

Stephen G. Kochan



Język C

Kompendium wiedzy

Wydanie IV

Kompletny przewodnik po języku C!



Helion

Tytuł oryginału: Programming in C, Fourth Edition

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-1645-4

Authorized translation from the English language edition, entitled: PROGRAMMING IN C, FOURTH EDITION; ISBN 0321776410; by Stephen G. Kochan; published by Pearson Education, Inc, publishing as SAMS Publishing.

Copyright © 2015 Pearson Education, Inc.

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 S.A. Copyright © 2015.

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 Wydawnictwo HELION 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 Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION

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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/jckom4.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/jckom4>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

	O autorze	13
	Wprowadzenie	15
Rozdział 1	Podstawy	19
	Programowanie	19
	Języki wysokiego poziomu	20
	Systemy operacyjne	20
	Kompilowanie programów	21
	Zintegrowane środowiska programistyczne	23
	Interpretery	24
Rozdział 2	Kompilujemy i uruchamiamy pierwszy program	25
	Kompilujemy nasz program	26
	Uruchamianie programu	26
	Analiza naszego pierwszego programu	27
	Wyświetlanie wartości zmiennych	29
	Komentarze	31
	Ćwiczenia	32
Rozdział 3	Zmienne, typy danych i wyrażenia arytmetyczne	35
	Typy danych i stałe	35
	Podstawowy typ danych int	36
	Typ zmiennoprzecinkowy float	37
	Rozszerzony typ double	37
	Pojedyncze znaki, typ char	38
	Logiczny typ danych, _Bool	38
	Określniki typu: long, long long, short, unsigned i signed	40
	Użycie zmiennych	42
	Wyrażenia arytmetyczne	44
	Arytmetyka liczb całkowitych i jednoargumentowy operator minus	46
	Łączenie działań z przypisaniem	51
	Typy _Complex i _Imaginary	52
	Ćwiczenia	53

Rozdział 4	Pętle w programach	55
	Liczby trójkątne	55
	Instrukcja for	56
	Operatory porównania	58
	Wyrównywanie wyników	62
	Dane wejściowe dla programu	62
	Zagnieżdżone pętle for	64
	Odmiany pętli for	66
	Instrukcja while	67
	Instrukcja do	71
	Instrukcja break	72
	Instrukcja continue	72
	Ćwiczenia	73
Rozdział 5	Podjęmowanie decyzji	75
	Instrukcja if	75
	Konstrukcja if-else	79
	Złożone warunki porównania	81
	Zagnieżdżone instrukcje if	83
	Konstrukcja else if	85
	Instrukcja switch	91
	Zmienne logiczne	94
	Operator wyboru	98
	Ćwiczenia	99
Rozdział 6	Tablice	101
	Definiowanie tablicy	102
	Użycie tablic jako liczników	106
	Generowanie ciągu Fibonacciego	108
	Zastosowanie tablic do generowania liczb pierwszych	109
	Inicjalizowanie tablic	111
	Tablice znakowe	112
	Użycie tablic do zamiany podstawy liczb	113
	Kwalifikator const	115
	Tablice wielowymiarowe	117
	Tablice o zmiennej wielkości	119
	Ćwiczenia	121
Rozdział 7	Funkcje	123
	Definiowanie funkcji	123
	Parametry i zmienne lokalne	126
	Deklaracja prototypu funkcji	127
	Automatyczne zmienne lokalne	128
	Zwracanie wyników funkcji	129

Nic, tylko wywoływanie i wywoływanie...	133
Deklarowanie zwracanych typów, typy argumentów	136
Sprawdzanie parametrów funkcji	138
Programowanie z góry na dół	139
Funkcje i tablice	140
Operatory przypisania	143
Sortowanie tablic	145
Tablice wielowymiarowe	147
Zmienne globalne	152
Zmienne automatyczne i statyczne	155
Funkcje rekurencyjne	158
Ćwiczenia	160
Rozdział 8 Struktury	163
Podstawowe wiadomości o strukturach	163
Struktura na daty	164
Użycie struktur w wyrażeniach	166
Funkcje i struktury	168
Struktura na czas	173
Inicjalizowanie struktur	176
Literały złożone	177
Tablice struktur	178
Struktury zawierające inne struktury	181
Struktury zawierające tablice	182
Wersje struktur	185
Ćwiczenia	186
Rozdział 9 Łańcuchy znakowe	189
Rozszerzenie wiadomości o łańcuchach	189
Tablice znaków	190
Łańcuchy znakowe zmiennej długości	192
Inicjalizowanie i pokazywanie tablic znakowych	194
Porównywanie dwóch łańcuchów znakowych	197
Wprowadzanie łańcuchów znakowych	199
Wczytanie pojedynczego znaku	201
Łańcuch pusty	205
Cytowanie znaków	208
Jeszcze o stałych łańcuchach	210
Łańcuchy znakowe, struktury i tablice	211
Lepsza metoda szukania	214
Operacje na znakach	218
Ćwiczenia	221

Rozdział 10 Wskaźniki	225
Wskaźniki i przekierowania	225
Definiowanie zmiennej wskaźnikowej	226
Wskaźniki w wyrażeniach	229
Wskaźniki i struktury	230
Struktury zawierające wskaźniki	233
Listy powiązane	234
Słowo kluczowe const a wskaźniki	241
Wskaźniki i funkcje	243
Wskaźniki i tablice	247
Parę słów o optymalizacji programu	251
To tablica czy wskaźnik?	251
Wskaźniki na łańcuchy znakowe	253
Stałe łańcuchy znakowe a wskaźniki	254
Jeszcze raz o inkrementacji i dekrementacji	256
Operacje na wskaźnikach	258
Wskaźniki na funkcje	260
Wskaźniki a adresy w pamięci	261
Ćwiczenia	262
Rozdział 11 Operacje bitowe	265
Podstawowe wiadomości o bitach	265
Operatory bitowe	266
Bitowy operator AND	267
Bitowy operator OR	269
Bitowy operator OR wyłączającego	270
Operator negacji bitowej	271
Operator przesunięcia w lewo	273
Operator przesunięcia w prawo	273
Funkcja przesuwająca	274
Rotowanie bitów	275
Pola bitowe	278
Ćwiczenia	281
Rozdział 12 Preprocesor	283
Dyrektywa #define	283
Rozszerzalność programu	287
Przenośność programu	288
Bardziej złożone definicje	289
Operator #	294
Operator ##	295
Dyrektywa #include	296
Systemowe pliki włączane	298

Kompilacja warunkowa	298
Dyrektywy #ifdef, #endif, #else i #ifndef	298
Dyrektywy preprocesora #if i #elif	300
Dyrektywa #undef	301
Ćwiczenia	302
Rozdział 13 Jeszcze o typach danych	
— wyliczenia, definicje typów oraz konwersje typów	303
Wyliczeniowe typy danych	303
Instrukcja typedef	306
Konwersje typów danych	309
Znak wartości	310
Konwersja parametrów	311
Ćwiczenia	312
Rozdział 14 Praca z większymi programami	313
Dzielenie programu na wiele plików	313
Kompilowanie wielu plików z wiersza poleceń	314
Komunikacja między modułami	316
Zmienne zewnętrzne	316
Static a extern: porównanie zmiennych i funkcji	319
Wykorzystanie plików nagłówkowych	320
Inne narzędzia służące do pracy z dużymi programami	322
Narzędzie make	322
Narzędzie cvs	324
Narzędzia systemu Unix	324
Rozdział 15 Operacje wejścia i wyjścia w języku C	327
Wejście i wyjście znakowe: funkcje getchar i putchar	328
Formatowanie wejścia i wyjścia: funkcje printf i scanf	328
Funkcja printf	328
Funkcja scanf	335
Operacje wejścia i wyjścia na plikach	339
Przekierowanie wejścia-wyjścia do pliku	339
Koniec pliku	342
Funkcje specjalne do obsługi plików	343
Funkcja fopen	343
Funkcje getc i putc	345
Funkcja fclose	345
Funkcja feof	347
Funkcje fprintf i fscanf	347
Funkcje fgets i fputs	348
Wskaźniki stdin, stdout i stderr	348
Funkcja exit	349
Zmiana nazw i usuwanie plików	350
Ćwiczenia	351

Rozdział 16	Różnorodności, techniki zaawansowane	353
	Pozostałe instrukcje języka	353
	Instrukcja goto	353
	Instrukcja pusta	354
	Użycie unii	355
	Przecinek jako operator	357
	Kwalifikatory typu	358
	Kwalifikator register	358
	Kwalifikator volatile	359
	Kwalifikator restrict	359
	Parametry wiersza poleceń	360
	Dynamiczna alokacja pamięci	363
	Funkcje calloc i malloc	364
	Operator sizeof	364
	Funkcja free	367
	Ćwiczenia	368
Rozdział 17	Usuwanie błędów z programów	369
	Usuwanie błędów za pomocą preprocesora	369
	Usuwanie błędów przy użyciu programu gdb	375
	Użycie zmiennych	377
	Pokazywanie plików źródłowych	379
	Kontrola nad wykonywaniem programu	379
	Uzyskiwanie śladu stosu	383
	Wywoływanie funkcji, ustawianie tablic i zmiennych	384
	Uzyskiwanie informacji o poleceniach gdb	384
	Na koniec	386
Rozdział 18	Programowanie obiektowe	389
	Czym zatem jest obiekt?	389
	Instancje i metody	390
	Program w C do obsługi ułamków	392
	Klasa Objective-C obsługująca ułamki	392
	Klasa C++ obsługująca ułamki	397
	Klasa C# obsługująca ułamki	399
Dodatek A	Język C w skrócie	403
	1.0. Dwuznaki i identyfikatory	403
	2.0. Komentarze	404
	3.0. Stałe	405
	4.0. Typy danych i deklaracje	408
	5.0. Wyrażenia	417
	6.0. Klasy zmiennych i zakres	430
	7.0. Funkcje	432
	8.0. Instrukcje	434
	9.0. Preprocesor	438

Dodatek B	Standardowa biblioteka C	445
	Standardowe pliki nagłówkowe	445
	Funkcje obsługujące łańcuchy znakowe	448
	Obsługa pamięci	450
	Funkcje obsługi znaków	451
	Funkcje wejścia i wyjścia	452
	Funkcje formatujące dane w pamięci	457
	Konwersja łańcucha na liczbę	458
	Dynamiczna alokacja pamięci	459
	Funkcje matematyczne	460
	Arytmetyka zespolona	466
	Funkcje ogólnego przeznaczenia	468
Dodatek C	Kompilator gcc	471
	Ogólna postać polecenia	471
	Opcje wiersza poleceń	471
Dodatek D	Typowe błędy	475
Dodatek E	Zasoby	481
	Język programowania C	481
	Kompilatory C i zintegrowane środowiska programistyczne	482
	Różne	483
	Skorowidz	485

Operacje wejścia i wyjścia w języku C

Jak dotąd, wszelki odczyt i zapis danych był realizowany za pośrednictwem terminala. Kiedy chcieliśmy podać programowi jakieś dane, używaliśmy funkcji `scanf` lub `getchar`. Wyniki działania wszystkich programów pokazywaliśmy, wywołując funkcję `printf`.

Sam język C nie ma żadnych specjalnych instrukcji realizujących operacje wejścia i wyjścia (I/O — *input/output*). Wszystkie te operacje realizujemy, wywołując z biblioteki standardowej specjalne funkcje. W tym rozdziale znajduje się opis niektórych funkcji wejścia i wyjścia oraz metod pracy z plikami. Oto lista poruszanych tematów:

- podstawowe wiadomości o funkcjach `putchar()` i `getchar()`;
- optymalne techniki wykorzystania funkcji `printf()` i `scanf()` polegające na użyciu znaczników i modyfikatorów;
- przekierowywanie wejścia i wyjścia z plików;
- zastosowanie funkcji plikowych i wskaźników.

