji.
- Podczas rysowania schematu blokowego przedstawiającego pełny algorytm pierwszym używanym symbolem jest zaokrąglony prostokąt ze słowem *Początek*, a ostatnim — zaokrąglony prostokąt ze słowem *Koniec*. Gdy rysujesz tylko fragment algorytmu, wówczas pomijasz zaokrąglone prostokąty na rzecz małych symboli okręgu nazywanych konektorami (str. 102).

- Symbol **rombu** to symbol **decyzji** (str. 102), który wskazuje decyzję do podjęcia.
- **Polecenie pojedynczego wyboru, if**, wybiera lub ignoruje jedną akcję.
- **Polecenie podwójnego wyboru, if-else**, (str. 102) wybiera między dwiema różnymi akcjami.
- **Polecenie wielokrotnego wyboru, switch**, (str. 102) na podstawie wartości wyrażenia wybiera jedną akcję z wielu dostępnych.
- Język C oferuje trzy rodzaje **struktur iteracji** (nazywanych również poleceniami powtórzeń): **while**, **do-while** i **for**.
- Segmenty schematów blokowych poleceń kontrolnych mogą być łączone ze sobą na zasadzie **stosu** (str. 102) — przez połączenie punktu wejścia jednej struktury z punktem wejścia innej.
- Istnieje też drugi sposób łączenia poleceń kontrolnych — jest to **zagnieżdżanie**.

3.5. Polecenie wyboru if

- Polecenie wyboru jest używane do wybrania jednego z alternatywnych przebiegów akcji.
- Symbol **decyzji** zawiera wyrażenie, np. warunek, które może być prawdziwe lub fałszywe. Z symbolu decyzji wychodzą dwie linie schematu blokowego: jedna wskazuje kierunek wybierany, gdy wyrażenie w symbolu jest prawdziwe, a druga — gdy to wyrażenie jest nieprawdziwe.
- Decyzja może być podjęta na podstawie dowolnego wyrażenia — jeżeli przyjmie ono wartość zero, będzie traktowane jako **fałszywe**, wartość niezerowa zaś powoduje potraktowanie wyrażenia jako **prawdziwego**.
- Konstrukcja **if** również jest poleceniem kontrolnym **pojedynczego wejścia i pojedynczego wyjścia** (str. 103).

3.6. Polecenie wyboru if-else

- Język C oferuje **operator warunkowy (?)** (str. 104), bardzo ściśle związany z konstrukcją **if-else**.
- Operator warunkowy to jedyny w języku C **operator trójargumentowy** — pobiera trzy operandy. Pierwszym operandem jest warunek, drugim operandem jest wartość całego wyrażenia warunkowego (str. 104), gdy warunek został spełniony, a trzeci operand to wartość całego wyrażenia warunkowego, gdy warunek nie został spełniony.
- **Zagnieżdżone konstrukcje if-else** (str. 105) pozwalają sprawdzić wiele przypadków dzięki umieszczeniu konstrukcji **if-else** w innych.
- Konstrukcja **if** oczekuje tylko jednego polecenia w jej definicji. Jeżeli w definicji konstrukcji **if** chcesz umieścić więcej niż jedno polecenie, muszą one znajdować się w nawiasie klamrowym.
- Zestaw poleceń umieszczonych w nawiasie klamrowym jest nazywany **poleceniem złożonym** lub **blokiem** (str. 106).
- **Błąd składni** jest wychwytywany przez kompilator. **Błąd logiczny** pojawia się w trakcie wykonywania programu. **Błąd logiczny o znaczeniu krytycznym** powoduje awarię programu i jego przedwczesne zakończenie. **Błąd logiczny, który nie ma znaczenia krytycznego**, pozwala kontynuować działanie programu, choć wygenerowane przez niego dane mogą być nieprawidłowe.

3.7. Polecenie iteracji while

- **Polecenie iteracji** (str. 107) pozwala określić akcję, która ma być powtarzana, gdy warunek został spełniony. W pewnym momencie warunek nie zostanie spełniony — wówczas nastąpi zakończenie iteracji i wykonanie pierwszego polecenia znajdującego się po tej strukturze iteracji.

3.8. Studium przypadku tworzenia algorytmu 1 — iteracja kontrolowana przez licznik

- **Iteracja oparta na liczniku** (str. 109) używa zmiennej nazywanej **licznikiem** (str. 109) do określenia, ile razy ma zostać wykonany zestaw poleceń.

- Iteracja oparta na liczniku jest często nazywana **iteracją skończoną** (str. 109), ponieważ liczba iteracji jest znana przed rozpoczęciem wykonywania pętli.
- **Wartość całkowita** to zmienna używana do akumulacji sumy szeregu wartości. Zmienna przeznaczona do przechowywania wartości całkowitej powinna zostać zainicjalizowana jako 0 przed jej użyciem w programie. W przeciwnym razie suma będzie uwzględniała także poprzednią wartość przechowywaną w pamięci zarezerwowanej dla zmiennej wartości całkowitej.
- Licznik to zmienna użyta do zliczania. Zmienna licznika jest zwykle inicjalizowana z wartością 0 lub 1, w zależności od sposobu użycia.
- **Niezainicjalizowana zmienna** zawiera **wartość „śmieciową”** (str. 110) — ostatnią wartość przechowywaną w pamięci zarezerwowanej dla danej zmiennej.

