Wydawnictwa Naukowo-Techniczne

KLASYKA INFORMATYKI

Brian W. Kernighan
Dennis M. Ritchie

Język ANSI C

„Język C jest językiem ogólnego stosowania ... Jednak C nie jest przywiązany do żadnego systemu operacyjnego lub maszyny. Wprawdzie nazwano go »językiem programowania systemowego«, jest bowiem wygodnym narzędziem do konstruowania kompilatorów systemów operacyjnych, ale okazało się, że nadawał się równie dobrze do napisania ważniejszych programów z wielu różnych dziedzin".

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

Rosnąca popularność języka C i zmiany powstałe podczas wielu lat jego użytkowania spowodowały konieczność stworzenia bardziej precyzyjnej jednoznacznej definicji tego języka. W roku 1983 Amerykański Narodowy Instytut Standaryzacji (ANSI) powołał komitet, którego zadaniem było sformułowanie takiej definicji. W efekcie powstał standard ANSI języka C.

W niniejszej książce opisano język C tak, jak go zdefiniowano w standardzie ANSI – stąd tytuł polskiego przekładu „Język ANSI C".

Cena 39,- zł
ISBN 83-204-2620-0

Język ANSI C

Wydawnictwa
Naukowo-
Techniczne
Warszawa

KLASYKA INFORMATYKI

W skład serii „Klasyka Informatyki" wchodzą dzieła najwybitniejszych uczonych świata w dziedzinie informatyki – książki o nieprzemijającej wartości, stanowiące bazę solidnego, klasycznego wykształcenia każdego profesjonalnego informatyka.

Wydawnictwa Naukowo-Techniczne przygotowały tę serię ze szczególną pieczołowitością, powierzając tłumaczenie poszczególnych tomów znakomitym specjalistom. Wyboru książek dokonano w ścisłej współpracy z polskim środowiskiem akademickim, dedykując serię głównie studentom informatyki i młodym pracownikom naukowym.

Brian W. Kernighan
Dennis M. Ritchie

Język ANSI C

Wydanie piąte

Z angielskiego przełożyli
Danuta i Marek Kruszewscy

O Autorach:

Brian W. Kernighan kieruje Działem Badań Struktur Komputerowych w Bell Laboratories w Murray Hill, w New Jersey. Ma stopień bakałarza z fizyki stosowanej (University of Toronto 1964 r.) i stopień doktora z elektrotechniki (Princeton University 1969 r.). Współpracę z Bell Labs rozpoczął w 1969 roku. Zajmował się optymalizacją kombinatoryczną, systemami opracowywania dokumentacji, językami programowania i programami narzędziowymi. Obecnie prowadzi badania nad językami programowania zorientowanymi użytkowo, metodyką programowania i interfejsami użytkownika.

Dennis M. Ritchie kieruje Działem Badań Oprogramowania Systemowego w Bell Laboratories w Murray Hill, w New Jersey. Ma stopień bakałarza, który uzyskał na Harvard University. W 1968 roku napisał pracę doktorską. Współpracę z Bell Labs rozpoczął w 1967 roku. Zajmował się systemami Multics i Unix. Obecnie prowadzi badania nad rozproszonymi systemami operacyjnymi, językami oprogramowania i sprzętem służącym do trasowania i przełączania.

Dane o oryginale

The C Programming Language

Second Edition

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE
AT&T Bell Laboratories, Murray Hill, New Jersey

Translation copyright © 1994
by Wydawnictwa Naukowo-Techniczne

The C Programming Language
Second Edition

Copyright © 1988
All Rights Reserved
Published by arrangement with
the original publisher, Prentice-Hall, Inc.
a division of Simon & Schuster

UNIX jest zarejestrowanym znakiem towarowym AT&T

Redaktor *Ewa Zdanowicz*

Okładkę i strony tytułowe projektował *Wojciech J. Steifer*
Opracowanie techniczne *Ewa Eckhardt, Marta Jeczeń*
Przygotowanie do druku *Marianna Szwagrzak, Marianna Zadrożna*

© Copyright for the Polish edition by Wydawnictwa Naukowo-Techniczne
Warszawa 1994, 1998, 1999, 2000

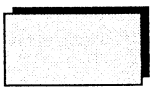
All Rights Reserved
Printed in Poland

Utwór w całości ani we fragmentach nie może być powielany ani rozpowszechniany za pomocą urządzeń elektronicznych, mechanicznych, kopiujących, nagrywających i innych, w tym również nie może być umieszczany ani rozpowszechniany w postaci cyfrowej zarówno w Internecie, jak i w sieciach lokalnych bez pisemnej zgody posiadacza praw autorskich.

Adres poczty elektronicznej: wnt@pol.pl
Strona WWW: www.wnt.com.pl

ISBN 83-204-2620-0

WNT Warszawa 2000
Wyd. V.
Ark. wyd. 21,1. Ark. druk. 22,5
Symbol Et/83594/WNT
Druk i oprawa: Zakład Poligraficzno-Wydawniczy POZKAL



SPIS TREŚCI

PRZEDMOWA 13

PRZEDMOWA DO PIERWSZEGO WYDANIA KSIĄŻKI 15

WPROWADZENIE 17

1 ELEMENTARZ 23

1.1 Zaczynamy ... 23
1.2 Zmienne i wyrażenia arytmetyczne ... 27
1.3 Instrukcja for ... 33
1.4 Stałe symboliczne ... 34
1.5 Wejście i wyjście znakowe ... 35
1.5.1 Kopiowanie plików ... 36
1.5.2 Zliczanie znaków ... 39
1.5.3 Zliczanie wierszy ... 40
1.5.4 Zliczanie słów ... 42
1.6 Tablice ... 44
1.7 Funkcje ... 47
1.8 Argumenty – przekazywanie przez wartość ... 51
1.9 Tablice znakowe ... 52
1.10 Zmienne zewnętrzne i zasięg zmiennych ... 55

2 TYPY, OPERATORY I WYRAŻENIA 60

2.1 Nazwy zmiennych ... 60
2.2 Typy i rozmiary danych ... 61
2.3 Stałe ... 62
2.4 Deklaracje ... 66
2.5 Operatory arytmetyczne ... 67
2.6 Relacje i operatory logiczne ... 68
2.7 Przekształcenia typów ... 69
2.8 Operatory zwiększania i zmniejszania ... 74
2.9 Operatory bitowe ... 76

2.10 Operatory i wyrażenia przypisania ... 78
2.11 Wyrażenia warunkowe ... 80
2.12 Priorytety i kolejność obliczeń ... 81

3 STEROWANIE 84

3.1 Instrukcje i bloki ... 84
3.2 Instrukcja if-else ... 84
3.3 Konstrukcja else-if ... 86
3.4 Instrukcja switch ... 88
3.5 Pętle while i for ... 90
3.6 Pętla do-while ... 94
3.7 Instrukcje break i continue ... 96
3.8 Instrukcja goto i etykiety ... 97

4 FUNKCJE I STRUKTURA PROGRAMU 99

4.1 Wprowadzenie ... 100
4.2 Funkcje zwracające wartości niecałkowite ... 104
4.3 Zmienne zewnętrzne ... 107
4.4 Zasięg nazw ... 114
4.5 Pliki nagłówkowe ... 117
4.6 Zmienne statyczne ... 118
4.7 Zmienne rejestrowe ... 119
4.8 Struktura blokowa ... 120
4.9 Inicjowanie ... 121
4.10 Rekurencja ... 122
4.11 Preprocesor języka C ... 125
4.11.1 Wstawianie plików ... 125
4.11.2 Makrorozwinięcie ... 126
4.11.3 Kompilacja warunkowa ... 128

5 WSKAŹNIKI I TABLICE 130

5.1 Wskaźniki i adresy ... 130
5.2 Wskaźniki i argumenty funkcji ... 132
5.3 Wskaźniki i tablice ... 136
5.4 Arytmetyka na adresach ... 139
5.5 Wskaźniki znakowe i funkcje ... 143
5.6 Tablice wskaźników; wskaźniki do wskaźników ... 148
5.7 Tablice wielowymiarowe ... 152
5.8 Inicjowanie tablic wskaźników ... 155
5.9 Wskaźniki a tablice wielowymiarowe ... 155
5.10 Argumenty wywołania programu ... 157
5.11 Wskaźniki do funkcji ... 162
5.12 Skomplikowane deklaracje ... 166

6 STRUKTURY 173

6.1 Podstawowe wiadomości o strukturach ... 173
6.2 Struktury i funkcje ... 176
6.3 Tablice struktur ... 180
6.4 Wskaźniki do struktur ... 184
6.5 Struktury odwołujące się do samych siebie ... 187
6.6 Przeglądanie tablic ... 193
6.7 Deklaracja typedef ... 196
6.8 Unie ... 198
6.9 Pola bitowe ... 200

7 WEJŚCIE I WYJŚCIE 203

7.1 Standardowe wejście i wyjście ... 203
7.2 Formatowane wyjście – funkcja printf ... 206
7.3 Zmienna długość list argumentów ... 208
7.4 Formatowane wejście – funkcja scanf ... 210
7.5 Obsługa plików ... 214
7.6 Obsługa błędów – plik stderr i funkcja exit ... 217
7.7 Wprowadzanie i wyprowadzanie wierszy tekstu ... 219
7.8 Kilka użytecznych funkcji ... 221
7.8.1 Operacje na tekstach ... 221
7.8.2 Badanie klasy znaków i ich przekształcenia ... 222
7.8.3 Funkcja ungetc ... 222
7.8.4 Wykonanie polecenia ... 222
7.8.5 Zarządzanie pamięcią ... 223
7.8.6 Funkcje matematyczne ... 224
7.8.7 Generowanie liczb losowych ... 224

8 ŚRODOWISKO SYSTEMU UNIX 225

8.1 Deskryptory plików ... 225
8.2 Wejście i wyjście niskiego poziomu – funkcje read i write ... 226
8.3 Funkcje open, creat, close, unlink ... 229
8.4 Dostęp swobodny – funkcja lseek ... 232
8.5 Przykład – realizacja funkcji fopen i getc ... 233
8.6 Przykład – wypisywanie zawartości skorowidzów ... 237
8.7 Przykład – dystrybutor pamięci ... 244

DODATEK A: PRZEWODNIK JĘZYKA C 251

A1 Wstęp ... 251
A2 Konwencje leksykalne ... 251
A2.1 Jednostki leksykalne ... 252
A2.2 Komentarze ... 252
A2.3 Identyfikatory (nazwy) ... 252

A2.4 Słowa kluczowe ... 252
A2.5 Stałe ... 253
A2.5.1 Stałe całkowite ... 253
A2.5.2 Stałe znakowe ... 254
A2.5.3 Stałe zmiennopozycyjne ... 255
A2.5.4 Stałe wyliczeń ... 255
A2.6 Napisy ... 255
A3 Notacja opisu składni ... 255
A4 Znaczenie identyfikatorów ... 256
A4.1 Klasy pamięci ... 256
A4.2 Typy podstawowe ... 257
A4.3 Typy pochodne ... 258
A4.4 Kwalifikatory typów ... 258
A5 Obiekty i l-wartości ... 259
A6 Przekształcenia typów ... 259
A6.1 Promocja typu całkowitego ... 259
A6.2 Przekształcenia całkowite ... 259
A6.3 Wartości całkowite i zmiennopozycyjne ... 260
A6.4 Typy zmiennopozycyjne ... 260
A6.5 Przekształcenia arytmetyczne ... 260
A6.6 Wskaźniki i wartości całkowite ... 261
A6.7 Typ void ... 262
A6.8 Wskaźniki do typu void ... 262
A7 Wyrażenia ... 263
A7.1 Generowanie wskaźników ... 264
A7.2 Wyrażenia proste ... 264
A7.3 Wyrażenia przyrostkowe ... 265
A7.3.1 Odwołania do tablic ... 265
A7.3.2 Wywołania funkcji ... 265
A7.3.3 Odwołania do struktur ... 267
A7.3.4 Zwiększanie i zmniejszanie przyrostkowe ... 267
A7.4 Operatory jednoargumentowe ... 268
A7.4.1 Przedrostkowe operatory zwiększania i zmniejszania ... 268
A7.4.2 Operator adresu ... 268
A7.4.3 Operator adresowania pośredniego ... 268
A7.4.4 Jednoargumentowy plus ... 269
A7.4.5 Jednoargumentowy minus ... 269
A7.4.6 Operator dopełnienia jedynkowego ... 269
A7.4.7 Operator negacji logicznej ... 269
A7.4.8 Operator sizeof ... 269
A7.5 Rzutowanie ... 270
A7.6 Operatory multyplikatywne ... 270
A7.7 Operatory addytywne ... 271
A7.8 Operatory przesunięcia ... 272
A7.9 Operatory relacji ... 272
A7.10 Operatory przyrównania ... 273
A7.11 Bitowy operator koniunkcji (AND) ... 273
A7.12 Bitowy operator różnicy symetrycznej (XOR) ... 273
A7.13 Bitowy operator alternatywy (OR) ... 274
A7.14 Operator iloczynu logicznego (logiczne AND) ... 274

A7.15 Operator sumy logicznej (logiczne OR) ... 274
A7.16 Operator warunkowy ... 275
A7.17 Wyrażenia przypisania ... 275
A7.18 Operator przecinkowy ... 276
A7.19 Wyrażenia stałe ... 276
A8 Deklaracje ... 277
A8.1 Specyfikatory klasy pamięci ... 278
A8.2 Specyfikatory typu ... 279
A8.3 Deklaracje struktur i unii ... 280
A8.4 Wyliczenia ... 284
A8.5 Deklaratory ... 285
A8.6 Znaczenie deklaratorów ... 286
A8.6.1 Deklaratory wskaźników ... 286
A8.6.2 Deklaratory tablic ... 287
A8.6.3 Deklaratory funkcji ... 288
A8.7 Inicjowanie ... 290
A8.8 Nazwy typów ... 293
A8.9 Nazwy typedef ... 294
A8.10 Równoważność typów ... 295
A9 Instrukcje ... 295
A9.1 Instrukcje etykietowane ... 295
A9.2 Instrukcja wyrażeniowa ... 296
A9.3 Instrukcja złożona (blok) ... 296
A9.4 Instrukcje wyboru ... 297
A9.5 Instrukcje powtarzania (pętle) ... 298
A9.6 Instrukcje skoku ... 299
A10 Deklaracje zewnętrzne ... 299
A10.1 Definicje funkcji ... 300
A10.2 Deklaracje zewnętrzne ... 302
A11 Zasięg i łączność nazw ... 303
A11.1 Zasięg leksykalny ... 303
A11.2 Łączność nazw ... 304
A12 Preprocesor ... 305
A12.1 Sekwencje trzyznakowe ... 305
A12.2 Sklejanie wierszy ... 306
A12.3 Definicje i rozwinięcia makr ... 306
A12.4 Włączanie plików ... 309
A12.5 Kompilacja warunkowa ... 309
A12.6 Numeracja wierszy ... 311
A12.7 Generowanie błędów ... 311
A12.8 Instrukcja pragma ... 311
A12.9 Pusta instrukcja preprocesora ... 312
A12.10 Nazwy zdefiniowane w preprocesorze ... 312
A13 Gramatyka ... 312

DODATEK B: BIBLIOTEKA STANDARDOWA 321

B1 Wejście i wyjście: nagłówek <stdio.h> ... 322
B1.1 Operacje na plikach ... 322

B1.2 Formatowane wyjście ... 324
B1.3 Formatowane wejście ... 326
B1.4 Funkcje realizujące wejście i wyjście znakowe ... 328
B1.5 Funkcje realizujące bezpośrednie wejście i wyjście ... 330
B1.6 Funkcje wyznaczające pozycję w pliku ... 330
B1.7 Obsługa błędów ... 331
B2 Klasyfikowanie znaków: nagłówek <ctype.h> ... 331
B3 Operacje na tekstach: nagłówek <string.h> ... 332
B4 Funkcje matematyczne: nagłówek <math.h> ... 334
B5 Funkcje narzędziowe: nagłówek <stdlib.h> ... 335
B6 Diagnostyka: nagłówek <assert.h> ... 338
B7 Zmienne listy argumentów: nagłówek <stdarg.h> ... 339
B8 Skoki odległe: nagłówek <setjmp.h> ... 340
B9 Sygnały: nagłówek <signal.h> ... 340
B10 Obsługa daty i czasu: nagłówek <time.h> ... 341
B11 Ograniczenia implementacji: nagłówki <limits.h> i <float.h> ... 343

DODATEK C: PODSUMOWANIE ZMIAN 345

SKOROWIDZ 349

PRZEDMOWA

Od czasu opublikowania w 1978 r. pierwszego* wydania książki *Język C* świat komputerowy przeżył rewolucję. Duże komputery stały się jeszcze większe, a komputery osobiste mają dziś możliwości porównywalne z dużymi systemami obliczeniowymi sprzed dekady. Przez ten czas język C również się zmienił, choć skromnie, i rozpowszechnił daleko poza swoje oryginalne zastosowanie jako język programowania w systemie operacyjnym Unix.

Rosnąca popularność języka C, zmiany powstałe podczas wielu lat jego używania, tworzenie kompilatorów przez grupy ludzi nie uwikłanych w jego projektowanie – wszystko to razem spowodowało konieczność stworzenia bardziej precyzyjnej i nowoczesnej definicji języka niż przedstawiona w pierwszym wydaniu tej książki. W roku 1983 Amerykański Narodowy Instytut Standaryzacji (ANSI) powołał komitet, którego zadaniem było sformułowanie „jednoznacznej i maszynowo niezależnej definicji języka C" przy jednoczesnym zachowaniu jego „ducha". W efekcie powstał standard ANSI języka C.

Standard definiuje konstrukcje, o których wspomniano, ale ich nie opisano w pierwszym wydaniu, w szczególności przypisania struktur i wyliczenia. Wprowadza nową postać deklaracji funkcji, która umożliwia wzajemną kontrolę definicji funkcji i jej wywołania. Zawiera specyfikacje funkcji tworzących standardową bibliotekę języka – jest to obszerny zbiór funkcji realizujących operacje wejścia i wyjścia, zarządzanie pamięcią, obsługę tekstów i inne. Standard precyzuje interpretację tych właściwości języka, które nie dość dokładnie opisano w wersji oryginalnej, i jednocześnie ustala, jakie aspekty języka pozostają zależne od maszyny.

W tym drugim** wydaniu książki *Język C* opisujemy C tak, jak go zdefiniowano w standardzie ANSI. Chociaż wynotowaliśmy miejsca, w których język się rozwinął,

---

* Polski przekład tego (pierwszego) wydania ukazał się dwukrotnie w latach 1987 – 1988 i był zatytułowany *Język C. – Przyp. tłum.*

** Niniejszy przekład tego (drugiego) wydania ukazuje się na polskim rynku pod zmienionym tytułem *Język ANSI C* i zastępuje wydaną poprzednio książkę *Język C. – Przyp. red.*

postanowiliśmy jednak programy pisać wyłącznie w nowej postaci. W większej części książki nie sprawia to istotnej różnicy; najbardziej widoczną zmianą jest nowa postać deklaracji i definicji funkcji. Nowoczesne kompilatory realizują już większość możliwości standardu.

Staraliśmy się zachować zwięzłość pierwszego wydania. C nie jest dużym językiem i duża książka źle by mu służyła. Wzbogaciliśmy komentarz dotyczący krytycznych konstrukcji języka, takich jak wskaźniki, które są podstawą dla programowania w C. Wysubtelniliśmy oryginalne przykłady i w wielu rozdziałach dodaliśmy nowe. Dla przykładu opis skomplikowanych deklaracji rozbudowano, dodając programy przekształcające deklaracje na opis słowny i vice versa. Jak poprzednio, wszystkie przykłady zostały sprawdzone bezpośrednio z tekstu programów, który ma postać czytelną dla maszyny.

Dodatek A, czyli przewodnik języka C, nie jest standardem, a jedynie naszą próbą wyrażenia istoty standardu „na małej przestrzeni". Z założenia ma on być łatwy do przyswojenia przez programistów, a nie stanowić definicję języka dla twórców kompilatorów – ta rola słusznie należy się samemu standardowi. Dodatek B jest podsumowaniem udogodnień, jakich dostarcza biblioteka standardowa. I tu z założenia odwołujemy się do programistów, a nie do instalatorów. Dodatek C jest zwięzłym podsumowaniem zmian w stosunku do wersji oryginalnej.

W przedmowie do pierwszego wydania napisaliśmy, że „umiejętność posługiwania się językiem C wzrasta wraz z doświadczeniem jego użytkownika". Z nowymi dziesięcioletnimi doświadczeniami ciągle jesteśmy tego zdania. Mamy nadzieję, że ta książka pomoże Ci nauczyć się języka C i poprawnie go stosować.

Zaciągnęliśmy dług wdzięczności u przyjaciół, którzy pomogli nam przygotować to drugie wydanie. Jon Bentley, Doug Gwyn, Doug McIlroy, Peter Nelson i Rob Pike umieścili rzeczowe komentarze na prawie każdej stronie rękopisu. Za uważne przeczytanie tekstu naszą wdzięczność zaskarbili sobie Al Aho, Dennis Allison, Joe Campbell, G. R. Emlin, Karen Fortgang, Allen Holub, Andrew Hume, Dave Kristol, John Linderman, Dave Prosser, Gene Spafford i Chris Van Wyk. Wiele pomocnych sugestii dali nam: Bill Cheswick, Mark Kernighan, Andy Koenig, Robin Lake, Tom London, Jim Reeds, Clovis Tondo i Peter Weinberger. Dave Prosser odpowiedział na wiele szczegółowych pytań dotyczących standardu ANSI. Do lokalnego testowania naszych programów intensywnie używaliśmy kompilatora języka C++, którego autorem jest Bjarne Stroustrup, a testowanie końcowe za pomocą kompilatora ANSI C umożliwił nam Dave Kristol. Rich Drechsler był wielce pomocny przy tworzeniu maszynopisu.

Wszystkim serdecznie dziękujemy

Brian W. Kernighan
Dennis M. Ritchie

PRZEDMOWA DO PIERWSZEGO WYDANIA KSIĄŻKI*

Język C jest językiem ogólnego stosowania. Charakteryzuje się prostotą wyrażeń, nowoczesnym sterowaniem, nowoczesnymi strukturami danych oraz bogatym zestawem operatorów. C nie jest językiem „bardzo wysokiego poziomu", nie jest nawet „duży", i nie został opracowany dla jakiejś szczególnej dziedziny zastosowań. Brak ograniczeń oraz ogólność powodują, że w wielu przypadkach jest wygodniejszy i bardziej sprawny od innych języków programowania o pozornie większych możliwościach. Język C opracował i zrealizował Dennis Ritchie dla systemu operacyjnego Unix działającego na minikomputerze DEC PDP-11. System operacyjny, kompilator C i niemal wszystkie programy usługowe systemu Unix (razem z całym oprogramowaniem potrzebnym do przygotowania oryginału tej książki) są napisane w C. Kompilatory języka C tworzy się także dla innych maszyn, łącznie z IBM/370, Honeywell 6000 i Interdata 8/32. Jednak C nie jest związany z żadnym konkretnym systemem czy sprzętem i łatwo w nim pisać programy, które można uruchamiać bez zmian na dowolnej maszynie, na której język C jest zrealizowany.

Niniejsza książka jest tak pomyślana, aby pomóc Czytelnikowi w nauce programowania w C. Elementarne wprowadzenie pozwala nowym użytkownikom rozpocząć programowanie na tyle szybko, na ile jest to możliwe. Każdą charakterystyczną cechę języka omówiono w oddzielnym rozdziale, a podsumowanie wiedzy o języku C zawarto w końcowym przewodniku. Większość materiału jest oparta na czytaniu, pisaniu i korekcie przykładów, a nie jedynie na wyliczaniu reguł. Prawie wszędzie przykłady stanowią prawdziwe, kompletne programy, tylko nieliczne z nich są oderwanymi fragmentami programów. Wszystkie przykłady zostały sprawdzone bezpośrednio z tekstu, który ma postać czytelną dla maszyny. Poza pokazaniem, jak można sprawnie używać języka, próbowaliśmy także tam, gdzie było to możliwe, przedstawiać pożyteczne algorytmy oraz zasady dobrego stylu programowania i prawidłowego projektowania.

Książka nie jest podręcznikiem „wstępu do programowania"; zakłada pewną znajomość podstawowych elementów, takich jak zmienne, instrukcje przypisania, pętle

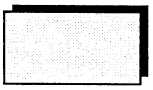
* Patrz przypis na str. 13.

czy funkcje. Niemniej jednak programiści-nowicjusze powinni dać sobie radę z czytaniem i uczeniem się języka, chociaż pomoc bardziej doświadczonych kolegów może się przydać.

Nasze doświadczenia udowodniły, że C jest przyjemnym, wyrazistym i uniwersalnym językiem dla szerokiej gamy programów. Jest łatwy do nauczenia, a umiejętność posługiwania się nim wzrasta wraz z doświadczeniem jego użytkownika. Mamy nadzieję, że ta książka pomoże Ci w poprawnym jego stosowaniu.

Konstruktywna krytyka i sugestie wielu przyjaciół i kolegów wielce wpłynęły na postać tej książki i na naszą przyjemność jej pisania. Szczególnie Mike Bianchi, Jim Blue, Stu Feldman, Doug McIlroy, Bill Roome, Bob Rosin i Larry Rosler – wszyscy czytali z uwagą wiele jej wersji. Naszą wdzięczność zaskarbili sobie Al Aho, Steve Bourne, Dan Dvorak, Chuck Haley, Debbie Haley, Marion Harris, Rick Holt, Steve Johnson, John Mashey, Bob Mitze, Ralph Muha, Peter Nelson, Elliot Pinson, Bill Plauger, Jerry Spivack, Ken Thompson i Peter Weinberger za pomocne komentarze na różnych etapach pracy oraz Mike Lesk i Joe Ossanna za nieocenioną pomoc przy tworzeniu maszynopisu.

Brian W. Kernighan
Dennis M. Ritchie



WPROWADZENIE

Język C jest językiem ogólnego stosowania. Był on ściśle związany z systemem Unix, dla którego został opracowany, ponieważ zarówno sam system operacyjny, jak i większość programów działających pod jego kontrolą napisano w C. Jednak C nie jest przywiązany do żadnego systemu operacyjnego lub maszyny. Wprawdzie nazwano go „językiem programowania systemowego", jest bowiem wygodnym narzędziem do konstruowania kompilatorów i systemów operacyjnych, ale okazało się, że nadawał się równie dobrze do napisania ważniejszych programów z wielu różnych dziedzin.

Dużo znaczących idei C pochodzi z języka BCPL, opracowanego przez Martina Richardsa. Wpływ języka BCPL na język C zaznaczył się pośrednio poprzez język B, stworzony w 1970 r. przez Kena Thompsona dla pierwszego systemu Unix działającego na DEC PDP-7.

Języki BCPL i B należą do klasy języków „beztypowych". W przeciwieństwie do nich w języku C występują różnorodne typy danych. Podstawowymi obiektami danych są znaki, liczby całkowite i liczby zmiennopozycyjne o różnych zakresach. Dodatkowe możliwości stwarzają typy pochodne, tworzone z użyciem wskaźników, tablic, struktur i unii. Wyrażenia buduje się z operatorów i ich argumentów; dowolne wyrażenie – łącznie z przypisaniem lub wywołaniem funkcji – może pełnić rolę instrukcji. Wskaźniki służą do wykonywania obliczeń na adresach w sposób niezależny od maszyny.

Język C jest wyposażony w podstawowe konstrukcje sterujące, niezbędne przy dobrym programowaniu strukturalnym: grupowanie instrukcji, podejmowanie decyzji (if-else), wybór jednego ze zbioru możliwych przypadków (switch), powtarzanie ze sprawdzaniem warunku zatrzymania na początku (while, for) lub na końcu pętli (do) oraz przerwanie pętli (break).

Funkcje mogą zwracać wartości o typach podstawowych, struktury, unie i wskaźniki. Dowolną funkcję można wywoływać rekurencyjnie. Zmienne lokalne funkcji są zwykle „automatyczne", czyli tworzone na nowo przy każdym jej wywołaniu. Definicje funkcji nie mogą być zagnieżdżone, natomiast zmienne można deklarować według

zasad struktury blokowej. Funkcje programu napisanego w C można umieścić w osobnych plikach źródłowych i tłumaczyć oddzielnie. Wyróżnia się zmienne: wewnętrzne funkcji; zewnętrzne, lecz znane tylko w obrębie jednego pliku źródłowego; bądź całkowicie globalne, dostępne w całym programie.

W przebiegu wstępnej kompilacji programu (tzw. faza preprocesora) wykonuje się makrogenerację, włączanie do programu zawartości innych plików źródłowych oraz kompilację warunkową.

C jest językiem względnie „niskiego poziomu". Określenie to nie zawiera w sobie nic negatywnego; po prostu znaczy tyle, że C posługuje się tym samym rodzajem obiektów, co większość komputerów, a więc znakami, liczbami i adresami. Obiekty te mogą być łączone lub przemieszczane za pomocą zwykłych operacji arytmetycznych i logicznych dostępnych na istniejących maszynach.

W języku C nie istnieją operacje na takich obiektach złożonych, jak ciągi znaków, zbiory, listy czy tablice. Nie ma operacji do manipulowania całą tablicą czy napisem, chociaż struktury mogą być kopiowane w całości. Język nie daje innych możliwości przydziału pamięci niż definicja statyczna i obsługa stosu realizowana przez zmienne lokalne funkcji; nie ma takich pojęć, jak sterta (ang. *heap*) czy odśmiecanie (ang. *garbage collection*). I wreszcie, sam język C nie dostarcza narzędzi wejścia-wyjścia: nie ma instrukcji READ (czytaj) lub WRITE (pisz) ani wbudowanych metod dostępu do pliku. Są to mechanizmy wyższego poziomu i muszą być wprowadzone do języka przez jawnie wywoływane funkcje. W większości instalacji C występuje sensowny standardowy zestaw takich funkcji.

Podobnie, język C oferuje jedynie proste, jednościeżkowe konstrukcje sterujące: instrukcje warunkowe, pętle, grupowanie instrukcji i podprogramy; nie zapewnia wieloprogramowości, współbieżności, synchronizacji czy współprogramów.

Jakkolwiek brak kilku takich cech może wydawać się poważną usterką („Sugerujecie, że dla porównania dwóch ciągów znaków będę musiał wywoływać jakąś funkcję?"), to sprowadzenie języka do tak skromnych rozmiarów przyniosło wymierne korzyści. Ponieważ C jest względnie mały, jego opis zajmuje mało miejsca i można go szybko przyswoić. Wolno więc rozsądnie założyć, że programista zna, rozumie i rzeczywiście regularnie używa pełnej mocy języka.

Przez wiele lat definicją języka C był przewodnik zawarty w pierwszym wydaniu książki *Język C*. W 1983 r. Amerykański Narodowy Instytut Standaryzacji (ANSI) powołał komitet, którego zadaniem było sformułowanie nowoczesnej, wszechstronnej definicji języka C. Ostatecznie pod koniec 1988 r. powstało opracowanie tzw. standardu ANSI lub inaczej „ANSI C". Współczesne kompilatory realizują już większość cech tego standardu.

Definicję standardu oparto na oryginalnym przewodniku języka C. Sam język relatywnie mało się zmienił; jednym z głównych założeń standardu było bowiem zagwarantowanie poprawności większości już istniejących programów, a jeśli to niemożliwe – aby kompilatory mogły produkować komunikaty ostrzegające o nowym postępowaniu.

Dla większości programistów najważniejszą zmianą jest nowa składnia deklaracji i definicji funkcji. Deklaracja funkcji może teraz zawierać opis parametrów funkcji. Ta dodatkowa informacja znacznie ułatwia kompilatorom wykrywanie błędów spowodowanych niezgodnością parametrów i argumentów wywołania funkcji. Z naszego doświadczenia wynika, że jest to bardzo pożyteczne uzupełnienie języka.

W języku wystąpiły też mniej ważne zmiany. Od dawna powszechnie stosowane przypisania struktur i wyliczenia są dziś oficjalną częścią języka. Obliczenia zmiennopozycyjne można teraz wykonywać w pojedynczej precyzji. Klarownie wyłożono cechy arytmetyki, szczególnie dla obiektów całkowitych bez znaku. Starannie opracowano możliwości preprocesora. Przeważająca część tych zmian dla większości programistów będzie miała niewielkie znaczenie.

Drugim znaczącym wkładem standardu jest definicja biblioteki towarzyszącej C. Są tam wymienione funkcje odwołujące się do systemu operacyjnego (np. czytanie z pliku lub pisanie do pliku), realizujące formatowane wejście i wyjście, przydział pamięci, operacje na tekstach i inne podobne. Zestaw standardowych plików nagłówkowych zapewnia jednolity dostęp do deklaracji tych funkcji i ich typów danych. Gwarantuje się, że programy, które do współpracy z systemem operacyjnym korzystają z biblioteki standardowej, będą wszędzie zachowywać się jednakowo. Biblioteka ta prawie w całości jest ściśle wzorowana na tzw. „standardowej bibliotece wejścia-wyjścia" systemu Unix. Ją właśnie opisano w pierwszym wydaniu i powszechnie stosowano równie dobrze w innych systemach. I znów większość programistów nie dostrzeże tu istotnych zmian.

Biblioteka czasu wykonania, wymagana przy instalacji samodzielnych programów, jest niewielka, ponieważ typy danych i struktury sterujące języka C występują bezpośrednio w prawie wszystkich istniejących komputerach. Funkcje z biblioteki standardowej są wywoływane jedynie w sposób jawny, można więc ich nie włączać do programu, jeśli nie są potrzebne. Większość z nich można przenośnie napisać w C – oprócz tych, które ukrywają szczegóły dotyczące systemu operacyjnego.

Chociaż język C odzwierciedla możliwości wielu komputerów, nie jest zależny od architektury żadnej konkretnej maszyny. Dzięki temu – z pewną dozą ostrożności – łatwo można pisać programy „przenośne", co znaczy tyle, że mogą być uruchamiane bez zmian na różnorodnym sprzęcie. Standard wyraźnie określa warunki przenośności i ustanawia zbiór stałych charakterystycznych dla maszyny, na której ma działać program.

C nie jest językiem o silnie określonych typach danych, ale wraz z jego rozwojem wzmogła się także kontrola tych typów. Oryginalna definicja C dopuszczała, choć z dezaprobatą, wymienność wskaźników i liczb całkowitych. Od dawna było to eliminowane i teraz standard wymaga poprawnych deklaracji oraz jawnych przekształceń, które zostały już wymuszone przez dobre kompilatory. Nowe deklaracje funkcji są innym krokiem w tym samym kierunku. Kompilatory mają ostrzegać o najczęstszych błędach typów, wyeliminowano także automatyczne przekształcenia danych o niezgodnych typach. Niemniej jednak język C zachował swoją zasadniczą filozofię, według której programista wie co robi; wymaga się jedynie, aby jawnie sformułował swoje zamiary.

Na koniec C, jak każdy inny język, ma swoje wady. Kilka operatorów ma źle określony priorytet; pewne części składni mogłyby być lepsze. Mimo to C udowodnił, że jest niezwykle sprawnym i wyrazistym językiem dla szerokiej gamy zastosowań.

Materiał przedstawiony w tej książce został podzielony w następujący sposób. Rozdział 1 jest elementarnym wprowadzeniem głównych idei języka C. Chodzi o to, aby Czytelnik zaczął programować w C tak szybko, jak tylko jest to możliwe, mocno bowiem wierzymy, że jedynym sposobem nauczenia się nowego języka jest pisanie w nim programów. „Elementarz'' zakłada praktyczną znajomość podstawowych zasad programowania. Nie ma w nim ani wyjaśnień dotyczących komputera czy kompilacji, ani znaczenia wyrażeń typu n=n+1. I choć próbowaliśmy wszędzie, gdzie to możliwe, pokazywać praktyczne techniki programowania, książka z założenia nie jest podręcznikiem uczącym struktur danych czy algorytmów. Gdy trzeba było wybierać, koncentrowaliśmy się na języku.

W rozdziałach od 2 do 6 są omówione różne aspekty języka C – bardziej szczegółowo i nieco bardziej formalnie niż w rozdz. 1. Nadal jednak większy nacisk położono na przykłady kompletnych programów, niż na ich odosobnione fragmenty. W rozdziale 2 wprowadzono podstawowe typy danych, operatory i wyrażenia. Rozdział 3 traktuje o sterowaniu: if-else, switch, while, for itp. W rozdziale 4 opisano funkcje i strukturę programu – zmienne zewnętrzne, reguły dostępności, rozmaite pliki źródłowe itd. Wspomniano tam też i o preprocesorze. W rozdziale 5 omówiono wskaźniki i arytmetykę na adresach, rozdział 6 zaś zawiera szczegóły dotyczące struktur i unii.

W rozdziale 7 przedstawiono bibliotekę standardową, zapewniającą jednolity kontakt z systemem operacyjnym. Tę bibliotekę zdefiniowano w standardzie ANSI z intencją, aby była instalowana na wszystkich maszynach oferujących C. Programy korzystające z niej przy obsłudze wejścia-wyjścia i innych operacjach dostępu do systemu mogą więc być bez zmian przenoszone z jednego systemu do innego.

W rozdziale 8 omówiono związek między programami w C i systemem operacyjnym Unix, skupiając się na wejściu-wyjściu, systemie plików i przydziale pamięci. Część tego rozdziału dotyczy specyfiki systemu Unix, ale programiści pracujący z innymi systemami nadal mogą tu znaleźć pożyteczne informacje, choćby o tym, jak zrealizowano jedną z wersji biblioteki standardowej czy sugestie dotyczące tworzenia programów przenośnych.

Dodatek A jest przewodnikiem języka C. Oficjalnym dokumentem zawierającym opis składni i semantyki C jest standard ANSI. Jest on jednak przeznaczony głównie dla konstruktorów kompilatorów. Nasz przewodnik wyraża definicję języka bardziej zwięźle i bez tego urzędowego stylu. Dodatek B jest streszczeniem zawartości biblioteki standardowej; i tu znów piszemy raczej dla użytkowników niż dla instalatorów. Dodatek C jest krótkim podsumowaniem zmian, jakie zaszły w oryginalnym języku. Jednak w sytuacjach wątpliwych ostatecznymi autorytetami są standard i dostępny kompilator.

1 ELEMENTARZ

Rozpoczynamy od szybkiego zapoznania się z językiem C. Naszym celem jest pokazanie podstawowych elementów języka za pomocą rzeczywistych programów, ale bez zagłębiania się w szczegóły, reguły formalne czy wyjątki. Nie próbujemy tu przedstawić tematu wyczerpująco ani nawet dokładnie (dbając jednak o poprawność przykładów). Chcemy, tak szybko, jak tylko to możliwe, doprowadzić Cię do punktu, w którym już mógłbyś pisać potrzebne programy. Aby to osiągnąć, musimy skupić się na podstawowych elementach języka, takich jak: zmienne i stałe, arytmetyka, sterowanie przebiegiem programu, funkcje i początki wejścia-wyjścia. W tym rozdziale celowo opuszczamy te elementy języka, które mają istotne znaczenie przy pisaniu dużych programów, a więc wskaźniki, struktury, większość z bogatego zestawu operatorów języka C, kilka instrukcji sterujących i bibliotekę standardową.

Oczywiście takie podejście ma swoje wady. Największą z nich jest to, że kompletnego opisu dowolnego elementu języka nie można znaleźć w jednym miejscu oraz że „elementarz", będąc tylko streszczeniem, może wprowadzać pewne niejasności. Także przykłady nie są tak zwięzłe i eleganckie, jak mogłyby być, ponieważ nie wykorzystują pełnej mocy języka C. Próbowaliśmy zminimalizować te efekty, jednak ostrzegamy. Inną wadą jest to, że informacje zawarte w tym rozdziale z konieczności powtarzają się w następnych rozdziałach. Mamy nadzieję, że takie powtórzenia raczej pomogą Ci, niż zirytują.

W każdym razie doświadczeni programiści będą mogli od razu skorzystać z wiadomości podanych w tym rozdziale. Początkujący winni je uzupełnić, pisząc samemu podobne małe programy. Obie grupy mogą traktować je jako szablon, który będzie wypełniany bardziej szczegółowymi wyjaśnieniami rozpoczynającymi się w rozdz. 2.

1.1 Zaczynamy

Jedynym sposobem nauczenia się nowego języka programowania jest pisanie programów w tym języku. Pierwszy program jest na ogół taki sam dla wszystkich języków:

Wypisz tekst

```
ahoj, przygodo
```

I już pojawia się pierwsza duża przeszkoda. Aby ją pokonać, powinieneś umieć: sporządzić gdzieś tekst programu, pomyślnie go przetłumaczyć, załadować i uruchomić. Musisz także wiedzieć, gdzie szukać wyniku. Po opanowaniu takich mechanicznych czynności cała reszta jest już dość prosta.

Oto program w języku C wypisujący tekst „ahoj, przygodo":

```
#include <stdio.h>

main()
{
    printf("ahoj, przygodo\n");
}
```

To, jak wykonać program, zależy od systemu, którego używasz. W przypadku systemu operacyjnego Unix musisz utworzyć program źródłowy w pliku o nazwie zakończonej „.c", przypuśćmy ahoj.c, a następnie przetłumaczyć go za pomocą polecenia

```
cc ahoj.c
```

Jeśli niczego nie spartaczysz (jak choćby ominięcie znaku czy błąd składniowy), to tłumaczenie przebiegnie cicho i powstanie plik o nazwie a.out. Załadowanie go i uruchomienie programu poleceniem

```
a.out
```

spowoduje wypisanie

```
ahoj, przygodo
```

W innych systemach zasady postępowania mogą być inne; skonsultuj się z miejscowym ekspertem.

Teraz kilka uwag dotyczących samego programu. Program w języku C, niezależnie od rozmiaru, jest zbudowany z *funkcji* i *zmiennych*. Funkcja zawiera *instrukcje* określające, jakie operacje procesu obliczeniowego należy wykonać, zmienne zaś przechowują wartości używane podczas tego procesu. Funkcje języka C są podobne do podprogramów i funkcji Fortranu lub do procedur i funkcji Pascala. W naszym przykładzie występuje funkcja o nazwie main. Zwykle masz prawo nadawać funkcjom dowolne nazwy, lecz main jest nazwą specjalną – Twój program rozpoczyna działanie od początku funkcji main. To znaczy, że każdy program *musi* zawierać (gdziekolwiek) funkcję o takiej nazwie.

```
#include <stdio.h>                    włączenie informacji o bibliotece standardowej

main()                                definicja funkcji o nazwie main, która nie
                                      oczekuje żadnych wartości argumentów

{                                     nawiasy klamrowe otaczają instrukcje funkcji
  printf ("ahoj, przygodo\n");        funkcja main do wypisania ciągu znaków
                                      wywołuje biblioteczną funkcję printf;
                                      \n reprezentuje znak nowego wiersza
}
```

Pierwszy program napisany w języku C

Na ogół main, aby wykonać zadanie, woła na pomoc inne funkcje. Niektóre z tych funkcji napisałeś sam, a inne pochodzą z bibliotek, które Ci dostarczono. Pierwszy wiersz programu

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

zleca kompilatorowi dołączenie do programu informacji o standardowej bibliotece wejścia-wyjścia. Ten wiersz występuje na początku wielu plików źródłowych języka C. Biblioteka standardowa jest opisana w rozdz. 7 i w dodatku B.

Jedną z metod komunikacji między funkcjami jest przekazywanie danych. Funkcja wywołująca tworzy listę wartości zwanych *argumentami* i dostarcza ją funkcji wywoływanej. Listę argumentów umieszcza się w nawiasach okrągłych bezpośrednio po nazwie funkcji. W naszym przykładzie main jest funkcją, która nie oczekuje żadnych argumentów, na co wskazują puste nawiasy ( ).

Instrukcje wykonywane przez funkcję umieszcza się w nawiasach klamrowych { }. Nasza funkcja main zawiera tylko jedną instrukcję

```
printf("ahoj, przygodo\n");
```

Funkcję wywołuje się, podając jej nazwę i w nawiasach listę argumentów; nasza instrukcja jest zatem wywołaniem funkcji printf z jednym argumentem "ahoj, przygodo\n". Funkcja ta, pochodząca z biblioteki standardowej, wypisuje teksty na wyjście – w tym przypadku ciąg znaków ujęty w znaki cudzysłowu.

Ciąg znaków ujęty w znaki cudzysłowu, jak "ahoj, przygodo\n", nazywa się *stałą napisową* lub *napisem*. Na razie stałe napisowe będziemy stosować jedynie w argumentach funkcji printf i innych.

W naszym przykładzie występuje sekwencja \n, która – zgodnie z notacją języka C – reprezentuje *znak nowego wiersza*; znak ten powoduje przerwanie wypisywania w bieżącym wierszu i wznowienie wypisywania od lewego marginesu w następnym wierszu. Jeśli opuścisz \n (pożyteczny eksperyment), to okaże się, że po wypisaniu tekstu nie nastąpi przejście do nowego wiersza. Jedynym sposobem uzyskania znaku nowego wiersza w argumencie printf jest użycie sekwencji \n; jeśli spróbujesz czegoś takiego, jak

```
printf("ahoj, przygodo
");
```

to kompilator języka C wypisze komunikat o błędzie.

Funkcja printf nigdy nie dostawia automatycznie znaku nowego wiersza, wywołując ją wielokrotnie można więc stopniowo budować jeden wiersz wyniku. Zatem nasz pierwszy program da się napisać również w postaci

```
#include <stdio.h>

main()
{
    printf("ahoj, ");
    printf("przygodo");
    printf("\n");
}
```

a wynik będzie identyczny.

Zwróć uwagę na to, że sekwencja \n reprezentuje tylko jeden znak. Takie *sekwencje specjalne*, jak \n, stanowią uniwersalny aparat pozwalający reprezentować znaki niegraficzne lub trudne do uzyskania. W języku C między innymi występują sekwencje: \t dla znaku tabulacji, \b dla znaku cofania (ang. *backspace*), \" dla znaku cudzysłowu oraz \\ dla samego (jednego) znaku \ (ang. *backslash*). Kompletną listę sekwencji specjalnych znajdziesz się w p. 2.3.

Ćwiczenie 1.1. Wykonaj program wypisujący tekst „ahoj, przygodo" pod kontrolą Twojego systemu operacyjnego. Spróbuj opuszczać fragmenty programu, aby poznać komunikaty o błędach.

Ćwiczenie 1.2. Spróbuj sprawdzić, co się stanie, gdy w argumencie funkcji printf wystąpi sekwencja \c, w której c jest dowolnym znakiem różnym od wyżej wymienionych.

1.2 Zmienne i wyrażenia arytmetyczne

Kolejny program wypisuje następujące zestawienie temperatur w skali Fahrenheita i ich odpowiedników w (stustopniowej) skali Celsjusza, wyliczonych według wzoru $C=(5/9)(F-32)$:

```
0    -17
20   -6
40   4
60   15
80   26
100  37
120  48
140  60
160  71
180  82
200  93
220  104
240  115
260  126
280  137
300  148
```

Sam program ciągle składa się z definicji jednej funkcji o nazwie main. Jest on dłuższy niż program wypisujący tekst „ahoj, przygodo", ale nie jest bardziej skomplikowany. Wprowadzono w nim kilka nowych pojęć, jak komentarze, deklaracje, zmienne, wyrażenia arytmetyczne, pętle i formatowane wypisywanie danych.

```
#include <stdio.h>

/* wypisz zestawienie temperatur Fahrenheita-Celsjusza
   dla f = 0, 20, ..., 300 */
main()
{
   int fahr, celsius;
   int lower, upper, step;

   lower = 0;      /* dolna granica temperatur */
   upper = 300;    /* górna granica */
   step  = 20;     /* rozmiar kroku */
```

```
    fahr = lower;
    while (fahr <= upper) {
        celsius = 5 * (fahr-32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

Dwa wiersze programu

```
/* wypisz zestawienie temperatur Fahrenheita-Celsjusza
   dla f = 0, 20, ..., 300 */
```

są *komentarzem*, który w tym przypadku zwięźle opisuje działanie programu. Kompilator ignoruje wszystkie znaki zawarte między ogranicznikami /* i */. Komentarze mogą pojawić się wszędzie tam, gdzie może wystąpić odstęp, znak tabulacji i znak nowego wiersza. Odpowiednio stosowane sprawiają, że programy stają się bardziej zrozumiałe.

W języku C wszystkie zmienne muszą być zadeklarowane przed ich użyciem, zwykle na początku funkcji przed pierwszą wykonywalną instrukcją. *Deklaracja* zapowiada właściwości zmiennych. Słada się ona z nazwy *typu* i z listy zmiennych, jak w naszym przykładzie

```
int fahr, celsius;
int lower, upper, step;
```

Typ int oznacza, że wymienione zmienne są liczbami całkowitymi, w przeciwieństwie do float, wprowadzającego liczby zmiennopozycyjne, czyli liczby z częścią ułamkową. Zakres obu typów (int i float) zależy od maszyny, z którą pracujesz; 16-bitowe obiekty typu int o wartościach z przedziału od −32 768 do +32 767) są równie powszechne, jak obiekty 32-bitowe. Obiekt typu float zwykle zajmuje 32 bity, co daje w przybliżeniu co najmniej 6 cyfr znaczących dla wartości z zakresu od 10^{-38} do 10^{+38}.

Oprócz int i float w języku C występuje także kilka innych podstawowych typów danych:

char	znak – jeden bajt,
short	liczba całkowita krótka,
long	liczba całkowita długa,
double	liczba zmiennopozycyjna podwójnej precyzji.

Rozmiary tych obiektów również zależą od maszyny. W języku C występują także *tablice*, *struktury* i *unie* zbudowane z obiektów o typach podstawowych, *wskaźniki* do tych obiektów oraz *funkcje* zwracające ich wartości. Z nimi wszystkimi zapoznasz się w odpowiednim czasie.

Faktyczne działanie programu przekształcania temperatur rozpoczynają *instrukcje przypisania*

```
lower = 0;
upper = 300;
step  = 20;
fahr  = lower;
```

w których zmiennym nadaje się wartości początkowe. Poszczególne instrukcje są zakończone średnikami.

Wszystkie wiersze zestawienia tworzy się w ten sam sposób, w programie użyliśmy więc pętli, której wykonanie jest powtarzane kolejno dla każdego wiersza. Zadanie to realizuje pętla while (dopóki)

```
while (fahr <= upper) {
...
}
```

Działanie pętli while jest następujące: Najpierw sprawdza się warunek ujęty w nawiasy okrągłe. Jeśli jest prawdziwy (tzn. fahr jest mniejsze lub równe upper), to wykonuje się treść pętli (trzy instrukcje zawarte w nawiasach klamrowych). Potem znów jest sprawdzany warunek i, jeśli będzie prawdziwy, ponownie wykona się treść pętli. W chwili, gdy warunek stanie się fałszywy (fahr przekroczy upper), nastąpi koniec pętli, a kolejną wykonywaną instrukcją będzie pierwsza instrukcja po pętli. Ponieważ w tym programie nie ma więcej instrukcji, więc program zakończy działanie.

Treść pętli while może składać się z jednej lub kilku instrukcji zawartych w nawiasach klamrowych (tak jak w programie przekształcania temperatur) lub z jednej instrukcji bez tych nawiasów, np.

```
while (i < j)
    i = 2 * i;
```

W obu przypadkach instrukcje wykonywane przez pętlę while zawsze umieszczamy w wierszach programu źródłowego z wcięciem o wielkości jednego znaku tabulacji (w przykładach pokazany jako stały odstęp). Na pierwszy rzut oka widać więc instrukcje występujące wewnątrz pętli. Wcięcia podkreślają logiczną strukturę programu. Choć dla kompilatorów języka C wygląd zewnętrzny programów nie ma znaczenia, to jednak odpowiednie stosowanie wcięć i odstępów ma istotne znaczenie dla czytelnika-człowieka. My zalecamy pisanie tylko jednej instrukcji w wierszu i otaczanie operatorów znakami odstępu dla uwypuklenia grupowania argumentów operacji. Położenie nawiasów klamrowych jest mniej istotne, chociaż istnieją żarliwi wyznawcy różnych koncepcji. My wybraliśmy jeden z wielu popularnych stylów. Wybierz styl, który Ci najbardziej odpowiada, i stosuj go konsekwentnie we wszystkich programach.

Większość pracy programu jest realizowana przez pętlę while. Obliczenie temperatury w skali Celsjusza i przypisanie zmiennej celsius tej wartości wykonuje instrukcja

```
celsius = 5 * (fahr-32) / 9;
```

Mnożenie przez 5 i dzielenie przez 9 zamiast mnożenia po prostu przez 5/9 jest spowodowane tym, że w języku C (jak w wielu innych językach) dzielenie całkowite *zaokrągla* wynik, to znaczy odrzuca część ułamkową tego wyniku. Liczby 5 i 9 są całkowite, więc wyrażenie 5/9 dałoby w wyniku zero i wtedy wszystkie temperatury Celsjusza miałyby wartość zero.

Ten przykład także mówi coś więcej o działaniu funkcji printf. W rzeczywistości printf jest funkcją formatowanego wypisywania danych. Jej szczegółowy opis znajduje się w rozdz. 7. Pierwszym argumentem funkcji printf jest ciąg znaków, które należy wypisać. Każdy znak % symbolicznie wskazuje miejsce na wartość kolejnego argumentu (tzn. drugiego, trzeciego itd.) oraz podaje format, w jakim ta wartość będzie wypisana. Na przykład specyfikacja %d wskazuje na argument całkowity, więc instrukcja

```
printf("%d\t%d\n", fahr, celsius);
```

spowoduje wypisanie wartości dwóch argumentów całkowitych fahr i celsius przedzielonych znakiem tabulacji (\t).

Każda specyfikacja przekształcenia (którą wprowadza znak %) zawarta w pierwszym argumencie funkcji printf stanowi parę z odpowiadającym jej (drugim, trzecim itd.) argumentem wywołania tej funkcji. Argumenty muszą być właściwie uporządkowane ze względu na numer i typ specyfikacji, gdyż inaczej możesz uzyskać błędne wyniki.

Przy okazji zwróć uwagę na to, że funkcja printf nie jest częścią języka C – on sam nie definiuje wejścia ani wyjścia. Po prostu printf jest pożyteczną funkcją ze standardowej biblioteki funkcji powszechnie dostępnych dla programów napisanych w języku C. Zachowanie się funkcji printf jest zdefiniowane w standardzie ANSI C, więc jej właściwości powinny być takie same we wszystkich kompilatorach i bibliotekach dostosowanych do standardu.

Aby skoncentrować się na samym języku C, nie będziemy na razie rozwodzić się na temat wejścia-wyjścia aż do rozdz. 7. W szczególności pominiemy formatowane wejście, które tam opisano. Jeśli będziesz zmuszony wprowadzić liczby, to przeczytaj omówienie funkcji scanf w p. 7.4. Funkcja ta jest bardzo podobna do printf z tym, że czyta dane z wejścia zamiast wypisywać wyniki na wyjście.

Z programem przekształcania temperatur wiąże się kilka problemów. Najprostszym z nich jest ten, że wynik nie wygląda ładnie, ponieważ liczby w kolumnach nie są wyrównane do prawej strony. To da się łatwo naprawić: gdy każdą specyfikację %d w instrukcji printf uzupełnimy szerokością pola, wówczas wypisywane liczby będą dosunięte do prawej granicy pola. Możemy na przykład napisać

```
    printf("%3d %6d\n", fahr, celsius);
```

i w każdym wierszu pierwsza liczba zostanie wypisana w polu dla trzech cyfr, a druga – w polu dla sześciu cyfr; wynik będzie wówczas wyglądał tak

```
  0    -17
 20     -6
 40      4
 60     15
 80     26
100     37
 ...
```

Bardziej poważnym problemem jest dokładność wyniku. Wyliczone temperatury Celsjusza nie są zbyt dokładne, zastosowaliśmy bowiem arytmetykę liczb całkowitych; np. odpowiednikiem temperatury 0° F faktycznie jest temperatura ok. −17,8°C, a nie −17. Aby uzyskać dokładniejsze wyniki, musimy zamiast arytmetyki całkowitej zastosować arytmetykę liczb zmiennopozycyjnych. To zaś wymaga paru zmian w programie. Oto jego nowa wersja

```
#include <stdio.h>

/* wypisz zestawienie temperatur Fahrenheita-Celsjusza
   dla fahr = 0, 20, ..., 300; wersja zmiennopozycyjna */
main()
{
    float fahr, celsius;
    int lower, upper, step;

    lower = 0;      /* dolna granica temperatur */
    upper = 300;    /* górna granica */
    step = 20;      /* rozmiar kroku */

    fahr = lower;
    while (fahr <= upper) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%3.0f %6.1f\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

Program wygląda prawie tak samo jak poprzednio, przy czym teraz zmienne fahr i celsius są zadeklarowane jako float, a specyfikacje przekształceń wyniku mają bardziej naturalną postać. W poprzedniej wersji nie mogliśmy użyć wyrażenia 5/9, ponieważ dzielenie całkowite odrzuca część ułamkową wyniku. Kropka dziesiętna w stałej informuje o tym, że jest to liczba zmiennopozycyjna, toteż wynik dzielenia 5.0/9.0 nie zostanie zaokrąglony, wyraża bowiem proporcję między dwiema liczbami zmiennopozycyjnymi.

W przypadku, gdy oba argumenty operatora arytmetycznego są całkowite, wykonuje się operację całkowitą. Jeśli jednak jeden z nich jest całkowity, a drugi zmiennopozycyjny, to liczbę całkowitą przekształca się do typu zmiennopozycyjnego przed wykonaniem operacji. Gdyby w programie było wyrażenie fahr–32, to i tak liczba 32 byłaby automatycznie przekształcona na zmiennopozycyjną. Niemniej jednak zapisywanie stałych zmiennopozycyjnych z kropką dziesiętną nawet wtedy, gdy ich wartości są całkowite, podkreśla dla czytelnika ich zmiennopozycyjną naturę.

Szczegółowe zasady przekształcania liczb całkowitych na zmiennopozycyjne podano w rozdz. 2. Na razie zapamiętaj, że zarówno przypisanie

```
fahr = lower;
```

jak i sprawdzenie warunku

```
while (fahr <= upper)
```

zachowują się naturalnie – typ int jest przekształcany do float przed wykonaniem operacji.

Specyfikacja przekształcenia %3.0f w funkcji printf rezerwuje dla liczby zmiennopozycyjnej (tutaj fahr) co najmniej trzy znaki, przy czym należy ją wypisać bez części ułamkowej i bez kropki dziesiętnej. Specyfikacja %6.1f opisuje format następnej liczby (celsius): ma ona zająć co najmniej sześć znaków i zawierać jedną cyfrę po kropce dziesiętnej. Wynikowe zestawienie wygląda więc tak:

```
  0   -17.8
 20    -6.7
 40     4.4
...
```

Szerokość pola i precyzję wyniku można w specyfikacji przekształcenia liczby pominąć. Na przykład %6f przeznacza na liczbę co najmniej sześć znaków, %.2f wymusza dwa miejsca po kropce dziesiętnej, lecz nie określa sztywnego rozmiaru pola, a %f po prostu zleca wypisanie liczby w postaci zmiennopozycyjnej.

W wyniku następujących specyfikacji przekształcenia argument zostanie wypisany jako:

%d	liczba dziesiętna;
%6d	liczba dziesiętna, zajmująca co najmniej 6 znaków;
%f	liczba zmiennopozycyjna;
%6f	liczba zmiennopozycyjna, zajmująca co najmniej 6 znaków;
%.2f	liczba zmiennopozycyjna z 2 miejscami po kropce dziesiętnej;
%6.2f	liczba zmiennopozycyjna z 2 miejscami po kropce, zajmująca co najmniej 6 znaków.

Funkcja printf rozpoznaje także specyfikacje: %o – powodującą wypisanie liczby w postaci ósemkowej, %x – liczby w postaci szesnastkowej, %c – jednego znaku, %s – ciągu znaków oraz %% jako żądanie wypisania znaku %.

Ćwiczenie 1.3. Zmień program przekształcania temperatur tak, aby wypisywał również nagłówek zestawienia.

Ćwiczenie 1.4. Napisz program wypisujący zestawienie temperatur w skali Celsjusza i ich odpowiedników w skali Fahrenheita.

1.3 Instrukcja for

Każdy program realizujący konkretne zadanie można napisać na wiele sposobów. Spróbujmy więc napisać inny wariant programu przekształcania temperatur.

```
#include <stdio.h>

/* zestawienie temperatur Fahrenheita-Celsjusza */
main()
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Ten program produkuje identyczne wyniki, ale w istotny sposób różni się od poprzedniego. Główną zmianą jest eliminacja większości zmiennych; pozostała tylko zmienna fahr, znów jako obiekt typu int. Dolna i górna granica oraz rozmiar kroku występują jedynie jako stałe w nowej instrukcji for, natomiast wyrażenie obliczające temperaturę w skali Celsjusza jest teraz trzecim argumentem wywołania funkcji printf, a nie osobną instrukcją przypisania.

Ta ostatnia zmiana jest ilustracją podstawowej zasady języka C: wszędzie tam, gdzie może wystąpić wartość zmiennej pewnego typu, możesz zastosować bardziej skomplikowane wyrażenie tego typu. Trzeci argument funkcji printf powinien być wartością zmiennopozycyjną (odpowiadającą formatowi %6.1f), a więc może nim być dowolne wyrażenie zmiennopozycyjne.

Instrukcja for jest pętlą bardziej ogólną niż while. Gdy porównasz tę instrukcję z instrukcją while w poprzednim przykładzie, wówczas jej działanie stanie się zrozumiałe. W nawiasach okrągłych instrukcji for występują trzy części oddzielone średnikami. Pierwszą część, inicjującą pętlę:

```
fahr = 0
```

wykonuje się raz przed wejściem do właściwej pętli. Druga część jest warunkiem sterującym powtarzaniem pętli:

```
fahr <= 300
```

Po obliczeniu wartości warunku i sprawdzeniu, że jest prawdziwy, wykona się treść pętli (tu jedynie wywołanie funkcji printf) oraz trzecia część, nazywana przyrostem

```
fahr = fahr + 20
```

w której zwiększa się wartość zmiennej. Następnie znów jest obliczana wartość warunku. Powtarzanie pętli kończy się z chwilą, gdy warunek stanie się fałszywy. Treść pętli, podobnie jak w instrukcji while, może być jedną instrukcją lub grupą instrukcji ujętą w nawiasy klamrowe. Części: inicjowanie, warunek i przyrost mogą być dowolnymi wyrażeniami.

Wybór między instrukcjami while i for zależy od tego, która z nich wydaje się bardziej przydatna. Instrukcję for zwykle stosuje się w pętlach, w których części inicjowania i przyrostu są pojedynczymi, logicznie związanymi instrukcjami. Jej postać jest bowiem bardziej zwarta niż postać instrukcji while i skupia w jednym miejscu instrukcje sterujące wykonaniem pętli.

Ćwiczenie 1.5. Zmień program przekształcania temperatur tak, aby wypisywał zestawienie w odwrotnej kolejności, to znaczy od 300 stopni do zera.

1.4 Stałe symboliczne

Jeszcze kilka ostatnich spostrzeżeń, zanim na zawsze porzucimy nasz program przekształcania temperatur. Do złej praktyki programowania należy zaszywanie w pro-

gramie takich „tajemniczych'' liczb, jak 300 czy 20: będą one niewiele znaczyć dla kogoś, kto w przyszłości będzie musiał przeczytać ten program. Trudno je także zmienić w sposób systematyczny. Jedną z metod postępowania z tajemniczymi liczbami jest nadawanie im znaczących nazw. Wiersz #define wprowadza definicję *nazwy symbolicznej*, nazwanej także *stałą symboliczną*, która ma reprezentować określony ciąg znaków:

#define *nazwa tekst zastępujący*

Od tej chwili każde wystąpienie *nazwy* (oprócz nazw zawartych w cudzysłowach oraz stanowiących fragment innej nazwy) zostanie zamienione na odpowiadający jej *tekst zastępujący*. Nazwa symboliczna ma taką samą postać, jak nazwa zmiennej: jest ciągiem liter i cyfr, rozpoczynającym się od litery. Zastępujący ją tekst może być zupełnie dowolnym ciągiem znaków, nie jest ograniczony tylko do liczb.

```
#include <stdio.h>

#define LOWER 0    /* dolna granica temperatur */
#define UPPER 300 /* górna granica */
#define STEP   20   /* rozmiar kroku */

main()   /* zestawienie temperatur Fahrenheita-Celsjusza */
{
    int fahr;

    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Obiekty LOWER, UPPER i STEP nie są zmiennymi, lecz stałymi symbolicznymi, toteż nie pojawiają się wśród deklaracji. Nazwy symboliczne umownie zapisuje się wielkimi literami alfabetu w celu odróżnienia ich od nazw zmiennych. Zwróć uwagę na brak średnika na końcu wiersza zawierającego #define.

1.5 Wejście i wyjście znakowe

Rozważymy teraz rodzinę programów służących do przetwarzania danych znakowych. Odkryjesz później, że wiele programów jest tylko rozszerzoną wersją prototypów, które tutaj omawiamy.

Model wprowadzania i wyprowadzania danych, realizowany przez funkcje z biblioteki standardowej, jest bardzo prosty. Wejściowy lub wyjściowy tekst – niezależnie od tego, skąd pochodzi ani dokąd zmierza – jest związany ze strumieniem znaków. Taki *strumień znaków* jest ciągiem znaków podzielonym na wiersze. Każdy wiersz składa się z zera lub więcej znaków, po których następuje znak nowego wiersza. To biblioteka ponosi odpowiedzialność za dostosowanie każdego strumienia wejściowego czy wyjściowego do tego modelu. Programista piszący w języku C i używający funkcji z biblioteki nie musi się troszczyć o to, jak wiersze są reprezentowane poza programem.

Standardowa biblioteka języka C zawiera kilka funkcji czytających lub wypisujących jeden znak. Funkcje getchar i putchar są najprostsze. Przy każdym wywołaniu funkcja getchar pobiera ze strumienia znaków *następny znak* i wraca z jego wartością. A więc po wykonaniu instrukcji

```
c = getchar()
```

zawartością zmiennej c jest następny znak z wejścia. Zwykle znaki pochodzą z klawiatury komputera; danymi pochodzącymi z pliku zajmiemy się w rozdz. 7.

Każde wywołanie funkcji putchar powoduje wypisanie jednego znaku. Instrukcja

```
putchar(c)
```

powoduje więc wypisanie zawartości zmiennej całkowitej c jako znaku na pewne urządzenie wyjściowe, którym zwykle jest ekran monitora. Wywołania funkcji putchar i printf mogą być przemieszane; postać wyniku będzie zgodna z kolejnością tych wywołań.