Przypomnijmy sobie następującą dyrektywę `include` z programu, w którym używaliśmy funkcji `printf`:

```
#include <stdio.h>
```

Włączany tutaj plik `stdio.h` zawiera deklaracje funkcji i makr związanych z operacjami wejścia i wyjścia z biblioteki standardowej. Wobec tego, kiedy używamy funkcji z tej biblioteki, musimy włączyć do programu powyższy plik.

W tym rozdziale powiemy o wielu innych funkcjach I/O z biblioteki standardowej. Niestety, z uwagi na szczupłość miejsca nie możemy wdawać się w zbyt szczegółowe rozważania. Listę obejmującą większość funkcji z biblioteki standardowej podajemy w dodatku B.

Wejście i wyjście znakowe: funkcje `getchar` i `putchar`

Funkcja `getchar` przydała się już, kiedy chcieliśmy odczytywać dane znak po znaku. Widzieliśmy, jak można na jej bazie utworzyć funkcję `readLine` odczytującą z terminala cały wiersz tekstu — funkcja `getchar` była wywoływana raz za razem, aż do odczytania znaku nowego wiersza.

Istnieje analogiczna do `getchar` funkcja wypisująca pojedyncze znaki — `putchar`.

Wywołanie funkcji `putchar` jest doprawdy proste — jedynym parametrem jest wyświetlany znak. Zatem wywołanie:

```
putchar (c);
```

gdzie `c` jest typu `char`, spowoduje wyświetlenie znaku zapisanego w zmiennej `c`.

Wywołanie:

```
putchar ('\n');
```

spowoduje wyświetlenie znaku nowego wiersza, czyli kursor przesunie się na początek następnego wiersza.

Formatowanie wejścia i wyjścia: funkcje `printf` i `scanf`

Funkcji `printf` i `scanf` używaliśmy już wielokrotnie. W tym rozdziale dowiemy się, jakie są możliwości formatowania danych za pomocą tych funkcji.

Pierwszy parametr `printf` i `scanf` to wskaźnik na znak. Wskazuje on łańcuch formatujący. Łańcuch ten pokazuje, jak pozostałe parametry mają być wyświetlane (`printf`) lub interpretowane (`scanf`).

Funkcja `printf`

We wcześniejszych przykładowych programach widzieliśmy, jak można między znakiem `%` a tak zwanym znakiem konwersji umieszczać dodatkowe znaki dokładniej opisujące sposób wyświetlania danych. Na przykład: w programie 4.3A widzieliśmy, jak umieszczona tam liczba całkowita pozwala określić *szerokość pola*. Łańcuch formatujący `%2i` mówi, że ma być wyświetlona wyrównana do prawej strony liczba całkowita, a pole ma mieć dwa znaki szerokości. W ćwiczeniu 6. z rozdziału 4. pokazaliśmy, jak można użyć znaku minus do wyrównania wartości w polu do lewej strony.

Ogólny format specyfikacji konwersji w funkcji `printf` wygląda następująco:

```
%[flagi][szerokość][.precyzja][hll]typ
```

Pola opcjonalne ujęto w nawiasy kwadratowe, ich kolejność musi być taka, jak pokazano powyżej. W tabelach 15.1, 15.2 i 15.3 zestawiono wszystkie możliwe znaki i wartości, jakie można umieszczać bezpośrednio po znaku % i przed określeniem typu.

Tabela 15.1. Flagi funkcji printf

Flaga	Znaczenie
-	wyrównanie wartości do lewej strony
+	poprzedzenie wartości znakiem + lub -
(<i>spacja</i>)	poprzedzenie spacją wartości dodatnich
0	dopełnianie liczb zerami
#	poprzedzenie wartości ósemkowej cyfrą 0, wartości szesnastkowej napisem 0x (lub 0X); w przypadku wartości zmiennoprzecinkowych pokazanie kropki dziesiętnej; w przypadku formatów g i G zostawienie końcowych zer

Tabela 15.2. Modyfikatory funkcji printf określające szerokość i precyzję

Wartość	Znaczenie
<i>liczba</i>	minimalna szerokość pola
*	następny parametr funkcji printf ma być potraktowany jako szerokość pola
. <i>liczba</i>	minimalna liczba cyfr dla liczb całkowitych; liczba miejsc dziesiętnych dla formatów e i f; maksymalna liczba cyfr znaczących dla formatu g; maksymalna liczba znaków dla formatu s
.*	następny parametr funkcji printf zostanie potraktowany jako precyzja i zinterpretowany, jak to opisano powyżej

Tabela 15.3. Modyfikatory typu stosowane w funkcji printf

Typ	Znaczenie
hh	wyświetlenie liczby całkowitej jako znaku
h*	wyświetlenie wartości typu short integer
l*	wyświetlenie wartości typu long integer
ll*	wyświetlenie wartości typu long long integer
L	wyświetlenie wartości typu long double
j*	wyświetlenie wartości typu intmax_t lub uintmax_t
t*	wyświetlenie wartości typu ptrdiff_t
z*	wyświetlenie wartości typu size_t

*Uwaga: modyfikatory oznaczone gwiazdką mogą też występować przed znakiem konwersji n, co wskazuje, że dany argument, będący wskaźnikiem, jest określonego typu.

W tabeli 15.4 zestawiono znaki konwersji umieszczane w łańcuchu formatującym.

Tabela 15.4. Znaki konwersji funkcji printf

Znak	Służy do wyświetlania...
i lub d	liczby całkowitej
u	liczby całkowitej bez znaku
o	liczby całkowitej ósemkowej
x	liczby całkowitej szesnastkowej; jako cyfry są używane znaki a do f
X	liczby całkowitej szesnastkowej; jako cyfry są używane znaki A do F
f lub F	liczby zmiennoprzecinkowej, domyślnie z sześcioma miejscami po przecinku
e lub E	liczby zmiennoprzecinkowej w notacji naukowej (przed wykładnikiem umieszczany jest odpowiednio znak e lub E)
g	liczby zmiennoprzecinkowej w formacie f lub e
G	liczby zmiennoprzecinkowej w formacie F lub E
a lub A	liczby zmiennoprzecinkowej w formacie szesnastkowym, <code>0x.dddd±d</code>
c	pojedynczego znaku
s	łańcucha znakowego zakończonego znakiem null
p	wskaźnika
n	nie wyświetla niczego; liczba znaków dotąd wypisanych jest umieszczana w zmiennej typu <code>int</code> wskazywanej przez odpowiedni parametr (zobacz uwagę do tabeli 15.3.)
%	symbolu procentu

Tabele od 15.1 do 15.4 mogą wydawać się bardziej skomplikowane, niż to potrzebne. Jak widać, format wyników można kontrolować na różne sposoby. Najlepszym sposobem jest po prostu wykonanie dostatecznie wielu praktycznych doświadczeń. Trzeba tylko pamiętać, aby liczba znaków % w łańcuchu formatującym była równa liczbie pozostałych parametrów (oczywiście nie dotyczy to zapisu %). W przypadku użycia znaku * zamiast liczby, funkcja printf powinna otrzymać dodatkowy parametr odpowiadający gwiazdce.

Program 15.1 pokazuje część możliwości formatowania za pomocą funkcji printf.

Program 15.1. Użycie formatów funkcji printf

```
// Program pokazujący różne formaty używane w funkcji printf

#include <stdio.h>

int main (void)
{
    char    c = 'X';
    char    s[] = "abcdefghijklmnopqrstuvwxyz";
```

```

int          i = 425;
short int    j = 17;
unsigned int u = 0xf179U;
long int     l = 75000L;
long long int L = 0x1234567812345678LL;
float        f = 12.978F;
double       d = -97.4583;
char         *cp = &c;
int          *ip = &i;
int          c1, c2;

printf ("Liczby całkowite:\n");
printf ("%i %o %x %u\n", i, i, i, i);
printf ("%x %X %#x %#X\n", i, i, i, i);
printf ("%+i %i %07i %.7i\n", i, i, i, i);
printf ("%i %o %x %u\n", j, j, j, j);
printf ("%i %o %x %u\n", u, u, u, u);
printf ("%ld %lo %lx %lu\n", l, l, l, l);
printf ("%lli %llo %llx %llu\n", L, L, L, L);

printf ("\nLiczby zmiennoprzecinkowe:\n");
printf ("%f %e %g\n", f, f, f);
printf ("%2f %.2e\n", f, f);
printf ("%0f %.0e\n", f, f);
printf ("%7.2f %7.2e\n", f, f);
printf ("%f %e %g\n", d, d, d);
printf ("%.*f\n", 3, d);
printf ("%*.*f\n", 8, 2, d);

printf ("\nZnaki:\n");
printf ("%c\n", c);
printf ("%3c%3c\n", c, c);
printf ("%x\n", c);

printf ("\nŁańcuchy znakowe:\n");
printf ("%s\n", s);
printf ("%5s\n", s);
printf ("%30s\n", s);
printf ("%20.5s\n", s);
printf ("%-20.5s\n", s);

printf ("\nWskaźniki:\n");
printf ("%p %p\n", ip, cp);

printf ("Ale%n jazda!%n\n", &c1, &c2);
printf ("c1 = %i, c2 = %i\n", c1, c2);

return 0;
}

```

Program 15.1. Wyniki

```

Liczby całkowite:
425 651 1a9 425
1a9 1A9 0x1a9 0X1A9
+425 425 0000425 0000425
17 21 11 17

```

```
61817 170571 f179 61817
75000 222370 124f8 75000
1311768465173141112 110642547402215053170 1234567812345678 1311768465173141112
```

```
Liczby zmiennoprzecinkowe:
12.978000 1.297800e+01 12.978
12.98 1.30e+01
13 1e+01
    12.98 1.30e+01
-97.458300 -9.745830e+01 -97.4583
-97.458
-97.46
```

```
Znaki:
X
  X X
58
```

```
Łańcuchy znakowe:
abcdefghijklmnopqrstuvwxy
abcde
    abcdefghijklmnopqrstuvwxy
        abcde
abcde
```

```
Wskaźniki:
0xbffffc20 0xbffffbf0
```

```
Ale jazda!
c1 = 3, c2 = 8
```

Warto poświęcić nieco czasu na szczegółowe omówienie uzyskanych wyników. W pierwszym zestawie pokazujemy liczby całkowite: `short`, `long`, `unsigned` i „normalne” `int`. Pierwszy wiersz pokazuje wartość `i` dziesiętnie (`%i`), ósemkowo (`%o`), szesnastkowo (`%x`) oraz bez znaku (`%u`). Zauważmy, że przy wyświetlaniu liczby ósemkowej nie są poprzedzane zerem.

W następnym wierszu pokazano ponownie wartość `i` — najpierw szesnastkowo w formacie `%x`. Potem używamy wielkiego `X` (`%#X`), co powoduje użycie jako cyfr wielkich liter od `A` do `F`. Modyfikator `#` powoduje, że przed liczbą pojawia się wiodące `0x` (lub `0X` w przypadku formatu `%#X`).

W czwartym wywołaniu funkcji `printf` wykorzystujemy flagę `+`, aby wymusić pokazanie znaku, nawet jeśli wartość jest dodatnia (normalnie znak plus nie jest pokazywany). Dalej używamy spacji jako modyfikatora, aby wymusić umieszczenie wiodącej spacji przed liczbami dodatnimi. Czasami przydaje się to do wyrównywania mieszanych danych, czyli dodatnich i ujemnych. Następnie używamy formantu `%07` do pokazania wartości `i` wyrównanej do prawej strony w polu o długości 7 znaków. Flaga `0` oznacza wypełnienie zerami. Wobec tego przed wartością `i` — `425` — zostaną dodane cztery zera. Ostatnia konwersja w tym wywołaniu to `%.7i`. Powoduje ona wyświetlenie wartości `i` na przynajmniej siedmiu cyfrach. Ostatecznie efekt jest taki sam jak w przypadku formantu `%07i`, czyli cztery wiodące zera i dalej trzycyfrowa liczba `425`.