3.9. Studium przypadku tworzenia algorytmu 2 — iteracja kontrolowana przez wartownik

- **Wartość wartownika** (str. 111) — inne jej nazwy to **wartość sygnału**, **wartość fikcyjna** i **wartość flagi** — wskazuje na **zakończenie podawania danych**.
- **Iteracja kontrolowana przez wartość wartownika** jest często określana mianem **iteracji w nieskończoność** (str. 111), ponieważ liczba iteracji jest nieznana przed rozpoczęciem wykonywania pętli.
- Wartość wartownika musi zostać wybrana w taki sposób, aby nie było możliwe jej pomylenie z akceptowaną wartością danych wejściowych.
- W technice „od początku do końca, stopniowe udoskonalanie” (str. 111) **początek** to pojedyncze polecenie wyrażające ogólny sposób funkcjonowania programu. Dlatego początek praktycznie w pełni przedstawia przeznaczenie programu. W **procesie udoskonalania** następuje podział **początku** na ciąg małych zadań wymienionych w kolejności, w której powinny być wykonane.
- Typ **float** (str. 114) przedstawia liczbę zawierającą przecinek dziesiętny (taka liczba jest nazywana **liczbą zmiennoprzecinkową**).
- Dzielenie dwóch liczb całkowitych powoduje, że część ułamkowa wyniku zostaje **skrócona** (str. 115).
- Jeżeli chcesz przeprowadzić operację zmiennoprzecinkową z użyciem liczb całkowitych, konieczne jest rzutowanie liczb całkowitych na zmiennoprzecinkowe. Język C oferuje **jednoargumentowy operator rzutowania**, (**float**), który pozwala wykonać takie zadanie.
- Operator rzutowania (str. 115) przeprowadza tzw. **jawną konwersję**.
- Większość komputerów przeprowadza obliczenie wyrażenia arytmetycznego tylko wtedy, gdy typy danych operandów są identyczne. Aby zagwarantować, że operandy są tego samego typu, kompilator wykonuje operację nazywaną **niejawną konwersją** (str. 116) wybranych operandów.
- Do utworzenia operatora rzutowania używa się nawiasów, w które jest ujęta nazwa typu danych. Każdy operator rzutowania jest **operatorem jednoargumentowym**, czyli pobiera tylko jeden operand.
- **Kolejność wykonywania operatorów rzutowania to od prawej do lewej strony**, a ich pierwszeństwo jest takie samo jak innych operatorów jednoargumentowych takich jak + i -. To pierwszeństwo jest o jeden poziom wyżej niż **operatorów wielokrotności**, np. *, / i %.
- Specyfikator konwersji **%.2f** w funkcji **printf()** oznacza, że wyświetlana jest wartość zmiennoprzecinkowa z dwiema cyframi po przecinku. Jeżeli zostanie użyty specyfikator konwersji **%f**, czyli bez podanej dokładności, zostanie wykorzystana **dokładność domyślna** (str. 116) wynosząca 6.
- Podczas wyświetlania liczb zmiennoprzecinkowych są one **zaokrąglane** (str. 116) do podanej liczby cyfr po przecinku.

3.11. Operatory przypisania

- Język C oferuje wiele operatorów przypisania umożliwiających **skrócenie wyrażeń przypisania** (str. 121).
- Operator += wartość wyrażenia znajdującego się po prawej stronie operatora dodaje do wartości zmiennej po lewej stronie operatora, a wynik przypisuje zmiennej po lewej stronie operatora.
- Każde polecenie o postaci:

zmienna = *zmienna* *operator* *wyrażenie*;

w którym *operator* to jeden z operatorów dwuargumentowych +, -, *, / lub % (albo któryś spośród omówionych w rozdziale 10.) może zostać zapisane w postaci:

zmienna *operator* = *wyrażenie*;

3.12. Operatory inkrementacji i dekrementacji

- Język C oferuje jednoargumentowy **operator inkrementacji** (++) (str. 121) i jednoargumentowy **operator dekrementacji** (--) (str. 121) przeznaczone do użycia z typami liczb całkowitych.
- Jeżeli operatory inkrementacji lub dekrementacji zostały umieszczone przed zmienną, wówczas noszą nazwę **operatorów** odpowiednio **preinkrementacji** i **predekrementacji**. Natomiast umieszczenie operatorów inkrementacji lub dekrementacji po zmiennej powoduje, że są one określane mianem **operatorów** odpowiednio **postinkrementacji** i **postdekrementacji**.
- Preinkrementacja (predekrementacja) zmiennej powoduje jej inkrementację (dekrementację) o 1, a następnie użycie nowej wartości w wyrażeniu, w którym ta zmienna występuje.
- Postinkrementacja (postdekrementacja) zmiennej powoduje użycie w wyrażeniu wartości bieżącej, a następnie inkrementację (dekrementację) wartości tej zmiennej o 1.
- Podczas inkrementacji lub dekrementacji zmiennej o samą siebie w poleceniu formy preinkrementacji i postinkrementacji mają taki sam efekt. Tylko wtedy, gdy zmienna pojawia się w kontekście większego wyrażenia, preinkrementacja i postinkrementacja mają odmienny efekt (podobnie rzecz się ma z predekrementacją i postdekrementacją).

3.13. Bezpieczne programowanie w języku C

- Dodawanie liczb całkowitych może doprowadzić do tego, że wartość będzie zbyt duża, aby mogła być przechowywana w typie int. Ten problem jest znany pod nazwą **przepełnienia arytmetycznego** i może doprowadzić do niezdefiniowanego zachowania, potencjalnie umożliwiając przeprowadzenie ataku na system.
- Wartości minimalna i maksymalna, które w konkretnym systemie mogą być przechowywane w zmiennej typu int, są przedstawiane przez zmienne odpowiednio INT_MAX i INT_MIN zdefiniowane w pliku nagłówkowym *limits.h*.
- Za dobrą praktykę programistyczną uznaje się zagwarantowanie jeszcze przed przeprowadzeniem obliczeń arytmetycznych, że nie spowodują one przepełnienia. W solidnym kodzie produkcyjnym takie operacje sprawdzenia powinny być przeprowadzane dla wszystkich obliczeń, które mogą spowodować przepełnienie lub niedomiar (str. 124).
- Ogólnie rzecz biorąc, licznik przechowujący tylko wartości nieujemne powinien zostać zadeklarowany wraz z użyciem `unsigned` przed nazwą typu liczby całkowitej. Zmienna typu `unsigned` może przedstawiać wartości od 0 do mniej więcej dwukrotnie większego zakresu dodatniego dla danego typu liczb całkowitych.
- Maksymalną wartość `unsigned int` dla używanej platformy można odczytać z wartości zmiennej `UINT_MAX` w pliku nagłówkowym *limits.h*.

- **Dodatek K** standardu C11 wprowadza **znacznie bezpieczniejsze wersje funkcji `printf()` i `scanf()`** — odpowiednio `printf_s()` i `scanf_s()`. Dodatek K standardu został oznaczony jako opcjonalny, więc nie każdy twórca kompilatora C będzie go implementował.
- Firma Microsoft zaimplementowała własne wersje funkcji `printf_s()` i `scanf_s()` jeszcze przed ich opublikowaniem w standardzie C11. Dlatego kompilator Microsoftu generuje komunikat ostrzeżenia dla każdego wywołania `scanf()`. W tym komunikacie programista jest informowany, że funkcja `scanf()` jest przestarzała (nie powinna być już stosowana) i zamiast niej powinien rozważyć użycie funkcji `scanf_s()`.
- Wiele organizacji stosuje standardy tworzenia kodu źródłowego wymagające, aby kompilacja była przeprowadzana bez komunikatów ostrzeżeń. Istnieją dwa sposoby na wyeliminowanie dotyczących `scanf()` komunikatów ostrzeżeń w kompilatorze Microsoft Visual C++: użycie funkcji `scanf_s()` zamiast `scanf()` lub wyłączenie komunikatów ostrzeżeń.