1.5.1 Kopiowanie plików

Korzystając z funkcji getchar i putchar, możesz napisać zaskakująco dużo pożytecznych programów bez dodatkowych wiadomości o mechanizmach wejścia-wyjścia. Najprostszym przykładem jest program kopiujący dane znak po znaku ze swojego wejścia na swoje wyjście. Ogólny schemat takiego programu wygląda następująco:

```
przeczytaj znak
while (znak nie jest sygnałem końca pliku)
    wypisz właśnie przeczytany znak
    przeczytaj następny znak
```

Po przekształceniu go na język C otrzymujemy program:

```
#include <stdio.h>

/* przepisz wejście na wyjście; wersja 1 */
main()
{
    int c;

    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```

Operator relacji != oznacza „różny od".

To, co pojawia się jako znak z klawiatury lub wyświetla na ekranie, ma oczywiście – jak wszystko inne – swoją reprezentację wewnętrzną, którą jest wzorzec bitowy. Do przechowywania takich danych znakowych wprowadzono specjalny typ char, lecz można używać dowolnego typu całkowitego. My stosujemy typ int z pewnego subtelnego, ale ważnego powodu.

Problem polega na tym, jak odróżnić koniec danych wejściowych od poprawnej danej. Przyjęto konwencję, że gdy wejście jest już puste, wówczas funkcja getchar zwraca wartość, której nie można pomylić z żadnym prawdziwym znakiem. Tę wartość nazwano symbolicznie EOF (ang. *end of file*), czyli koniec pliku. Typ zmiennej c musimy deklarować tak, aby mogła ona przechować każdą wartość zwracaną przez funkcję getchar. Nie możemy zastosować typu char, ponieważ zmienna c – poza wszystkimi możliwymi wartościami typu char – musi dodatkowo pomieścić wartość EOF. Właśnie dlatego w takich sytuacjach stosujemy typ int.

Stała EOF jest obiektem całkowitym, zdefiniowanym w standardowym pliku nagłówkowym <stdio.h>. Rzeczywista wartość tej stałej nie ma znaczenia dopóty, dopóki różni się od wszystkich wartości typu char. Zastosowanie stałej symbolicznej EOF upewnia nas w tym, że nasz program nie zależy od żadnej specyficznej wartości numerycznej.

Programiści zaawansowani w języku C mogą napisać program kopiowania w bardziej zwartej postaci. W języku C każde przypisanie, np.

```
c = getchar()
```

jest wyrażeniem i ma wartość – jest nią wartość lewego argumentu przypisania po wykonaniu tej operacji. Oznacza to, że przypisanie może pojawić się jako część większego wyrażenia. Jeśli przypisanie znaku zmiennej c zostanie umieszczone w części warunkowej pętli while, to program kopiowania można napisać tak:

```
#include <stdio.h>

/* przepisz wejście na wyjście; wersja 2 */
main()
{
    int c;

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

Pętla while otrzymuje znak, przypisuje go zmiennej c, a następnie sprawdza, czy to, co otrzymała, jest sygnałem końca pliku. Jeśli nie, to wykonuje się treść pętli, w której jest wypisywany znak, i wszystko powtarza się od początku. Po osiągnięciu końca pliku pętla while kończy działanie, a wraz z nią cały program main.

Wersja ta skupia w jednym miejscu obsługę wejścia – jest tylko jedno odwołanie do funkcji getchar – oraz skraca tekst programu. Nowa wersja jest bardziej zwarta i, po przyswojeniu nowego zwrotu języka, bardziej czytelna. Często spotkasz się z takim stylem programowania. (Można także przesadzić i tworzyć programy zupełnie nieczytelne; jest to jednak tendencja, którą usiłujemy zwalczać.)

Nawiasy otaczające przypisanie wewnątrz warunku pętli while są konieczne. *Priorytet* operatora relacji != jest wyższy niż priorytet operatora przypisania =, co znaczy że przy braku nawiasów sprawdzenie warunku != wykonałoby się przed przypisaniem =. Zatem zwrot

```
c = getchar() != EOF
```

jest równoważny ze zwrotem

```
c = (getchar() != EOF)
```

co ma zupełnie niepożądany skutek, przypisuje bowiem zmiennej c wartość 0 lub 1 w zależności od tego, czy funkcja getchar osiągnęła koniec pliku czy nie. (Więcej informacji na ten temat znajdziesz w rozdz. 2.)

Ćwiczenie 1.6. Sprawdź, że wyrażenie getchar() != EOF może mieć wartość 0 lub 1.

Ćwiczenie 1.7. Napisz program wypisujący wartość stałej symbolicznej EOF.

1.5.2 Zliczanie znaków

Następny program zlicza znaki; jest on podobny do programu kopiowania.

```
#include <stdio.h>

/* zlicz znaki wejściowe; wersja 1 */
main()
{
    long nc;

    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

Instrukcja

```
++nc;
```

prezentuje nowy operator ++, który oznacza *zwiększenie o jeden*. Zamiast tego możesz napisać nc=nc+1, ale ++nc jest bardziej zwięzłe i często bardziej efektywne. W języku C istnieje także operator *zmniejszania o jeden*; oznacza się go symbolem --. Operatory ++ i -- mogą występować albo jako operatory przedrostkowe (++nc), albo przyrostkowe (nc++). Obie formy różnią się wartościami przyjmowanymi w wyrażeniach, co opiszemy w rozdz. 2, jednak ++nc i nc++ tak samo zwiększają o jeden wartość nc. Na razie ograniczymy się do formy przedrostkowej.

Program zliczania znaków gromadzi ich liczbę w zmiennej typu long, a nie int. Obiekty całkowite typu long zajmują co najmniej 32 bity. Chociaż typy int i long na pewnych maszynach są tego samego rozmiaru, to na innych typ int zajmuje tylko 16 bitów. Oznacza to maksymalną wartość 32 767, a więc nawet dla względnie krótkiego pliku wejściowego mogłoby powstać przepełnienie licznika typu int. Specyfikacja przekształcenia %ld jest sygnałem dla funkcji printf, że odpowiedni argument jest liczbą całkowitą typu long.

Można sobie poradzić z jeszcze większymi liczbami, stosując typ double (tj. typ float podwójnej precyzji). Ponadto, aby zilustrować inny sposób tworzenia pętli, użyjemy instrukcji for zamiast while.

```
#include <stdio.h>

/* zlicz znaki wejściowe; wersja 2 */
main()
{
    double nc;

    for (nc = 0; getchar() != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

Funkcja printf używa formatu %f dla obu typów zmiennopozycyjnych float i double. Specyfikacja %.0f zapobiega wypisywaniu kropki dziesiętnej i części ułamkowej, która ma wartość zero.

W tym przypadku treść pętli for jest pusta, ponieważ cała praca zostanie wykonana w jej częściach: warunkowej i przyrostu. Zasady gramatyki języka C wymagają jednak, aby pętla for miała treść. Samotny średnik, zwany *instrukcją pustą*, spełnia to wymaganie. Napisaliśmy go w osobnym wierszu, żeby był bardziej widoczny.

Zanim porzucimy program zliczania znaków, zwróć uwagę na to, że jeśli plik wejściowy nie zawiera żadnych znaków, to warunek w pętli while czy for będzie fałszywy już przy pierwszym wywołaniu funkcji getchar. W tym przypadku program wypisze wartość zero, która jest odpowiedzią poprawną. Jest to bardzo ważne. Jedną z milszych cech instrukcji while i for jest to, że sprawdzenie warunku odbywa się zawsze na początku pętli, przed wykonaniem jej treści. Gdy nie ma nic do zrobienia, nic nie będzie zrobione, nawet jeśli oznacza to, że pętla nigdy się nie wykona. Program powinien postępować inteligentnie nawet wówczas, gdy jego strumień wejściowy ma długość zero. Instrukcje while i for wspomagają podejmowanie odpowiednich działań w sytuacjach wyjątkowych.

1.5.3 Zliczanie wierszy

Kolejny program zlicza wiersze pliku wejściowego. Jak już wspomnieliśmy, biblioteka standardowa zapewnia, że w wejściowym strumieniu znaków tekst jest ciągiem wierszy zakończonych znakiem nowego wiersza. Dlatego też zliczanie wierszy polega po prostu na zliczaniu znaków nowego wiersza.

```
#include <stdio.h>

/* zlicz wiersze wejściowe */
main()
{
    int c, nl;

    nl = 0;
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;
    printf("%d\n", nl);
}
```

Tym razem pętla while składa się z instrukcji if, która czuwa nad zwiększaniem licznika ++nl. Instrukcja ta sprawdza warunek ujęty w nawiasy okrągłe i, jeśli warunek jest prawdziwy, wykonuje następującą po niej instrukcję (lub grupę instrukcji zawartych w nawiasach klamrowych). Znów musimy wyjaśnić, co czym steruje.

W języku C podwójny znak równości == oznacza operator relacji „równe" (jak pojedynczy znak = w Pascalu czy .EQ. w Fortranie). Symbol ten wprowadzono po to, by odróżnić test na równość od znaku = oznaczającego przypisanie. Tu mała przestroga: nowicjusze w języku C niekiedy piszą = mając na myśli ==. Jak zobaczymy w rozdz. 2, w wyniku powstaje poprawne wyrażenie, tak że nie spodziewaj się komunikatu ostrzegającego o błędzie *.

Znak zawarty między dwoma apostrofami reprezentuje wartość całkowitą równą numerycznej wartości kodu tego znaku, określoną na podstawie maszynowego zbioru znaków. Jest to tzw. *stała znakowa*, choć w rzeczywistości stanowi jedynie inny sposób zapisania niewielkiej liczby całkowitej. Przykładowo: 'A' jest stałą znakową; dla zbioru znaków ASCII jej wartością jest 65, czyli wartość wewnętrznej reprezentacji znaku A. Oczywiście lepiej jest stosować stałą 'A' niż liczbę 65; znaczenie stałej jest oczywiste, a także nie zależy od konkretnego zbioru znaków.

Sekwencje specjalne używane w stałych napisowych są także poprawne w stałych znakowych, a więc stała znakowa '\n' oznacza wartość znaku nowego wiersza, która dla zbioru znaków ASCII równa się 10. Musisz na zawsze zapamiętać, że stała '\n' jest jednym znakiem i że w wyrażeniach jest równoważna jednej liczbie całkowitej. Natomiast stała napisowa "\n" jest ciągiem znaków, który przypadkowo tylko zawiera jeden znak. O znakach i napisach pomówimy szerzej w rozdz. 2.

* Niektóre przyzwoite kompilatory w sytuacjach wątpliwych produkują jednak komunikaty ostrzegawcze. – *Przyp. tłum.*

Ćwiczenie 1.8. Napisz program zliczający znaki odstępu, tabulacji i nowego wiersza.

Ćwiczenie 1.9. Napisz program, który – przepisując wejście na wyjście – będzie zastępować jednym znakiem odstępu każdy ciąg złożony z jednego lub kilku takich znaków.

Ćwiczenie 1.10. Napisz program, który podczas kopiowania wejścia na wyjście zastępuje każdy znak tabulacji przez sekwencję znaków \t, każdy znak cofania przez sekwencję \b oraz każdy znak \ przez dwa takie znaki.

1.5.4 Zliczanie słów

Czwarty program z naszej serii pożytecznych programów zlicza wiersze, słowa i znaki. Przyjęliśmy tu swobodną definicję słowa jako ciągu znaków nie zawierającego odstępów, znaków tabulacji i znaków nowego wiersza. Nasz program jest ubogą wersją usługowego programu wc w systemie Unix.

```
#include <stdio.h>

#define IN   1 /* wewnątrz słowa */
#define OUT 0 /* poza słowem */

/* zlicz wejściowe wiersze, słowa i znaki */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

Za każdym razem po rozpoznaniu pierwszego znaku słowa program zwiększa licznik słów nw. Zmienna state (stan) wskazuje, czy program jest właśnie wewnątrz słowa, czy nie; początkowo jej wartością jest stan „poza słowem", czemu przypisano wartość OUT. Zalecamy stosowanie stałych symbolicznych typu IN (w) i OUT (poza) dla tak prozaicznych wartości, jak 1 i 0; program staje się wówczas bardziej czytelny. W tak małym programie, jak nasz, nie jest to bardzo istotne, ale w większych programach wzrost czytelności naprawdę jest wart tego dodatkowego wysiłku – pisania w ten sposób od początku. Zauważysz później, że łatwiej jest wprowadzać rozległe zmiany w programach, w których magiczne liczby pojawiają się jedynie jako stałe symboliczne.

W wierszu

```
nl = nw = nc = 0;
```

wszystkim trzem zmiennym nadaje się wartość zero. Nie jest to przypadek specjalny, tylko naturalna konsekwencja faktu, że operacja przypisania jest wyrażeniem, a więc ma wartość, oraz że operatory przypisania są prawostronnie łączne. Znaczy to tyle, co

```
nl = (nw = (nc = 0));
```

Operator logiczny || oznacza „lub" (ang. *OR*), zatem w wierszu

```
if (c == ' ' || c == '\n' || c == '\t')
```

mówimy „jeśli c jest odstępem *lub* c jest znakiem nowego wiersza, *lub* c jest znakiem tabulacji, to". (Przypominamy, że sekwencja specjalna \t jest wizualną reprezentacją znaku tabulacji). W języku C występuje również operator logiczny && oznaczający „i" (ang. *AND*); jego priorytet jest wyższy niż priorytet operatora ||. Wyrażenia połączone operatorami || i && są obliczane od lewej strony do prawej i gwarantuje się, że obliczanie skończy się w chwili, w której jest już znana wartość całego wyrażenia: „prawda" lub „fałsz". W przypadku gdy zmienna c zawiera odstęp, nie potrzeba dalej sprawdzać, czy zawiera znak nowego wiersza czy tabulacji, a więc te możliwości nie będą sprawdzane. Nie jest to takie ważne w naszej sytuacji, lecz jest bardzo ważne w bardziej skomplikowanych warunkach, co wkrótce pokażemy.

W przykładzie występuje także słowo else, wprowadzające alternatywną akcję podejmowaną w chwili, gdy wartość warunku instrukcji if jest fałszywa. Ogólna postać instrukcji if jest następująca:

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

Wykonuje się jedną i tylko jedną z dwóch instrukcji związanych z if-else. Jeśli wyrażenie ma wartość „prawda", to zostanie wykonana *instrukcja1*, jeśli „fałsz" – *instrukcja2*. Każda *instrukcja* może być pojedynczą instrukcją lub kilkoma instrukcjami

ujętymi w nawiasy klamrowe. W naszym programie zliczania słów po else występuje instrukcja if, która steruje wykonaniem dwóch instrukcji zawartych w nawiasach klamrowych.

Ćwiczenie 1.11. Jak mógłbyś sprawdzić poprawność programu zliczającego słowa? Jaki rodzaj danych wejściowych najlepiej nadaje się do demaskowania błędów, jeśli tu występują?

Ćwiczenie 1.12. Napisz program, który każde słowo wejściowe wypisze w osobnym wierszu.

1.6 Tablice

Napiszmy program zliczający liczbę wystąpień każdej cyfry, każdego białego znaku (tj. znaku odstępu, tabulacji i nowego wiersza) oraz wszystkich innych znaków. Jest to oczywiście problem sztuczny, ale pozwoli nam przedstawić kilka właściwości języka C w jednym programie.

Wyróżniamy aż dwanaście rodzajów danych wejściowych, a więc dla cyfr wygodniej jest zastosować tablicę przechowującą liczniki wystąpień każdej z nich, niż dziesięć osobnych zmiennych. Oto jedna z wersji tego programu.

```
#include <stdio.h>

/* zlicz cyfry, białe znaki, inne */
main()
{
    int c, i, nwhite, nother;
    int ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; ++i)
        ndigit[i] = 0;

    while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
            ++nother;
```

```
    printf("cyfry =");
    for (i = 0; i < 10; ++i)
        printf(" %d", ndigit[i]);
    printf(", białe znaki = %d, inne = %d\n", nwhite, nother);
}
```

Wynikiem tego programu uruchomionego dla własnego tekstu źródłowego jest wiersz

```
cyfry = 9 3 0 0 0 0 0 0 0 1, białe znaki = 123, inne = 339
```

Deklaracja

```
int ndigit[10];
```

mówi, że ndigit jest tablicą 10 liczb całkowitych. Indeksy tablic w języku C zawsze rozpoczynają się od zera, elementami tablicy są zatem ndigit[0], ndigit[1], ..., ndigit[9]. Odzwierciedlają to pętle for, z których jedna inicjuje, a druga wypisuje elementy tablicy.

Indeks może być dowolnym wyrażeniem o wartości całkowitej, zawierającym zmienne całkowite (jak i) oraz stałe całkowite.

Szczególnie ten program wykorzystuje właściwości wewnętrznej reprezentacji cyfr. Na przykład instrukcja

```
if (c >= '0' && c <= '9') ...
```

sprawdza, czy znak w c jest cyfrą. Jeśli tak, to wartością numeryczną tej cyfry będzie

```
c - '0'
```

Takie wyrażenie jest poprawne jedynie wtedy, kiedy znaki '0', '1', ... '9' są kolejnymi wartościami uporządkowanymi rosnąco. Na szczęście tak właśnie jest we wszystkich powszechnie stosowanych zbiorach znaków.

Obiekty typu char są z definicji po prostu niewielkimi liczbami całkowitymi, toteż w wyrażeniach arytmetycznych zmienne i stałe typu char są traktowane identycznie jak int. Jest to całkiem naturalne i wygodne; przykładowo c–'0' jest wyrażeniem całkowitym o wartości między 0 a 9, która odpowiada jednemu ze znaków od '0' do '9' zawartemu w zmiennej c. Wartość ta jest poprawnym indeksem tablicy ndigit.

Decyzję o tym, czy znak jest cyfrą, białym znakiem czy czymkolwiek innym, podejmuje się w następującym fragmencie programu:

```
if (c >= '0' && c <= '9')
    ++ndigit[c-'0'];
else if ( c == ' ' || c == '\n' || c == '\t')
    ++nwhite;
else
    ++nother;
```

Wzorzec

```
if (warunek-1)
    instrukcja-1
else if (warunek-2)
    instrukcja-2
...
    ...
else
    instrukcja-n
```

pojawia się ciągle w programach jako sposób wyrażania decyzji wielowariantowych. *Warunki* są obliczane kolejno, począwszy od najwyższego, aż zostanie znaleziony *warunek* prawdziwy, po czym wykonuje się związaną z nim *instrukcję* i resztę się pomija. (Każda *instrukcja* może być zbudowana z kilku instrukcji zawartych w nawiasach klamrowych.) Niespełnienie żadnego z warunków spowoduje wykonanie instrukcji znajdującej się po ostatnim słowie else, o ile takie występuje. Jeśli końcowe else i *instrukcja* są opuszczone, jak w programie zliczania słów, to nie podejmuje się żadnej akcji. Między początkowym if i końcowym else może wystąpić dowolna liczba zestawów

```
else if (warunek)
    instrukcja
```

Sposób zapisu tej konstrukcji jest kwestią stylu, radzimy jednak robić to tak, jak pokazaliśmy; gdyby każde wystąpienie if było odsunięte od lewego marginesu wyznaczonego przez poprzednie else, wówczas długa sekwencja takich decyzji wymaszerowałaby poza prawą krawędź strony.

W rozdziale 3 omawiamy instrukcję switch pozwalającą zapisać decyzje wielowariantowe w inny sposób. Jest ona szczególnie użyteczna w sytuacjach, w których sprawdza się przynależność wartości wyrażenia całkowitego lub znakowego do określonego zbioru stałych. Dla porównania w p. 3.4 prezentujemy inną wersję tego programu, zawierającą właśnie instrukcję switch.

Ćwiczenie 1.13. Napisz program tworzący histogram długości słów wejściowych. Łatwiej rysuje się histogram z wykresami poziomymi; pionowa orientacja jest bardziej wymagająca.

Ćwiczenie 1.14. Napisz program tworzący histogram częstości występowania różnych znaków pochodzących z wejścia.

1.7 Funkcje

Funkcja w języku C jest równoważna podprogramowi lub funkcji w Fortranie, a także procedurze lub funkcji w Pascalu. Funkcja jest wygodnym sposobem zamknięcia pewnych obliczeń w „czarnej skrzynce", której później można używać nie dbając o to, jak je zrealizowano. Dzięki dobrze zaprojektowanym funkcjom można zignorować to, *jak* zadanie zostało wykonane; wystarczy wiedzieć *co* będzie zrobione. Język C opracowano tak, aby posługiwanie się funkcjami było łatwe, wygodne i skuteczne; często w programach napotkasz funkcje, których definicje składają się z kilku wierszy i które są wywołane tylko raz – po prostu dla większej czytelności fragmentu programu.

Dotychczas korzystaliśmy jedynie z funkcji, w które zostaliśmy wyposażeni, jak printf, getchar czy putchar; nadszedł czas, aby napisać kilka własnych. W języku C nie występuje operator potęgowania, taki jak ** w Fortranie, mechanizm definiowania funkcji zademonstrujemy więc pisząc funkcję power(m,n), podnoszącą liczbę całkowitą m do potęgi całkowitej dodatniej n. Na przykład wartością wyrażenia power(2,5) jest 32. Funkcja ta nie jest zbyt użyteczna, gdyż oblicza jedynie dodatnie potęgi małych liczb całkowitych, ale dla naszych potrzeb zupełnie wystarczy. (W bibliotece standardowej występuje funkcja pow(x,y), która podnosi wartość x do potęgi y.)
Oto funkcja power i program główny, który ją wywołuje; od razu można więc zobaczyć całą strukturę programu.

```
#include <stdio.h>

int power(int m, int n);

/* testowanie funkcji power */
main()
{
    int i;

    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2,i), power(-3,i));
    return 0;
}
```

```
/* power: podnieś base do potęgi n; n >= 0 */
int power(int base, int n)
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

Ogólnie definicja funkcji ma następującą postać:

```
typ-zwracanej-wartości nazwa-funkcji (deklaracje parametrów, jeśli występują)
{
    deklaracje
    instrukcje
}
```

Definicje funkcji mogą pojawić się w dowolnej kolejności w jednym lub kilku plikach źródłowych, przy czym żadna funkcja nie może być rozdzielona między plikami. Jeśli program źródłowy mieści się w kilku plikach, prawdopodobnie będziesz musiał wykonać więcej czynności przy tłumaczeniu i ładowaniu programu niż w przypadku jednego pliku, który zawiera wszystko naraz. Jest to jednak zagadnienie związane z systemem operacyjnym, a nie właściwość języka. Na razie zakładamy, że obie funkcje mieszczą się w tym samym pliku, ciągle więc obowiązuje wszystko to, czego nauczyłeś się o uruchamianiu programów napisanych w języku C.

W wierszu funkcji main:

```
printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

funkcja power jest wywoływana dwukrotnie. Każde wywołanie przekazuje do niej dwa argumenty i za każdym razem funkcja wraca z liczbą przeznaczoną do sformatowania i wypisania. Wyrażenie power(2,i) ma wartość całkowitą tak samo, jak 2 oraz i. (Nie wszystkie funkcje produkują wartość całkowitą; zajmiemy się tym w rozdz. 4.)

W pierwszym wierszu samej funkcji power:

```
power(int base, int n)
```

są zawarte deklaracje nazw i typów parametrów funkcji oraz typ wyniku zwracanego przez funkcję. Nazwy parametrów używane przez funkcję power są dla niej całkowicie lokalne i są niedostępne dla wszystkich innych funkcji, inne funkcje mogą więc

bezkonfliktowo używać tych samych nazw. Dotyczy to także zmiennych i oraz p; zmienna i z funkcji power nie ma nic wspólnego (oprócz nazwy) ze zmienną i w funkcji main.

Zazwyczaj będziemy stosować słowo *parametr* dla nazwanej zmiennej występującej w nawiasach okrągłych w definicji funkcji oraz słowo *argument* dla wartości używanych w wywołaniu funkcji. Czasami dla takiego rozróżnienia używa się zwrotów *argument formalny* i *argument aktualny**.

Obliczona przez funkcję power wartość jest przekazywana do main za pomocą instrukcji return. Po słowie return może wystąpić dowolne wyrażenie:

```
return wyrażenie;
```

Funkcja nie musi zwracać wartości: instrukcja return bez wyrażenia oznacza, że do miejsca wywołania zostanie przekazane jedynie sterowanie – bez wartości użytecznej, tak jak to się dzieje przy „przekroczeniu końca" funkcji po napotkaniu zamykającego nawiasu klamrowego. Funkcja wywołująca może także zignorować zwracaną przez funkcję wartość.

Zauważyłeś pewnie, że na końcu main występuje instrukcja return. Ponieważ main jest taką samą funkcją jak inne, również ona może przekazywać wartość do miejsca swojego wywołania, którym w rzeczywistości jest otoczenie, w jakim wykonuje się program. Zwykle wartość powrotna równa zero oznacza normalne zakończenie działania; wartości różne od zera sygnalizują niezwykłe lub błędne okoliczności tego zakończenia. Dotychczas dla prostoty opuszczaliśmy instrukcje return w naszych funkcjach main, ale będziemy je odtąd umieszczać w programach, aby przypominać o tym, że programy powinny informować otoczenie o swoim stanie.

Deklaracja

```
int power(int m, int n);
```

tuż przed funkcją main informuje, że power jest funkcją, która oczekuje dwóch argumentów typu int i wraca z wartością typu int. Ta deklaracja, zwana *prototypem funkcji*, musi być zgodna z definicją i wywołaniem funkcji power. W przypadku, gdy definicja funkcji lub dowolne jej użycie nie są zgodne z prototypem, jest sygnalizowany błąd.

Nazwy parametrów nie muszą się zgadzać. Tak naprawdę, to w prototypie funkcji są one opcjonalne, a więc nasz prototyp może wyglądać tak:

```
int power(int, int);
```

* Albo: *parametr formalny* i *parametr aktualny*. – *Przyp. tłum.*

Właściwie dobrane nazwy stanowią jednak dobry komentarz, tak że będziemy z nich często korzystać.

A teraz krótka nota historyczna: Największą zmianą różniącą ANSI C od poprzednich wersji języka jest sposób deklarowania i definiowania funkcji. Zgodnie z oryginalną definicją języka C funkcja power wyglądałaby tak:

```
/* power: podnieś base do potęgi n; n >= 0 */
/*        (wersja w starym stylu) */
power(base, n)
int base, n;
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

Nazwy parametrów występują w nawiasach okrągłych, a ich typy są zadeklarowane przed otwierającym nawiasem klamrowym. Przyjmuje się, że parametry bez deklaracji są typu int. (Treść funkcji jest taka sama jak poprzednio.)

Deklaracja funkcji power na początku programu wyglądałaby tak:

```
int power();
```

Umieszczanie listy parametrów w deklaracji funkcji było zabronione, toteż kompilator – choćby chciał – nie mógł sprawdzić, czy funkcję power wywołano poprawnie. W rzeczywistości całą deklarację można było w ogóle pominąć, gdyż przyjęto by domyślnie, że power jest funkcją zwracającą wartość typu int.

Nowa składnia prototypu funkcji ułatwia kompilatorowi wykrywanie błędów w liczbie argumentów lub w ich typach. Stary styl deklaracji i definicji nadal obowiązuje w opisie ANSI C, przynajmniej w okresie przejściowym; my jednak mocno polecamy używanie nowej postaci, jeśli tylko Twój kompilator potrafi ją obsłużyć.

Ćwiczenie 1.15. Napisz nową wersję programu przekształcania temperatur z p. 1.2, w której przekształceń tych dokonuje funkcja.

1.8 Argumenty – przekazywanie przez wartość

Jedna z właściwości funkcji języka C może nie być bliska programistom posługującym się pewnymi innymi językami programowania, zwłaszcza zaś Fortranem. W języku C wszystkie argumenty funkcji są przekazywane „przez wartość". Oznacza to, że wywołana funkcja zamiast oryginałów otrzymuje wartości swoich argumentów w zmiennych tymczasowych. Mechanizm ten wprowadza kilka innych właściwości niż występujące przy wywołaniu „przez referencję" w językach takich jak Fortran lub przy parametrach var w Pascalu. Tam wywołany podprogram ma dostęp do oryginału argumentu, a nie do jego kopii.

Podstawowa różnica polega na tym, że w języku C wywołana funkcja nie może bezpośrednio zmienić wartości zmiennej w funkcji wywołującej; może jedynie zmienić swoją prywatną, tymczasową kopię.

Przekazywanie argumentów przez wartość nie jest jednak wadą, lecz zaletą. Zwykle prowadzi do bardziej zwartych programów z kilkoma swobodnymi zmiennymi, gdyż w wywoływanej funkcji parametry mogą być traktowane jak zwyczajnie inicjowane zmienne lokalne. W następnej wersji funkcji power skorzystano właśnie z tej właściwości.

```
/* power: podnieś base do potęgi n; n >= 0; wersja 2 */
int power(int base, int n)
{
    int p;

    for (p = 1; n > 0; --n)
        p = p * base;
    return p;
}
```

Parametrem n posłużono się jako zmienną tymczasową, zmniejszaną stopniowo do zera (pętla for, która biegnie wstecz). Zmienna i nie jest już dłużej potrzebna. Cokolwiek zrobiono ze zmienną n wewnątrz funkcji power, nie ma to żadnego wpływu na wartość argumentu, z którym funkcja ta została wywołana.

Można spowodować, jeśli to konieczne, aby funkcja zmieniała wartość zmiennej w funkcji wywołującej. W tym celu funkcja wywołująca musi przekazać *adres* zmiennej (technicznie *wskaźnik* do zmiennej), a funkcja wywoływana musi zadeklarować

odpowiedni parametr jako wskaźnik i za jego pomocą pośrednio odwoływać się do tej zmiennej. Wskaźniki opiszemy szczegółowo w rozdz. 5.

Z tablicami jest inaczej. Gdy argumentem wywołania funkcji jest nazwa tablicy, wówczas przekazywaną wartością jest położenie (inaczej adres) początku tablicy; elementy tablicy nie są kopiowane. Przez indeksowanie tej wartości funkcja ma dostęp – a więc i możliwość zmiany – do wartości dowolnego elementu tablicy. Jest to tematem następnego punktu.

1.9 Tablice znakowe

Najczęściej używanymi tablicami w języku C są tablice znaków. Aby pokazać sposób korzystania z tych tablic oraz funkcje manipulujące nimi, napiszmy program, który czyta zbiór wierszy i wypisuje najdłuższy. Schemat problemu jest dość prosty:

```
while (istnieje inny wiersz)
    if (jest dłuższy od poprzednio najdłuższego)
        zachowaj go
        zachowaj jego długość
wypisz najdłuższy wiersz
```

Ze schematu jasno wynika, że program w naturalny sposób dzieli się na części. W jednej z nich jest czytany nowy wiersz, w innej sprawdza się jego długość, w jeszcze innej zachowuje wiersz, reszta zaś steruje całym procesem.

Problem dał się ładnie podzielić na zadania, dobrze byłoby więc napisać program w ten sam sposób. Napiszmy zatem najpierw oddzielną funkcję getline, pobierającą z wejścia następny wiersz. Spróbujemy napisać ją tak, aby mogła być używana w innych programach. Jako minimum funkcja getline musi zwracać sygnał o napotkaniu końca pliku. Bardziej pożyteczne byłoby zwracanie długości wiersza lub zera po osiągnięciu końca pliku. Można zaakceptować zero jako sygnał końca pliku, nie jest ono bowiem nigdy poprawną długością wiersza. Każdy wiersz zawiera co najmniej jeden znak; nawet wiersz składający się tylko ze znaku nowego wiersza ma długość 1.

Po stwierdzeniu, że dany wiersz jest dłuższy od poprzednio najdłuższego, należy go gdzieś zapamiętać. Sugeruje to utworzenie funkcji copy przepisującej nowy wiersz w bezpieczne miejsce.

Ponadto jest potrzebny program sterujący funkcjami getline i copy. Oto wynik naszych rozważań.

```
#include <stdio.h>

#define MAXLINE 1000 /* maksymalny rozmiar wiersza */

int getline(char line[ ], int maxline);
void copy(char to[ ], char from[ ]);

/* wypisz najdłuższy wiersz */
main()
{
    int len;              /* długość bieżącego wiersza */
    int max;              /* poprzednia maks. długość */
    char line[MAXLINE];       /* bieżący wiersz z wejścia */
    char longest[MAXLINE]; /* przechowywany maks. wiersz */

    max = 0;
    while ((len = getline(line, MAXLINE)) > 0)
        if (len > max) {
          max = len;
          copy(longest, line);
        }

    if (max > 0)      /* znaleziono wiersz */
        printf("%s", longest);
    return 0;
}

/* getline: wczytaj wiersz do s, podaj jego długość */
int getline(char s[ ], int lim)
{
    int c, i;

    for (i = 0; i<lim-1 && (c = getchar()) != EOF && c != '\n'; ++i)
        s[i] = c;
    if (c == '\n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return i;
}
```

```
/* copy: przepisz from do to;      */
/* to musi być dostatecznie duże */
void copy(char to[ ], char from[ ])
{
    int i;

    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}
```

Funkcje getline i copy są zadeklarowane na początku programu, który z założenia mieści się w jednym pliku źródłowym.

Funkcje main i getline komunikują się między sobą za pomocą pary argumentów i zwracanej wartości. Parametry funkcji getline są zadeklarowane w wierszu

```
int getline(char s[ ], int lim)
```

który mówi, że pierwszy parametr (s) jest tablicą a drugi (lim) liczbą całkowitą. Powodem, dla którego w deklaracji tablicy podaje się jej rozmiar, jest rezerwacja pamięci. W funkcji getline długość tablicy s nie jest potrzebna, ponieważ rozmiar tablicy jest określony w main. Instrukcja return służy funkcji getline do odesłania wartości w miejsce wywołania tak samo, jak w przypadku funkcji power. Przytoczony wiersz deklaruje również, że getline zwraca wartość typu int. Tę część deklaracji można pominąć, gdyż int jest domyślnym typem zwracanej wartości.

Jedne funkcje zwracają wartości użyteczne; inne – jak copy – są stosowane tylko dla efektów ich działania i nie zwracają żadnej wartości. Deklarowanym typem copy jest więc void, który wyraźnie stwierdza, że funkcja nie zwraca żadnej wartości.

Funkcja getline wstawia znak '\0' (znak *null*: nic, którego wartością jest zero) na koniec tworzonej przez siebie tablicy, aby zaznaczyć koniec ciągu znaków. Ta konwencja obowiązuje także dla języka C: gdy w programie źródłowym w języku C pojawi się stała napisowa, np.

```
"ahoj\n"
```

wówczas taką stałą zapamiętuje się jako tablicę zawierającą ciąg znaków stałej i zaznacza jej koniec znakiem '\0'.

Format %s w funkcji printf wymaga, aby odpowiedni argument był ciągiem znaków o takiej właśnie postaci. Funkcja copy także liczy na to, że jej argument wejściowy jest zakończony znakiem '\0' i znak ten przepisuje do argumentu wyjściowego. (Z tego wszystkiego wynika, że '\0' nie jest częścią normalnego tekstu.)

Przy okazji warto jeszcze wspomnieć, że nawet w tak małym programie, jak ten, występują trudne problemy projektowe. Co na przykład powinna zrobić funkcja main po napotkaniu wiersza dłuższego niż podane ograniczenie? Funkcja getline pracuje bezpiecznie, to znaczy przerywa zbieranie znaków po wypełnieniu tablicy nawet wtedy, gdy nie wystąpił znak nowego wiersza. Funkcja main może sprawdzić, czy wiersz był za długi przez zbadanie długości wiersza i ostatnio zapamiętanego znaku, a następnie postąpić według własnego uznania. Dla zwięzłości programu zignorowaliśmy jednak to zagadnienie.

Użytkownik funkcji getline nie jest w stanie z góry przewidzieć, jak długie będą wiersze wejściowe, funkcja sprawdza zatem przepełnienie. Z drugiej strony, użytkownik funkcji copy już zna (lub może poznać) długości napisów, postanowiliśmy więc nie dodawać do niej kontroli błędów.

Ćwiczenie 1.16. Popraw główną funkcję programu szukającego najdłuższego wiersza tak, aby program zawsze poprawnie wypisywał rozmiar dowolnie długich wierszy i tylko tyle tekstu, ile jest możliwe.

Ćwiczenie 1.17. Napisz program wypisujący wszystkie wiersze wejściowe dłuższe niż 80 znaków.

Ćwiczenie 1.18. Napisz program, który będzie pomijać początkowe znaki odstępu i tabulacji oraz usuwać całkowicie białe (puste) wiersze.

Ćwiczenie 1.19. Napisz funkcję reverse(s) odwracającą kolejność znaków tekstu w argumencie s. Zastosuj ją w programie odwracającym kolejno wszystkie wiersze wejściowe.

1.10 Zmienne zewnętrzne i zasięg zmiennych

Takie zmienne w funkcji main, jak line, longest itd., są prywatne lub lokalne dla tej funkcji. Żadna inna funkcja nie może mieć do nich bezpośredniego dostępu, ponieważ są one zadeklarowane wewnątrz funkcji main. To samo dotyczy zmiennych zadeklarowanych w innych funkcjach; np. zmienna i w funkcji getline nie jest związana ze zmienną i w funkcji copy. Każda zmienna lokalna funkcji zaczyna istnieć

dopiero w chwili wywołania funkcji i znika po zakończeniu jej działania. Z tego powodu zmienne takie są zwykle znane jako *automatyczne* (stosując terminologię innych języków programowania). Odtąd będziemy używać określenia automatyczna w odniesieniu do tych zmiennych lokalnych. (W rozdziale 4 omówimy klasę pamięci static, dzięki której zmienne lokalne funkcji zachowują swoje wartości między jej wywołaniami.)

Zmienne automatyczne pojawiają się i znikają razem z wywołaniem funkcji, nie zachowują więc swoich wartości z jednego wykonania do następnego i muszą być jawnie określane od nowa przy każdym wejściu do funkcji. Jeśli ich wartości początkowe nie zostaną określone, to zmienne będą zawierać śmiecie.

Przeciwieństwem zmiennych automatycznych są zmienne *zewnętrzne* w stosunku do wszystkich funkcji. Są to zmienne globalne, dostępne przez nazwę w dowolnej funkcji programu. (Ten mechanizm jest podobny do COMMON w Fortranie lub do deklarowania zmiennych w najbardziej zewnętrznym bloku w Pascalu.) Ze względu na to, że zmienne zewnętrzne są ogólnie dostępne, mogą być stosowane zamiast list argumentów przy przekazywaniu danych między funkcjami. Co więcej, ponieważ zmienne zewnętrzne istnieją stale, a nie pojawiają się i znikają razem z wywołaniem i zakończeniem funkcji, zachowują więc swoje wartości nawet po zakończeniu działania tych funkcji, które nadały im wartości.

Zmienna zewnętrzna musi być *zdefiniowana* – dokładnie jeden raz – na zewnątrz wszystkich funkcji; definicja przydziela jej rzeczywistą pamięć. Taka zmienna musi być także *zadeklarowana* w każdej funkcji, która chce mieć do niej dostęp; deklaracja ustala typ zmiennej. Można to zrobić albo przez jawną deklarację extern, albo niejawnie przez kontekst. Aby dyskusja stała się rzeczowa, napiszmy od nowa program wypisujący najdłuższy wiersz ze zmiennymi line, longest i max jako zmiennymi zewnętrznymi. Wymaga to zmiany wywołań, deklaracji i treści wszystkich trzech funkcji.

```
#include <stdio.h>

#define MAXLINE 1000  /* maksymalny rozmiar wiersza */

int max;              /* poprzednia maks. długość */
char line[MAXLINE];   /* bieżący wiersz z wejścia */
char longest[MAXLINE]; /* przechowywany maks. wiersz */

int getline(void);
void copy(void);
```

```
/* wypisz najdłuższy wiersz; wersja specjalna */
main()
{
    int len;            /* długość bieżącego wiersza */
    extern int max;
    extern char longest[ ];

    max = 0;
    while ((len = getline()) > 0)
        if (len > max) {
            max = len;
            copy();
        }

    if (max > 0)  /* znaleziono wiersz */
        printf("%s", longest);
    return 0;
}

/* getline: wersja specjalna */
int getline(void)
{
    int c, i;
    extern char line[ ];

    for (i = 0; i < MAXLINE-1
            && (c=getchar()) != EOF && c != '\n'; ++i)
        line[i] = c;
    if (c == '\n') {
        line[i] = c;
        ++i;
    }
    line[i] = '\0';
    return i;
}

/* copy: wersja specjalna */
void copy(void)
{
    int i;
    extern char line[ ], longest[ ];

    i = 0;
    while ((longest[i] = line[i]) != '\0')
        ++i;
}
```

Zmienne zewnętrzne dla fukcji main, getline i copy są zdefiniowane na początku programu: ustala się ich typ oraz określa przydzielaną im pamięć. Zewnętrzne definicje są składniowo dokładnie takie same, jak definicje zmiennych lokalnych; ponieważ jednak występują na zewnątrz funkcji, więc wprowadzają zmienne zewnętrzne. Zanim funkcja będzie mogła użyć zmiennej zewnętrznej, nazwa tej zmiennej musi stać się jej znana. Jednym ze sposobów jest napisanie w funkcji deklaracji extern, która ma taką samą postać, jak zwykła deklaracja, z wyjątkiem dodanego słowa kluczowego extern.

W pewnych przypadkach deklarację extern można pominąć. Jeśli definicja zmiennej zewnętrznej pojawia się w pliku źródłowym przed użyciem zmiennej w konkretnej funkcji, to nie ma potrzeby umieszczania w tej funkcji deklaracji extern. Deklaracje te w funkcjach main, getline i copy są więc zbędne. W rzeczywistości definicje wszystkich zmiennych zewnętrznych zwykle umieszcza się na początku pliku źródłowego, a następnie w całym pliku pomija deklaracje extern.

Jeśli program jest zawarty w kilku plikach źródłowych, a zmienna została zdefiniowana np. w *pliku1* i korzysta się z niej w *pliku2* oraz w *pliku3*, to deklaracje extern w *pliku2* i w *pliku3* są niezbędne do połączenia wszystkich wystąpień zmiennej. W praktyce zwykle zbiera się wszystkie deklaracje extern dotyczące zmiennych oraz funkcji i umieszcza je w oddzielnym pliku, historycznie zwanym *nagłówkiem*, jest on bowiem wstawiany na początku każdego pliku źródłowego za pomocą polecenia #include. Typowym zakończeniem nazwy pliku nagłówkowego jest przyrostek .h. Funkcje z biblioteki standardowej są zadeklarowane w takich plikach, np. <stdio.h>. Kwestię tę omówiono szczegółowo w rozdz. 4, a samą bibliotekę w rozdz. 7 i dodatku B.

Specjalne wersje funkcji getline i copy są bezargumentowe, logicznie byłoby więc zaproponować, aby na początku programu ich prototypy miały postać getline( ) i copy( ). Ze względu na zgodność ze starymi programami w języku C w standardzie przyjmuje się jednak, że pusta lista parametrów jest deklaracją w starym stylu, i wyłącza wszelką kontrolę listy argumentów. Dlatego też dla wyraźnego zaznaczenia, że lista parametrów ma być pusta, należy użyć słowa void. Dalszą dyskusję na ten temat kontynuujemy w rozdz. 4.

Powinieneś zauważyć, że w tym rozdziale z rozwagą używamy słów *definicja* i *deklaracja* przy omawianiu zmiennych zewnętrzych. „Definicja" odnosi się do miejsca, w którym zmienna jest faktycznie tworzona, tj. gdzie ma przydzielaną pamięć. „Deklaracja" dotyczy miejsca, w którym określa się naturę zmiennej, lecz nie przydziela jej pamięci.

Przy okazji jeszcze jedna uwaga. Istnieje tendencja do tworzenia wszystkiego z użyciem zmiennych zewnętrznych, wydaje się bowiem, że to upraszcza komunikację – listy argumentów są krótkie, a zmienne są dostępne wszędzie tam, gdzie są potrzebne. Lecz zmienne zewnętrzne występują zawsze – nawet wtedy, gdy ich nie potrzebujesz. Zbyt mocna wiara w zmienne zewnętrzne jest pełna niebezpieczeństw, ponieważ prowadzi

do powstawania programów, w których powiązania między danymi nie są oczywiste: obiekty mogą być zmieniane w nieoczekiwany, a nawet nieumyślny, sposób. Taki program trudno jest zmienić. Druga wersja programu wypisującego najdłuższy wiersz jest gorsza od poprzedniej częściowo z tych powodów, a częściowo dlatego, że psuje ogólność dwóch bardzo pożytecznych funkcji przez wpisanie do nich nazw zmiennych, którymi mają się posługiwać.

Wyczerpaliśmy już to, co umownie można nazwać rdzeniem języka C. Z tej garści klocków można tworzyć wiele pożytecznych programów o niebagatelnych rozmiarach i chyba dobrym pomysłem jest zrobienie w tym celu przerwy. W następujących ćwiczeniach proponujemy napisanie programów trochę bardziej skomplikowanych niż programy prezentowane wcześniej w tym rozdziale.

Ćwiczenie 1.20. Napisz program detab zastępujący znaki tabulacji odpowiednią liczbą znaków odstępu, określoną przez następny punkt tabulacyjny. Przyjmij stały zbiór punktów tabulacyjnych, przypuśćmy co n pozycji. Czy n powinno być zmienną, czy raczej parametrem symbolicznym?

Ćwiczenie 1.21. Napisz program entab, który zastąpi ciągi znaków odstępu przez minimalną liczbę znaków tabulacji i znaków odstępu tak, aby odstępy zostały zachowane. Użyj tych samych punktów tabulacyjnych, co w programie detab. Jeśli zarówno znak tabulacji, jak i jeden znak odstępu wystarczy do osiągnięcia punktu tabulacyjnego, to który z nich powinien mieć pierwszeństwo?

Ćwiczenie 1.22. Napisz program, który „łamie'' długie wiersze wejściowe na dwa lub więcej krótszych wierszy po ostatnim znaku różnym od znaku odstępu i tabulacji, mieszczącym się przed n-tą pozycją wejścia. Zapewnij, aby Twój program postępował inteligentnie z bardzo długimi wierszami oraz wtedy, gdy przed wskazaną pozycją nie wystąpi żaden znak odstępu lub tabulacji.

Ćwiczenie 1.23. Napisz program usuwający wszystkie komentarze z dowolnego programu w języku C. Nie zapomnij o właściwym traktowaniu stałych znakowych i napisowych. Komentarze języka C nie mogą być zagnieżdżone.

Ćwiczenie 1.24. Napisz program, który sprawdza szczątkową poprawność składniową dowolnego programu w języku C, tj. sygnalizuje błędy w rodzaju brakujących nawiasów okrągłych, kwadratowych czy klamrowych. Nie zapomnij także o apostrofach i cudzysłowach oraz o sekwencjach specjalnych i komentarzach. (Ten program byłby bardzo trudny, gdybyś napisał go w pełni ogólnie.)

2 TYPY, OPERATORY I WYRAŻENIA

Zmienne i stałe są podstawowymi obiektami danych, jakimi posługuje się program. Deklaracje wprowadzają potrzebne zmienne oraz ustalają ich typy i ewentualnie ich wartości początkowe. Operatory określają, co należy z tymi obiektami zrobić. Wyrażenia wiążą zmienne i stałe, tworząc nowe wartości. Typ obiektu determinuje zbiór jego wartości i operacje, jakie można na nim wykonać. Oto zagadnienia zawarte w tym rozdziale.

W standardzie ANSI C wprowadzono wiele małych zmian i dodatków do opisu podstawowych typów i wyrażeń. Wszystkie typy całkowite mogą teraz wystąpić w postaci liczby ze znakiem (signed) i bez znaku (unsigned). Wprowadzono notację dla stałych bez znaku oraz dla szesnastkowych stałych znakowych. Można dokonywać operacji zmiennopozycyjnych w pojedynczej precyzji. Wprowadzono także typ long double dla obliczeń w rozszerzonej precyzji. Stałe napisowe można sklejać na etapie kompilacji. Wreszcie po latach formalnie uznane wyliczenia stały się częścią języka. Obiekty można deklarować jako const (stałe), co ma je chronić przed zmianą wartości. Zasady automatycznych wymuszeń wśród typów arytmetycznych rozszerzono tak, aby obowiązywały dla bogatszego zbioru typów.

2.1 Nazwy zmiennych

Chociaż nie mówiliśmy o tym w rozdz.1, to jednak w języku C występuje kilka ograniczeń dotyczących nazw zmiennych i stałych symbolicznych. Nazwy tworzy się z liter i cyfr; pierwszym znakiem nazwy musi być litera. Znak podkreślenia „_" jest traktowany jak litera – czasami przydaje się do poprawienia czytelności długich nazw. Nie rozpoczynaj jednak nazw zmiennych od znaku podkreślenia, gdyż takie nazwy często występują w podprogramach bibliotecznych. W języku C rozróżnia się wielkie i małe litery alfabetu, toteż x oraz X są dwiema różnymi nazwami. Tradycyjnie w programach w języku C nazwy zmiennych pisze się małymi, a nazwy stałych symbolicznych wielkimi literami.

Co najmniej 31 początkowych znaków nazwy wewnętrznej ma znaczenie. Nazwy funkcji i nazwy zmiennych zewnętrznych mogą być ograniczone do mniej niż 31znaków, ponieważ posługują się nimi asemblery i programy ładujące, a na ich działanie język nie ma wpływu. W standardzie przyjęto, że w nazwach zewnętrznych gwarantuje się unikalność tylko 6 początkowych znaków, przy czym nie rozróżnia się wielkich i małych liter alfabetu. Słowa kluczowe, takie jak if, else, int, float itd., są zarezerwowane – nie możesz ich używać jako nazw zmiennych. Wszystkie słowa kluczowe muszą być pisane małymi literami.

Roztropniej jest nadawać zmiennym nazwy, których znaczenie wiąże się z ich zastosowaniem oraz które są na tyle różne, by nie myliły się w druku. My mamy skłonność do nadawania krótkich nazw zmiennym lokalnym (zwłaszcza tym, które sterują wykonaniem pętli) oraz do stosowania długich nazw dla zmiennych zewnętrznych.

2.2 Typy i rozmiary danych

W języku C występuje tylko kilka podstawowych typów danych:

char	jeden bajt, zdolny pomieścić jeden znak z lokalnego zbioru znaków;
int	typ całkowity, zwykle odzwierciedlający naturalny rozmiar liczb całkowitych komputera;
float	typ zmiennopozycyjny pojedynczej precyzji;
double	typ zmiennopozycyjny podwójnej precyzji.

Dodatkowo występuje kilka kwalifikatorów stosowanych razem z tymi podstawowymi typami danych. Kwalifikatory short (krótki) oraz long (długi) odnoszą się do obiektów całkowitych:

```
short int sh;
long  int counter;
```

W takich deklaracjach słowo int może być – i zwykle jest – pominięte.

Typy short i long wprowadzono po to, aby umożliwić posługiwanie się różnymi zakresami liczb całkowitych tam, gdzie to się może przydać. Typ int na ogół odzwierciedla naturalny rozmiar wynikający z architektury danej maszyny. Obiekt typu short często zajmuje 16 bitów, obiekt typu long – 32 bity, a obiekt typu int zarówno 16 bitów, jak i 32 bity. Każdy kompilator może dowolnie wybierać rozmiary odpowiednie dla sprzętu, na jakim działa, pod warunkiem jednak, że stosuje następujące ograniczenia: obiekty typu short i int są co najmniej 16-bitowe, obiekty typu long – co najmniej 32-bitowe, a obiekt short nie może być dłuższy niż int, który z kolei nie może być dłuższy niż long.

Kwalifikatory signed (ze znakiem liczby) i unsigned (bez znaku liczby) można stosować razem z typem char lub dowolnym typem całkowitym. Liczby unsigned są zawsze dodatnie lub równe zero i podlegają regułom arytmetyki modulo 2^n, gdzie n jest liczbą bitów reprezentujących dany typ. A więc, na przykład, gdy obiekt typu char zajmuje 8 bitów, wówczas zmienne typu unsigned char przyjmują wartości od 0 do 255, podczas gdy zmienne typu signed char wartości od −128 do 127 (dla maszyn, w których stosuje się notację uzupełnieniową do dwójki). To, czy zwykły obiekt typu char jest liczbą ze znakiem czy bez znaku, zależy od maszyny, przy czym wartości znaków drukowalnych z założenia są zawsze dodatnie.

Typ long double wprowadza liczbę zmiennopozycyjną o rozszerzonej precyzji. Podobnie jak dla typów całkowitych, rozmiary obiektów zmiennopozycyjnych zależą od implementacji; typy float, double i long double mogą reprezentować jeden, dwa lub trzy odmienne rozmiary obiektów.

W standardowych plikach nagłówkowych <limits.h> i <float.h> zdefiniowano stałe symboliczne dla wszystkich tych rozmiarów danych oraz dla innych właściwości maszyny i kompilatora. Ten temat omawiamy szerzej w dodatku B.

Ćwiczenie 2.1. Napisz program wyznaczający dziedziny wartości zmiennych typu char, short, int i long opatrzonych kwalifikatorami signed i unsigned. Wynik powinien zawierać odpowiednie wartości ze standardowych plików nagłówkowych oraz wartości obliczone bezpośrednio przez program. Trudniejszym zadaniem jest wyliczenie zakresów dla różnych typów zmiennopozycyjnych.

2.3 Stałe

Stała całkowita, taka jak 1234, jest obiektem typu int. W stałej typu long na końcu występuje litera l (el) lub L, jak w 123456789L. Stała całkowita nie mieszcząca się w zakresie typu int jest także traktowana jak stała typu long. W stałych typu unsigned na końcu występuje litera u lub U, a końcówka ul oraz UL oznacza stałą typu unsigned long.

Stałe zmiennopozycyjne zawierają albo kropkę dziesiętną (np. 123.4), albo wykładnik (np. 1e-2), albo obie te części naraz. Typem stałej zmiennopozycyjnej jest double, chyba że końcówka stanowi inaczej. Występująca na końcu litera f lub F oznacza obiekt typu float, a litera l lub L – obiekt typu long double.

Wartość całkowitą – zamiast w postaci dziesiętnej – można przedstawiać również w postaci ósemkowej lub szesnastkowej. Początkowe zero w stałej całkowitej oznacza liczbę ósemkową, natomiast początkowe znaki 0x lub 0X oznaczają liczbę szesnastkową. Zatem na przykład liczbę dziesiętną 31 można przedstawić jako 037 ósemkowo oraz jako 0x1f (bądź 0X1F) szesnastkowo. Umieszczenie litery L na końcu stałej

ósemkowej lub szesnastkowej tworzy z niej stałą typu long, a umieszczenie litery U tworzy stałą bez znaku. Zapis 0XFUL oznacza stałą typu unsigned long o wartości dziesiętnej równej 15.

Stała znakowa jest liczbą całkowitą; taką stałą tworzy jeden znak ujęty w apostrofy, np. 'x'. Jej wartością jest wartość kodu znaku w maszynowym zbiorze znaków. Na przykład w zbiorze znaków ASCII wartością stałej '0' jest 48 – liczba nie mająca nic wspólnego z numeryczną wartością 0. Napisanie '0' zamiast wartości numerycznej, takiej jak 48 (która zależy od zbioru znaków), uniezależnia program od specyficznych wartości i sprawia, że jest on bardziej czytelny. Stałe znakowe uczestniczą w operacjach numerycznych na równi z innymi liczbami całkowitymi, najczęściej jednak są stosowane do porównań z innymi znakami.

Pewne znaki niegraficzne mogą być reprezentowane w stałych znakowych i napisowych przez sekwencje specjalne, takie jak \n (znak nowego wiersza). Sekwencja specjalna wygląda jak dwa znaki, ale reprezentuje tylko jeden znak. Dodatkowo za pomocą takiej sekwencji można utworzyć dowolny bitowy wzorzec bajtu w postaci

'*ooo*'

gdzie *ooo* oznacza jedną, dwie lub trzy cyfry ósemkowe (0...7), bądź w postaci

'\x*hh*'

gdzie *hh* oznacza jedną lub więcej cyfr szesnastkowych (0...9, a...f, A...F). Możemy więc napisać

```
#define VTAB '\013'      /* ASCII: pionowy tabulator */
#define BELL '\007'      /* ASCII: znak alarmu */
```

lub to samo w postaci szesnastkowej

```
#define VTAB '\xb'       /* ASCII: pionowy tabulator */
#define BELL '\x7'       /* ASCII: znak alarmu */
```

Niżej podajemy kompletną listę sekwencji specjalnych języka C:

\a	znak alarmu	\\	znak \
\b	znak cofania	\?	znak zapytania
\f	znak nowej strony	\'	znak apostrofu
\n	znak nowego wiersza	\"	znak cudzysłowu
\r	znak powrotu karetki	*ooo*	liczba ósemkowa
\t	znak tabulacji poziomej	\x*hh*	liczba szesnastkowa
\v	znak tabulacji pionowej		

Stała znakowa '\0' reprezentuje znak o wartości zero, tzw. znak pusty. Często stosuje się ją zamiast liczby 0 dla podkreślenia znakowej natury pewnych wyrażeń, zawsze jednak jej numeryczną wartością jest po prostu 0.

Wyrażenie stałe jest wyrażeniem, w którym występują wyłącznie stałe. Takie wyrażenia mogą być obliczane na etapie kompilacji programu, a nie podczas jego wykonania. W programie mogą występować wszędzie tam, gdzie używa się stałych. Przykładami wyrażeń stałych są

```
#define MAXLINE 1000
char line[MAXLINE+1];
```

oraz

```
#define LEAP 1      /* w roku przestępnym */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

Stała napisowa lub *napis* jest ciągiem złożonym z zera lub więcej znaków, zawartym między znakami cudzysłowu, np.

```
"Jestem napisem"
```

lub

```
"" /* napis pusty */
```

Znaki cudzysłowu nie są częścią napisu, służą jedynie do określenia jego granic. W stałych napisowych można stosować te same sekwencje specjalne, co w stałych znakowych; zapis \" reprezentuje tu jeden znak cudzysłowu. Napisy mogą być sklejane podczas kompilacji programu:

```
"ahoj," " przygodo"
```

jest równoznaczne z

```
"ahoj, przygodo"
```

Ta możliwość jest bardzo użyteczna przy dzieleniu długich napisów na wiersze w programie źródłowym.

Technicznie stała napisowa jest tablicą, której elementami są znaki. Wewnętrzna reprezentacja napisu zawiera na końcu znak '\0', toteż rozmiar fizycznej pamięci przeznaczonej na napis jest o jeden większy niż liczba znaków zawartych między znakami cudzysłowu. Taka reprezentacja oznacza, że praktycznie nie ma żadnego ograniczenia dotyczącego długości tekstów. Programy muszą jednak badać cały tekst, aby określić jego długość. Funkcja strlen(s) ze standardowej biblioteki zwraca długość tekstu w argumencie s, nie licząc końcowego znaku \0. Oto nasza wersja tej funkcji:

```
/* strlen: podaj długość tekstu w s */
int strlen(char s[])
{
    int i;

    i=0;
    while (s[i] != '\0')
        ++i;
    return i;
}
```

Funkcja strlen i inne funkcje manipulujące tekstami są zadeklarowane w standardowym pliku nagłówkowym <string.h>.

Zwróć uwagę na różnicę między stałymi znakowymi a napisami zawierającymi jeden znak: stała 'x' nie oznacza tego samego co "x". Pierwsza jest znakiem o wartości numerycznej kodu litery x w maszynowym zbiorze znaków. Druga zaś jest tablicą znaków, która zawiera jeden znak (literę x) i dodatkowo znak \0.

W języku C zdefiniowano jeszcze jeden rodzaj stałej – *stałą wyliczenia* (ang. *enumeration constant*). Wyliczenie jest listą wartości stałych całkowitych, np.

```
enum boolean { NO, YES }
```

Pierwsza nazwa na liście wyliczenia (między nawiasami klamrowymi) ma wartość 0, następna – wartość 1 i tak dalej, chyba że wystąpi jawnie podana wartość. Jeśli nie podano jawnie wszystkich wartości dla nazw, to kolejne nie sprecyzowane wartości stanowią postęp arytmetyczny, który rozpoczyna się od ostatnio określonej wartości. Tak właśnie jest w drugim z niżej podanych przykładów.

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
               NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };

enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL,
              AUG, SEP, OCT, NOV, DEC };
    /* miesiące: luty jest drugi, marzec – trzeci itd. */
```

Nazwy występujące w różnych wyliczeniach muszą być różne. W tym samym wyliczeniu wartości mogą się powtarzać.

Wyliczenia są wygodnym sposobem kojarzenia stałych wartości z nazwami, przy czym przewagą wyliczeń w stosunku do alternatywnego sposobu #define jest to, że wartości mogą być dla Ciebie generowane. Choć istnieje możliwość deklarowania zmiennych typu enum, to kompilator nie musi sprawdzać, czy to, co przypisujesz ta-

kiej zmiennej, jest poprawną wartością wyliczenia. Zmienne wyliczeniowe dają jednak szansę takiej kontroli, a więc są często lepsze niż #define. Co więcej, program uruchomieniowy (ang. *debugger*) czasem potrafi wypisywać wartości zmiennych wyliczeniowych w ich symbolicznej postaci.

2.4 Deklaracje

Wszystkie zmienne muszą być zadeklarowane przed użyciem, chociaż pewne deklaracje można podać niejawnie przez kontekst. W deklaracji określa się typ, a następnie wymienia jedną lub kilka zmiennych tego typu, np.

```
int   lower, upper, step;
char  c, line[1000];
```

Zmienne można deklarować na wiele sposobów; powyższy przykład równie dobrze może wyglądać następująco:

```
int   lower;
int   upper;
int   step;
char  c;
char  line[1000];
```

Ten drugi sposób zajmuje więcej miejsca, lecz jest wygodny przy dodawaniu komentarza do każdej deklaracji lub dla późniejszych zmian.

W deklaracjach można także nadawać zmiennym wartości początkowe. Gdy po nazwie zmiennej występuje znak = i pewne wyrażenie, wówczas wyrażenie to pełni rolę inicjatora, jak w następujących przykładach:

```
char  esc = '\\';
int   i = 0;
int   limit = MAXLINE+1;
float eps = 1.0e-5;
```

Jeśli zmienna, o której mowa, nie jest automatyczna, to wartość początkową nadaje się jej tylko raz – jak gdyby przed rozpoczęciem wykonywania programu; jej inicjatorem musi być wyrażenie stałe. Zmiennym automatycznym jawnie określone wartości początkowe nadaje się za każdym razem przy wywołaniu funkcji lub przy wejściu do zawierającego je bloku; tutaj inicjatorem może być dowolne wyrażenie. Zmiennym zewnętrznym i statycznym przez domniemanie nadaje się wartość początkową zero.

Zmienne automatyczne bez jawnie określonej wartości początkowej mają wartości przypadkowe (tj. śmiecie).

Kwalifikator const (stały) można zastosować do deklaracji dowolnej zmiennej; mówi on, że wartość tej zmiennej nie będzie zmieniana. Użycie kwalifikatora const w deklaracji tablicy oznacza, że żaden z jej elementów nie ulegnie zmianie.

```
const double e = 2.71828182845905;
const char msg[ ] = "Uwaga: ";
```

Deklarację const można również zastosować do tablicowych parametrów funkcji, by wskazać, że funkcja ma nie zmieniać takiej tablicy:

```
int strlen (const char[ ]);
```

Próba zmiany wartości zmiennej zadeklarowanej jako const kończy się w sposób zależny od implementacji.

2.5 Operatory arytmetyczne

Dwuargumentowymi operatorami arytmetycznymi są +, –, *, / oraz operator dzielenia modulo %. Dzielenie całkowite obcina część ułamkową wyniku. Wyrażenie

```
x % y
```

daje w wyniku resztę z dzielenia x przez y, jest zatem równe zero, gdy y jest podzielnikiem x. Na przykład rok jest przestępny, jeżeli jest podzielny przez 4 i nie dzieli się przez 100 – z wyjątkiem lat podzielnych przez 400 (są przestępne). A więc

```
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
    printf("%d jest rokiem przestępnym\n", year);
else
    printf("%d nie jest rokiem przestępnym\n", year);
```

Operatora % nie można stosować do danych typu float i double. Dla ujemnych argumentów operacji zarówno kierunek zaokrąglania wyniku po obcięciu części ułamkowej przez dzielenie całkowite /, jak i znak liczby, która jest wynikiem dzielenia modulo %, są zależne od maszyny. Akcje podejmowane po wystąpieniu nadmiaru lub niedomiaru także zależą od maszyny.

Dwuargumentowe operatory + i – mają ten sam priorytet. Jest on niższy niż priorytet operatorów *, / oraz %, który z kolei jest niższy niż priorytet jednoargumentowych operatorów + i –. Operatory arytmetyczne są lewostronnie łączne.

W tablicy 2.1 na końcu tego rozdziału podano priorytety i łączność wszystkich operatorów.

2.6 Relacje i operatory logiczne

Operatorami relacji są

```
>  >=  <  <=
```

Wszystkie mają ten sam priorytet. Tuż za nimi – ze względu na priorytet – występują operatory przyrównania

```
==  !=
```

Operatory relacji mają priorytety niższe niż priorytety operatorów arytmetycznych, zatem wyrażenie typu i<lim–1 jest traktowane zgodnie z oczekiwaniami jako i<(lim-1).

Bardziej interesujące są operatory logiczne && i ||. Wyrażenia połączone tymi operatorami oblicza się od lewej strony do prawej, przy czym koniec obliczania następuje natychmiast po określeniu wyniku jako „prawda'' lub „fałsz''. Wiele programów bazuje na tych właściwościach. Przykładem jest pętla wyjęta z wejściowej funkcji getline, którą napisaliśmy w rozdz. 1:

```
for (i=0; i<lim–1 && (c=getchar()) != '\n' && c != EOF; ++i)
    s[i]=c;
```

Przed przeczytaniem następnego znaku należy sprawdzić, czy jest miejsce na zapamiętanie go w tablicy s. Wobec tego sprawdzenie, czy i<lim–1, *musi* być wykonane jako pierwsze. Co więcej, jeśli zabraknie miejsca, to po prostu musimy się zatrzymać, by nie przeczytać następnego znaku.

Podobnie niefortunne byłoby porównanie wartości zmiennej c ze stałą EOF przed wywołaniem funkcji getchar, zatem wywołanie to i przypisanie wartości zmiennej c musi pojawić się przed sprawdzeniem znaku w c.

Priorytet operatora && jest wyższy niż priorytet operatora ||, a oba są niższe niż priorytety operatorów relacji i przyrównania, toteż wyrażenia podobne do

```
i < lim–1 && (c = getchar()) != '\n' && c != EOF
```

nie potrzebują dodatkowych nawiasów. Ponieważ jednak priorytet operatora != jest wyższy niż priorytet operatora przypisania, więc na przykład w zwrocie

```
(c = getchar()) != '\n'
```

nawiasy są niezbędne do osiągnięcia żądanego wyniku, to znaczy najpierw przypisania wartości zmiennej c, a potem przyrównania jej do '\n'.

Z definicji numeryczną wartością wyrażenia logicznego lub relacyjnego jest 1 – jeżeli jest ono prawdziwe, bądź 0 – jeżeli jest fałszywe.

Jednoargumentowy operator negacji ! (wykrzyknik) zamienia argument różny od zera na wartość 0, a argument równy zero – na wartość 1. Na ogół operator ! jest stosowany w konstrukcjach podobnych do tej:

```
if (!valid)
```

zamiast

```
if (valid == 0)
```

Trudno orzec, który z tych sposobów jest lepszy. Zdanie z !valid czyta się milej („jeśli niepoprawne"), bardziej skomplikowane konstrukcje mogą jednak być niezrozumiałe.

Ćwiczenie 2.2. Napisz pętlę równoważną z powyższą pętlą for, nie używając operatorów logicznych && i ||.

2.7 Przekształcenia typów

Jeżeli argumentami operatora są obiekty różnych typów, to są one przekształcane do jednego wspólnego typu według kilku reguł. Ogólna zasada mówi, że automatycznie wykonuje się tylko takie przekształcenia, w których argument „ciaśniejszy" jest zamieniany na „obszerniejszy" bez utraty informacji, jak przy przekształceniu liczby całkowitej na zmiennopozycyjną w wyrażeniu f + i. Wyrażenia, które nie mają sensu, np. używanie obiektu typu float do indeksowania tablicy, są niedozwolone. Wyrażenia, w których może wystąpić utrata informacji, jak po przypisaniu wartości o dłuższym typie całkowitym zmiennej krótszego typu czy po zamianie wartości zmiennopozycyjnej na całkowitą, mogą powodować wypisanie komunikatu ostrzegawczego, ale nie są zabronione.