Piąte wywołanie `printf` pokazuje wartość zmiennej `j` typu `short int` w różnych formatach. Można używać tu dowolnych formatów całkowitoliczbowych.

Następne wywołanie `printf` pokazuje, co się stanie, jeśli użyjemy `%i` do wyświetlenia wartości typu `unsigned int`. Wartość przypisana zmiennej `u` jest większa od maksymalnej dodatniej wartości typu `signed int`, w przypadku użycia formantu `%i` pokazywana jest liczba ujemna.

Przedostatnie wywołanie `printf` pokazuje, jak modyfikator `l` jest wykorzystywany do wyświetlenia liczb całkowitych `long`. Ostatnie wywołanie `printf` demonstruje sposób wyświetlania liczb całkowitych `long long`.

Drugi zestaw wyników przedstawia różne możliwości formatowania wartości zmiennoprzecinkowych — typów `float` i `double`. Pierwszy wiersz z tej grupy to wynik wyświetlenia wartości `float` za pomocą formantów `%f`, `%e` i `%g`. Jak już wspominaliśmy, jeśli nie zostanie podane inaczej, domyślnie używanych jest sześć miejsc dziesiętnych. W przypadku formantu `%g` to `printf` decyduje, czy pokazać wartość w formacie `%e` czy `%f`; zależy to od wielkości liczby oraz od ustalonej dokładności. Jeśli wykładnik jest mniejszy niż -4 lub większy niż opcjonalnie podawana dokładność (pamiętajmy, dokładnie jest to 6), używany jest format `%e`. W przeciwnym razie używany jest format `%f`. Tak czy inaczej, usuwane są końcowe zera, a kropka dziesiętna jest pokazywana tylko wtedy, kiedy jest część ułamkowa. Ogólnie rzecz biorąc, `%g` jest formatem lepszym dla liczb zmiennoprzecinkowych, gdyż estetyczniej wygląda.

W następnym wierszu wyników wykorzystano modyfikator `.2`, aby ograniczyć wyświetlanie `f` do dwóch miejsc po przecinku. Jak widać, funkcja `printf` jest na tyle miła, że automatycznie zaokrągliła wartość `f`. W następnym wierszu mamy dokładność `.0`, co oznacza niepokazywanie żadnych miejsc po przecinku ani kropki dziesiętnej. Wartość `f` ponownie jest zaokrąglana.

Modyfikatory `7.2` użyte do utworzenia następnego wiersza wyniku pokazują wartość wyświetlaną w przynajmniej 7 kolumnach, z dwoma miejscami po przecinku. Obie wartości mają mniej niż siedem cyfr, więc `printf` wyrównuje wartość do prawej strony (dodając z lewej strony spacje).

W następnym trzech wierszach wyświetlana jest wartość zmiennej `d` typu `double`. Użyte są takie same znaki formatujące jak w przypadku `float`; przypomnijmy, że wartości `float` przekazywane do funkcji są automatycznie konwertowane na typ `double`. Wywołanie:

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

powoduje, że zmienna `d` jest wyświetlana z trzema miejscami po przecinku. Gwiazdka znajdująca się za kropką mówi, że funkcja `printf` ma pobrać następny parametr z listy i potraktować go jako precyzję. W tym wypadku następnym parametrem jest `3`. Wartość ta mogłaby też zostać podana w zmiennej:

```
printf ("%.*f\n", dokladnosc, d);
```

co pozwala dynamicznie zmieniać format wyświetlania wartości.

Ostatni wiersz dotyczący wartości `float` i `double` pokazuje wynik użycia znaków formatujących `%*`. `*f` do wyświetlania wartości zmiennej `d`. W tym wypadku w parametrach funkcji `printf` przekazywane są zarówno szerokość pola, jak i dokładność. Pierwszym parametrem po łańcuchu formatującym jest `8`, więc jest to szerokość pola. Drugim parametrem jest `2`, i to staje się liczbą miejsc po przecinku. Wobec tego wartość `d` jest wyświetlana na ośmiu znakach, z dwoma miejscami po przecinku. Zauważmy, że znak minus i kropka dziesiętna wliczane są do długości pola; dotyczy to zresztą wszystkich specyfikatorów pola.

Dalej wyświetlamy znak `c`, który początkowo miał wartość `x`. Najpierw pokazujemy go, korzystając z dobrze znanego formantu `%c`, następnie pokazujemy go dwukrotnie w polu o szerokości `3`. W wyniku tego otrzymujemy dwie wiodące spacje.

Możemy wyświetlić znak, korzystając z dowolnej całkowitoliczbowej specyfikacji formatu. W naszym wypadku otrzymujemy szesnastkową wartość `58`, czyli wartość odpowiadającą znakowi `x`.

Ostatni zbiór danych wynikowych to wyświetlanie łańcucha znakowego `s`. Najpierw korzystamy ze zwykłego formantu `%s`. Następnie dodajemy specyfikator szerokości pola — `5`. Pokazywanych jest pięć pierwszych znaków łańcucha, czyli pięć pierwszych liter alfabetu.

Trzecia instrukcja `printf` z tej grupy pokazuje cały łańcuch, gdyż ustawiliśmy szerokość pola na `30`. Jak widać, łańcuch jest w tym polu wyrównany do prawej strony.

Ostatnie dwa wiersze tej grupy pokazują łańcuch `s` wyświetlony w polu o szerokości `20`. Za pierwszym razem pięć znaków wyrównano do prawej strony. Za drugim razem użycie znaku minus powoduje pokazanie pierwszych pięciu liter wyrównanych do lewej. Pokazana została pionowa kreska, aby sprawdzić, że łańcuch formatujący `%-20.5s` faktycznie pokazuje `20` znaków (pięć liter, dalej `15` spacji).

Za pomocą formantu `%p` pokazujemy wartość wskaźnika. Tutaj wyświetlamy wskaźnik na liczbę typu `int` — `ip` — oraz wskaźnik znaku — `cp`. Czytelnicy otrzymają inne liczby, gdyż prawdopodobnie wskaźniki będą pokazywały inne adresy w pamięci.

Postać wyniku w przypadku użycia formatu `%p` jest zależna od konkretnej implementacji; w naszym przykładzie pokazujemy adresy zapisane szesnastkowo. Zgodnie z pokazanym wynikiem zmienna wskaźnikowa `ip` zawiera adres `ffffffc20`, a wskaźnik `cp` — `ffffffbf0`.

Ostatnie wyniki demonstrują użycie formantu `%n`. Parametrem funkcji `printf` odpowiadającym temu formantowi musi być wskaźnik na liczbę `int`, chyba że użyte zostaną modyfikatory `hh`, `h`, `l`, `ll`, `j`, `z` lub `t`. Funkcja w przekazanej zmiennej umieści liczbę znaków zapisanych dotąd do wyniku. Wobec tego pierwsze wywołanie `%n` powoduje zapisanie w zmiennej `c1` liczby `3`, gdyż do chwili wstawienia tego formantu wypisane są już trzy znaki. Drugie użycie `%n` daje już wartość `8`, gdyż tyle znaków wyświetliła dotąd funkcja `printf`. Zauważmy, że włączenie do łańcucha formatującego `%n` nie wpływa na postać uzyskiwanego wyniku.

Funkcja scanf

Podobnie jak printf, tak samo scanf pozwala użyć wielu różnych opcji w łańcuchu formatującym. Tak samo jak w printf, między znakiem % a znakiem konwersji wstawia się opcjonalne modyfikatory. Zostały one zestawione w tabeli 15.5. Dopuszczalne znaki konwersji zestawiono z kolei w tabeli 15.6.

Tabela 15.5. Modyfikatory konwersji w funkcji scanf

Modyfikator	Znaczenie
*	pole pomijane lub nieprzypisywane
<i>rozmiar</i>	maksymalna wielkość pola wejściowego
hh	wartość będzie zapisana jako signed lub unsigned char
h	wartość będzie zapisana jako short int
l	wartość będzie zapisana w zmiennej typu long int, double lub wchar_t
j, z lub t	wartość będzie zapisana w zmiennej typu size_t (%j), ptrdiff_r (%z), intmax_t lub uintmax_t (%t)
ll	wartość będzie zapisana jako long long int
L	wartość będzie zapisana jako long double
<i>typ</i>	znak konwersji

Kiedy funkcja scanf przeszukuje łańcuch wejściowy, zawsze pominie wiodące białe znaki (spacje, tabulatory '\t', tabulatory pionowe '\v', znaki powrotu karetki '\r', znaki nowego wiersza '\n' lub nowej strony '\f'). Wyjątkiem jest formant %c, w którym odczytywany jest następny znak, choćby był biały, oraz łańcuch znakowy w nawiasach kwadratowych — wtedy w tych nawiasach zapisane jest, jakie znaki mogą wchodzić w skład wczytywanego łańcucha (lub jakie nie mogą wchodzić).

Kiedy funkcja scanf wczytuje jakąś wartość, czytanie to kończy się po wczytaniu liczby znaków określonej w szerokości pola lub po natknięciu się na niedozwolony znak. Dla liczb całkowitych dopuszczalne są ciągi cyfr, ewentualnie poprzedzone znakiem. Zestaw cyfr jest zależny od stosowanego zapisu: 0 – 7 dla liczb ósemkowych, 0 – 9 dla dziesiętnych, 0 – 9 i a – f lub A – F dla zapisu szesnastkowego. Dla wartości zmiennoprzecinkowych można wczytywać łańcuchy cyfr, za którymi może być kropka dziesiętna i drugi łańcuch cyfr, dalej litera e lub E oraz wykładnik, ewentualnie ze znakiem. W przypadku formantu %a czytana wartość szesnastkowa musi być podana w formacie z wiodącym 0x, dalej ciąg cyfr szesnastkowych z opcjonalną kropką dziesiętną i opcjonalnym wykładnikiem poprzedzonym literą p lub P.

Jeśli łańcuch znakowy jest wczytywany za pomocą formantu %s, poprawne są wszystkie znaki inne niż białe. W przypadku formantu %c czytane są dowolne znaki. W końcu łańcuch z nawiasami kwadratowymi dopuszcza znaki z nawiasów (lub znaki w nawiasach niewystępujące).

Przypomnijmy sobie rozdział 8., w którym pisaliśmy programy żądające od użytkownika podania czasu, kiedy wszelkie znaki nienależące do formantu musiały pojawić się w danych wejściowych `scanf`. Na przykład wywołanie funkcji `scanf`:

```
scanf ("%i:%i:%i", &hour, &minutes, &seconds);
```

wymusza wczytanie trzech liczb całkowitych i zapisanie ich w zmiennych `hour`, `minutes` i `seconds`. W łańcuchu formatującym znak `:` oznacza sam siebie, to znaczy musi się pojawić we wprowadzanych danych między poszczególnymi liczbami.