Powtórzenie materiału

3.1. Uzupełnij puste pola w następujących zdaniach:

- Procedura rozwiązywania problemu w kategoriach akcji przeznaczonych do wykonania i kolejności, w jakiej powinny zostać wykonane, nosi nazwę _____.
- Określenie kolejności wykonywania poleceń przez komputer to _____.
- Każdy program można utworzyć z wykorzystaniem jedynie trzech struktur kontrolnych: _____, _____ i _____.
- Polecenie wyboru _____ jest używane do wykonania pewnej akcji, gdy warunek jest spełniony, i innej akcji w przypadku niespełnienia warunku.
- Kilka poleceń grupowanych razem w nawiasie klamrowym nosi nazwę _____.
- Polecenie iteracji _____ wskazuje, że polecenie lub grupa poleceń będą nieustannie wykonywane dopóty, dopóki dany warunek jest spełniony.
- Iteracja zestawu poleceń pewną liczbę razy nosi nazwę iteracji _____.
- Gdy wcześniej nie wiadomo, ile razy zestaw poleceń będzie powtórzony, wówczas do zakończenia iteracji można użyć wartości _____.

3.2. Utwórz cztery różne polecenia w języku C, których działanie polega na dodaniu 1 do wartości zmiennej `x`.

3.3. Utwórz pojedyncze polecenia w języku C, które wykonują następujące zadania:

- Mnożenie zmiennej `product` przez 2 za pomocą operatora `*`.
- Mnożenie zmiennej `product` przez 2 za pomocą operatorów `=` i `*`.
- Sprawdzenie, czy wartość zmiennej `count` jest większa niż 10. Jeżeli tak, powinien zostać wyświetlony komunikat Wartość `count` jest większa niż 10.
- Sprawdzenie reszty po dzieleniu `q` przez `divisor` i przypisanie wyniku zmiennej `q`. To polecenie należy zapisać na dwa sposoby.
- Wyświetlenie wartości 123.4567 jako liczby zmiennoprzecinkowej o precyzji dwóch cyfr po przecinku. Jaka wartość zostanie wyświetlona?
- Wyświetlenie wartości 3.14159 jako liczby zmiennoprzecinkowej o precyzji trzech cyfr po przecinku. Jaka wartość zostanie wyświetlona?

3.4. Utwórz polecenia w języku C, które wykonują następujące zadania:

- Zdefiniowanie zmiennych `sum` i `x` typu `int`.
- Przypisanie zmiennej `x` wartości 1.
- Przypisanie zmiennej `sum` wartości 0.

- d) Dodanie wartości zmiennej `x` do zmiennej `sum` i przypisanie wyniku zmiennej `sum`.
- e) Wyświetlenie komunikatu `Suma wynosi : i` i wartości zmiennej `sum`.
- 3.5.** Polecenia utworzone w poprzednim ćwiczeniu połącz w program obliczający sumę liczb całkowitych z przedziału od 1 do 10. Użyj konstrukcji `while` do przeprowadzenia pętli w trakcie obliczeń i inkrementacji wartości. Działanie pętli powinno się zakończyć, gdy zmienna `x` osiągnie wartość 11.
- 3.6.** Utwórz pojedyncze polecenia w języku C, które wykonuje następujące zadania:
- Pobranie za pomocą funkcji `scanf()` danych wejściowych w postaci liczby całkowitej dla zmiennej `x`. Skorzystaj ze specyfikatora konwersji `%u`.
 - Pobranie za pomocą funkcji `scanf()` danych wejściowych w postaci liczby całkowitej dla zmiennej `y`. Skorzystaj ze specyfikatora konwersji `%u`.
 - Przypisanie wartości 1 zmiennej `i` typu liczby całkowitej bez znaku.
 - Przypisanie wartości 1 zmiennej `power` typu liczby całkowitej bez znaku.
 - Mnożenie wartości zmiennej `power` i `x` oraz przypisanie wyniku zmiennej `power`.
 - Inkrementacja o 1 wartości zmiennej `i`.
 - Sprawdzenie w warunku konstrukcji `while`, czy wartość zmiennej `i` jest mniejsza lub równa `y`.
 - Wyświetlenie za pomocą funkcji `printf()` wartości zmiennej `power` w postaci liczby całkowitej bez znaku. Skorzystaj ze specyfikatora konwersji `%u`.
- 3.7.** Wykorzystując polecenia z poprzedniego ćwiczenia, utwórz program w języku C, który będzie obliczał wartość `x` podniesioną do potęgi `y`. W tym programie należy zastosować konstrukcję `while`.
- 3.8.** Odszukaj i usuń błędy w następujących fragmentach kodu:
- ```
while (c <= 5) {
 product *= c;
 ++c;
```
  - ```
scanf( "%.4f", &value );
```
 - ```
if (gender == 1)
 puts("kobieta");
else;
 puts("mężczyzna");
```
- 3.9.** Na czym polega problem z przedstawioną tutaj konstrukcją iteracji `while` (przyjmij założenie, że `z` ma wartość 100), która jest przeznaczona do obliczenia sumy liczb całkowitych z przedziału od 100 do 1?

```
while (z >= 0)
 sum += z;
```

## Odpowiedzi

- 3.1.**
- algorytm
  - struktura kontrolna
  - sekwencji, wyboru, iteracji
  - `if-else`
  - polecenie złożone lub blok
  - `while`
  - oparta na liczniku, ewentualnie skończona
  - wartownik

### 3.2.

```
x=x+1;
x += 1;
++x;
x++;
```

### 3.3.

```
a) product *= 2;
b) product = product * 2;
c)
 if (count > 10)
 puts("Wartość count jest większa niż 10.");
d)
 q %= divisor;
 q = q % divisor;
```

```
e)
 printf("%.2f", 123.4567);
```

Wyświetlona jest wartość 123,46.

```
f)
 printf("%.3f\n", 3.14159);
```

Wyświetlona jest wartość 3,142.

### 3.4.

```
a) int sum, x;
b) x = 1;
c) sum = 0;
d) sum += x; lub sum = sum + x;
e) printf("Suma wynosi: %d\n", sum);
```

### 3.5. Oto kod źródłowy programu:

```
1 // Obliczanie sumy liczb całkowitych z przedziału od 1 do 10
2 #include <stdio.h>
3
4 int main(void)
5 {
6 unsigned int x = 1; // Inicjalizacja zmiennej x
7 unsigned int sum = 0; // Inicjalizacja zmiennej sum
8
9 while (x <= 10) { // Pętla działa, dopóki wartość x jest mniejsza lub równa 10
10 sum += x; // Dodanie wartości x do sum
11 ++x; // Inkrementacja wartości zmiennej x
12 } // Koniec pętli while
13
14 printf("Suma wynosi: %u\n", sum); // Wyświetlenie obliczonej sumy
15 } // Koniec funkcji main()
```

### 3.6.

```
a) scanf("%u", &x);
b) scanf("%u", &y);
c) i = 1;
d) power = 1;
e) power *= x;
f) ++i;
```

```
g) while (i <= y)
h) printf("%d", power);
```