Obiekty typu char są po prostu niewielkimi liczbami całkowitymi, można je więc swobodnie stosować w wyrażeniach arytmetycznych. Zapewnia to znaczną elastyczność przy różnego rodzaju przekształceniach znaków. Ilustracją jednego z nich jest następująca prościutka wersja funkcji atoi, która zamienia ciąg cyfr na jego numeryczny odpowiednik.

```
/* atoi: zamień s na wartość całkowitą */
int atoi(char s[ ])
{
    int i, n;
```

```
    n=0;
    for (i=0; s[i] >= '0' && s[i] <= '9'; ++i)
        n=10 * n + (s[i] - '0');
    return n;
}
```

Jak już wspomnieliśmy w rozdz. 1, wyrażenie

```
s[i] - '0'
```

daje numeryczną wartość cyfry zawartej w s[i], gdyż wartości kodów znaków '0', '1' itd. tworzą ciąg rosnących liczb całkowitych.

Innym przykładem przekształcenia typu char na int jest funkcja lower, która zamienia wielkie litery alfabetu na małe i działa poprawnie tylko *dla zbioru znaków ASCII*. Jeśli znak nie jest wielką literą, to funkcja lower zwraca jego wartość bez zmian.

```
/* lower: zamień c na małą literę; tylko dla ASCII */
int lower(int c)
{
    if (c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
    else
        return c;
}
```

Poprawność działania dla zbioru ASCII wynika z tego, że odległość między kodami odpowiednich wielkich i małych liter jest stała oraz że oba alfabety są ciągłe – między literami A i Z występują jedynie litery. Ta ostatnia właściwość nie obowiązuje jednak w zbiorze znaków EBCDIC, toteż dla tego zbioru nasza funkcja zamieniłaby nie tylko litery.

W standardowym pliku nagłówkowym <ctype.h> (opisanym w dodatku B) zdefiniowano rodzinę funkcji realizujących podobne sprawdzenia i przekształcenia niezależnie od zbioru znaków. Jedną z nich, na przykład, jest funkcja tolower(c), która zwraca wartość małej litery alfabetu określonej przez c, jeśli c jest wielką literą; tolower jest zatem przenośnym zamiennikiem naszej funkcji lower. Podobnie sprawdzenie

```
c >= '0' && c <= '9'
```

można zastąpić ogólniejszym

```
isdigit(c)        /* czy c jest cyfrą */
```

Od tej pory będziemy używać funkcji z pliku nagłówkowego <ctype.h>.

Przekształcanie znaków na liczby całkowite ma jeden słaby punkt. Język C nie określa, czy zmienne typu char są wielkościami ze znakiem liczby czy bez. Gdy typ char jest zamieniany na int, czy kiedykolwiek może powstać liczba ujemna? Odpowiedź zmienia się z maszyny na maszynę, odzwierciedlając różnice w ich architekturze. W niektórych znak, którego skrajnie lewy bit jest równy 1, będzie przekształcony na liczbę ujemną (przez tzw. „powielenie bitu znaku"). W innych typ char awansuje do typu int przez dodanie zer z lewej strony, a więc jest zawsze dodatni.

Definicja języka C gwarantuje, że żaden znak ze standardowego maszynowego zbioru znaków drukowalnych nie będzie ujemny, a więc takie znaki w wyrażeniach będą zawsze wartościami dodatnimi. Lecz dowolne wzorce bitowe zapamiętane w zmiennych znakowych mogą być dla jednych maszyn ujemne, dla innych zaś dodatnie. Ze względu na przenośność oprogramowania, w deklaracjach zmiennych typu char używaj kwalifikatora signed lub unsigned, jeśli masz zamiar przechowywać w nich dane „nieznakowe".

Z definicji wyrażenia relacyjne typu i > j oraz wyrażenia logiczne połączone operatorami && i || będą miały wartość 1 – jeśli są prawdziwe, bądź 0 – jeśli są fałszywe. Zatem w przypisaniu

```
d = c >= '0' && c <= '9'
```

zmienna d otrzyma wartość 1, jeśli c jest cyfrą, lub 0, jeśli nie jest. Funkcje biblioteczne, jak np. isdigit, mogą dla oznaczenia stanu „prawda" zwracać dowolne wartości różne od zera. W instrukcjach takich, jak if, while czy for, nie robi to jednak większej różnicy, gdyż „prawda" w ich częściach warunkowych znaczy tyle samo, co „nie zero".

Niejawne przekształcenia arytmetyczne działają zgodnie z oczekiwaniami. Ogólnie, jeśli argumenty operatora dwuargumentowego (np. + czy *) są różnego typu, to typ „mniejszy" jest *promowany* do typu „większego" przed wykonaniem operacji. Typem wyniku jest typ „większy". Szczegółowe reguły przekształceń typów przedstawiono w dodatku w p.A6. Jeśli jednak nie używasz argumentów unsigned, to na razie wystarczy Ci następujący, nieformalny zestaw reguł:

- Jeśli którykolwiek z argumentów jest typu long double, to ten drugi zostanie przekształcony do long double.
- W przeciwnym przypadku, jeśli typem któregokolwiek argumentu jest double, to drugi argument będzie przekształcony do double.
- W przeciwnym przypadku, jeśli typem jednego z argumentów jest float, to drugi argument będzie przekształcony do float.
- W przeciwnym przypadku, wszystkie obiekty typu char i short są przekształcane do int.

- Następnie, jeśli którykolwiek z argumentów ma kwalifikator long, to ten drugi zostanie przekształcony do long.

Zauważ, że w wyrażeniu argumenty typu float nie są automatycznie przekształcane do double – jest to zmiana w stosunku do pierwotnej definicji języka. Zazwyczaj z podwójnej precyzji będą korzystać funkcje matematyczne, takie jak zdefiniowane w pliku nagłówkowym <math.h>. Głównym powodem posługiwania się obiektami typu float jest oszczędność pamięci przy rezerwowaniu dużych tablic lub – rzadziej – oszczędność czasu przy działaniu na maszynach, w których arytmetyka podwójnej precyzji jest szczególnie kosztowna.

Reguły przekształceń są bardziej skomplikowane dla argumentów unsigned. Problem polega na tym, że skutek porównania dwóch obiektów, z których jeden jest liczbą ze znakiem, a drugi bez, zależy od maszyny. Porównania takie zależą bowiem od długości reprezentacji różnych typów całkowitych. Załóżmy na przykład, że typ int zajmuje 16 bitów, a typ long 32 bity. Wtedy -1L < 1U, ponieważ argument 1U jest typu int, zatem w wyrażeniach jest promowany do signed long. Jednocześnie zaś -1L > 1UL, gdyż argument -1L jest promowany do unsigned long i wobec tego ma wygląd wielkiej liczby dodatniej.

Przekształcenia zachodzą również w przypisaniach: wartość prawej strony jest przekształcana do typu lewej strony, który będzie typem wyniku.

Obiekt znakowy zamienia się w liczbę całkowitą z powieleniem bitu znaku lub bez (jak to omówiono wcześniej).

Dłuższe liczby całkowite są przekształcane do krótszych przez odrzucenie wystających bardziej znaczących bitów. Zatem po przypisaniach

```
int i;
char c;

i = c;
c = i;
```

wartość c nie ulegnie zmianie bez względu na to, czy bit znaku jest powielany czy nie. Niemniej jednak odwrotna kolejność przypisań może spowodować utratę informacji.

Jeśli x jest typu float, a i jest typu int, to w obu przypisaniach: x = i oraz i = x nastąpią odpowiednie przekształcenia typów; zamiana float na int spowoduje przy tym obcięcie części ułamkowej. Od implementacji zależy, czy przy przekształcaniu do float obiektu typu double jego wartość zostanie zaokrąglona czy obcięta.

Każdy argument funkcji jest wyrażeniem, toteż przy przekazywaniu funkcjom ich argumentów również zachodzą przekształcenia typów. Gdy nie podano prototypu funkcji, wówczas typy char i short stają się int, a typ float staje się double. Właśnie z tej przyczyny argumenty funkcji deklarowaliśmy jako int lub double nawet wtedy, kiedy wywoływaliśmy ją z argumentami char lub float.

Na koniec, w dowolnym wyrażeniu można jawnie wymusić przekształcenie typów za pomocą jednoargumentowego operatora zwanego *rzutem* (ang. *cast*). W konstrukcji

(*nazwa-typu*) *wyrażenie*

wyrażenie jest przekształcane według podanych reguł do typu określonego przez *nazwa-typu*. Dokładniej mówiąc, rzut działa tak, jak gdyby wartość wyrażenia przypisano zmiennej wskazanego typu, a następnie użyto tej zmiennej zamiast całej konstrukcji. Na przykład funkcja biblioteczna sqrt oczekuje argumentu typu double, będzie więc produkować nonsensowne wyniki, jeśli przypadkowo otrzyma coś innego. (Funkcję sqrt zdefiniowano w pliku nagłówkowym <math.h>.) Tak więc, jeżeli n jest liczbą całkowitą, możemy napisać

```
sqrt((double) n)
```

aby wartość n przekształcić do double przed przekazaniem jej do funkcji sqrt. Zapamiętaj, że rzut produkuje *wartość* argumentu n o odpowiednim typie; sam argument n nie ulega zmianie. Operator rzutowania ma ten sam priorytet, co inne operatory jednoargumentowe (zajrzyj do tablicy na końcu tego rozdziału).

Jeśli argumenty zadeklarowano w prototypie funkcji, tak jak to należy robić prawidłowo, taka deklaracja powoduje automatyczne wymuszenie zmiany typu dla dowolnego argumentu, z którym funkcja została wywołana. Tak więc, przy danym prototypie funkcji

```
double sqrt(double);
```

jej wywołanie

```
root2 = sqrt(2); /* pierwiastek kwadratowy z 2 */
```

automatycznie wymusza zamianę całkowitej liczby 2 na zmiennopozycyjną wartość podwójnej precyzji 2.0, a zatem rzutowanie nie jest potrzebne.

Biblioteka standardowa zawiera przenośne wersje generatora liczb pseudo-losowych i funkcji inicjującej zarodek (ang. *seed*) dla ciągu takich liczb. Pierwsza z funkcji ilustruje zastosowanie rzutu:

```
unsigned long int next = 1;

/* rand: daj pseudo-losowo liczbę całkowitą z przedziału 0..32767 */
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int) (next/65536) % 32768;
}

/* srand: ustal zarodek dla funkcji rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

Ćwiczenie 2.3. Napisz funkcję htoi(s), która zamieni ciąg cyfr szesnastkowych na równoważną mu liczbę całkowitą. W funkcji należy uwzględnić, że ciąg może zaczynać się od 0x lub 0X. Dozwolonymi cyframi szesnastkowymi są: cyfry od 0 do 9 i litery od a do f oraz od A do F.

2.8 Operatory zwiększania i zmniejszania

Język C oferuje dwa niezwykłe operatory zwiększania i zmniejszania wartości zmiennych. Operator zwiększania ++ dodaje 1 do swojego argumentu, podczas gdy operator zmniejszania −− odejmuje 1. Już przedtem często używaliśmy operatora ++ do zwiększania wartości zmiennych, np.

```
if (c == '\n')
    ++nl;
```

Niezwykłe jest to, że operatory ++ i −− mogą być używane zarówno jako przedrostkowe (przed zmienną: ++n), jak i przyrostkowe (po zmiennej: n++). W obu przypadkach wynikiem jest zmiana wartości zmiennej n, ale wyrażenie ++n zwiększa n *przed* użyciem jej wartości, natomiast wyrażenie n++ zwiększa zmienną n *po* użyciu jej poprzedniej wartości. Oznacza to, że w kontekście, w którym jest istotna także wartość (a nie tylko efekt) zwiększenia, wyrażenia ++n oraz n++ są różne. Jeśli n równa się 5, to

```
x = n++;
```

nadaje x wartość 5, ale

```
x = ++n;
```

nadaje x wartość 6. W obu przypadkach wartością zmiennej n stanie się 6. Operatory zwiększania i zmniejszania można stosować tylko do zmiennych. Takie wyrażenia, jak (i+j)++, są błędne.

W kontekście, w którym nie jest potrzebna wartość, a tylko sam efekt zwiększenia, np.

```
if (c == '\n')
    nl++;
```

użycie operatora przedrostkowego i przyrostkowego jest równoważne. Istnieją jednak sytuacje, w których specjalnie stosuje się jeden z tych operatorów. Dla przykładu rozważmy funkcję squeeze(s,c) usuwającą wszystkie wystąpienia znaku c z tekstu zawartego w s.

```
/* squeeze: usuń wszystkie znaki c z s */
void squeeze(char s[ ], int c)
{
    int i, j;

    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}
```

Każde pojawienie się znaku różnego od c spowoduje przepisanie go na bieżącą j-tą pozycję ciągu s, a następnie – i tylko wtedy – zwiększenie zmiennej j, by wskazywała bieżącą pozycję dla następnego znaku. Jest to dokładnie równoważne z konstrukcją

```
if (s[i] != c) {
    s[j] = s[i];
    j++;
}
```

Podobny przykład znajdziemy w funkcji getline, którą napisaliśmy już w rozdz. 1. Konstrukcję

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```

możemy zapisać w bardziej zwartej postaci

```
if (c != '\n')
    s[i++] = c;
```

Trzecim przykładem do rozważenia niech będzie standardowa funkcja strcat(s,t), która dowiązuje tekst pochodzący z argumentu t do końca tekstu w argumencie s. Zakłada ona, że istnieje wystarczająco dużo miejsca, aby pomieścić tekst wynikowy. Tak jak ją tu napisaliśmy, nie zwraca ona żadnej wartości; standardowa wersja biblioteczna funkcji strcat zwraca wskaźnik do wynikowego ciągu znaków.

```
/* strcat: dowiąż tekst z t do końca tekstu w s;
           tablica s musi być dostatecznie duża */
void strcat(char s[ ], char t[ ])
{
    int i, j;

    i = j = 0;
    while (s[i] != '\0') /* znajdź koniec tekstu w s */
        i++;
    while ((s[i++] = t[j++]) != '\0') /* przepisz tekst */
        ;
}
```

Każdy znak z tablicy t jest przepisywany do tablicy s, a więc zastosowanie operatora przyrostkowego do obu zmiennych (i oraz j) zapewnia, że wskażą one odpowiednie pozycje w następnym przebiegu pętli.

Ćwiczenie 2.4. Napisz inną wersję funkcji squeeze(s1,s2) tak, aby z tekstu w argumencie s1 usuwała każdy znak występujący w *tekście* argumentu s2.

Ćwiczenie 2.5. Napisz funkcję any(s1,s2), która zwraca albo pozycję pierwszego wystąpienia dowolnego znaku z s2 w tekście argumentu s1, albo –1, jeśli tekst w s1 nie zawiera żadnego znaku z s2. (Standardowa funkcja biblioteczna strpbrk wykonuje to samo zadanie, ale zwraca wskaźnik do znalezionej pozycji.)

2.9 Operatory bitowe

Język C oferuje sześć operatorów pozwalających na manipulację bitami; można je stosować jedynie do argumentów całkowitych, to znaczy char, short, int oraz long, zarówno ze znakiem liczby, jak i bez znaku. Oto one:

& bitowa koniunkcja (AND),
| bitowa alternatywa (OR),
^ bitowa różnica symetryczna (XOR),
<< przesunięcie w lewo,
>> przesunięcie w prawo,
~ dopełnienie jedynkowe (operator jednoargumentowy).

Bitowy operator koniunkcji & jest często stosowany do „zasłaniania'' pewnego zbioru bitów; na przykład w instrukcji

```
n = n & 0177;
```

zeruje się wszystkie oprócz 7 najmłodszych bitów wartości zmiennej n.

Bitowego operatora alternatywy | używa się do „ustawiania'' bitów; na przykład w instrukcji

```
x = x | SET_ON;
```

ustawia się jedynki na tych bitach w zmiennej x, które w SET_ON są równe 1.

Operator bitowej różnicy symetrycznej ^ ustawia jedynkę na każdej pozycji bitowej tam, gdzie bity w obu argumentach są różne, a zero tam, gdzie są takie same.

Należy zwrócić szczególną uwagę na różnice między bitowymi operatorami & i | a logicznymi operatorami && i ||, które powodują obliczenie (od lewej do prawej) wartości logicznej „prawda'' lub „fałsz''. Dla przykładu: jeśli x równa się 1, a y równa się 2, to x & y ma wartość 0, natomiast x && y ma wartość 1.

Operatory << i >> służą do przesuwania bitów argumentu stojącego po lewej stronie operatora (odpowiednio w lewo i w prawo) o liczbę pozycji określoną przez argument stojący po stronie prawej – jego wartość musi być dodatnia. A więc x<<2 przesuwa wartość x w lewo o dwie pozycje, zwolnione bity wypełniając zerami (operacja ta jest równoważna z mnożeniem przez 4). Przy przesuwaniu w prawo wielkości bez znaku (unsigned) zwolnione bity zawsze są wypełniane zerami. Przesuwanie w prawo wielkości ze znakiem spowoduje na pewnych maszynach wypełnienie tych miejsc bitem znaku (przesunięcie „arytmetyczne''), na innych zaś zerami (przesunięcie „logiczne'').

Jednoargumentowy operator ~ powoduje dopełnienie jedynkowe wartości swego całkowitego argumentu: zamienia każdy bit 1 na 0 i odwrotnie. Na przykład w wyrażeniu

```
x = x & ~077
```

nastąpi wyzerowanie ostatnich sześciu bitów x. Zauważ, że wyrażenie x & ~077 nie zależy od długości słowa maszyny i dlatego powinno być stosowane zamiast na przykład wyrażenia x & 0177700, w którym zakłada się, że x jest wielkością 16-bitową. Taka przenośna postać operacji nie pociąga za sobą dodatkowych kosztów, albowiem

~077 jest wyrażeniem stałym, które może być obliczone podczas fazy tłumaczenia programu.

Aby pokazać kilka operatorów bitowych, rozważmy funkcję getbits(x,p,n), która zwraca n-bitowe pole wycięte z x od pozycji p, dosunięte do prawej strony wyniku. Przyjmujemy tu, że zerową pozycją bitu jest prawy koniec x oraz że n i p są sensownymi wartościami całkowitymi. Na przykład getbits(x,4,3) zwraca trzy bity – z pozycji 4, 3 i 2 – dosunięte do prawej strony wyniku.

```
/* getbits: daj n bitów x od pozycji p */
unsigned getbits(unsigned x, int p, int n)
{
    return (x >> (p+1-n)) & ~(~0 << n);
}
```

Wyrażenie x >> (p+1–n) dosuwa wybrane pole do prawego końca słowa. Zapis ~0 oznacza same jedynki; przesunięcie ich w lewo o n pozycji bitowych (~0<<n) tworzy maskę z zerami na prawych skrajnych n bitach. Dopełnienie jedynkowe tej maski za pomocą operatora ~ tworzy maskę zbudowaną z jedynek na prawych skrajnych n bitach.

Ćwiczenie 2.6. Napisz funkcję setbits(x,p,n,y) zwracającą wartość x, w której n bitów – poczynając od pozycji p – zastąpiono przez n skrajnych bitów z prawej strony y. Pozostałe bity x nie powinny ulec zmianie.

Ćwiczenie 2.7. Napisz funkcję inwert(x,p,n) zwracającą wartość x, w której n bitów – poczynając od pozycji p – zamieniono z 1 na 0 i odwrotnie. Pozostałe bity x nie powinny ulec zmianie.

Ćwiczenie 2.8. Napisz funkcję rightrot(x,n), która zwraca wartość całkowitego argumentu x przesuniętą cyklicznie w prawo o n pozycji bitowych.

2.10 Operatory i wyrażenia przypisania

Wyrażenia podobne do

```
i = i + 2
```

w których zmienna występująca po lewej stronie operatora przypisania = powtarza się natychmiast po prawej stronie, można zapisać w bardziej zwartej postaci

```
i += 2
```

Operator += jest nazywany *operatorem przypisania.*

Dla większości operatorów dwuargumentowych (jak +, który ma lewy i prawy argument) występuje odpowiedni operator przypisania *op*=, gdzie *op* jest jednym z operatorów

```
+  -  *  /  %  <<  >>  &  ^  |
```

Jeśli *wyr1* i *wyr2* są wyrażeniami, to

wyr1 op= wyr2

jest równoważne z

wyr1 = (*wyr1*) *op* (*wyr2*)

przy czym wyrażenie *wyr1* oblicza się tylko raz. Zwróć uwagę na nawiasy otaczające wyrażenie *wyr2*; przypisanie

```
x *= y + 1
```

jest odpowiednikiem

```
x = x * (y + 1)
```

a nie

```
x = x * y + 1
```

Na przykład funkcja bitcount zlicza bitowe jedynki swojego całkowitego argumentu.

```
/* bitcount: policz bity 1 w x */
int bitcount(unsigned x)
{
    int b;

    for (b = 0; x != 0; x >>= 1)
        if (x & 01)
            b++;
    return b;
}
```

Zadeklarowanie argumentu x jako unsigned daje pewność, że podczas przesuwania w prawo – niezależnie od maszyny, na której działa program – zwolnione bity zostaną wypełnione zerami, a nie bitami znaku liczby.

Oprócz zwięzłości operatory przypisania mają jeszcze tę zaletę, że odpowiadają sposobowi myślenia człowieka. Mówimy „dodaj 2 do i" albo „zwiększ i o 2", nie zaś

„weź i, dodaj 2, a następnie umieść wynik z powrotem w i". Dlatego właśnie wyrażenie i += 2 jest lepsze niż i = i + 2. Co więcej, w skomplikowanych wyrażeniach, jak np.

```
yyval[yypv[p3+p4] + yypv[p1+p2]] += 2
```

taki operator przypisania ułatwia zrozumienie programu, Czytelnik nie musi bowiem starannie sprawdzać, czy dwa długie wyrażenia są rzeczywiście identyczne, lub zastanawiać się, dlaczego nie są. Operator przypisania może nawet pomóc kompilatorowi przy generowaniu bardziej efektywnego kodu.

Widzieliśmy już, że przypisanie ma wartość i może pojawić się w wyrażeniach; najczęściej spotykanym przykładem jest

```
while ((c = getchar()) != EOF)
...
```

W wyrażeniach mogą pojawić się także inne operatory przypisania (jak +=, −= itd.), jednak częstość ich występowania jest mniejsza.

We wszystkich takich wyrażeniach typem wyrażenia przypisania jest typ jego lewego argumentu, a jego wartością staje się przypisana wartość.

Ćwiczenie 2.9. W arytmetyce uzupełnieniowej do 2 wyrażenie x &= (x−1) usuwa jedynkę ze skrajnie prawego bitu zmiennej x. Wyjaśnij, dlaczego. Zastosuj tę obserwację do napisania szybszej wersji funkcji bitcount.

2.11 Wyrażenia warunkowe

Instrukcja

```
if (a > b)
    z = a;
else
    z = b;
```

powoduje wstawienie do z większej z liczb a i b. Wyrażenie warunkowe utworzone za pomocą trzyargumentowego operatora „?:" pozwala zapisać tę i podobne konstrukcje w inny sposób. W wyrażeniu

wyr1 ? *wyr2* : *wyr3*

najpierw oblicza się wyrażenie *wyr1*. Jeśli jego wartość jest różna od zera („prawdziwa"), to oblicza się wyrażenie *wyr2* i ta wartość będzie wartością całego wyrażenia warunkowego. W przeciwnym przypadku oblicza się wyrażenie *wyr3* i jego war-

tość stanie się wartością wyniku. Spośród wyrażeń *wyr2* oraz *wyr3* oblicza się tylko jedno. Aby wyznaczyć z jako większą z liczb a i b, wystarczy więc napisać

```
z = (a > b) ? a : b; /* z = max(a,b) */
```

Zwróć uwagę na to, że wyrażenie warunkowe jest prawdziwym wyrażeniem i może być stosowane wszędzie tam, gdzie mogą wystąpić inne wyrażenia. Jeśli *wyr2* i *wyr3* mają różne typy, to typ wyniku będzie określony zgodnie z regułami przekształceń, opisanymi wcześniej w tym rozdziale. Na przykład, jeśli f jest typu float, a n jest typu int, to typem wartości wyrażenia

```
(n > 0) ? f : n
```

będzie float niezależnie od tego, czy wartość n jest dodatnia czy nie.

W wyrażeniu warunkowym nawiasy otaczające pierwsze wyrażenie nie są konieczne, priorytet operatora ?: jest bowiem bardzo niski – tuż nad priorytetem przypisań. Warto jednak stosować nawiasy dla uwypuklenia części warunkowej wyrażenia.

Wyrażenie warunkowe często wpływa na zwięzłość programu. Na przykład poniższa pętla wypisuje n elementów tablicy a, po 10 w każdym wierszu, oddzielając każdą kolumnę odstępem oraz kończąc każdy wiersz (wraz z ostatnim) znakiem nowego wiersza.

```
for (i = 0; i < n; i++)
    printf("%6d%c", a[i], (i%10 == 9 || i == n-1) ? '\n' : ' ');
```

Znak nowego wiersza jest wypisywany po każdym dziesiątym elemencie tablicy oraz po elemencie n-tym. Po wszystkich innych elementach jest wypisywany jeden znak odstępu. Może wygląda to jak sztuczka, ale jest bardziej zwarte niż równoważna konstrukcja zbudowana z if-else. Innym dobrym przykładem jest instrukcja

```
printf("Masz %d część%s.\n", n, n == 1 ? "ć" : "ci");
```

Ćwiczenie 2.10. W nowej wersji funkcji lower, zamieniającej wielkie litery alfabetu na małe, zamiast konstrukcji if-else zastosuj wyrażenie warunkowe.

2.12 Priorytety i kolejność obliczeń

Tablica 2-1 zawiera podsumowanie dotyczące priorytetów i łączności* wszystkich operatorów, także tych, o których dotychczas nie wspominaliśmy. Operatory wymienione w jednym wierszu mają ten sam priorytet. Wiersze są uporządkowane według priorytetu malejącego, a więc np. operatory *, / i % mają ten sam priorytet i jest on wyższy niż priorytet dwuargumentowych operatorów + i –. „Operator" () dotyczy wywołania funkcji. Operatory –> oraz . są używane przy dostępie do elementów

struktury; będą szczegółowo opisane w rozdz. 6 razem z operatorem sizeof (rozmiar obiektu). W rozdziale 5 poznamy jednoargumentowe operatory * (dostęp pośredni za pomocą wskaźnika) oraz & (adres obiektu), a w rozdz. 3 zapoznamy się z operatorem , (przecinek).

Zauważ, że priorytet bitowych operatorów &, ^ oraz | jest niższy niż priorytet operatorów przyrównania == i !=. Wynika z tego, że wyrażenia testujące bity, np.

```
if ((x & MASK) == 0) ...
```

aby dawały poprawne wyniki, muszą zawierać odpowiednią liczbę nawiasów.

Tablica 2-1. Priorytety i łączność operatorów

Operatory	Łączność
() [] -> .	lewostronna
! ~ ++ -- + - * & (*typ*) sizeof	prawostronna
* / %	lewostronna
+ -	lewostronna
<< >>	lewostronna
< <= > >=	lewostronna
== !=	lewostronna
&	lewostronna
^	lewostronna
|	lewostronna
&&	lewostronna
||	lewostronna
?:	prawostronna
= += -= *= /= %= ^= |= <<= >>=	prawostronna
,	lewostronna

Jednoargumentowe operatory +, -, * oraz & mają priorytet wyższy niż ich odpowiedniki dwuargumentowe.

W języku C, jak w większości języków programowania, nie określa się kolejności obliczania wartości argumentów operatora. (Do wyjątków należą operatory &&, ||, ?: oraz ','.) Na przykład w instrukcji

```
x = f() + g();
```

funkcja f może być wykonana przed funkcją g lub odwrotnie. A więc w przypadku, gdy jedna z funkcji (f lub g)zmienia wartość zmiennej, od której zależy ta druga, wartość zmiennej x może zależeć od kolejności wykonania tych funkcji. Aby zapewnić

* Operator jest *lewostronnie* (*prawostronnie*) *łączny*, jeżeli w wyrażeniu zawierającym co najmniej dwa takie operatory na tym samym poziomie struktury nawiasowej najpierw jest wykonywany operator lewy (prawy). Operator jest *łączny*, jeżeli kolejność wykonywania jest dowolna. – *Przyp. tłum.*

szczególną kolejność obliczeń, wyniki pośrednie można przechowywać w zmiennych tymczasowych.

Podobnie kolejność obliczania wartości argumentów funkcji nie jest określona, toteż instrukcja

```
printf("%d %d\n", ++n, power(2,n)); /* ŹLE */
```

może dla różnych kompilatorów produkować różne wyniki zależnie od tego, czy n jest zwiększane przed czy po wywołaniu funkcji power. Rozwiązaniem jest oczywiście napisanie programu inaczej, np.

```
++n;
printf("%d %d\n", n, power(2,n));
```

Wywołania funkcji, zagnieżdżone instrukcje przypisania oraz operatory zwiększania i zmniejszania powodują „efekty uboczne" – przy okazji obliczania wyrażenia pewna zmienna otrzymuje nową wartość. W wyrażeniu powodującym efekty uboczne może pojawić się subtelna zależność od kolejności aktualizacji wartości zmiennych biorących udział w obliczeniach. Ilustracją takiej niefortunnej sytuacji jest instrukcja

```
a[i] = i++;
```

Pytanie brzmi: czy indeks tej tablicy jest starą wartością i czy nową? Kompilatory mogą przetłumaczyć tę instrukcję na wiele sposobów i wygenerować różne odpowiedzi według własnej interpretacji. W standardzie specjalnie nie uściślano tego rodzaju spraw. To, kiedy w wyrażeniu nastąpi efekt uboczny (przypisanie wartości zmiennej), pozostawia się decyzji kompilatora, ponieważ najlepsza kolejność obliczeń ściśle zależy od architektury maszyny. (Jednocześnie w standardzie stwierdzono, że wszystkie efekty uboczne obliczenia argumentów funkcji muszą mieć miejsce przed jej wywołaniem, ale to niewiele pomaga w powyższym wywołaniu funkcji printf.)

Morał z tej dyskusji jest następujący: pisanie programów zależnych od kolejności wykonywania obliczeń należy do złej praktyki programowania w każdym języku. Naturalnie trzeba wiedzieć, czego unikać, lecz jeśli nie wiesz, *jak* pewne rzeczy zostały zrobione na różnych maszynach, nie będzie Cię kusiło, aby wykorzystywać specyfikę konkretnej implementacji.

3 STEROWANIE

W każdym języku programowania instrukcje sterujące ustalają kolejność wykonywania obliczeń. Z większością konstrukcji sterujących spotkaliśmy się już przy omawianiu przykładów. Tu uzupełnimy ich zestaw, a także bardziej szczegółowo opiszemy te, o których już była mowa.

3.1 Instrukcje i bloki

Wyrażenie, takie jak x = 0 czy i++ lub printf(...), staje się *instrukcją*, jeśli jest zakończone średnikiem, np.

```
x = 0;
i++;
printf(...);
```

W języku C średnik jest ogranicznikiem instrukcji, a nie separatorem, jak w językach podobnych do Pascala.

Nawiasy klamrowe { i } są używane do grupowania deklaracji i instrukcji w jedną *instrukcję złożoną*, czyli *blok*, by całość stała się składniowo równoważna jednej instrukcji. Jednym z oczywistych przykładów są nawiasy klamrowe otaczające treść funkcji; innym – nawiasy klamrowe zawierające kilka instrukcji po if, else, while lub for. (Zmienne można deklarować wewnątrz *dowolnego* bloku – omówimy to w rozdz. 4.) Po nawiasie klamrowym zamykającym blok nie może występować średnik.

3.2 Instrukcja **if-else**

Instrukcję if-else stosuje się przy podejmowaniu decyzji. Formalna postać tej instrukcji jest następująca:

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

przy czym część else można pominąć. Najpierw oblicza się *wyrażenie*. Jeśli jest prawdziwe (tzn. jego wartość jest różna od zera), to zostanie wykonana *instrukcja1*. Jeśli zaś jest fałszywe (tzn. wartość *wyrażenia* jest równa zero) oraz jeśli istnieje część else, to zostanie wykonana *instrukcja2*.

Instrukcja if po prostu sprawdza numeryczną wartość wyrażenia, pewne warunki można więc uprościć. Najczęściej pisze się

```
if (wyrażenie)
```

zamiast

```
if (wyrażenie != 0)
```

Czasami jest to naturalne i zrozumiałe, a czasami bywa tajemnicze.

Część else instrukcji if-else nie jest obowiązkowa, toteż może wystąpić pewna dwuznaczność, gdy w ciągu zagnieżdżonych instrukcji if jedna z części else zostanie pominięta. Rozwiązano to w ten sposób, że każda z części else jest przyporządkowana najbliższej z poprzednich instrukcji if nie zawierającej części else. Na przykład w konstrukcji

```
if (n > 0)
    if (a > b)
        z = a;
    else
        z = b;
```

część else jest związana z wewnętrzną instrukcją if, jak to podkreśliliśmy przez wcięcie. Jeśli nie oto chodzi, należy użyć nawiasów klamrowych wymuszających prawidłowe przyporządkowanie:

```
if (n > 0) {
    if (a > b)
        z = a;
}
else
    z = b;
```

Dwuznaczność ta jest szczególnie szkodliwa w sytuacjach podobnych do tej:

```
if (n >= 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf("...");
            return i;
        }
else /* ŹLE */
    printf("Błąd -- n jest ujemne\n");
```

Wcięcie wyraźnie pokazuje, o co Ci chodzi, lecz kompilator o tym nie wie i przyporządkuje else wewnętrznej instrukcji if. Ten rodzaj błędu może być bardzo trudny do wykrycia; przy zagnieżdżonych instrukcjach if dobrym pomysłem jest stosowanie nawiasów klamrowych.

Przy okazji zwróć uwagę na średnik po wyrażeniu z = a w konstrukcji

```
if (a > b)
    z = a;
else
    z = b;
```

Wynika to z faktu, że zgodnie z gramatyką języka C po if następuje *instrukcja*, a instrukcja-wyrażenie, jak z = a;, jest zawsze zakończona średnikiem.

3.3 Konstrukcja **else-if**

Konstrukcja

```
if (wyrażenie)
    instrukcja
else if (wyrażenie)
    instrukcja
else if (wyrażenie)
    instrukcja
else if (wyrażenie)
    instrukcja
else
    instrukcja
```

pojawia się na tyle często, że warto ją omówić oddzielnie. Taki ciąg instrukcji if jest najbardziej ogólnym sposobem zapisania decyzji wielowariantowych. Kolejno oblicza się wartości *wyrażeń*: pierwsze napotkane wyrażenie prawdziwe spowoduje wykonanie związanej z nim *instrukcji* i zakończenie wykonywania całej konstrukcji. Jak zwykle, *instrukcją* może być zarówno jedna instrukcja, jak i grupa instrukcji ujęta w nawiasy klamrowe.

Ostatnia część else oznacza „żaden z powyższych warunków", określa więc akcję podejmowaną w sytuacji, w której wszystkie poprzednie wyrażenia były fałszywe. Czasem taka akcja nie jest jawnie określona, wówczas końcówkę

```
else
    instrukcja
```

można pominąć bądź użyć do sygnalizacji błędu – wystąpienia sytuacji „niemożliwej".

Aby zilustrować decyzję trójwariantową, napiszemy funkcję wyszukiwania metodą bisekcji, która sprawdza, czy dana wartość x występuje w uporządkowanej tablicy v. Elementy tablicy v muszą być uporządkowane rosnąco. Jeśli wartość x jest elementem tablicy v, to funkcja zwróci jego pozycję (liczbę z przedziału od 0 do n–1); w przeciwnym przypadku zwróci –1.

Przy wyszukiwaniu metodą bisekcji najpierw porównuje się daną wartość x ze środkowym elementem tablicy v. Jeśli jest ona mniejsza niż wartość tego elementu, poszukiwanie skupia się na pierwszej – „niższej" – połowie tablicy, jeśli zaś jest większa – na drugiej „wyższej". W obu przypadkach następnym krokiem jest porównanie wartości x ze środkowym elementem wybranej połowy. Ten proces dzielenia terenu poszukiwań na pół powtarza się dopóty, dopóki wartość nie zostanie znaleziona lub okaże się, że teren poszukiwań jest pusty.

Zasadniczą decyzję podejmuje się w zależności od tego, czy wartość x jest mniejsza, większa czy równa wartości środkowego elementu v[mid], wskazanego w każdym kroku pętli. Tego rodzaju rozstrzyganie jest charakterystyczne dla konstrukcji else-if.

```
/* binsearch: szukaj x wśród v[0] <= v[1] <=...<= v[n-1] */
int binsearch(int x, int v[], int n)
{
    int low, high, mid;

    low = 0;
    high = n - 1;
```

```
    while (low <= high) {
        mid = (low+high) / 2;
        if (x < v[mid])
            high = mid - 1;
        else if (x > v[mid])
            low = mid + 1;
        else        /* znaleziono */
            return mid;
    }
    return -1;      /* nie znaleziono */
}
```

Ćwiczenie 3.1. Nasza funkcja wyszukiwania metodą bisekcji wykonuje dwa sprawdzenia wewnątrz pętli, podczas gdy wystarczyłoby jedno (za cenę zwielokrotnienia testów na zewnątrz). Napisz nową wersję tej funkcji z tylko jednym sprawdzeniem wewnątrz pętli i porównaj czasy wykonania obu wersji.

3.4 Instrukcja **switch**

Instrukcja switch służy do podejmowania decyzji wielowariantowych, w których sprawdza się, czy wartość pewnego wyrażenia pasuje do jednej z kilku całkowitych *stałych* wartości, i wykonuje odpowiedni skok.

```
switch (wyrażenie) {
    case wyrażenie-stałe: instrukcje
    case wyrażenie-stałe: instrukcje
    default: instrukcje
}
```

Z każdym wariantem jest związana jedna lub kilka całkowitych wartości stałych bądź wyrażeń stałych. Jeśli jeden z przypadków (ang. *case*) jest zgodny z wartością *wyrażenia*, to od niego rozpocznie się dalsze wykonywanie programu. Wszystkie wartości *wyrażeń-stałych* w przypadkach muszą być różne. Przypadek nazwany default zostanie wykonany wtedy, kiedy żaden inny przypadek nie jest zgodny z wartością *wyrażenia*. Przypadek default nie jest obowiązkowy – jeśli nie występuje, a wartość wyrażenia nie pasuje do innych przypadków, to nie podejmuje się żadnej akcji. Przypadki, łącznie z default, mogą występować w dowolnej kolejności.

W rozdziale 1 napisaliśmy program, w którym częstości pojawiania się cyfr, białych znaków oraz wszystkich innych znaków obliczono za pomocą ciągu instrukcji if ... else if ... else. Oto ten sam program używający instrukcji switch:

```
#include <stdio.h>

main() /* zlicz cyfry, białe znaki, inne */
{
    int c, i, nwhite, nother, ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; i++)
        ndigit[i] = 0;
    while ((c = getchar()) != EOF) {
        switch (c) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            ndigit[c-'0']++;
            break;
        case ' ':
        case '\n':
        case '\t':
            nwhite++;
            break;
        default:
            nother++;
            break;
        }
    }
    printf("cyfry =");
    for (i = 0; i < 10; i++)
        printf(" %d", ndigit[i]);
    printf(", białe znaki = %d, inne = %d\n", nwhite, nother);
    return 0;
}
```

Instrukcja **break** powoduje natychmiastowy wyskok z instrukcji **switch**. Przypadki znaczą tyle co etykiety, toteż po wykonaniu instrukcji związanych z jednym przypadkiem sterowanie *przechodzi do następnego* przypadku, jeśli jawnie nie podejmiesz akcji przerywającej. Użycie instrukcji **break** lub **return** jest najczęściej stosowanym sposobem opuszczenia instrukcji **switch**. Za pomocą instrukcji **break** można także wymusić natychmiastowe wyjście z pętli **while**, **for** i **do**, co będzie opisane dalej w tym rozdziale.

Przechodzenie do następnego przypadku budzi mieszane uczucia. Zaletą jest to, że można definiować wiele różnych przypadków dla jednej akcji, jak w naszym przykładzie z cyframi. Z drugiej strony, każdy „normalny'' przypadek musi być zakończony

instrukcją break dla uniknięcia takiego przejścia. Słabą stroną przechodzenia z jednego przypadku do następnego jest podatność na rozsypanie się całej konstrukcji switch podczas modyfikacji programu. Z wyjątkiem wielu etykiet dla jednej akcji przechodzenie przez przypadki powinno być stosowane oszczędnie i zawsze opatrzone komentarzem.

Do dobrego stylu programowania należy wstawianie break po ostatniej instrukcji ostatniego przypadku (jak po default w naszym przykładzie), mimo że nie jest to konieczne. Pewnego dnia, gdy dopiszesz na końcu jakiś inny przypadek, ta odrobina zapobiegliwości może Cię uratować.

Ćwiczenie 3.2. Napisz funkcję escape(s,t), która przepisując tekst z argumentu t do s, zamienia takie znaki, jak znak nowego wiersza czy tabulacji, na czytelne sekwencje specjalne, np. \n i \t. Zastosuj instrukcję switch. Napisz także funkcję działającą w odwrotnym kierunku, tzn. przy kopiowaniu zamieniającą sekwencje specjalne na rzeczywiste znaki.

3.5 Pętle while i for

Z pętlami while i for już się spotkaliśmy. W pętli

```
while (wyrażenie)
    instrukcja
```

najpierw oblicza się *wyrażenie*. Jeśli jego wartość jest różna od zera, to wykonuje się *instrukcję*, a następnie ponownie oblicza *wyrażenie*. Ten cykl powtarza się do chwili, w której wartość *wyrażenia* stanie się zerem. Wówczas sterowanie przechodzi do instrukcji następującej po pętli.

Pętla

```
for(wyr1; wyr2; wyr3)
    instrukcja
```

jest równoważna rozwinięciu

```
wyr1;
while (wyr2) {
    instrukcja
    wyr3;
}
```

jeśli nie uwzględniać skutku instrukcji continue, którą opiszemy w p. 3.7.

Gramatycznie wszystkie trzy składniki instrukcji for są wyrażeniami. Najczęściej *wyr1* i *wyr3* są przypisaniami lub wywołaniami funkcji, a *wyr2* jest wyrażeniem warunkowym. Dowolną z tych części można pominąć, wówczas jednak musi pozostać jej średnik. Pominięcie *wyr1* lub *wyr3* powoduje usunięcie tej części z rozwinięcia. Jeśli brak warunku *wyr2*, to przyjmuje się, że jest on zawsze prawdziwy, a więc

```
for (;;) {
    ...
}
```

jest pętlą nieskończoną, która prawdopodobnie zostanie przerwana w inny sposób, choćby przez instrukcję break czy return.

Stosowanie pętli while bądź for w dużym stopniu zależy od osobistych upodobań. Na przykład w instrukcji

```
while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
    ; /* przeskocz białe znaki */
```

nie występuje ani część inicjująca, ani część modyfikująca, toteż użycie pętli while jest bardziej naturalne.

Tam, gdzie występują proste części inicjująca i przyrostu, lepiej jest korzystać z pętli for, ponieważ skupia ona *instrukcje* sterujące w widocznym miejscu na szczycie pętli. Najlepiej to widać w instrukcji

```
for (i = 0; i < n; i++)
    ...
```

która jest charakterystycznym zwrotem języka C używanym przy przetwarzaniu początkowych n elementów pewnej tablicy, podobnym do pętli DO w Fortranie lub do pętli for w Pascalu. Podobieństwo nie jest jednak doskonałe, gdyż w języku C wartość zmiennej sterującej oraz granicę pętli for można zmieniać wewnątrz pętli, a zmienna sterująca i zachowuje swoją wartość po zakończeniu pętli niezależnie od przyczyny zakończenia. Ponieważ częściami for są dowolne wyrażenia, stosowanie tej pętli nie ogranicza się do postępów arytmetycznych. Mimo to w złym stylu jest „wpychanie'' do instrukcji for obliczeń, które jako inicjowanie i przyrost nie mają ze sobą nic wspólnego; lepiej te części zarezerwować dla operacji sterujących wykonaniem pętli.

Większym przykładem będzie inna wersja funkcji atoi przekształcającej ciąg cyfr dziesiętnych na jego numeryczny odpowiednik. Wersja ta jest troszeczkę ogólniejsza od napisanej w rozdz. 2 – pomija początkowe białe znaki oraz uwzględnia znak liczby

(nieobowiązkowy + albo –). (W rozdziale 4 pokażemy funkcję atof, która wykonuje takie przekształcenie dla liczb zmiennopozycyjnych.)

Struktura programu odzwierciedla postać danych wejściowych:

pomiń białe znaki, jeśli występują
weź znak liczby, jeśli występuje
weź część całkowitą i przekształć ją

Każdy krok wykonuje swoje zadanie i pozostawia następnemu jasną sytuację. Cały proces kończy się na pierwszym znaku nie należącym do liczby.

```
#include <ctype.h>

/* atoi: zamień s na liczbę całkowitą; wersja 2 */
int atoi(char s[ ])
{
    int i, n, sign;

    for (i = 0; isspace(s[i]); i++) /* przeskocz białe znaki */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-') /* przeskocz znak liczby */
        i++;
    for (n = 0; isdigit(s[i]); i++)
        n = 10 * n + (s[i] - '0');
    return sign * n;
}
```

Standardowa biblioteka zawiera bardziej dopracowaną funkcję strtol zamieniającą teksty na liczby całkowite długie; zajrzyj do punktu 5 w dodatku B.

Korzyści wynikające ze skupienia instrukcji sterujących pętlą w jednym miejscu są bardziej oczywiste, gdy w programie występuje kilka zagnieżdżonych pętli. Następna funkcja porządkuje tablicę liczb całkowitych metodą Shell-sort. Algorytm tej metody opracował D. L. Shell w 1959 roku. Podstawowym jej założeniem jest to, że w fazach początkowych porównuje się elementy oddalone od siebie, a nie sąsiadujące, jak w prostych metodach zamiany. Celem jest szybkie wyeliminowanie dużego bałaganu, aby w późniejszych fazach było mniej do zrobienia. Odległości między porównywanymi elementami zmniejszają się stopniowo do 1 i od tej chwili metoda Shell-sort staje się metodą sąsiednich zamian.

```
/* shellsort: uporządkuj v[0] ... v[n-1] rosnąco */
void shellsort(int v[ ], int n)
{
    int gap, i, j, temp;

    for (gap = n/2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j = i-gap; j>=0 && v[j] > v[j+gap]; j -= gap) {
                temp = v[j];
                v[j] = v[j+gap];
                v[j+gap] = temp;
            }
}
```

Występują tu trzy zagnieżdżone pętle. Najbardziej zewnętrzna steruje odstępem (ang. *gap*) między porównywanymi elementami, zmniejszając go od wartości początkowej n/2 poprzez kolejne dzielenia przez dwa aż do zera. Pętla środkowa kroczy wzdłuż szeregu elementów tablicy, a pętla najbardziej wewnętrzna porównuje każdą parę elementów oddalonych od siebie o gap i zamienia je miejscami, jeśli nie są uporządkowane rosnąco. Na ostatek, gdy zmienna gap osiągnie wartość jeden, wszystkie elementy tablicy będą uporządkowane poprawnie. Zauważ, że ogólność pętli for pozwala na to, aby pętla zewnętrzna miała tę samą postać co pozostałe, chociaż nie jest związana z postępem arytmetycznym.

Ostatnim z operatorów języka C, często stosowanym w instrukcji for, jest przecinek „,". Parę wyrażeń oddzielonych przecinkiem oblicza się od lewej strony do prawej, przy czym typem i wartością wyniku jest typ i wartość prawego argumentu. A zatem w instrukcji for można umieścić kilka takich wyrażeń w różnych jej częściach, na przykład do sterowania równolegle dwoma indeksami. Przedstawiamy to na przykładzie funkcji reverse(s), która odwraca kolejność znaków argumentu s tekstu w miejscu.

```
#include <string.h>

/* reverse: odwróć tekst s w miejscu */
void reverse(char s[ ])
{
    int c, i, j;

    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

Przecinki oddzielające argumenty funkcji, zmienne w deklaracjach itp. *nie są* operatorami i *nie gwarantują* obliczeń od lewej strony do prawej.

Przecinek jako operator powinien być używany oszczędnie. Najbardziej właściwym zastosowaniem są konstrukcje, w których elementy są ściśle ze sobą powiązane, jak w naszej pętli for w funkcji reverse, a także makra, w których wielokrokowe obliczenie musi być pojedynczym wyrażeniem. Wyrażenie przecinkowe byłoby również odpowiednie do odwracania tekstu w funkcji reverse, gdyż zamianę znaków można potraktować jak pojedynczą operację:

```
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
    c = s[i], s[i] = s[j], s[j] = c;
```

Ćwiczenie 3.3. Napisz funkcję expand(s1,s2), która przepisując tekst z argumentu s1 do s2, rozwija skrócone zapisy typu a-z do równoważnych im pełnych ciągów abc...xyz. Uwzględnij wielkie i małe litery alfabetu oraz cyfry. Bądź również przygotowany na obsłużenie zapisów w rodzaju a-b-c oraz a-z0-9, a także -a-z. Przyjmij, że początkowe i końcowe znaki „-'' są traktowane literalnie.

3.6 Pętla do-while

Jak już pisaliśmy w rozdz. 1, warunek zatrzymania w pętlach while i for jest sprawdzany na początku pętli. W trzeciej pętli języka C, do-while, jest przeciwnie: warunek zatrzymania sprawdza się na końcu, *po* każdym obrocie pętli. Pętlę tę zatem zawsze wykonuje się co najmniej raz.

Składnia pętli do jest następująca:

```
do
    instrukcja
while (wyrażenie);
```

Najpierw wykonuje się *instrukcję*, a następnie oblicza *wyrażenie*. Jeżeli jest prawdziwe, to ponownie wykonuje się *instrukcję* i tak dalej. Pętla zostanie zatrzymana wtedy, kiedy *wyrażenie* stanie się fałszywe. Z wyjątkiem interpretacji warunku pętla do-while jest równoważna instrukcji repeat-until w Pascalu.

Doświadczenie mówi, że pętla do-while jest stosowana o wiele rzadziej niż while i for. Niemniej jednak czasami bywa wartościowa, jak w następującej funkcji itoa, która zamienia liczbę na ciąg znaków (przeciwnie do funkcji atoi). Zadanie jest odrobinę bardziej skomplikowane, niż się wydaje na pierwszy rzut oka, ponieważ łatwe

metody generowania cyfr tworzą liczbę w odwrotnej kolejności. My wybraliśmy metodę generowania ciągu cyfr od końca, a następnie odwrócenia go.

```
/* itoa: zamień liczbę n na znaki w tablicy s */
void itoa(int n, char s[ ])
{
    int i, sign;

    if ((sign = n) < 0) /* zanotuj znak liczby */
        n = -n;          /* n będzie dodatnie */
    i = 0;
    do {        /* generuj cyfry w odwrotnym porządku */
        s[i++] = n % 10 + '0';   /* weź następną cyfrę */
    } while ((n /= 10) > 0);     /* usuń ją */
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);   /* odwróć kolejność cyfr */
}
```

Pętla do-while jest tu konieczna, a przynajmniej wygodna, ponieważ co najmniej jeden znak należy wstawić do tablicy s nawet wtedy, kiedy wartością n jest zero. Użyliśmy także nawiasów klamrowych otaczających jedyną instrukcję tworzącą treść pętli do-while. Choć są niepotrzebne, to nieuważny czytelnik na pewno nie pomyli części while z *początkiem* pętli while.

Ćwiczenie 3.4. W notacji uzupełnieniowej do 2 nasza wersja funkcji itoa nie obsłuży największej liczby ujemnej, gdy wartość n jest równa $-(2^{rozmiar_słowa-1})$. Wyjaśnij, dlaczego. Zmień tę funkcję tak, aby wypisywała poprawną wartość liczby niezależnie od maszyny, na której będzie uruchamiana.

Ćwiczenie 3.5. Napisz funkcję itob(n,s,b) zamieniającą wartość całkowitą n na znakową reprezentację liczby w systemie o podstawie b i zapisującą wynik w s. W szczególności itob(n,s,16) przekształca n na postać szesnastkową, którą umieszcza w s.

Ćwiczenie 3.6. Napisz inną wersję funkcji itoa tak, aby akceptowała nie dwa, a trzy argumenty. Niech trzeci argument określa minimalny rozmiar pola; jeśli przekształcona liczba będzie za krótka, to należy ją uzupełnić z lewej strony znakami odstępu.

3.7 Instrukcje **break** i **continue**

Czasami wygodnie jest mieć możliwość kontrolowanego opuszczenia pętli w inny sposób niż przez sprawdzenie warunku na początku lub na końcu pętli. Instrukcja break pozwala na wcześniejsze opuszczenie pętli for, while i do tak samo, jak instrukcji switch. Instrukcja ta powoduje natychmiastowy wyskok z najbardziej zagnieżdżonej pętli lub instrukcji switch, w której występuje.

Następująca funkcja trim usuwa znaki odstępu, tabulacji i nowego wiersza występujące na końcu tekstu, wykorzystując instrukcję break do opuszczenia pętli po wykryciu pierwszego od końca znaku różnego od odstępu, znaku tabulacji i nowego wiersza.

```
/* trim: usuń z s końcowe znaki odstępu, tabulacji, nowego wiersza */
int trim(char s[ ])
{
    int n;

    for (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
    s[n+1] = '\0';
    return n;
}
```

Funkcja strlen zwraca długość tekstu. Pętla for rozpoczyna działanie od ostatniego znaku tekstu i – posuwając się wstecz – kolejno bada każdy znak szukając pierwszego, który nie jest znakiem odstępu, tabulacji i nowego wiersza. Pętlę przerywa się po wykryciu takiego znaku lub wtedy, kiedy n stanie się ujemne (co oznacza, że przebadano cały tekst). Sprawdź, że algorytm jest poprawny nawet wówczas, gdy tekst jest pusty lub zawiera jedynie białe znaki.

Instrukcja continue jest spokrewniona z break, ale stosuje się ją rzadziej; powoduje przerwanie bieżącego i wykonanie od początku następnego kroku zawierającej ją pętli for, while lub do. Dla pętli while i do oznacza to natychmiastowe sprawdzenie warunku zatrzymania, natomiast w pętli for powoduje przekazanie sterowania do części przyrostowej. Instrukcja continue dotyczy jedynie pętli; w instrukcji switch nie ma zastosowania. Instrukcja continue występująca wewnątrz pętli w instrukcji switch powoduje przejście do następnego kroku pętli.

Przykładem może być fragment programu przetwarzającego tylko nieujemne elementy tablicy a; elementy ujemne są pomijane.

```
for (i = 0; i < n; i++) {
    if (a[i] < 0) /* pomiń element ujemny */
        continue;
    ...           /* przetwarzaj element nieujemny */
}
```

Instrukcję continue często stosuje się wtedy, gdy następująca po niej część pętli jest tak skomplikowana, że odwrócenie warunku i tworzenie następnego poziomu może zagnieździć program zbyt głęboko.

3.8 Instrukcja **goto** i etykiety

Język C oferuje „bezgranicznie nadużywaną" instrukcję skoku goto oraz etykiety wskazujące miejsca, do których można skakać. Formalnie instrukcja goto nigdy nie jest konieczna, a w praktyce prawie zawsze można się bez niej obejść. W tej książce dotychczas jej nie używaliśmy.

Niemniej jednak zdarzają się sytuacje, w których instrukcja goto może się przydać. Najczęściej jest ona stosowana do zaniechania przetwarzania w głęboko zagnieżdżonych strukturach programu, na przykład do jednoczesnego przerwania działania dwóch lub więcej pętli. W tym przypadku nie można po prostu użyć instrukcji break, gdyż przerywa tylko jedną z zawierających ją pętli – tę, w której występuje. A więc:

```
for ( ... )
    for ( ... ) {
        ...
        if (niepowodzenie)
            goto error;   /* skocz do obsługi błędów */
    }
...
error:
    /* napraw sytuację lub wypisz komunikat */
```

Ten schemat jest wygodny wówczas, gdy obsługa błędów nie jest banalna oraz gdy błędy mogą być wykryte w wielu miejscach.

Etykieta ma taką samą postać, jak nazwa zmiennej i jest zakończona dwukropkiem. Etykietę można dołączyć do każdej instrukcji w tej samej funkcji, w której występuje instrukcja goto. Zasięgiem etykiety jest więc cała ta funkcja.

Dla przykładu rozważmy problem: jak zbadać, czy w dwóch tablicach a i b występuje taki sam element. Jednym z rozwiązań może być:

```
for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
        if (a[i] == b[j])
            goto found;   /* znaleziony */
/* nie ma takiego elementu */
...
found:   /* znaleziony: a[i] == b[j] */
...
```

Program zawierający instrukcję goto zawsze można napisać bez niej, chociaż prawdopodobnie kosztem kilku powtórzonych sprawdzeń i dodatkowych zmiennych. Powyższy przykład przeszukiwania tablic może zatem wyglądać następująco:

```
found = 0;
for (i = 0; i < n && !found; i++)
    for (j = 0; j < m && !found; j++)
        if (a[i] == b[j])
            found = 1;
if (found)
    /* znaleziony: a[i] == b[j] */
    ...
else
    /* nie ma takiego elementu */
    ...
```

Program, który skonstruowano za pomocą instrukcji goto (z paroma wyjątkami, jak te omówione), zazwyczaj trudniej jest zrozumieć i aktualizować niż program bez tych instrukcji. Nie jesteśmy w tej sprawie dogmatyczni, wydaje się jednak, że jeśli instrukcja goto ma być w ogóle stosowana, to powinna być stosowana rzadko.

4 FUNKCJE I STRUKTURA PROGRAMU

Funkcje pomagają podzielić duże przedsięwzięcia obliczeniowe na mniejsze zadania. Dzięki nim można korzystać z tego, co już zostało przez innych zrobione, zamiast rozpoczynać zawsze od zera. Odpowiednie funkcje ukrywają szczegóły pewnych operacji przed częściami programu, w których znajomość tych szczegółów jest zbędna. Całość jest wówczas bardziej przejrzysta, a ponadto łatwiej wprowadza się zmiany.

Język C opracowano tak, aby posługiwanie się funkcjami było wygodne i skuteczne. Na ogół programy w języku C składają się z wielu małych funkcji, a nie z kilku dużych. Każdy program można umieścić w jednym lub kilku plikach źródłowych. Pliki te mogą być tłumaczone oddzielnie i ładowane razem z funkcjami, które zostały uprzednio przetłumaczone i umieszczone w bibliotekach. Nie będziemy jednak tutaj wnikać w szczegóły tego procesu, zmieniają się one bowiem w zależności od dostępnego systemu.

Deklaracje i definicje funkcji stanowią dziedzinę, w której standard ANSI przeprowadził najbardziej widoczne zmiany. Jak pokazaliśmy w rozdz. 1, można teraz deklarować typy argumentów przy deklaracji funkcji. Zmieniła się też składnia definicji funkcji tak, aby deklaracje i definicje były zgodne. Dzięki temu kompiltor potrafi wykryć dużo więcej błędów niż dotychczas. Co więcej, jeśli parametry zadeklarowano poprawnie, to odpowiednie przekształcenia typów argumentów są wykonywane automatycznie.

W standardzie wyjaśniono reguły dotyczące zasięgu nazw; w szczególności wymaga się dokładnie jednej definicji każdego obiektu zewnętrznego. Inicjowanie stało się ogólniejsze: można teraz nadawać wartości początkowe automatycznym tablicom i strukturom.

Rozszerzono także zakres możliwości preprocesora C. Do nowych właściwości preprocesora należą: pełniejszy zbiór poleceń dla kompilacji warunkowej, możliwość sklejania argumentów tekstowych w makrach oraz lepsza kontrola samego procesu rozwijania makr.

4.1 Wprowadzenie

Na początku opracujemy i napiszemy program wypisujący każdy wiersz wejściowy, w którym występuje określony „wzorzec", czyli pewien ciąg znaków. (Jest to szczególny przypadek usługowego programu grep z systemu Unix.) Na przykład wyszukanie wzorca liter „nie" w zbiorze wierszy:

W co się bawić – w co się bawić?
gdy komis każdy sprzeda ci niewidkę czapkę
i nawet ślepa babka grać przestanie w klasy,
a klasy dawno już nie grają w ślepą babkę.*

spowoduje wypisanie następujących wierszy:

gdy komis każdy sprzeda ci niewidkę czapkę
i nawet ślepa babka grać przestanie w klasy,
a klasy dawno już nie grają w ślepą babkę.

Zasadniczą strukturę zadania da się podzielić zręcznie na trzy części:

```
while (istnieje następny wiersz)
    if (wiersz zawiera wzorzec)
        wypisz ten wiersz
```

Można oczywiście wszystkie te części umieścić w main, lepiej jest jednak skorzystać z naturalnej struktury zadania i z każdej z nich utworzyć osobną funkcję. Trzema małymi kawałkami łatwiej się posługiwać niż jednym dużym: w funkcjach można ukryć nieistotne szczegóły; także prawdopodobieństwo niezamierzonych wzajemnych oddziaływań jest minimalne. Poza tym utworzone funkcje mogą być przydatne do innych celów.

„Dopóki (while) istnieje następny wiersz" to po prostu getline – funkcja, którą napisaliśmy w rozdz. 1. „Wypisz ten wiersz" to funkcja printf, którą ktoś kiedyś dla nas przygotował. Musimy więc napisać jedynie tę funkcję, która zadecyduje, czy wiersz zawiera wzorzec.

* Jest to fragment piosenki Wojciecha Młynarskiego. – *Przyp. tłum.*

Aby rozwiązać ten problem, napiszemy funkcję strindex(s,t) zwracającą pozycję (numer) znaku tekstu zawartego w s, od której rozpoczyna się ciąg znaków t. Ze względu na to, że początkową pozycją w tablicach języka C jest zero, poprawny numer może być równy zero albo dodatni, a zatem wartość ujemna, np. −1, jest wygodnym sygnałem sytuacji awaryjnej. Kiedy w przyszłości będziemy potrzebować bardziej wyrafinowanej metody dopasowywania wzorca, będziemy musieli wymienić jedynie funkcję strindex; reszta programu pozostanie ta sama. (W bibliotece standardowej występuje funkcja strstr, która jest podobna do naszej strindex z tym, że zamiast numeru zwraca wskaźnik.)

Mając do dyspozycji tak opisany projekt, wypełnienie programu szczegółami jest już proste. Podajemy go w całości, możesz więc zobaczyć, jak jego części ze sobą współpracują. Na razie wyszukiwany wzorzec jest dosłownym napisem, co nie jest zbyt ogólnym mechanizmem. Wrócimy wkrótce do dyskusji o inicjowaniu tablic znakowych, a w rozdz. 5 pokażemy, co należy zrobić, aby wzorzec stał się parametrem, to znaczy aby mógł być ustalany przy uruchamianiu programu. Prezentujemy tu także nieznacznie zmienioną wersję funkcji getline; porównanie jej z pierwszą wersją (z rozdz. 1) może być pouczające.

```
#include <stdio.h>
#define MAXLINE 1000   /* maksymalna długość wiersza */

int getline(char line[ ], int max);
int strindex(char source[ ], char searchfor[ ]);

char pattern[ ] = "nie";   /* szukany wzorzec */

/* wypisz wszystkie wiersze zawierające wzorzec */
main( )
{
    char line[MAXLINE];
    int found = 0;

    while (getline(line, MAXLINE) > 0)
        if (strindex(line, pattern) >= 0) {
            printf("%s", line);
            found++;
        }
    return found; /* podaj liczbę znalezionych wierszy */
}
```

```
/* getline: wczytaj wiersz do tablicy s; podaj jego długość */
int getline(char s[ ], int lim)
{
    int c, i;

    i = 0;
    while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}

/* strindex: określ pozycję tekstu t w s lub -1, gdy brak */
int strindex(char s[ ], char t[ ])
{
    int i, j, k;

    for (i = 0; s[i] != '\0'; i++) {
        for (j=i, k=0; t[k] != '\0' && s[j] == t[k]; j++, k++)
            ;
        if (k > 0 && t[k] == '\0')
            return i;
    }
    return -1;
}
```

Każda definicja funkcji ma następujący format:

typ-powrotu nazwa-funkcji (*deklaracje parametrów*)
{
    deklaracje i instrukcje
}

Różne części tej definicji można pominąć; najmniejszą funkcją jest funkcja dummy (atrapa):

```
dummy( )  { }
```

która nic nie robi i nie zwraca żadnej wartości. Takie nic nie robiące fukcje, jak ta, przydają się czasami do zarezerwowania miejsca podczas stopniowego rozwoju pro-

gramu. Jeśli w definicji pominięto *typ-powrotu*, to przyjmuje się, że funkcja zwraca wartość typu int.

Program jest właściwie zbiorem definicji zmiennych i funkcji. Komunikacja między funkcjami odbywa się za pośrednictwem argumentów wywołania funkcji i wartości zwracanych przez funkcje, a także za pośrednictwem zmiennych zewnętrznych. W pliku źródłowym funkcje mogą występować w dowolnej kolejności, a program można podzielić między kilka plików źródłowych pod warunkiem, że żadna z funkcji nie zostanie podzielona.

Instrukcja return jest narzędziem, dzięki któremu wywołana funkcja przekazuje do miejsca wywołania wartość pewnego wyrażenia. Po słowie return może wystąpić dowolne wyrażenie:

return *wyrażenie*;

Jeśli zajdzie taka potrzeba, *wyrażenie* zostanie przekształcone do typu wartości zwracanej przez funkcję. *Wyrażenie* jest często otaczane nawiasami okrągłymi, ale nie jest to konieczne.

Funkcja wywołująca może zignorować zwracaną wartość. Poza tym instrukcja return nie musi zawierać wyrażenia; w takim przypadku do miejsca wywołania nie przekazuje się żadnej wartości. Sterowanie wraca bez wartości także wtedy, kiedy wykonywanie funkcji zakończy się po osiągnięciu nawiasu klamrowego zamykającego funkcję. Nie jest błędem, choć prawdopodobnie oznacza kłopoty, jeśli funkcja zwraca wartość z jednego miejsca, a z innego nie*. W każdym razie „wartością'' funkcji, której nie udało się zwrócić poprawnej wartości, na pewno są śmiecie.

Program wyszukiwania wzorca za pomocą funkcji main zwraca stan osiągnięty po zakończeniu działania, czyli liczbę wierszy pasujących do wzorca. Z tej wartości może skorzystać otoczenie, w którym uruchomiono program.

Techniki tłumaczenia i ładowania programów w języku C, podzielonych między kilka plików źródłowych, zmieniają się zależnie od systemu. Na przykład w systemie Unix zadanie to jest realizowane przez polecenie cc, wspomniane w rozdz. 1. Przypuśćmy, że każda z naszych trzech funkcji jest zapamiętana w osobnym pliku i pliki te nazywają się: main.c, getline.c oraz strindex.c. Wówczas polecenie

```
cc main.c getline.c strindex.c
```

przetłumaczy je, umieści ich wynikowe kody pośrednie (ang. *object code*) w plikach pośrednich main.o, getline.o i strindex.o, a następnie załaduje wszystkie razem do

* Przyzwoite kompilatory wychwytują takie przypadki. – *Przyp. tłum.*

wykonywalnego pliku zwanego a.out. Jeśli wystąpi błąd, np. w main.c, to plik ten można ponownie przetłumaczyć oddzielnie i wynik załadować razem z pozostałymi plikami pośrednimi za pomocą polecenia

```
cc main.c getline.o strindex.o
```

Polecenie cc rozróżnia pliki zgodnie z przyjętą konwencją oznaczania plików źródłowych końcówką „.c", a plików pośrednich końcówką „.o".