Tabela 15.6. Znaki konwersji funkcji `scanf`

Znak	Działanie
d	Wczytywana wartość będzie liczbą dziesiętną. Odpowiedni parametr jest typu <code>int</code> , chyba że zastosowano modyfikatory <code>h</code> , <code>l</code> lub <code>ll</code> ; wtedy parametry są odpowiednio typu <code>short</code> , <code>long</code> lub <code>long long int</code> .
i	Działa podobnie jak <code>%d</code> , ale możliwe jest także wczytywanie wartości ósemkowych (z wiodącym zerem) oraz szesnastkowych (wiodące <code>0x</code> lub <code>0X</code>).
u	Wartość jest wczytywana jak liczba dziesiętna, natomiast parametr jest wskaźnikiem do zmiennej typu <code>unsigned int</code> .
o	Wczytywana wartość jest zapisana ósemkowo, opcjonalnie może być poprzedzona zerem. Odpowiedni parametr jest typu <code>int</code> , chyba że zastosowano modyfikatory <code>h</code> , <code>l</code> lub <code>ll</code> ; wtedy parametry są odpowiednio typu <code>short</code> , <code>long</code> lub <code>long long</code> .
x	Wczytywana wartość jest zapisana szesnastkowo, może być opcjonalnie poprzedzona ciągiem <code>0x</code> lub <code>0X</code> . Odpowiedni parametr jest typu <code>unsigned int</code> , chyba że zastosowano modyfikatory <code>h</code> , <code>l</code> lub <code>ll</code> .
a, e, f lub g	Wartość będzie czytana jako liczba zmiennoprzecinkowa; może być poprzedzona znakiem <code>i</code> ewentualnie zapisana w notacji naukowej (na przykład <code>3.45 e-3</code>). Odpowiedni parametr jest wskaźnikiem na liczbę <code>float</code> , chyba że użyte zostaną modyfikatory <code>l</code> lub <code>L</code> oznaczające wskaźnik odpowiednio typu <code>double</code> lub <code>long double</code> .
c	Wczytany zostanie pojedynczy znak. Będzie to najbliższy znak, nawet jeśli będzie to spacja, tabulator, znak nowego wiersza czy nowej strony. Odpowiedni parametr jest wskaźnikiem na <code>char</code> . Przed <code>c</code> może pojawić się licznik mówiący, ile znaków należy odczytać.
s	Wczytywany jest łańcuch znakowy zaczynający się pierwszym niebiałym znakiem, kończący się na pierwszym białym znaku. Odpowiedni parametr jest wskaźnikiem na tablicę znakową, która musi mieć dość miejsca, aby odczytać cały łańcuch plus znak <code>null</code> , który zostanie automatycznie dodany. Jeśli przed <code>s</code> będzie podana liczba, odczytanych zostanie tyle znaków, chyba że wcześniej pojawi się biały znak.

Tabela 15.6. Znaki konwersji funkcji scanf (ciąg dalszy)

Znak	Działanie
[...]	Znaki podane w nawiasach kwadratowych oznaczają wczytanie łańcucha znakowego, podobnie jak %s, używać można w tym łańcuchu tylko znaków z nawiasów. Jakikolwiek znak spoza nawiasów kończy wczytywanie łańcucha. Możemy odwrócić interpretację nawiasów, podając po otwierającym nawiasie klamrowym karetkę ^. Wtedy wczytywane będą jedynie znaki niewystępujące w nawiasie, a dowolny znak z nawiasu przerwie wprowadzanie danych.
n	Nic nie jest wczytywane, ale do zmiennej typu int wskazywanej przez następny parametr funkcji scanf jest wstawiana liczba odczytanych dotąd znaków.
p	Wczytywana jest wartość wskaźnika w takim samym formacie, w jakim pokazuje wskaźniki funkcja printf w przypadku użycia formantu %p. Odpowiedni parametr musi być wskaźnikiem typu void.
%	Następnym niebiałym znakiem wejściowym musi być %.

Aby wskazać, że w danych wejściowych powinien pojawić się symbol procentu, trzeba włączyć do łańcucha podwójne wystąpienie takiego znaku:

```
scanf ("%i%%", &percentage);
```

Białe znaki występujące w łańcuchu zastępują dowolną liczbę białych znaków w danych wejściowych. Wobec tego wywołanie:

```
scanf ("%i%c", &i, &c);
```

kiedy podano następujące dane:

```
29 w
```

spowoduje przypisanie zmiennej i wartości 29, a zmiennej c spacji, gdyż jest to pierwszy znak po 29. Gdyby z kolei użyty został zapis:

```
scanf ("%i %c", &i, &c);
```

i podane zostałyby takie same dane, zmiennej i także przypisano by wartość 29, a zmiennej c przypisany zostałby znak 'w', gdyż spacja pojawiająca się w łańcuchu formatującym powoduje, że funkcja scanf pomija wszystkie białe znaki występujące po wartości 29.

W tabeli 15.5 napisano, że można użyć gwiazdki do pomijania pól. Jeśli funkcję scanf wywołamy następująco:

```
scanf ("%i %5c %*f %s", &i1, text, string);
```

i podamy jej dane:

```
144abcde 736.55 (wino i ser)
```

to w zmiennej `i1` zapisana zostanie wartość 144. Pięć znaków — `abcde` — zostanie zapisanych w tablicy znakowej `text`. Dalej dobrana zostanie wartość 736.55, która nie zostanie przypisana żadnej zmiennej. W zmiennej `string` zostanie umieszczony łańcuch "(wino" uzupełniony znakiem null. Następne wywołanie `scanf` zacznie swoje działanie od miejsca, gdzie poprzednie skończyło. Wobec tego następne wywołanie, jeśli będzie miało postać:

```
scanf ("%s %s %i", string2, string3, &i2);
```

to w `string2` zapisze łańcuch "i", w `string3` łańcuch "ser)". W końcu funkcja będzie czekała na podanie wartości całkowitoliczbowej.

Pamiętajmy, że funkcji `scanf` trzeba podawać wskaźniki do zmiennych, w których mają być zapisywane odczytane wartości. Z rozdziału 10. wiemy, dlaczego jest to niezbędne — dzięki temu `scanf` może zmieniać wartości tych zmiennych, czyli zapisywać w nich odczytane dane. Pamiętajmy też o tym, że aby wskazać tablicę, wystarczy podać jej nazwę. Jeśli zatem `text` jest tablicą znakową odpowiedniej wielkości, wywołanie `scanf`:

```
scanf ("%80c", text)
```

odczyta 80 znaków i zapisze je w zmiennej `text`.

Wywołanie funkcji `scanf`:

```
scanf ("%[^/]", text);
```

oznacza, że wczytywany łańcuch składa się z dowolnych znaków z wyjątkiem ukośnika. Jeśli zatem mamy dane:

```
(wino i ser)/
```

to w łańcuchu `text` zapisane zostanie "(wino i ser)"; przerwanie czytania nastąpi na znaku / (który pozostaje do następnego wywołania `scanf`). Aby wczytać cały wiersz z terminala do tablicy znakowej `buf`, używamy nawiasów kwadratowych, a jako znak wyłączonego podajemy znak nowego wiersza:

```
scanf ("%[^\n]\n", buf);
```

Znak nowego wiersza jest powtórzony za nawiasami, dzięki czemu `scanf` dopasuje go do znaku, który przerywa dobieranie znaków do nawiasów kwadratowych i nic nie zostanie do następnego wywołania `scanf`. Omawiana funkcja zawsze zaczyna swoje działanie od miejsca, w którym jej poprzedniczka je skończyła.

Kiedy wczytywane dane nie pasują do wartości oczekiwanej przez `scanf` (na przykład pojawia się znak `x`, a chcemy wczytać liczbę całkowitą), funkcja ta kończy swoje działanie i nie szuka już dalszych dopasowań. Funkcja zwraca liczbę odczytanych danych, ta zwrócona wartość może służyć do sprawdzania błędów wczytywania. Na przykład w wywołaniu:

```
if ( scanf ("%i %f %i", &i, &f, &l) != 3 )
    printf ("Błąd w danych wejściowych\n");
```

sprawdzamy, czy `scanf` prawidłowo odczytała wszystkie trzy wartości. Jeśli nie, pokazujemy stosowny komunikat.

Pamiętajmy w końcu, że wartość zwracana przez `scanf` to liczba wartości wczytanych i *przypisanych* zmiennym, zatem wywołanie:

```
scanf ("%i %*d %i", &i1, &i3)
```

zwróci 2, a nie 3, gdyż czytamy i przypisujemy wartości *dwóch* liczb całkowitych (jedną liczbę pomijamy). Zauważmy, że formant `%n` określający liczbę wczytanych znaków nie jest uwzględniany w wartości zwracanej przez `scanf`.

Warto poeksperymentować z różnymi opcjami formatowania w funkcji `scanf`. Tak jak w przypadku funkcji `printf`, dobre zrozumienie formantów możemy osiągnąć tylko w czasie sprawdzania ich działania w praktyce.

Operacje wejścia i wyjścia na plikach

Jak dotąd, kiedy wywoływaliśmy funkcję `scanf`, dane były zawsze czytane z terminala. Analogicznie wszystkie wywołania funkcji `printf` powodowały wyświetlanie danych w aktywnym oknie. Teraz nauczymy się czytać dane z plików i pisać je do plików, aby móc pisać jeszcze przydatniejsze programy.

Przekierowanie wejścia-wyjścia do pliku

Zarówno czytanie, jak i pisanie danych z plików i do nich jest łatwe w wielu systemach operacyjnych — jak choćby Linux, Unix czy Windows — po prostu nie musimy robić niczego specjalnego w programie. Spójrz na program 15.2. Jest bardzo prosty, a jego działanie ogranicza się do wykonania pewnych prostych operacji na podanej liczbie.

Program 15.2. Prosty przykład

```
// Pobranie prostej liczby i wyświetlenie kilku wyników obliczeń
```

```
#include <stdio.h>

main()
{
    float d = 6.5;
    float half, square, cube;

    half = d/2;
    square = d*d;
    cube = d*d*d;

    printf("\nPodana liczba to: %.2f\n", d);
    printf("Połowa tej liczby to: %.2f\n", half);
    printf("Kwadrat tej liczby to: %.2f\n", square);
    printf("Sześcian tej liczby to: %.2f\n", cube);

    return 0;
}
```

Nie ma tu nic skomplikowanego, ale wyobraź sobie, że chcesz zapisać wyniki w pliku o nazwie *results.txt*. Wówczas w systemie Unix i Windows wystarczy przejść do wiersza poleceń i wykonać polecenie przekierowujące dane wytworzone przez program do wybranego pliku, jak w poniższym przykładzie:

```
program1502 > results.txt
```

Powyższe polecenie nakazuje systemowi wykonać program *program1502*, ale jednocześnie przekierować wyniki normalnie wyświetlane na terminalu do pliku *results.txt*. Wobec tego wszelkie wartości wyświetlane przez `printf` nie pojawiają się w oknie, ale są zapisywane we wskazanym pliku.

Choć program z listingu 15.2 jest ciekawy, byłby o wiele bardziej interesujący, gdyby prosił użytkownika o podanie liczby i dopiero na niej wykonywał różne działania. Na listingu 15.3. pokazano realizację tego pomysłu.

Program 15.3. Prosty, ale bardziej interaktywny przykład

// Program odbierający od użytkownika jedną liczbę i zwracający wyniki kilku działań arytmetycznych.

```
#include <stdio.h>

main()
{
    float d ;
    float half, square, cube;

    printf("Wpisz liczbę od 1 do 100: \n");
    scanf("%f", &d);
    half = d/2;
    square = d*d;
    cube = d*d*d;

    printf("\nPodana liczba to %.2f\n", d);
    printf("Połowa tej liczby to: %.2f\n", half);
    printf("Kwadrat tej liczby to: %.2f\n", square);
    printf("Sześcian tej liczby to: %.2f\n", cube);
    return 0;
}
```

Teraz wyobraź sobie, że chcesz zapisać dane z programu w pliku o nazwie *results2.txt*. W tym celu napisałbyś następujące polecenie:

```
program1503 > results2.txt
```

Tym razem program może wyglądać, jakby się zawiesił. Istotnie, częściowo tak jest. Program zatrzymał się, ponieważ oczekuje aż użytkownik wprowadzi liczbę, na której mają zostać wykonane obliczenia. Jest to wada tej metody przekierowywania wyników do pliku. Wszystko zostaje przekierowane, nawet instrukcja wywołania funkcji `printf()` użyta do wyświetlenia prośby o wpisanie liczby. Jeśli zajrzysz do pliku *results2.txt*, to znajdziesz w nim następującą zawartość (przy założeniu, że na wejściu podano liczbę 6.5):

```
Wpisz liczbę z przedziału od 1 do 100:
```



```
Podana liczba to: 6.50
Połowa tej liczby to: 3.25
Kwadrat tej liczby to: 42.25
Sześcian tej liczby to: 274.63
```

Zatem faktycznie wyniki programu zostały skierowane do naszego pliku. Moglibyśmy też to samo doświadczenie wykonać z wieloma wierszami wynikowymi, aby się przekonać, że takie rozwiązanie zawsze działa prawidłowo.