### 3.7. Oto kod źródłowy programu:

```
1 // Podniesienie x do potęgi y
2 #include <stdio.h>
3
4 int main(void)
5 {
6 printf("%s", "Podaj pierwszą liczbę całkowitą: ");
7 unsigned int x;
8 scanf("%u", &x); // Odczyt podanej przez użytkownika wartości dla zmiennej x
9 printf("%s", "Podaj drugą liczbę całkowitą: ");
10 unsigned int y;
11 scanf("%u", &y); // Odczyt podanej przez użytkownika wartości dla zmiennej y
12
13 unsigned int i = 1;
14 unsigned int power = 1; // Określenie potęgi
15
16 while (i <= y) { // Pętla działa, dopóki wartość i jest mniejsza lub równa y
17 power *= x; // Mnożenie wartości power przez x
18 ++i; // Inkrementacja wartości zmiennej i
19 } // Koniec pętli while
20
21 printf("%u\n", power); // Wyświetlenie obliczonej wartości
22 } // Koniec funkcji main()
```

### 3.8.

- Błąd: brak prawego nawiasu klamrowego w definicji pętli while.
- Poprawka: dodanie brakującego nawiasu po poleceniu ++c;.
- Błąd: określenie dokładności w specyfikatorze konwersji funkcji scanf().
- Poprawka: usunięcie .4 ze specyfikatora konwersji.
- Błąd: średnik umieszczony po poleceniu else w konstrukcji if-else powoduje powstanie błędu logicznego. W omawianym przypadku drugie wywołanie puts() zawsze będzie wykonywane.  
Poprawka: usunięcie średnika po poleceniu else.

### 3.9. Wartość z w pętli while nigdy się nie zmienia. To prowadzi do powstania pętli działającej w nieskończoność. Dla uniknięcia takiej pętli należy dekrementować wartość z, aby ostatecznie spadła do zera.

## Ćwiczenia

### 3.10. Odszukaj i usuń błędy w następujących fragmentach kodu (uwaga: fragment kodu może zawierać więcej niż jeden błąd):

- ```
a)
    if ( age >= 65 );
        puts( "Wartość zmiennej age musi być większa lub równa 65." );
    else
        puts( "Wartość zmiennej jest mniejsza niż 65." );
```
- ```
b)
 int x = 1, total;

 while (x <= 10) {
 total += x;
 ++x;
 }
```

```

c)
While (x <= 100)
 total += x;
 ++x;

d)
while (y > 0) {
 printf("%d\n", y);
 ++y;
}

```

**3.11.** Uzupełnij puste pola w następujących zdaniach:

- Rozwiązanie dowolnego problemu polega na wykonaniu ciągu akcji w określonej \_\_\_\_\_.
- Synonim procedury to \_\_\_\_\_.
- Zmienna akumulująca sumę wielu liczb to \_\_\_\_\_.
- Wartość specjalna używana do wskazania zakończenia przekazywania danych nosi nazwę wartości \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ lub \_\_\_\_\_.
- \_\_\_\_\_ to graficzny sposób przedstawienia algorytmu.
- Na schemacie blokowym kolejność, w której powinny być wykonywane kroki, jest wskazywana przez \_\_\_\_\_.
- Symbol prostokąta odpowiada obliczeniom, które są przeprowadzane przez polecenia, lub operacjom wejścia-wyjścia wykonywanym przez wywołania do \_\_\_\_\_ i \_\_\_\_\_ funkcji biblioteki standardowej.
- Element zapisany w symbolu decyzji jest nazywany \_\_\_\_\_.