Ćwiczenie 4.1. Napisz funkcję strrindex(s,t) zwracającą pozycję *pierwszego od końca* wystąpienia wzorca t w s lub –1, jeśli wzorzec nie występuje w s.

4.2 Funkcje zwracające wartości niecałkowite

Dotychczas w naszych przykładach funkcje albo nie zwracały żadnej wartości (void), albo zwracały wartości typu int. Ale co się dzieje wówczas, gdy funkcja musi zwrócić wartość jakiegoś innego typu? Wiele funkcji numerycznych, jak sqrt, sin czy cos, zwraca wartości typu double; inne wyspecjalizowane funkcje zwracają wartości innych typów. W celu zilustrowania metody postępowania napiszemy i zastosujemy funkcję atof(s), przekształcającą ciąg cyfr zawarty w s na jego zmiennopozycyjny odpowiednik w podwójnej precyzji. Funkcja atof jest rozszerzeniem funkcji atoi, której dwie wersje przedstawiliśmy w rozdz. 2 i 3. Funkcja obsługuje nieobowiązkowy znak liczby i kropkę dziesiętną, a także wykrywa obecność lub brak zarówno części całkowitej, jak i ułamkowej. Nasza wersja *nie* jest najwyższej jakości; taka zajmowałaby dużo więcej miejsca, niż możemy jej poświęcić. Funkcja atof występuje w bibliotece standardowej; plik nagłówkowy <stdlib.h> zawiera jej deklarację.

Po pierwsze, funkcja atof musi sama zadeklarować typ zwracanej przez siebie wartości, gdyż typem tym nie jest int. Nazwa typu poprzedza nazwę funkcji:

```
#include <ctype.h>

/* atof: przekształć ciąg cyfr s na wartość zmiennopozycyjną */
double atof(char s[ ])
{
    double val, power;
    int i, sign;
```

```
    for (i = 0; isspace(s[i]); i++) /* pomiń białe znaki */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')
        i++;
    for (val = 0.0; isdigit(s[i]); i++)
        val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]); i++) {
        val = 10.0 * val + (s[i] - '0');
        power *= 10.0;
    }
    return sign * val / power;
}
```

Po drugie, i równie ważne, funkcja wywołująca musi wiedzieć, że atof zwraca wartość niecałkowitą. Jednym ze sposobów, które to zapewniają, jest jawne zadeklarowanie funkcji atof w funkcji wywołującej. Taką deklarację pokazano w programie prymitywnego kalkulatora biurowego (nadaje się zaledwie do bilansowania książeczki czekowej). Program czyta liczby, każdą w oddzielnym wierszu, być może poprzedzone znakiem liczby, dodaje je i wypisuje bieżącą sumę po każdej przeczytanej liczbie.

```
#include <stdio.h>

#define MAXLINE 100

/* prymitywny kalkulator biurowy */
main()
{
    double sum, atof(char []);
    char line[MAXLINE];
    int getline(char line[], int max);

    sum = 0;
    while (getline(line, MAXLINE) > 0)
        printf("\t%g\n", sum += atof(line));
    return 0;
}
```

Deklaracja

```
double sum, atof(char[ ]);
```

mówi, że zmienna sum jest typu double oraz że atof jest funkcją, która oczekuje jednego argumentu typu char[] (tablicy znakowej) i zwraca wartość typu double.

Deklaracja i definicja funkcji atof muszą być zgodne. Jeśli typy funkcji atof i jej wywołania w main są sprzeczne, a obie funkcje występują w tym samym pliku źródłowym, to kompilator zasygnalizuje błąd. Natomiast gdyby funkcja atof była tłumaczona oddzielnie (co jest bardzo prawdopodobne), to niezgodność ta nie zostałaby wykryta: atof produkowałaby wartości typu double, które funkcja main traktowałaby jako int; otrzymane wyniki nie miałyby sensu.

W świetle tego, co dotychczas powiedzieliśmy na temat deklaracji – że muszą być zgodne z definicją – takie nieporozumienie może wydawać się niespodzianką. Przyczyną niezgodności jest to, że jeśli nie ma prototypu funkcji, to jest ona przez domniemanie deklarowana swoim pierwszym pojawieniem się w wyrażeniu, np.

```
sum += atof(line);
```

Jeżeli nazwa, która nie była uprzednio zadeklarowana, pojawi się w wyrażeniu, a bezpośrednio po niej następuje otwierający nawias okrągły, to jest ona deklarowana przez kontekst jako nazwa funkcji i przez domniemanie przyjmuje się, że funkcja ta zwraca wartość typu int. Nie ma jednak żadnych przesłanek na temat parametrów funkcji. Co więcej, jeśli w deklaracji funkcji nie podano parametrów, np.

```
double atof( );
```

to także przyjmuje się, że o parametrach funkcji atof nie należy robić żadnych założeń; wyłącza się więc całą kontrolę poprawności. Takie specjalne znaczenie pustej listy argumentów w deklaracji funkcji wprowadzono po to, by nowym kompilatorom umożliwić tłumaczenie starszych programów napisanych w języku C. Ale złym pomysłem jest stosowanie tego w nowych programach. Jeśli funkcja wymaga parametrów, zadeklaruj je; jeśli nie – użyj typu void.

Mając poprawnie zadeklarowaną funkcję atof, możemy napisać funkcję atoi (zamiana ciągu cyfr na liczbę całkowitą):

```
/* atoi: zamień ciąg cyfr s na wartość całkowitą; użyj atof */
int atoi(char s[ ])
{
    double atof(char s[ ]);

    return (int) atof(s);
}
```

Zwróć uwagę na strukturę deklaracji i na instrukcję return. Wartość wyrażenia w

return *wyrażenie*;

jest przekształcana do typu funkcji przed wykonaniem instrukcji return. A zatem wartość atof, typu double, zostanie automatycznie przekształcona do int przed wykonaniem tej instrukcji, gdyż typem funkcji atoi jest int. Ta operacja może jednak powodować utratę dokładności, więc pewne kompilatory przed tym ostrzegają. Operator rzutowania (int) wyraźnie stwierdza, że taka operacja jest zamierzona, i likwiduje wszelkie komentarze.

Ćwiczenie 4.2. Uzupełnij funkcję atof tak, aby obsługiwała także „naukową" (wykładniczą) notację o postaci

123.45e-6

w której bezpośrednio po liczbie zmiennopozycyjnej może wystąpić litera e lub E oraz wykładnik ewentualnie ze znakiem liczby.

4.3 Zmienne zewnętrzne

Program w języku C składa się ze zbioru obiektów zewnętrznych, którymi mogą być zarówno zmienne, jak i funkcje. Przymiotnika „zewnętrzny" (ang. *external*) użyto głównie dla kontrastu z przymiotnikiem „wewnętrzny" (ang. *internal*), który opisuje argumenty i zmienne definiowane wewnątrz funkcji. Zmienne zewnętrzne definiuje się poza wszystkimi funkcjami, są więc potencjalnie dostępne dla wielu funkcji. Same funkcje są zawsze zewnętrzne; w języku C nie dopuszcza się definiowania funkcji wewnątrz innej funkcji. Przez domniemanie przyjmuje się, że wszystkie odwołania do zmiennych zewnętrznych i do funkcji za pomocą tej samej nazwy, nawet z funkcji tłumaczonych oddzielnie, są odwołaniami do tego samego obiektu. (W standardzie taka właściwość nazywa się *zewnętrzną łącznością* nazwy.) W tym sensie zmienne zewnętrzne są podobne do bloków COMMON w Fortranie lub do zmiennych zadeklarowanych w najbardziej zewnętrznym bloku w Pascalu. Zobaczymy później, jak definiować zmienne zewnętrzne i funkcje widziane jedynie wewnątrz jednego pliku źródłowego.

To, że zmienne zewnętrzne są ogólnie dostępne, ma znaczenie przy przesyłaniu danych między funkcjami – są one alternatywą dla argumentów funkcji i zwracanych przez nie wartości. Dowolna funkcja może odwołać się do zmiennej zewnętrznej za pomocą jej nazwy, jeżeli tylko ta nazwa jest gdzieś zadeklarowana.

Dla funkcji wymagających dostępu do dużej liczby wspólnych danych zmienne zewnętrzne są wygodniejsze i bardziej skuteczne niż długie listy argumentów. Jak wykazano w rozdz. 1, zmienne zewnętrzne powinny być jednak stosowane z pewną

rozwagą, ponieważ mogą niekorzystnie wpływać na strukturę programu, a także przyczyniać się do powstawania programów ze zbyt wieloma powiązaniami między funkcjami przez dane.

Zmienne zewnętrzne są także użyteczne ze względu na ich zasięg oraz okres ich istnienia. Zmienne automatyczne są dla funkcji wewnętrzne; zaczynają istnieć w chwili wywołania funkcji i nikną po jej zakończeniu. Natomiast zmienne zewnętrzne istnieją stale. Nie pojawiają się i nie znikają, a więc zachowują swoje wartości między jednym a drugim wywołaniem funkcji. Jeśli dwie funkcje muszą mieć kilka wspólnych danych i żadna z nich nie wywołuje drugiej, to często najwygodniej jest trzymać te dane w zmiennych zewnętrznych, zamiast przekazywać je w argumentach.

Bardziej szczegółowo omówimy ten temat na większym przykładzie: napiszemy inny program kalkulatora, który dopuszcza operatory +, –, * oraz /. Zamiast notacji wrostkowej program będzie oparty na Odwrotnej Notacji Polskiej, ponieważ jest ona łatwiejsza do realizacji. (Odwrotna Notacja Polska jest stosowana w wielu kalkulatorach kieszonkowych, a także w takich językach, jak Forth czy Postscript.)

W Odwrotnej Notacji Polskiej każdy operator następuje po swoich argumentach; na przykład wyrażenie wrostkowe

```
(1 – 2) * (4 + 5)
```

jest wprowadzane w postaci

```
1 2 – 4 5 + *
```

Nawiasy nie są potrzebne; notacja jest jednoznaczna dopóty, dopóki wiemy, ilu argumentów spodziewa się każdy operator.

Realizacja tego zadania jest prosta. Każdy argument jest wstawiany na stos. Gdy pojawia się operator, wówczas ze stosu zdejmuje się odpowiednią liczbę argumentów (dwa dla operatorów dwuargumentowych), na nich wykonuje się obliczenie wskazane przez operator, a następnie wynik zapamiętuje z powrotem na stosie. Wyjaśnimy to na ostatnim przykładzie: argumenty 1 i 2 zapamiętujemy na stosie, następnie zastępujemy je przez ich różnicę, tj. –1. Z kolei wstawiamy na stos 4 i 5, i zaraz potem zastępujemy je przez ich sumę 9. Operator mnożenia stosujemy do argumentów –1 i 9, po czym na stosie zastępujemy je ich iloczynem –9. Po osiągnięciu końca danych wejściowych zdejmujemy ze stosu wartość uzyskaną na jego szczycie i wypisujemy ją na wyjście.

Struktura programu jest więc pętlą, w której z chwilą pojawienia się każdego operatora lub argumentu wykonuje się odpowiednią dla niego operację:

```
while (następny operator lub argument nie jest znacznikiem końca pliku)
    if (liczba)
        zapamiętaj ją na stosie
    else if (operator)
        weź argumenty ze stosu
        wykonaj obliczenie
        zapamiętaj wynik na stosie
    else
        błąd
```

Operacje wstawiania na stos i zdejmowania ze stosu są banalne. Ponieważ jednak uzupełniono je wykrywaniem i sygnalizacją błędów, są one na tyle długie, że lepiej napisać dla nich osobne funkcje, zamiast powtarzać stale te same instrukcje. W programie powinna także wystąpić osobna funkcja pobierająca z wejścia następny operator lub argument.

Dotychczas nie rozważyliśmy jeszcze głównego założenia projektowego dotyczącego stosu – gdzie on jest, to znaczy jakie funkcje będą mieć do niego bezpośredni dostęp. Jedną z możliwości jest umieszczenie go w funkcji main i przekazywanie go oraz bieżącej pozycji jego wierzchołka do funkcji realizujących obsługę stosu. Ale funkcja main nie potrzebuje informacji o zmiennych sterujących operacjami na stosie; ona wykonuje jedynie operacje „wstaw" i „zdejmij". Podjęliśmy więc decyzję, aby stos i związane z nim informacje były przechowywane w zmiennych zewnętrznych, dostępnych dla funkcji obsługi stosu push (wstaw) i pop (zdejmij), lecz niedostępnych dla funkcji main.

Napisanie programu dla podanego schematu jest dość proste. Jeśli teraz wyobrazimy sobie, że cały program jest zawarty w jednym pliku źródłowym, to będzie on wyglądał tak:

```
#includes   /* pliki nagłówkowe */
#defines    /* definicje stałych symbolicznych */

deklaracje funkcji wywoływanych w main

main() { ... }

zmienne zewnętrzne używane przez funkcje push i pop

void push(double f) { ... }
double pop(void) { ... }

int getop(char s[]) { ... }

procedury wołane przez getop
```

Później omówimy zagadnienie podziału tego pliku na dwa lub więcej plików źródłowych.

Funkcja main jest pętlą zawierającą ogromną instrukcję switch, która z kolei steruje działaniem kalkulatora zgodnie z typem operatora lub argumentu. Być może jest to bardziej typowe zastosowanie instrukcji switch niż przedstawione w p. 3.4.

```
#include <stdio.h>
#include <stdlib.h> /* dla atof( ) */

#define MAXOP 100 /* max. długość operandu lub operatora */
#define NUMBER '0' /* sygnał znalezienia liczby */

int getop(char [ ]);
void push(double);
double pop(void);
/* kalkulator wg Odwrotnej Notacji Polskiej */
main( )
{
    int type;
    double op2;
    char s[MAXOP];

    while ((type = getop(s)) != EOF) {
        switch (type) {
        case NUMBER:
            push(atof(s));
            break;
        case '+':
            push(pop( ) + pop( ));
            break;
        case '*':
            push(pop( ) * pop( ));
            break;
        case '-':
            op2 = pop( );
            push(pop( ) - op2);
            break;
        case '/':
            op2 = pop( );
            if (op2 != 0.0)
                push(pop( ) / op2);
            else
                printf("błąd: dzielenie przez 0\n");
            break;
```

```
        case '\n':
            printf("\t%.8g\n", pop());
            break;
        default:
            printf("błąd: nieznane polecenie %s\n", s);
            break;
        }
    }
    return 0;
}
```

Operatory dodawania + i mnożenia * są przemienne, kolejność pobierania ich argumentów nie jest więc istotna. Należy natomiast rozróżnić argumenty lewy i prawy dla operatorów odejmowania – i dzielenia /. W instrukcji

```
push(pop() - pop()); /* ŹLE */
```

kolejność, w jakiej wywołuje się obie funkcje pop, nie jest określona. Aby zapewnić poprawną kolejność obliczeń, należy najpierw wartość ze szczytu stosu pobrać do zmiennej tymczasowej, tak jak to zrobiliśmy w main.

```
#define MAXVAL 100 /* maks. głębokość stosu */

int sp = 0;              /* następne wolne miejsce na stosie */
double val[MAXVAL]  /* stos wartości */

/* push: wstaw f na stos */
void push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        printf("błąd: pełen stos; nie można umieścić %g\n", f);
}

/* pop: zdejmij i zwróć wartość ze szczytu stosu */
double pop(void)
{
    if (sp > 0)
        return val[--sp];
    else {
        printf("błąd: pusty stos\n");
        return 0.0;
    }
}
```

Zmienna jest zewnętrzna, jeśli zdefiniowano ją na zewnątrz wszystkich funkcji. Zatem stos i indeks jego wierzchołka – wspólne dla funkcji push i pop – definiujemy na zewnątrz tych funkcji. Ale funkcja main sama nie odwołuje się do stosu lub jego wierzchołka – ich reprezentację można przed nią ukryć.

Zajmiemy się teraz realizacją funkcji getop, która pobiera z wejścia następny operator lub argument. Jej zadanie jest proste. Najpierw pomija wiodące znaki odstępu i tabulacji. Jeśli kolejny znak nie jest cyfrą lub kropką dziesiętną, to zwraca ten znak. W przeciwnym przypadku z następnych cyfr (i być może kropki dziesiętnej) buduje ciąg cyfr liczby i zwraca NUMBER, co jest sygnałem otrzymania liczby.

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getop: pobierz następny operator lub argument */
int getop(char s[ ])
{
    int i, c;

    while ((s[0] = c = getch( )) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    if (! isdigit(c) && c != '.')
        return c;       /* to nie liczba */
    i = 0;
    if (isdigit(c))     /* buduj część całkowitą */
        while (isdigit(s[++i] = c = getch( )))
            ;
    if (c == '.')       /* buduj ułamek */
        while(isdigit(s[++i] = c = getch( )))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}
```

Co robią funkcje getch i ungetch? Często zdarza się, że program czytający dane z wejścia nie może stwierdzić, że przeczytał dość, zanim nie przeczyta za dużo. Dob-

rym tego przykładem jest zbieranie znaków, które tworzą liczbę: póki nie widać znaku różnego od cyfry, liczba nie jest kompletna. Potem jednak okazuje się, że program przeczytał o jeden znak za dużo – znak, na którego przyjęcie nie był jeszcze przygotowany.

Problem byłby rozwiązany, gdyby istniała możliwość „oddawania'' nie chcianego znaku. Wówczas program, który przeczytał o jeden znak za dużo, mógłby go oddać z powrotem na wejście, aby dla reszty programu był nowym, nigdy przedtem nie czytanym znakiem. Na szczęście łatwo można takie oddawanie znaku symulować. Realizują to dwie współpracujące funkcje: getch dostarcza do sprawdzenia następny znak z wejścia, ungetch zapamiętuje oddane z powrotem na wejście znaki tak, aby w kolejnych wywołaniach funkcja getch pobierała je, zanim zacznie czytać nowe znaki z wejścia.

Współpraca tych funkcji jest prosta: ungetch zapamiętuje oddawane znaki we wspólnym buforze – tablicy znaków; getch czyta z bufora wtedy, kiedy coś w nim jest, gdy zaś bufor jest pusty – wywołuje funkcję getchar. Musi istnieć także zmienna indeksująca pozycję bieżącego znaku w buforze.

Ponieważ bufor i indeks znaku są wspólne dla funkcji getch i ungetch, a ponadto muszą zachowywać swoje wartości między wywołaniami tych funkcji, to powinny być zewnętrzne dla nich obu. Możemy więc zdefiniować getch, ungetch i ich wspólne zmienne na przykład tak:

```
#define BUFSIZE 100   /* maks. rozmiar bufora */

char buf[BUFSIZE];    /* bufor na zwroty z ungetch */
int bufp = 0;         /* następne wolne miejsce w buforze */

int getch(void) /* weź znak, być może oddany na wejście */
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c)   /* oddaj znak z powrotem na wejście */
{
    if (bufp >= BUFSIZE)
        printf("ungetch: za wiele zwrotów\n");
    else
        buf[bufp++] = c;
}
```

W bibliotece standardowej występuje funkcja ungetc, która przechowuje tylko jeden zwrócony na wejście znak; omówimy ją w rozdz. 7. Tu do przechowywania zwróconych znaków zastosowaliśmy tablicę znaków, a nie jeden znak, aby zilustrować ogólniejsze podejście do zagadnienia.

Ćwiczenie 4.3. Posługując się podstawowym schematem, łatwo można rozszerzyć funkcje kalkulatora. Dodaj więc obsługę operatora dzielenia modulo % oraz możliwość wprowadzania liczb ujemnych.

Ćwiczenie 4.4. Wprowadź polecenia: wypisywania liczby z wierzchołka stosu bez zdejmowania jej oraz zamiany miejscami dwóch szczytowych elementów stosu. Dodaj też polecenie czyszczące stos.

Ćwiczenie 4.5. Zorganizuj dostęp do bibliotecznych funkcji numerycznych, jak sin, exp czy pow. Zajrzyj do opisu nagłówka <math.h> w dodatku B4.

Ćwiczenie 4.6. Dodaj polecenia pozwalające używać zmiennych. (Łatwo to zrobić dla dwudziestu sześciu zmiennych o nazwach jednoliterowych.) Wprowadź obsługę zmiennej, w której pamięta się wartość ostatnio wypisanej liczby.

Ćwiczenie 4.7. Napisz funkcję ungets(s), która będzie oddawać na wejście cały tekst z argumentu s. Czy funkcja ta powinna coś wiedzieć o obiektach buf i bufp, czy też po prostu można w niej skorzystać z funkcji ungetch?

Ćwiczenie 4.8. Przypuśćmy, że nigdy nie zajdzie potrzeba oddawania na wejście więcej niż jednego znaku. Zgodnie z tym założeniem zmień odpowiednio funkcje getch i ungetch.

Ćwiczenie 4.9. Nasze funkcje getch i ungetch nie obsługują poprawnie znacznika końca pliku EOF. Określ, jakie właściwości powinny mieć te funkcje, żeby EOF mógł być oddawany, a następnie zrealizuj swój projekt.

Ćwiczenie 4.10. Przy innej organizacji program mógłby korzystać z funkcji getline, która czyta z wejścia cały wiersz; wówczas funkcje getch i ungetch w ogóle nie byłyby potrzebne. Rozpatrz na nowo program kalkulatora z zastosowaniem tego podejścia.

4.4 Zasięg nazw

Funkcje i zmienne zewnętrzne, z których składa się program w języku C, nie muszą być tłumaczone wszystkie naraz. Tekst źródłowy programu można umieścić w wielu

plikach, a uprzednio przetłumaczone kawałki mogą być dołączane z bibliotek. Interesują nas odpowiedzi na cztery pytania:

- Jak powinny być zadeklarowane zmienne, aby tłumaczenie było poprawne?
- Jak należy rozmieścić deklaracje w tekście programu, aby wszystkie fragmenty zostały poprawnie połączone podczas jego ładowania?
- Jak poprawnie zorganizować deklaracje, aby dla każdej z nich występowała tylko jedna kopia?
- Jak zmiennym zewnętrznym nadać ich wartości początkowe?

W celu omówienia tych kwestii zreorganizujemy program kalkulatora tak, aby mieścił się w kilku plikach źródłowych. Z praktycznego punktu widzenia program ten jest zbyt mały, żeby zasługiwał na podział, ale jest dobrym przykładem tych problemów, które mogą pojawić się w większych programach.

Zasięgiem nazwy jest ta część programu, wewnątrz której można daną nazwę stosować. Dla zmiennej automatycznej deklarowanej na początku funkcji zasięgiem jest cała funkcja zawierająca deklarację zmiennej. Zmienne lokalne o tej samej nazwie, które występują w różnych funkcjach, nie są ze sobą związane w żaden sposób. To samo dotyczy parametrów funkcji, które w rzeczywistości są zmiennymi lokalnymi.

Zasięg zmiennej zewnętrznej i funkcji rozciąga się od miejsca, w którym została ona zadeklarowana w pliku źródłowym podlegającym kompilacji, do końca tego pliku. Na przykład, jeśli main, sp, val, push i pop są zdefiniowane w jednym pliku w następującej kolejności:

```
main() { ... }

int sp = 0;
double val[MAXVAL];

void push(double f) { ... }

double pop(void) { ... }
```

to funkcje push i pop mogą odwoływać się do zmiennych sp i val po prostu przez nazwy; żadne inne deklaracje nie są potrzebne. Ale nazwy tych zmiennych nie są widoczne dla funkcji main, tak samo zresztą, jak obie funkcje push i pop.

Z drugiej strony, jeśli odwołanie do zmiennej zewnętrznej występuje przed jej definicją lub jeśli jej definicja znajduje się w innym pliku źródłowym niż odwołanie, to deklaracja extern jest obowiązkowa.

Rozróżnienie *deklaracji* zmiennej zewnętrznej i jej *definicji* jest bardzo ważne. Deklaracja informuje o właściwościach zmiennej (przede wszystkim o jej typie); definicja dodatkowo powoduje rezerwację pamięci. Jeśli wiersze

```
int sp;
double val[MAXVAL];
```

występują w programie na zewnątrz wszystkich funkcji, to *definiują* one zmienne zewnętrzne sp i val, powodują przydzielenie wymaganej pamięci, a także są deklaracjami tych zmiennych dla reszty pliku źródłowego. Natomiast wiersze

```
extern int sp;
extern double val[ ];
```

deklarują reszcie pliku źródłowego, że sp jest zmienną typu int, a val jest tablicą o elementach typu double (której rozmiar został gdzieś określony), nie tworzą jednak tych zmiennych ani nie rezerwują dla nich pamięci.

We wszystkich plikach składających się na jeden program źródłowy może wystąpić tylko jedna *definicja* każdej zmiennej zewnętrznej. Pozostałe pliki mogą zawierać deklaracje extern zapewniające dostęp do takiej zmiennej. (Deklarację extern można umieścić także w pliku zawierającym definicję.) Definicje tablic muszą określać ich rozmiar, natomiast w deklaracjach extern rozmiary nie są obowiązkowe.

Inicjowanie zmiennej zewnętrznej jest możliwe jedynie przy jej definicji.

Funkcje push i pop można zdefiniować w jednym pliku źródłowym, a zmienne val i sp zdefiniować i zainicjować w innym (organizacja zupełnie nieodpowiednia dla tego programu). Wówczas, aby powiązać wszystko w całość, należy podać następujące definicje i deklaracje:

W *pliku1*:

```
extern int sp;
extern double val[ ];

void push(double f) { ... }

double pop(void) { ... }
```

W *pliku2*:

```
int sp = 0;
double val[MAXVAL];
```

W *pliku1* deklaracje extern są umieszczone powyżej i na zewnątrz definicji funkcji, dotyczą więc wszystkich funkcji w pliku; jeden zbiór deklaracji wystarcza na całą zawartość *pliku1*. Taka sama organizacja byłaby wymagana w przypadku, gdy definicje zmiennych sp i val występowałyby w tym samym pliku po ich użyciu.

4.5 Pliki nagłówkowe

Rozważmy teraz rozdzielenie fragmentów programu kalkulatora między kilka plików źródłowych tak, jak mogłyby być rozdzielone, gdyby były znacznie większe. Funkcja main mogłaby figurować w jednym pliku nazwanym main.c, funkcje push i pop oraz ich zmienne – w drugim o nazwie stack.c, a funkcja getop – w trzecim, getop.c. Na koniec, czwarty plik źródłowy o nazwie getch.c mógłby zawierać funkcje getch i ungetch; odłączyliśmy je od pozostałych, gdyż w rzeczywistym programie mogłyby pochodzić z oddzielnie przetłumaczonej biblioteki funkcji.

calc.h:

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
    ...
}
```

getop.c:

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop() {
    ...
}
```

stack.c:

```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
    ...
}
double pop(void) {
    ...
}
```

getch.c:

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
    ...
}
void ungetch(int) {
    ...
}
```

Jest jeszcze jedna rzecz, o którą należy się zatroszczyć – wspólne definicje i deklaracje dla wszystkich plików. Chcemy je wszystkie zgromadzić w jednym miejscu tak, aby występowała tylko jedna kopia każdej z nich, teraz i w przyszłości, kiedy będziemy rozwijać program. A zatem te wspólne dane umieścimy w *pliku nagłówkowym* o nazwie calc.h i będziemy go włączać w razie potrzeby. (Wiersz #include jest dokładniej opisany w p. 4.11.) Program wynikowy wygląda więc tak, jak to pokazano na str. 117.

Występuje pewna sprzeczność między wymaganiem, aby z każdego pliku źródłowego mieć dostęp jedynie do tych informacji, które są niezbędne do działania, a praktycznym realizmem, z którego wynika, że trudniej jest utrzymywać porządek w wielu plikach nagłówkowych. Do pewnego średniego rozmiaru programu prawdopodobnie najlepszą metodą jest opracowanie jednego pliku nagłówkowego, w którym mieści się wszystko to, co może być wspólne dla dowolnych dwóch części programu; taką właśnie decyzję podjęliśmy w naszym przykładzie. Dużo większe programy wymagają więcej prac organizacyjnych i większej liczby nagłówków.

4.6 Zmienne statyczne

Zmienne sp i val w pliku źródłowym stack.c oraz zmienne buf i bufp w pliku getch.c zdefiniowano na wyłączny użytek odpowiednich funkcji z tych plików źródłowych; nie zamierzano udostępnić ich innym funkcjom. Deklaracja static zastosowana do zmiennych zewnętrznych i funkcji ogranicza ich zasięg od miejsca wystąpienia do końca tłumaczonego pliku źródłowego. Deklarowanie zewnętrznych obiektów jako static jest więc sposobem na ukrycie ich nazw. Na przykład zmienne buf i bufp muszą być zewnętrzne, aby mogły być wspólne dla pary funkcji getch-ungetch, powinny jednak być niewidoczne dla użytkowników tych funkcji.

Pamięć statyczną określa się przez poprzedzenie normalnej deklaracji słowem kluczowym static. Jeśli obie funkcje i obie zmienne zostaną umieszczone w jednym pliku źródłowym, np.

```
static char buf[BUFSIZE]; /* bufor na zwroty z ungetch */
static int bufp = 0;          /* następne wolne miejsce w buf */

int getch(void) {...}

void ungetch(int c) {...}
```

to żadna inna funkcja nie będzie miała dostępu do zmiennych buf i bufp, a ich nazwy nie będą kolidować z takimi samymi nazwami w innych plikach tego samego progra-

mu. W ten sam sposób można ukryć zmienne sp i val, z których korzystają funkcje push i pop przy obsłudze stosu: wystarczy zadeklarować je jako static.

Zewnętrzną deklarację static najczęściej stosuje się do zmiennych, ale można ją także stosować do funkcji. Nazwy funkcji są zwykle globalne, widoczne dla wszystkich części całego programu. Jeśli jednak funkcję zadeklarowano jako static, jej nazwa staje się niewidzialna poza plikiem zawierającym jej deklarację.

Deklarację static można również stosować do zmiennych wewnętrznych. Wewnętrzne zmienne statyczne są lokalne dla poszczególnych funkcji tak samo, jak zmienne automatyczne. Jednak w przeciwieństwie do automatycznych nie pojawiają się i nie znikają razem z wywołaniem funkcji, lecz istnieją między jej wywołaniami. To znaczy, że wewnętrzne zmienne static stanowią prywatną, stałą pamięć wewnątrz pojedynczej funkcji.

Ćwiczenie 4.11. Zmień funkcję getop tak, aby nie potrzebowała funkcji ungetch.
Rada: użyj wewnętrznej zmiennej static.

4.7 Zmienne rejestrowe

Deklaracja register powiadamia kompilator o tym, że zmienna, której ta deklaracja dotyczy, będzie intensywnie używana. Pomysł polega na tym, aby takie zmienne register umieszczać w rejestrach maszyny, co w efekcie może zmniejszyć i przyspieszyć programy. Kompilatory mogą jednak tę informację zignorować.

Oto typowa deklaracja register:

```
register int x;
register char c;
```

Deklarację register można stosować jedynie do zmiennych automatycznych i do formalnych parametrów funkcji. W tym ostatnim przypadku deklaracja ma postać:

```
f(register unsigned m, register long n)
{
    register int i;
    ...
}
```

W praktyce, przy korzystaniu ze zmiennych rejestrowych występują pewne ograniczenia odzwierciedlające rzeczywiste możliwości dostępnego sprzętu. Każda funkcja może przechowywać w rejestrach tylko kilka zmiennych, ponadto nie wszystkie ich typy są dozwolone. Nadliczbowe deklaracje zmiennych rejestrowych są jednak nieszkodliwe,

gdyż słowo register w deklaracjach nadliczbowych czy też niepoprawnych jest ignorowane. Nie ma także możliwości uzyskania adresu zmiennej rejestrowej (zagadnienie omawiane w rozdz. 5) niezależnie od tego, czy zmienną rzeczywiście umieszczono w rejestrze, czy nie. Szczegółowe ograniczenia dotyczące liczby i typów zmiennych rejestrowych zależą od maszyny.

4.8 Struktura blokowa

Język C nie ma struktury blokowej w sensie Pascala czy jemu podobnych języków, gdyż nie można definiować funkcji wewnątrz innych funkcji. Z drugiej strony, zmienne mogą być definiowane według zasad struktury blokowej wewnątrz funkcji. Deklaracje zmiennych (łącznie z ich inicjowaniem) można umieścić po otwierającym nawiasie klamrowym *dowolnej* instrukcji złożonej, a nie tylko po tym, który rozpoczyna funkcję. Tak zadeklarowane zmienne zasłaniają identycznie nazwane zmienne z bloków zewnętrznych i istnieją do napotkania odpowiedniego zamykającego nawiasu klamrowego. W przykładzie

```
if (n > 0) {
    int i; /* definicja nowego i */

    for (i = 0; i < n; i++)
        ...
}
```

zasięgiem zmiennej i jest gałąź „prawdy'' instrukcji if; ta zmienna i nie jest związana z żadną inną zmienną i występującą poza tym blokiem. Zmienna automatyczna deklarowana i inicjowana wewnątrz bloku otrzymuje swoją wartość początkową za każdym razem od nowa, przy każdym wejściu do bloku. Zmienna static jest inicjowana tylko raz, przy pierwszym wejściu do bloku, w którym ją zadeklarowano.

Zmienne automatyczne, łącznie z parametrami funkcji, zasłaniają także zmienne zewnętrzne i funkcje o tych samych nazwach, np. po deklaracjach

```
int x;
int y;

f(double x)
{
    double y;
    ...
}
```

pojawienie się x wewnątrz funkcji f jest odwołaniem do parametru funkcji, który jest typu double; poza funkcją f takie odwołania dotyczą zmiennej zewnętrznej x o typie int. Ta sama zasada obowiązuje dla zmiennej y.

Jest to kwestia stylu, ale lepiej unikać nazw zmiennych zasłaniających nazwy występujące w otoczeniu; niebezpieczeństwo wprowadzenia bałaganu i wystąpienia błędu jest zbyt duże.

4.9 Inicjowanie

Dotychczas temat nadawania wartości początkowych był poruszany wielokrotnie, zawsze jednak jako drugorzędny w stosunku do innych zagadnień. Po omówieniu różnych klas pamięci możemy wreszcie sformułować kilka zasad inicjowania zmiennych.

Jeśli nie podano jawnie wartości początkowych, to zmienne zewnętrzne i statyczne zawsze będą inicjowane zerami, natomiast wartości początkowe zmiennych automatycznych i rejestrowych będą nieokreślone (tj. przypadkowe).

Zmienne jednowymiarowe można inicjować przy ich definicji, umieszczając po nazwie zmiennej znak równości i pewne wyrażenie:

```
int x = 1;
char squote = '\'';  /* apostrof */
long day = 1000L * 60L * 60L * 24L;  /* milisekundy/dzień */
```

Wartością początkową zmiennych zewnętrznych i statycznych musi być wyrażenie stałe; inicjowanie odbywa się tylko raz, ogólnie mówiąc – zanim pogram rozpocznie działanie. Zmienne automatyczne i rejestrowe są inicjowane przy każdym wejściu do funkcji lub bloku.

Wartością początkową zmiennej automatycznej i rejestrowej nie musi być stała: może nią być dowolne wyrażenie zawierające uprzednio zdefiniowane wartości a nawet wywołania funkcji. Na przykład, w programie wyszukiwania metodą bisekcji z p. 3.3 inicjowanie zmiennych można zapisać tak:

```
int binsearch(int x, int v[ ], int n)
{
    int low = 0;
    int high = n - 1;
    int mid;
    ...
}
```

zamiast

```
int low, high, mid;

low = 0;
high = n - 1;
```

A więc inicjowanie zmiennych jest w istocie skróconym zapisem instrukcji przypisania. Wybór jednego z tych sposobów jest kwestią stylu. Dotychczas na ogół stosowaliśmy jawne przypisania, gdyż wartości początkowe w deklaracjach są mniej widoczne i położone daleko od miejsca użycia zmiennych.

Tablicę można zainicjować umieszczając po jej deklaracji znak równości, a następnie – w nawiasach klamrowych – listę wartości początkowych rozdzielonych przecinkami. Na przykład, aby zainicjować tablicę days liczbami dni przypadającymi na każdy miesiąc, można napisać tak:

```
int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Jeżeli w deklaracji tablicy pominięto jej rozmiar, to kompilator obliczy jej długość na podstawie liczby podanych wartości początkowych, w tym przypadku 12.

Jeśli liczba wartości początkowych jest mniejsza od podanego rozmiaru tablicy, to brakujące elementy w zmiennych zewnętrznych, statycznych i automatycznych otrzymają wartość zero. Podanie zbyt wielu wartości początkowych jest błędem. Nie ma sposobu ani na sformułowanie powtórzeń wartości początkowej, ani na zainicjowanie środkowego elementu tablicy bez podania wszystkich wartości pośrednich.

Tablice znakowe mogą być inicjowane w szczególny sposób: zamiast nawiasów klamrowych i listy wartości z przecinkami można użyć stałej napisowej:

```
char pattern[ ] = "nie";
```

jest skróconym zapisem równoważnej deklaracji

```
char pattern[ ] = { 'n', 'i', 'e', '\0' };
```

W tym przypadku rozmiar tablicy wynosi 4 (trzy znaki wzorca plus znacznik końca tekstu '\0').

4.10 Rekurencja

Funkcje języka C mogą być wywoływane rekurencyjnie, tzn. funkcja może wywołać samą siebie zarówno bezpośrednio, jak i pośrednio. Rozważmy wypisywanie liczby w postaci ciągu znaków. Jak już wcześniej wspomnieliśmy, cyfry generuje się w od-

wrotnej kolejności: mniej znaczące cyfry są znane przed bardziej znaczącymi, a powinny być wypisane po tych ostatnich.

Istnieją dwa rozwiązania tego problemu. Jednym z nich jest zapamiętanie cyfr w tablicy w kolejności ich generowania, a następnie wypisanie ich w odwrotnym porządku. Tak właśnie zrobiliśmy w funkcji itoa z p. 3.6. Alternatywą jest rozwiązanie rekurencyjne, w którym funkcja printd najpierw wywołuje samą siebie dla uzyskania cyfr początkowych, a następnie wypisuje cyfrę końcową. Także ta wersja może działać niepoprawnie dla największej możliwej liczby ujemnej.

```
#include <stdio.h>

/* printd: wypisz n dziesiętnie */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10)
        printd(n / 10);
    putchar(n % 10 + '0');
}
```

Gdy funkcja wywołuje rekurencyjnie samą siebie, wówczas każde jej wznowienie otrzymuje nowy komplet wszystkich zmiennych automatycznych, niezależny od poprzedniego. Po wywołaniu printd(123) pierwsza funkcja printd otrzymuje więc argument n o wartości 123. Następnie przekazuje drugiej funkcji printd argument 12, która z kolei przekazuje trzeciej wartość 1. Funkcja printd z trzeciego poziomu wypisuje 1 i wraca na drugi poziom. Tutejsza printd wypisuje 2 i wraca na poziom pierwszy. Ta zaś printd wypisuje 3 i kończy działanie.

Innym dobrym przykładem rekurencji jest algorytm porządkowania „quicksort'' (szybkie sortowanie), który wymyślił C. A. Hoare w 1962 r. Dla danej tablicy wybiera się jeden element, a pozostałe rozdziela na dwa podzbiory – tych elementów, które są mniejsze od wybranego, i tych, które są od niego większe lub są mu równe. Następnie proces ten powtarza się rekurencyjnie dla każdego z podzbiorów. Gdy podzbiór ma mniej niż dwa elementy, nie potrzebuje już być porządkowany; osiągnięcie tego stanu kończy rekurencję.

Nasza wersja algorytmu szybkiego sortowania nie jest najszybszą z możliwych, ale jest jedną z najprostszych. W naszym rozwiązaniu elementem podziału każdej tablicy na dwie „podtablice'' jest element środkowy.

```
/* qsort: uporządkuj v[left]...v[right] rosnąco */
void qsort(int v[ ], int left, int right)
{
    int i, last;
    void swap(int v[ ], int i, int j);

    if (left >= right)   /* nic nie rób, jeśli tablica zawiera */
        return;          /* mniej niż dwa elementy */
    swap(v, left, (left + right)/2);   /* element podziału */
    last = left;                       /* przesuń do v[0] */
    for (i = left+1; i <= right; i++)  /* podział */
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last);       /* odtwórz element podziału */
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

Operację zamiany elementów miejscami napisaliśmy jako oddzielną funkcję, ponieważ trzy razy pojawia się w qsort.

```
/* swap: zamień miejscami v[i] i v[j] */
void swap(int v[ ], int i, int j)
{
    int temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

W bibliotece standardowej występuje taka wersja funkcji qsort, która potrafi sortować obiekty dowolnego typu.

Rekurencja nie musi przynosić oszczędności pamięci, ponieważ trzeba gdzieś przechowywać stos używanych wartości. Nie przyspiesza też działania funkcji. Postać rekurencyjna jest jednak bardziej zwarta i często łatwiejsza do napisania i zrozumienia, niż jej nierekurencyjny odpowiednik. Funkcje rekurencyjne nadają się zwłaszcza do obsługi rekurencyjnie zdefiniowanych struktur danych, np. drzew; w p. 6.5 zobaczymy dobry przykład takiego zastosowania.

Ćwiczenie 4.12. Zastosuj idee funkcji printd do napisania rekurencyjnej wersji funkcji itoa, tj. zamieniającej liczbę całkowitą na ciąg cyfr za pomocą wywołania rekurencyjnego.

Ćwiczenie 4.13. Napisz rekurencyjną wersję funkcji reverse(s), która odwraca kolejność znaków tekstu w s.

4.11 Preprocesor języka C

C pozwala na pewne rozszerzenia języka za pomocą preprocesora, który jest pojęciowo oddzielnym, pierwszym krokiem tłumaczenia programu. Z tych rozszerzeń najczęściej są stosowane dwa: polecenie #include wstawienia zawartości pewnego pliku podczas kompilacji oraz definicja #define, umożliwiająca zastępowanie pewnego zwrotu dowolnym ciągiem znaków. Do innych rozszerzeń tu opisanych należą kompilacja warunkowa i makra z argumentami.

4.11.1 Wstawianie plików

Wstawianie plików ułatwia, między innymi, posługiwanie się zestawami deklaracji i definicji #define. Każdy wiersz źródłowy programu o postaci

#include "*nazwa-pliku*"

lub

#include <*nazwa-pliku*>

jest zastępowany zawartością pliku o wskazanej nazwie. Jeśli *nazwa-pliku* jest ograniczona cudzysłowami, poszukiwanie pliku zaczyna się tam, gdzie znaleziono właściwy program źródłowy. Jeśli w tym miejscu go nie ma lub jeśli *nazwa-pliku* jest zawarta między znakami < i >, to tego pliku szuka się zgodnie z zasadami obowiązującymi w danej implementacji. Wstawiany plik może sam zawierać wiersze #include.

Często kilka wierszy o takiej postaci pojawia się na początku pliku źródłowego; ich zadaniem jest dołączenie wspólnych definicji #define i deklaracji extern lub wprowadzenie deklaracji prototypów funkcji bibliotecznych ze standardowych nagłówków jak <stdio.h>. (Ściśle mówiąc, nie muszą być one plikami; szczegóły dotyczące dostępu do nagłówków są zależne od implementacji.)

Stosowanie #include jest zalecanym sposobem sporządzania deklaracji dla dużego programu. Gwarantuje to, że wszystkie pliki źródłowe będą zaopatrzone w te same definicje i deklaracje zmiennych, co eliminuje szczególnie złośliwy rodzaj błędów.

Jeśli wstawiany plik zostanie zmieniony, to naturalnie wszystkie pliki korzystające z niego należy ponownie przetłumaczyć.

4.11.2 Makrorozwinięcie

Definicja o następującej postaci:

```
#define nazwa zastępujący-tekst
```

jest makrodefinicją najprostszego rodzaju: dalsze wystąpienia *nazwy* będą zastępowane przez ciąg znaków tworzący *zastępujący-tekst*. Nazwa w #define ma taką samą postać, jak nazwa zmiennej; zastępujący ją tekst jest dowolny. Zwykle tym tekstem jest reszta wiersza, ale długie definicje można kontynuować w następnych wierszach po umieszczeniu na końcu każdego przedłużanego wiersza znaku \. Zasięg nazwy wprowadzanej przez #define rozciąga się od miejsca definicji do końca tłumaczonego pliku źródłowego. Definicja może korzystać z poprzednich definicji. Makrorozwinięć dokonuje się jedynie dla całych jednostek leksykalnych (leksemów) i nie stosuje wewnątrz stałych napisowych ograniczonych cudzysłowami. Jeśli na przykład YES jest tak zdefiniowaną nazwą, to nie zostanie zastąpiona w instrukcji printf("YES") ani w zwrocie YESMAN.

W ten sposób można definiować dowolne nazwy i zastępować je dowolnym tekstem. Na przykład w wierszu

```
#define forever for (;;) /* pętla nieskończona */
```

definiuje się nowe słowo forever oznaczające pętlę nieskończoną.

Istnieje także możliwość definiowania makr z argumentami, czyli że zastępujący tekst może być różny dla różnych makrowywołań. Dla przykładu zdefiniujemy makro o nazwie max:

```
#define max(A, B)  ((A) > (B) ? (A) : (B))
```

Choć wygląda jak wywołanie funkcji, to jednak wywołanie max powoduje wstawienie rozwiniętego tekstu makra bezpośrednio do tekstu programu. W makrorozwinięciu każde wystąpienie parametru formalnego (tutaj A lub B) zostanie zastąpione przez aktualny argument. A zatem wiersz programu:

```
x = max(p+q, r+s);
```

zostanie zastąpiony następującym:

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Tak długo, jak argumenty są traktowane w sposób konsekwentny, makro może być stosowane dla dowolnych typów danych; nie ma potrzeby podawania różnych definicji max dla różnych typów, jak to jest z funkcjami.

Oglądając dokładnie rozwinięcie makra max, na pewno zauważysz kilka zasadzek. Wyrażenia są obliczane dwukrotnie; będzie źle, jeżeli powodują efekty uboczne, jak operatory zwiększania czy operacje wejścia lub wyjścia. Na przykład w wywołaniu

```
max(i++, j++) /* ŹLE */
```

dwukrotnie nastąpi zwiększenie większej wartości. Trochę uwagi należy także poświęcić odpowiedniemu stawianiu nawiasów, by zapewnić poprawną kolejność obliczeń; rozważ, co się stanie, gdy makro

```
#define square(x) x * x /* ŹLE */
```

(kwadrat liczby) zostanie wywołane w ten sposób: square(z+1).

Mimo wszystko makra są bardzo pożyteczne. Jeden z praktycznych przykładów pochodzi z nagłówka <stdio.h>, w którym getchar i putchar często są definiowane jako makra; tym sposobem unika się narzutów spowodowanych realizacją wywołania funkcji dla każdego przetwarzanego znaku. Także funkcje z nagłówka <ctype.h> są zwykle implementowane jako makra.

Definicję nazwy można skasować za pomocą polecenia #undef, zwykle po to, by zagwarantować, że rzeczywiście oznacza nazwę funkcji, a nie makro:

```
#undef getchar

int getchar(void) { ... }
```

Parametry formalne makra nie są zastępowane w stałych napisowych ograniczonych cudzysłowami. Jeśli jednak nazwę parametru w zastępującym tekście poprzedza znak #, to cała kombinacja (ten znak i nazwa parametru) zostanie rozwinięta w ciąg znaków ograniczony cudzysłowami, gdzie parametr będzie zastąpiony argumentem aktualnym. To zaś można połączyć ze sklejaniem napisów, aby utworzyć, powiedzmy, makro pożyteczne w fazie testowania programu:

```
#define dprint(expr) printf(#expr " = %g\n", expr)
```

To makro wywołane przykładowo tak:

```
dprint(x/y);
```

zostanie rozwinięte w wiersz

```
printf("x/y" " = %g\n", x/y);
```

w którym nastąpi sklejenie napisów, a więc w efekcie otrzymamy

```
printf("x/y = %g\n", x/y);
```

Występujące w aktualnym argumencie wszystkie znaki cudzysłowu " są zastępowane przez kombinację \", a wszystkie znaki \ przez kombinację \\, toteż wynik jest poprawną stałą napisową.

Operator preprocesora ## umożliwia sklejanie argumentów aktualnych podczas rozwijania makra. Jeśli w zastępującym tekście parametr sąsiaduje z operatorem ##, to ten parametr zastępuje się aktualnym argumentem, następnie usuwa operator ## wraz z otaczającymi go białymi znakami, a wynik przegląda ponownie. Oto przykład makra paste sklejającego swoje dwa argumenty:

```
#define paste(front, back) front ## back /*sklej początek z końcem */
```

Wywołanie paste(name, 1) utworzy słowo name1.

Zasady zagnieżdżania operatora ## są tajemnicze; więcej szczegółów na ten temat można znaleźć w dodatku A.

Ćwiczenie 4.14. Napisz definicję makra swap(t,x,y) zamieniającego miejscami wartości argumentów x i y o typach t. (Skorzystaj ze struktury blokowej.)

4.11.3 Kompilacja warunkowa

Procesem tłumaczenia można sterować za pomocą instrukcji warunkowych, które są wykonywane w fazie preprocesora. Tym sposobem fragmenty kodu włącza się do programu wybiórczo, w zależności od wartości warunków obliczanych podczas kompilacji.

W wierszu zaczynającym się od #if oblicza się wartość stałego wyrażenia całkowitego (nie może zawierać sizeof, rzutowania ani stałych wyliczeń). Jeśli wyrażenie ma wartość różną od zera, to kolejne wiersze – aż do jednego z wierszy zaczynających się od #endif, #elif lub #else – są włączane do programu. (Instrukcja preprocesora #elif jest podobna do else if.) Wyrażenie defined(*nazwa*) w instrukcji #if jest równe 1, jeśli *nazwa* została uprzednio zdefiniowana za pomocą #define; w przeciwnym przypadku ma wartość 0.

Na przykład, aby zapewnić, że zawartość pliku nagłówkowego hdr.h jest wstawiana do programu źródłowego tylko raz, zawartość tego pliku otacza się wyrażeniami warunkowymi preprocesora:

```
#if ! defined(HDR)
#define HDR

/* zawartość pliku hdr.h jest tutaj */

#endif
```

Pierwsze wstawienie pliku hdr.h definiuje nazwę HDR; przy każdej następnej próbie wstawienia tego pliku nazwa HDR jest już zdefiniowana, zawartość pliku jest więc omijana aż do wiersza zawierającego #endif. W podobny sposób można zapobiec wielokrotnemu wstawianiu różnych plików. Jeśli jest on stosowany konsekwentnie, to każdy nagłówek może sam sobie dołączać dowolne inne nagłówki, od których zależy jego zawartość; użytkownik tego nagłówka nie musi już się martwić o wzajemne zależności.

W następującym fragmencie programu sprawdza się wartość nazwy SYSTEM, by zadecydować, którą z wersji nagłówka należy wstawić do programu:

```
#if SYSTEM == SYSV
   #define HDR "sysv.h"
#elif SYSTEM == BSD
   #define HDR "bsd.h"
#elif SYSTEM == MSDOS
   #define HDR "msdos.h"
#else
   #define HDR "default.h"
#endif
#include HDR
```

Do sprawdzenia, czy nazwa została uprzednio zdefiniowana, wprowadzono dwie specjalne instrukcje #ifdef oraz #ifndef. Pierwszy z przykładów, korzystający z #if, można więc napisać tak:

```
#ifndef HDR
#define HDR

/* zawartość pliku hdr.h jest tutaj */

#endif
```

5 WSKAŹNIKI I TABLICE

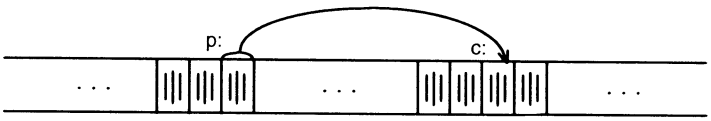
Wskaźnik jest zmienną, która zawiera adres innej zmiennej. W języku C chętnie korzysta się ze wskaźników częściowo dlatego, że czasami jest to jedyny sposób przedstawienia algorytmu obliczenia, częściowo zaś dlatego, że ich użycie zwykle prowadzi do bardziej zwartego i efektywnego kodu niż otrzymywany innymi sposobami. Wskaźniki i tablice są blisko spokrewnione; w tym rozdziale badamy to pokrewieństwo i pokazujemy, jak można z niego korzystać.

Wskaźniki wraz z instrukcją goto „cudownie'' nadają się do tworzenia zupełnie niezrozumiałych programów. Dzieje się tak bez wątpienia wówczas, gdy wskaźniki są stosowane nieostrożnie. Łatwo jest bowiem utworzyć wskaźnik, który wskazuje nie wiadomo na co. Jednak przy przestrzeganiu odpowiedniej dyscypliny dzięki wskaźnikom można osiągnąć znaczną przejrzystość i prostotę algorytmów. Właśnie ten aspekt spróbujemy zilustrować.

Główna zmiana w ANSI C polega na jawnym określeniu reguł, według których można manipulować wskaźnikami, a więc w efekcie jest uprawomocnieniem tego, co dobrzy programiści od dawna stosują, a czego dobre kompilatory od dawna przestrzegają. Dodatkowo zamiast typu char * (wskaźnik do typu char) wprowadzono typ void * (wskaźnik do typu void) jako jedyny poprawny typ dla wskaźnika ogólnego.

5.1 Wskaźniki i adresy

Rozpoczniemy od pokazania prościutkiego obrazka, na którym widać, jak jest zorganizowana pamięć. W typowej maszynie jest to tablica kolejno numerowanych lub adresowanych komórek pamięci; można nimi manipulować indywidualnie lub posługiwać się całymi grupami sąsiednich komórek. Ogólny stan rzeczy jest taki, że dowolny bajt można traktować jako obiekt typu char, parę jednobajtowych komórek – jako obiekt całkowity typu short, a cztery przylegające do siebie bajty – jako obiekt całkowity typu long. Wskaźnik jest grupą komórek (często dwie lub cztery), które mogą pomieścić adres. A więc jeśli c jest obiektem typu char, a p jest wskaźnikiem, który wskazuje na c, to taką sytuację możemy zilustrować następująco:



Jednoargumentowy operator & podaje adres obiektu, zatem instrukcja

```
p = &c;
```

przypisuje zmiennej p adres zmiennej c; teraz zmienna p „wskazuje na" zmienną c. Operator adresu & może być stosowany tylko do obiektów zajmujących pamięć: zmiennych i elementów tablic. Nie można go stosować do wyrażeń, stałych i zmiennych register.

Jednoargumentowy operator * oznacza *adresowanie pośrednie* lub *odwołanie pośrednie*; zastosowany do wskaźnika daje zawartość obiektu wskazywanego przez ten wskaźnik. Przypuśćmy, że x i y są obiektami całkowitymi, a ip jest wskaźnikiem do obiektu typu int. Następujący sztuczny ciąg instrukcji pokazuje, jak zadeklarować wskaźnik oraz jak stosować operatory & i *:

```
int x = 1, y = 2, z[10];
int *ip;        /* ip jest wskaźnikiem do obiektów typu int */

ip = &x;        /* teraz ip wskazuje na x */
y = *ip;        /* y ma teraz wartość 1 */
*ip = 0;        /* x ma teraz wartość 0 */
ip = &z[0];     /* teraz ip wskazuje na element z[0] */
```

Deklaracje obiektów x, y i z znamy już od dawna. Deklaracja wskaźnika ip:

```
int *ip;
```

z założenia jest mnemotechniczna; mówi ona, że wyrażenie *ip ma wartość typu int. Składnia deklaracji tej zmiennej naśladuje składnię wyrażenia, w którym zmienna może wystąpić. Takie rozumowanie stosuje się również do deklaracji funkcji. Na przykład deklaracja

```
double *dp, atof(char *);
```

mówi, że w wyrażeniu obie konstrukcje *dp i atof(s) mają wartości typu double oraz że argument funkcji atof jest wskaźnikiem do typu char.

Zapamiętaj także płynący stąd wniosek: od wskaźnika wymaga się wskazywania obiektów określonego rodzaju. (Z jednym wyjątkiem – wskaźnik do void, czyli „wskaźnik do niczego", może przechowywać wskaźnik do obiektów dowolnego typu, ale nie można go stosować do adresowania pośredniego. Wrócimy do tego w p. 5.11.)

Jeżeli wskaźnik ip wskazuje na zmienną całkowitą x, to *ip może wystąpić wszędzie tam, gdzie może wystąpić x, a więc na przykład

```
*ip = *ip + 10;
```

zwiększa obiekt wskazywany przez ip o 10.

Jednoargumentowe operatory & i * wiążą silniej niż operatory arytmetyczne, zatem w przypisaniu

```
y = *ip + 1
```

najpierw bierze się wartość obiektu, na który wskazuje ip, dodaje do niej 1 i wynik przypisuje zmiennej y. Natomiast zwiększenie tego obiektu, na który wskazuje ip, można zapisać tak:

```
*ip += 1
```

lub tak:

```
++*ip
```

lub jeszcze inaczej:

```
(*ip)++
```

W tym ostatnim przypadku nawiasy są niezbędne: bez nich efektem obliczenia byłoby zwiększenie wskaźnika ip, a nie obiektu wskazywanego przez ip, gdyż operacje określone przez jednoargumentowe operatory * i ++ są wykonywane od prawej strony do lewej.

Na koniec, wskaźniki są zwykłymi zmiennymi, mogą więc być używane bez adresowania pośredniego. Dla przykładu, jeśli iq jest innym wskaźnikiem do obiektów typu int, to

```
iq = ip
```

kopiuje zawartość ip do iq sprawiając, że iq będzie wskazywać na to samo, na co wskazuje ip.

5.2 Wskaźniki i argumenty funkcji

W języku C argumenty są przekazywane przez wartość. Nie ma więc bezpośredniego sposobu na to, aby wywołana funkcja mogła zmienić wartość zmiennej należącej

do funkcji wywołującej. Na przykład podprogram sortujący mógłby zamieniać miejscami dwa nieuporządkowane elementy za pomocą funkcji swap. Nie wystarczy jednak napisać

```
swap(a, b);
```

jeśli funkcja swap jest zdefiniowana następująco:

```
void swap(int x, int y) /*ŹLE */
{
   int temp;

   temp = x;
   x = y;
   y = temp;
}
```

Na skutek przekazywania argumentów przez wartość funkcja ta nie ma dostępu do argumentów a i b należących do wywołującego ją podprogramu. Powyższa funkcja swap jedynie zamienia miejscami wartości *kopii* argumentów a i b.

Jedyny sposób na osiągnięcie zamierzonego celu polega na przekazaniu przez program wywołujący *wskaźników* do obiektów, których wartości należy zamienić miejscami:

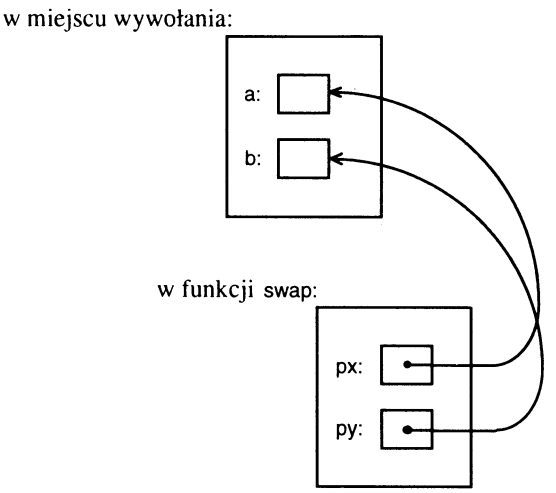
```
swap(&a, &b);
```

Operator & daje adres zmiennej, zatem &a jest wskaźnikiem do a. Deklarując w samej funkcji swap parametry jako wskaźniki, otrzymujemy – za ich pomocą – pośredni dostęp do rzeczywistych argumentów operatora adresu.

```
void swap(int *px, int *py) /* zamień miejscami *px i *py */
{
   int temp;

   temp = *px;
   *px = *py;
   *py = temp;
}
```

Obrazowo wygląda to tak:



Argumenty wskaźnikowe pozwalają funkcji mieć dostęp do (a więc i zmieniać wartość) obiektów należących do funkcji wywołującej. Jako przykład rozważmy funkcję getint, która wczytując nie sformatowane dane wejściowe, przekształca strumień znaków na wartości całkowite (jedną na każde wywołanie). Funkcja ta musi przekazać znalezioną wartość oraz sygnał końca pliku, gdy nie ma już więcej znaków z wejścia. Obie wartości muszą być przesłane dwiema oddzielnymi ścieżkami, gdyż niezależnie od tego, jaką wartość ma EOF, może ona być również wartością wczytanej liczby.

Jedno z rozwiązań polega na tym, że funkcja getint zwraca sygnał o napotkaniu końca pliku jako swoją wartość funkcyjną, podczas gdy znalezioną liczbę całkowitą przekazuje do funkcji wywołującej za pomocą argumentu wskaźnikowego. Taki sam schemat jest podstawą działania standardowej funkcji scanf; zajrzyj do p. 7.4.

W następującej pętli do wypełnienia tablicy wartościami całkowitymi użyto funkcji getint:

```
int n, array[SIZE], getint(int *);

for (n = 0; n < SIZE && getint(&array[n]) != EOF; n++)
    ;
```

W każdym kroku pętli wywołanie funkcji getint umieszcza w array[n] wartość kolejnej wejściowej liczby; zwiększa się też licznik n. Zauważ, że przekazanie do funkcji getint adresu elementu array[n] jest konieczne, inaczej bowiem nie ma sposobu na to, by mogła ona obliczoną wartość przekazać do miejsca wywołania.

Nasza wersja funkcji getint zwraca: EOF – gdy napotka koniec pliku, zero – jeśli kolejne znaki z wejścia nie tworzą liczby, oraz wartość dodatnią – jeśli na wejściu podano poprawną liczbę*.

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getint: wczytaj następną liczbę całkowitą */
int getint(int *pn)
{
    int c, sign;

    while (isspace(c = getch())) /* pomiń białe znaki */
        ;
    if (! isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetch(c); /* to nie jest liczba */
        return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getch();
    for (*pn = 0; isdigit(c); c = getch())
        *pn = 10 * *pn + (c - '0');
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return c;
}
```

W całej funkcji getint konstrukcja *pn jest używana jak zwykła zmienna całkowita. Ponadto skorzystaliśmy z funkcji getch i ungetch (opisanych w p. 4.3), aby dodatkowy znak, który trzeba było przeczytać, mógł być z powrotem odesłany na wejście.

* Przy takiej definicji funkcji getint warunek w powyższej pętli trzeba uzupełnić sprawdzeniem, czy funkcja zwraca zero. Przy podanym warunku funkcja getint zatnie się na pierwszym znaku nie należącym do liczby (zawsze sygnalizując zero), mimo że pętla będzie się kręcić dalej. – *Przyp. tłum.*

Ćwiczenie 5.1. Tak jak została napisana, funkcja getint traktuje znaki + i –, po których nie następuje cyfra, jako poprawne reprezentacje zera. Zmień tę funkcję tak, by oddawała je z powrotem na wejście.

Ćwiczenie 5.2. Napisz funkcję getfloat – zmiennopozycyjny odpowiednik funkcji getint. Jakiego typu wartość powinna zwracać getfloat jako swoją wartość funkcyjną?

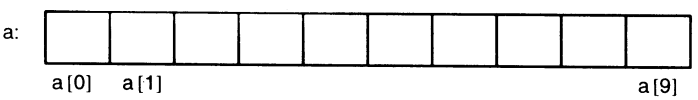
5.3 Wskaźniki i tablice

W języku C występuje ścisła zależność między wskaźnikami i tablicami, ścisła do tego stopnia, że wskaźniki i tablice powinny być rozpatrywane jednocześnie. Każdą operację, którą można przeprowadzić za pomocą indeksowania tablicy, można również wykonać za pomocą wskaźników. Wersja wskaźnikowa będzie na ogół szybsza, ale – zwłaszcza dla początkujących – trochę trudniejsza do zrozumienia.

Deklaracja

```
int a[10];
```

definiuje tablicę a o rozmiarze 10, a więc blok dziesięciu kolejnych obiektów nazwanych a[0], a[1], ..., a[9].



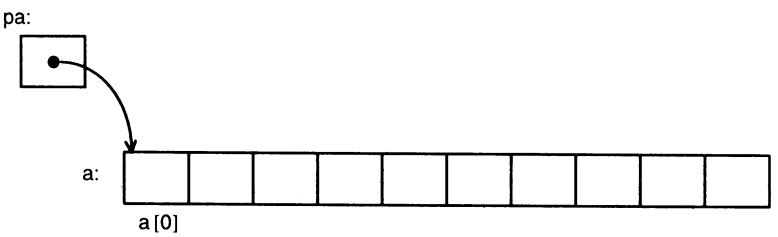
Zapis a[i] oznacza i-ty element tablicy a. Niech pa będzie wskaźnikiem do obiektów całkowitych, zadeklarowanym jako

```
int *pa;
```

wówczas przypisanie

```
pa = &a[0];
```

ustawi pa tak, aby wskazywał na zerowy element tablicy a; wskaźnik pa zawiera więc adres elementu a[0].



Teraz przypisanie

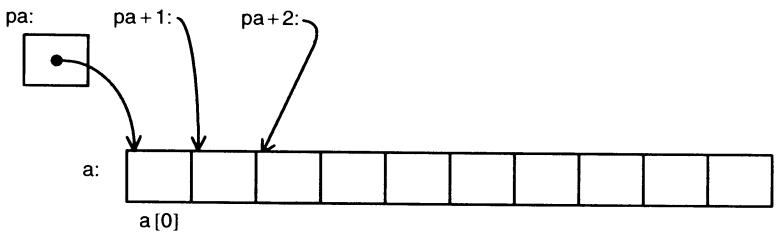
```
x = *pa;
```

skopiuje zawartość a[0] do x.

Jeśli pa wskazuje na pewien element tablicy, to – z definicji – pa+1 wskazuje na element następny, pa+i odnosi się do i-tego elementu po pa, a pa–i do i-tego elementu przed pa. Jeżeli więc pa wskazuje na a[0], to

```
*(pa+1)
```

odnosi się do zawartości a[1]; pa+i jest adresem a[i], a *(pa+i) jest zawartością a[i].



Te spostrzeżenia są prawdziwe niezależnie od typu lub rozmiaru elementów tablicy a. Znaczenie operacji „dodawanie 1 do wskaźnika", a szerzej – całej arytmetyki na wskaźnikach, jest oparte na tym, że pa+1 wskazuje następny obiekt, a pa+i odnosi się do obiektu oddalonego od pa o i takich obiektów.

Ścisła odpowiedniość między indeksowaniem i arytmetyką na wskaźnikach jest oczywista. Wartością zmiennej lub wyrażenia typu tablica jest z definicji adres zerowego elementu tej tablicy. Zatem po przypisaniu

```
pa = &a[0];
```

pa oraz a mają identyczne wartości. Ponieważ nazwa tablicy reprezentuje położenie jej elementu początkowego, przypisanie pa=&a[0] można także napisać w postaci

```
pa = a;
```

Większą niespodzianką, przynajmniej na pierwszy rzut oka, będzie to, że odwołanie do a[i] można zapisać jako *(a+i). W języku C wyrażenie a[i] przy obliczaniu jest przekształcane bezpośrednio na *(a+i) – obie te formy są równoważne. Po zastosowaniu operatora adresu & do obu form widać, że &a[i] oraz a+i także są identyczne: a+i jest adresem elementu oddalonego o i elementów od a. A z drugiej strony, jeśli pa jest wskaźnikiem, to w wyrażeniach może wystąpić z indeksem: pa[i] jest równoważne z *(pa+i). Podsumowując, wyrażenie w postaci „tablica i indeks" jest równoważne z wyrażeniem w postaci „wskaźnik i przesunięcie" (ang. *offset*).