Podobne przekierowanie można odnieść do danych wejściowych programu. Wszelkie wywołania funkcji normalnie odczytujących dane w oknie będą korzystały z pliku; dotyczy to na przykład `scanf` i `getchar`. Utwórz plik zawierający jedną liczbę (w ramach przykładu wykorzystam plik o nazwie `simp4.txt` z liczbą 4) i ponownie uruchom program 1503, ale tym razem za pomocą poniższego polecenia:

```
program1503 < simp4.txt
```

W terminalu pojawią się następujące informacje:

```
Wpisz lczbę z przedziału od 1 do 100:
```

```
Podana liczba to: 4.00
Połowa tej liczby to: 2.00
Kwadrat tej liczby to: 16.00
Sześcian tej liczby to: 64.00
```

Zauważmy, że program zażądał podania liczby, ale na nią nie czekał. Po prostu wejście programu zostało przekierowane do pliku, natomiast wyjście już nie. Wobec tego `scanf` wczytuje wartości z pliku `simp4.txt`. W pliku tym dane trzeba wpisywać tak samo, jak podaje się je w oknie terminala. Dla funkcji `scanf` nie ma znaczenia, skąd biorą się jej dane, z okna czy z pliku. Ważne jest tylko, aby były one poprawnie sformatowane.

Oczywiście można jednocześnie przekierować wejście i wyjście programu:

```
program1503 < simp4.txt > results3.txt
```

Teraz program sam odczyta dane z pliku `simp4.txt`, a zapisze je w pliku `results3.txt`.

Przekierowywanie wejścia i wyjścia do pliku bardzo często jest wystarczającym rozwiązaniem. Załóżmy na przykład, że piszemy artykuł do gazety i wpisywaliśmy tekst do pliku `article`. Program 9.8 zliczał słowa w tekście. Tego samego programu możemy teraz użyć do zliczenia słów w naszym artykule; wystarczy wydać polecenie¹:

```
wordcount < article
```

Oczywiście musimy pamiętać o wstawieniu dodatkowego znaku na końcu pliku `article`, gdyż nasz program stwierdzał koniec danych na podstawie istnienia wiersza zawierającego tylko znak nowego wiersza.

¹ System Unix ma polecenie `wc`, które także zlicza słowa. Przypomnijmy, że nasz program przeznaczony jest do pracy z plikami tekstowymi, a nie na przykład z plikami programu MS Word.

Zauważmy, że przekierowanie wejścia i wyjścia nie jest częścią definicji C zgodnej z ANSI. Oznacza to, że możemy natknąć się na system operacyjny, w którym takie przekierowanie nie zadziała.

Koniec pliku

Powyższa uwaga o końcu danych wymaga dokładniejszego omówienia. Kiedy mamy do czynienia z plikami, warunek końca danych zastępujemy warunkiem *końca pliku*. Warunek ten zachodzi, kiedy z pliku odczytano ostatni fragment danych. Próba czytania za końcem pliku mogłaby spowodować zakończenie programu z błędem lub wejście programu w pętlę nieskończoną. Na szczęście większość funkcji wejścia i wyjścia ma specjalną flagę wskazującą, kiedy program osiągnął koniec pliku. Wartość tej flagi to specjalna wartość — EOF — która jest zdefiniowana w nagłówku `<stdio.h>`.

W ramach przykładu użycia warunku EOF z funkcją `getchar` spójrzmy na program 15.4. Wczytuje on znaki i pokazuje je w oknie terminala, aż osiągnięty zostanie koniec pliku. Zwróćmy uwagę na wyrażenie występujące w pętli `while`. Jak widać, przypisanie nie musi być wykonywane w osobnej instrukcji.

Program 15.4. Kopiowanie znaków ze standardowego wejścia na standardowe wyjście

// Program pokazujący podawane znaki aż do napotkania końca pliku

```
#include <stdio.h>

int main (void)
{
    int c;

    while ( (c = getchar ()) != EOF )
        putchar (c);

    return 0;
}
```

Jeśli skompilujemy i uruchomimy program 15.4 (nazwijmy go *copyprog*), a następnie przekierujemy wejście z pliku:

```
copyprog < infile
```

program pokaże na terminalu zawartość pliku *infile*. Spróbujmy! Tak naprawdę program ten działa tak samo jak polecenie `cat` dostępne w systemie Unix, pozwalające wyświetlić zawartość wybranego pliku tekstowego.

W pętli `while` programu 15.4 znak zwracany przez funkcję `getchar` jest umieszczany w zmiennej `c` i porównany ze stałą EOF zdefiniowaną dyrektywą `define`. Jeśli wartości te są sobie równe, oznacza to, że odczytaliśmy znak końca pliku. Trzeba tu wspomnieć o jednym ważnym aspekcie działania funkcji `getchar` — nie zwraca ona wartości typu `char`, lecz typu `int`. Chodzi o to, że EOF musi być niepowtarzalne — takiej samej wartości nie może mieć żaden znak zwracany normalnie przez `getchar`. Wobec tego wartość zwracaną przez

getchar przypisujemy zmiennej typu `int`, a nie `char`. Działa to poprawnie, gdyż język C pozwala przechowywać znaki w zmiennych typu `int`, choć może to być nie najlepsza praktyka programistyczna.

Jeśli wynik działania funkcji `getchar` umieszczalibyśmy w zmiennej typu `char`, efekt działania programu byłby nieprzewidywalny. Kod mógłby działać poprawnie w systemach wykorzystujących jako znaki wartości ze znakiem. W systemach niemających rozszerzającego znaku moglibyśmy wpaść w nieskończoną pętlę.

Aby program zawsze działał poprawnie, trzeba po prostu zapisywać wynik funkcji `getchar` w zmiennej typu `int`; wtedy można będzie bezproblemowo wykryć koniec pliku.

To, że przypisanie wykonujemy w samym warunku pętli, pokazuje elastyczność języka C w zakresie zapisywania wyrażeń. Przypisanie trzeba umieścić w nawiasach, gdyż operator przypisania ma priorytet niższy niż operator „nierówne”.

Funkcje specjalne do obsługi plików

Bardzo prawdopodobne, że wiele tworzonych programów całą obsługę wejścia i wyjścia będzie realizowało przy użyciu funkcji `getchar`, `putchar`, `scanf` i `printf` oraz przekierowania. Jednak czasami potrzebna jest większa elastyczność obsługi plików. Konieczne bywa na przykład czytanie danych z wielu plików lub zapisywanie wyników do wielu plików. Aby obsłużyć tego typu sytuacje, utworzono specjalne funkcje służące do obsługi plików. Teraz opiszemy niektóre z nich.

Funkcja `fopen`

Zanim zaczniemy wykonywać jakiegokolwiek operacje wejścia i wyjścia na pliku, musimy najpierw ten plik *otworzyć*. Aby otworzyć plik, musimy podać jego nazwę. System sprawdza, czy plik istnieje, a w pewnych sytuacjach może plik utworzyć. Kiedy plik jest otwierany, trzeba podać rodzaj operacji, jakie będą na nim wykonywane. Jeśli plik służy do odczytu danych, zwykle otwiera się go w trybie *do odczytu*. Gdy chcemy w pliku zapisywać dane, otwieramy go w trybie *do zapisu*. Kiedy chcemy dopisywać informacje na końcu pliku, otwieramy go w trybie *dopisywania*. W dwóch ostatnich trybach plik zostanie utworzony, jeśli w systemie nie istnieje. Jeśli w trybie odczytu plik nie istnieje, pojawia się błąd.

Program może jednocześnie używać wielu różnych plików, więc trzeba mieć jakiś sposób pozwalający na wskazanie, którego pliku chcemy używać w danej chwili. Służy do tego *wskaźnik pliku*.

Funkcja `fopen` ze standardowej biblioteki pozwala otwierać plik; funkcja ta zwraca niepowtarzalny identyfikator pliku, który jest potem używany do identyfikowania tego pliku. Funkcja ma dwa parametry — łańcuch znakowy określający nazwę pliku oraz drugi łańcuch znakowy wskazujący, w jakim trybie plik ma być otwarty. Funkcja zwraca wskaźnik pliku używany do identyfikowania danego pliku przez inne funkcje biblioteczne.

Jeśli z jakiegoś powodu nie można otworzyć pliku, funkcja zwraca wartość NULL zdefiniowaną w pliku nagłówkowym `<stdio.h>`². Także w tym pliku znajduje się definicja typu `FILE`. Funkcja `fopen` zwraca wartość będącą wskaźnikiem do zmiennej typu `FILE`.

Zbierzmy powyższe uwagi w formie fragmentu gotowego kodu, otwierającego plik *data* w trybie do odczytu:

```
#include <stdio.h>

FILE *inputFile;

inputFile = fopen ("data", "r");
```

Tryb zapisu oznaczamy łańcuchem "w", a tryb dopisywania — łańcuchem "a". Wywołanie funkcji `fopen` zwraca identyfikator otwartego pliku będący wskaźnikiem na typ danych `FILE`. Odpowiednia zmienna wskaźnikowa u nas nazywa się `inputFile`. Następnie sprawdzamy, czy zmienna ta nie jest pusta, czyli czy nie ma wartości NULL:

```
if ( inputFile == NULL )
    printf ("*** nie można otworzyć pliku data.\n");
else
    // odczyt danych z pliku
```

Teraz wiemy już, czy udało się plik otworzyć.

Zawsze trzeba pamiętać o sprawdzeniu wyniku wywołania funkcji `fopen`, gdyż używanie wartości NULL może mieć nieprzewidywalne konsekwencje.

Często otwarcie pliku za pomocą funkcji `fopen`, przypisanie uzyskanego wskaźnika na `FILE` i sprawdzenie, czy wskaźnik nie jest pusty, wykonuje się w jednej instrukcji:

```
if ( (inputFile = fopen ("data", "r")) == NULL )
    printf ("*** nie można otworzyć pliku data.\n");
```

Funkcja `fopen` obsługuje jeszcze trzy inne tryby otwarcia pliku — tryby *aktualizacji* ("r+", "w+" i "a+"). Wszystkie trzy pozwalają czytać dane z pliku i zapisywać je. Tryb "r+" otwiera istniejący plik do czytania i pisania. Tryb "w+" działa jak tryb zapisu (jeśli plik już istniał, jego zawartość jest usuwana; jeśli plik nie istniał, jest tworzony), ale dane można też czytać z pliku. Tryb "a+" otwiera istniejący plik lub tworzy nowy, jeśli dotąd wskazanego pliku nie było. Dane można odczytywać z dowolnego miejsca w pliku, ale dopisywać je wolno tylko na końcu.