**3.12.** Jaki będzie efekt działania przedstawionego tutaj programu?

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5 unsigned int x = 1;
6 unsigned int total = 0;
7 unsigned int y;
8
9 while (x <= 10) {
10 y = x * x;
11 printf("%d\n", y);
12 total += y;
13 ++x;
14 } // Koniec pętli while
15
16 printf("Wartość całkowita wynosi %d\n", total);
17 } // Koniec funkcji main()

```

**3.13.** Utwórz pojedyncze polecenia pseudokodu, które wykonują następujące zadania:

- Wyświetlenie komunikatu Podaj dwie liczby.
- Przypisanie zmiennej p sumy wartości zmiennych x, y i z.
- Sprawdzenie w konstrukcji if-else następującego warunku: wartość bieżąca zmiennej m jest większa niż dwukrotna wartość bieżąca zmiennej v.
- Pobranie z klawiatury wartości dla zmiennych s, r i t.

**3.14.** Przygotuj w pseudokodzie algorytmy dla następujących zadań:

- Pobranie dwóch liczb wpisanych przez użytkownika za pomocą klawiatury, obliczenie ich sumy i wyświetlenie wyniku.
- Pobranie dwóch liczb wpisanych przez użytkownika za pomocą klawiatury, ustalenie większej z nich (o ile są różne) i wyświetlenie jej na ekranie.

- c) Pobranie wpisanych przez użytkownika za pomocą klawiatury serii liczb dodatnich, obliczenie ich sumy, a następnie wyświetlenie jej na ekranie. Przyjmij założenie, że -1 to wartość wartownika oznaczająca „koniec danych”.
- 3.15.** Wskaż, które z przedstawionych tutaj zdań są *prawdziwe*, a które *falsywe*. Jeżeli zdanie uznasz za *falsywe*, uzasadnij odpowiedź.
- Doświadczenie pokazuje, że najtrudniejszym etapem rozwiązywania problemu jest opracowanie działającego programu w języku C.
  - Wartość wartownika musi być wartością, której nie będzie można pomylić z akceptowaną wartością danych wejściowych.
  - Linie schematu blokowego wskazują akcje do wykonania.
  - Warunek zapisany w symbolu decyzji zawsze zawiera operator arytmetyczny, np. +, -, \*, / i %.
  - W technice „od początku do końca, stopniowe udoskonalanie” każdy etap udoskonalania w pełni przedstawia dany algorytm.

**Dla ćwiczeń od 3.16 do 3.20 wykonaj następujące kroki:**

- Zapoznaj się z problemem.
  - Przygotuj algorytm za pomocą pseudokodu oraz techniki „od początku do końca, stopniowe udoskonalanie”.
  - Utwórz program w języku C.
  - Przetestuj program, usuń ewentualne błędy, a następnie go uruchom.
- 3.16. (Zasięg pojazdu)** Kierowcy zastanawiają się, ile kilometrów mogą pokonać po zatankowaniu samochodu do pełna. Pewien kierowca zebrał trochę danych, zapisując liczbę przejechanych kilometrów i ilość paliwa nalanego do zbiornika w trakcie każdego tankowania. Opracuj program pobierający dane wejściowe w postaci liczby przejechanych kilometrów i liczby litrów zatankowanego paliwa. Program powinien obliczyć i wyświetlić liczbę kilometrów przejechanych na litrze paliwa. Po przetworzeniu wszystkich danych wejściowych program powinien obliczyć i wyświetlić połączone zużycie paliwa uwzględniające wszystkie podane tankowania. Oto przykładowe dane wygenerowane przez taki program:

Podaj zużycie paliwa (-1 kończy program): 12.8  
 Podaj liczbę przejechanych kilometrów: 287  
 Liczba kilometrów przejechanych na litrze paliwa wynosi 22.421875

Podaj zużycie paliwa (-1 kończy program): 10.3  
 Podaj liczbę przejechanych kilometrów: 200  
 Liczba kilometrów przejechanych na litrze paliwa wynosi 19.417475

Podaj zużycie paliwa (-1 kończy program): 5  
 Podaj liczbę przejechanych kilometrów: 120  
 Liczba kilometrów przejechanych na litrze paliwa wynosi 24.000000

Podaj zużycie paliwa (-1 kończy program): -1

Średnia liczba kilometrów przejechanych na litrze paliwa wynosi 21.601423

- 3.17. (Kalkulator limitu kredytowego)** Opracuj program w języku C pozwalający ustalić w trakcie zakupu, czy dana transakcja spowoduje przekroczenie limitu dostępnego dla klienta. Dysponujesz następującymi danymi każdego klienta:
- numer konta,
  - saldo na początku miesiąca,
  - całkowita liczba transakcji klienta w danym miesiącu,
  - całkowita kwota kredytu wykorzystana przez klienta w danym miesiącu,

e) dostępny limit kredytowy.

Program powinien pobrać niezbędne dane wejściowe, obliczyć nowe saldo (wynosi ono *obecne saldo + obciążenie – limit kredytowy*) i ustalić, czy nowe saldo będzie wyższe niż limit kredytowy przyznany klientowi. Jeżeli tak, program powinien wyświetlić numer konta klienta, wysokość przyznanego mu limitu kredytowego, wysokość nowego salda i komunikat „Przekroczono limit kredytowy”. Oto przykładowe dane wygenerowane przez taki program:

Podaj numer konta (-1 kończy program): **100**  
Podaj saldo początkowe: **5394.78**  
Podaj sumę wszystkich obciążeń: **1000.00**  
Podaj sumę wszystkich uznań: **500.00**  
Podaj wysokość limitu kredytowego: **5500.00**  
Numer konta: 100  
Wysokość limitu kredytowego: 5500.00  
Saldo: 5894.78  
Przekroczono limit kredytowy.

Podaj numer konta (-1 kończy program): **200**  
Podaj saldo początkowe: **1000.00**  
Podaj sumę całkowitą wszystkich obciążeń: **123.45**  
Podaj sumę całkowitą wszystkich uznań: **321.00**  
Podaj wysokość limitu kredytowego: **1500.00**

Podaj numer konta (-1 kończy program): **300**  
Podaj saldo początkowe: **500.00**  
Podaj sumę całkowitą wszystkich obciążeń: **274.73**  
Podaj sumę całkowitą wszystkich uznań: **100.00**  
Podaj wysokość limitu kredytowego: **800.00**

Podaj numer konta (-1 kończy program): -1

**3.18. (Prowizja od sprzedaży)** Wynagrodzenie handlowca w pewnej dużej firmie chemicznej jest oparte na prowizji od sprzedaży. Handlowiec otrzymuje 200 zł tygodniowo plus 9% od wypracowanej wielkości sprzedaży w danym tygodniu. Na przykład jeśli w danym tygodniu handlowiec sprzeda produkty za kwotę 5000 zł, wówczas otrzyma 200 zł plus 9% od 5000 zł, czyli razem 650 zł. Opracuj program pobierający od użytkownika wielkość sprzedaży w poprzednim tygodniu, a następnie obliczający wynagrodzenie handlowca. Jednorazowo program ma przetwarzać dane dotyczące tylko jednego handlowca. Oto przykładowe dane wygenerowane przez taki program:

Podaj wyrażoną w złotych wielkość sprzedaży (-1 kończy program): **5000.00**  
Wynagrodzenie wynosi: 650.00 zł

Podaj wyrażoną w złotych wielkość sprzedaży (-1 kończy program): **1234.56**  
Wynagrodzenie wynosi: 311.11 zł

Podaj wyrażoną w złotych wielkość sprzedaży (-1 kończy program): -1

**3.19. (Kalkulator odsetek)** Odsetki kredytu można obliczyć za pomocą następującego wzoru:

$$\text{odsetki} = \text{kwota} * \text{oprocentowanie} * \text{dni} / 365;$$

W tym wzorze oprocentowanie jest podawane w skali rocznej, stąd dzielenie przez 365 dni. Opracuj program pobierający dane o kilku pożyczkach, a następnie obliczający i wyświetlający odsetki dla każdej z nich. Oto przykładowe dane wygenerowane przez taki program:

Podaj kwotę pożyczki (-1 kończy program): **1000.00**  
Podaj wysokość oprocentowania: **.1**  
Podaj wyrażony w dniach okres pożyczki: **365**  
Odsetki wynoszą 100.00 zł

Podaj kwotę pożyczki (-1 kończy program): **1000.00**  
 Podaj wysokość oprocentowania: **.08375**  
 Podaj wyrażony w dniach okres pożyczki: **224**  
 Odsetki wynoszą 51.40 zł

Podaj kwotę pożyczki (-1 kończy program): -1

- 3.20. (Kalkulator wynagrodzenia)** Opracuj program obliczający wynagrodzenie dla każdego pracownika firmy. Dla pierwszych 40 godzin stawka godzinowa jest standardowa, dla kolejnych zaś jest półtora raza wyższa. Otrzymujesz listę pracowników firmy, liczbę przepracowanych godzin w poprzednim tygodniu oraz stawkę godzinową każdego pracownika. Program powinien pobierać informacje dla każdego pracownika, a następnie obliczać i wyświetlać jego wynagrodzenie. Oto przykładowe dane wygenerowane przez taki program:

Podaj liczbę przepracowanych godzin (-1 kończy program): **39**  
 Podaj stawkę godzinową pracownika (0,00 zł): **10.00**  
 Wynagrodzenie wynosi 390.00 zł

Podaj liczbę przepracowanych godzin (-1 kończy program): **40**  
 Podaj stawkę godzinową pracownika (0,00 zł): **10.00**  
 Wynagrodzenie wynosi 400.00 zł

Podaj liczbę przepracowanych godzin (-1 kończy program): **41**  
 Podaj stawkę godzinową pracownika (0,00 zł): **10.00**  
 Wynagrodzenie wynosi 415.00 zł

Podaj liczbę przepracowanych godzin (-1 kończy program): -1

- 3.21. (Predekrementacja kontra postdekrementacja)** Utwórz program przedstawiający na podstawie operatora -- różnice między predekrementacją i postdekrementacją.
- 3.22. (Liczy w pętli)** Utwórz program wykorzystujący pętlę do wyświetlenia w jednym wierszu rozdzielonych trzema spacjami liczb od 1 do 10.
- 3.23. (Największa liczba)** Proces wyszukiwania największej liczby (np. wartości maksymalnej w grupie liczb) jest dość często stosowany w aplikacjach komputerowych. Na przykład program określający najlepszego pracownika będzie pobierał liczbę produktów sprzedanych przez poszczególnych handlowców. Ten, który sprzedał najwięcej produktów, zostaje zwycięzcą. Utwórz najpierw pseudokod i później na jego podstawie program pobierający serię 10 liczb nieujemnych, a następnie ustalający i wyświetlający największą z nich. *Podpowiedź:* program powinien wykorzystać następujące zmienne:
- counter — licznik odliczający do 10 (służy do monitorowania liczby podanych wartości i ustalenia, czy wszystkie zostały przetworzone);
  - number — aktualny numer wprowadzanej wartości;
  - largest — największa znaleziona dotychczas wartość.
- 3.24. (Tabelaryczne dane wyjściowe)** Utwórz program używający pętli do wyświetlenia przedstawionej tutaj tabeli wartości. Do rozdzielania kolumn tabulatorami wykorzystaj w funkcji `printf()` sekwencję sterującą `\t`.

| N  | 10*N | 100*N | 1000*N |
|----|------|-------|--------|
| 1  | 10   | 100   | 1000   |
| 2  | 20   | 200   | 2000   |
| 3  | 30   | 300   | 3000   |
| 4  | 40   | 400   | 4000   |
| 5  | 50   | 500   | 5000   |
| 6  | 60   | 600   | 6000   |
| 7  | 70   | 700   | 7000   |
| 8  | 80   | 800   | 8000   |
| 9  | 90   | 900   | 9000   |
| 10 | 100  | 1000  | 10000  |

**3.25. (Tabelaryczne dane wyjściowe)** Utwórz program używający pętli do wyświetlenia przedstawionej tutaj tabeli wartości:

| A  | A+2 | A+4 | A+6 |
|----|-----|-----|-----|
| 3  | 5   | 7   | 9   |
| 6  | 8   | 10  | 12  |
| 9  | 11  | 13  | 15  |
| 12 | 14  | 16  | 18  |
| 15 | 17  | 19  | 21  |

**3.26. (Dwie największe liczby)** Korzystając z rozwiązania podobnego do zastosowanego w ćwiczeniu 3.23, wyszukaj *dwie* największe wartości w grupie dziesięciu liczb. (Uwaga: podawane liczby powinny być unikatowe).

**3.27. (Weryfikacja danych wejściowych użytkownika)** Zmodyfikuj program przedstawiony na listingu 3.5, aby przeprowadzał sprawdzanie danych wejściowych użytkownika. Jeżeli zostanie podana wartość inna niż 1 lub 2, program nie powinien jej przyjąć i powinien poprosić o podanie prawidłowej.

**3.28.** Jaki będzie wynik działania tego programu?

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5 unsigned int count = 1; // Inicjalizacja zmiennej count
6
7 while (count <= 10) { // Iteracja ma się odbyć 10 razy
8
9 // Wyświetlenie wiersza tekstu
10 puts(count % 2 ? "*****" : "++++++");
11 ++count; // Inkrementacja zmiennej count
12 } // Koniec pętli while
13 } // Koniec funkcji main()
```