Między nazwą tablicy a wskaźnikiem jest jednak istotna różnica, o której należy pamiętać. Wskaźnik jest zmienną, więc operacje pa=a czy pa++ są dozwolone. Natomiast nazwa tablicy nie jest zmienną, zatem konstrukcje w rodzaju a=pa i a++ są niedozwolone.

Gdy argumentem funkcji jest nazwa tablicy, wówczas funkcja otrzymuje położenie początkowego elementu tej tablicy. Wewnątrz wywołanej funkcji temu argumentowi odpowiada zwykła zmienna lokalna. Zatem parametr będący nazwą tablicy jest w istocie wskaźnikiem, czyli zmienną zawierającą adres. Korzystając z tego, możemy napisać inną wersję funkcji strlen obliczającej długość tekstu.

```
/* strlen: podaj długość tekstu s */
int strlen(char *s)
{
    int n;

    for (n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

Zwiększanie s jest całkowicie poprawne, s jest bowiem zmienną wskaźnikową. Operacja s++ nie ma żadnego wpływu na ciąg znaków w miejscu wywołania funkcji strlen, zwiększa jedynie jej prywatną kopię wskaźnika. Z tego wynika, że wywołania mające postać

```
strlen("ahoj, przygodo"); /* stała napisowa */
strlen(array);            /* char array[100]; */
strlen(ptr);              /* char *ptr; */
```

działają poprawnie.

Następujące definicje parametrów formalnych funkcji:

```
char s[ ];
```

i

```
char *s;
```

są równoważne; my dajemy pierwszeństwo tej ostatniej, gdyż wyraźniej mówi o tym, że parametr jest wskaźnikiem. Jeśli do funkcji jest przekazywana nazwa tablicy, to – w zależności od tego, co jest wygodniejsze – w funkcji tej można przyjąć, że jako argument otrzymuje ona albo nazwę tablicy, albo wskaźnik i odpowiednio tym argu-

mentem manipulować. Można nawet jednocześnie stosować oba rodzaje odwołań, o ile wydaje się to stosowne i przejrzyste.

Funkcji wolno przekazać część tablicy, podając wskaźnik do początku podtablicy. Na przykład, jeżeli a jest tablicą, to w obu przypadkach:

```
f(&a[2])
```

i

```
f(a+2)
```

funkcji f zostanie przekazany adres podtablicy, która zaczyna się od elementu a[2]. Wewnątrz funkcji f deklaracja parametru formalnego może mieć postać

```
f(int arr[ ]) {...}
```

lub

```
f(int *arr) {...}
```

Z punktu widzenia funkcji f to, że jej argument wskazuje w rzeczywistości na fragment większej tablicy, nie ma żadnego znaczenia.

Można także odwoływać się do elementów tablicy wstecz, tj. p[–1], p[–2] itd., jeśli jest rzeczą pewną, że te elementy istnieją. Taki zapis jest składniowo poprawny i odnosi się do obiektów, które bezpośrednio poprzedzają obiekt p[0]. Oczywiście odwoływanie się do obiektów, które leżą poza granicami tablicy, jest nielegalne.

5.4 Arytmetyka na adresach

Jeżeli p jest wskaźnikiem do pewnego elementu tablicy, to po zwiększeniu p++ wskazuje on na następny element, a po zwiększeniu p+=i – na element oddalony o i pozycji od aktualnie wskazywanego. Te i podobne konstrukcje są najprostszymi oraz najczęściej stosowanymi formami arytmetyki na wskaźnikach lub adresach.

Stosowana w języku C arytmetyka na adresach jest spójna i regularna. Związek między wskaźnikami, tablicami i arytmetyką na adresach stanowi jedną z najważniejszych cech świadczących o mocy języka. Zilustrujemy to na przykładzie prymitywnego programu dystrybutora pamięci. Tworzą go dwa podprogramy. Pierwszy, o nazwie alloc(n), udostępnia wskaźnik p do obszaru pamięci zajmującego n kolejnych pozycji znakowych; obszar ten można w funkcji wywołującej alloc wykorzystać do przechowywania znaków. Drugi podprogram, o nazwie afree(p), zwalnia uprzednio przydzieloną pamięć tak, aby mogła być ponownie wykorzystana. Ten mechanizm przydziału pamięci jest „prymitywny", ponieważ wywołania afree muszą następować w odwrotnej kolejności do wywołań alloc. Oznacza to, że pamięć obsługiwana przez

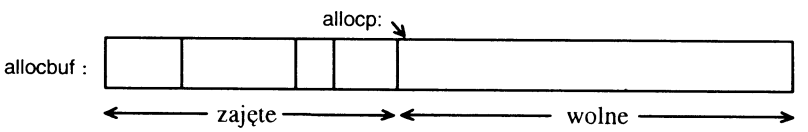
alloc i afree jest stosem lub listą LIFO (ang. *last-in, first-out*: ostatni przychodzi, pierwszy wychodzi). Biblioteka standardowa oferuje analogiczne funkcje o nazwach malloc i free, które nie wprowadzają takich ograniczeń; w p. 8.7 pokażemy, jak można je zrealizować.

W najprostszej wersji funkcja alloc przydziela części dużej tablicy znakowej, którą nazwiemy allocbuf. Jest ona prywatną własnością funkcji alloc i afree. Funkcje te będą używać wskaźników, a nie indeksów, nazwa tablicy nie musi więc być znana innym podprogramom. Tablicę możemy zatem zadeklarować jako static w tym pliku źródłowym, w którym mieszczą się obie funkcje – w ten sposób stanie się niedostępna dla otoczenia. W praktyce tablica ta nawet nie musi w ogóle mieć nazwy: możemy ją otrzymać wywołując funkcję malloc lub żądając od systemu operacyjnego wskaźnika do pewnego nie nazwanego bloku pamięci.

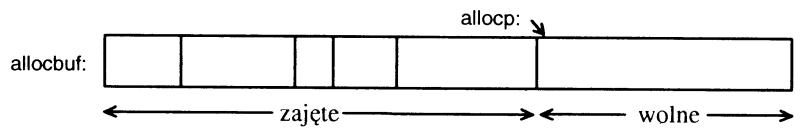
Potrzebna jest również informacja o tym, jak duża część tablicy allocbuf jest już zajęta. W tym celu użyjemy wskaźnika allocp, który wskazuje na następny wolny element. Funkcja alloc przed przydzieleniem n znaków sprawdza, czy w tablicy allocbuf jest wystarczająco dużo wolnego miejsca. Jeżeli tak, to funkcja wraca z bieżącą wartością wskaźnika allocp (to znaczy adresem początku wolnego obszaru), jednocześnie zwiększając ten wskaźnik o n, aby wskazywał na kolejny wolny obszar. Jeśli w tablicy nie ma wolnego miejsca, to funkcja alloc zwraca zero. Funkcja afree(p) po prostu wstawia wartość wskaźnika p do allocp, jeżeli p wskazuje na wnętrze tablicy allocbuf.

Oto rysunek, na którym przedstawiono działanie dystrybutora pamięci:

przed wywołaniem funkcji alloc:



po wywołaniu funkcji alloc:




```
#define ALLOCSIZE 10000 /* rozmiar dostępnej pamięci */

static char allocbuf[ALLOCSIZE]; /* pamięć dla alloc */
static char *allocp = allocbuf;  /* następna wolna pozycja */
```

```
char *alloc(int n) /* podaj wskaźnik do n znaków */
{
    if (allocbuf + ALLOCSIZE - allocp >= n) { /* wystarczy miejsca */
        allocp += n;
        return allocp - n; /* zwróć starą wartość allocp */
    } else      /* za mało miejsca */
        return 0;
}

void afree(char *p) /* zwolnij pamięć wskazaną przez p */
{
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
}
```

Ogólnie wskaźnikom można nadawać wartości początkowe tak samo, jak innym zmiennym. Jednak zwykle jedynymi znaczącymi wartościami są zero oraz wyrażenia zawierające adresy uprzednio zdefiniowanych danych właściwego typu. Deklaracja

```
static char *allocp = allocbuf;
```

definiuje allocp jako wskaźnik do znaków i inicjuje go tak, aby wskazywał na początek tablicy allocbuf, czyli na pierwszą wolną pozycję przy starcie programu. Można to również napisać inaczej:

```
static char *allocp = &allocbuf[0];
```

ponieważ nazwa tablicy *jest* adresem jej zerowego elementu.

Test

```
if (allocbuf + ALLOCSIZE - allocp >= n) { /* wystarczy miejsca */
```

sprawdza, czy w tablicy jest dostatecznie dużo wolnego miejsca, aby spełnić żądanie przydzielenia n znaków. Jeżeli tak, to nowa wartość allocp może wskazywać na miejsce przekraczające koniec obszaru allocbuf co najwyżej o jeden znak. Funkcja alloc zwraca wówczas wskaźnik do początku bloku znaków (zwróć uwagę na deklarację samej funkcji). Jeśli nie, to alloc musi zwrócić jakiś sygnał o braku pamięci. Język C gwarantuje, że zero nigdy nie jest poprawnym adresem danych, a więc może być sygnałem zajścia nienormalnego zdarzenia – w tym przypadku braku miejsca w pamięci.

Wskaźniki i liczby całkowite nie są wymienne. Zero jest jedynym wyjątkiem: stałą zero można przypisać wskaźnikowi, można też porównać wskaźnik ze stałą zero. Często zamiast zera używa się stałej symbolicznej NULL, by podkreślić, że chodzi

o specjalną wartość wskaźnikową. Stała NULL jest zdefiniowana w nagłówku <stdio.h>. Odtąd będziemy się nią posługiwać.

Testy w rodzaju:

```
if (allocbuf + ALLOCSIZE - allocp >= n) { /* wystarczy miejsca */
```

lub

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

ilustrują kilka ważnych cech arytmetyki na wskaźnikach. Po pierwsze, przy pewnych ograniczeniach wskaźniki można ze sobą porównywać. Jeżeli p i q wskazują na elementy tej samej tablicy, to relacje ==, !=, <, >= itp. działają poprawnie. Na przykład

```
p < q
```

jest prawdą, jeśli p wskazuje na wcześniejszy element tablicy niż q. Dowolny wskaźnik zawsze można przyrównać do zera. Natomiast skutek operacji arytmetycznych lub porównań dla wskaźników odwołujących się do elementów z różnych tablic jest nieokreślony. (Z jednym wyjątkiem: w arytmetyce na wskaźnikach można posłużyć się adresem pierwszego miejsca po ostatnim elemencie tablicy.)

Po drugie, obserwowaliśmy już, że wskaźnik i wartość całkowita mogą być dodawane i odejmowane. Konstrukcja

```
p + n
```

oznacza adres n-tego elementu od miejsca, na które wskazuje p. Jest to prawdziwe niezależnie od tego, jakiego rodzaju obiekty wskazuje p: wartość n jest skalowana odpowiednio do rozmiaru obiektów wskazywanych przez p, wynikającego z deklaracji p. Jeśli typ int zajmuje np. cztery bajty, to dla wskaźnika do obiektów typu int czynnikiem skalującym wartość n jest cztery.

Odejmowanie wskaźników jest także poprawne: jeżeli p i q wskazują na elementy tej samej talicy i p<q, to q–p+1 jest liczbą elementów między p i q (razem z q). Z faktu tego można skorzystać w jeszcze jednej wersji funkcji strlen:

```
/* strlen: podaj długość tekstu s */
int strlen(char *s)
{
    char *p = s;

    while (*p != '\0')
        p++;
    return p - s;
}
```

Wskaźnik p jest w deklaracji zainicjowany wartością s po to, by wskazywał na pierwszy znak tekstu. W pętli while bada się po kolei każdy znak do napotkania znaku '\0' kończącego tekst. Wskaźnik p odnosi się do znaków, zatem operacja p++ za każdym razem przesuwa p do następnego znaku, a p–s podaje liczbę znaków, o jaką go przesunięto, czyli długość tekstu. (Liczba znaków w tekście może być zbyt duża, aby mogła być przechowywana w zmiennej typu int. W nagłówku <stddef.h> zdefiniowano typ ptrdiff_t, który jest wystarczająco obszerny, aby pomieścić wartość (ze znakiem) różnicy dwóch wskaźników. Gdybyśmy jednak byli bardzo ostrożni, wówczas dla zwracanej przez strlen wartości powinniśmy użyć typu size_t, aby dopasować ją do wersji z biblioteki standardowej: size_t jest typem wartości całkowitej bez znaku, otrzymywanej za pomocą operatora sizeof.)

Zasady arytmetyki na wskaźnikach są konsekwentne: gdybyśmy mieli do czynienia z obiektami typu float, które zajmują więcej pamięci niż znaki, i gdyby p był wskaźnikem do obiektów typu float, wówczas operacja p++ spowodowałaby przesunięcie p do następnego obiektu typu float. Zatem moglibyśmy napisać inną wersję dystrybutora pamięci przydzielającego pamięć dla obiektów zmiennopozycyjnych (zamiast znaków), zamieniając jedynie słowa char na float w funkcjach alloc i afree. Wszystkie operacje na wskaźnikach są automatycznie dostosowywane do rozmiaru wskazywanych obiektów.

Do poprawnych operacji wskaźnikowych należą: przypisanie wskaźników do obiektów tego samego typu, dodawanie lub odejmowanie wskaźnika i liczby całkowitej, odejmowanie bądź porównywanie dwóch wskaźników do elementów tej samej tablicy oraz przypisanie wskaźnikowi wartości zero lub przyrównanie wskaźnika do zera. Wszystkie inne operacje na wskaźnikach są nielegalne. Nie wolno dodawać do siebie dwóch wskaźników ani ich mnożyć, dzielić, przesuwać albo składać z maskami, ani też dodawać do nich liczb typu float lub double. Nie wolno nawet – z wyjątkiem typu void * – wskaźnikowi do obiektów jednego typu przypisać bez rzutowania wskaźnika do obiektów innego typu.

5.5 Wskaźniki znakowe i funkcje

Stała napisowa, na przykład taka:

```
"Jestem napisem"
```

jest tablicą znaków. W wewnętrznej reprezentacji taka tablica jest zakończona znakiem '\0', więc programy mogą znaleźć jej koniec. Długość tekstu w pamięci jest zatem o jeden większa od liczby znaków zawartych między dwoma znakami cudzysłowu.

Stałe napisowe chyba najczęściej występują jako argumenty funkcji, jak w wywołaniu

```
printf("ahoj, przygodo\n");
```

Gdy taki ciąg znaków pojawia się w programie, wówczas dostęp do niego jest realizowany za pomocą wskaźnika znakowego; funkcja printf otrzymuje wskaźnik do początku tablicy znaków. Oznacza to, że stała napisowa jest dostępna poprzez wskaźnik do swojego pierwszego znaku.

Stałe napisowe nie muszą być argumentami funkcji. Jeśli pmessage zadeklarowano tak:

```
char *pmessage; /* wskaźnik do komunikatu */
```

to w instrukcji

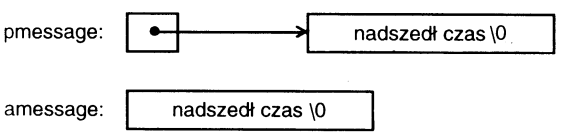
```
pmessage = "nadszedł czas";
```

zmiennej pmessage przypisuje się wskaźnik do określonej tablicy znaków. To *nie* jest kopia napisu; tu są zaangażowane wyłącznie wskaźniki. W języku C nie ma operacji do obsługi ciągów znaków traktowanych jako całość.

Zachodzi istotna różnica między takimi definicjami:

```
char amessage[ ] = "nadszedł czas";   /* tablica */
char *pmessage = "nadszedł czas";     /* wskaźnik */
```

amessage jest tablicą wystarczająco dużą, aby pomieścić swój inicjator, czyli ciąg znaków zakończony znakiem '\0'. Poszczególne znaki w tablicy można zmieniać, ale nazwa amessage zawsze będzie odwołaniem do tego samego miejsca w pamięci. Z drugiej strony, pmessage jest wskaźnikiem zainicjowanym tak, aby wskazywał stałą napisową; wskaźnik można później zmieniać, by wskazywał na cokolwiek, ale próba zmiany w stałej napisowej ma skutek nieokreślony.



Więcej właściwości wskaźników i tablic omówimy przy prezentacji dwóch pożytecznych funkcji pochodzących z biblioteki standardowej. Pierwsza z funkcji – strcpy(s,t) – kopiuje tekst z t do s. Byłoby bardzo przyjemnie, gdyby można było napisać s = t, ale niestety taki zapis zamiast znaków kopiuje jedynie wskaźniki.

Do kopiowania znaków jest niezbędna pętla. Jako pierwszą przedstawiamy tablicową wersję tej funkcji:

```
/* strcpy: skopiuj t do s; wersja z indeksowaniem tablic */
void strcpy(char *s, char *t)
{
    int i;

    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

I dla porównania ta sama funkcja w wersji wskaźnikowej:

```
/* strcpy: skopiuj t do s; wersja wskaźnikowa 1 */
void strcpy(char *s, char *t)
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

Argumenty są przekazywane funkcjom ,,przez wartość", a więc z parametrów s i t funkcja strcpy może korzystać, jak jej się podoba. Tutaj parametrami są dogodnie inicjowane wskaźniki, które jednocześnie maszerują wzdłuż tablic aż do napotkania znaku '\0' kończącego tekst w t i skopiowania go do s.

W praktyce funkcję strcpy pisze się jeszcze inaczej, niż pokazaliśmy. Programiści z doświadczeniem w języku C napisaliby ją raczej tak:

```
/* strcpy: skopiuj t do s; wersja wskaźnikowa 2 */
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != '\0')
        ;
}
```

Zwiększenie wskaźników s i t zostało przeniesione do części warunkowej pętli. Wartością operacji *t++ jest znak, na który wskazywał wskaźnik t przed zwiększeniem;

operator przyrostkowy ++ nie zmieni t, dopóki nie zostanie pobrany wskazywany znak. Na tej samej zasadzie znak jest wstawiany w miejsce wskazywane przez s przed zwiększeniem wskaźnika s. Ten sam znak jest także wartością porównywaną ze znakiem '\0' przy sterowaniu pętlą. Miłym zakończeniem całej operacji jest to, że znaki tekstu z t przepisano do s wraz z kończącym znakiem '\0'.

Przed ostatecznym skróceniem funkcji zauważ, że porównanie ze znakiem '\0' jest zbyteczne, ponieważ pytanie dla warunku pętli zawsze po prostu brzmi: czy wartością wyrażenia jest zero? Funkcję strcpy można zatem napisać w ten sposób:

```
/* strcpy: skopiuj t do s; wersja wskaźnikowa 3 */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

Na pierwszy rzut oka może to wyglądać tajemniczo, wygoda w zapisie jest jednak ogromna. Takie skróty warto opanować choćby tylko dlatego, że często możesz je spotkać w programach w C.

Funkcja strcpy z biblioteki standardowej (<string.h>) zwraca wynikowy tekst jako swoją wartość funkcyjną.

Drugą funkcją, którą chcemy tu omówić, jest strcmp(s,t). Funkcja ta porównuje dwa teksty zawarte w s oraz t i zwraca odpowiednio wartość ujemną, równą zero lub dodatnią w zależności od tego, czy tekst w s jest leksykograficznie mniejszy, równy lub większy niż tekst w t. Zwracaną wartość otrzymuje się odejmując od siebie znaki na pierwszej pozycji, na której teksty się różnią.

```
/* strcmp: zwróć: <0 jeśli s < t, 0 jeśli s == t, >0 jeśli s > t */
int strcmp(char *s, char *t)
{
    int i;

    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}
```

A oto wskaźnikowa wersja funkcji strcmp:

```
/* strcmp: zwróć: <0 jeśli s < t, 0 jeśli s == t, >0 jeśli s > t */
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}
```

Ponieważ ++ i −− są operatorami zarówno przedrostkowymi, jak i przyrostkowymi, mogą wystąpić (choć rzadziej) również inne kombinacje operatorów *, ++ i −−. Na przykład

```
*--p
```

zmniejsza p przed pobraniem znaku wskazywanego przez p. W rzeczywistości parę wyrażeń:

```
*p++ = val;   /* wstaw val na szczyt stosu */
val = *--p;   /* zdejmij wartość ze szczytu stosu do val */
```

można nazwać standardowymi idiomami języka C służącymi do obsługi stosu; patrz omówienie w p. 4.3.

Nagłówek <string.h> zawiera deklaracje omówionych tu funkcji oraz duży wybór innych funkcji manipulujących tekstami, pochodzących z biblioteki standardowej.

Ćwiczenie 5.3. Napisz wskaźnikową wersję funkcji strcat opisanej w rozdz. 2: funkcja strcat(s,t) dopisuje tekst z t na koniec tekstu w s.

Ćwiczenie 5.4. Napisz funkcję strend(s,t), która zwraca 1, jeśli tekst z t występuje na końcu tekstu w s; w przeciwnym przypadku zwraca zero.

Ćwiczenie 5.5. Napisz własne wersje funkcji bibliotecznych strncpy, strncat oraz strncmp, które obsługują co najwyżej n początkowych znaków swoich argumentów. Dla przykładu, funkcja strncpy(s,t,n) kopiuje co najwyżej n znaków tekstu z t do s. Pełne opisy tych funkcji znajdziesz w dodatku B.

Ćwiczenie 5.6. Napisz na nowo odpowiednie programy i ćwiczenia z poprzednich rozdziałów, stosując w nich wskaźniki zamiast indeksowania tablic. Interesujące

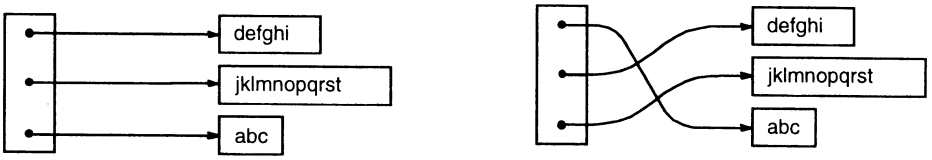
możliwości są zawarte w funkcjach: getline (rozdz. 1 i 4), atoi, itoa i ich wariantach (rozdz. 2, 3 i 4), reverse (rozdz. 3) oraz w strindex i getop (rozdz. 4).

5.6 Tablice wskaźników; wskaźniki do wskaźników

Wskaźniki same są zmiennymi, można więc z nich budować tablice tak samo, jak z innych zmiennych. Dla ilustracji napiszemy program ustawiający w porządku alfabetycznym zbiór wierszy tekstu. Będzie to okrojona wersja programu sort należącego do zestawu programów użytkowych systemu Unix.

W rozdziale 3 prezentowaliśmy funkcję porządkującą tablicę liczb całkowitych według metody Shell-sort, a w rozdz. 4 udoskonaliliśmy ją według metody szybkiego sortowania. Użyjemy tych samych algorytmów z tym, że teraz będziemy mieć do czynienia z wierszami tekstu. Wiersze są różnej długości i – w przeciwieństwie do liczb – nie można ich porównywać ani przesyłać za pomocą pojedynczej operacji. Potrzebujemy więc reprezentacji danych pozwalającej w sposób wygodny i efektywny obsługiwać wiersze tekstu o zmiennej długości.

Tu wkracza tablica wskaźników. Jeżeli przeznaczone do sortowania wiersze tekstu umieścić jeden za drugim w dużej tablicy znakowej, to każdy wiersz może być dostępny za pomocą wskaźnika do jego pierwszego znaku. Te wskaźniki można umieścić w innej tablicy. Wiersze porównuje się, przekazując ich wskaźniki funkcji strcmp. Gdy dwa nie uporządkowane wiersze należy zamienić miejscami, wówczas wystarczy w tablicy wskaźników zamienić miejscami ich wskaźniki, nie zaś same teksty.



Eliminuje to podwójny problem skomplikowanego zarządzania pamięcią oraz wysokich kosztów związanych z przesuwaniem samych wierszy tekstu.

Proces porządkowania przebiega w trzech etapach:

przeczytaj wszystkie wiersze z wejścia
uporządkuj je
wypisz wiersze we właściwej kolejności

Jak zwykle, najlepiej jest podzielić program na funkcje realizujące kolejne zadania wynikające z tego podziału, z główną funkcją sterującą działaniem pozostałych. Zostawimy na chwilę etap porządkowania i skoncentrujemy się nad zagadnieniem struktur danych oraz nad wejściem i wyjściem.

Zadaniem funkcji wejściowej jest zebranie i przechowanie wszystkich znaków z każdego wiersza, a także zbudowanie tablicy wskaźników do tych wierszy. Powinna ona także zliczyć wiersze, informacja ta będzie bowiem potrzebna przy ich porządkowaniu i wypisywaniu. Funkcja wejściowa może obsłużyć ograniczoną liczbę wierszy z wejścia. Jeśli więc jest ich zbyt dużo, może zwrócić niepoprawną ich liczbę, np. −1.

Funkcja wyjściowa służy jedynie do wypisania wierszy w kolejności, w jakiej występują w tablicy wskaźników.

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

#define MAXLINES 5000 /* maks. liczba wierszy do sortowania */

char *lineptr[MAXLINES]; /* wskaźniki do wierszy tekstu */

int readlines(char *lineptr[ ], int nlines);
void writelines(char *lineptr[ ], int nlines);

void qsort(char *lineptr[ ], int left, int right);

/* uporządkuj wiersze wejściowe */
main( )
{
    int nlines; /* liczba wczytanych wierszy */

    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort(lineptr, 0, nlines−1);
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("błąd: za dużo wierszy do sortowania\n");
        return 1;
    }
}
```

```
#define MAXLEN 1000 /* maks. długość wiersza wejściowego */
int getline(char *, int);
char *alloc(int);

/* readlines: wczytaj wiersze z wejścia */
int readlines(char *lineptr[ ], int maxlines)
{
    int len, nlines;
    char *p, line[MAXLEN];

    nlines = 0;
    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines || (p = alloc(len)) == NULL)
            return -1;
        else {
            line[len-1] = '\0'; /* usuń znak nowego wiersza */
            strcpy(p, line);
            lineptr[nlines++] = p;
        }
    return nlines;
}

/* writelines: wypisz wiersze na wyjście */
void writelines(char *lineptr[ ], int nlines)
{
    int i;

    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}
```

Funkcja getline pochodzi z punktu 1.9.

Główną nowością jest deklaracja lineptr:

```
char *lineptr[MAXLINES];
```

Mówi ona, że lineptr jest tablicą o MAXLINES elementach, z których każdy jest wskaźnikiem do znaków. Zatem lineptr[i] jest wskaźnikiem do znaku, a *lineptr[i] jest tym wskazywanym znakiem, czyli pierwszym znakiem i-tego przechowanego wiersza tekstu.

Sama lineptr jest nazwą tablicy, można ją więc traktować jak wskaźnik w ten sam sposób, co inne nazwy tablic we wcześniejszych przykładach, a funkcję writelines można napisać inaczej:

```
/* writelines: wypisz wiersze na wyjście */
void writelines(char *lineptr[ ], int nlines)
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
}
```

Początkowo *lineptr wskazuje na pierwszy wiersz; każde zwiększenie przenosi lineptr do następnego wskaźnika wiersza, podczas gdy wartość licznika nlines za każdym razem maleje.

Mając już funkcje wejścia i wyjścia, możemy zająć się sortowaniem. Metoda szybkiego sortowania z rozdz. 4 wymaga kilku małych zmian: należy zmodyfikować deklaracje, a dla operacji porównania wywołać funkcję strcmp. Podstawowy algorytm pozostaje taki sam, co dodaje nam pewności, że będzie nadal poprawny.

```
/* qsort: uporządkuj teksty v[left]...v[right] rosnąco */
void qsort(char *v[ ], int left, int right)
{
    int i, last;
    void swap(char *v[ ], int i, int j);

    if (left >= right)   /* nic nie rób, jeśli tablica zawiera */
        return;          /* mniej niż dwa elementy */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if (strcmp(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

Podobnie funkcja swap, zamieniająca miejscami wskazane elementy, wymaga jedynie banalnych zmian:

```
/* swap: zamień miejscami v[i] i v[j] */
void swap(char *v[ ], int i, int j)
{
    char *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Każdy element tablicy v (synonim lineptr) jest wskaźnikiem do znaków, toteż zmienna tymczasowa temp też musi być takim wskaźnikiem, aby poprawnie kopiować jeden na drugi.

Ćwiczenie 5.7. Zmień funkcję readlines tak, aby budowała wiersze w tablicy dostarczonej przez funkcję main, a nie w obszarach pamięci przydzielanych przez funkcję alloc. O ile szybszy jest program?

5.7 Tablice wielowymiarowe

W języku C występują prostokątne tablice wielowymiarowe, w praktyce jednak są one rzadziej stosowane niż tablice wskaźników. Pokażemy tu kilka ich właściwości.

Rozważmy problem przekształcania daty, tzn. zamiany dnia miesiąca na dzień roku i odwrotnie. Na przykład 1 marca jest 60 dniem roku zwykłego i 61 dniem roku przestępnego. Zdefiniujemy dwie funkcje: day_of_year przekształcającą miesiąc i dzień na dzień roku oraz month_day przekształcającą dzień roku na miesiąc i dzień. Ta ostatnia funkcja ma obliczyć dwie wartości, niech więc argumenty miesiąc i dzień będą wskaźnikami. Wówczas wywołanie

```
month_day(1988, 60, &m, &d)
```

przypisze zmiennej m wartość 2, a zmiennej d wartość 29 (29 lutego).

Obie funkcje wymagają tej samej informacji – tablicy liczby dni w każdym miesiącu („wrzesień ma 30 dni"...). Ponieważ liczby dni w miesiącu różnią się dla lat zwykłych i przestępnych (ang. *leap*), jest więc prościej rozdzielić je na dwa wiersze dwuwymiarowej tablicy, niż śledzić, co się dzieje z lutym podczas obliczeń. Tablica i funkcje przekształcające wyglądają następująco:

```
static char daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year: podaj dzień roku na podstawie miesiąca i dnia */
int day_of_year(int year, int month, int day)
{
    int i, leap;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; i < month; i++)
        day += daytab[leap][i];
    return day;
}

/* month_day: podaj miesiąc i dzień na podstawie dnia roku */
void month_day(int year, int yearday, int *pmonth, int *pday)
{
    int i, leap;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; yearday > daytab[leap][i]; i++)
        yearday -= daytab[leap][i];
    *pmonth = i;
    *pday = yearday;
}
```

Przypominamy, że arytmetyczną wartością wyrażenia logicznego, takiego jak przypisywane zmiennej leap, jest albo zero (fałsz), albo 1 (prawda), można jej więc użyć jako indeks tablicy daytab.

Tablica daytab powinna być zewnętrzna dla funkcji day_of_year i month_day, aby obie mogły z niej korzystać. Elementy tablicy zadeklarowaliśmy jako char, ilustrując tym samym legalne zastosowanie obiektów typu char do przechowywania małych liczb całkowitych nie reprezentujących żadnych znaków.

Tablica daytab jest pierwszą tablicą dwuwymiarową, z jaką mamy do czynienia. W języku C tablica dwuwymiarowa jest w rzeczywistości tablicą jednowymiarową, w której każdy z elementów jest tablicą. Zatem indeksy pisze się

```
daytab[i][j]   /* [wiersz][kolumna] */
```

a nie

```
daytab[i,j]   /* ŹLE */
```

Oprócz tej różnicy w zapisie tablicę dwuwymiarową traktuje się w bardzo podobny sposób, jak w innych językach programowania. Elementy są umieszczane w pamięci wierszami, a więc skrajnie prawy indeks (numer kolumny) zmienia się najszybciej, gdy elementy są obsługiwane w kolejności ich położenia w pamięci.

Tablicę inicjuje się listą wartości początkowych umieszczoną w nawiasach klamrowych; każdy wiersz tablicy dwuwymiarowej jest inicjowany odpowiednią podlistą. W tablicy daytab pierwszą kolumnę wypełniliśmy zerami, numery miesięcy można zatem zmieniać w sposób naturalny od 1 do 12, a nie od 0 do 11. Chociaż tracimy pamięć, jest to bardziej czytelne niż przesuwanie indeksów.

Gdy funkcji trzeba przekazać tablicę dwuwymiarową, wówczas w deklaracji odpowiedniego parametru funkcji musimy podać liczbę kolumn; liczba wierszy nie jest istotna, gdyż funkcji zostanie przekazany, tak jak poprzednio, wskaźnik do tablicy wierszy (gdzie każdy wiersz jest tablicą 13 liczb typu int[*]). W tym szczególnym przypadku jest to wskaźnik do obiektów, które są tablicami 13 liczb całkowitych. A zatem, jeśli tablica daytab ma być przekazana funkcji f, to deklaracja tej funkcji powinna mieć taką postać:

```
f(int daytab[2][13]) {...}
```

albo taką:

```
f(int daytab[ ][13]) {...}
```

ponieważ liczba wierszy nie jest ważna, lub też taką:

```
f(int (*daytab)[13]) {...}
```

Ta ostatnia deklaracja mówi, że parametr jest wskaźnikiem do tablicy 13 liczb całkowitych. Nawiasy okrągłe są w tym przypadku konieczne, gdyż nawiasy kwadratowe [ ] mają wyższy priorytet niż operator adresowania pośredniego *. Bez nawiasów okrągłych deklaracja

```
int *daytab[13]
```

wprowadza tablicę 13 wskaźników do obiektów całkowitych. Ogólniej mówiąc, tylko pierwszy wymiar (indeks) tablicy jest dowolny; wszystkie inne muszą być jawnie określone.

Dalszą dyskusję na temat skomplikowanych deklaracji prowadzimy w punkcie 5.12.

Ćwiczenie 5.8. W funkcjach day_of_year i month_day nie ma żadnego sprawdzania poprawności danych. Napraw tę usterkę.

[*] Autorzy zapomnieli już, że zadeklarowali daytab jako dwuwymiarową tablicę elementów typu char. Aby dalsze rozważania miały sens, musimy teraz przyjąć, że tablica ma elementy całkowite (int) – *Przyp. tłum.*

5.8 Inicjowanie tablic wskaźników

Spróbujemy napisać funkcję month_name(n), która zwraca wskaźnik do nazwy n-tego miesiąca. Jest to idealny przykład zastosowania wewnętrznej tablicy statycznej. Funkcja month_name zawiera prywatną tablicę tekstów i po wywołaniu wraca ze wskaźnikiem do odpowiedniego tekstu. Tutaj pokażemy, jak zainicjować taką tablicę.

Składnia jest podobna do tej z poprzednich inicjowań:

```
/* month_name: podaj nazwę n-tego miesiąca */
char *month_name(int n)
{
    static char *name[ ] = {
        "błędny miesiąc",
        "styczeń", "luty", "marzec",
        "kwiecień", "maj", "czerwiec",
        "lipiec", "sierpień", "wrzesień",
        "październik", "listopad", "grudzień"
    };

    return (n < 1 || n > 12) ? name[0] : name[n];
}
```

Deklaracja name jako tablicy wskaźników znakowych jest taka sama, jak deklaracja lineptr w przykładzie sortowania. Inicjatorem jest lista napisów. Każdy napis jest przypisywany odpowiedniej pozycji tablicy, a dokładniej: znaki i-tego napisu zostaną umieszczone gdzie indziej, natomiast elementowi tablicy name[i] przypisuje się wskaźnik do początku tego napisu. W deklaracji nie podano rozmiaru tablicy, wobec tego kompilator sam określa ten rozmiar, zliczając podane wartości początkowe.

5.9 Wskaźniki a tablice wielowymiarowe

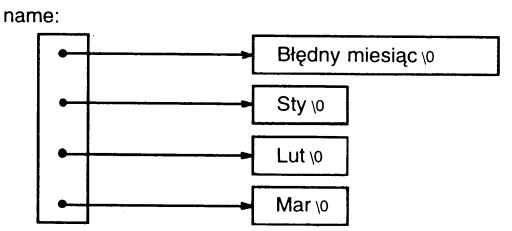
Początkujący programiści mają czasem kłopoty z przyswojeniem sobie różnicy między tablicą dwuwymiarową a tablicą wskaźników, taką jak name w poprzednim przykładzie. Po następujących definicjach:

```
int a[10][20];
int *b[10];
```

oba zapisy a[3][4] i b[3][4] są poprawnymi odwołaniami do pojedynczego obiektu typu int. Lecz a jest prawdziwą dwuwymiarową tablicą: zarezerwowano dla niej 200 miejsc o rozmiarze odpowiednim dla typu int, a do znalezienia elementu tablicy a[*wiersz*][*kolumna*] stosuje się tradycyjny przepis na wyliczenie indeksu dla tablic prostokątnych: 20 * *wiersz* + *kolumna*. Natomiast definicja b przydziela jedynie 10 miejsc pamięci na wskaźniki i nie inicjuje ich; nadanie wartości początkowych musi być zrobione jawnie – statycznie lub programowo. Przypuśćmy, że w istocie każdy z elementów tablicy b wskazuje na tablicę 20 elementów całkowitych. Wówczas mamy zarezerwowanych 200 miejsc pamięci na wartości całkowite plus te dziesięć komórek na wskaźniki. Ważną przewagą tablicy wskaźników jest to, że wiersze tablicy mogą mieć różne długości. W takim razie nie wszystkie elementy tablicy b muszą wskazywać na dwudziestoelementowe wektory; kilka z nich może wskazywać na wektory dwuelementowe, inne – na pięćdziesięcioelementowe, a jeszcze inne mogą w ogóle na nic nie wskazywać.

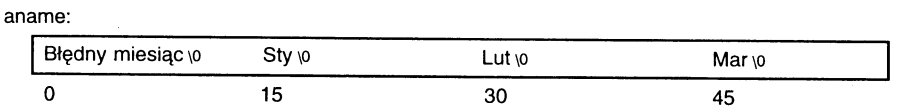
Chociaż ograniczyliśmy rozważania wyłącznie do obiektów całkowitych, w rzeczywistości z tablic wskaźników korzysta się najczęściej w sposób podobny do przedstawionego w funkcji month_name: zapamiętuje się w nich teksty o różnych długościach. Porównaj deklarację i ilustrację tablicy wskaźników:

```
char *name[ ] = { "Błędny miesiąc", "Sty", "Lut", "Mar" };
```



z deklaracją i ilustracją tablicy dwuwymiarowej:

```
char aname[][15] = {"Błędny miesiąc", "Sty", "Lut", "Mar" };
```



Ćwiczenie 5.9. Napisz na nowo funkcje day_of_year i month_day, stosując wskaźniki zamiast indeksowania tablic.

5.10 Argumenty wywołania programu

W środowiskach wyposażonych w C do uruchamianego programu można przekazać parametry, czyli argumenty wywołania, w wierszu polecenia wywołującego program. Działanie każdego programu rozpoczyna się wywołaniem funkcji main z dwoma argumentami. Pierwszy, umownie nazwany argc (od ang. *argument count*), jest liczbą argumentów, z jakimi program został wywołany; drugi, argv (od ang. *argument vector*), jest wskaźnikiem do tablicy zawierającej argumenty, każdy argument jako osobny tekst. Do operowania na tych tekstach zwyczajowo stosujemy kilka poziomów wskaźników.

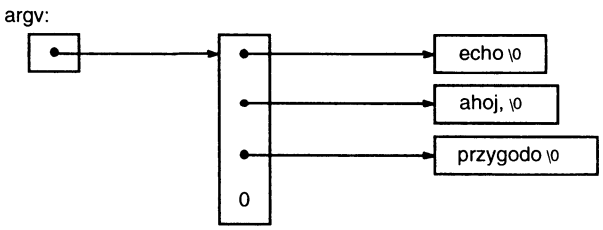
Najprostszym przykładem jest program echo, który jak echo powtarza argumenty swego wywołania, wypisując je w osobnym wierszu rozdzielone odstępami. Jeśli więc polecenie miało postać

```
echo ahoj, przygodo
```

to wynikiem programu będzie wiersz

```
ahoj, przygodo
```

Zgodnie z przyjętą konwencją argv[0] jest nazwą, z jaką program został wywołany, licznik argc jest więc co najmniej równy 1. Wartość argc równa 1 oznacza, że w wierszu polecenia po nazwie programu nie było argumentów. W powyższym przykładzie argc równa się 3, a argv[0], argv[1] i argv[2] wskazują odpowiednio napisy "echo", "ahoj," i "przygodo". Pierwszym prawdziwym argumentem jest argv[1], a ostatnim argv[argc–1]; dodatkowo standard wymaga, aby element argv[argc] był wskaźnikiem pustym, tj. równym NULL.



Pierwsza wersja programu echo traktuje argv jak tablicę wskaźników do znaków:

```
#include <stdio.h>

/* echo argumentów wywołania; wersja 1 */
main(int argc, char *argv[ ])
{
    int i;
```

```
    for (i = 1; i < argc; i++)
        printf("%s%s", argv[i], (i < argc-1) ? " " : "");
    printf("\n");
    return 0;
}
```

Zmienna argv jest wskaźnikiem do tablicy wskaźników, a więc – zamiast indeksować tę tablicę – możemy posłużyć się wskaźnikami. Następny wariant programu polega na zwiększaniu argv, którą potraktowano jako wskaźnik do wskaźnika do znaków, oraz na zmniejszaniu argc:

```
#include <stdio.h>

/* echo argumentów wywołania; wersja 2 */
main(int argc, char *argv[ ])
{
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) ? " " : "");
    printf("\n");
    return 0;
}
```

Zmienna argv jest wskaźnikiem do początku tablicy tekstów argumentów wywołania, a zatem zwiększając ją o jeden (++argv) sprawiamy, że wskazuje na oryginalny pierwszy argument argv[1], a nie na argv[0]. Każde następne zwiększenie przesuwa ten wskaźnik do kolejnego argumentu; *argv jest wskaźnikiem do znaków tworzących ten argument. W tym samym czasie jest zmniejszany licznik argc; gdy stanie się zerem, będzie to oznaczać, że nie ma już więcej argumentów do wypisania.

Wywołanie funkcji printf moglibyśmy napisać jeszcze inaczej:

```
printf((argc > 1) ? "%s " : "%s", *++argv);
```

co pokazuje, że argument opisujący format w funkcji printf może także być wyrażeniem.

Drugim przykładem niech będzie ulepszona wersja programu wyszukiwania wierszy zawierających wzorzec, opisanego w p. 4.1. Jak pamiętamy, wzorzec wyszukiwania był ukryty głęboko w programie, co zupełnie nas nie zadowalało. Opierając się na pochodzącym z systemu Unix programie grep zmienimy nasz program tak, aby wzorzec wyszukiwania był podawany jako pierwszy argument w poleceniu wywołania programu.

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);
```

```
/* find: wypisz wiersze pasujące do wzorca z 1. argumentu */
main(int argc, char *argv[ ])
{
    char line[MAXLINE];
    int found = 0;

    if (argc != 2)
        printf("Format wywołania: find wzorzec\n");
    else
        while (getline(line, MAXLINE) > 0)
            if (strstr(line, argv[1]) != NULL) {
                printf("%s", line);
                found++;
            }
    return found;
}
```

Pochodząca z biblioteki standardowej funkcja strstr(s,t) zwraca wskaźnik do pierwszego wystąpienia tekstu z t w s albo NULL, jeśli tekst t nie wystąpił w s. Jest ona zadeklarowana w nagłówku <string.h>.

Przerabiając dalej nasz model programu, możemy zaprezentować następne konstrukcje zawierające wskaźniki. Przypuśćmy, że wywołanie programu chcemy uzupełnić dwoma nieobowiązkowymi argumentami. Jeden mówi „wypisz wszystkie wiersze *z wyjątkiem* tych, które pasują do wzorca", a drugi – „poprzedź każdy wypisany wiersz jego numerem".

Ogólna konwencja w systemie Unix dla programów napisanych w C mówi, że argument wywołania rozpoczynający się znakiem minus wprowadza nieobowiązkowy sygnalizator lub parametr programu, czyli opcję. Jeżeli przyjmiemy, że –x (od ang. *except*: oprócz) oznacza sygnalizowanie odwrotnego działania programu, a –n (od ang. *number*: numer) oznacza żądanie numerowania wierszy, to polecenie

find –x –n *wzorzec*

spowoduje wypisanie każdego wiersza, który nie pasuje do wzorca, poprzedzonego swoim numerem wiersza. W programie należy przewidzieć możliwość podawania opcji w dowolnej kolejności, a właściwa część programu nie powinna zależeć od liczby podanych argumentów. Ponadto dużą wygodą dla użytkowników byłaby możliwość grupowania opcji, jak na przykład

find –nx *wzorzec*

A oto program:

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find: wypisz wiersze pasujące do wzorca z 1. obowiązkowego arg. */
main(int argc, char *argv[ ])
{
    char line[MAXLINE];
    long lineno = 0;
    int c, except = 0, number = 0, found = 0;

    while (--argc > 0 && (*++argv)[0] == '-')
        while (c = *++argv[0])
            switch (c) {
            case 'x':
                except = 1;
                break;
            case 'n':
                number = 1;
                break;
            default:
                printf("find: nieznana opcja %c\n", c);
                argc = 0;
                found = -1;
                break;
            }
    if (argc != 1)
        printf("Format wywołania: find -x -n wzorzec\n");
    else
        while (getline(line, MAXLINE) > 0) {
            lineno++;
            if ((strstr(line, *argv) != NULL) != except) {
                if (number)
                    printf("%ld:", lineno);
                printf("%s", line);
                found ++;
            }
        }
    return found;
}
```

Licznik argc zmniejsza się, a wskaźnik argv zwiększa przed każdym opcjonalnym argumentem. Jeżeli nie było błędów, to po zakończeniu pętli wartość argc określa liczbę argumentów, których jeszcze nie opracowano, a argv wskazuje na pierwszy z nich. Wobec tego licznik argc powinien być równy 1, a wskaźnik *argv powinien wskazywać na wzorzec. Zwróć uwagę, że *++argv jest wskaźnikiem do argumentu, a zatem (*++argv)[0] jest pierwszym znakiem tego argumentu. Nawiasy okrągłe są niezbędne, gdyż nawiasy kwadratowe tablicy [ ] wiążą silniej niż operator adresu * i operator zwiększania ++. Bez tych nawiasów wyrażenie byłoby równoważne z wyrażeniem *++(argv[0]). Właśnie tego wyrażenia użyliśmy w wewnętrznej pętli, której zadanie polega na maszerowaniu wzdłuż określonego ciągu znaków. W wewnętrznej pętli wyrażenie *++argv[0] zwiększa wskaźnik argv[0]!

Rzadko kiedy używa się bardziej skomplikowanych wyrażeń wskaźnikowych niż zaprezentowane; w takich przypadkach lepiej jest je podzielić na dwa lub trzy kroki.

Ćwiczenie 5.10. Napisz program expr obliczający wartość wyrażenia w Odwrotnej Notacji Polskiej, podanego w poleceniu wywołania programu: każdy operator lub operand jest oddzielnym argumentem wywołania. Na przykład polecenie

expr 2 3 4 + *

oblicza wartość 2×(3+4).

Ćwiczenie 5.11. Zmień programy entab i detab (napisane jako ćwiczenia w rozdz. 1) tak, aby argumentami ich wywołania mogły być listy punktów tabulacyjnych. Zastosuj domyślne punkty tabulacyjne, jeżeli nie podano argumentów.

Ćwiczenie 5.12. Rozbuduj programy entab i detab, aby akceptowały skrót

entab *−m +n*

oznaczający punkty tabulacyjne co *n* kolumn, poczynając od kolumny *m*. Wybierz dogodny (dla użytkownika) sposób postępowania programu w przypadku niekompletnej listy argumentów.

Ćwiczenie 5.13. Napisz program tail wypisujący *n* ostatnich wierszy tekstu podanego na wejściu. Niech *n* będzie domyślnie równe 10, ale tę wartość można zmienić za pomocą opcjonalnego argumentu wywołania programu. Zatem zlecenie

tail *−n*

spowoduje wypisanie *n* ostatnich wierszy. Program powinien działać sensownie niezależnie od tego, jak nierozsądne są dane wejściowe lub wartość *n*. Napisz

ten program tak, aby najlepiej wykorzystywał dostępną mu pamięć: wiersze powinny być gromadzone podobnie jak w programie sortującym z p. 5.6, nie zaś w dwuwymiarowej tablicy o stałym rozmiarze.

5.11 Wskaźniki do funkcji

Funkcje w języku C same nie są zmiennymi, ale istnieje możliwość definiowania wskaźników do funkcji. Takim wskaźnikom można nadawać wartości, umieszczać je w tablicach, przekazywać do funkcji, zwracać je jako wartość funkcyjną itp. Wyjaśnimy to dokładniej, zmieniając program sortowania napisany wcześniej w tym rozdziale: podanie opcji –n spowoduje, że program będzie porządkował wiersze wejściowe numerycznie, a nie leksykograficznie.

Sortowanie często składa się z trzech części: porównania ustalającego porządek między każdymi dwoma obiektami, zamiany przestawiającej te obiekty oraz algorytmu sortowania dokonującego porównań i zamian aż do uporządkowania obiektów we właściwej kolejności. Algorytm sortowania nie zależy od sposobu porównywania i zamieniania, zatem – przyłączając różne funkcje porównujące i zamieniające – możemy przeprowadzić porządkowanie według różnych kryteriów. Ten chwyt zastosujemy w naszym nowym programie sortującym.

Jak poprzednio, porównania leksykograficznego dwóch wierszy dokona funkcja strcmp; będziemy więc potrzebować funkcji numcmp, która porówna dwa wiersze na podstawie ich wartości liczbowych i da odpowiedź tego samego rodzaju, co funkcja strcmp. Funkcje te są zadeklarowane przed funkcją main, a wskaźnik do właściwej zostanie przekazany funkcji qsort. Celowo opuściliśmy obsługę błędów w argumentach, aby skupić się na głównym zadaniu.

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

#define MAXLINES 5000      /* maks. liczba wierszy do sortowania */
char *lineptr[MAXLINES];   /* wskaźniki do wierszy tekstu */

int readlines(char *lineptr[ ], int nlines);
void writelines(char *lineptr[ ], int nlines);

void qsort(void *lineptr[ ], int left, int right,
           int (*comp)(void *, void *));
int numcmp(char *, char *);
```

```
/* uporządkuj wiersze z wejścia */
main(int argc, char *argv[ ])
{
    int nlines;         /* liczba wczytanych wierszy */
    int numeric = 0; /* 1, jeśli sortowanie numeryczne */

    if (argc > 1 && strcmp(argv[1], "-n") == 0)
        numeric = 1;
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort((void **) lineptr, 0, nlines-1,
              (int (*)(void*, void*))(numeric ? numcmp : strcmp));
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("za dużo wierszy do sortowania\n");
        return 1;
    }
}
```

W wywołaniu funkcji qsort nazwy strcmp i numcmp reprezentują adresy tych funkcji. Są one znane jako funkcje, toteż operator adresu & jest zbędny, tak samo jak nie jest potrzebny przed nazwą tablicy.

Funkcję qsort napiszemy tak, aby mogła operować na danych dowolnego typu, a nie tylko na tekstach. Jak wynika z prototypu funkcji, qsort oczekuje argumentów, którymi są: tablica wskaźników, dwie liczby całkowite oraz funkcja porównująca o dwóch argumentach wskaźnikowych. Dla argumentów wskaźnikowych zastosowano ogólny typ void *. Za pomocą operacji rzutowania dowolny wskaźnik można przekształcić do typu void * i z powrotem bez utraty informacji, wobec tego funkcję qsort możemy wywoływać z dowolnymi argumentami, stosując rzutowanie do typu void *. Szczegółowy rzut argumentu funkcyjnego dopasowuje argumenty funkcji porównującej do prototypu qsort. Dla faktycznej reprezentacji argumentów nie ma to żadnego znaczenia, ale upewnia kompilator, że wszystko jest w porządku.

```
/* qsort: uporządkuj v[left]...v[right] rosnąco */
void qsort(void *v[ ], int left, int right,
           int (*comp)(void *, void *))
{
    int i, last;
    void swap(void *v[ ], int, int);
```

```
    if (left >= right) /* nic nie rób, jeśli tablica zawiera */
        return;        /* mniej niż dwa elementy */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if ((*comp) (v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1, comp);
    qsort(v, last+1, right, comp);
}
```

Z pewną uwagą powinniśmy prześledzić deklaracje. Czwartym parametrem funkcji qsort jest

```
int (*comp) (void *, void *)
```

który mówi, że comp jest wskaźnikiem do funkcji o dwóch argumentach typu void *, zwracającej wartość całkowitą.

Zastosowanie wskaźnika comp w wierszu programu

```
if ((*comp) (v[i], v[left]) < 0)
```

jest zgodne z jego deklaracją: comp jest wskaźnikiem do funkcji, *comp jest tą funkcją, a

```
(*comp) (v[i], v[left])
```

jest jej wywołaniem. Nawiasy są konieczne w celu zapewnienia prawidłowego powiązania składowych. Bez nich deklaracja

```
int *comp (void *, void *)   /* ŹLE */
```

mówi, że comp jest funkcją zwracającą wskaźnik do obiektów całkowitych, a to jest zupełnie co innego.

Znamy już funkcję strcmp porównującą dwa teksty. A oto funkcja numcmp, która porównuje dwa ciągi cyfr na podstawie numerycznych wartości ich początków, obliczonych za pomocą wywołania funkcji atof:

```
#include <stdlib.h>

/* numcmp: porównaj numerycznie s1 i s2 */
int numcmp(char *s1, char *s2)
{
    double v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}
```

Funkcja swap zamieniająca miejscami dwa wskaźniki jest w istocie identyczna z wersją prezentowaną wcześniej w tym rozdziale: różni się jedynie deklaracjami, w których teraz występuje void *.

```
void swap(void *v[ ], int i, int j)
{
    void *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Do programu sortującego możemy dodać wiele różnorodnych opcji; niektóre z nich prowokują do ćwiczeń.

Ćwiczenie 5.14. Zmień program sortujący tak, aby przyjmował opcję -r wskazującą na porządkowanie w odwrotnej (malejącej) kolejności. Upewnij się, że opcja -r poprawnie współpracuje z opcją -n.

Ćwiczenie 5.15. Dodaj opcję -f powodującą utożsamianie małych i wielkich liter alfabetu tak, aby przynależność liter do różnych rejestrów nie miała wpływu na sposób sortowania; na przykład z porównania a i A ma wynikać równość.

Ćwiczenie 5.16. Dodaj opcję –d (kolejność słownikowa) sprawiającą, że w porównaniu biorą udział tylko litery, liczby i odstępy. Upewnij się, że program działa poprawnie w połączeniu z opcją –f.

Ćwiczenie 5.17. Dodaj mechanizm obsługi pól tak, by można było przeprowadzić porządkowanie zawartości pól wewnątrz wierszy, stosując dla każdego pola niezależny zestaw opcji. (Skorowidz angielskiego oryginału tej książki był sortowany z użyciem opcji –df dla haseł i –n dla numerów stron.)

5.12 Skomplikowane deklaracje

Język C jest czasami „karcony'' za składnię deklaracji, a szczególnie za deklaracje wymagające wskaźników do funkcji. Składnia ta jest próbą pogodzenia deklaracji obiektu i jego użycia. Założenie to sprawdza się dla prostych przypadków, ale dla trudniejszych może być kłopotliwe, ponieważ deklaracji nie da się czytać od lewej strony do prawej, a także trzeba „nadużywać'' nawiasów. Różnica między deklaracjami

```
int *f();      /* f: funkcja zwracająca wskaźnik do int */
```

i

```
int (*pf)()  /* pf: wskaźnik do funkcji zwracającej int */
```

ilustruje ten problem: operator adresu * jest operatorem przedrostkowym i ma niższy priorytet niż nawiasy funkcji (), wobec czego „dodatkowe'' nawiasy są konieczne w celu wymuszenia prawidłowego powiązania składowych.

W praktyce rzadko spotyka się naprawdę skomplikowane deklaracje, ważne jest jednak to, by je należycie rozumieć i – jeśli zajdzie taka potrzeba – wiedzieć, jak je tworzyć. Jednym z dobrych sposobów syntetyzowania deklaracji jest metoda małych kroków, korzystająca z definicji typów typedef, o której jest mowa w p. 6.7. Tu proponujemy inny sposób: prezentujemy parę programów, które zamieniają poprawny tekst napisany w języku C na zwykły opis słowny i odwrotnie. Opis słowny czyta się od lewej strony do prawej.

Pierwszy z programów, dcl, jest bardziej skomplikowany*. Jego zadaniem jest zamiana deklaracji C na opis słowny, jak w następujących przykładach:

* Aby nie komplikować jeszcze bardziej tego programu, nie będziemy tu zajmować się deklinacją polskich słów. – *Przyp. tłum.*

```
char **argv
    argv: wskaźnik do wskaźnik do char
int (*daytab)[13]
    daytab: wskaźnik do tablica[13] o elementach int
int *daytab[13]
    daytab: tablica[13] o elementach wskaźnik do int
void *comp()
    comp: funkcja zwracająca wskaźnik do void
void (*comp)()
    comp: wskaźnik do funkcja zwracająca void
char (*(*x())[])()
    x: funkcja zwracająca wskaźnik do tablica[] o elementach
    wskaźnik do funkcja zwracająca char
char (*(*x[3])())[5]
    x: tablica[3] o elementach wskaźnik do funkcja zwracająca
    wskaźnik do tablica[5] o elementach char
```

Program dcl działa na podstawie gramatyki, którą wprowadza deklarator, szczegółowo opisany w dodatku A8.5. Tutaj przedstawiamy uproszczoną postać reguł składniowych:

deklarator:
*opcjonalne *-i bezpośredni-deklarator*

bezpośredni-deklarator:
nazwa
(deklarator)
bezpośredni-deklarator()
bezpośredni-deklarator[opcjonalny rozmiar]

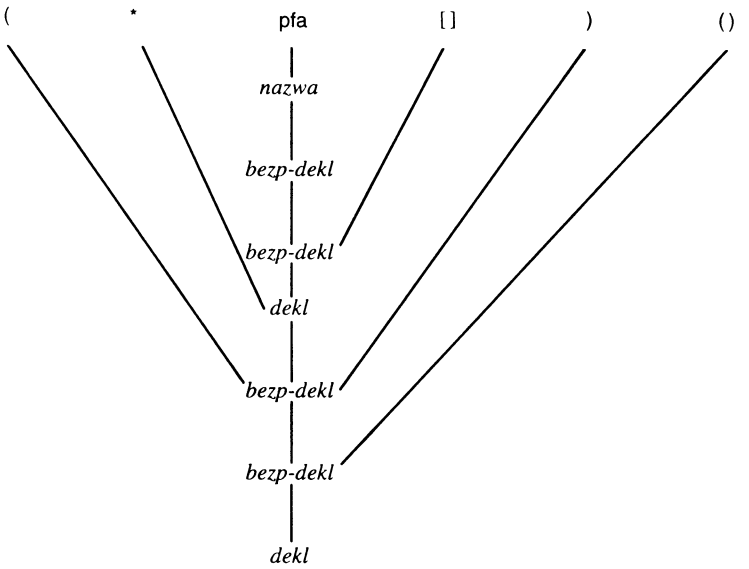
Mówiąc prościej, deklarator jest bezpośrednim deklaratorem być może poprzedzonym znakami *. Bezpośredni deklarator jest albo nazwą, albo deklaratorem ujętym w nawiasy okrągłe, albo bezpośrednim deklaratorem, po którym następują nawiasy okrągłe, albo bezpośrednim deklaratorem, po którym następują nawiasy kwadratowe ewentualnie zawierające rozmiar.

Z tej gramatyki można skorzystać przy rozbiorze deklaracji. Rozważmy na przykład deklarator

```
(*pfa[])()
```

Fragment pfa zostanie zidentyfikowany jako *nazwa*, a zatem jako *bezpośredni-deklarator*. Wobec tego konstrukcja pfa[] także jest bezpośrednim deklaratorem. Potem

konstrukcja *pfa[ ] zostanie rozpoznana jako *deklarator*, a zatem (*pfa[ ]) też jest bezpośrednim deklaratorem. Na koniec, konstrukcja (*pfa[ ])( ) jest bezpośrednim deklaratorem, a zatem jest deklaratorem. Ten rozbiór gramatyczny możemy także zilustrować graficznie za pomocą drzewa wywodu (w którym *deklarator* skróciliśmy do *dekl*, a *bezpośredni-deklarator* do *bezp-dekl*):



„Sercem" programu dcl jest para funkcji, dcl i dirdcl, które dokonują rozbioru gramatycznego deklaracji według podanej składni (dcl odpowiada jednostce składniowej *deklarator*, a dirdcl – jednostce składniowej *bezpośredni-deklarator*). Składnię zdefiniowano rekurencyjnie, zatem obie funkcje także wywołują się rekurencyjnie po rozpoznaniu fragmentu deklaracji. Taki program jest nazywany rekurencyjnie zstępującym analizatorem składniowym.

```
/* dcl: analiza składniowa deklaratora */
void dcl(void)
{
    int ns;

    for (ns = 0; gettoken() == '*'; ) /* zlicza *-i */
        ns++;
    dirdcl();
    while (ns-- > 0)
        strcat(out, " wskaźnik do");
}
```

```
/* dirdcl: analiza składniowa bezpośredniego deklaratora */
void dirdcl(void)
{
    int type;

    if (tokentype == '(') {          /* (deklarator) */
        dcl();
        if (tokentype != ')')
            printf("błąd: brak nawiasu )\n");
    } else if (tokentype == NAME ) /* nazwa zmiennej */
            strcpy(name, token);
    else
        printf("błąd: spodziewana nazwa lub (deklarator)\n");
    while ((type = gettoken()) == PARENS || type == BRACKETS)
        if (type == PARENS)              /* para nawiasów () */
            strcat(out, " funkcja zwracająca");
        else {                           /* para nawiasów [] */
            strcat(out, " tablica");
            strcat(out, token);          /* ew. rozmiar */
            strcat(out, " o elementach");
        }
}
```

Ponieważ te programy z zamierzenia są ilustracyjne, a nie „kuloodporne'', trzeba było poważnie ograniczyć możliwości programu dcl. Obsługuje on jedynie proste typy danych, jak char czy int. Nie zajmuje się typami argumentów funkcji ani kwalifikatorami jak np. const. Fałszywe odstępy wprawiają go w zakłopotanie. Program dcl nie szuka błędów zbyt dokładnie, więc niepoprawne deklaracje także wprawiają go w zakłopotanie. Takie udoskonalenia pozostawiono do opracowania jako ćwiczenia.

A oto zmienne globalne i główna procedura programu dcl:

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

#define MAXTOKEN 100

enum { NAME, PARENS, BRACKETS };

void dcl(void);
void dirdcl(void);
```

```
int gettoken(void);
int tokentype;                  /* typ ostatniego leksemu */
char token[MAXTOKEN];           /* tekst ostatniego leksemu */
char name[MAXTOKEN];            /* nazwa występująca w deklaracji */
char datatype[MAXTOKEN];        /* typ danych: char, int itp. */
char out[1000];                 /* wyjściowy opis słowny */

main() /*dcl: zamień deklaracje C na opis słowny */
{
    while (gettoken() != EOF) {   /* pierwszy leksem w wierszu */
        strcpy(datatype, token);  /* jest typem danych */
        out[0] = '\0';
        dcl();      /* analiza składniowa reszty wiersza */
        if (tokentype != '\n')
            printf("błąd składniowy\n");
        printf("%s: %s %s\n", name, out, datatype);
    }
    return 0;
}
```

Podczas czytania z wejścia funkcja gettoken najpierw pomija odstępy i znaki tabulacji, a potem buduje kolejny leksem; „leksemem'' jest tu nazwa, para nawiasów okrągłych, para nawiasów kwadratowych ewentualnie zawierających liczbę, a także każdy inny pojedynczy znak.

```
int gettoken(void) /* podaj następny leksem */
{
    int c, getch(void);
    void ungetch(int);
    char *p = token;

    while ((c = getch()) == ' ' || c == '\t')
        ;
    if (c == '(') {
        if ((c = getch()) == ')') {
            strcpy(token, "()");
            return tokentype = PARENS;
        } else {
            ungetch(c);
            return tokentype = '(';
        }
```

```
    } else if (c == '[') {
        for (*p++ = c; (*p++ = getch()) != ']'; )
            ;
        *p = '\0';
        return tokentype = BRACKETS;
    } else if (isalpha(c)) {
        for (*p++ = c; isalnum(c = getch()); )
            *p++ = c;
        *p = '\0';
        ungetch(c);
        return tokentype = NAME;
    } else
        return tokentype = c;
}
```

Funkcje getch i ungetch omówiliśmy w rozdz. 4.

Działanie odwrotne jest łatwiejsze, zwłaszcza jeśli nie przejmujemy się nadmiarowymi nawiasami. Program undcl zamienia opis słowny deklaracji, powiedzmy „x jest funkcją zwracającą wskaźnik do tablicy wskaźników do funkcji zwracających char", co możemy wyrazić tak:

```
x () * [] * () char
```

na deklarację w języku C

```
char (*(*x())[])()
```

Skrócona składnia danych na wejściu pozwala nam ponownie skorzystać z funkcji gettoken. Program undcl używa więc tych samych zmiennych zewnętrznych co dcl.

```
/* undcl: zamień opis słowny na deklarację */
main()
{
    int type;
    char temp[MAXTOKEN];

    while (gettoken() != EOF) {
        strcpy(out, token);
        while ((type = gettoken()) != '\n')
            if (type == PARENS || type == BRACKETS)
                strcat(out, token);
            else if (type == '*') {
                sprintf(temp, "(*%s)", out);
                strcpy(out, temp);
```

```
            } else if (type == NAME) {
                sprintf(temp, "%s %s", token, out);
                strcpy(out, temp);
            } else
                printf("niepoprawne dane wejściowe: %s\n", token);
            printf("%s\n", out);
    }
    return 0;
}
```

Ćwiczenie 5.18. Uodpornij program dcl na błędy w danych wejściowych.

Ćwiczenie 5.19. Zmień program undcl tak, aby w deklaracjach nie dodawał zbędnych nawiasów.

Ćwiczenie 5.20. Rozszerz możliwości programu dcl o obsługę typów argumentów funkcji, o rozpoznawanie kwalifikatorów takich jak const itp.

6 STRUKTURY

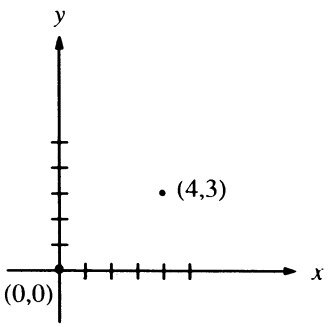
Struktura jest obiektem złożonym z jednej lub kilku zmiennych, być może różnych typów, dla wygody zgrupowanych pod jedną nazwą. (W niektórych językach, takich jak Pascal, struktury są nazywane „rekordami''.) Struktury ułatwiają zorganizowanie skomplikowanych danych, szczególnie w dużych programach, ponieważ grupę związanych ze sobą zmiennych pozwalają traktować jak jeden obiekt, a nie jak zestaw oddzielnych obiektów.

Jednym z tradycyjnych przykładów struktury jest pozycja listy płac: pracownik jest opisany przez zbiór takich atrybutów, jak nazwisko, adres, numer polisy ubezpieczeniowej, wynagrodzenie itp. Z kolei niektóre z tych atrybutów również mogą być strukturami: nazwisko ma kilka składowych, podobnie adres, a nawet wynagrodzenie. Inny przykład, bardziej typowy dla języka C, pochodzi z grafiki: punkt jest opisany przez parę współrzędnych, prostokąt jest opisany przez parę punktów itd.

Istotną zmianą w standardzie ANSI C jest zdefiniowanie operacji przypisania dla struktur: struktury można przypisywać jedna drugiej, kopiować jedną na drugą, przekazywać do funkcji i zwracać jako wartość funkcyjną. Na to wszystko większość kompilatorów pozwala już od wielu lat, ale teraz właściwości struktur zostały precyzyjnie sformułowane. Strukturom i tablicom automatycznym można również nadawać wartości początkowe.