W systemach takich jak Windows istnieje rozróżnienie między plikami tekstowymi a binarnymi. W przypadku tych ostatnich do łańcucha trybu trzeba dodać literę *b*. Jeśli o tym zapomnimy, otrzymamy dziwne wyniki, choć program nadal będzie działał. Wynika to stąd, że w niektórych systemach przy zapisie pliku tekstowego pary znaków „powrót karetki” + „nowy wiersz” są zastępowane znakiem nowego wiersza. Poza tym, jeśli wprowadzamy dane do pliku tekstowego, wciśnięcie *Ctrl+Z* powoduje wstawienie znaku końca pliku. Zatem instrukcja:

² „Oficjalnie” wartość NULL jest zdefiniowana w pliku `<stddef.h>`, ale zwykle definiuje się ją także w `<stdio.h>`.

```
inputFile = fopen ("data", "rb");
```

otwiera plik binarny do odczytu.

Funkcje `getc` i `putc`

Funkcja `getc` umożliwia odczyt pojedynczego znaku z pliku. Działa ona identycznie jak opisana wcześniej funkcja `getchar`. Jedyna różnica polega na tym, że funkcja `getc` ma jeden parametr — wskaźnik struktury `FILE` informującej, z jakiego pliku ma być odczytany znak. Jeśli zatem wywołamy `fopen`, tak jak powyżej, to wykonanie instrukcji:

```
c = getc (inputFile);
```

spowoduje odczytanie z pliku *data* jednego znaku. Następne wywołania `getc` pozwolą odczytać dalsze znaki.

Funkcja `getc` zwraca wartość EOF po dojściu do końca pliku; tak samo jak w funkcji `getchar` wartość odczytanego znaku musimy przechowywać w zmiennej typu `int`.

Zgodnie z oczekiwaniami funkcja `putc` jest odpowiednikiem funkcji `putchar` — zapisuje do wskazanego pliku pojedynczy znak. Drugim jej parametrem jest wskaźnik na `FILE`. Wobec tego wywołanie:

```
putc ('\n', outputFile);
```

zapisuje znak nowego wiersza w pliku wskazywanym przez strukturę `FILE` ze zmiennej `outputFile`. Oczywiście wskazany plik musi wcześniej zostać otwarty do zapisu lub do dopisywania (lub w jednym z trybów aktualizacji).

Funkcja `fclose`

Istnieje jeszcze jedna ważna operacja wykonywana na plikach, jest to zamykanie pliku. Funkcja `fclose` w pewnym sensie działa odwrotnie do funkcji `fopen` — informuje system, że nasz program nie będzie już korzystał z pliku. Po zamknięciu pliku system jeszcze „sprząta” strukturę z plikiem związane, zapisując między innymi dane z bufora na nośnik, a w końcu zrywa połączenie między identyfikatorem pliku a samym plikiem. Kiedy plik zostanie zamknięty, nie można z niego czytać ani do niego pisać, póki nie zostanie powtórnie otwarty.

Kiedy kończymy używanie pliku, dobrym zwyczajem jest zamykanie tego pliku. Gdy program normalnie zakończy swoje działanie, sam automatycznie pozamyka wszystkie otwarte pliki. Lepiej jednak robić to zamykanie samemu, na bieżąco. Zaletą takiego rozwiązania jest możliwość ograniczenia liczby otwartych jednocześnie plików — rzecz szczególnie istotna, jeśli program używa wielu plików. Czasami zbyt wiele otwartych plików może powodować problemy z działaniem programu.

Parametrem funkcji `fclose` jest wskaźnik do struktury `FILE` opisującej zamykany plik. Zatem wywołanie:

```
fclose (inputFile);
```

zamknie pliki związane ze wskaźnikiem `inputFile` typu `FILE`.

Mając do dyspozycji funkcje `fopen`, `putc`, `getc` i `fclose`, możemy przystąpić do napisania programu kopiującego pliki. Program 15.5 prosi użytkownika o podanie nazwy kopiowanego pliku i nazwy pliku docelowego. Bazuje on na programie 15.4. Można zajrzeć do tamtego programu, aby porównać kod.

Załóżmy, że w pliku *copyme* zostały wpisane następujące trzy wiersze:

Testujemy teraz kopiowanie pliku programem, który właśnie napisaliśmy, wykorzystując przy tym funkcje `fopen`, `fclose`, `getc` i `putc`.

Program 15.5. Kopiowanie plików

```
// Program kopiujący pliki

#include <stdio.h>

int main (void)
{
    char  inName[64], outName[64];
    FILE  *in, *out;
    int   c;

    // pobranie od użytkownika nazw plików

    printf ("Podaj nazwę kopiowanego pliku: ");
    scanf ("%63s", inName);
    printf ("Podaj nazwę pliku docelowego: ");
    scanf ("%63s", outName);

    // otwieramy plik wejściowy i wynikowy

    if ( (in = fopen (inName, "r")) == NULL ) {
        printf ("Nie mogę otworzyć pliku %s do czytania.\n", inName);
        return 1;
    }

    if ( (out = fopen (outName, "w")) == NULL ) {
        printf ("Nie mogę otworzyć pliku %s do pisania.\n", outName);
        return 2;
    }

    // kopiowanie pliku in na plik out

    while ( (c = getc (in)) != EOF )
        putc (c, out);

    // zamykanie otwartych plików

    fclose (in);
    fclose (out);

    printf ("Plik został skopiowany.\n");

    return 0;
}
```

Program 15.5. Wyniki

Podaj nazwę kopiowanego pliku: **copyme**
Podaj nazwę pliku docelowego: **here**
Plik został skopiowany.

Sprawdźmy teraz, co jest w pliku *here*. Plik ten powinien zawierać te same trzy wiersze, które wpisaliśmy do pliku *copyme*.

Wywołanie funkcji `scanf` na początku powyższego programu wykorzystuje pole stałej szerokości 63, aby zagwarantować, że nie nastąpi przepełnienie tablic znakowych `inName` i `outName`. Następnie program otwiera wskazany plik wejściowy do odczytu i wskazany plik wyjściowy do zapisu. Jeśli plik wyjściowy istnieje i jest otwierany w trybie do zapisu, jego dotychczasowa zawartość jest zamazywana.

Jeżeli któreś wywołanie funkcji `fopen` się powiedzie, program wyświetli stosowny komunikat i nie będzie kontynuował wykonywania programu, a jako kod wyjścia zwróci niezerową wartość. Jeśli oba pliki uda się otworzyć, plik będzie kopiowany znak po znaku za pomocą kolejnych wywołań funkcji `getc` i `putc`, aż do jego końca. W końcu program zamyka oba pliki i zwraca status równy 0.

Funkcja `feof`

Aby sprawdzić, czy doszliśmy do końca danego pliku, używamy funkcji `feof`. Jej jedyny parametr to wskaźnik `FILE`. Funkcja zwraca liczbę całkowitą niezerową, jeśli próbowaliśmy wyjść poza koniec pliku, oraz zero w przeciwnym wypadku. Wobec tego instrukcje:

```
if ( feof ( inFile ) ) {  
    printf ("Koniec danych.\n");  
    return 1;  
}
```

spowodują wyświetlenie na ekranie komunikatu "Koniec danych".

Pamiętajmy, że funkcja `feof` informuje, że ktoś próbował przejść za koniec pliku; a to coś innego niż odczyt ostatniego znaku z pliku. Musimy zatem odczytać ostatni znak, a potem jeszcze próbować przejść dalej — dopiero wtedy nasza funkcja zwróci niezerową wartość.

Funkcje `fprintf` i `fscanf`

Funkcje `fprintf` i `fscanf` działają analogicznie jak funkcje `printf` i `scanf`, ale działają na pliku. Mają dodatkowy pierwszy parametr — wskaźnik `FILE` oznaczający plik, do którego chcemy pisać dane lub z którego mamy zamiar je odczytywać. Aby zatem zapisać napis: "Programowanie w C to niezła zabawa.\n" do pliku wskazywanego przez `outFile`, możemy napisać:

```
fprintf ( outFile, "Programowanie w C to niezła zabawa.\n");
```

Analogicznie, aby odczytać liczbę zmiennoprzecinkową z pliku wskazywanego przez `inFile` do zmiennej `fv`, używamy instrukcji:

```
fscanf (inFile, "%f", &fv);
```

Zarówno `scanf`, jak i `fscanf` zwracają liczbę odczytanych poprawnie elementów lub zwracają `E0F`, jeśli podczas wykonywania konwersji osiągnięto koniec pliku.

Funkcje `fgets` i `fputs`

Podczas czytania i pisania całych wierszy danych do pliku i z niego używamy funkcji `fgets` i `fputs`. Funkcja `fgets` jest wywoływana następująco:

```
fgets (bufor, n, wskPliku);
```

Parametr `bufor` to wskaźnik do tablicy znakowej, w której umieszczane będą odczytane dane. Parametr `n` to liczba całkowita mówiąca, ile maksymalnie znaków mieści się w buforze. W końcu `wskPliku` wskazuje plik, z którego odczytywane są dane.

Funkcja odczytuje znaki z podanego pliku aż do napotkania znaku nowego wiersza (który zostanie umieszczony w buforze) lub aż do odczytania znaku `n-1`. Funkcja automatycznie dostawia znak null po ostatnim znaku bufora. Funkcja zwraca wartość bufora (pierwszy parametr), jeśli odczyt się uda, lub zwraca `NULL`, gdy wystąpi błąd lub nastąpi próba czytania za końcem pliku.

Funkcja `fgets` w połączeniu ze `sscanf` (zobacz dodatek B) pozwala wczytywać dane z poszczególnych wierszy w sposób bardziej elastyczny niż przy użyciu samego `scanf`.

Funkcja `fputs` wpisuje kolejne wiersze znaków do podanego pliku. Funkcję tę wywołuje się następująco:

```
fputs (bufor, wskPliku);
```

Znaki zapisywane w tablicy wskazywanej przez `bufor` są umieszczane w pliku `wskPliku` tak długo, aż zostanie odczytany znak null. Końcowy znak null nie jest umieszczany w pliku.

Istnieją też analogiczne funkcje `gets` i `puts` służące do pisania do terminala i czytania z terminala. Funkcje te opisano w dodatku B.

Wskaźniki `stdin`, `stdout` i `stderr`

Kiedy uruchamiany jest program napisany w języku C, na jego potrzeby automatycznie otwierane są trzy pliki. Pliki te są wskazywane stałymi wskaźnikami `FILE` — `stdin`, `stdout` i `stderr`, zdefiniowanymi w pliku nagłówkowym `<stdio.h>`. Wskaźnik `stdin` typu `FILE` wskazuje standardowe wejście programu, normalnie jest związany z terminalem. Wszystkie standardowe funkcje I/O wczytujące dane, niemające wskaźnika `FILE` jako parametru, korzystają ze `stdin`. Funkcja `scanf` na przykład wczytuje dane ze `stdin`, a wywołanie jej jest równoważne wywołaniu funkcji `fscanf` z pierwszym parametrem równym `stdin`. Zatem wywołanie:

```
fscanf (stdin, "%i", &i);
```


wczyta następną liczbę całkowitą ze standardowego wejścia — czyli zwykle z terminala. Jeśli wejście programu zostało przekierowane do pliku, następną liczbą całkowitą zostanie wczytana z danego pliku.