**3.29.** Jaki będzie wynik działania tego programu?

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5 unsigned int row = 10; // Inicjalizacja zmiennej row
6
7 while (row >= 1) { // Pętla działa, dopóki wartość row nie będzie mniejsza niż 1
8 unsigned int column = 1; // Przypisanie zmiennej column wartości 1 na początku iteracji
9
10 while (column <= 10) { // Iteracja ma się odbyć 10 razy
11 printf("%s", row % 2 ? "<:" : ">"); // Dane wyjściowe
12 ++column; // Inkrementacja zmiennej column
13 } // Koniec wewnętrznej konstrukcji while
14
15 --row; // Dekrementacja zmiennej row
16 puts(""); // Rozpoczęcie nowego wiersza danych wyjściowych
17 } // Koniec zewnętrznej konstrukcji while
18 } // Koniec funkcji main()
```

**3.30. (Zapomniany blok else)** Określ dane wyjściowe dla każdego z przedstawionych tutaj przypadków przy założeniu, że wartość *x* wynosi 9, a wartość *y* wynosi 11, a także gdy wartość *x* wynosi 11, a wartość *y* wynosi 9. Kompilator ignoruje wcięcia w kodzie programu w C. Ponadto kompilator zawsze powiązuje blok *else* z wcześniejszym poleceniem *if*, o ile nie nakażesz inaczej przez użycie nawiasu klamrowego, `{}`. Ponieważ na początku możesz nie wiedzieć, które polecenia *if* i *else*

do siebie pasują, jest to określane mianem problemu zapomnianego bloku `else`. Usunęliśmy wcięcia w przedstawionych fragmentach kodu, aby wyzwanie stało się nieco trudniejsze. (Podpowiedź: zastosuj poznane dotychczas konwencje wcięć).

a)

```
if (x < 10)
if (y > 10)
puts("*****");
else
puts("#####");
puts("$$$$$");
```

b)

```
if (x < 10) {
if (y > 10)
puts("*****");
}
else {
puts("#####");
puts("$$$$$");
}
```

**3.31. (Kolejny problem zapomnianego bloku `else`)** Zmodyfikuj przedstawiony tutaj fragment kodu w taki sposób, aby generować podane dane wyjściowe. Wykorzystaj prawidłowe techniki wcięć. Nie wolno wprowadzać żadnych zmian poza dodaniem nawiasów. Kompilator ignoruje wcięcia w kodzie programu. Usunęliśmy wcięcia w przedstawionych fragmentach kodu, aby wyzwanie stało się nieco trudniejsze. (Podpowiedź: może się zdarzyć, że nie trzeba będzie wprowadzać żadnych modyfikacji).