6.1 Podstawowe wiadomości o strukturach

Zbudujemy teraz kilka struktur odpowiednich dla grafiki. Podstawowym obiektem jest punkt, który z założenia opisują dwie współrzędne całkowite: x i y.



Obie współrzędne można zadeklarować jako składowe struktury, na przykład tak:

```
struct point {
    int x;
    int y;
};
```

Słowo kluczowe struct rozpoczyna deklarację struktury, którą tworzy lista deklaracji zawartych między nawiasami klamrowymi. Po słowie struct może występować opcjonalna nazwa, zwana *etykietką struktury* (w naszym przykładzie point). Etykietka identyfikuje ten rodzaj struktury i może być później używana jako skrót dla tej części deklaracji, która występuje w nawiasach klamrowych.

O nazwanych zmiennych występujących w strukturze mówi się, że są *składowymi struktury*. Składowej struktury, etykietce i normalnej zmiennej (tj. nie będącej składową struktury) można nadać tę samą nazwę. Nie ma obawy o konflikt, ponieważ są one zawsze rozróżnialne przez kontekst, w jakim mogą się pojawić. Co więcej, takie same nazwy można nadać składowym różnych struktur, z tym jednak, że do dobrego tonu należy używanie takich samych nazw jedynie dla obiektów ściśle ze sobą związanych.

Deklaracja struct jest definicją typu. Po prawej klamrze kończącej listę składowych struktury może występować lista zmiennych, tak jak po każdym z podstawowych typów. Zatem definicja

```
struct {...} x, y, z;
```

odpowiada składniowo definicji

```
int x, y, z;
```

w tym sensie, że obie deklarują x, y i z jako zmienne wskazanego typu oraz rezerwują dla nich pamięć.

Deklaracja struktury, po której nie występuje lista zmiennych, nie rezerwuje żadnej pamięci; po prostu opisuje wzorzec lub kształt struktury. Jeżeli jednak deklaracja zawiera etykietkę, to etykietka może być później użyta w definicjach konkretnych „wcieleń" struktury. Na przykład, mając już deklarację struktury point, możemy zdefiniować zmienną pt jako strukturę tego typu:

```
struct point pt;
```

Strukturę można zainicjować dopisując na końcu jej definicji listę wartości początkowych jej składowych; każda wartość początkowa musi być wyrażeniem stałym:

```
struct point maxpt = { 320, 200 };
```

Automatyczną strukturę można także zainicjować za pomocą przypisania lub wywołania funkcji, która zwraca strukturę właściwego typu.

W wyrażeniach dostęp do składowej określonej struktury umożliwia konstrukcja

nazwa-struktury. składowa

Operator składowej struktury . (kropka) wiąże nazwę struktury z nazwą składowej. Aby wypisać współrzędne punktu pt, możemy zatem użyć instrukcji

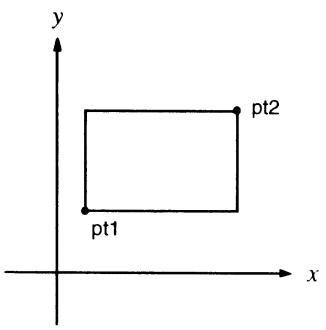
```
printf("%d,%d", pt.x, pt.y);
```

a do wyliczenia algebraicznej odległości między punktami pt i (0,0) zastosować znany algorytm

```
double dist, sqrt(double); /* sqrt: pierwiastek kwadratowy */

dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

Struktury mogą być zagnieżdżone. Jedną z reprezentacji prostokąta jest para punktów, które wskazują dwa przeciwległe wierzchołki:



```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

Struktura rect zawiera dwie struktury point. Jeśli zadeklarujemy screen (ekran) jako

```
struct rect screen;
```

to konstrukcja

```
screen.pt1.x
```

odnosi się do współrzędnej x punktu pt1, który jest składową zmiennej screen.

6.2 Struktury i funkcje

Dozwolonymi operacjami dla struktury są: przypisanie jej innej struktury w całości, skopiowanie jej w całości na inną strukturę, pobranie jej adresu za pomocą operatora & oraz odwołania do jej składowych. Przez kopiowanie i przypisanie rozumie się także przesyłanie argumentów funkcjom i zwracanie przez funkcje wartości. Struktur nie można natomiast porównywać. Strukturę można zainicjować listą stałych wartości początkowych jej składowych; automatyczną strukturę można też zainicjować za pomocą przypisania.

Spróbujemy zbadać właściwości struktur, pisząc kilka funkcji manipulujących punktami i prostokątami. Istnieją co najmniej trzy sposoby podejścia do zagadnienia: przekazywanie składników oddzielnie, przekazywanie całej struktury lub przekazywanie wskaźnika do tej struktury. Każdy z nich ma swoje wady i zalety.

Pierwsza funkcja, makepoint, z podanych dwóch wartości całkowitych buduje i zwraca strukturę typu point:

```
/* makepoint: utwórz punkt ze współrzędnych x i y */
struct point makepoint(int x, int y)
{
    struct point temp;

    temp.x = x;
    temp.y = y;
    return temp;
}
```

Zwróć uwagę na to, że nie ma konfliktu między nazwami argumentów i nazwami składowych struktury; ponowne użycie tych samych nazw faktycznie jeszcze bardziej uwypukla wzajemny związek między nimi.

Funkcję makepoint można teraz stosować do dynamicznego inicjowania dowolnej struktury lub do budowania strukturowych argumentów funkcji:

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);

screen.pt1 = makepoint(0, 0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
                   (screen.pt1.y + screen.pt2.y)/2);
```

Teraz zbudujemy kilka funkcji realizujących arytmetykę na punktach. Na przykład

```
/* addpoint: dodaj dwa punkty */
struct point addpoint(struct point p1, struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

Tutaj oba argumenty i zwracana wartość są strukturami. Zamiast tworzyć zmienną tymczasową, zwiększyliśmy składowe w strukturze p1 po to, by podkreślić, że struktury są tak samo przekazywane funkcji przez wartość, jak wszystkie inne argumenty.

Innym przykładem jest funkcja ptinrect, w której sprawdza się, czy punkt leży wewnątrz prostokąta. Przyjęliśmy tu konwencję, że prostokąt zawiera swoje krawędzie lewą i dolną, ale nie zawiera krawędzi górnej i prawej:

```
/* ptinrect: zwróć 1 jeśli p należy do r, 0 jeśli nie należy */
int ptinrect(struct point p, struct rect r)
{
    return p.x >= r.pt1.x && p.x < r.pt2.x
        && p.y >= r.pt1.y && p.y < r.pt2.y;
}
```

Ta funkcja zakłada, że prostokąt jest reprezentowany w standardowej postaci (kanonicznej), gdzie współrzędne punktu p1 są mniejsze niż współrzędne punktu p2. Następująca funkcja sprowadza prostokąt do postaci kanonicznej:

```
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))

/* canonrect: znormalizuj współrzędne prostokąta do postaci kanonicznej */
struct rect canonrect(struct rect r)
{
    struct rect temp;

    temp.pt1.x = min(r.pt1.x, r.pt2.x);
    temp.pt1.y = min(r.pt1.y, r.pt2.y);
    temp.pt2.x = max(r.pt1.x, r.pt2.x);
    temp.pt2.y = max(r.pt1.y, r.pt2.y);
    return temp;
}
```

Jeśli funkcji trzeba przekazać dużą strukturę, to przekazywanie wskaźnika jest zwykle bardziej skuteczne niż kopiowanie zawartości całej struktury. Wskaźniki do struktur są takimi samymi wskaźnikami, jak do zwykłych zmiennych. Deklaracja

```
struct point *pp;
```

mówi, że pp jest wskaźnikiem do struktury typu struct point. Jeśli pp wskazuje na strukturę typu point, to *pp jest taką strukturą, a (*pp).x oraz (*pp).y są jej składowymi. Posługując się wskaźnikiem pp, możemy na przykład napisać

```
struct point origin, *pp;

pp = &origin;
printf("punkt początkowy (%d,%d)\n", (*pp).x, (*pp).y);
```

W wyrażeniu (*pp).x nawiasy są konieczne, gdyż operator składowej struktury . ma priorytet wyższy niż operator adresowania pośredniego *. Wyrażenie *pp.x znaczy więc tyle, co *(pp.x), a to wyrażenie jest tutaj błędne, ponieważ x nie jest wskaźnikiem.

Wskaźników do struktur używa się tak często, że w języku występuje specjalna notacja umożliwiająca skrócony zapis. Jeśli p jest wskaźnikiem do struktury, to

```
p->składowa-struktury
```

jest odwołaniem do konkretnej składowej tej struktury. (Operator –> jest zbudowany ze znaku minus i bezpośrednio po nim następującego znaku >.) Teraz możemy więc napisać

```
printf("punkt początkowy (%d,%d)\n", pp->x, pp->y);
```

Oba operatory . i –> są lewostronnie łączne, toteż na przykład po takiej definicji

```
struct rect r, *rp = &r;
```

następujące cztery wyrażenia są równoważne:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

Operatory strukturowe . i –>, wraz z nawiasami okrągłymi ( ) wywołania funkcji i kwadratowymi [ ] indeksowania tablicy, znajdują się na szczycie hierarchii priorytetów, a więc najsilniej wiążą swoje argumenty. Dla przykładu po deklaracji

```
struct {
    int len;
    char *str;
} *p;
```

wyrażenie

```
++p->len
```

zwiększa zmienną len, a nie wskaźnik p, ponieważ stawiając nawiasy zgodnie z priorytetem operatorów otrzymamy ++(p->len). Stosując nawiasy można zmieniać przywiązanie argumentów: wyrażenie (++p)->len zwiększa p przed odwołaniem do len, a wyrażenie (p++)->len zwiększa p po takim odwołaniu. (W tym ostatnim wyrażeniu nawiasy są zbędne.)

Na tej samej zasadzie wyrażenie *p->str udostępnia to coś, na co wskazuje str, wyrażenie *p->str++ zwiększa str po udostępnieniu obiektu wskazywanego przez str (tak samo, jak konstrukcja *s++), wyrażenie (*p->str)++ zwiększa to coś, na co wskazuje str, a wyrażenie *p++->str zwiększa p po udostępnieniu obiektu wskazywanego przez str.

6.3 Tablice struktur

Napiszmy program zliczający wystąpienia każdego słowa kluczowego języka C. Potrzebujemy więc tablicy tekstów przeznaczonej na ich nazwy oraz tablicy liczb całkowitych przeznaczonej na ich liczniki. Jedną z możliwości jest użycie dwóch równoległych tablic keyword i keycount:

```
char *keyword[NKEYS];   /* słowa kluczowe */
int keycount[NKEYS];    /* liczniki tych słów */
```

Ale właśnie to, że tablice są równoległe, sugeruje możliwość innej organizacji – tablicy struktur. Informacje związane z każdym słowem kluczowym tworzą parę:

```
char *word;   /* słowo */
int count;    /* licznik */
```

Mamy więc tablicę par. Deklaracja strukturowa

```
struct key {
    char *word:
    int count;
} keytab[NKEYS];
```

deklaruje strukturowy typ key, definiuje tablicę keytab o elementach będących strukturami tego typu oraz rezerwuje dla nich pamięć. Każdy element tablicy jest strukturą. Można to również zapisać tak:

```
struct key {
    char *word;
    int count;
};

struct key keytab[NKEYS];
```

Tablica struktur keytab zawiera stały zbiór nazw, najprościej jest więc zrobić z niej zmienną zewnętrzną i zainicjować raz na zawsze przy jej definicji. Inicjowanie tablicy struktur przeprowadza się podobnie, jak poprzednio – po definicji podaje się ujętą w klamry listę wartości początkowych:

```
struct key {
    char *word;
    int count;
} keytab[ ] = {
    "auto", 0,
    "break", 0,
    "case", 0,
    "char", 0,
    "const", 0,
    "continue", 0,
    "default", 0,
    /* ... */
    "unsigned", 0,
    "void", 0,
    "volatile", 0,
    "while", 0
};
```

Wartości początkowe są wymienione parami odpowiadającymi składowym struktury. Precyzyjniej byłoby ująć w klamry wartości początkowe dla każdego „wiersza" tablicy (każdej struktury), na przykład:

```
{ "auto", 0 },
{ "break", 0 },
{ "case", 0 },
...
```

ale gdy wartościami początkowymi są proste stałe lub napisy i gdy podano wszystkie wartości, wówczas wewnętrzne klamry można opuścić. Jak zwykle, jeśli pominięto wymiar tablicy (nawiasy [ ] są puste) i podano listę wartości początkowych, to liczba elementów tablicy keytab zostanie wyliczona automatycznie.

Program zliczający wystąpienia słów kluczowych rozpoczyna się deklaracją tablicy keytab. Funkcja main czyta dane wejściowe, cyklicznie wywołując funkcję getword, która za każdym razem pobiera z wejścia jedno słowo. Każde słowo jest poszukiwane w tablicy keytab za pomocą funkcji wyszukiwania metodą bisekcji, znanej z rozdz. 3. Lista słów kluczowych musi być w tablicy uporządkowana rosnąco.

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

#define MAXWORD 100
```

```
int getword(char *, int);
int binsearch(char *, struct key *, int);

/* zlicz słowa kluczowe C */
main()
{
    int n;
    char word[MAXWORD]; /* słowo */

    while(getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((n = binsearch(word, keytab, NKEYS)) >= 0)
                keytab[n].count++;
    for (n = 0; n < NKEYS; n++)
        if (keytab[n].count > 0)
            printf("%4d %s", keytab[n].count, keytab[n].word);
    return 0;
}

/* binsearch: szukaj słowa w tab[0]...tab[n-1] */
int binsearch(char *word, struct key tab[ ], int n)
{
    int cond;
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low+high) / 2;
        if ((cond = strcmp(word, tab[mid].word)) < 0)
            high = mid - 1;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return -1;
}
```

Funkcję getword pokażemy za chwilę; na teraz wystarczy wiedzieć, że każde jej wywołanie znajduje na wejściu słowo, które jest kopiowane do tablicy nazwanej tak, jak jej pierwszy argument.

Wielkość NKEYS jest liczbą słów kluczowych zawartych w tablicy keytab. Chociaż możemy policzyć je ręcznie, będzie lepiej i bezpieczniej zrobić to z użyciem maszy-

ny, tym bardziej, że lista może ulec zmianie. Jedną z możliwości byłoby zakończenie listy wartości początkowych wskaźnikiem zerowym i przebiegnięcie tablicy keytab aż do końca.

Nie potrzeba jednak aż tak wiele, rozmiar tablicy jest bowiem całkowicie określony podczas kompilacji programu. Rozmiarem tablicy jest rozmiar jej jednego elementu pomnożony przez liczbę elementów, a więc liczba elementów wynosi dokładnie

rozmiar keytab / *rozmiar* struct key

Język C został wyposażony w jednoargumentowy operator zwany sizeof, który może służyć do obliczenia rozmiaru dowolnego obiektu. Wyrażenia

sizeof *obiekt*

i

sizeof(*nazwa typu*)

dają w wyniku wartość całkowitą równą rozmiarowi wskazanego obiektu lub typu w bajtach. (Dokładniej, sizeof podaje wartość całkowitą bez znaku o typie size_t zdefiniowanym w standardowym nagłówku <stddef.h>.) Obiektem może być zmienna, tablica lub struktura. Nazwą typu może być nazwa jednego z typów podstawowych, jak int lub double, albo nazwa typu pochodnego, jak struktura lub wskaźnik.

W naszym przypadku szukaną liczbą słów kluczowych jest rozmiar tablicy podzielony przez rozmiar jednego elementu. To obliczenie zastosowano w instrukcji #define do określenia wartości NKEYS:

```
#define NKEYS   (sizeof keytab / sizeof(struct key))
```

To samo obliczenie można napisać inaczej, dzieląc rozmiar tablicy przez rozmiar jej konkretnego elementu:

```
#define NKEYS   (sizeof keytab / sizeof keytab[0])
```

Druga postać ma tę przewagę, że nie trzeba jej zmienić po zmianie typu elementów tablicy.

Operatora sizeof nie można stosować w instrukcjach preprocesora #if, ponieważ w fazie prekompilacji nie ma analizy nazw typów. Wyrażenie w #define nie jest jednak obliczane przez preprocesor, zatem tutaj sizeof jest dozwolony.

Wróćmy teraz do funkcji getword. Napisaliśmy ją nieco bardziej ogólnie, niż było to konieczne dla naszego programu, ale nie jest przez to bardziej skomplikowana. Funkcja getword pobiera następne „słowo" z wejścia, przy czym słowem jest zarówno ciąg liter i cyfr zaczynający się od litery, jak i pojedynczy nie biały znak. Jej wartością funkcyjną jest pierwszy znak słowa lub EOF po napotkaniu końca pliku, lub też sam znak, jeśli nie jest literą.

```
/* getword: weź następne słowo lub znak z wejścia */
int getword(char *word, int lim)
{
    int c, getch(void);
    void ungetch(int);
    char *w = word;

    while (isspace(c = getch()))
        ;
    if (c != EOF)
        *w++ = c;
    if (! isalpha(c)) {
        *w = '\0';
        return c;
    }
    for ( ; --lim > 0; w++)
        if (! isalnum(*w = getch())) {
            ungetch(*w);
            break;
        }
    *w = '\0';
    return word[0];
}
```

Funkcja getword korzysta z funkcji getch i ungetch, które napisaliśmy w rozdz. 4. Kompletując znaki alfanumeryczne, getword czyta o jeden znak za dużo. Wywołanie funkcji ungetch oddaje ten znak z powrotem na wejście, umożliwiając wczytanie go przy ponownym wywołaniu. Funkcja getword korzysta z następujących makr zdefiniowanych w standardowym nagłówku <ctype.h>: isspace do pomijania białych znaków, isalpha do rozstrzygnięcia, czy dany znak jest literą, oraz isalnum do zidentyfikowania liter i cyfr.

Ćwiczenie 6.1. Nasza wersja funkcji getword nie obsługuje poprawnie znaków podkreślenia, stałych napisowych, komentarzy oraz instrukcji preprocesora. Napisz lepszą wersję tej funkcji.

6.4 Wskaźniki do struktur

Aby zilustrować niektóre z rozważań związanych ze wskaźnikami do struktur i tablicami struktur, napiszemy jeszcze raz program zliczający słowa kluczowe języka C, tym razem używając wskaźników zamiast indeksów tablic.

Zewnętrznej deklaracji tablicy keytab nie trzeba zmieniać, modyfikacji wymagają natomiast funkcje main i binsearch.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100

int getword(char *, int);
struct key *binsearch(char *, struct key *, int);

/* zlicz słowa kluczowe C; wersja wskaźnikowa */
main()
{
    char word[MAXWORD];
    struct key *p;

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((p = binsearch(word, keytab, NKEYS)) != NULL)
                p->count++;
    for (p = keytab; p < keytab + NKEYS; p++)
        if (p->count > 0)
            printf("%4d %s", p->count, p->word);
    return 0;
}

/* binsearch: szukaj słowa w tab[0]...tab[n-1] */
struct key *binsearch(char *word, struct key *tab, int n)
{
    int cond;
    struct key *low = &tab[0];
    struct key *high = &tab[n];
    struct key *mid;

    while (low < high) {
        mid = low + (high-low) / 2;
        if ((cond = strcmp(word, mid->word)) < 0)
            high = mid;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return NULL;
}
```

Warto tutaj omówić kilka spraw. Po pierwsze, deklaracja funkcji binsearch musi informować, że funkcja zwraca wskaźnik do struktury typu key, a nie wartość całkowitą; jest to zadeklarowane zarówno w prototypie funkcji, jak i w binsearch. Jeżeli funkcja ta znajdzie słowo, to zwraca wskaźnik do niego, jeżeli nie – zwraca NULL.

Po drugie, elementy tablicy keytab są teraz dostępne za pomocą wskaźników. To zaś wymaga znacznych zmian w funkcji binsearch.

Wartości początkowe zmiennych low i high są teraz wskaźnikami odnoszącymi się do początku tablicy i miejsca tuż za końcem tablicy.

Obliczenie środkowego elementu nie może już być takie proste

```
mid = (low+high) / 2    /* ŹLE */
```

ponieważ dodawanie do siebie wskaźników jest niedozwolone. Odejmowanie natomiast jest dozwolone: high–low oznacza liczbę elementów między wskaźnikami, a więc

```
mid = low + (high–low) / 2
```

ustawia mid tak, aby wskazywał element położony w połowie odległości między low i high.

Najpoważniejszą zmianą jest zaprojektowanie algorytmu tak, by z całą pewnością nie generował niepoprawnego wskaźnika lub by nie próbował odwoływać się do elementu spoza tablicy. Problem polega na tym, że oba adresy &tab[–1] i &tab[n] leżą poza granicami tablicy tab. Pierwszy z nich jest całkowicie błędny, błędem będzie również odwołanie pośrednie poprzez ten drugi. W definicji języka zagwarantowano jednak, że arytmetyka na wskaźnikach zadziała poprawnie dla pierwszego elementu poza końcem tablicy (to znaczy &tab[n]).

W funkcji main napisaliśmy

```
for (p = keytab; p < keytab + NKEYS; p++)
```

Jeśli p jest wskaźnikiem do struktury, to wszelkie związane z nim obliczenia wykonuje się z uwzględnieniem rozmiaru tej struktury. Zatem p++ zwiększa wskaźnik o potrzebną wielkość tak, aby otrzymać następny element tablicy struktur, a sprawdzenie warunku zatrzymuje pętlę we właściwym momencie.

Nie spodziewaj się jednak, że rozmiar struktury jest prostą sumą rozmiarów jej składowych. Ze względu na wymagania, stawiane przez różne obiekty na położenie w pamięci, może się okazać, że w strukturze będą nienazwane „dziury''. A zatem, jeśli na przykład typ char zajmuje jeden bajt, a typ int cztery bajty, to struktura

```
struct {
    char c;
    int i;
};
```

może – zamiast pięciu – równie dobrze wymagać ośmiu bajtów. Za pomocą operatora sizeof otrzymuje się właściwy rozmiar obiektu.

Na koniec krótka uwaga dotycząca formy pisania programów. Jeżeli funkcja zwraca wartość o skomplikowanym typie, powiedzmy wskaźnik do struktury, np.

```
struct key *binsearch(char *word, struct key *tab, int n)
```

to w takiej formie trudno zauważyć nazwę funkcji lub znaleźć ją za pomocą edytora tekstów. Niekiedy więc stosuje się alternatywny styl zapisu:

```
struct key *
binsearch(char *word, struct key *tab, int n)
```

Jest to rzecz gustu: wybierz ten styl, który lubisz, i stosuj go konsekwentnie.

6.5 Struktury odwołujące się do samych siebie

Przypuśćmy, że chcemy rozwiązać bardziej ogólny problem zliczania wystąpień *wszystkich* słów jakiegoś tekstu wejściowego. Lista słów nie jest z góry znana, nie możemy więc ich tak łatwo uporządkować i skorzystać z funkcji wyszukiwania metodą bisekcji. Nie możemy też dla każdego nadchodzącego słowa stosować przeszukiwania liniowego, by sprawdzić, czy już wcześniej wystąpiło; nasz program pracowałby zbyt długo. (Dokładniej: należy oczekiwać, że czas działania programu rośnie proporcjonalnie do kwadratu liczby słów wejściowych.) Jak zatem mamy zorganizować nasze dane, aby skutecznie uporać się z listą dowolnych słów?

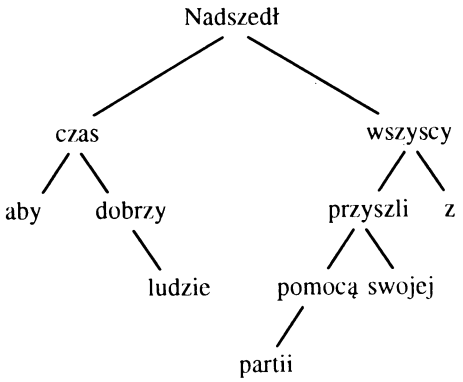
Jednym z rozwiązań jest sortowanie na bieżąco listy dotychczas wczytanych słów, wstawiając każde nowe słowo we właściwe miejsce listy. Nie należy tego robić przesuwając słowa w tablicy liniowej – to także zajmuje dużo czasu. Zastosujemy strukturę danych, którą zwykle nazywa się *drzewem binarnym*.

Drzewo składa się z „węzłów", po jednym dla każego różnego słowa; każdy węzeł zawiera:

- wskaźnik do tekstu słowa,
- licznik krotności wystąpień,
- wskaźnik do węzła lewego potomka,
- wskaźnik do węzła prawego potomka.

Żaden węzeł nie może mieć więcej niż dwóch potomków; może natomiast mieć jednego lub nie mieć żadnego.

Węzły w drzewie są uporządkowane tak, aby dla każdego węzła jego lewe poddrzewo zawierało słowa leksykograficznie mniejsze od słowa w tym węźle, a prawe poddrzewo – słowa większe. Prezentujemy drzewo binarne zdania „Nadszedł czas, aby wszyscy dobrzy ludzie przyszli z pomocą swojej partii" tak, jak je budowano wstawiając po kolei każde napotkane słowo.



Aby sprawdzić, czy nowo wczytane słowo znajduje się już w drzewie, porównujemy je ze słowem zapisanym w korzeniu drzewa. Jeżeli jest tym właśnie słowem, to mamy odpowiedź twierdzącą. Jeżeli nowe słowo jest mniejsze od słowa zawartego w węźle, to poszukiwanie kontynuujemy od lewego potomka; w przeciwnym przypadku badamy prawego potomka. Jeśli nie ma potomka po właściwej stronie znaczy to, że nowe słowo nie występuje w drzewie i że brakujący potomek jest właściwym miejscem dla tego słowa. Proces ten jest rekurencyjnie zstępujący (dziedziczny), gdyż poszukiwanie rozpoczęte w dowolnym węźle korzysta z poszukiwania rozpoczynającego się od jednego z jego potomków. Wobec tego zastosowanie rekurencyjnych procedur wstawiających i wypisujących słowa będzie najbardziej naturalne.

Wracając do opisu węzła, jego odpowiednią reprezentacją jest struktura o czterech składowych:

```
struct tnode {          /* węzeł drzewa */
    char *word;         /* wskaźnik do tekstu słowa */
    int count;          /* licznik wystąpień */
    struct tnode *left; /* lewy potomek */
    struct tnode *right; /* prawy potomek */
};
```

Taka rekurencyjna deklaracja węzła może wygląda ryzykownie, lecz jest zupełnie poprawna. Nie dopuszcza się, aby struktura zawierała w sobie swoje własne wcielenie, ale

```
struct tnode *left;
```

deklaruje left jako wskaźnik do struktury tnode, a nie jako samą strukturę.

Nieraz bywa potrzebna inna odmiana struktur odwołujących się do siebie samych: dwie struktury, które odwołują się do siebie nawzajem. Osiągnąć to można w następujący sposób:

```
struct t {
    ...
    struct s *p; /* p wskazuje na strukturę s */
};
struct s {
    ...
    struct t *q; /* q wskazuje na strukturę t */
};
```

Cały program jest zaskakująco mały dzięki garści wcześniej napisanych funkcji, na przykład getword. Funkcja main czyta kolejne słowa za pomocą getword i umieszcza je w drzewie za pomocą funkcji addtree.

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

#define MAXWORD 100
struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int getword(char *, int);

/* zliczanie wystąpień słów */
main()
{
    struct tnode *root; /* wskaźnik do korzenia drzewa */
    char word[MAXWORD];

    root = NULL;
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            root = addtree(root, word);
    treeprint(root);
    return 0;
}
```

Funkcja addtree jest rekurencyjna. Funkcja main wprowadza wczytane słowo na najwyższy poziom (korzeń) drzewa. Na każdym etapie słowo to jest porównywane ze słowem uprzednio zapamiętanym w danym węźle i, jeśli trzeba, przesłane w dół do lewego lub prawego poddrzewa poprzez rekurencyjne wywołanie funkcji addtree. Szukane słowo może już występować w drzewie, wówczas jest zwiększany jego licznik. Napotkanie zerowego wskaźnika poddrzewa oznacza, że trzeba utworzyć i w tym miejscu dołączyć do drzewa nowy węzeł. Po utworzeniu nowego węzła funkcja addtree zwraca jego adres, który jest wstawiany zamiast zerowego wskaźnika w węźle przodka.

```
struct tnode *talloc(void);
char *strdup(char *);

/* addtree: dodaj węzeł dla w; szukaj w p lub poniżej p */
struct tnode *addtree(struct tnode *p, char *w)
{
    int cond;

    if (p == NULL) {    /* w jest nowym słowem */
        p = talloc();     /* utwórz nowy węzeł */
        p->word = strdup(w);
        p->count = 1;
        p->left = p->right = NULL;
    } else if ((cond = strcmp(w, p->word)) == 0)
        p->count++;   /* powtórzone słowo */
    else if (cond < 0) /* mniejsze: do lewego poddrzewa */
        p->left = addtree(p->left, w);
    else                /* większe: do prawego poddrzewa */
        p->right = addtree(p->right, w);
    return p;
}
```

Pamięć dla nowego węzła udostępnia funkcja talloc, która zwraca wskaźnik do wolnego obszaru pamięci wystarczającego na przechowanie węzła. Nowe słowo jest kopiowane w bezpieczne miejsce przez funkcję strdup. (Dokładniej zajmiemy się tymi funkcjami za chwilę.) Wskaźnik do tego miejsca zapamiętuje się w węźle, inicjuje licznik wystąpień słowa oraz zeruje oba wskaźniki potomków. Ten fragment programu jest wykonywany tylko dla liści drzewa, gdy dodaje się nowy węzeł. Pominęliśmy tu (nieostrożnie) obsługę błędów sygnalizowanych wartościami zwracanymi przez funkcje strdup i talloc.

Funkcja treeprint wypisuje zawartość drzewa w uporządkowanej kolejności; dla każdego węzła zostanie najpierw wypisane jego lewe poddrzewo (wszystkie słowa mniejsze niż słowo w tym węźle), potem słowo tego węzła, a następnie jego prawe poddrzewo (wszystkie słowa większe). Jeśli nie czujesz się pewnie w rekurencji, sprawdź, jak działa treeprint dla narysowanego wcześniej drzewa.

```
/* treeprint: wypisz uporządkowane drzewo p */
void treeprint(struct tnode *p)
{
    if (p != NULL) {
        treeprint(p->left);
        printf("%4d %s\n", p->count, p->word);
        treeprint(p->right);
    }
}
```

Uwaga praktyczna: jeżeli drzewo staje się „niezrównoważone" z tego powodu, że słowa nadchodzą w nieprzypadkowej kolejności, to czas działania programu może być coraz dłuższy. W najgorszym przypadku, gdy nadchodzące słowa są już uporządkowane, program w sposób bardzo kosztowny symuluje przeszukiwanie liniowe. Istnieją konstrukcje ogólniejsze od drzew binarnych, przy których nie odczuwa się skutków takiego „złego zachowania się" danych, ale nie będziemy ich tu opisywać.

Zanim opuścimy ten przykład, jeszcze jedna dygresja dotycząca problemu przydziału pamięci. Oczywiście dobrze byłoby, gdyby w programie znajdował się tylko jeden dystrybutor pamięci – nawet wtedy, kiedy przydziela pamięć różnym obiektom. Ale jeśli ten jeden dystrybutor ma realizować żądania dotyczące, powiedzmy, wskaźników do znaków i wskaźników do struktury tnode, to musimy najpierw odpowiedzieć na dwa pytania. Po pierwsze, jak uwzględnić ograniczenia występujące w większości maszyn, dotyczące położenia różnego rodzaju obiektów w pamięci (na przykład liczby całkowite często muszą być umieszczane pod parzystym adresem)? I po drugie, jaka deklaracja poradzi sobie z tym, że dystrybutor z konieczności musi zwracać różne rodzaje wskaźników?

Ograniczenia dotyczące położenia w pamięci można, kosztem jednak pewnej straty pamięci, łatwo ominąć. Wystarczy przyjąć, że dystrybutor zawsze przydziela pamięć spełniającą *wszystkie* ograniczenia. Funkcja alloc z rozdz. 5 nie gwarantuje spełnienia jakichkolwiek ograniczeń, skorzystamy więc ze standardowej, bibliotecznej funkcji malloc, która je respektuje. W rozdziale 8 pokażemy jeden ze sposobów realizacji funkcji malloc.

Pytanie dotyczące deklaracji typu takiej funkcji, jak malloc, jest drażliwe w każdym języku, który poważnie podchodzi do kontroli zgodności typów. W języku C właściwe

postępowanie polega najpierw na zadeklarowaniu malloc jako funkcji zwracającej wskaźnik do typu void, a następnie na wymuszeniu właściwego typu tego wskaźnika za pomocą operacji rzutowania. Funkcja malloc i procedury pokrewne są zadeklarowane w standardowym nagłówku <stdlib.h>. A zatem naszą funkcję talloc można napisać tak:

```
#include <stdlib.h>

/* talloc: utwórz węzeł */
struct tnode *talloc(void)
{
    return (struct tnode *) malloc(sizeof(struct tnode));
}
```

Funkcja strdup po prostu kopiuje tekst ze swojego argumentu w bezpieczne miejsce, otrzymane również przez wywołanie funkcji malloc:

```
/* strdup: sporządź kopię s */
char *strdup(char *s)
{
    char *p;

    p = (char *) malloc(strlen(s)+1); /* +1 dla '\0' */
    if (p != NULL)
        strcpy(p, s);
    return p;
}
```

Funkcja malloc zwraca NULL, jeśli brakuje miejsca w pamięci; funkcja strdup przekaże tę wartość dalej, pozostawiając obsługę takich błędów procedurze wywołującej.

Pamięć otrzymaną za pomocą malloc można zwolnić do ponownego użycia, wywołując funkcję free; zajrzyj do rozdz. 7 i 8.

Ćwiczenie 6.2. Napisz program, który czyta tekst programu napisanego w C i wypisuje w porządku alfabetycznym wszystkie grupy nazw zmiennych o identycznych sześciu początkowych znakach i różniących się którymkolwiek z dalszych znaków. Nie zliczaj słów występujących w stałych napisowych i w komentarzach. Niech liczba 6 będzie parametrem, który można zmienić w wierszu wywołania programu.

Ćwiczenie 6.3. Napisz program tworzący skorowidz, tj. wypisujący listę wszystkich słów dokumentu i dla każdego słowa listę numerów wierszy, w których to słowo wystąpiło. Ze skorowidza usuń słowa-ozdobniki w rodzaju „ten", „lub" itp.

Ćwiczenie 6.4. Napisz program, który zlicza różne słowa podane na wejściu i wypisuje je uporządkowane według malejącej krotności ich wystąpień. Każde słowo poprzedź jego krotnością.

6.6 Przeglądanie tablic

Aby zilustrować dalsze właściwości struktur, napiszemy teraz pakiet podprogramów służących do przeglądania tablic. Będą to typowe narzędzia, z jakimi możemy się zetknąć przy obsłudze tablic symboli w makrogeneratorach i kompilatorach. Jako przykład weźmy instrukcję #define. Po napotkaniu wiersza w rodzaju:

```
#define IN 1   /* wewnątrz */
```

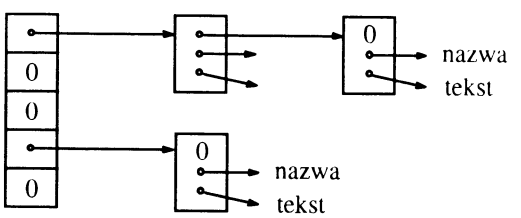
nazwa IN i zastępujący ją tekst 1 zostaną zapamiętane w odpowiedniej tablicy. Od tej pory, jeżeli nazwa IN pojawi się w jakiejś instrukcji, np.

```
state = IN;   /* stan */
```

to musi być zastąpiona przez znak 1.

Mamy dwie procedury działające na nazwach i zastępujących je tekstach. Funkcja install(s,t) rejestruje nazwę s i odpowiadający jej tekst t w pewnej tablicy; s oraz t są po prostu ciągami znaków. Druga funkcja – lookup(s) – przegląda tę tablicę w poszukiwaniu nazwy s i zwraca wskaźnik do miejsca, w którym znaleziono nazwę, lub NULL, jeżeli takiej nazwy nie ma w tablicy.

Zastosujemy algorytm przeszukiwania rozproszonego (ang. *hash*): nadchodzącą nazwę przekształca się na pewną niewielką liczbę nieujemną (wartość rozproszenia), której następnie używa się do indeksowania tablicy wskaźników. Element tablicy wskazuje na początek połączonej w łańcuch listy opisów nazw z tą samą wartością rozproszenia. Jest on równy NULL, jeśli nie ma nazwy o takiej wartości.



Ogniwo łańcucha jest strukturą zawierającą wskaźniki do nazwy, do zastępującego ją tekstu oraz do następnego opisu w łańcuchu. Zerowy wskaźnik następnego ogniwa oznacza koniec listy.

```
struct nlist {   /* ogniwo łańcucha */
    struct nlist *next;   /* następne ogniwo */
    char *name;           /* definiowana nazwa */
    char *defn;           /* zastępujący ją tekst */
};
```

Tablica wskaźników ma postać

```
#define HASHSIZE 101

static struct nlist *hashtab[HASHSIZE]; /* tablica wskaźników */
```

Funkcja rozpraszająca, używana przez funkcje lookup i install, dodaje wartość kolejnego znaku tekstu do wymieszanej kombinacji poprzednich znaków, dając w wyniku resztę z dzielenia (modulo) tak obliczonej wartości przez rozmiar tablicy. Ta funkcja nie jest najlepszą z funkcji rozpraszających, ma jednak tę zaletę, że jest krótka i skuteczna.

```
/* hash: daj wartość rozproszenia dla tekstu s */
unsigned hash(char *s)
{
    unsigned hashval;

    for (hashval = 0; *s != '\0'; s++)
        hashval = *s + 31 * hashval;
    return hashval % HASHSIZE;
}
```

Arytmetyka na liczbach bez znaku (unsigned) daje gwarancję, że obliczona wartość rozproszenia jest liczbą nieujemną.

Wynikiem rozproszenia jest indeks początkowego wskaźnika w tablicy hashtab; jeśli tylko dany tekst jest gdziekolwiek zapisany, znajdzie się on w w łańcuchu opisów rozpoczynającym się od tego miejsca. Przeszukiwania dokonuje funkcja lookup. Jeżeli dany tekst już kiedyś wystąpił, to funkcja zwraca wskaźnik do jego opisu, w przeciwnym przypadku zwraca NULL.

```
/* lookup: szukaj tekstu z s w tablicy hashtab */
struct nlist *lookup(char *s)
{
    struct nlist *np;

    for (np = hashtab[hash(s)]; np != NULL; np = np->next)
        if (strcmp(s, np->name) == 0)
            return np;   /* znaleziono */
    return NULL;         /* nie znaleziono */
}
```

Pętla for w funkcji lookup jest standardowym zwrotem języka C, służącym do maszerowania wzdłuż listy połączonych ze sobą obiektów:

```
for (ptr = head; ptr != NULL; ptr = ptr->next)
    ...
```

Funkcja install używa funkcji lookup do rozstrzygania, czy wprowadzana nazwa już wystąpiła; jeżeli tak, to nowa definicja musi zastąpić starą. W przeciwnym przypadku jest tworzone nowe ogniwo łańcucha.

```
struct nlist *lookup(char *);
char *strdup(char *);

/* install: wstaw (name, defn) do tablicy hashtab */
struct nlist *install(char *name, char *defn)
{
    struct nlist *np;
    unsigned hashval;

    if ((np = lookup(name)) == NULL) {      /* nie znaleziono */
        np = (struct nlist *) malloc(sizeof(*np));
        if (np == NULL || (np->name = strdup(name)) == NULL)
            return NULL;
        hashval = hash(name);
        np->next = hashtab[hashval];
        hashtab[hashval] = np;
    } else      /* już była */
        free((void *) np->defn);      /* zwolnij starą definicję */
    if ((np->defn = strdup(defn)) == NULL)
        return NULL;
    return np;
}
```

Ćwiczenie 6.5. Napisz funkcję undef, która usuwa nazwę i jej definicję z tablicy obsługiwanej przez funkcje lookup i install.

Ćwiczenie 6.6. Skonstruuj prostą wersję preprocesora obsługującego intrukcje #define (bezargumentowe), działającego na programach napisanych w języku C. Zastosuj funkcje opisane w tym rozdziale. Możesz też skorzystać z funkcji getch i ungetch.

6.7 Deklaracja typedef

W języku C wprowadzono mechanizm, zwany typedef, do tworzenia nowych nazw typów danych. Na przykład deklaracja

```
typedef int Lenght;  /* Długość */
```

tworzy dla typu int synonim Lenght. Z typu Lenght można korzystać w deklaracjach, rzutowaniach itp. dokładnie tak samo, jak z typu int:

```
Lenght len, maxlen;
Lenght *lenghts[ ];
```

Podobnie deklaracja

```
typedef char *String;  /* Tekst */
```

wprowadza synonim String dla typu char *, z którego można później korzystać w deklaracjach i rzutowaniu, np.

```
String p, lineptr[MAXLINES], alloc(int);
int strcmp(String, String);
p = (String) malloc(100);
```

Zwróć uwagę, że typ deklarowany w typedef pojawia się w miejscu nazwy zmiennej, a nie tuż za słowem typedef. Składniowo słowo typedef odpowiada klasie pamięci, jak extern, static itp. Aby wyróżnić nazwy definiowanych typów, rozpoczynamy je wielką literą.

Bardziej skomplikowanym przykładem może być użycie typedef do definiowania węzłów drzewa opisanego wcześniej w tym rozdziale:

```
typedef struct tnode *Treeptr;  /* Wskaźnik do węzła */

typedef struct tnode {   /* węzeł drzewa: */
    char *word;           /* wskaźnik do tekstu słowa */
    int count;            /* licznik wystąpień */
    Treeptr left;         /* lewy potomek */
    Treeptr right;        /* prawy potomek */
} Treenode;              /* Węzeł */
```

Obie deklaracje tworzą dwa nowe słowa kluczowe typów: Treenode (struktura) oraz Treeptr (wskaźnik do takiej struktury). Zatem procedurę talloc można napisać tak:

```
Treeptr talloc(void)
{
    return(Treeptr) malloc(sizeof(Treenode));
}
```

Należy podkreślić, że deklaracja typedef w żadnym sensie nie tworzy nowego typu; ona po prostu daje nową nazwę dla pewnego już istniejącego typu. Nie ma tu także żadnej nowej semantyki: zmienne deklarowane w ten sposób mają dokładnie te same właściwości co zmienne, których deklaracje zostały podane szczegółowo. Deklaracja typedef jest w istocie podobna do #define z tą różnicą, że jest interpretowana przez kompilator, a więc może uczestniczyć w podstawieniach tekstowych, które są poza możliwościami preprocesora języka C. Na przykład deklaracja

```
typedef int (*PFI)(char *, char *);
```

tworzy typ o nazwie PFI jako „wskaźnik do funkcji zwracającej wartość typu int (o dwóch argumentach typu char *)". Możemy go stosować w takich kontekstach, jak

```
PFI strcmp, numcmp;
```

(zamiast prototypów tych funkcji) w programie sortującym z rozdz. 5.

Poza skutkiem czysto estetycznym są dwa główne powody przemawiające za używaniem deklaracji typedef. Pierwszy dotyczy parametryzacji programu w związku z problemem przenoszenia oprogramowania. Jeżeli deklaracje typedef są stosowane dla tych typów danych, które mogą zależeć od maszyny, to tylko te deklaracje będą wymagać zmiany przy przenoszeniu programu. Jedną z częściej spotykanych sytuacji jest użycie typedef do zdefiniowania nazw różnych wielkości całkowitych, a następnie dokonanie

wyboru odpowiednich spośród typów short, int lub long dla każdej docelowej maszyny. Dobrymi przykładami są typy size_t i ptrdiff_t pochodzące z biblioteki standardowej.

Drugą przyczyną stosowania typedef jest to, że taka deklaracja lepiej komentuje program – typ o nazwie Treeptr można łatwiej zrozumieć niż zadeklarowany wprost jako wskaźnik do skomplikowanej struktury.

6.8 Unie

Unia jest zmienną, która (w różnych momentach) może zawierać obiekty różnych typów i rozmiarów, przy czym to kompilator dba o zadośćuczynienie wymaganiom dotyczącym rozmiaru i położenia w pamięci. Unie pozwalają manipulować danymi różnego rodzaju w tym samym miejscu pamięci bez wprowadzania do programu informacji o ich cechach zależnych od maszyny. Unie są podobne do rekordów wariantowych w Pascalu.

Jako przykład, który można znaleźć w programie zarządzającym tablicą symboli kompilatora, przypuśćmy, że stała może być typu int, float lub wskaźnikiem znakowym. Wartość konkretnej stałej należy zapamiętać w zmiennej o właściwym typie, jednakże dla obsługi tablicy najwygodniej jest, aby każda taka wartość zajmowała obszar tej samej wielkości i (niezależnie od jej typu) była wstawiana w to samo miejsce pamięci. Taki jest właśnie cel unii – udostępnić jedną zmienną, która jest uprawniona do przechowywania wartości kilku różnych typów. Składnia unii jest wzorowana na strukturach:

```
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```

Zmienna u będzie wystarczająco obszerna, aby pomieścić w sobie wartość największego z tych trzech typów; właściwy rozmiar zależy od implementacji. Zmiennej u można przypisać wartość każdego z tych typów, a następnie używać jej w wyrażeniach dopóty, dopóki użycie to jest prawidłowe: typ wartości pobieranej musi być typem tej wartości, która została ostatnio przypisana. Do programisty należy kontrola typu obiektu aktualnie znajdującego się w unii; jeśli coś zostanie zapamiętane z jednym typem, a pobrane z innym, to wynik zależy od implementacji.

Postać odwołania do składowych unii jest następująca:

nazwa-unii. składowa

lub

wskaźnik-do-unii->składowa

czyli identyczna, jak dla struktur. Jeżeli zmienna utype służy do kontroli typu wartości zmiennej u, to możemy się spotkać na przykład z takim fragmentem programu:

```
if (utype == INT)
    printf("%d\n", u.ival);
else if (utype == FLOAT)
    printf("%f\n", u.fval);
else if (utype == STRING)
    printf("%s\n", u.sval);
else
    printf("zły typ %d w utype\n", utype);
```

Unie mogą występować w strukturach i tablicach, i vice versa. Notacja dla odwołania do składowej unii w strukturze (lub odwrotnie) jest taka sama, jak dla zagnieżdżonych struktur. Na przykład dla tablicy struktur:

```
struct {
    char *name;    /* nazwa symbolu */
    int flags;     /* znaczniki stanu */
    int utype;     /* typ wartości */
    union {        /* wartość */
        int ival;
        float fval;
        char *sval;
    } u;
} symtab[NSYM];   /* tablica symboli */
```

odwołanie do składowej ival ma postać

```
symtab[i].u.ival
```

a odwołanie do pierwszego znaku tekstu wskazywanego przez sval można zapisać na dwa równoważne sposoby

```
*symtab[i].u.sval
symtab[i].u.sval[0]
```

W konsekwencji unia jest strukturą, w której wszystkie składowe są umieszczone bez przesunięcia względem jej początku i która jest dostatecznie duża, aby pomieścić „najobszerniejszą'' ze składowych. Położenie unii w pamięci spełnia przy tym wymagania typów wszystkich jej składowych. Operacje dozwolone dla struktur są również dozwolone dla unii: przypisywanie i kopiowanie unii traktowanych jako całość, pobieranie adresu unii i dostęp do ich składowych.

Unię można zainicjować jedynie wartością o typie jej pierwszej składowej, a zatem naszą unię u można zainicjować tylko wartością całkowitą.

Dystrybutor pamięci, opisany w rozdz. 8, pokazuje, jak można skorzystać z unii do wymuszenia określonego położenia zmiennej w pamięci maszyny.

6.9 Pola bitowe

Gdy trzeba oszczędzać pamięć, wówczas pakowanie kilku obiektów w jedno słowo maszyny może się okazać koniecznością. Postępuje się tak zwykle ze zbiorami jednobitowych znaczników opisujących stan obiektów, na przykład w tablicach symboli kompilatora. Narzucone z zewnątrz formaty danych, jak w przypadku łączy z urządzeniami zewnętrznymi, często także wymagają dostępu do kawałków słów.

Wyobraźmy sobie fragment kompilatora zarządzający tablicą symboli. Z każdym identyfikatorem w programie są związane pewne informacje, na przykład, że jest lub nie jest słowem kluczowym, że jest lub nie jest zewnętrzny, statyczny itp. Najbardziej zwartym sposobem kodowania takich informacji jest gromadzenie jednobitowych znaczników w pojedynczym obiekcie typu char lub int.

Zazwyczaj robi się to przez zdefiniowanie zbioru „masek'' odpowiadających właściwym pozycjom bitów, na przykład w ten sposób:

```
#define KEYWORD 01  /* słowo kluczowe */
#define EXTERNAL 02  /* obiekt zewnętrzny */
#define STATIC   04  /* obiekt statyczny */
```

lub też

```
enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };
```

Liczby w maskach muszą być potęgami dwójki. „Żonglowanie'' tymi bitami staje się potem sprawą dobrania odpowiednich operatorów przesuwania, maskowania i dopełniania, znanych nam z rozdz. 2.

Pewne zwroty pojawiają się bardzo często, np.

```
flags |= EXTERNAL | STATIC;
```

ustawia bity EXTERNAL i STATIC w zmiennej flags,

```
flags &= ~(EXTERNAL | STATIC);
```

kasuje te bity, a warunek

```
if ((flags & (EXTERNAL | STATIC)) == 0) ...
```

jest prawdziwy, kiedy oba bity są skasowane.

Chociaż takie zwroty są chętnie stosowane, język C oferuje alternatywny mechanizm definiowania i bezpośredniej obsługi pól wewnątrz słowa bez korzystania z bitowych operatorów logicznych. *Polem bitowym*, lub krócej *polem*, jest zbiór przylegających do siebie bitów znajdujących się w jednej, zależnej od implementacji, jednostce pamięci zwanej „słowem''. Składnia definicji pola i odwołania do pola jest wzorowana na strukturach. Na przykład, zestaw symboli zdefiniowanych powyżej za pomocą #define można zastąpić definicją trzech pól:

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern  : 1;
    unsigned int is_static  : 1;
} flags;
```

Definiuje ona zmienną o nazwie flags, zawierającą trzy jednobitowe pola. Liczba występująca po dwukropku oznacza rozmiar pola w bitach. Pola zadeklarowano jako unsigned int, aby zagwarantować, że są wielkościami bez znaku.

Odwołania do poszczególnych pól są takie same, jak do innych składowych struktury: flags.is_keyword, flags.is_extern itp. Pola zachowują się jak małe zmienne całkowite i mogą występować w wyrażeniach arytmetycznych na równi z innymi obiektami całkowitymi. Zatem poprzednie przykłady można teraz napisać w sposób bardziej naturalny: wyrażenie

```
flags.is_extern = flags.is_static = 1;
```

ustawia bity,

```
flags.is_extern = flags.is_static = 0;
```

kasuje je, a warunek

```
if (flags.is_extern == 0 && flags.is_static == 0) ...
```

sprawdza, czy oba są skasowane.

Prawie wszystko, co wiąże się z polami bitowymi, zależy od implementacji. To, czy pole może przekraczać granice słowa jest zależne od implementacji. Pola nie muszą mieć nazw; nie nazwane pola (tylko dwukropek i rozmiar) są używane do zapychania dziur (nie wykorzystanych bitów między polami). Specjalny rozmiar 0 służy do wymuszenia przesunięcia kolejnych pól w granice następnego słowa.

W pewnych maszynach bity pola umieszcza się w słowie od lewej strony do prawej, w innych zaś od prawej do lewej. Pola są więc całkiem użyteczne przy obsłudze danych zdefiniowanych lokalnie. Dla danych pochodzących z zewnątrz należy natomiast starannie zbadać, który koniec pola pojawia się jako pierwszy; programy zależące od takich rzeczy nie są przenośne. Pola można deklarować jedynie z typem int; ze względu na przenośność oprogramowania należy je jawnie kwalifikować jako signed lub unsigned. Pola nie są tablicami i nie mają adresów, zatem nie można stosować do nich operatora adresu &.

7 WEJŚCIE I WYJŚCIE

Mechanizmy wejścia i wyjścia nie są częścią samego języka C, więc aż do tej pory nie omawialiśmy ich zbyt dokładnie. Niemniej jednak sposoby komunikowania się programów ze swoim otoczeniem są znacznie bardziej skomplikowane od dotychczas prezentowanych. W tym rozdziale przedstawimy bibliotekę standardową, czyli zestaw funkcji realizujących operacje wejścia-wyjścia, obsługę tekstów, zarządzanie pamięcią, operacje numeryczne i wiele, wiele innych usług przydatnych w programach w języku C. Skoncentrujemy się jednak na obsłudze wejścia i wyjścia.

Funkcje biblioteczne są w standardzie ANSI na tyle szczegółowo zdefiniowane, że mogą występować w zgodnej formie we wszystkich systemach, w których istnieje C. Dzięki temu programy, w których komunikacja z systemem ogranicza się do udogodnień dostarczanych przez bibliotekę standardową, mogą być przenoszone z jednego systemu do innego bez zmian.

Właściwości funkcji bibliotecznych opisano w więcej niż tuzinie nagłówków; widzieliśmy już kilka z nich, w tym <stdio.h>, <string.h> i <ctype.h>. Nie pokażemy tutaj całej biblioteki, gdyż bardziej interesuje nas pisanie takich programów w języku C, które z niej korzystają. Biblioteka jest szczegółowo opisana w dodatku B.

7.1 Standardowe wejście i wyjście

Jak powiedzieliśmy w rozdz. 1, w bibliotece standardowej zaimplementowano prosty model znakowego wejścia i wyjścia. W tym modelu strumień znaków składa się z ciągu wierszy; każdy wiersz jest zakończony znakiem nowego wiersza. Jeśli system operacyjny pracuje w inny sposób, to do biblioteki należy zrobienie wszystkiego, co jest konieczne, by z punktu widzenia programu system pracował zgodnie z tym modelem. Na przykład funkcje biblioteczne mogą zamieniać wejściową parę znaków powrotu karetki i zmiany wiersza na znak nowego wiersza oraz wyjściowy znak nowego wiersza z powrotem na taką parę znaków.

Najprostszym mechanizmem wejścia jest czytanie po jednym znaku ze *standardowego wejścia*, zwykle klawiatury, za pomocą funkcji getchar:

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

Funkcja ta przy każdym wywołaniu podaje następny znak z wejścia lub EOF, gdy napotkała koniec pliku. Stała symboliczna EOF jest zdefiniowana w nagłówku <stdio.h>. Jej wartością jest na ogół –1, ale w testach należy używać EOF, aby uniezależnić się od takiej specyficznej wartości.

W wielu środowiskach klawiaturę można zastąpić plikiem, stosując konwencję przełączania źródła danych za pomocą symbolu <. Jeśli program prog używa funkcji getchar, to wywołujące ten program polecenie

```
prog <infile
```

spowoduje, że prog będzie czytał znaki z pliku infile, a nie z klawiatury. Przełączenie źródła danych następuje tak, że program nie odczuwa zmiany; w szczególności tekst „<infile" nie jest dołączany do listy argv argumentów wywołania programu. Przełączenie źródła danych jest również niewidoczne wówczas, gdy dane wejściowe napływają z innego programu przez potok (ang. *pipe*); w pewnych systemach polecenie

```
otherprog | prog
```

uruchamia dwa programy otherprog i prog, łącząc potokiem standardowe wyjście otherprog ze standardowym wejściem prog.

Funkcja

```
int putchar(int)
```

służy do wypisywania danych: wywołanie putchar(c) wysyła znak c do *standardowego wyjścia*, którym domyślnie jest ekran. Funkcja ta zwraca wartość wypisanego znaku lub EOF (jako sygnał wystąpienia błędu). Wyniki można także kierować do pliku używając symbolu >: jeśli program prog korzysta z funkcji putchar, to polecenie

```
prog >outfile
```

spowoduje, że prog zamiast do standardowego wyjścia będzie pisać do pliku outfile. Jeżeli system udostępnia potoki, to w poleceniu

```
prog | anotherprog
```

standardowe wyjście prog zostanie połączone potokiem ze standardowym wejściem programu anotherprog.

Dane produkowane przez funkcję printf również są kierowane do standardowego wyjścia. Wywołania putchar i printf mogą się przeplatać – wyniki pojawią się na wyjściu zgodnie z kolejnością tych wywołań.

W każdym pliku źródłowym programu, który korzysta z funkcji bibliotecznych realizujących operacje wejścia-wyjścia, przed pierwszym odwołaniem do biblioteki musi wystąpić wiersz

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

Gdy nazwa nagłówka jest ujęta w nawiasy < i >, wówczas tego nagłówka szuka się w miejscach standardowo wyróżnionych (np. w systemach Unix zwykle jest to skorowidz /usr/include).

Wiele programów czyta zaledwie jeden strumień danych wejściowych i produkuje jeden strumień danych na wyjściu. Dla takich programów mechanizm wejścia-wyjścia z użyciem funkcji getchar, putchar i printf w zupełności wystarczy. A już na pewno wystarczy początkującym programistom, zwłaszcza gdy używają mechanizmu przełączania oraz mechanizmu potoków do łączenia wyjścia jednego programu z wejściem następnego. Jako przykład rozważmy program lower, który przekształca wielkie litery tekstu wejściowego na małe:

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

main()   /* lower: zamień wielkie litery na małe */
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

Funkcja tolower jest zdefiniowana w nagłówku <ctype.h>; przekształca ona wielkie litery na małe nie zmieniając innych znaków. Jak wspomnieliśmy wcześniej, takie „funkcje", jak getchar i putchar z nagłówka <stdio.h> oraz tolower z nagłówka <ctype.h>, często są makrami, co pozwala uniknąć narzutów, jakie wiążą się z wywołaniem funkcji dla każdego znaku. Jak to zostało zrobione, pokażemy w p. 8.5. Niezależnie od tego, jak funkcje z nagłówka <ctype.h> zostały zrealizowane w kon-

kretnej maszynie, programom z nich korzystającym oszczędzono wiedzy o dostępnym zbiorze znaków.

Ćwiczenie 7.1. Napisz program, który przekształca wielkie litery na małe lub małe litery na wielkie w zależności od tego, z jaką nazwą został wywołany; nazwa ta figuruje w argv[0].

7.2 Formatowane wyjście – funkcja printf

Wyjściowa funkcja printf tłumaczy wewnętrzne wartości na znaki. W poprzednich rozdziałach stosowaliśmy ją nieformalnie. Opis, który tu przedstawiamy, wyczerpuje najbardziej typowe zastosowania, ale nie jest kompletny – taki opis znajdziesz w dodatku B.

int printf(char *format, *arg1, arg2* ...)

Funkcja printf pod nadzorem argumentu format przekształca, formatuje i wypisuje swoje argumenty do standardowego wyjścia. Jej wartością jest liczba wypisanych znaków.

Format zawiera obiekty dwojakiego rodzaju: zwykłe znaki, które są kopiowane do strumienia wyjściowego, oraz specyfikacje przekształceń, z których każda wskazuje sposób przekształcenia i wypisania kolejnego argumentu funkcji printf. Każdą specyfikację przekształcenia rozpoczyna znak %, a kończy znak charakterystyczny dla tego przekształcenia. Między znakiem % i znakiem przekształcenia mogą – w następującej kolejności – wystąpić:

- Minus, zlecający dosunięcie przekształconego argumentu do lewego krańca jego pola.
- Liczba określająca minimalny rozmiar pola. Przekształcony argument będzie wpisany do pola o co najmniej takim rozmiarze. Jeśli trzeba, pole zostanie uzupełnione do pełnego rozmiaru z lewej strony (lub z prawej, jeśli żądano dosunięcia w lewo).
- Kropka, oddzielająca rozmiar pola od precyzji.
- Liczba określająca precyzję, tj. maksymalną liczbę znaków dla tekstu, liczbę cyfr po kropce dziesiętnej dla wartości zmiennopozycyjnej lub minimalną liczbę cyfr dla wartości całkowitej.
- Jedna z liter: h – jeśli argument całkowity należy wypisać jako short, lub l (litera el) – jeśli jako long.

Znaki przekształcenia figurują w tabl. 7.1. Działanie funkcji nie jest określone, jeżeli znak następujący po % nie jest znakiem przekształcenia.

Tablica 7.1. Podstawowe przekształcenia funkcji **printf**

Znak	Typ argumentu	Dana wyjściowa
d, i	int	liczba dziesiętna
o	int	liczba ósemkowa bez znaku (bez wiodącego zera)
x, X	int	liczba szesnastkowa bez znaku (bez wiodących 0x lub 0X) z literami abcdef lub ABCDEF dla 10, ..., 15
u	int	liczba dziesiętna bez znaku
c	int	jeden znak
s	char *	ciąg znaków wypisywany do napotkania '\0' lub wyczerpania liczby znaków określonej przez precyzję
f	double	[–]*m.dddddd*, gdzie liczbę cyfr *d* określa precyzja (domyślnie 6)
e, E	double	[–]*m.ddddd*e±*xx* lub [–]*m.ddddd*E±*xx*, gdzie liczbę cyfr *d* określa precyzja (domyślnie 6)
g, G	double	wypisana w formacie %e lub %E, jeśli wykładnik jest mniejszy niż –4 albo większy lub równy precyzji; w przeciwnym przypadku wypisana w formacie %f; nie wypisuje się nieznaczących zer i kończącej kropki dziesiętnej
p	void *	wskaźnik (postać zależna od implementacji)
%		nie ma przekształcenia argumentu; wypisany znak %

Szerokość pola lub precyzję można w specyfikacji zastąpić znakiem *, co oznacza, że żądaną liczbę należy obliczyć przekształcając kolejny argument funkcji (argument musi być typu int). Na przykład, wypisanie co najwyżej max znaków z s wygląda tak:

```
printf("%.*s", max, s);
```

Większość przekształceń formatujących prezentowaliśmy już w poprzednich rozdziałach. Jednym z wyjątków jest precyzja odnosząca się do tekstów. Następujące zestawienie pokazuje działanie różnych specyfikacji podczas wypisywania tekstu „ahoj, przygodo" (14 znaków). Aby można było ocenić rozmiary pól, otoczyliśmy je dwukropkami.

```
:%s:          :ahoj, przygodo:
:%10s:        :ahoj, przygodo:
:%.10s:       :ahoj, przy:
:%-10s:       :ahoj, przygodo:
:%.20s:       :ahoj, przygodo:
:%-20s:       :ahoj, przygodo      :
:%20.10s:     :          ahoj, przy:
:%-20.10s:    :ahoj, przy          :
```

Uwaga: funkcja printf używa swojego pierwszego argumentu do określenia liczby i typów pozostałych argumentów. Jeżeli nie podałeś wystarczającej liczby argumentów lub są one złego typu, to funkcja będzie zdezorientowana, a Ty otrzymasz błędne wyniki. Powinieneś więc zdawać sobie sprawę z różnicy między tymi wywołaniami:

```
printf(s);          /* ŹLE, jeśli w s występuje % */
printf("%s", s);   /* BEZPIECZNIE */
```

Funkcja sprintf dokonuje tych samych przekształceń, co printf, ale wynik zapisuje w tablicy znakowej:

```
int sprintf(char *string, char *format, arg1, arg2, ...)
```

Funkcja sprintf formatuje wartości argumentów *arg1*, *arg2* itd. na podstawie specyfikacji przekształceń podanych w argumencie format (według powyższego opisu). Jednak zamiast kierować wynik do wyjścia, umieszcza go w miejscu wskazanym argumentem string. Pamięć wskazywana przez string musi być wystarczająco obszerna, aby pomieścić cały wynikowy tekst.

Ćwiczenie 7.2. Napisz program, który w jakiejś sensownej formie wypisze dowolny strumień znaków wejściowych. Program powinien przynajmniej wypisywać znaki niegraficzne w postaci ósemkowej lub szesnastkowej (zależnie od miejscowych zwyczajów), a także dzielić zbyt długie wiersze.

7.3 Zmienna długość list argumentów

Prezentujemy tutaj realizację minimalnej wersji funkcji printf, aby pokazać, jak napisać przenośnie funkcję posługującą się listą argumentów o zmiennej liczbie elementów. Głównie interesuje nas przetwarzanie argumentów, toteż nasza funkcja minprintf będzie sama opracowywała format i resztę argumentów, ale do wykonania przekształceń formatujących wywoła prawdziwą printf.

A oto właściwa deklaracja funkcji printf:

```
int printf(char *fmt, ...)
```

w której deklaracja ... (trzy kropki) oznacza, że liczba i typy pozostałych argumentów nie są znane. Taka deklaracja może wystąpić jedynie na końcu listy argumentów. Zatem naszą funkcję minprintf deklarujemy następująco:

```
void minprintf(char *fmt, ...)
```

nie będziemy bowiem zwracać licznika znaków, tak jak to robi printf.

Sztuczka polega na tym, że minprintf maszeruje wzdłuż listy argumentów, choć ta nie ma nawet nazwy. Standardowy nagłówek <stdarg.h> zawiera zestaw makr, które definiują sposób poruszania się po takiej liście. Realizacje tego nagłówka będą różnić

się między sobą zależnie od maszyny, ale posługiwanie się nim jest jednakowe w każdym środowisku C.

Zmienną odnoszącą się po kolei do każdego argumentu deklaruje się z typem va_list. W funkcji minprintf taka zmienna nazywa się ap, czyli wskaźnik do argumentów (od ang. *argument pointer*). Standardowe makro va_start inicjuje zmienną ap tak, aby wskazywała na pierwszy nienazwany argument. Ogólnie na liście argumentów musi wystąpić co najmniej jeden argument z nazwą; makro va_start, aby rozpocząć działanie, potrzebuje ostatniego nazwanego argumentu.

Każde wywołanie makra va_arg udostępnia jeden argument i przesuwa ap do następnego; do określenia typu szukanej wartości i rozmiaru kroku, o jaki trzeba przesunąć ap, makro va_arg potrzebuje nazwy typu. Ostatnie makro va_end czyści wszystko, co wymaga czyszczenia; va_end musi być wywołane przed zakończeniem działania funkcji.

Te właściwości tworzą podstawę działania naszej uproszczonej wersji printf:

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

/* minprintf: minimalna printf ze zmienną listą argumentów */
void minprintf(char *fmt, ...)
{
    va_list ap;   /* wskazuje po kolei każdy nienazwany argument */
    char *p, *sval;
    int ival;
    double dval;

    va_start(ap, fmt); /* ap wskazuje 1. nienazwany argument */
    for (p = fmt; *p; p++) {
        if (*p != '%') {
            putchar(*p);
            continue;
        }
        switch (*++p) {
        case 'd':
            ival = va_arg(ap, int);
            printf("%d", ival);
            break;
        case 'f':
            dval = va_arg(ap, double);
            printf("%f", dval);
            break;
```

```
            case 's':
                for (sval = va_arg(ap, char *); *sval; sval++)
                    putchar(*sval);
                break;
            default:
                putchar(*p);
                break;
            }
        }
        va_end(ap); /* po pracy wyczyść co trzeba */
    }
```

Ćwiczenie 7.3. Uzupełnij minprintf tak, aby obsługiwała więcej możliwości funkcji printf.

7.4 Formatowane wejście – funkcja **scanf**

Funkcja scanf jest wejściowym odpowiednikiem funkcji printf – umożliwia większość tych samych przekształceń, lecz w przeciwnym kierunku.