Jak nietrudno zgadnąć, `stdout` oznacza standardowe wyjście, normalnie także związane z terminalem. Zatem wywołanie:

```
printf ("hej tam!\n");
```

można równoważnie zastąpić wywołaniem funkcji `fprintf` z pierwszym parametrem równym `stdout`:

```
fprintf (stdout, "hej tam!\n");
```

Wskaźnik `stderr` typu `FILE` wskazuje standardowy plik błędów. Tutaj są zapisywane komunikaty błędów generowane przez system; normalnie ten plik także jest związany z terminalem. Powodem istnienia `stderr` jest to, że komunikaty błędów mogą być zapisywane gdzie indziej, nie tam, gdzie normalne komunikaty. Jest to szczególnie przydatne, kiedy wyjście programu jest przekierowane do pliku. Wtedy wyniki działania programu pojawiają się w pliku, ale komunikaty błędów są pokazywane na ekranie. Z tego samego powodu przydatne jest zapisywanie własnych komunikatów błędów w pliku. Na przykład następujące wywołanie funkcji `fprintf`:

```
if ( ( inFile = fopen ("data", "r") ) == NULL )
{
    fprintf (stderr, "Nie mogę otworzyć pliku do odczytu.\n");
    ....
}
```

wysła komunikat błędu do `stderr`, gdy nie można otworzyć pliku `data` do odczytu. Co więcej, jeśli standardowe wyjście zostanie przekierowane do pliku, powyższy komunikat także pojawi się w naszym oknie.

Funkcja `exit`

Czasami, kiedy na przykład program wykryje poważny błąd, chcielibyśmy przerwać jego działanie. Wiemy, że działanie programu kończy się, kiedy wykonana zostanie ostatnia instrukcja w funkcji `main` lub kiedy w funkcji `main` zostanie wykonana instrukcja `return`. Aby jawnie zakończyć działanie programu niezależnie od tego, gdzie w danej chwili jesteśmy, używamy funkcji `exit`. Wywołanie:

```
exit (n);
```

powoduje zakończenie działania programu. Otwarte pliki są automatycznie zamykane, a do systemu operacyjnego zwracany jest *kod wyjścia* `n` mający takie samo znaczenie jak parametr instrukcji `return` użytej w funkcji `main`.

Standardowy plik nagłówkowy `<stdlib.h>` zawiera definicję wartości `EXIT_FAILURE`, używanej do poinformowania o awaryjnym zakończeniu działania programu, oraz definicję wartości `EXIT_SUCCESS`, wskazującą na prawidłowe zakończenie programu.

Jeśli program zakończy swoje działanie wskutek wykonania ostatniej instrukcji w funkcji `main`, kod wyjścia jest nieokreślony. Jeśli kod wyjścia jest niezbędny, nie możemy do tego dopuścić — wtedy zawsze musimy zakończyć działanie za pomocą funkcji `exit` lub instrukcji `return` z podaniem kodu wyjścia.

Oto przykład użycia funkcji `exit`. Poniższa funkcja powoduje zakończenie działania programu z kodem `EXIT_FAILURE`, jeśli podany w jej parametrach plik nie może być otwarty do czytania. Oczywiście zamiast działać tak brutalnie i kończyć program, można by po prostu wyświetlić komunikat o błędzie otwarcia pliku.

```
#include <stdlib.h>
#include <stdio.h>

FILE *openFile (const char *file)
{
    FILE *inFile;

    if ( (inFile = fopen (file, "r")) == NULL ) {
        fprintf (stderr, "Nie mogę otworzyć pliku %s do odczytu.\n", file);
        exit (EXIT_FAILURE);
    }

    return inFile;
}
```

Pamiętajmy, że tak naprawdę nie ma różnicy między wywołaniem funkcji `exit` a użyciem instrukcji `return` w funkcji `main`. W obu przypadkach program kończy swoje działanie, a do systemu zwracany jest kod powrotu. Główna różnica między `exit` a `return` polega na tym, że `return` musi być wywołana z funkcji `main`, a `exit` z dowolnego miejsca. Wywołanie funkcji `exit` kończy działanie programu *natychmiast*, podczas gdy `return` po prostu przekazuje sterowanie w miejsce jej wywołania.

Zmiana nazw i usuwanie plików

Funkcja `rename` z biblioteki standardowej może zostać użyta do zmiany nazwy plików. Ma ona dwa parametry — starą nazwę pliku i nową. Jeśli z jakiegoś powodu zmiana nazwy się nie powiedzie (bo na przykład pierwszy plik nie istnieje albo system nie pozwala nadpisać pliku docelowego), funkcja `rename` zwraca wartość różną od zera. Poniższy fragment kodu:

```
if ( rename ("tempfile", "database") ) {
    fprintf (stderr, "Nie mogę zmienić nazwy pliku tempfile\n");
    exit (EXIT_FAILURE);
}
```

zmienia nazwę pliku *tempfile* na *database* i sprawdza wynik operacji, aby upewnić się, że operacja się udała.

Funkcja `remove` usuwa plik przekazany jej jako parametr. Zwraca wartość niezerową, jeśli usunięcie się nie powiedzie. Kod:

```

if ( remove ("tempfile" ) ) {
    fprintf (stderr, "Nie mogę usunąć pliku tempfile\n");
    exit (EXIT_FAILURE);
}

```

próbuje usunąć plik *tempfile*, a jeśli się to nie powiedzie, pokazuje komunikat błędu i kończy swoje działanie.<<F2-k>>

Przydatne może być użycie funkcji `perror` pokazującej komunikaty błędów funkcji z biblioteki standardowej. Szczegóły podajemy w dodatku B.

Na tym kończymy omawianie funkcji wejścia i wyjścia w języku C. Jak zapowiadaliśmy, z uwagi na ograniczone miejsce nie zostały omówione wszystkie funkcje. Standardowa biblioteka C zawiera mnóstwo funkcji operujących na łańcuchach znakowych, funkcje wejścia i wyjścia o dostępie *swobodnym*, funkcje do obliczeń matematycznych oraz do dynamicznego zarządzania pamięcią. W dodatku B wyliczono wiele funkcji z tej biblioteki.

Ćwiczenia

1. Przepisz i uruchom trzy programy pokazane w tym rozdziale. Uzyskane wyniki porównaj z wynikami pokazanymi w tekście.
2. Wróć do programów tworzonych we wcześniejszych rozdziałach, poeksperymentuj z przekierowywaniem w nich wejścia i wyjścia do plików.
3. Napisz program kopiujący plik do innego, ale jednocześnie zamieniający wszystkie małe litery na ich wielkie odpowiedniki.
4. Napisz program łączący naprzemiennie wiersze z dwóch plików i zapisujący wyniki w `stdout`. Jeśli jeden z plików ma mniej wierszy niż drugi, pozostałe wiersze z większego pliku należy normalnie skopiować.
5. Napisz program wysyłający do pliku `stdout` kolumny m do n z każdego wiersza. Niech program pobiera wartości m i n z terminala.
6. Napisz program pokazujący zawartość pliku na terminalu, po 20 wierszy naraz. Na koniec każdych 20 wierszy program ma czekać na wciśnięcie jakiegoś klawisza. Jeśli będzie to klawisz `q`, program nie powinien pokazywać dalszej części pliku. Każdy inny znak spowoduje pokazanie następnych 20 wierszy.

Skorowidz

A

adresy w pamięci, 261
aktualizacja czasu, 174
algorytm, 68
 Sito Erastotenesa, 122
 sortowania, 145
alokacja pamięci, 37, 363, 459
analiza programu, 27
ANSI, American National Standards
 Institute, 15
ANSI C11, 15
ANSI C99, 52
arytmetyka
 liczb całkowitych, 46
 zespolona, 466
ASCII, 209, 219
assembler, 22
automatyczne zmienne lokalne, 128

B

bajt, 265
biblioteka, 23
bit najmniej znaczący, 265
bitowy operator
 AND, 267
 OR, 269
 OR wyłączającego, 270
blok, 60
błędy, 369
 typowe, 475–479
budowanie programu, 23

C

ciąg Fibonacciego, 108
cytowanie znaków, 208
czas, 173

D

dane wejściowe, 62
data, 167, 171, 321
debugger gdb, 16
definicje
 typów, 303
 złożone, 289
definiowanie
 funkcji, 123, 432
 tablicy, 102
 zmiennej wskaźnikowej, 226
deklarowanie, 408
 prototypu funkcji, 127
 zwracanych typów, 136
dekrementacja, 256
długość łańcucha, 259
dwuznaki, 403
dynamiczna alokacja pamięci, 121, 363, 459
dyrektywa
 #, 443
 #define, 283, 439
 #elif, 300
 #else, 298
 #endif, 298
 #error, 441
 #if, 300, 441
 #ifdef, 298, 442

dyrektywa
 #ifndef, 298, 442
 #include, 296, 442
 #line, 443
 #pragma, 443
 #undef, 301, 443
 include, 327
 dyrektywy preprocesora, 438
 działania
 na strukturach, 426
 na tablicach, 425
 na wskaźnikach, 426
 dzielenie programu, 313

E

edytor vim, 22
 EOF, 342

F

flagi, 278
 funkcji printf, 329
 formatowanie wejścia i wyjścia, 328
 funkcja, 123, 140, 168, 243, 430
 auto_static, 157
 calloc, 364
 copyString, 253, 258
 exit, 349
 fclose, 345
 feof, 347
 fgets, 348
 fopen, 343, 344
 fprintf, 347
 fputs, 348
 free, 367
 fscanf, 347
 gcd, 131
 getc, 345
 getchar, 327
 malloc, 364
 printf, 327, 329, 330
 printMessage, 125
 putc, 345
 putchar, 327

scanf, 335–327
 signum, 86
 funkcje
 definicja, 432
 wskaźniki, 434
 wywołanie, 433
 formatujące dane, 457
 matematyczne, 460
 obsługujące znaki, 451
 obsługujące łańcuchy, 448
 ogólnego przeznaczenia, 468
 przesuwające wartości, 274
 rekurencyjne, 158
 wejścia i wyjścia, 452

G

generowanie
 ciągu Fibonacciego, 108, 119
 liczb, 59
 pierwszych, 109

I

IDE, Integrated Development
 Environment, 23
 identyfikator DEBUG, 370
 identyfikatory, 404
 predefiniowane, 443
 ilustracja operatorów bitowych, 272
 implementacja funkcji
 rotującej, 276
 signum, 86
 informacje o poleceniach gdb, 384
 inicjalizowanie
 struktur, 176
 tablic, 111
 tablic znakowych, 194
 inkrementacja, 256
 instancje, 390
 instrukcja, 20
 break, 72, 434
 continue, 72, 435
 do, 71, 435
 for, 56, 435

- goto, 353, 436
- if, 75, 83, 436
- printf, 60
- return, 129, 437
- switch, 91, 437
- typedef, 306, 416
- while, 67, 438

instrukcje

- puste, 354, 436
- złożone, 434

interpretery, 24

J

jawne przypisanie wartości, 317

język

- C#, 483
- C++, 483
- Objective-C, 484

języki wysokiego poziomu, 20

K

klasa

- C#, 399
- C++, 397
- Objective-C, 392

klasy zmiennych, 430

kod

- pośredni, 22
- znaku, 209

kodowanie ASCII, 88

komentarze, 31, 404

kompilacja warunkowa, 298

kompilator, 26

- gcc, 22, 471, 481
- GNU C, 242

kompilowanie

- programów, 21
- wielu plików, 314

komunikacja między modułami, 316, 320

koniec pliku, 342

konkatenacja, 191

konsolidowanie, 22

konstrukcja

- else if, 85
- if-else, 79

kontrola nad wykonywaniem programu, 379

kontynuowanie łańcuchów, 210

konwersja

- liczb, 114, 153
- liczb ujemnych, 266
- łańcucha na liczbę, 458
- między liczbami, 49
- parametrów, 311
- typów, 303, 309, 429

kwalifikator

- const, 115
- register, 358
- restrict, 359
- volatile, 359

kwalifikatory typu, 358

kwantyfikatory static, 319

L

liczba

- całkowita, 49
- parzysta, 79, 80
- pierwsza, 94, 109
- trójkątna, 55, 127
- ujemna, 266
- zespolona, 466
- zmiennoprzecinkowa, 49

liczby Fibonacciego, 119

licznik, 106

lista

- podwójnie powiązana, 263
- powiązana, 234, 238

literały złożone, 177, 428

Ł

łańcuch

- pusty, 205
- znakowy, 189, 199, 211
- znakowy zmiennej długości, 192

łączenie
 działań, 51
 łańcuchów znakowych, 195, 407
 tablic znakowych, 191

M

macierz, 151
 makro, 210, 291
 DEBUG, 372
 KWADRAT, 292
 zmienna liczba parametrów, 293
 metoda Newtona-Raphsona, 134
 metody, 390
 wyszukiwania haseł, 214
 moduł, 316
 modyfikator
 const, 416
 restrict, 416
 volatile, 416
 modyfikatory
 konwersji, 335
 typu, 329