```
if (y == 8)
if (x == 5)
puts("@@@@@");
else
puts("#####");
puts("$$$$$");
puts("&&&&&");
```

a) Jeżeli  $x = 5$  i  $y = 8$ , to zostaną wygenerowane następujące dane wyjściowe:

```
@@@@@
$$$$$
&&&&&
```

b) Jeżeli  $x = 5$  i  $y = 8$ , to zostaną wygenerowane następujące dane wyjściowe:

```
@@@@@
```

c) Jeżeli  $x = 5$  i  $y = 8$ , to zostaną wygenerowane następujące dane wyjściowe:

```
@@@@@
&&&&&
```

d) Jeżeli  $x = 5$  i  $y = 7$ , to zostaną wygenerowane następujące dane wyjściowe:

```
#####
$$$$$
&&&&&
```

**3.32. (Wypełniony kwadrat z gwiazdek)** Utwórz program pobierający dane wejściowe w postaci długości boku kwadratu, a następnie generujący na tej podstawie wypełniony kwadrat z gwiazdek. Program powinien obsługiwać kwadraty o wielkości boku w przedziale od 1 do 20. Na przykład jeśli bok kwadratu zostanie podany jako 4, program powinien wygenerować następujące dane wyjściowe:

```



```

- 3.33. (Pusty kwadrat z gwiazdek)** Zmodyfikuj program utworzony w ćwiczeniu 3.32 w taki sposób, aby wyświetlał pusty kwadrat. Na przykład jeśli bok kwadratu zostanie podany jako 5, program powinien wygenerować następujące dane wyjściowe:

```

* *
* *
* *

```

- 3.34. (Palindrom)** Palindrom to liczba lub tekst, które brzmią tak samo podczas odczytu zarówno od przodu, jak i od tyłu. Na przykład każda z następujących liczb jest palindromem: 12321, 55555, 45554, 11611. Utwórz program odczytujący pięciocyfrową liczbę całkowitą i określający, czy jest ona palindromem. (*Podpowiedź:* użyj operatorów dzielenia i reszty z dzielenia do podziału grupy na poszczególne cyfry).
- 3.35. (Odpowiednik dziesiętny liczby binarnej)** Utwórz program pobierający dane wejściowe w postaci liczby całkowitej (maksymalnie pięciocyfrowej) składającej się tylko z zer i jedynek (tzw. liczba binarna), a następnie wyświetlający jej dziesiętny odpowiednik. (*Podpowiedź:* operatorów dzielenia i reszty z dzielenia użyj do pobierania cyfr liczby binarnej pojedynczo, od prawej do lewej strony. Podobnie jak w systemie dziesiętnym, w którym pierwsza cyfra po prawej stronie ma wartość 1, druga ma wartość 10, trzecia 100, czwarta 1000 itd., w systemie binarnym pierwsza cyfra po prawej ma wartość 1, druga ma wartość 2, trzecia 4, czwarta 8 itd. Dlatego liczbę dziesiętną 234 można zinterpretować jako  $4 * 1 + 3 * 10 + 2 * 100$ . Dziesiętny odpowiednik liczby binarnej 1101 to  $1 * 1 + 0 * 2 + 1 * 4 + 1 * 8$ , czyli  $1 + 0 + 4 + 8$ , czyli 13).
- 3.36. (Jak szybko działa Twój komputer?)** W jaki sposób można określić faktyczną szybkość działania komputera? Utwórz program z pętlą `while` przeprowadzającą iterację od 1 do 1 000 000 000 w kroku co 1. Po każdym osiągnięciu wielokrotności 100 000 program powinien wyświetlić tę liczbę na ekranie. Użyj zegarka do sprawdzenia, ile czasu potrzebuje Twój komputer na wykonanie 100 milionów iteracji tej pętli.
- 3.37. (Wielokrotność liczby 10)** Utwórz program wyświetlający 100 gwiazdek, jedna po drugiej. Po każdym dziesięciu program powinien wyświetlić znak nowego wiersza. (*Podpowiedź:* odliczaj od 1 do 100. Operator reszty z dzielenia wykorzystaj do ustalenia, czy wartość licznika jest wielokrotnością liczby 10).
- 3.38. (Zliczanie siódemek)** Utwórz program pobierający (maksymalnie pięciocyfrową) liczbę całkowitą, a następnie określający, ile znajduje się w niej siódemek.
- 3.39. (Wzorzec szachownicy utworzonej z gwiazdek)** Utwórz program wyświetlający z gwiazdek następujący wzorzec szachownicy:

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

Program ma korzystać tylko z trzech poleceń generujących dane wyjściowe:

```
printf("%s", " ");
printf("%s", " ");
puts(" "); // Wygenerowanie znaku nowego wiersza
```

- 3.40. (Wielokrotność liczby 2 w pętli działającej w nieskończoność)** Utwórz program wyświetlający w nieskończoność wielokrotności liczby 2, czyli 2, 4, 8, 16, 32, 64 itd. Pętla nigdy nie powinna zostać zakończona (czyli działa w nieskończoność). Co się stanie po uruchomieniu tego programu?
- 3.41. (Średnica, obwód i pole okręgu)** Utwórz program pobierający promień okręgu (w postaci wartości typu float), a następnie wyświetlający średnicę, obwód i pole okręgu. Dla pi wykorzystaj stałą wartość 3,14159.
- 3.42.** Na czym polega problem z przedstawionym tutaj poleceniem? Zmodyfikuj je w taki sposób, aby osiągnąć efekt zamierzony przez programistę.