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

Funkcja scanf wczytuje znaki ze standardowego wejścia, interpretuje je zgodnie ze specyfikacjami zawartymi w argumencie format i zapamiętuje wyniki w miejscach określonych przez pozostałe argumenty. Argument formatujący będzie opisany dalej; pozostałe argumenty, *z których każdy musi być wskaźnikiem,* wskazują, gdzie należy przekazać odpowiednio przekształcone dane wejściowe. Tak jak w przypadku printf, niniejszy opis jest tylko podsumowaniem najczęściej używanych możliwości funkcji, a nie ich wyczerpująco przedstawioną listą.

Funkcja scanf zatrzyma się wtedy, kiedy zinterpretuje wszystkie znaki formatu lub gdy pewna dana nie pasuje do żądanej specyfikacji przekształcenia. Jej wartością jest liczba szczęśliwie wczytanych i przypisanych danych wejściowych. Z wartości tej można skorzystać przy ustalaniu liczby znalezionych danych. Po napotkaniu końca pliku funkcja zwraca EOF; podkreślamy to, gdyż ta wartość jest różna od 0 oznaczającego, że najbliższy znak wejściowy nie pasuje do pierwszej specyfikacji przekształcenia. Kolejne wywołanie scanf wznawia szukanie bezpośrednio za ostatnim już wczytanym znakiem.

Mamy także funkcję sscanf, która zamiast ze standardowego wejścia czyta znaki ze wskazanej tablicy:

```
int sscanf(char *string, char *format, arg1, arg2, ...)
```

Funkcja ta interpretuje tekst zawarty w tablicy string zgodnie z formatem zadanym w argumencie format i wynikowe wartości umieszcza w miejscach wskazanych przez pozostałe argumenty *arg1*, *arg2* itp. Te argumenty muszą być wskaźnikami.

Format zawiera zwykle specyfikacje przekształceń używane do sterowania przekształcaniem danych wejściowych. W formacie mogą wystąpić:

- Odstępy oraz znaki tabulacji – są ignorowane.
- Zwykłe czarne znaki (ale nie %), które spodziewamy się zastać w strumieniu wejściowym.
- Specyfikacje przekształceń złożone ze znaku %, opcjonalnego znaku * wstrzymującego przypisanie, opcjonalnej liczby określającej maksymalny rozmiar pola, jednego z opcjonalnych znaków h, l lub L ustalających rozmiar wyniku oraz ze znaku przekształcenia.

Specyfikacja przekszałcenia steruje przekształceniem następnego pola wejściowego. Normalnie wynik jest wstawiany do zmiennej wskazanej odpowiednim argumentem. Jeśli jednak przypisanie ma być wstrzymane (znak *), to dane pole wejściowe pomija się – nie będzie żadnego przypisania. Polem wejściowym jest z definicji ciąg czarnych znaków; rozciąga się ono albo do najbliższej „białej plamy" (ciągu białych znaków), albo – jeśli podano rozmiar pola – na odległość wskazaną tym rozmiarem. Wynika z tego, że funkcja scanf w poszukiwaniu pól będzie przekraczać granice wierszy, ponieważ znak nowego wiersza jest białym znakiem. (Białymi znakami są: odstęp, znak

Tablica 7.2. Podstawowe przekształcenia funkcji **scanf**

Znak	Dana wejściowa	Typ argumentu
d	liczba całkowita dziesiętna	int *
i	liczba całkowita; może wystąpić w postaci ósemkowej (z wiodącym 0) lub szesnastkowej (z wiodącymi 0x lub 0X)	int *
o	liczba całkowita w postaci ósemkowej (razem z wiodącym 0 lub bez)	int *
u	liczba całkowita dziesiętna bez znaku	unsigned int *
x	liczba całkowita w postaci szesnastkowej (z wiodącymi 0x lub 0X, albo bez)	int *
c	znaki; następne znaki z wejścia (domyślnie 1) umieszcza się we wskazanej tablicy; nie obowiązuje zwykła zasada pomijania białych plam; aby przeczytać najbliższy czarny znak, należy użyć %1s	char *
s	tekst (ale nie napis, tj. ciąg znaków występujący bez znaków cudzysłowu); argument powinien wskazywać na tablicę znakową o rozmiarze wystarczającym do przyjęcia tekstu wraz z dodanym na końcu znakiem '\0'	char *
e, f, g	liczba zmiennopozycyjna z opcjonalnym znakiem, opcjonalną kropką dziesiętną i opcjonalnym wykładnikiem	float *
%	literalnie znak %; nie będzie żadnego przypisania	

tabulacji, znak nowego wiersza, znak powrotu karetki, znak tabulacji pionowej i znak nowej strony.)

Znak przekształcenia określa sposób interpretacji pola wejściowego. Odpowiadający temu polu argument musi być wskaźnikiem, jak tego wymaga obowiązujący w języku C mechanizm przekazywania argumentów funkcji przez wartość. Znaki przekształcenia pokazano w tabl. 7.2.

Znaki przekształceń d, i, o, u i x można poprzedzić literą h informującą, że odpowiedni argument nie jest wskaźnikiem do obiektu typu int, tylko typu short, oraz literą l, która mówi, że na liście argumentów powinien wystąpić wskaźnik do obiektu typu long. Podobnie znaki przekształcenia e, f i g można poprzedzić literą l dla argumentów wskaźnikowych do obiektów typu double, a nie float.

Pierwszym z przykładów niech będzie program prymitywnego kalkulatora z rozdz. 4; możemy go napisać inaczej, korzystając z funkcji scanf do przekształcania danych wejściowych:

```
#include <stdio.h>

main() /* prymitywny kalkulator */
{
    double sum, v;

    sum = 0;
    while (scanf("%lf", &v) == 1)
        printf("\t%.2f\n", sum += v);
    return 0;
}
```

Przypuśćmy, że chcemy wprowadzać wiersze zawierające daty w postaci

```
25 Grudnia 1988
```

W tym celu można użyć funkcji scanf:

```
int day, year;          /* dzień, rok */
char monthname[20];     /* nazwa miesiąca */

scanf("%d %s %d", &day, monthname, &year);
```

Argument monthname występuje bez operatora adresu &, ponieważ jako nazwa tablicy jest wskaźnikiem.

W formacie mogą występować zwykłe znaki; wówczas muszą literalnie pasować do takich samych znaków z wejścia. Dzięki temu za pomocą wywołania scanf możemy wczytywać daty w postaci mm/dd/yy:

```
int day, month, year;

scanf("%d/%d/%d", &month, &day, &year);
```

W formacie funkcja scanf ignoruje odstępy i znaki tabulacji. Co więcej, poszukując danych w strumieniu znaków wejściowych, pomija białe plamy (odstępy, znaki tabulacji, znaki nowego wiersza itp.). Często lepiej jest – dla danych o nie ustalonym formacie – najpierw wczytać cały kolejny wiersz, a następnie „wydłubywać" z niego poszczególne kawałki używając funkcji sscanf. Dla przykładu przypuśćmy, że chcemy czytać wiersze, w których data może wystąpić w jednej z obu wyżej podanych postaci. W takim razie możemy napisać

```
while (getline(line, sizeof(line)) > 0 {
    if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
        printf("poprawna: %s\n", line); /*format: 25 Grudnia 1988*/
    else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
        printf("poprawna: %s\n", line); /*format: mm/dd/yy */
    else
        printf("niepoprawna: %s\n", line); /*błędny format*/
}
```

Wywołania scanf mogą się przeplatać z wywołaniami innych funkcji wejściowych. Przy kolejnym wywołaniu dowolnej funkcji wejściowej czytanie rozpocznie się od pierwszego znaku, którego scanf jeszcze nie przeczytała.

I końcowe ostrzeżenie: argumenty funkcji scanf i sscanf *muszą* być wskaźnikami. Zbyt często pojawiającym się błędem jest wywołanie

```
scanf("%d", n);
```

zamiast poprawnego

```
scanf("%d", &n);
```

Taki błąd zazwyczaj nie jest wykrywany przez kompilator.

Ćwiczenie 7.4. Napisz prywatną wersję funkcji scanf, analogiczną do minprintf z poprzedniego punktu.

Ćwiczenie 7.5. Napisz na nowo program kalkulatora przyrostkowego z rozdz. 4, stosując funkcję scanf i być może funkcję sscanf do wczytywania i przekształcania liczb.

7.5 Obsługa plików

W napisanych dotychczas programach dane czytaliśmy wyłącznie ze standardowego wejścia, a wyniki wysyłaliśmy do standardowego wyjścia, przy czym wejście i wyjście były automatycznie udostępniane każdemu programowi przez lokalny system operacyjny.

Następnym krokiem będzie napisanie programu obsługującego plik, który *nie* został wcześniej dołączony do programu. Jednym z przykładów ilustrujących potrzebę takiej obsługi jest program cat, który do standardowego wyjścia wysyła dane pochodzące z kilku wskazanych, sklejonych ze sobą plików. Program służy do wypisywania zawartości plików na ekranie, a także jako kolektor danych (ogólnego zastosowania) dla programów, które nie potrafią obsługiwać plików wskazanych przez nazwy. Na przykład polecenie

```
cat x.c y.c
```

wypisuje do standardowego wyjścia zawartość plików x.c i y.c (i nic więcej).

Nasuwa się pytanie, jak zorganizować czytanie nazwanych plików – to znaczy jak powiązać zewnętrzne nazwy, o które chodzi użytkownikowi, z instrukcjami służącymi do czytania danych.

Sposób jest prosty. Przed czytaniem z pliku lub pisaniem do pliku należy go *otworzyć* za pomocą bibliotecznej funkcji fopen. Funkcja ta bierze zewnętrzną nazwę (jak x.c lub y.c), robi z nią jakieś sztuczki, następnie przeprowadza jakieś negocjacje z systemem operacyjnym (szczegóły nas nie interesują), a w końcu udostępnia wskaźnik, którego używa się w programie przy późniejszych operacjach czytania z pliku i pisania do pliku.

Ten wskaźnik, nazywany *wskaźnikiem pliku*, pokazuje na pewną strukturę zawierającą następujące informacje o pliku: położenie bufora, bieżąca pozycja znaku w buforze, rodzaj dostępu do pliku (czytanie, pisanie itp.), sygnały o wystąpieniu błędów lub o napotkaniu końca pliku itd. Użytkownik nie musi znać tych szczegółów, znajdują się one bowiem w strukturze o nazwie FILE, zadeklarowanej w nagłówku <stdio.h>. Niezbędne deklaracje dla wskaźnika pliku demonstruje przykład

```
FILE *fp;
FILE *fopen(char *name, char *mode);
```

Mówią one, że fp jest wskaźnikiem do struktury typu FILE i że funkcja fopen zwraca taki wskaźnik. Zauważ, że FILE jest nazwą typu, jak int, a nie etykietką struktury

– jest zatem zdefiniowana za pomocą typedef. (Szczegóły o tym, jak funkcję fopen można zrealizować w systemie Unix, znajdziesz w p. 8.5.)

W programie wywołanie funkcji fopen ma postać

```
fp = fopen(name, mode);
```

Pierwszym argumentem fopen jest nazwa pliku. Drugi argument informuje o *rodzaju dostępu* do pliku, tzn. jak zamierza się korzystać z tego pliku. Wśród dozwolonych rodzajów są: czytanie ("r"), pisanie ("w") i dopisywanie ("a"). W pewnych systemach rozróżnia się pliki tekstowe i binarne; dla tych ostatnich do rodzaju dostępu należy dołączyć "b".

Jeżeli otworzysz do pisania lub dopisywania plik, który nie istnieje, to zostanie on utworzony (o ile będzie to możliwe). Otwarcie istniejącego pliku do pisania powoduje zamazanie jego poprzedniej zawartości, podczas gdy otwarcie do dopisywania chroni ją. Próba czytania z pliku, który nie istnieje, jest błędem. Oczywiście jest wiele innych błędnych sytuacji, choćby próba czytania z pliku bez odpowiednich uprawnień. W przypadku błędu funkcja fopen zwraca NULL. (Błąd ten można zidentyfikować bardziej precyzyjnie – przeczytaj omówienie funkcji obsługi błędów na końcu p. 1 w dodatku B.)

Teraz, gdy plik jest już otwarty, musimy poznać sposoby czytania i pisania. Wyróżnia się ich kilka, z których najprostszym jest użycie funkcji getc i putc. Funkcja getc zwraca kolejny znak wczytany z pliku; w tym celu wymaga wskaźnika mówiącego o jaki plik chodzi.

```
int getc(FILE *fp)
```

Zatem getc zwraca kolejny znak ze strumienia znaków wskazanego przez fp, a jako sygnał końca pliku lub wystąpienia błędu zwraca EOF.

Funkcja putc jest funkcją wyjściową:

```
int putc(int c, FILE *fp)
```

Funkcja putc zapisuje znak c do pliku wskazanego przez fp i zwraca wartość tego znaku lub EOF jako sygnał wystąpienia błędu. Podobnie jak getchar i putchar, procedury getc i putc mogą nie być funkcjami, lecz makrami.

Przy uruchamianiu programu napisanego w języku C środowisko systemu operacyjnego jest odpowiedzialne za otwarcie trzech plików i udostępnienie programowi ich wskaźników. Plikami tymi są: standardowe wejście, standardowe wyjście oraz standardowe wyjście błędów. Wskaźniki tych plików nazywają się odpowiednio: stdin, stdout i stderr; są one zadeklarowane w nagłówku <stdio.h>. Normalnie stdin jest związany z klawiaturą, a stdout i stderr z ekranem, ale wskaźniki stdin oraz stdout można przyłączyć do innych plików lub potoków, jak to opisano w p. 7.1.

Funkcje getchar i putchar można zdefiniować jako makra za pomocą nazw getc, putc, stdin oraz stdout:

```
#define getchar()    getc(stdin)
#define putchar(c)   putc((c), stdout)
```

Przy formatowanym czytaniu z plików i pisaniu do plików możemy korzystać z funkcji fscanf i fprintf. Działają one identycznie jak scanf i printf z jednym wyjątkiem: ich pierwszym argumentem jest wskaźnik do obsługiwanego pliku; format jest drugim argumentem.

```
int fscanf(FILE *fp, char *format, ...)
int fprintf(FILE *fp, char *format, ...)
```

Z tym bagażem wiedzy możemy już napisać program cat służący do sklejania zawartości plików. Przyjmiemy jedno założenie, które zresztą przydaje się też w innych programach: jeżeli w wierszu polecenia wywołującego program podano argumenty, to będą one interpretowane jako nazwy plików i obsługiwane w kolejności wystąpienia. Jeżeli argumentów nie podano, to obsługuje się standardowe wejście.

```
#include <stdio.h>

/* cat: sklej zawartość plików, wersja 1 */
main(int argc, char *argv[ ])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);

    if (argc == 1) /* bez argumentów; kopiuj std. wejście */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                printf("cat: nie mogę otworzyć %s\n", *argv);
                return 1;
            } else {
                filecopy(fp, stdout);
                fclose(fp);
            }
    return 0;
}
```

```
/* filecopy: kopiuj zawartość pliku ifp do pliku ofp */
void filecopy(FILE *ifp, FILE *ofp)
{
    int c;

    while ((c = getc(ifp)) != EOF)
        putc(c, ofp);
}
```

Wskaźniki plików stdin i stdout są obiektami typu FILE *. Są jednak stałymi, a *nie* zmiennymi, przypisywanie im czegokolwiek jest więc niemożliwe.

Funkcja

```
int fclose(FILE *fp)
```

działa odwrotnie niż fopen – zrywa połączenie ustalone przez fopen między wskaźnikiem pliku a nazwą zewnętrzną, zwalniając wskaźnik dla innego pliku. Większość systemów operacyjnych nakłada ograniczenia na liczbę jednocześnie otwartych plików w jednym programie, dobrym pomysłem jest więc zwalnianie wskaźników plików, jak w programie cat, gdy nie są już dłużej potrzebne. Istnieje także drugi powód, aby zamykać pliki wyjściowe – funkcja fclose opróżnia bufor, w którym putc gromadziła dane wyjściowe. Funkcja fclose jest wywoływana automatycznie dla wszytkich jeszcze otwartych plików, gdy program kończy się normalnie. (Możesz zamknąć standardowe pliki stdin i stdout, jeśli nie są Ci potrzebne. Można je także przełączyć za pomocą bibliotecznej funkcji freopen.)

7.6 Obsługa błędów – plik **stderr** i funkcja **exit**

Obsługa błędów w programie cat nie jest idealna. Kłopot polega na tym, że jeżeli z jakiegoś powodu jeden z plików nie jest dostępny, to odpowiedni komunikat zostanie dopisany na koniec sklejanego tekstu wyjściowego. Można to ewentualnie zaakceptować wówczas, gdy wyniki są kierowane na ekran, nie można natomiast wtedy, kiedy przesyła się je do innego pliku lub potokiem do innego programu.

Abyśmy mogli sobie lepiej radzić z takimi sytuacjami, mamy drugi strumień danych wyjściowych, o nazwie stderr, przydzielony programowi w ten sam sposób, jak strumienie stdin i stdout. Dane kierowane do pliku stderr zwykle pojawiają się na ekranie nawet wtedy, kiedy standardowe wyjście jest skierowane gdzie indziej.

Poprawmy więc program cat tak, by swoje komunikaty o błędach wypisywał do standardowego pliku błędów.

```
#include <stdio.h>

/* cat: sklej zawartość plików, wersja 2 */
main(int argc, char *argv[ ])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);
    char *prog = argv[0]; /* nazwa programu do komunikatów */

    if (argc == 1) /* bez argumentów: kopiuj std. wejście */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                fprintf(stderr, "%s: nie mogę otworzyć %s\n", prog, *argv);
                exit(1);
            } else {
                filecopy(fp, stdout);
                fclose(fp);
            }
    if (ferror(stdout)) {
        fprintf(stderr, "%s: błąd pisania do stdout\n", prog);
        exit(2);
    }
    exit(0);
}
```

Program ten sygnalizuje błędy na dwa sposoby. Po pierwsze, komunikaty produkowane przez funkcję fprintf są kierowane do pliku stderr. Znajdą zatem drogę, aby pojawić się na ekranie, zamiast ugrzęznąć gdzieś wśród danych wyjściowych lub w potoku. W komunikatach uwzględniliśmy nazwę, z jaką ten program został wywołany (argv[0]), znane więc będzie źródło błędu, gdy pracuje on wśród innych programów.

Po drugie, w programie skorzystaliśmy ze standardowej funkcji exit, której wywołanie powoduje zatrzymanie programu. Argument funkcji exit będzie dostępny dla każdego procesu wywołującego ten program jako swój podproces. Badając tę wartość możemy dowiedzieć się o sukcesie lub porażce naszego programu. Przyjęto, że wartość zero świadczy o poprawnym wykonaniu programu, natomiast jakakolwiek niezerowa wartość sygnalizuje sytuację awaryjną. Funkcja exit wywołuje funkcję fclose dla wszystkich otwartych plików wyjściowych w celu wypisania danych pozostałych w buforach.

Wewnątrz funkcji main instrukcja return *wyr* jest równoważna wywołaniu exit(*wyr*). Funkcja exit ma tę przewagę, że można ją wywołać w dowolnej innej funkcji. Takie

wywołania często można spotkać w programach wyszukiwania według wzorca, jak nasz program z rozdz. 5.

Funkcja ferror zwraca wartość niezerową wówczas, gdy dla strumienia danych fp wystąpił jakiś błąd.

```
int ferror(FILE *fp)
```

Wprawdzie błędy dotyczące wyjścia są sporadyczne, ale jednak się zdarzają (np. przepełnienie dysku), więc program produkujący dane powinien również sprawdzać ich wystąpienie.

Funkcja feof(FILE*) działa podobnie do funkcji ferror: zwraca wartość niezerową po napotkaniu końca pliku wskazanego argumentem fp.

```
int feof(FILE *fp)
```

W naszych małych programach ilustracyjnych zazwyczaj nie martwiliśmy się o stan zakończonego programu, ale w poważnym programie koniecznie trzeba dbać o zwracanie sensownych, użytecznych wartości opisujących ten stan.

7.7 Wprowadzanie i wyprowadzanie wierszy tekstu

Biblioteka standardowa zawiera wejściową funkcję fgets, bardzo podobną do używanej przez nas w poprzednich rozdziałach funkcji getline:

```
char *fgets(char *line, int maxline, FILE *fp)
```

Funkcja fgets czyta kolejny wiersz (łącznie ze znakiem nowego wiersza) z pliku wskazanego przez wskaźnik fp i wstawia ten wiersz do tablicy znakowej line. Funkcja fgets czyta co najwyżej maxline–1 znaków. Wynikowy wiersz będzie zakończony znakiem '\0'. Normalnie funkcja ta zwraca wartość wskaźnika line; po napotkaniu końca pliku lub po wykryciu błędu jej wartością funkcyjną jest NULL. (Nasza funkcja getline zwraca długość wczytanego wiersza, co jest informacją bardziej użyteczną; zero oznacza koniec pliku.)

Wyjściowa funkcja fputs wypisuje do wskazanego pliku tekst (który nie musi zawierać znaku nowego wiersza):

```
int fputs(char *line, FILE *fp)
```

Zwracaną przez funkcję wartością jest zero, a w przypadku błędu – EOF.

Działanie funkcji bibliotecznych gets i puts jest podobne do fgets i fputs, ale operują one na strumieniach standardowych stdin oraz stdout. Mętlik wprowadza tylko to, że gets usuwa kończący znak nowego wiersza '\n', a puts dopisuje go do tworzonego wiersza.

Aby wykazać, że nie ma nic magicznego w takich funkcjach, jak fgets i fputs, przedstawiamy je tutaj skopiowane bezpośrednio z dostępnej nam biblioteki standardowej:

```
/* fgets: weź co najwyżej n znaków z pliku iop */
char *fgets(char *s, int n, FILE *iop)
{
    register int c;
    register char *cs;

    cs = s;
    while (--n > 0 && (c = getc(iop)) != EOF)
        if ((*cs++ = c) == '\n')
            break;
    *cs = '\0';
    return (c == EOF && cs == s) ? NULL : s;
}

/* fputs: wypisz s do pliku iop */
int fputs(char *s, FILE *iop)
{
    int c;

    while (c = *s++)
        putc(c, iop);
    return ferror(iop) ? EOF : 0;
}
```

W standardzie określono, że po wystąpieniu błędu funkcja ferror ma zwracać wartość niezerową; fputs zwraca w takim przypadku EOF, a w pozostałych przypadkach wartość nieujemną.

Teraz, za pomocą funkcji fgets, łatwo możemy zrealizować naszą funkcję getline:

```
/* getline: przeczytaj wiersz, podaj jego długość */
int getline(char *line, int max)
{
    if (fgets(line, max, stdin) == NULL)
        return 0;
    else
        return strlen(line);
}
```

Ćwiczenie 7.6. Napisz program porównujący dwa pliki i wypisujący pierwszy wiersz, w którym pliki się różnią.

Ćwiczenie 7.7. Zmień program wyszukujący według wzorca (z rozdz. 5) tak, aby przyjmował dane wejściowe z zestawu nazwanych plików lub ze standardowego wejścia, jeśli w argumentach wywołania nie podano żadnej nazwy pliku. Czy razem ze znalezionym wierszem trzeba wypisywać nazwę pliku, w którym go znaleziono?

Ćwiczenie 7.8. Utwórz program wypisujący zawartość zestawu plików, z których każdy rozpoczyna się na nowej stronie. Wszystkie strony powinny być opatrzone tytułem i bieżącym numerem strony w ramach każdego pliku z osobna.

7.8 Kilka użytecznych funkcji

W bibliotece standardowej występuje wiele różnorodnych funkcji. W tym punkcie przedstawiamy tylko krótkie opisy tych najczęściej stosowanych. Więcej szczegółów i wiele innych funkcji możesz znaleźć w dodatku B.

7.8.1 Operacje na tekstach

Wspominaliśmy już o funkcjach operujących na tekstach – spośród nich omówiliśmy funkcje strlen, strcpy, strcat i strcmp. Ich deklaracje znajdują się w nagłówku <string.h>. Oto zestawienie takich funkcji; w następujących opisach s oraz t są wskaźnikami do znaków (char *), a c oraz n są typu int.

strcat(s,t)	dopisuje t na koniec s
strncat(s,t,n)	dopisuje n znaków z t na koniec s
strcmp(s,t)	zwraca wartość ujemną, zero lub wartość dodatnią odpowiednio dla s < t, s == t lub s > t
strncmp(s,t,n)	robi to samo, co strcmp, ale tylko dla początkowych n znaków t
strcpy(s,t)	kopiuje t do s
strncpy(s,t,n)	kopiuje co najwyżej n znaków t do s
strlen(s)	zwraca długość s
strchr(s,c)	zwraca wskaźnik do pierwszego wystąpienia c w s lub NULL – gdy c nie występuje w s
strrchr(s,c)	zwraca wskaźnik do ostatniego wystąpienia c w s lub NULL – gdy c nie występuje w s

7.8.2 Badanie klasy znaków i ich przekształcenia

Wiele funkcji ze standardowego nagłówka <ctype.h> bada lub przekształca znaki. W następujących opisach c jest argumentem całkowitym, reprezentującym wartości typu int lub unsigned char (także EOF). Wszystkie funkcje zwracają wartość typu int; prawda oznacza wartość różną od zera.

isalpha(c)	prawda, jeśli c jest literą lub cyfrą; 0 – jeśli nie
isupper(c)	prawda, jeśli c jest wielką literą; 0 – jeśli nie
islower(c)	prawda, jeśli c jest małą literą; 0 – jeśli nie
isdigit(c)	prawda, jeśli c jest cyfrą; 0 – jeśli nie
isalnum(c)	prawda, jeśli isalpha(c) lub isdigit(c) są prawdziwe; 0 – jeśli nie
isspace(c)	prawda, jeśli c jest odstępem lub jednym ze znaków: tabulacji, nowego wiersza, powrotu karetki, nowej strony lub tabulacji pionowej
toupper(c)	zwraca c przekształcone na wielką literę
tolower(c)	zwraca c przekształcone na małą literę

7.8.3 Funkcja ungetc

W bibliotece standardowej występuje dość okrojona wersja funkcji ungetch, napisanej w rozdz. 4. Nazywa się tu ungetc i jest zadeklarowana następująco:

```
int ungetc(int c, FILE *fp)
```

Funkcja ungetc oddaje znak c z powrotem do pliku wskazanego przez fp i zwraca wartość znaku c lub EOF, gdy wystąpił błąd. Dla danego pliku można wycofać tylko jeden znak. Funkcję ungetc można stosować wraz z dowolną standardową funkcją wejściową, jak scanf, getc lub getchar.

7.8.4 Wykonanie polecenia

Funkcja system(char *s) wykonuje polecenie zawarte w argumencie s, po czym wznawia wykonywanie bieżącego programu. Zawartość s ściśle zależy od lokalnego systemu operacyjnego. Banalnym przykładem pochodzącym z systemu Unix jest instrukcja

```
system("date");
```

która uruchamia program date wypisujący do standardowego wyjścia aktualną datę i czas. Funkcja system zwraca zależną od systemu wartość całkowitą opisującą stan,

z jakim zakończyło się uruchomione polecenie. W systemie Unix wartością stanu jest wartość zwracana przez funkcję exit.

7.8.5 Zarządzanie pamięcią

Standardowe funkcje malloc i calloc dynamicznie pobierają od systemu żądane bloki pamięci.

```
void *malloc(size_t n)
```

Funkcja malloc zwraca wskaźnik do n bajtów nie zainicjowanej pamięci albo NULL, jeśli żądanie nie może być spełnione.

```
void *calloc(size_t n, size_t size)
```

Funkcja calloc zwraca wskaźnik do obszaru mogącego pomieścić tablicę n elementów podanego rozmiaru size. Funkcja zwraca NULL, jeżeli żądania nie można spełnić. Przydzielona pamięć jest inicjowana zerami.

Zwracany przez obie funkcje wskaźnik pokazuje na obszar pamięci położony zgodnie z wymaganiami danego obiektu, ale wartość tego wskaźnika należy zrzutować na właściwy typ, jak w następującym przykładzie:

```
int *ip;

ip = (int *) calloc(n, sizeof(int));
```

Funkcja free(p) zwalnia pamięć wskazywaną przez p, przy czym wartość p musi być wynikiem wcześniejszego wywołania funkcji malloc lub calloc. Nie ma ograniczeń dotyczących kolejności zwalniania pamięci. Okropnym za to błędem jest zwalnianie czegoś, co nie było uprzednio przydzielone za pomocą funkcji malloc lub calloc.

Błędem jest także używanie czegoś, co już zostało zwolnione. Typowym – i niepoprawnym – fragmentem programu jest następująca pętla, która zwalnia bloki pamięci powiązane w łańcuch:

```
for (p = head; p != NULL; p = p->next) /* ŹLE */
    free(p);
```

Poprawną metodą jest przechowanie wszystkiego, co jeszcze będzie potrzebne, przed zwolnieniem obszaru pamięci:

```
for (p = head; p != NULL; p = q) {
    q = p->next;
    free(p);
}
```

W punkcie 8.7 pokażemy taką realizację dystrybutora pamięci podobnego do malloc, w którym przydzielane bloki mogą być zwalniane w dowolnej kolejności.

7.8.6 Funkcje matematyczne

W standardowym nagłówku <math.h> zadeklarowano ponad dwadzieścia funkcji matematycznych. Tutaj wymieniamy tylko te najczęściej używane. Każda z nich oczekuje jednego lub dwóch argumentów typu double i zwraca wartość typu double.

sin(x)	sinus x; wartość x w radianach
cos(x)	cosinus x; wartość x w radianach
atan2(y,x)	arcus tangens y/x; wartość w radianach
exp(x)	funkcja wykładnicza e^x
log(x)	logarytm naturalny x (przy podstawie e); $x > 0$
log10(x)	logarytm x (przy podstawie 10); $x > 0$
pow(x,y)	funkcja potęgowa x^y
sqrt(x)	pierwiastek kwadratowy z x; $x \geq 0$
fabs(x)	wartość bezwzględna x

7.8.7 Generowanie liczb losowych

Funkcja rand( ) oblicza ciąg pseudolosowych liczb całkowitych z przedziału od zera do RAND_MAX; ta górna wartość jest zdefiniowana w nagłówku <stdlib.h>. Jednym ze sposobów obliczenia losowych liczb zmiennopozycyjnych większych lub równych zero, lecz mniejszych niż jeden, jest zdefiniowanie makra

```
#define frand( ) ((double) rand( ) / (RAND_MAX+1.0))
```

(Jeśli w Twojej bibliotece występuje funkcja obliczająca losowe liczby zmiennopozycyjne, to należy się spodziewać, że statystyczne właściwości tej funkcji są lepsze niż właściwości tak zdefiniowanego makra.)

Funkcja srand(unsigned) określa zarodek (ang. *seed*) dla funkcji rand. Zalecaną przez standard przenośną implementację funkcji rand i srand opisano w p. 2.7.

Ćwiczenie 7.9. Funkcje podobne do isupper można zrealizować tak, aby oszczędzały pamięć albo oszczędzały czas. Zbadaj obie możliwości.

8 ŚRODOWISKO SYSTEMU UNIX

System operacyjny Unix oferuje swoje usługi poprzez zestaw *odwołań systemowych*. W rzeczywistości są to funkcje rezydujące wewnątrz systemu operacyjnego, które mogą być wywoływane w programach użytkowników. Tutaj wyjaśniamy, jak w programach w języku C skorzystać z kilku najważniejszych odwołań systemowych. Jeśli pracujesz z systemem Unix, te informacje mogą Ci się bezpośrednio przydać. Czasem trzeba bowiem odwołać się do systemu, by osiągnąć maksymalną sprawność programu lub zastosować taką usługę systemu, dla której nie ma funkcji w bibliotece standardowej. A jeżeli nawet korzystasz z języka C w innym systemie operacyjnym, to po przestudiowaniu przykładów będziesz mieć lepsze wyobrażenie o programowaniu w języku C; szczegóły mogą ulec zmianie, ale podobny kod można znaleźć w każdym systemie. Biblioteka standardowa ANSI C w wielu przypadkach jest wzorowana na możliwościach systemu Unix, zatem nasze przykłady mogą Ci się przydać także do lepszego zrozumienia samej biblioteki.

Materiał tego rozdziału podzielono na trzy zasadnicze części: wejście-wyjście, system plików oraz zarządzanie pamięcią. W dwóch pierwszych zakłada się umiarkowaną znajomość zewnętrznych właściwości systemu Unix.

W rozdziale 7 zajmowaliśmy się obsługą wejścia i wyjścia, ujednoliconą dla różnych systemów operacyjnych. W każdym konkretnym systemie podprogramy z biblioteki standardowej muszą korzystać z mechanizmów obowiązujących w tym właśnie systemie. W kilku następnych punktach opiszemy te odwołania systemowe obowiązujące w systemie Unix, które realizują wejście i wyjście, a następnie pokażemy, jak posługując się nimi można zaprogramować niektóre elementy biblioteki standardowej.

8.1 Deskryptory plików

W systemie Unix wszelkie operacje wejścia i wyjścia wyraża się za pomocą czytania z plików lub pisania do plików. Wszystkie bowiem urządzenia zewnętrzne – nawet klawiatura i ekran – są plikami wchodzącymi w skład systemu plików. Oznacza to, że

całą komunikację programu z urządzeniami zewnętrznymi obsługuje wspólny, jednorodny aparat.

W najbardziej ogólnym przypadku, zanim zaczniemy czytać z pliku lub pisać do pliku, musimy poinformować system o naszym zamiarze. Proces ten nazywa się *otwieraniem* pliku. Jeśli mamy zamiar pisać do pliku, to może się okazać, że plik ten trzeba najpierw utworzyć lub skasować jego dotychczasową zawartość. System sprawdzi, czy mamy do tego prawo. (Czy plik istnieje? Czy mamy pozwolenie na korzystanie z niego?) Jeżeli wszystko jest w porządku, to system przekazuje do programu pewną niewielką, nieujemną liczbę całkowitą zwaną *deskryptorem pliku*. We wszystkich operacjach wejścia-wyjścia zamiast nazwy identyfikującej plik używa się właśnie tego deskryptora. (Deskryptor pliku jest obiektem analogicznym do wskaźnika pliku używanego przez bibliotekę standardową lub do opisu pliku (ang. *the file handle*) w systemie MS-DOS.) Wszystkie informacje o otwartym pliku są przetwarzane przez system; programy użytkowe odwołują się do plików wyłącznie za pomocą deskryptorów.

Ze względu na to, że znaczna część przesłań danych odbywa się między programem a klawiaturą i ekranem, stworzono specjalne mechanizmy ułatwiające tę komunikację. Interpretator poleceń (shell) uruchamiając program otwiera trzy pliki z deskryptorami 0, 1 i 2, nazwane odpowiednio standardowym wejściem, standardowym wyjściem i standardowym wyjściem błędów. Czytając z pliku o deskryptorze 0 lub pisząc do pliku o deskryptorze 1 lub 2, program może wprowadzać dane i wypisywać wyniki bez otwierania plików.

Użytkownik programu może przełączyć standardowe wejście lub wyjście na inne pliki, stosując w poleceniu notację;

```
prog <infile >outfile
```

W takim przypadku interpretator poleceń zmieni domyślne dowiązania deskryptorów 0 i 1 oraz połączy je z nazwanymi plikami. Zwykle deskryptor 2 pozostaje dowiązany do ekranu, aby można było tam wysyłać komunikaty o błędach. Podobnie dzieje się, gdy standardowe wejście lub wyjście jest związane z potokiem. We wszystkich tych przypadkach dowiązania do plików są zmieniane przez interpretator poleceń, a nie przez program. Dopóki program używa pliku 0 dla wejścia oraz plików 1 i 2 dla wyjścia, dopóty nie wie, skąd dane są pobierane i dokąd wysyłane.

8.2 Wejście i wyjście niskiego poziomu – funkcje **read** i **write**

Operacje wejścia i wyjścia używają odwołań systemowych read i write; są one dostępne w programach C dzięki dwóm funkcjom read i write. Pierwszym argumentem obu funkcji jest deskryptor pliku. Drugim argumentem jest tablica znakowa w progra-

mie użytkownika, która przechowuje dane przychodzące do programu lub z niego wysyłane. Trzeci argument określa liczbę bajtów do przesłania.

```
int n_read = read(int fd, char *buf, int n);
int n_written = write(int fd, char *buf, int n);
```

Każde wywołanie zwraca liczbę rzeczywiście przesłanych bajtów. Przy czytaniu wartość ta może być mniejsza niż liczba żądanych bajtów. Zero oznacza koniec pliku, a −1 informuje o wystąpieniu jakiegoś błędu. Przy pisaniu zwracaną wartością jest liczba wypisanych bajtów; jeśli nie jest ona równa liczbie bajtów przeznaczonych do wypisania, to znaczy że wystąpił błąd.

W jednym wywołaniu można zażądać przeczytania lub wypisania dowolnej liczby bajtów. Najczęściej pojawiającymi się wartościami są: 1, co oznacza przesłanie jednego znaku („niebuforowane"), oraz liczby w rodzaju 1024 lub 4096, które odpowiadają rozmiarom fizycznych bloków danych dla urządzeń zewnętrznych. Im liczba jednocześnie przesyłanych znaków jest większa, tym operacje przesyłania są bardziej efektywne, gdyż wymagają mniejszej liczby odwołań do systemu.

Podsumowując to wszystko, co zostało powiedziane, możemy napisać prosty program kopiujący dane z wejścia na wyjście – odpowiednik programu napisanego w rozdz. 1. Ten program będzie kopiował cokolwiek na cokolwiek, można bowiem zarówno wejście, jak i wyjście, przełączyć na dowolny plik lub urządzenie.

```
#include "syscalls.h"

main()   /* kopiuj wejście na wyjście */
{
    char buf[BUFSIZ];
    int n;

    while ((n = read(0, buf, BUFSIZ)) > 0)
        write(1, buf, n);
    return 0;
}
```

Prototypy funkcji niezbędnych do obsługi odwołań systemowych umieściliśmy w pliku o nazwie syscalls.h. Możemy więc go włączać do programów w tym rozdziale. Nie jest on jednak nagłówkiem standardowym.

Parametr BUFSIZ także został zdefiniowany w syscalls.h; jego wartość reprezentuje rozmiar bufora najbardziej odpowiedni w lokalnym systemie. Jeżeli rozmiar pliku nie jest wielokrotnością BUFSIZ, to pewne wywołanie funkcji read zwróci mniejszą niż

BUFSIZ liczbę bajtów, które mają być wypisane przez write; następne po tym wywołanie read zwróci wartość zero.

Warto zobaczyć, jak za pomocą funkcji read i write można zbudować funkcje wyższego poziomu, jak getchar, putchar itp. Przykładowo prezentujemy wersję funkcji getchar, która czyta z wejścia bez buforowania danych, to znaczy po jednym znaku na raz.

```
#include "syscalls.h"

/* getchar: wejście jednoznakowe niebuforowane */
int getchar(void)
{
    char c;

    return (read(0, &c, 1) == 1) ? (unsigned char) c : EOF;
}
```

Zmienna c musi być typu char, ponieważ funkcja read akceptuje jedynie wskaźniki do znaków. Rzutowanie c do typu unsigned char w instrukcji return eliminuje jakiekolwiek problemy związane z powielaniem bitu znaku.

W drugiej wersji funkcja getchar wczytuje dane wielkimi porcjami, a oddaje każdorazowo po jednym znaku.

```
#include "syscalls.h"

/* getchar: prosta wersja buforująca */
int getchar(void)
{
    static char buf[BUFSIZ];
    static char *bufp = buf;
    static int n = 0;

    if (n == 0) {  /* bufor jest pusty */
        n = read(0, buf, sizeof buf);
        bufp = buf;
    }
    return (--n >= 0) ? (unsigned char) *bufp++ : EOF;
}
```

Jeśli tę wersję funkcji getchar chciałbyś przetłumaczyć z dołączonym nagłówkiem <stdio.h>, to gdy standardową wersję getchar zrealizowano jako makro, wówczas jego definicję należy odwołać za pomocą #undef.

8.3 Funkcje open, creat, close, unlink

Pliki inne niż standardowe wejście, standardowe wyjście i standardowe wyjście błędów muszą być jawnie otwarte, zanim zaczniesz z nich czytać lub do nich pisać. Do tego celu służą dwa odwołania systemowe open i creat (sic! – od ang. *create* – utwórz).

Funkcja open przypomina omówioną w rozdz. 7 funkcję fopen, zamiast wskaźnika do pliku zwraca jednak deskryptor pliku (deskryptor jest typu int). W przypadku błędu open zwraca –1.

```
#include <fcntl.h>

int fd;
int open(char *name, int flags, int perms);

fd = open(name, flags, perms);
```

Podobnie jak w funkcji fopen, argument name jest ciągiem znaków zawierającym nazwę pliku. Drugi argument, flags, jest liczbą typu int, określającą sposób otwarcia wskazanego pliku. Jego najważniejszymi wartościami są:

O_RDONLY	otwarcie tylko do pisania
O_WRONLY	otwarcie tylko do czytania
O_RDWR	otwarcie zarówno do czytania, jak i pisania

Te stałe zdefiniowano w nagłówku <fcntl.h> dla rodziny systemów Unix System V oraz w nagłówku <sys/file.h> dla systemów Unix w wersji Berkeley (BSD).

Aby otworzyć już istniejący plik do czytania, należy na przykład napisać

```
fd = open(name, O_RDONLY, 0);
```

W omawianych przez nas przykładach otwarć plików trzeci argument, perms, zawsze będzie równy zero.

Próba otwarcia pliku, który nie istnieje, jest błędem. Do tworzenia nowego pliku lub do zapisania na nowo już istniejącego służy odwołanie systemowe creat.

```
int creat(char *name, int perms);

fd = creat(name, perms);
```

Funkcja creat zwraca deskryptor pliku, jeśli mogła utworzyć wskazany plik, a w przeciwnym przypadku zwraca –1. Jeżeli natomiast plik już istnieje, to creat skraca go do zerowej długości, kasując tym samym jego poprzednią zawartość; tworzenie pliku, który już istnieje, nie jest błędem.

Jeśli plik jest całkiem nowy, to creat utworzy go z uprawnieniami określonymi argumentem perms (od ang. *permission*). W systemie Unix z każdym plikiem w systemie plików jest związane dziewięciobitowe pole przeznaczone na informacje o uprawnieniach. Wartość tego pola określa uprawnienia do czytania, pisania i wykonywania pliku przez właściciela pliku, jego zespół oraz przez resztę użytkowników systemu. Zakres uprawnień najlepiej zatem oddaje trzycyfrowa liczba ósemkowa. Na przykład 0755 nadaje właścicielowi prawo do czytania, pisania i wykonywania, a członkom jego zespołu i wszystkim pozostałym użytkownikom tylko prawo do czytania i wykonywania.

Dla ilustracji prezentujemy uproszczoną wersję jednego z programów narzędziowych systemu Unix – cp, który kopiuje zawartość jednego pliku do innego. Nasza wersja kopiuje tylko jeden plik, nie zezwala, aby drugim argumentem wywołania programu był skorowidz, oraz wymyśla uprawnienia, zamiast je kopiować.

```
#include <stdio.h>
#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666   /* czytanie i pisanie dla wszystkich */

void error(char *, ...);

/* cp: kopiuj f1 do f2 */
main(int argc, char *argv[ ])
{
    int f1, f2, n;
    char buf[BUFSIZ];

    if (argc != 3)
        error("Format wywołania: cp skąd dokąd");
    if ((f1 = open (argv[1], O_RDONLY, 0)) == -1)
        error("cp: nie mogę otworzyć %s", argv[1]);
    if ((f2 = creat(argv[2], PERMS)) == -1)
        error("cp: nie mogę utworzyć %s z uprawnieniami %03o",
              argv[2], PERMS);
    while ((n = read(f1, buf, BUFSIZ)) > 0)
        if (write(f2, buf, n) != n)
            error("cp: błąd pisania do pliku %s", argv[2]);
    return 0;
}
```

Ten program tworzy plik wyjściowy ze stałymi uprawnieniami 0666. Korzystając z odwołania systemowego stat, opisanego w p. 8.6, możemy ustalić uprawnienia istniejącego pliku, a zatem utworzyć jego kopię z tak określonymi uprawnieniami.

Zwróć uwagę na funkcję error, którą – podobnie jak printf – wywołujemy ze zmienną listą argumentów. Realizacja funkcji error ilustruje zastosowanie jeszcze jednego członka rodziny printf: funkcja vprintf z biblioteki standardowej działa jak printf z tym, że zamiast zmiennej listy argumentów obsługuje pojedynczy argument. Ten argument został utworzony w fazie inicjowania listy argumentów funkcji error za pomocą makra va_start. Innymi funkcjami standardowymi z tej rodziny są: vfprintf – odpowiednik funkcji fprintf oraz vsprintf – odpowiednik funkcji sprintf.

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

/* error: wypisz komunikat i zakończ program */
void error(char *fmt, ...)
{
    va_list args;

    va_start(args, fmt);
    fprintf(stderr, "error: ");
    vfprintf(stderr, fmt, args);
    fprintf(stderr, "\n");
    va_end(args);
    exit(1);
}
```

Systemy operacyjne na ogół ograniczają liczbę jednocześnie otwartych plików w jednym programie (często do 20). Wobec tego każdy program, w którym używa się wielu plików, musi być przystosowany do wielokrotnego korzystania z tych samych deskryptorów plików. Funkcja close(int fd) zamyka plik, tj. przerywa połączenie między deskryptorem pliku a otwartym plikiem, zwalniając tym samym deskryptor do ponownego użycia z jakimś innym plikiem. Funkcja ta jest odpowiednikiem funkcji fclose z biblioteki standardowej, ale nie ma buforów do opróżniania. Zakończenie wykonywania programu przez funkcję exit lub za pomocą instrukcji return w funkcji main powoduje zamknięcie wszystkich otwartych plików.

Funkcja unlink(char *name) usuwa plik name z systemu plików. Funkcja ta jest odpowiednikiem funkcji remove z biblioteki standardowej.

Ćwiczenie 8.1. Zmień program cat z rozdz. 7 używając funkcji read, write, open i close zamiast ich odpowiedników z biblioteki standardowej. Przeprwadź eksperymenty porównujące względne szybkości obu wersji.

8.4 Dostęp swobodny – funkcja lseek

Operacje wejścia i wyjścia są zwykle sekwencyjne: każde wywołanie funkcji read lub write odnosi się do miejsca w pliku tuż za pozycją osiągniętą przy poprzedniej operacji. Niekiedy zachodzi jednak potrzeba czytania z pliku lub pisania do pliku w dowolnej kolejności. Poruszanie się po pliku bez czytania czy pisania danych umożliwia odwołanie systemowe lseek:

```
long lseek(int fd, long offset, int origin);
```

Funkcja lseek zmienia bieżącą pozycję pliku o deskryptorze fd na pozycję wskazaną przez argument offset (odstęp), którą oblicza się względem punktu odniesienia określonego przez argument origin. Kolejne czytanie lub pisanie rozpocznie się od tej właśnie pozycji. Wartością origin może być 0, 1 lub 2 oznaczające odpowiednio, że odstęp ma być mierzony względem początku, bieżącej pozycji lub końca pliku. Aby na przykład dopisać dane do pliku, należy – przed pisaniem – poszukać jego końca:

```
lseek(fd, 0L, 2);
```

(Tak reaguje interpretator poleceń w systemie Unix na przełączenie przepływu danych >> oraz tak realizuje się tryb dostępu do pliku "a" w funkcji fopen.)

Wywołanie

```
lseek(fd, 0L, 0);
```

powoduje powrót do początku pliku („przewinięcie'' pliku – od ang. *rewind*).

Zwróć uwagę na argument 0L; można to także zapisać (long)0 albo po prostu 0, jeśli funkcję lseek zadeklarowano poprawnie.

Mając do dyspozycji funkcję lseek, możemy traktować pliki mniej więcej tak, jak wielkie tablice (kosztem czasu dostępu). Na przykład, następująca funkcja czyta dowolną liczbę bajtów z dowolnego miejsca pliku. Funkcja ta zwraca liczbę przeczytanych bajtów lub –1 po wykryciu błędu.

```
#include "syscalls.h"

/* get: przeczytaj n bajtów od pozycji pos */
int get(int fd, long pos, char *buf, int n)
{
    if (lseek(fd, pos, 0) >= 0) /* ustaw pozycję */
        return read(fd, buf, n);
    else
        return -1;
}
```

Wartością zwracaną przez lseek jest liczba typu long dla nowej pozycji w pliku lub −1 w przypadku wystąpienia błędu. Funkcja fseek z biblioteki standardowej jest podobna do lseek z tym, że jej pierwszym argumentem jest wskaźnik pliku (FILE *) oraz że w przypadku błędu zwracana wartość jest różna od zera.

8.5 Przykład – realizacja funkcji fopen i getc

Aby zilustrować współpracę niektórych z opisanych odwołań systemowych, pokażemy realizację dwóch funkcji z biblioteki standardowej: fopen i getc.

Przypominamy, że w bibliotece standardowej pliki są identyfikowane za pomocą wskaźników plików, a nie za pomocą deskryptorów. Wskaźnik pliku jest wskaźnikiem do struktury zawierającej różne informacje o pliku: wskaźnik do bufora, dzięki któremu można czytać z pliku dużymi porcjami, licznik znaków pozostałych w buforze, wskaźnik do pozycji następnego znaku w buforze, deskryptor pliku oraz kilka znaczników, które opisują rodzaj dostępu do pliku (czytanie/pisanie), stan programu (wystąpienie błędów) itp.

Struktura danych opisujących plik jest zdefiniowana w standardowym nagłówku <stdio.h>. Nagłówek ten musi być włączany (za pomocą #include) do każdego pliku źródłowego, w którym używa się funkcji z biblioteki standardowej realizujących operacje wejścia-wyjścia. Jest on również włączony w funkcjach z tej biblioteki. W podanym dalej wyciągu z typowego pliku <stdio.h> te nazwy, które z założenia są przeznaczone do użytku tylko dla funkcji bibliotecznych, rozpoczynają się znakiem podkreślenia (przez co zmniejsza się prawdopodobieństwo kolizji z nazwami w programie użytkownika). Tę konwencję stosują wszystkie podprogramy z biblioteki standardowej.

```
#define NULL        0
#define EOF         (-1)
#define BUFSIZ      1024
#define OPEN_MAX    20 /* maks. liczba jednocześnie otwartych plików */

typedef struct _iobuf {
    int cnt;            /* liczba znaków w buforze */
    char *ptr;          /* pozycja następnego znaku */
    char *base;         /* położenie bufora */
    int flag;           /* sposób dostępu do pliku */
    int fd;             /* deskryptor pliku */
} FILE;
extern FILE _iob[OPEN_MAX];
```

```
#define stdin     (&_iob[0])
#define stdout    (&_iob[1])
#define stderr    (&_iob[2])

enum _flags {
    _READ  = 01,  /* plik otwarty do czytania */
    _WRITE = 02,  /* plik otwarty do pisania */
    _UNBUF = 04,  /* plik niebuforowany */
    _EOF   = 010, /* napotkano koniec pliku */
    _ERR   = 020  /* wystąpił błąd przy obsłudze pliku */
};

int _fillbuf(FILE *);          /* wypełnij bufor */
int _flushbuf(int, FILE *); /* wypisz bufor */

#define feof(p)    (((p)->flag & _EOF) != 0)
#define ferror(p)  (((p)->flag & _ERR) != 0)
#define fileno(p)  ((p)->fd)   /* deskryptor pliku */

#define getc(p)    (--(p)->cnt >= 0 \
                   ? (unsigned char) *(p)->ptr++ : _fillbuf(p))
#define putc(x,p)  (--(p)->cnt >= 0 \
                   ? *(p)->ptr++ = (x) : _flushbuf((x),p))

#define getchar()  getc(stdin)
#define putchar(x) putc((x), stdout)
```

Makro getc zazwyczaj zmniejsza licznik znaków, przesuwa wskaźnik i daje znak. (Przypominamy, że zbyt długi wiersz #define jest kontynuowany za pomocą znaku \.) Jeżeli jednak licznik znaków stanie się ujemny, to getc wywoła funkcję _fillbuf, która ponownie wypełni bufor, zainicjuje zawartość struktury i zwróci znak. Znaki otrzymywane za pomocą makra getc są zrzutowane do unsigned char, co gwarantuje, że każdy znak będzie dodatni.

Definicję makra putc zamieściliśmy po to, aby pokazać (bez zagłębiania się w szczegóły), że działa prawie w ten sam sposób co getc, wywołując funkcję _flushbuf, kiedy bufor jest pełny. Dołączyliśmy także makra udostępniające deskryptor pliku oraz takie informacje o stanie programu, jak wartość znacznika końca pliku czy znacznika błędów.

Teraz możemy już napisać funkcję fopen. Jej działanie polega przede wszystkim na dostarczeniu otwartego pliku, określeniu prawidłowej pozycji w tym pliku i ustawieniu bitowych znaczników opisujących jego właściwy stan. Funkcja fopen nie przy-

dziela pamięci na bufor; robi to funkcja _fillbuf wtedy, kiedy plik jest czytany po raz pierwszy.

```
#include <stdio.h>
#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666 /* czytanie i pisanie dla wszystkich */

/* fopen: otwórz plik, daj wskaźnik pliku */
FILE *fopen(char *name, char *mode)
{
    int fd;
    FILE *fp;

    if (*mode != 'r' && *mode != 'w' && *mode != 'a')
        return NULL;
    for (fp = _iob; fp < _iob + OPEN_MAX; fp++)
        if ((fp->flag & (_READ | _WRITE)) == 0)
            break;          /* wolne miejsce w tablicy plików */
    if (fp >= _iob + OPEN_MAX)
        return NULL;    /* nie ma wolnego miejsca */

    if (*mode == 'w')
        fd = creat(name, PERMS);
    else if (*mode == 'a') {
        if ((fd = open(name, O_WRONLY, 0)) == -1)
            fd = creat(name, PERMS);
        lseek(fd, 0L, 2);
    } else
        fd = open(name, O_RDONLY, 0);
    if (fd == -1)           /* nie ma dostępu do pliku */
        return NULL;
    fp->fd = fd;
    fp->cnt = 0;
    fp->base = NULL;
    fp->flag = (*mode == 'r') ? _READ : _WRITE;
    return fp;
}
```

Ta wersja funkcji fopen nie obsługuje wszystkich możliwych sposobów dostępu do pliku, jakie opisano w standardzie, chociaż dodanie ich nie wymaga specjalnie dużego

kodu. W szczególności nasza funkcja nie rozpoznaje ani rodzaju "b" (dostęp binarny), ponieważ w systemach Unix nie ma zastosowania, ani rodzaju "+", który dopuszcza jednocześnie czytanie i pisanie.

Przy pierwszym wywołaniu funkcji getc licznik znaków danego pliku równa się zero, co wymusza wywołanie funkcji _fillbuf. Jeśli funkcja _fillbuf stwierdzi, że plik nie został otwarty do czytania, to natychmiast wraca z sygnałem końca pliku EOF. W przeciwnym przypadku próbuje przydzielić bufor (jeżeli czytanie ma być buforowane).

Gdy bufor jest już ustalony, wówczas funkcja _fillbuf wywołuje funkcję read, aby zapełniła ten bufor, po czym nadaje wartości licznikowi oraz wskaźnikom i wraca z pierwszym znakiem znajdującym się w buforze. Następne wywołania funkcji _fillbuf korzystają z przydzielonego już bufora.

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

/* _fillbuf: przydziel i wypełnij bufor wejściowy */
int _fillbuf(FILE *fp)
{
    int bufsize;

    if ((fp->flag & (_READ | _EOF | _ERR)) != _READ)
        return EOF;
    bufsize = (fp->flag & _UNBUF) ? 1 : BUFSIZ;
    if (fp->base == NULL)       /* jeszcze nie ma bufora */
        if ((fp->base = (char *) malloc(bufsize)) == NULL)
            return EOF;         /* brak miejsca na bufor */
    fp->ptr = fp->base;
    fp->cnt = read(fp->fd, fp->ptr, bufsize);
    if (--fp->cnt < 0) {
        if (fp->cnt == -1)
            fp->flag |= _EOF;
        else
            fp->flag |= _ERR;
        fp->cnt = 0;
        return EOF;
    }
    return (unsigned char) *fp->ptr++;
}
```

Jedyne, co jeszcze pozostało, to wyjaśnienie, jak wszystko zaczyna razem współpracować. Tablica _iob musi być uprzednio zdefiniowana i zainicjowana dla plików stdin, stdout i stderr:

```
FILE _iob[OPEN_MAX] ={   /* stdin, stdout, stderr: */
    { 0, (char *) 0, (char *) 0, _READ, 0 },
    { 0, (char *) 0, (char *) 0, _WRITE, 1 },
    { 0, (char *) 0, (char *) 0, _WRITE | _UNBUF, 2 }
};
```

Ze sposobu zainicjowania składowych flag w tablicy struktur wynika, że plik stdin jest przeznaczony do czytania, plik stdout do pisania, a plik stderr do pisania niebuforowanego.

Ćwiczenie 8.2. Napisz nowe wersje funkcji fopen i _fillbuf, używając pól zamiast jawnych operacji bitowych. Porównaj rozmiar kodu i prędkość działania obu wersji.

Ćwiczenie 8.3. Zaprojektuj i napisz funkcje _flushbuf, fflush i fclose.

Ćwiczenie 8.4. Biblioteka standardowa zawiera funkcję

```
int fseek(FILE *fp, long offset, int origin)
```

identyczną z funkcją lseek z wyjątkiem tego, że fp jest wskaźnikiem pliku, a nie deskryptorem pliku, oraz że zwracaną przez funkcję wartością jest stan pliku wyrażony liczbą całkowitą (int), a nie pozycja w pliku. Napisz tę funkcję tak, aby prawidłowo obsługiwała dane buforowane dla innych funkcji bibliotecznych.

8.6 Przykład – wypisywanie zawartości skorowidzów

Czasem są potrzebne narzędzia współpracujące z systemem plików w inny sposób – przekazujące informacje *o pliku*, a nie informacje w nim zawarte. Polecenie systemu Unix o nazwie ls jest przykładem takiego programu narzędziowego: wypisuje nazwy plików zawartych w skorowidzu oraz – na żądanie – inne informacje o tych plikach, jak ich rozmiary, uprawnienia itp. Polecenie dir w systemie MS-DOS działa podobnie.

Ponieważ skorowidz w systemie Unix jest po prostu plikiem, to w programie ls wystarczy przeczytać ten plik, aby wydobyć z niego nazwy plików. Ale inne informacje o pliku, jak jego rozmiar, z konieczności uzyskuje się za pomocą odwołania systemowego. W innych systemach bywa tak, że nawet nazwy plików skorowidza uzyskuje się jedynie przez odwołanie systemowe; tak jest np. w systemie MS-DOS. Zmierzamy do tego, by dostęp do żądanej informacji był względnie niezależny od systemu, choćby nawet jego realizacja była ściśle zależna od systemu.

Zilustrujemy to na przykładzie programu fsize, który jest specjalną wersją programu ls: wypisuje rozmiary wszystkich plików wymienionych z nazwy w wierszu polecenia. Jeśli jednym z nich jest skorowidz, to podprogram fsize rekurencyjnie wywołuje sam siebie dla tego skorowidza. Jeżeli w ogóle nie podano argumentów, to program przetwarza bieżący skorowidz.

Na początek podamy krótki zarys struktury plików systemu Unix. *Skorowidz* jest plikiem zawierającym listę nazw plików wraz z pewnymi wskazówkami dotyczącymi ich położenia. „Położenie'' jest indeksem pewnej tablicy zwanej „tablicą węzłów informacyjnych'' (ang. *inode table*). W *węźle* są przechowywane wszystkie informacje o pliku z wyjątkiem jego nazwy. Pozycja w skorowidzu zwykle składa się tylko z dwóch obiektów: numeru węzła i nazwy pliku.

Pożałowania godny jest fakt, że format i szczegółowa zawartość skorowidza nie są takie same we wszystkich wersjach systemu. Zatem, aby wyodrębnić fragmenty zależne od systemu, musimy podzielić zadanie na dwie części. Na najwyższym poziomie zdefiniujemy typ strukturowy o nazwie Dirent (pozycja skorowidza) oraz trzy podprogramy opendir, readdir i closedir, które mają zapewnić niezależny od systemu dostęp do nazwy pliku i numeru węzła w pozycji skorowidza. Skorzystamy z tego mechanizmu w programie fsize, a potem pokażemy, jak go zrealizować w systemach, w których struktura skorowidza jest taka sama, jak w systemach Unix Version 7 czy Unix System V; warianty pozostawiamy jako ćwiczenia.

Struktura Dirent zawiera numer węzła i nazwę. Maksymalna długość składowej nazwy jest określona przez stałą symboliczną NAME_MAX, której wartość zależy od systemu. Funkcja opendir zwraca wskaźnik do struktury o typie nazwanym DIR przez analogię do typu FILE; z tego wskaźnika korzystają funkcje readdir i closedir. Wszystkie te informacje zebraliśmy w pliku nagłówkowym o nazwie dirent.h:

```
#define NAME_MAX 14   /* najdłuższa nazwa pliku; */
                      /* zależy od systemu */

typedef struct {          /* przenośny opis pozycji skorowidza: */
    long ino;                     /* numer węzła */
    char name[NAME_MAX+1]; /* nazwa + kończący znak '\0' */
} Dirent;

typedef struct {          /* min. opis DIR: bez buforowania itp. */
    int fd;                       /* deskryptor pliku skorowidza */
    Dirent d;                     /* pozycja skorowidza */
} DIR;

DIR *opendir(char *dirname);
Dirent *readdir(DIR *dfd);
void closedir(DIR *dfd);
```

Odwołanie systemowe stat dla danej nazwy pliku podaje wszystkie informacje zawarte w węźle tego pliku; stat zwraca –1, jeśli wystąpił błąd.

```
char *name;
struct stat stbuf;
int stat(char *, struct stat *);

stat(name, &stbuf);
```

Funkcja stat wypełnia strukturę stbuf informacjami z węzła pliku wskazanego przez argument name. Struktura opisująca wartości zwracane przez funkcję stat znajduje się w nagłówku <sys/stat.h> i zwykle wygląda tak:

```
struct stat     /* informacje z węzła zwracane przez stat */
{
    dev_t   st_dev;     /* urządzenie związane z węzłem */
    ino_t   st_ino;     /* numer węzła */
    short   st_mode;    /* bity atrybutów pliku */
    short   st_nlink;   /* liczba dowiązań do pliku */
    short   st_uid;     /* identyfikator właściciela */
    short   st_gid;     /* identyfikator zespołu */
    dev_t   st_rdev;    /* dla plików specjalnych */
    off_t   st_size;    /* rozmiar pliku w znakach */
    time_t  st_atime;   /* data ostatniego dostępu */
    time_t  st_mtime;   /* data ostatniej modyfikacji */
    time_t  st_ctime;   /* data ostatniej zmiany w węźle */
};
```

Większość tych wartości objaśniają komentarze. Typy danych, jak dev_t czy ino_t, są zdefiniowane w nagłówku <sys/types.h>, który także należy dołączyć do programu.

Składowa st_mode zawiera zbiór znaczników opisujących atrybuty pliku. Definicje tych znaczników są również zawarte w nagłówku <sys/stat.h>; tutaj potrzebujemy tylko kilku z nich, dotyczących rodzaju pliku:

```
#define     S_IFMT  0160000     /* rodzaj pliku: */
#define     S_IFDIR   0040000   /* skorowidz */
#define     S_IFCHR   0020000   /* specjalny znakowy */
#define     S_IFBLK   0060000   /* specjalny blokowy */
#define     S_IFREG   0100000   /* zwyczajny */

/* ... */
```

Teraz jesteśmy już gotowi do napisania programu fsize. Jeżeli z atrybutów otrzymanych od funkcji stat wynika, że plik nie jest skorowidzem, to jego rozmiar mamy w zasięgu ręki i możemy od razu go wypisać. Jeżeli jednak plik jest skorowidzem, to musimy w tym skorowidzu przeglądać kolejno plik po pliku; każdy z tych plików może być podskorowidzem, zatem proces jest rekurencyjny.

Funkcja główna zajmuje się argumentami wywołania programu; każdy z tych argumentów jest przekazywany funkcji fsize.

```
#include <stdio.h>
#include <string.h>
#include "syscalls.h"
#include <fcntl.h>        /* znaczniki dla czytania i pisania */
#include <sys/types.h> /* definicje typów */
#include <sys/stat.h>   /* struktura zwracana przez stat */
#include "dirent.h"

void fsize(char *);

/* wypisz rozmiary plików */
main(int argc, char **argv)
{
    if (argc == 1) /* domyślnie: bieżący skorowidz */
        fsize(".");
    else
        while (--argc > 0)
            fsize(*++argv);
    return 0;
}
```

Funkcja fsize wypisuje rozmiary plików. Jeśli jednak plik jest skorowidzem, to fsize najpierw wywołuje funkcję dirwalk, która przetwarza wszystkie pliki w tym skorowidzu. Zwróć uwagę na to, jak skorzystano ze znaczników S_IFMT i S_IFDIR (pochodzących z nagłówka <sys/stat.h>) przy sprawdzeniu, czy dany plik jest skorowidzem. Szczególne znaczenie ma tu postawienie nawiasów, gdyż priorytet operatora bitowej koniunkcji & jest niższy niż priorytet operatora przyrównania ==.

```
int stat(char *, struct stat *);
void dirwalk(char *, void (*fcn)(char *));

/* fsize: wypisz rozmiar pliku name */
void fsize(char *name)
{
    struct stat stbuf;
```

```
    if (stat(name, &stbuf) == -1) {
        fprintf(stderr, "fsize: nie mogę znaleźć %s\n", name);
        return;
    }
    if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
        dirwalk(name, fsize);   /* skorowidz: rekurencja */
    printf("%8ld %s\n", stbuf.st_size, name);
}
```

Funkcja dirwalk jest ogólną procedurą, która stosuje pewną funkcję dla każdego pliku zawartego w skorowidzu. Procedura ta otwiera skorowidz i w pętli przetwarza kolejno każdy plik, wywołując dla niego daną funkcję. Na koniec dirwalk zamyka skorowidz i wraca do miejsca wywołania. Funkcja fsize wywołuje dirwalk dla każdego skorowidza, zatem obie te funkcje wywołują się nawzajem rekurencyjnie.

```
#define MAX_PATH 1024

/* dirwalk: zastosuj funkcję fcn do wszystkich plików w dir */
void dirwalk(char *dir, void (*fcn)(char *))
{
    char name[MAX_PATH];
    Dirent *dp;
    DIR *dfd;

    if ((dfd = opendir(dir)) == NULL) {
        fprintf(stderr, "dirwalk: nie mogę otworzyć %s\n", dir);
        return;
    }
    while ((dp = readdir(dfd)) != NULL) {
        if (strcmp(dp->name, ".") == 0
         || strcmp(dp->name, "..") == 0)
            continue;       /* pomiń siebie i przodka */
        if (strlen(dir)+strlen(dp->name)+2 > sizeof(name))
            fprintf(stderr, "dirwalk: za długa nazwa %s/%s\n",
                    dir, dp->name);
        else {
            sprintf(name, "%s/%s", dir, dp->name);
            (*fcn)(name);
        }
    }
    closedir(dfd);
}
```

Każde wywołanie funkcji readdir zwraca wskaźnik do informacji o następnym pliku lub NULL, jeśli nie ma więcej plików. Każdy skorowidz zawiera pozycję nazwaną ".", odnoszącą się do niego samego, oraz pozycję o nazwie "..", odnoszącą się do jego przodka; pozycje te należy pominąć, inaczej bowiem program będzie działać bez końca.