N

największy wspólny dzielnik, 68, 130
 narzędzia
 programistyczne, 484
 systemu Unix, 324
 narzędzie
 cvs, 324
 make, 322
 nawiasy klamrowe, 118
 nazwy znaków uniwersalnych, 404
 null, 239
 NWD, 68, 161
 NWW, 161

O

obiekt, 389
 obsługa
 dat, 321
 łańcuchów znakowych, 448
 pamięci, 450

plików, 343
 ułamków, 392, 397
 znaków, 451
 odwracanie cyfr liczby, 70
 określanie długości łańcucha, 259
 określnik
 long, 40
 long long, 40
 short, 40
 signed, 40
 unsigned, 40
 OOP, object-oriented programming, 16
 opcje
 programu gcc, 472, 473
 wiersza poleceń, 471
 operacje
 bitowe, 265
 na wskaźnikach, 258
 na znakach, 218
 wejścia i wyjścia, 327
 wejścia i wyjścia na plikach, 339
 operator
 #, 294
 ##, 295
 adresu, 226
 AND, 267
 inkrementacji, 61
 logiczny AND, 81
 logiczny OR, 81
 minus, 46
 modulo, 48, 79
 negacji bitowej, 271
 negacji logicznej, 96
 OR, 269
 OR wyłączającego, 270
 pola struktury, 237
 porównania, 58, 422
 przecinek, 357, 425
 przesunięcia w lewo, 273
 przesunięcia w prawo, 273
 przypisania, 51, 143, 423
 rzutowania typów, 51, 424
 sizeof, 364, 424
 wyboru, 98, 424
 wyłuskania, 227, 231

- operatory
 - arytmetyczne, 421
 - bitowe, 266, 423
 - inkrementacji i dekrementacji, 423
 - języka C, 418
 - logiczne, 422
- optymalizacja programu, 251

P

- parametry, 126, 291
 - wiersza poleceń, 360
- pętla, 55
 - for
 - deklarowanie zmiennych, 66
 - pomijanie składników, 66
 - wiele wyrażeń, 66
 - zagnieżdżona, 64
 - while, 285
- pierwszy program, 25
- plik nagłówkowy, 320, 445
 - <complex.h>, 466
 - <ctype.h>, 302
 - <float.h>, 447
 - <limits.h>, 446
 - <stdbool.h>, 313, 447
 - <stddef.h>, 445
 - <stdint.h>, 447
 - <stdio.h>, 298, 313
- pliki
 - exe, 23
 - koniec, 342
 - przekierowanie wejścia-wyjścia, 339
 - usuwanie, 350
 - włączane, 285, 298
 - zmiana nazw, 350
 - źródłowe, 379
- podjmowanie decyzji, 75
- podstawowe typy danych, 42
- pojedyncze znaki, 38
- pokazywanie
 - plików źródłowych, 379
 - tablic znakowych, 194
- poła bitowe, 278, 281

- polecenie, 471
 - gdb, 375, 387, 380, 384
 - step, 380
- porównywanie łańcuchów znakowych, 197
- postać polecenia, 471
- preprocesor, 283, 300, 438
- procedura printf, 27
- program, 19
 - dzielenie, 313
 - przenośność, 288
 - rozszerzalność, 287
- program uruchomieniowy, 16, 375
- programowanie
 - obiektywne, 389, 483
 - z góry na dół, 139
- prototyp funkcji, 127
- przechodzenie po liście, 239
- przecinek, 357
- przekierowanie, 225
 - wejścia-wyjścia do pliku, 339
- przenośność programu, 288
- przeszukiwanie słownika, 216
- punkt przzerwania, 379, 383
 - usuwanie, 383
 - wstawianie, 379
 - wyliczanie, 383
- pusty wskaźnik, 239

R

- rekurencja, 158
- rok przestępny, 82
- rotowanie bitów, 275
- rozszerzalność programu, 287
- rzutowanie typów, 51

S

- silnia, 158
- sito Eratostenesa, 122
- słownik, 212, 216
- słowo kluczowe, 404
 - const, 241, 416
 - extern, 319
 - static, 156, 319

sortowanie tablic, 145
 sprawdzanie parametrów funkcji, 138
 stałe, 35

- całkowitoliczbowe, 405
- łańcuchy znakowe, 254, 407
- szerokie znaki, 407
- wyliczeniowe, 407
- zmiennoprzecinkowe, 405
- znakowe, 406

 standardowa biblioteka C, 445
 standardowe pliki nagłówkowe, 445
 struktura, 163, 211, 412, 426

- na czas, 173
- na daty, 164
- packed_struct, 279

 struktury zawierające

- inne struktury, 181
- tablice, 182
- wskaźniki, 233, 234

 sumowanie elementów, 252
 system operacyjny, 20

Ś

śląd stosu, 383

T

tablica, 101, 140, 211, 425

- jako licznik, 106
- jednowymiarowa, 410
- liczb pierwszych, 97
- liczb trójkątnych, 59
- o zmiennej długości, 150, 411
- o zmiennej wielkości, 119
- struktur, 178
- wielowymiarowa, 117, 147, 150, 411
- znakowa, 112, 190, 194

 technika iteracyjna Newtona-Raphsona, 133
 terminal, 87
 trójkątniki, 209, 438
 tryby otwarcia pliku, 344
 typ danych, 35, 42, 303

- _Bool, 38
- _Complex, 52

_Imaginary, 52
 char, 38
 double, 37
 float, 37
 int, 36
 typy

- argumentów, 136
- danych
 - pochodne, 410
 - podstawowe, 408
 - wyliczeniowe, 305

U

ułamki, 392, 399
 unia, 355, 414
 uruchamianie programu, 26
 ustalanie daty, 171
 ustawianie

- tablic, 384
- zmiennych, 384

 usuwanie

- błędów, 369, 371
 - przy użyciu programu gdb, 375
 - za pomocą preprocesora, 369
- plików, 350
- punktów przerwania, 383

 uzyskiwanie śladu stosu, 383
 użycie

- dyrektywy #include, 297
- formatów funkcji printf, 330
- list powiązanych, 236
- plików nagłówkowych, 320
- struktur, 166
- struktur zawierających wskaźniki, 233
- tablic, 106, 113
- tablic struktur, 180
- typów danych, 38
- unii, 355
- wskaźników do zamiany wartości, 244
- wskaźników i funkcji, 243
- wskaźników na struktury, 231
- wskaźników na tablice, 250
- wyliczeniowych typów danych, 305
- zmiennych, 42, 377

W

wartość
 NULL, 344
 wyrażenia, 129
 wczytanie pojedynczego znaku, 201
 wersje struktur, 185
 wielokrotne włączanie plików, 300
 wiersz
 poleceń, 314, 360
 wynikowy, 28
 włączanie plików nagłówkowych, 300
 wprowadzanie łańcuchów znakowych, 199
 wskaźnik, 225, 230, 261, 415, 426
 elementu tablicy, 248
 pusty, 285
 stderr, 348
 stdin, 348
 stdout, 348
 wskaźniki
 funkcji, 434
 na funkcje, 260
 na łańcuchy znakowe, 253
 na struktury, 231, 428
 na tablice, 250, 427
 w wyrażeniach, 229
 wstawienie elementu na listę, 238
 wyliczanie
 pierwiastka kwadratowego, 134
 punktów przerwania, 383
 średniej, 77
 wartości bezwzględnej, 132
 wyliczeniowe typy danych, 303, 416
 wyrażenia, 90, 417
 arytmetyczne, 35, 44
 stałe, 420
 wyrażenie, 129
 wyrównywanie wyników, 62
 wyszukiwanie binarne, 215
 wyświetlanie wartości zmiennych, 29
 wywoływanie funkcji, 125, 384, 433

Z

zagnieżdżone
 instrukcje if, 83
 pętle for, 64
 zakres, 430
 wartości, 37
 zamiana
 łańcucha znakowego, 220
 podstawy liczb, 113
 zastosowanie tablic, 109
 zintegrowane środowisko programistyczne,
 IDE, 23, 482
 zliczanie
 słów, 203, 206
 znaków łańcucha, 193
 złożone warunki porównania, 81
 zmiennie, 35, 377, 431
 automatyczne, 155
 globalne, 152
 logiczne, 94
 lokalne, 126
 statyczne, 155
 wskaźnikowe, 226
 zewnętrzne, 316
 znajdowanie wartości minimalnej, 140
 znak
 ampersand, 260
 Escape, 209
 gwiazdki, 44
 minus, 44
 nowego wiersza, 27
 odwrotnego ukośnika, 27
 plus, 44
 ukośnika, 44
 wartości, 310
 znaki
 cytowane, 208
 konwersji, 330, 336
 specjalne, 406
 wielobajtowe, 407
 zwracanie
 przez funkcję wskaźnika, 245
 wyników funkcji, 129

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ 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>



Wykorzystaj potencjał języka C!

Twórcą języka programowania C jest Dennis Ritchie. Opracował go w laboratoriach AT&T Bell na początku lat 70. XX wieku. Musiała jednak upłynąć jeszcze niemal dekada, by język ten zyskał popularność i powszechne uznanie — dopiero w 1990 roku opublikowano pierwszą oficjalną wersję standardu ANSI C. Później przez długi czas język C dominował na rynku i do tej pory nie ma sobie równych w wielu dziedzinach programowania.

Kolejne wydanie kultowej książki o języku C zostało zaktualizowane i poprawione. Znajdziesz tu informacje o najnowszych dodatkach, wprowadzonych w standardzie ANSI C11, a także sprawdzone, konkretne wiadomości na temat składni języka i najlepszych praktyk tworzenia programów z wykorzystaniem potencjału języka C. Pętle, instrukcje warunkowe, struktury, wskaźniki, operacje bitowe oraz polecenia preprocesora to klasyczne zagadnienia — bez ich znajomości żaden programista sobie nie poradzi. Jeżeli szukasz kompletnego podręcznika zawierającego najbardziej aktualne informacje na temat języka C, trzymasz w rękach idealną książkę!

Dzięki tej książce:

- napiszesz, skompilujesz i uruchomisz swój pierwszy program
- poznasz typy danych oraz ich cechy charakterystyczne
- opanujesz składnię oraz typowe konstrukcje języka C
- zaznajomisz się z tablicami oraz ze strukturami
- poskromisz wskaźniki oraz ulepszysz zarządzanie pamięcią
- opanujesz język C

Stephen G. Kochan — autor i współautor klasycznych książek o programowaniu oraz o systemie Unix. Był konsultant ds. oprogramowania w AT&T Bell Laboratories. Do jego obowiązków należało m.in. przygotowywanie i prowadzenie kursów z systemu Unix oraz języka C. Jest autorytetem w swojej specjalizacji.

Helion

36273 numer katalogowy

kolegarnia internetowa

<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Informatyka w najlepszym wydaniu

Sprawdź najnowsze promocje:

- <http://helion.pl/promocje>
 - <http://helion.pl/bestsellery>
- Zamów informacje o nowościach:
• <http://helion.pl/nowości>

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

sięgnij po WIĘCEJ



KOD KORZYŚCI

ISBN 978-83-283-1645-4



9 788328 316454

cena: 79,00 zł


Addison
Wesley