```
printf("%d", ++(x + y));
```

- 3.43. (Krawędzie trójkąta)** Utwórz program pobierający trzy niezerowe wartości w postaci liczb całkowitych, a następnie ustalający i wyświetlający, czy mogą one przedstawiać krawędzie trójkąta.
- 3.44. (Krawędzie trójkąta prostokątnego)** Utwórz program pobierający trzy niezerowe wartości w postaci liczb całkowitych, a następnie ustalający i wyświetlający, czy mogą one przedstawiać krawędzie trójkąta prostokątnego.
- 3.45. (Silnia)** Silnia dla nieujemnej liczby całkowitej  $n$  jest zapisywana jako  $n!$  i definiowana następująco:

$$n! = n * (n - 1) * (n - 2) * \dots * 1 \text{ (dla wartości } n \text{ większej lub równej 1)}$$

i

$$n! = 1 \text{ (dla } n = 0)$$

Na przykład  $5! = 5 * 4 * 3 * 2 * 1$ , czyli 120.

- a) Utwórz program pobierający nieujemną liczbę całkowitą, a następnie obliczającą i wyświetlającą jej silnię.
- b) Utwórz program obliczający wartość stałej matematycznej  $e$  za pomocą następującego wzoru:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

- c) Utwórz program obliczający wartość  $e^x$  za pomocą następującego wzoru:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

## To robi różnicę

- 3.46. (Kalkulator wzrostu populacji na świecie)** Odszukaj w internecie informacje o aktualnej wielkości populacji na świecie i stopniu jej corocznego wzrostu. Utwórz aplikację pobierającą dane wejściowe w postaci odszukanych przez Ciebie informacji, a następnie wyświetlającą oszacowaną populację na świecie za rok, dwa, trzy, cztery i pięć lat.
- 3.47. (Kalkulator docelowej częstości rytmu serca)** Podczas ćwiczeń korzysta się z monitora pokazującego rytm serca, aby mieć pewność, że pozostanie on w bezpiecznych granicach zalecanych przez trenerów i lekarzy. Zgodnie z amerykańskim stowarzyszeniem AHA, *maksymalna częstość rytmu serca* na minutę to 220 minus wiek podany w latach. *Docelowa częstość rytmu serca* to wartość w przedziale od 50% do 85% maksimum. (Uwaga: ten wzór został podany przez stowarzyszenie AHA. Maksymalna i docelowa częstość rytmu serca będą zależały m.in. od ogólnej kondycji zdrowotnej pacjenta, sprawności fizycznej, płci itd. *Zawsze należy skonsultować się z lekarzem lub wykwalifikowanym trenerem przed przystąpieniem do wykonywania lub modyfikowania programu ćwiczeń*). Utwórz program pobierający datę urodzenia użytkownika i datę bieżącą (dzień, miesiąc i rok), a następnie obliczający i wyświetlający na ekranie wiek użytkownika (w latach) oraz maksymalną i docelową częstość rytmu serca.

**3.48. (Zapewnienie prywatności dzięki kryptografii)** Szybki rozwój komunikacji internetowej i magazynowania danych w komputerach połączonych z internetem skutkuje wzrostem zagrożenia dla prywatności użytkowników. Kryptografia zajmuje się kodowaniem danych w taki sposób, aby uzyskanie nieuprawnionego dostępu do danych było utrudnione (a najlepiej, dzięki zaawansowanym technikom szyfrowania, niemożliwe). W tym ćwiczeniu zajmiesz się prostym schematem *szyfrowania* i *deszyfrowania* danych. Firma chce wysyłać dane przez internet i zleciła Ci opracowanie programu szyfrującego, aby informacje mogły być przekazywane w znacznie bezpieczniejszy sposób. Wszystkie dane przeznaczone do wysłania mają postać czterocyfrowych liczb. Aplikacja ma pobrać liczbę wpisaną przez użytkownika, a następnie *zaszyfrować* ją w następujący sposób: każda cyfra ma zostać zastąpiona wynikiem dodania do niej 7 i pobrania reszty z dzielenia przez 10, później cyfry pierwsza i trzecia oraz druga i czwarta mają być zamienione miejscami, a na końcu program ma wyświetlić zaszyfrowaną liczbę. Utwórz również oddzielny program, który ma pobrać zaszyfrowaną liczbę i ją *odszyfrować* (przez odwrócenie procesu szyfrowania) na postać pierwotnej liczby. (*Projekt opcjonalny*: w solidnych aplikacjach produkcyjnych stosowane są znacznie silniejsze techniki szyfrowania niż przedstawiona w rozdziale. Poszukaj w internecie informacji o kryptografii klucza publicznego oraz o stosowanym w PGP (ang. *Pretty Good Privacy*) schemacie klucza publicznego. Możesz również poczytać o schemacie RSA, który jest dość powszechnie stosowany w solidnych aplikacjach produkcyjnych).

# PROGRAM PARTNERSKI

— GRUPY HELION —

- 
1. ZAREJESTRUJ SIĘ
  2. PREZENTUJ KSIĄŻKI
  3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

<http://program-partnerski.helion.pl>

GRUPA  
**Helion** 

# C: UCZ SIĘ OD NAJLEPSZYCH I PROGRAMUJ JAK MISTRZ!

Początki programowania bywają trudne, jednak jest to bardzo ważny czas dla programisty. Od tego, jakich nabierze nawyków, w jakim stopniu opanuje zasady tworzenia bezpiecznego kodu i na ile będzie przestrzegał dobrych praktyk, naprawdę wiele zależy. Konieczne jest również dogłębne zrozumienie takich podstaw informatyki jak działanie procesora, zarządzanie pamięcią, wątki czy działanie kompilatora. Okazuje się, że skuteczne przyswojenie podstaw jest zadaniem bardzo trudnym, ale wyjątkowo ważnym i odpowiedzialnym — zarówno dla ucznia, jak i nauczyciela.

To ósme, zaktualizowane i uzupełnione wydanie znakomitego podręcznika przeznaczonego dla adeptów języka C i ich nauczycieli. Zawiera doskonałe wprowadzenie do C oraz inżynierii oprogramowania. Materiał zamieszczony w książce jest aktualny i zgodny z nowoczesnymi zasadami pracy. Obszernie wyjaśniono tu zasady tworzenia i działania kodu, a także zagadnienia związane z typami danych, funkcjami, tablicami, operacjami na bitach, wyliczeniami, pracą na plikach i innymi kwestiami ważnymi z punktu widzenia funkcjonalności, wydajności i bezpieczeństwa kodu. To pozycja oparta na zasadach nowoczesnej dydaktyki — zawiera mnóstwo przydatnych przykładów, ćwiczeń, wskazówek i podsumowań. Poszczególne koncepcje wyjaśniono z użyciem pseudokodu, algorytmów i schematów, dzięki czemu zrozumienie języka C staje się dużo łatwiejsze.

## W tej książce między innymi:

- obszerne omówienie języka C
- funkcje wprowadzone w standardach C99 i C11
- zasady bezpiecznego programowania
- testy i debugowanie kodu
- kwestie wydajności a wielowątkowość i systemy wielordzeniowe
- zasady programowania zorientowanego obiektowo: wprowadzenie do C++

**Paul Deitel** jest dyrektorem w firmie Deitel & Associates, Inc. Uczył programowania w ramach współpracy z firmami: Cisco, IBM, Siemens, Sun Microsystems, Dell, Lucent Technologies, NASA, Boeing, Puma, iRobot i wieloma innymi. Jest współautorem wielu podręczników programowania.

**Dr Harvey Deitel** jest prezesem Deitel & Associates, Inc. Informatyką zajmuje się od ponad pięćdziesięciu lat. Jest autorem i współautorem licznych podręczników oraz świetnym dydaktykiem. Opracował setki kursów dla instytucji edukacyjnych, firm, urzędów i wojska.

**Publikacje Deitelów są rozpoznawane na całym świecie i były tłumaczone na wiele języków.**

|                                                                                                      |                                                                                                                                                                                                              |                                                                                                                                                                                                                                                                             |
|------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Helion</b><br> | <i>Sprawdź nasze szkolenia!</i><br><br><b>SZKOLENIA</b><br><b>AKADEMIA IT &amp; BUSINESS</b><br><b>HELIONSZKOLENIA.PL</b> | <b>KOD KORZYŚCI</b><br><i>Sięgnij po więcej!</i><br><br>ISBN 978-83-283-6286-4<br><br>9 788328 362864 |
|  <b>helion.pl</b>  |  <b>HELION SA</b><br>ul. Kościuszki 1c<br>44-100 Gliwice<br>tel.: 32 230 98 63<br>helion@helion.pl                         | <b>INFORMATYKA W NAJLEPSZYM WYDANIU</b><br>Cena: 199,00 zł                                                                                                                                                                                                                  |