Dotychczas kod był niezależny od rzeczywistego formatu skorowidza. W następnym kroku pokażemy minimalne wersje funkcji opendir, readdir i closedir, zrealizowane w konkretnym systemie. Napisane przez nas podprogramy są poprawne w systemach Unix Version 7 i Unix System V, korzystają bowiem z informacji o skorowidzu zawartych w nagłówku <sys/dir.h>:

```
#ifndef DIRSIZ
#define DIRSIZ      14
#endif
struct direct      /* opis pozycji skorowidza */
{
   ino_t d_ino;                /* numer węzła */
   char  d_name[DIRSIZ]; /* długa nazwa nie ma kończącego '\0' */
};
```

W pewnych wersjach systemu dopuszcza się jeszcze dłuższe nazwy, a struktura pozycji skorowidza jest bardziej skomplikowana.

Typ ino_t, skonstruowany za pomocą typedef, opisuje indeks węzła w tablicy węzłów. Zdarza się, że jest nim unsigned short – jak w systemie, którego regularnie używamy. Niemniej jednak informacji tego rodzaju nie włącza się do programu: w różnych systemach może być różnie, dlatego też lepiej zastosować typedef. Kompletny zestaw typów „systemowych'' można znaleźć w nagłówku <sys/types.h>.

Funkcja opendir otwiera plik podany jako skorowidz i sprawdza, czy rzeczywiście jest on skorowidzem (tym razem zastosowaliśmy funkcję fstat, która różni się od stat tylko tym, że korzysta z deskryptora pliku). Następnie skorowidzowi przydziela się strukturę DIR i rejestruje w niej uzyskane informacje*:

* Uwaga: tak napisana funkcja opendir nie zamyka pliku w przypadku niespełnienia jednego z wymaganych warunków. Po dłuższej serii niepowodzeń z różnymi nazwami plików może się wyczerpać zbiór deskryptorów przydzielonych programowi. – *Przyp. tłum.*

```
int fstat(int fd, struct stat *);

/* opendir: otwórz skorowidz dla wywołań readdir */
DIR *opendir(char *dirname)
{
    int fd;
    struct stat stbuf;
    DIR *dp;

    if ((fd = open(dirname, O_RDONLY, 0)) == -1
     || fstat(fd, &stbuf) == -1
     || (stbuf.st_mode & S_IFMT) != S_IFDIR
     || (dp = (DIR *) malloc(sizeof(DIR))) == NULL)
        return NULL;
    dp->fd = fd;
    return dp;
}
```

Funkcja closedir zamyka skorowidz i zwalnia przydzieloną pamięć:

```
/* closedir: zamknij skorowidz otwarty przez opendir */
void closedir(DIR *dp)
{
    if (dp) {
        close(dp->fd);
        free(dp);
    }
}
```

I na ostatku funkcja readdir czyta każdą pozycję skorowidza za pomocą funkcji read. Jeśli pozycja w skorowidzu nie jest aktualnie używana (ponieważ plik został usunięty), to numer węzła tej pozycji jest równy zero, zatem taką pozycję się pomija. W pozostałych przypadkach numer węzła i nazwę umieszcza się w statycznej strukturze, a wskaźnik do tej struktury udostępnia użytkownikowi. Każde wywołanie funkcji readdir zamazuje informacje uzyskane w poprzednim wywołaniu.

```
#include <sys/dir.h> /* opis lokalnej struktury skorowidza */

/* readdir: czytaj kolejno pozycje skorowidza */
Dirent *readdir(DIR *dp)
{
    struct direct dirbuf;  /* lokalna struktura skorowidza */
    static Dirent d;       /* zwracana struktura przenośna */
```

```
    while (read(dp->fd, (char *) &dirbuf, sizeof(dirbuf)) == sizeof(dirbuf)) {
        if (dirbuf.d_ino == 0) /* pusta pozycja */
            continue;
        d.ino = dirbuf.d_ino;
        strncpy(d.name, dirbuf.d_name, DIRSIZ);
        d.name[DIRSIZ] = '\0'; /* zapewnia poprawny koniec */
        return &d;
    }
    return NULL;
}
```

Chociaż fsize jest programem raczej wyspecjalizowanym, ilustruje jednak kilka ważnych idei. Po pierwsze, nie wszystkie programy są „programami systemowymi"; niektóre po prostu korzystają z informacji udzielonych przez system operacyjny. Dla tych programów zasadnicze znaczenie ma to, że reprezentacja takich informacji „systemowych" występuje jedynie w standardowych nagłówkach, a zatem – zamiast wciskać do programu niezbędne deklaracje – w programie umieszcza się te nagłówki. I drugie spostrzeżenie: można – z pewną dozą ostrożności – zapewnić dostęp do obiektów zależnych od systemu w sposób względnie od systemu niezależny. Funkcje z biblioteki standardowej są tego najlepszym przykładem.

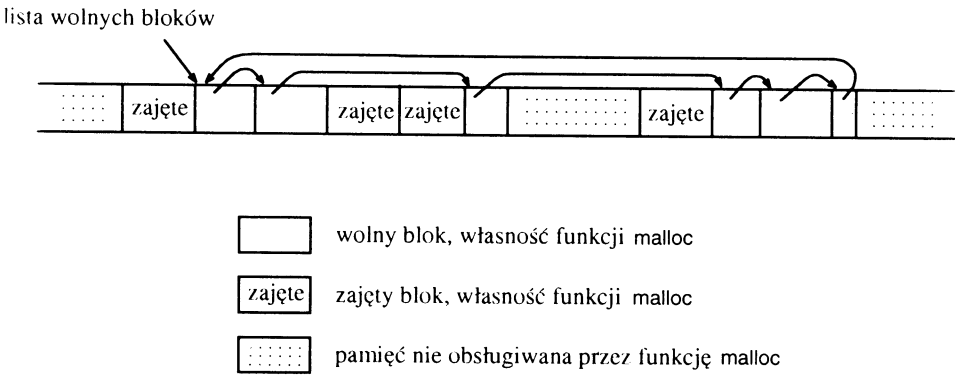
Ćwiczenie 8.5. Zmień program fsize tak, aby wypisywał również inne informacje pochodzące z węzła pliku.

8.7 Przykład – dystrybutor pamięci

W rozdziale 5 pokazaliśmy bardzo ograniczoną wersję dystrybutora pamięci, zrealizowaną metodą stosową. Wersja, którą teraz napiszemy, jest pozbawiona tych ograniczeń. Wywołania funkcji malloc i free mogą wystąpić w dowolnej kolejności; malloc zwraca się do systemu operacyjnego o dodatkową pamięć. Funkcje te ilustrują niektóre z naszych rozważań dotyczących tworzenia programów maszynowozależnych w sposób w miarę niezależny od maszyny. Demonstrują także autentyczne zastosowanie struktur, unii i definicji typów.

Zamiast przydzielać pamięć z tablicy o ustalonym rozmiarze, przydzielonej programowi podczas kompilacji, funkcja malloc będzie w miarę potrzeby pobierać pamięć od systemu operacyjnego. Różne fragmenty programu mogą niezależnie od siebie żądać przydziału pamięci, nie korzystając z tego dystrybutora, toteż obszar pamięci, którym zarządza funkcja malloc, może nie być ciągły. Z tego powodu wolna pamięć jest zorganizowana w łańcuch bloków. Każdy blok zawiera rozmiar, wskaźnik do następ-

nego bloku oraz obszar wolnej pamięci. Bloki są uporządkowane w kolejności rosnących adresów pamięci, a ostatni blok (o najwyższym adresie) wskazuje na pierwszy.



Żądanie przydziału pamięci powoduje wyszukanie w łańcuchu wolnych bloków obszaru o wystarczająco dużym rozmiarze. Ten algorytm jest nazywany „pierwszą przymiarką" (ang. *first fit*) w przeciwieństwie do algorytmu „najlepszej przymiarki" (ang. *best fit*), w którym szuka się najmniejszego z bloków spełniających to żądanie. Jeżeli obszar ma dokładnie żądany rozmiar, to blok jest odłączany od łańcucha i przekazywany użytkownikowi. Zbyt duży obszar jest dzielony tak, aby wystarczająco dużą część przekazać użytkownikowi; reszta pozostaje w łańcuchu wolnych bloków. Gdy w łańcuchu nie ma dostatecznie dużego obszaru, wówczas od systemu operacyjnego pobiera się nowy, wielki kawał pamięci i przyłącza go do łańcucha.

Zwalnianie pamięci także powoduje przeszukanie łańcucha wolnych bloków w celu znalezienia właściwego miejsca dla zwolnionego bloku. Jeżeli ten blok przylega z którejkolwiek strony do innego wolnego bloku, to zrasta się z nim, tworząc jeden większy blok; pamięć nie będzie więc zbyt rozczłonkowana. Sprawdzanie przylegania jest proste, ponieważ łańcuch wolnych bloków jest uporządkowany w kolejności rosnących adresów.

Pozostaje do omówienia zasygnalizowany już w rozdz. 5 problem: czy położenie obszarów pamięci przydzielanych przez funkcję malloc odpowiada wymaganiom obiektów, które będą w nich umieszczane. Chociaż maszyny się różnią, to dla każdej istnieje typ nakładający najsilniejsze ograniczenia: jeśli obiekty tego typu mogą być umieszczone w pamięci pod pewnym szczególnym adresem, to mogą również obiekty wszystkich innych typów. Dla pewnych maszyn takim wymagającym typem jest double; dla innych wystarczy int lub long.

Każdy wolny blok zawiera wskaźnik do następnego bloku łańcucha, liczbę określającą rozmiar bloku oraz sam obszar wolnej pamięci; taka informacja sterująca na początku bloku nazywa się „nagłówkiem" (ang. *header*). Dla uproszczenia kontroli

położenia w pamięci wszystkie bloki mają rozmiar będący wielokrotnością rozmiaru nagłówka, nagłówek zaś jest położony poprawnie. Efekt ten osiągnięto za pomocą unii zawierającej wymaganą strukturę nagłówka oraz obiekt o typie narzucającym najsilniejsze ograniczenia – tutaj arbitralnie przyjęliśmy, że jest nim long.

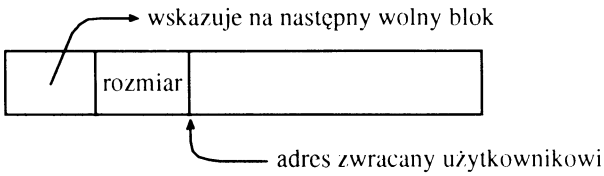
```
typedef long Align;   /* położenie dla obiektów typu long */

union header {      /* nagłówek bloku */
   struct {
      union header *ptr;      /* następny wolny blok */
      unsigned size;          /* rozmiar tego bloku */
   } s;
   Align x;           /* wymuszenie położenia bloku */
};

typedef union header Header;
```

Składowa unii o typie Align nigdy nie będzie używana; służy jedynie do wymuszenia położenia całego nagłówka odpowiednio dla najbardziej wymagającego typu.

Żądany rozmiar obszaru (w znakach) będzie przez funkcję malloc zaokrąglony do właściwej liczby porcji o rozmiarze nagłówka. Przydzielony blok zawiera o jedną porcję więcej – na sam nagłówek; ta liczba porcji jest właśnie wartością zanotowaną w składowej size nagłówka. Wskaźnik zwracany przez funkcję malloc wskazuje na wolny obszar, a nie na jego nagłówek. Z przydzieloną pamięcią użytkownik może robić co chce, ale jeśli cokolwiek zostanie zapisane na zewnątrz tej pamięci, to łańcuch bloków zamieni się w śmietnik.



Blok zwracany przez funkcję malloc

Składowa rozmiaru jest niezbędna, ponieważ bloki kontrolowane przez funkcję malloc mogą nie tworzyć ciągłej pamięci – arytmetyka na wskaźnikach nie jest w takim przypadku możliwa.

Zmienna base służy do rozpoczęcia działania dystrybutora. Jeśli wskaźnik wolnego bloku freep jest równy NULL, jak przy pierwszym wywołaniu funkcji malloc, to bu-

duje ona zdegenerowany łańcuch wolnych bloków, który zawiera jeden blok o rozmiarze zero ze wskaźnikiem do niego samego. Następnie – w każdym przypadku – malloc przeszukuje listę wolnych bloków. Poszukiwanie bloku o wymaganym rozmiarze rozpoczyna się od miejsca (freep), w którym ostatnio znaleziono wolny blok; strategia ta pomaga utrzymać łańcuch w jednorodnej postaci. Jeżeli znaleziony blok jest za duży, to użytkownikowi przekazuje się końcową część obszaru bloku – wystarczy wówczas jedynie uaktualnić rozmiar w oryginalnym nagłówku bloku. Przekazywany użytkownikowi wskaźnik zawsze pokazuje na wolny obszar wewnątrz bloku, rozpoczynający się tuż za nagłówkiem tego bloku.

```
static Header base;   /* pusty łańcuch na początek */
static Header *freep = NULL; /* początek łańcucha wolnych bloków */

/* malloc: ogólny dystrybutor pamięci */
void *malloc(unsigned nbytes)
{
    Header *p, *prevp;
    Header *morecore(unsigned);
    unsigned nunits;          /* liczba żądanych porcji */

    nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1;
    if ((prevp = freep) == NULL) {      /* pusty łańcuch */
        base.s.ptr = freep = prevp = &base;
        base.s.size = 0;
    }
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) {      /* dostatecznie duży blok */
            if (p->s.size == nunits )   /* dokładnie taki */
                prevp->s.ptr = p->s.ptr;
            else {                      /* za duży: przydziel koniec */
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void *) (p+1);
        }
        if (p == freep)         /* przeszukano cały łańcuch */
            if ((p = morecore(nunits)) == NULL)
                return NULL; /* brak pamięci */
    }
}
```

Funkcja morecore otrzymuje pamięć od systemu operacyjnego. Szczegóły jej realizacji są różne w różnych systemach. Pobieranie pamięci od systemu jest operacją dosyć kosztowną, nie chcemy więc tego robić przy każdym wywołaniu funkcji malloc. Wobec tego funkcja morecore żąda co najmniej NALLOC porcji; taki duży blok będzie w miarę potrzeby „posiekany'' na mniejsze kawałki. Po wypełnieniu składowej rozmiaru morecore wprowadza ten blok na scenę, wywołując funkcję free.

W systemie Unix odwołanie systemowe sbrk(n) zwraca wskaźnik do nowej porcji n bajtów pamięci. Jeżeli brakuje pamięci, to sbrk zwraca –1, chociaż NULL byłoby bardziej odpowiednie. Wartość –1 koniecznie trzeba zrzutować do typu char *, aby mogła być porównywana z wartością zwracaną przez funkcję. I znów dzięki rzutowaniu funkcja jest względnie odporna na szczegóły reprezentacji wskaźników w różnych maszynach. Niemniej jednak ciągle przyjmuje się założenie, że wskaźniki do różnych bloków dostarczanych przez sbrk mogą być sensownie porównywane. Tego standard nie gwarantuje; pozwala jedynie na porównania wskaźników w obrębie jednej tablicy. A zatem nasza wersja funkcji malloc jest przenośna tylko wśród tych maszyn, w których ogólne porównywanie wskaźników ma sens.

```
#define NALLOC   1024   /* minimalna liczba żądanych porcji */

/* morecore: poproś system o pamięć dla nu porcji */
static Header *morecore(unsigned nu)
{
    char *cp, *sbrk(int);
    Header *up;

    if (nu < NALLOC)
        nu = NALLOC;
    cp = sbrk(nu * sizeof(Header));
    if (cp == (char *) -1)   /* nie ma wolnej pamięci */
        return NULL;
    up = (Header *) cp;
    up->s.size = nu;
    free((void *) (up+1));
    return freep;
}
```

Pozostała nam do napisania tylko funkcja free. W poszukiwaniu miejsca dla zwalnianego bloku przegląda ona łańcuch wolnych bloków, rozpoczynając od freep. Miej-

sce to znajduje się albo między dwoma istniejącymi blokami, albo na jednym z końców łańcucha. W każdym przypadku, jeżeli zwalniany blok przylega do któregokolwiek z sąsiadów, to przylegające bloki zostaną sklejone. Jedynym problemem jest zapewnienie, aby wskaźniki poprawnie wskazywały na właściwe obiekty o właściwych rozmiarach.

```
/* free: zwróć blok ap do łańcucha wolnych bloków */
void free(void *ap)
{
    Header *bp, *p;

    bp = (Header *)ap - 1;   /* wskaż na nagłówek */
    for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
            break;      /* miejsce na jednym z końców łańcucha */

    if (bp + bp->s.size == p->s.ptr) { /* połącz z następnym blokiem */
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } else
        bp->s.ptr = p->s.ptr;
    if (p + p->s.size == bp) {  /* połącz z poprzednim blokiem */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
        p->s.ptr = bp;              /* samodzielny blok */
    freep = p;
}
```

Przydział pamięci istotnie zależy od maszyny, pokazane przykłady ilustrują jednak, jak zależności te można kontrolować i ograniczać do bardzo małego fragmentu programu. Zastosowanie konstrukcji typedef razem z union zapewnia prawidłowe położenie obszaru (pod warunkiem, że funkcja sbrk daje odpowiedni wskaźnik). Rzuty wymuszają jawne przekształcenia wskaźników, a nawet radzą sobie ze źle zdefiniowanym łączem z systemem operacyjnym. Choć uwagi te dotyczą przydziału pamięci, takie ogólne podejście można równie dobrze stosować w wielu innych sytuacjach.

Ćwiczenie 8.6. Funkcja calloc(n,size) z biblioteki standardowej zwraca wskaźnik do n obiektów o rozmiarze size, których pamięć została wypełniona zerami.

Napisz funkcję calloc, korzystając z wywołań malloc lub traktując tę funkcję jako wzorzec.

Ćwiczenie 8.7. Funkcja malloc akceptuje specyfikację rozmiaru bez sprawdzenia jej sensowności; funkcja free zakłada, że zwalniany blok zawiera poprawne pole rozmiaru. Zmień te funkcje tak, aby zadawały sobie więcej trudu przy wykrywaniu błędów.

Ćwiczenie 8.8. Napisz podprogram bfree(p,n), który dowolny n-znakowy blok pamięci o adresie p dołącza do łańcucha wolnych bloków obsługiwanych przez funkcje malloc i free. Dzięki bfree użytkownik będzie mógł w każdej chwili dołączyć do tego łańcucha tablice statyczne lub zewnętrzne.

DODATEK A PRZEWODNIK JĘZYKA C

A1 Wstęp

W dniu 31 października 1988 r. komitetowi ANSI przedstawiono w formie szkicu opis języka C do zatwierdzenia jako „American National Standard for Information Systems – Programming Language C, X3.159-1989". Niniejszy przewodnik zawiera opis języka C, który jest interpretacją zaproponowanego Standardu, a nie sam Standard; niemniej jednak dołożono wielu starań, aby uczynić go wiarygodnym.

Dokument ten w znacznej części opiera się na ogólnym zarysie Standardu, który z kolei był oparty na pierwszym wydaniu tej książki*; występują jednak duże różnice organizacyjne. Poza przemianowaniem kilku produkcji i brakiem formalizmu przy definiowaniu jednostek leksykalnych czy preprocesora, podana tu (dla kompletności języka) gramatyka jest równoważna zdefiniowanej przez Standard.

> Komentarze zamieszczone w tym przewodniku są złożone z wcięciem i mniejszym drukiem, tak jak niniejszy komentarz. Najczęściej komentarze podkreślają różnice między Standardem ANSI języka C a językiem zdefiniowanym w pierwszym wydaniu tej książki lub ulepszeniami wprowadzonymi później przez różne kompilatory.

A2 Konwencje leksykalne

Program składa się z jednej lub więcej *jednostek tłumaczenia* zapisanych w plikach. Program jest tłumaczony w kilku fazach, które opisano w p. A12. W pierwszej fazie dokonuje się wstępnej leksykalnej transformacji programu, tzn. interpretuje wiersze poleceń rozpoczynające się znakiem # oraz wykonuje makrodefinicje i makrorozwinięcia. Po zakończeniu fazy preprocesora, opisanej w p. A12, program jest zredukowany do ciągu jednostek leksykalnych.

* Patrz przypis na str. 11.

A2.1 Jednostki leksykalne

Istnieje sześć klas jednostek leksykalnych: identyfikatory, słowa kluczowe, stałe, napisy, operatory oraz różne separatory. Ignoruje się odstępy, znaki poziomej i pionowej tabulacji, znaki nowego wiersza i nowej strony oraz komentarze (wszystkie razem zwane „białymi plamami"), chyba że rozdzielają jednostki leksykalne. Niektóre białe plamy są jednak potrzebne do rozdzielenia przylegających do siebie identyfikatorów, słów kluczowych i stałych.

Jeżeli rozbiór leksykalny wejściowego tekstu przeprowadzono do pewnego znaku włącznie, to za następną jednostkę leksykalną uważa się najdłuższy ciąg znaków, z których można utworzyć jednostkę.

A2.2 Komentarze

Komentarz rozpoczyna się znakami /*, a kończy się znakami */. Komentarze nie mogą być zagnieżdżone i nie mogą wystąpić w napisach i stałych znakowych.

A2.3 Identyfikatory (nazwy)

Identyfikator jest ciągiem liter i cyfr. Pierwszym znakiem ciągu musi być litera; znak podkreślenia _ zalicza się do liter. Rozróżnia się wielkie i małe litery alfabetu. Identyfikatory mogą być dowolnej długości, przy czym dla identyfikatorów wewnętrznych znaczenie ma co najmniej 31 znaków początkowych; w niektórych implementacjach może być ich więcej. Identyfikatorami wewnętrznymi są nazwy makr i wszystkie inne nazwy nie mające zewnętrznej łączności (p. A11.2). Identyfikatory o zewnętrznej łączności są bardziej ograniczone: w konkretnej implementacji ich znacząca długość może wynosić tylko sześć początkowych znaków bez rozróżniania małych i wielkich liter.

A2.4 Słowa kluczowe

Następujące identyfikatory są zastrzeżone dla słów kluczowych i nie mogą być inaczej używane:

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

W niektórych implementacjach rezerwuje się również słowa fortran i asm*.

> Słowa kluczowe const, signed i volatile są nowe w ANSI C; enum i void – nieznane za czasów pierwszego wydania książki – są w powszechnym użyciu; entry, nigdy nie używane, nie jest już zarezerwowane.

A2.5 Stałe

Wyróżnia się kilka rodzajów stałych. Wszystkie mają jakiś typ; typy podstawowe są omówione w p. A4.2.

stała:
stała-całkowita
stała-znakowa
stała-zmiennopozycyjna
stała-wyliczenia

A2.5.1 Stałe całkowite

Stałą całkowitą składającą się z ciągu cyfr uważa się za ósemkową, jeżeli rozpoczyna się cyfrą 0; w przeciwnym przypadku – za dziesiętną. Stałe ósemkowe nie zawierają cyfr 8 i 9. Ciąg cyfr rozpoczynający się znakami 0x lub 0X (cyfra zero) jest stałą szesnastkową. Do cyfr szesnastkowych zalicza się litery od a (lub A) do f (lub F) o wartościach od 10 do 15.

Stała całkowita może być opatrzona przyrostkiem u lub U do oznaczenia, że jest bez znaku. Może być również opatrzona przyrostkiem l (litera l) lub L do oznaczenia, że jest długa (long).

Typ stałej całkowitej zależy od jej postaci, wartości i przyrostka. (Omówienie typów znajduje się w p. A4.) Typem stałej dziesiętnej bez przyrostka jest pierwszy z następujących typów, w którym można wyrazić jej wartość: int, long int lub unsigned long int. Typem stałej ósemkowej lub szesnastkowej bez przyrostka jest pierwszy możliwy z następujących typów: int, unsigned int, long int lub unsigned long int. Jeśli stała ma przyrostek u lub U, to jej typem jest pierwszy możliwy z typów unsigned int lub unsigned long int. Jeśli stała ma przyrostek l lub L, to możliwymi typami są long int i unsigned long int.

> Opracowanie typów stałych całkowitych daleko odbiega od tego z pierwszego wydania książki, gdzie duże stałe miały po prostu typ long. Przyrostki U są nowe.

* Kompilatory działające pod systemem operacyjnym MS-DOS na maszynach IBM-PC zwykle rezerwują również słowa near i far dla oznaczenia różnej długości wskaźników. – *Przyp. tłum.*

A2.5.2 Stałe znakowe

Stała znakowa jest ciągiem złożonym z jednego lub więcej znaków, zawartym w apostrofach, np. 'x'. Wartością stałej znakowej zawierającej tylko jeden znak jest numeryczna wartość tego znaku w zbiorze znaków maszyny wykonującej program. Wartość stałej wieloznakowej zależy od implementacji.

Stałe znakowe nie zawierają apostrofów i znaków nowego wiersza; dla wyrażenia tych oraz kilku innych znaków można stosować następujące sekwencje specjalne:

nowy wiersz	NL(LF)	\n	kreska ukośna w lewo	\	\\
tabulacja pozioma	HT	\t	znak zapytania	?	\?
tabulacja pionowa	VT	\v	apostrof	'	\'
cofanie	BS	\b	cudzysłów	"	\"
powrót karetki	CR	\r	liczba ósemkowa	*ooo*	*ooo*
nowa strona	FF	\f	liczba szesnastkowa	*hh*	\x*hh*
alarm	BEL	\a			

Sekwencja *ooo* składa się ze znaku \, po którym następuje jedna, dwie lub trzy cyfry ósemkowe określające wartość żądanego znaku. Szczególnym przypadkiem takiej sekwencji jest \0 (bez dalszych cyfr) reprezentujące znak NUL. Sekwencja \x*hh* składa się ze znaku \, litery x i ciągu cyfr szesnastkowych określających wartość znaku. Nie ma ograniczenia dotyczącego liczby cyfr, ale wynik nie jest zdefiniowany, jeżeli otrzymana wartość przekracza wartość największego znaku. Jeśli w danej implementacji typ char jest traktowany jak typ ze znakiem arytmetycznym, to bit znaku w ósemkowych i szesnastkowych sekwencjach specjalnych jest powielany tak samo, jak przy rzutowaniu do typu char. Wynik nie jest zdefiniowany, jeśli znak następujący po \ nie jest żadnym z wymienionych znaków.

W niektórych implementacjach istnieje rozszerzony zbiór znaków, który nie może być reprezentowany przez typ char. Zapis stałych z tego rozszerzonego zbioru poprzedza się literą L, np. L'x'. Stałe te nazywa się rozszerzonymi stałymi znakowymi. Mają one typ wchar_t – typ całkowity zdefiniowany w standardowym nagłówku <stddef.h>. Podobnie jak w przypadku zwykłych stałych znakowych, można tu również stosować ósemkowe i szesnastkowe sekwencje specjalne; wynik nie jest zdefiniowany, jeśli wskazana wartość przekracza możliwości typu wchar_t.

> Niektóre z sekwencji specjalnych są nowe, w szczególności szesnastkowa reprezentacja znaków. Rozszerzone znaki są także nowe. Zbiory znaków stosowane powszechnie w Ameryce i zachodniej Europie mogą być zakodowane w typie char; głównym powodem wprowadzenia typu wchar_t było dopasowanie się do języków azjatyckich.

A2.5.3 Stałe zmiennopozycyjne

Stała zmiennopozycyjna składa się z części całkowitej, kropki dziesiętnej, części ułamkowej, litery e lub E, wykładnika potęgi (być może ze znakiem) i opcjonalnego przyrostka typu (f, F, l lub L). Część całkowita i ułamkowa są ciągami cyfr. Część całkowitą albo część ułamkową (ale nie obie jednocześnie) można pominąć. Można także pominąć kropkę dziesiętną lub część wykładniczą wraz z literą e (ale nie obie naraz). Typ stałej wyznacza przyrostek: f lub F oznacza float, l lub L oznacza long double, brak przyrostka oznacza double.

Przyrostki w stałych zmiennopozycyjnych są nowe.

A2.5.4 Stałe wyliczeń

Identyfikatory zadeklarowane jako wyliczniki (patrz p. A8.4) są stałymi typu int.

A2.6 Napisy

Napis (stała napisowa) jest ciągiem znaków ujętym w znaki cudzysłowu, np. "...". Typem napisu jest „tablica znakowa", a klasą pamięci – static (patrz p. A4). Jego wartością jest dany ciąg znaków. Od implementacji zależy, czy rozróżnia się identyczne napisy. Zachowanie się programu, który próbuje modyfikować napisy, nie jest zdefiniowane.

Sąsiadujące ze sobą napisy są łączone w jeden napis. Po ewentualnych połączeniach, na koniec napisu jest dopisywany znak zerowy \0, aby program przeglądający tekst mógł znaleźć jego koniec. Napisy nie zawierają znaków cudzysłowu i znaków nowego wiersza. Do wyrażenia tych znaków można stosować te same sekwencje specjalne, co w stałych znakowych.

Podobnie jak w przypadku stałych znakowych, napisy złożone z rozszerzonych znaków poprzedza się literą L, np. L"...". Mają one typ „tablica elementów typu wchar_t". Łączenie normalnych i rozszerzonych napisów nie jest zdefiniowane.

Zastrzeżenie, że identyczne napisy mogą nie być rozróżniane, zakaz modyfikowania napisów, jak również łączenie sąsiadujących napisów są nowe w ANSI C. Rozszerzone napisy są nowe.

A3 Notacja opisu składni

W opisie składni, stosowanym w przewodniku, kategorie składniowe wyróżnia się *kursywą*, a słowa i znaki języka C pismem specjalnym. Alternatywne kategorie są zazwy-

czaj umieszczane w oddzielnych wierszach; w kilku przypadkach liczny zbiór krótkich alternatyw umieszczono w jednym wierszu i opatrzono zwrotem „jeden z". Symbole terminalne i nieterminalne, które można pominąć, zostały oznaczone przyrostkiem *opc*; zapis

{ *wyrażenie*$_{opc}$ }

oznacza więc opcjonalne wyrażenie ujęte w klamry. Pełną składnię języka podano w p. A13.

> W przeciwieństwie do gramatyki zamieszczonej w pierwszym wydaniu książki, w niniejszej gramatyce wyraźnie zaznaczono pierwszeństwo i łączność operatorów w wyrażeniach.

A4 Znaczenie identyfikatorów

Identyfikatory (nazwy) odnoszą się do różnych rzeczy: funkcji; etykietek struktur, unii i wyliczeń; składowych struktur i unii; stałych wyliczeń; nazw typów (typedef) i obiektów. Obiekt, zwany też zmienną, jest miejscem w pamięci; jego interpretacja opiera się na dwóch podstawowych atrybutach: *klasie pamięci* i *typie*. Klasa pamięci decyduje o żywotności pamięci związanej z identyfikowanym obiektem; typ nadaje znaczenie wartości obiektu. Nazwa ma także zasięg – tę część programu, w której jest ona znana, oraz łączność – która określa, czy ta sama nazwa z innym zasięgiem odnosi się do tego samego obiektu lub funkcji czy nie. Zasięg i łączność są omówione w p. A11.

A4.1 Klasy pamięci

Istnieją dwie klasy pamięci: automatyczna i statyczna. Klasę pamięci obiektu określa kilka słów kluczowych razem z kontekstem zawierającym deklarację obiektu. Obiekty automatyczne są lokalne w bloku (p. A9.3) i znikają, gdy sterowanie opuści blok. Deklaracje zawarte wewnątrz bloku tworzą obiekty automatyczne, jeśli nie podano specyfikacji klasy pamięci lub użyto specyfikatora auto. Obiekty zadeklarowane jako register są automatyczne i są umieszczane (jeśli to możliwe) w szybkich rejestrach maszyny.

Obiekty statyczne mogą być lokalne dla bloku lub zewnętrzne dla wszystkich bloków; w każdym jednak przypadku zachowują swoje wartości po opuszczeniu i ponownym wejściu do funkcji lub bloku. Wewnątrz bloku (również bloku tworzącego treść funkcji) obiekty statyczne deklaruje się za pomocą słowa static. Obiekty zadeklarowane

na zewnątrz wszystkich bloków, na tym samym poziomie co definicje funkcji, są zawsze statyczne. Można je uczynić wewnętrznymi dla danej jednostki kompilacji za pomocą słowa kluczowego static; to nadaje im *łączność wewnętrzną*. Obiekty stają się globalne dla całego programu, gdy ominie się specyfikację klasy pamięci lub zastosuje słowo kluczowe extern; to nadaje im *łączność zewnętrzną*.

A4.2 Typy podstawowe

Wyróżnia się kilka podstawowych typów danych. Opisany w dodatku B standardowy nagłówek <limits.h> definiuje największe i najmniejsze wartości dla tych typów w lokalnej implementacji. Liczby podane w dodatku B pokazują najmniejsze dopuszczalne wielkości.

Obiekty zadeklarowane jako znakowe (char) są dostatecznie duże, aby pomieścić dowolny element zbioru znaków danej instalacji. Wartość obiektu znakowego, do którego wstawiono naturalny znak z tego zbioru, jest nieujemną liczbą całkowitą równą wartości kodu tego znaku. Do zmiennych znakowych można wstawiać także inne wartości, ale zakres tych wartości, a zwłaszcza to, czy mają znak arytmetyczny, zależy od implementacji.

Obiekty zadeklarowane jako unsigned char zajmują tyle samo miejsca, co zwykłe znaki, są jednak zawsze nieujemne; obiekty zadeklarowane jako signed char także zajmują tyle samo miejsca, co zwykłe znaki.

> Typ unsigned char nie występował w pierwszym wydaniu książki, ale jest w powszechnym użyciu. Typ signed char jest nowy.

Oprócz obiektów typu char są jeszcze obiekty całkowite o trzech różnych długościach deklarowanych jako: short int (krótka), int (zwykła) i long int (długa). Zwykłe wielkości całkowite mają naturalny rozmiar wynikający z architektury danej maszyny; pozostałe rozmiary wprowadzono dla zaspokojenia szczególnych potrzeb. Dłuższe obiekty całkowite zajmują co najmniej tyle pamięci, co krótsze, ale w konkretnej implementacji zwykłe obiekty całkowite mogą być utożsamione albo z krótkimi obiektami, albo z długimi. Jeśli nie powiedziano inaczej, wszystkie odmiany typu int reprezentują wartości całkowite ze znakiem.

Obiekty całkowite bez znaku, deklarowane jako unsigned, podlegają regułom arytmetyki modulo 2^n, gdzie n jest liczbą bitów ich reprezentacji; w arytmetyce na wielkościach bez znaku nie może więc wystąpić nadmiar. Zbiór nieujemnych wartości, które można wstawić do obiektu ze znakiem, stanowi podzbiór zbioru wartości, które można wstawić do odpowiedniego obiektu bez znaku; reprezentacja pokrywających się wartości jest taka sama.

Wszystkie z typów zmiennopozycyjnych: pojedynczej (float), podwójnej (double) i rozszerzonej precyzji (long double) mogą być utożsamione, przy tym każdy następny typ z tej listy jest co najmniej tak precyzyjny, jak poprzedni.

> Typ long double jest nowy. Konstrukcja long float, w poprzednim wydaniu książki stosowana jako synonim double, została usunięta.

Wyliczenia są unikalnymi typami o wartościach całkowitych; z każdym wyliczeniem jest związany zbiór nazwanych stałych (p. A8.4). Wyliczenia zachowują się jak typy całkowite, przy czym kompilatory produkują zwykle komunikaty ostrzegawcze, jeśli jakiemuś obiektowi o typie wyliczeniowym przypisuje się wartość inną niż wartość jednej z jego stałych lub wyrażenia tego typu.

Ponieważ obiekty powyższych typów mogą być traktowane jak liczby, typy te będziemy nazywali *arytmetycznymi*. Typy char i int wszystkich rozmiarów (ze znakiem lub bez), a także wyliczenia będą wspólnie nazywane *całkowitymi*. Typy float, double i long double będą nazywane *zmiennopozycyjnymi*.

Typ void reprezentuje pusty zbiór wartości. Jest on używany jako typ funkcji, która

A4.3 Typy pochodne

Oprócz typów podstawowych mamy potencjalnie nieskończoną klasę typów pochodnych, konstruowanych z typów podstawowych:

tablice obiektów danego typu,
funkcje zwracające wartości danego typu,
wskaźniki do obiektów danego typu,
struktury zawierające zestawy obiektów różnych typów,
unie zawierające dowolny z zestawu obiektów o różnych typach.

Podane metody konstruowania obiektów mogą być na ogół stosowane rekurencyjnie.

A4.4 Kwalifikatory typów

Z typem obiektu mogą być związane kwalifikatory. Deklarując obiekt jako const wskazujemy, że ten obiekt nie będzie zmieniał wartości; deklarując go jako volatile wskazujemy, że będzie miał specjalne właściwości, ważne przy optymalizacji. Kwali-

A5 Obiekty i l-wartości

Obiekt jest nazwanym obszarem pamięci; *l-wartość* (ang. *lvalue*) jest wyrażeniem odnoszącym się do obiektu. Oczywistym przykładem l-wartości jest identyfikator o odpowiednim typie i klasie pamięci. Również niektóre operatory dają l-wartość: na przykład, jeśli E jest wyrażeniem wskaźnikowym, to *E jest l-wartością odnoszącą się do obiektu wskazywanego przez E. Nazwa „l-wartość'' wywodzi się z wyrażenia przypisania E1=E2, w którym lewy argument musi być l-wartością. Przy omawianiu poszczególnych operatorów zaznaczono, czy dany operator spodziewa się l-wartości jako argumentu i czy w wyniku daje l-wartość.

A6 Przekształcenia typów

W zależności od argumentów niektóre operatory mogą powodować przekształcenie wartości pewnego argumentu z jednego typu do innego. W tym punkcie wyjaśnimy, jakich wyników należy się spodziewać po takich przekształceniach. Podsumowanie przekształceń wymaganych przez większość zwykłych operatorów nastąpi w p. A6.5; jeśli zajdzie taka potrzeba, to będzie ono uzupełnione przy opisie poszczególnych operatorów.

A6.1 Promocja typu całkowitego

W każdym wyrażeniu, w którym może wystąpić wartość całkowita, mogą również wystąpić: znak, wartość całkowita krótka lub całkowite pole bitowe (wszystkie ze znakiem arytmetycznym lub bez) oraz obiekt typu wyliczeniowego. Jeśli typ int może reprezentować wszystkie wartości oryginalnego typu, to dana wartość jest przekształcana do typu int; w przeciwnym przypadku tę wartość przekształca się do typu unsigned int. Ten proces nazywa się *promocją typu całkowitego.*

A6.2 Przekształcenia całkowite

Dowolna wielkość całkowita jest przekształcana do danego typu bez znaku przez znalezienie takiej najmniejszej nieujemnej wartości, która jest przystająca (kongruentna) do tej wielkości całkowitej modulo największa wartość reprezentowalna w danym typie zwiększona o 1. W notacji uzupełnieniowej do 2 oznacza to obcięcie bitów z lewej strony, jeśli typ bez znaku jest krótszy, albo uzupełnienie zerami wartości bez znaku lub powielenie bitu znaku wartości ze znakiem, jeśli typ bez znaku jest dłuższy.

Gdy wartość całkowita jest przekształcana do typu ze znakiem, wówczas jej wartość nie ulegnie zmianie, jeśli może być reprezentowana w nowym typie; w przeciwnym przypadku wynik zależy od implementacji.

A6.3 Wartości całkowite i zmiennopozycyjne

Przy przekształcaniu wartości typu zmiennopozycyjnego do typu całkowitego gubi się jej część ułamkową; jeśli wyniku nie można wyrazić w tym typie całkowitym, to skutek przekształcenia nie jest zdefiniowany. W szczególności nie jest zdefiniowany wynik przekształcenia ujemnej wartości zmiennopozycyjnej do typu całkowitego bez znaku.

Jeśli wartość całkowita jest przekształcana do typu zmiennopozycyjnego i zawiera się w dopuszczalnym przedziale wartości tego typu, ale nie jest dokładnie reprezentowalna, to wynikiem jest najbliższa większa albo mniejsza reprezentowalna wartość. Jeśli wartość nie zawiera się w tym przedziale, skutek nie jest zdefiniowany.

A6.4 Typy zmiennopozycyjne

Przy przekształcaniu wielkości zmiennopozycyjnej do typu zmiennopozycyjnego o równej lub większej precyzji jej wartość nie zmienia się. Jeśli jest ona przekształcana do typu zmiennopozycyjnego o mniejszej precyzji i jej wartość zawiera się w reprezentowalnym przedziale, wynikiem jest najbliższa większa albo mniejsza reprezentowalna wartość. Jeśli wynik nie zawiera się w tym przedziale, skutek nie jest zdefiniowany.

A6.5 Przekształcenia arytmetyczne

Wiele operatorów powoduje przekształcenia swoich argumentów i określa typ wyniku według podobnych zasad. W efekcie chodzi o dopasowanie typów argumentów do wspólnego typu, który jest także typem wyniku. Poniższe reguły stanowią wzorzec *zwykłych przekształceń arytmetycznych*:

Po pierwsze, jeśli któryś z argumentów jest typu long double, to drugi jest przekształcany do long double.

W przeciwnym przypadku, jeśli któryś z argumentów jest typu double, to drugi jest przekształcany do double.

W przeciwnym przypadku, jeśli któryś z argumentów jest typu float, to drugi jest przekształcany do float.

W przeciwnym przypadku do obu argumentów stosuje się promocję typu całkowitego; jeśli teraz któryś z argumentów jest typu unsigned long int, to drugi jest przekształcany do unsigned long int.

W przeciwnym przypadku, jeśli jeden z argumentów jest typu long int, a drugi typu unsigned int, wynik zależy od tego, czy long int jest w stanie wyrazić wszystkie wartości typu unsigned int; jeśli tak, to argument typu unsigned int jest przekształcany do typu long int; jeśli nie – oba są przekształcane do typu unsigned long int.

W przeciwnym przypadku, jeśli jeden z argumentów jest typu long int, to drugi jest przekształcany do long int.

W przeciwnym przypadku, jeśli któryś z argumentów jest typu unsigned int, to drugi jest przekształcany do unsigned int.

W przeciwnym przypadku oba argumenty mają typ int.

> Zaszły tu dwie zmiany. Po pierwsze, arytmetykę zmiennopozycyjną można wykonywać w pojedynczej precyzji; w pierwszym wydaniu książki podkreślano, że całą arytmetykę zmiennopozycyjną realizuje się w podwójnej precyzji. Po drugie, krótsze typy całkowite bez znaku w kombinacjach z dłuższymi ze znakiem nie przenoszą właściwości braku znaku arytmetycznego na typ wyniku; uprzednio właściwość braku znaku była dominująca. Nowe zasady są nieco bardziej skomplikowane, ale trochę ograniczają niespodzianki wynikające ze spotkania się wartości bez znaku z wartościami ze znakiem. Nadal można otrzymać nieoczekiwane wyniki przy porównywaniu wielkości bez znaku z wielkościami ze znakiem o tym samym rozmiarze.

A6.6 Wskaźniki i wartości całkowite

Wartość całkowitą można dodać do lub odjąć od wskaźnika; jest ona wówczas przekształcana zgodnie z zasadami podanymi przy opisie operatora dodawania (p. A7.7).

Dwa wskaźniki do obiektów tego samego typu, należących do tej samej tablicy, można od siebie odjąć; w tym przypadku wynik jest przekształcany do wartości całkowitej zgodnie z zasadami podanymi przy opisie operatora odejmowania (p. A7.7).

Stałe wyrażenie całkowite o wartości zero, ewentualnie zrzutowane do typu void *, można przekształcić do wskaźnika dowolnego typu za pomocą rzutowania, przypisania lub porównania. Produkuje to pusty wskaźnik, równy innemu pustemu wskaźnikowi tego samego typu, ale różny od dowolnego wskaźnika do funkcji lub obiektu.

Dopuszcza się również inne przekształcenia wskaźników, ale są one zależne od maszyny. Takie przekształcenia muszą być realizowane za pomocą jawnych operatorów przekształcenia typów lub za pomocą rzutowania (p. A7.5 i A8.8).

Wskaźnik można przekształcić do typu całkowitego, wystarczająco obszernego, aby pomieścił wartość wskaźnika; wymagany rozmiar typu całkowitego zależy od implementacji. Od implementacji zależy również sposób takiego przekształcenia.

Obiekt typu całkowitego można jawnie przekształcić na wskaźnik. Przekształcenie to zawsze zachowa wartość wskaźnika, jeżeli tylko otrzymano go z przekształcenia dostatecznie obszernej wartości całkowitej, która przedtem powstała z przekształcenia tego wskaźnika; w pozostałych przypadkach zależy od implementacji.

Wskaźnik jednego typu może być przekształcony na wskaźnik innego typu. Wynikowy wskaźnik może powodować błąd adresowania, jeśli odnosi się do obiektu źle położonego w pamięci. Gwarantuje się, że przekształcenie wskaźnika pewnego typu na wskaźnik innego typu o mniejszych lub równych ograniczeniach dotyczących położenia w pamięci (i z powrotem) zachowa wartość wskaźnika wyjściowego. Znaczenie pojęcia „ograniczenie dotyczące położenia w pamięci" (ang. *alignment*) zależy od implementacji, ale obiekty typu char są najmniej ograniczone. Jak to będzie opisane w p. A6.8, dowolny wskaźnik można przekształcić do typu void * i z powrotem bez zmiany.

Wskaźnik może być przekształcony na inny wskaźnik o takim samym typie, ale z innym zestawem kwalifikatorów typu obiektów (p. A4.4, A8.2) wskazywanych przez ten wskaźnik. Jeśli dochodzą nowe kwalifikatory, to nowy wskaźnik jest równoważny ze starym, przy czym obowiązują dodatkowe ograniczenia wynikające z nowych kwalifikatorów. Jeśli kwalifikatorów ubywa, to operacje na wskazywanych obiektach nadal podlegają ograniczeniom wynikającym z kwalifikatorów podanych w oryginalnej deklaracji.

Na koniec, wskaźniki do funkcji można przekształcić na wskaźniki do funkcji innego typu. Wywołanie funkcji za pomocą przekształconego wskaźnika zależy od implementacji, natomiast wynik przekształcenia tego wskaźnika z powrotem do jego oryginalnego typu jest identyczny z oryginalnym wskaźnikiem.

A6.7 Typ void

Wartości (nie istniejącej) obiektu typu void nie można wykorzystać w żaden sposób ani nie można przekształcić, jawnie czy niejawnie, do innego typu różnego od void. Ponieważ wyrażenie typu void oznacza nie istniejącą wartość, takich wyrażeń można użyć jedynie w miejscach, gdzie wartość nie jest wymagana, na przykład jako instrukcję wyrażeniową (p. A9.2) lub jako lewy argument operatora przecinkowego (p. A7.18).

> Typ void nie występował w pierwszym wydaniu książki, lecz od dawna jest w powszechnym użyciu.

A6.8 Wskaźniki do typu void

Każdy wskaźnik do obiektu może być przekształcony do typu void * bez utraty informacji. Przekształcenie wyniku z powrotem do wskaźnika oryginalnego typu przywraca oryginalny wskaźnik. W przeciwieństwie do tych przekształceń wskaźników do

wskaźników, omówionych w p. A6.6, które wymagały jawnego rzutowania, wskaźnik do obiektu można wstawić do (lub otrzymać wartość od) wskaźnika typu void *, a także porównywać z takim wskaźnikiem.

> Ta interpretacja wskaźników typu void * jest nowa; poprzednio rolę wskaźników ogólnych spełniały wskaźniki typu char *. W przypisaniach i porównaniach ANSI C wyraźnie „błogosławi'' kojarzenie wskaźników do obiektów ze wskaźnikami typu void *, wymagając zarazem jawnego rzutowania w przypadku innych kombinacji wskaźników.

A7 Wyrażenia

Porządek priorytetów operatorów w wyrażeniach jest taki sam, jak porządek głównych podpunktów niniejszego punktu, poczynając od priorytetu najwyższego. Na przykład wyrażenia, o których mówi się jako o argumentach operatora + (p. A7.7), są wyrażeniami zdefiniowanymi w p. A7.1-A7.6. Opisane w poszczególnych podpunktach operatory mają równy priorytet. Tam określono również lewostronną lub prawostronną łączność* tych operatorów. Priorytety i łączność wszystkich operatorów są zestawione w gramatyce języka (p. A13).

Priorytety i łączność operatorów są w pełni określone; nie jest natomiast określona (poza kilkoma wyjątkami) kolejność obliczania wyrażeń, nawet jeśli podwyrażenia powodują efekty uboczne. Oznacza to, że jeśli definicja operatora nie gwarantuje obliczenia argumentów w szczególnej kolejności, to w implementacjach zezwala się na swobodę w kolejności obliczania argumentów, a nawet na mieszanie tych obliczeń. Każdy operator gromadzi jednak swoje argumenty zgodnie z rozbiorem składniowym wyrażenia, w którym występuje.

> Ta reguła anuluje dotychczasową swobodę w reorganizacji wyrażeń z operatorami, które są matematycznie przemienne i łączne, ale mogą nie być łączne obliczeniowo. Zmiana dotyczy tylko rachunku zmiennopozycyjnego na granicy dokładności i sytuacji, gdy jest możliwy nadmiar.

Sposób postępowania przy nadmiarze, błędach dzielenia i innych sytuacjach wyjątkowych podczas obliczania wyrażeń nie jest zdefiniowany w języku. Większość istniejących implementacji C ignoruje nadmiar przy obliczaniu wyrażeń i w przypisaniach dla wielkości całkowitych ze znakiem, ale to zachowanie się nie jest gwarantowane.

* Por. przypis na str. 82.

Implementacje różnią się reakcjami na dzielenie przez zero i na wszystkie błędy w rachunku zmiennopozycyjnym; te błędy są niekiedy obsługiwane przez niestandardowe funkcje biblioteczne.

A7.1 Generowanie wskaźników

Jeśli typem wyrażenia lub podwyrażenia jest „tablica elementów typu *T*", dla pewnego typu *T*, to wartością wyrażenia jest wskaźnik do pierwszego elementu tej tablicy, a typ wyrażenia zmienia się na „wskaźnik do typu *T*". Takie przekształcenie nie zachodzi, gdy wyrażenie jest argumentem jednoargumentowego operatora &, operatorów ++, −−, sizeof lub lewym argumentem operatorów przypisania lub . (kropka). Podobnie wyrażenie typu „funkcja zwracająca typ *T*", jeśli nie jest argumentem operatora &, przekształca się do typu „wskaźnik do funkcji zwracającej typ *T*".

A7.2 Wyrażenia proste

Wyrażeniami prostymi są: identyfikatory, stałe, napisy lub wyrażenia w nawiasach.

wyrażenie-proste:
    identyfikator
    stała
    napis
    (*wyrażenie*)

Identyfikator jest wyrażeniem prostym pod warunkiem, że został odpowiednio zadeklarowany (patrz p. A8). Deklaracja określa typ identyfikatora. Identyfikator jest l-wartością, jeżeli odnosi się do jakiegoś obiektu (p. A5) i jest typu arytmetycznego, strukturą, unią lub wskaźnikiem.

Stała jest wyrażeniem prostym. Jej typ zależy od jej postaci, tak jak to opisano w p. A2.5.

Napis jest wyrażeniem prostym. Jego początkowym typem jest „tablica znakowa" (dla rozszerzonych napisów – „tablica elementów typu wchar_t"), ale w następstwie reguł podanych w p. A7.1 ten typ zwykle zmienia się do typu „wskaźnik do char" (wchar_t), a wynikiem jest wskaźnik do pierwszego znaku napisu. (Wyjątkami są pewne inicjatory – patrz p. A8.7).

Wyrażenie w nawiasach jest wyrażeniem prostym. Jego typ i wartość są takie same, jak typ i wartość wyrażenia bez nawiasów. Nawiasy nie mają wpływu na to, czy wyrażenie jest l-wartością.

A7.3 Wyrażenia przyrostkowe

Operatory w wyrażeniach przyrostkowych są lewostronnie łączne.

wyrażenie-przyrostkowe:
wyrażenie-proste
wyrażenie-przyrostkowe [*wyrażenie*]
wyrażenie-przyrostkowe (*lista-argumentów*$_{opc}$)
wyrażenie-przyrostkowe . *identyfikator*
wyrażenie-przyrostkowe –> *identyfikator*
wyrażenie-przyrostkowe ++
wyrażenie-przyrostkowe – –

lista-argumentów:
wyrażenie-przypisania
lista-argumentów , *wyrażenie-przypisania*

A7.3.1 Odwołania do tablic

Wyrażenie przyrostkowe, po którym następuje wyrażenie ujęte w nawiasy prostokątne, jest wyrażeniem przyrostkowym oznaczającym indeksowane odwołanie do tablicy. Jedno z tych wyrażeń musi być typu „wskaźnik do typu *T*", dla pewnego typu *T*, a drugie typu całkowitego; typem całego wyrażenia indeksowego jest *T*. Wyrażenie E1[E2] jest identyczne (z definicji) z wyrażeniem *((E1)+(E2)). Dalsze wyjaśnienia na ten temat znajdują się w p. A8.6.2.

A7.3.2 Wywołania funkcji

Wywołanie funkcji składa się z wyrażenia przyrostkowego, zwanego oznacznikiem funkcji, po którym następuje ujęta w nawiasy okrągłe lista (być może pusta) wyrażeń przypisania (p. A7.17). Wyrażenia przypisania, oddzielone od siebie przecinkami, tworzą argumenty funkcji. Jeśli wyrażenie przyrostkowe jest identyfikatorem, dla którego w odpowiednim zasięgu nie ma deklaracji, to identyfikator ten jest domyślnie deklarowany tak, jakby w najbardziej wewnętrznym bloku zawierającym wywołanie funkcji występowała deklaracja

extern int *identyfikator*();

Wyrażenie przyrostkowe (być może po domyślnej deklaracji i generacji wskaźnika – p. A7.1) musi być typu „wskaźnik do funkcji zwracającej typ *T*", dla pewnego typu *T*, a wartość wywołania funkcji ma typ *T*.

> W pierwszym wydaniu książki typ wyrażenia przyrostkowego był ograniczony do „funkcji"; przy wywołaniu funkcji za pomocą wskaźnika należało jawnie stosować operator *. Standard ANSI „udziela błogosławieństwa" praktykom stosowanym przez niektóre kompilatory, zezwalając na tę samą składnię przy wywołaniu funkcji i wywołaniu funkcji za pomocą wskaźnika. Starsza składnia nadal może być stosowana.

Pojęcie *argument* oznacza wyrażenie przekazane przez wywołanie funkcji; pojęcie *parametr* oznacza wejściowy obiekt (lub jego identyfikator) otrzymany przez definicję funkcji lub opisany w deklaracji funkcji. To samo rozróżnienie formułuje się niekiedy stosując pojęcia, odpowiednio, „argument (parametr) aktualny" i „argument (parametr) formalny".

Podczas obsługi wywołania funkcji tworzy się kopie wszystkich argumentów; argumenty są przekazywane wyłącznie przez wartość. Funkcja może zmienić wartości swoich parametrów, które są kopiami argumentów, ale nie ma to wpływu na wartości faktycznych argumentów. Można natomiast przekazać funkcji wskaźnik, godząc się z tym, że funkcja może zmienić wartość wskazywanego obiektu.

Dopuszcza się dwa style deklaracji funkcji – stary i nowy. W nowym stylu typy parametrów są jawne i są częścią typu funkcji – taka deklaracja nazywa się prototypem funkcji. Deklaracja w starym stylu nie zawiera typów parametrów. Opis deklaracji funkcji znajduje się w p. A8.6.3 i A10.1.

Jeśli w zasięgu wywołania funkcji obowiązuje stary styl deklaracji funkcji, to stosuje się domyślną promocję typów argumentów wywołania: dla argumentów całkowitych stosuje się promocję typu całkowitego (p. A6.1), argumenty typu float przekształca się do double. Skutek wywołania funkcji nie jest zdefiniowany, jeśli liczba argumentów nie zgadza się z liczbą parametrów w definicji funkcji lub jeśli typ jakiegoś argumentu po zastosowaniu promocji nie zgadza się z typem odpowiedniego parametru. Zgodność typów zależy od tego, czy funkcja była zdefiniowana w nowym stylu czy w starym. Jeśli definicja funkcji jest w starym stylu, to porównanie zachodzi między promowanym typem argumentu wywołania a promowanym typem parametru. Jeśli definicja jest w nowym stylu, to promowany typ argumentu musi być taki sam, jak typ parametru bez promocji.

Jeśli w zasięgu wywołania funkcji obowiązuje nowy styl deklaracji funkcji, to argumenty są przekształcane (jak w przypisaniu) do typów odpowiednich parametrów prototypu funkcji. Liczba argumentów musi się równać liczbie jawnie podanych parametrów, chyba że listę parametrów w deklaracji kończy wielokropek (, ...). W tym przypadku liczba argumentów musi być nie mniejsza niż liczba parametrów; nadliczbowe argumenty (w stosunku do jawnie wyszczególnionych parametrów) podlegają domyślnej promocji typu, tak jak to zostało opisane w poprzednim akapicie. Jeśli definicja funkcji jest w starym stylu, to wszystkie parametry prototypu funkcji znajdującego się

w zasięgu wywołania muszą się zgadzać z odpowiednimi parametrami definicji funkcji (po zastosowaniu promocji typów do parametrów definicji).

> Te reguły są wyjątkowo skomplikowane, muszą się one bowiem stosować do mieszaniny funkcji w nowym i starym stylu. Jeśli to tylko możliwe, takiej mieszaniny należy się wystrzegać.

Kolejność obliczania argumentów nie jest zdefiniowana; należy pamiętać o tym, że poszczególne kompilatory postępują różnie. Pewne jest natomiast, że zanim funkcja rozpocznie działanie, argumenty i oznacznik funkcji są już całkowicie obliczone, włączając w to wszystkie efekty uboczne. Każdą funkcję można wywoływać rekurencyjnie.

A7.3.3 Odwołania do struktur

Wyrażenie przyrostkowe, po którym następuje kropka i identyfikator, jest wyrażeniem przyrostkowym. Pierwsze wyrażenie musi być strukturą lub unią, a identyfikator – nazwą składowej tej struktury lub unii. Wynikiem jest wskazana składowa, a typem – typ tej składowej. Wyrażenie to jest l-wartością, jeżeli pierwsze wyrażenie jest l-wartością, a składowa nie jest tablicą.

Wyrażenie przyrostkowe, po którym następuje strzałka (złożona ze znaków – i >) oraz identyfikator, jest wyrażeniem przyrostkowym. Pierwsze wyrażenie musi być wskaźnikiem do struktury lub unii, a identyfikator musi być nazwą składowej tej struktury lub unii. Wynikiem jest składowa struktury lub unii wskazanej przez wyrażenie wskaźnikowe, a jego typem – typ tej składowej; wynik jest l-wartością, jeśli typ ten nie jest typem tablicowym.

Zatem wyrażenie E1–>MOS jest tym samym, co (*E1).MOS. Struktury i unie są opisane w p. A8.3.

> Już w pierwszym wydaniu książki było regułą, że nazwa składowej w takim wyrażeniu musiała należeć do struktury lub unii określonej wyrażeniem przyrostkowym; zaznaczono jednak, że reguła ta nie była zbyt rygorystycznie przestrzegana. Nowe kompilatory, również ANSI, przestrzegają jej.

A7.3.4 Zwiększanie i zmniejszanie przyrostkowe

Wyrażenie przyrostkowe, po którym następuje operator ++ lub ––, jest wyrażeniem przyrostkowym. Wartością wyrażenia jest wartość argumentu. Po obliczeniu wartości argument jest zwiększany (++) lub zmniejszany (––) o jeden. Argument musi być l-wartością; dalsze wymagania w stosunku do argumentu oraz szczegóły operacji są

podane przy opisie operatorów addytywnych (p. A7.7) i operatorów przypisania (p. A7.17). Wynik nie jest l-wartością.

A7.4 Operatory jednoargumentowe

Operatory jednoargumentowe są prawostronnie łączne.

wyrażenie-jednoargumentowe:
wyrażenie-przyrostkowe
++ *wyrażenie-jednoargumentowe*
-- *wyrażenie-jednoargumentowe*
operator-jednoargumentowy wyrażenie-rzutowania
sizeof *wyrażenie-jednoargumentowe*
sizeof (*nazwa-typu*)

operator-jednoargumentowy: (jeden z)
& * + - ~ !

A7.4.1 Przedrostkowe operatory zwiększania i zmniejszania

Wyrażenie jednoargumentowe poprzedzone operatorem ++ lub -- jest wyrażeniem jednoargumentowym. Argument jest zwiększany (++) lub zmniejszany (--) o 1. Wartością wyrażenia jest wartość po zwiększeniu (zmniejszeniu). Argument musi być l-wartością; dalsze wymagania w stosunku do argumentu oraz szczegóły operacji są podane przy opisie operatorów addytywnych (p. A7.7) i operatorów przypisania (p. A7.17). Wynik nie jest l-wartością.

A7.4.2 Operator adresu

Jednoargumentowy operator & podaje adres swojego argumentu. Argument musi być l-wartością nie odnoszącą się ani do pola bitowego, ani do obiektu zadeklarowanego jako register; argument może również być funkcją. Wynikiem jest wskaźnik do obiektu lub funkcji wskazanych l-wartością. Jeśli typem argumentu jest *T*, to typem wyniku jest „wskaźnik do typu *T*".

A7.4.3 Operator adresowania pośredniego

Jednoargumentowy operator * oznacza adresowanie pośrednie i daje w wyniku obiekt lub funkcję wskazane przez jego argument. Wynik jest l-wartością, jeżeli argument jest wskaźnikiem do typu arytmetycznego, struktury, unii lub wskaźnika. Jeśli typem argumentu jest „wskaźnik do typu *T*", to typem wyniku jest *T*.

A7.4.4 Jednoargumentowy plus

Argument jednoargumentowego operatora + musi mieć typ arytmetyczny lub wskaźnikowy. Wynikiem jest wartość argumentu. Argument całkowity podlega promocji typu całkowitego. Typem wyniku jest promowany typ argumentu.

> Jednoargumentowy + jest nowy. Wprowadzono go dla symetrii z jednoargumentowym operatorem –.

A7.4.5 Jednoargumentowy minus

Argument jednoargumentowego operatora – musi mieć typ arytmetyczny. Wynikiem jest wartość argumentu ze zmienionym znakiem. Argument podlega promocji typu całkowitego. Zmiana znaku wielkości bez znaku polega na odjęciu promowanej wartości od największej wartości promowanego typu i dodaniu jeden; ujemne zero zawsze jednak równa się zero. Typem wyniku jest promowany typ argumentu.

A7.4.6 Operator dopełnienia jedynkowego

Argument operatora ~ musi mieć typ całkowity. Wynikiem jest dopełnienie jedynkowe wartości argumentu. Dokonuje się promocji typu całkowitego. Jeśli argument jest bez znaku, operacja polega na odjęciu jego wartości od największej wartości promowanego typu. Jeśli argument jest ze znakiem, operacja polega na przekształceniu promowanego argumentu do odpowiedniego typu bez znaku, zastosowaniu operatora ~ i ponownym przekształceniu do typu ze znakiem. Typem wyniku jest promowany typ argumentu.

A7.4.7 Operator negacji logicznej

Argument operatora ! musi mieć typ całkowity lub wskaźnikowy. Wynikiem jest 1, jeśli wartość argumentu równa się zero, a 0 w przeciwnym przypadku. Typem wyniku jest int.

A7.4.8 Operator sizeof

Operator sizeof podaje liczbę bajtów wymaganych do przechowania obiektu o typie wskazanym argumentem. Argument jest albo wyrażeniem (które nie jest obliczane), albo nazwą typu ujętą w nawiasy. Jeśli operator sizeof odnosi się do typu char, to

wynikiem jest 1; jeśli odnosi się do tablicy, to wynikiem jest całkowita liczba bajtów zajmowanych przez tablicę. Jeśli odnosi się do struktury lub unii, to wynikiem jest liczba bajtów zajmowanych przez ten obiekt z uwzględnieniem bajtów, którymi należałoby ewentualnie go uzupełnić, by mógł być elementem tablicy* – rozmiar tablicy o *n* elementach równa się *n* razy rozmiar jednego elementu. Operator nie może odnosić się do funkcji, niekompletnego typu lub pola bitowego. Wynikiem jest stała całkowita bez znaku. Typ wyniku zależy od implementacji; w standardowym nagłówku <stddef.h> (patrz dodatek B) typ ten zdefiniowano jako size_t.

A7.5 Rzutowanie

Poprzedzenie wyrażenia jednoargumentowego nazwą typu ujętą w nawiasy powoduje przekształcenie wartości tego wyrażenia do wskazanego typu.

wyrażenie-rzutowania:
wyrażenie-jednoargumentowe
(*nazwa-typu*) *wyrażenie-rzutowania*

Taka konstrukcja nazywa się *rzutem*. Nazwy typów są opisane w p. A8.8, a skutki przekształceń w p. A6. Wyrażenie rzutowania nie jest l-wartością.

A7.6 Operatory multyplikatywne

Operatory multyplikatywne *, / i % są lewostronnie łączne.

wyrażenie-multyplikatywne:
wyrażenie-rzutowania
wyrażenie-multyplikatywne * *wyrażenie-rzutowania*
wyrażenie-multyplikatywne / *wyrażenie-rzutowania*
wyrażenie-multyplikatywne % *wyrażenie-rzutowania*

Argumenty operatorów * i / muszą mieć typ arytmetyczny; argumenty operatora % muszą być całkowite. Argumenty podlegają zwykłym przekształceniom arytmetycznym, które również określają typ wyniku.

Dwuargumentowy operator * oznacza mnożenie.

* Chodzi o to, że wszystkie elementy dowolnej tablicy muszą być prawidłowo położone w pamięci. – *Przyp. tłum.*

Dwuargumentowy operator / daje w wyniku iloraz, a operator % – resztę z dzielenia pierwszego argumentu przez drugi. Wynik nie jest zdefiniowany, jeśli drugi argument równa się zero. W pozostałych przypadkach zawsze jest prawdą, że (a/b)*b + a%b równa się a. Jeśli oba argumenty są nieujemne, to reszta jest nieujemna i jest mniejsza niż dzielnik; w przeciwnym przypadku gwarantuje się tylko tyle, że wartość bezwzględna reszty jest mniejsza niż wartość bezwzględna dzielnika.

A7.7 Operatory addytywne

Operatory addytywne + i – są lewostronnie łączne. Jeśli oba argumenty mają typy arytmetyczne, to podlegają zwykłym przekształceniom arytmetycznym. Możliwe są również inne kombinacje typów argumentów tych operatorów.

wyrażenie-addytywne:
wyrażenie-multyplikatywne
wyrażenie-addytywne + *wyrażenie-multyplikatywne*
wyrażenie-addytywne – *wyrażenie-multyplikatywne*

Operator + daje w wyniku sumę swoich argumentów. Do wskaźnika do elementu dowolnej tablicy można dodać wartość o dowolnym typie całkowitym. Tę wartość przekształca się na przesunięcie adresowe, mnożąc ją przez rozmiar wskazywanego obiektu. Wynik jest wskaźnikiem tego samego typu, co wskaźnik początkowy, i wskazuje na inny element tej samej tablicy odpowiednio przesunięty względem elementu początkowego. Zatem jeśli P jest wskaźnikiem do elementu pewnej tablicy, to wyrażenie P+1 jest wskaźnikiem do następnego elementu tej tablicy. Wynik operacji nie jest zdefiniowany, jeśli otrzymany wskaźnik pokazuje poza obszar tablicy, za wyjątkiem pierwszej pozycji za końcem tablicy.

> Dopuszczenie wskaźników pokazujących tuż za koniec tablicy jest nowe. Sankcjonuje to idiom powszechnie stosowany przy przeglądaniu elementów tablic.

Operator – daje w wyniku różnicę swoich argumentów. Od wskaźnika można odjąć wartość dowolnego typu całkowitego; obowiązują wtedy te same przekształcenia i ograniczenia, jak w przypadku dodawania.

Wynikiem odjęcia od siebie dwóch wskaźników tego samego typu jest wartość całkowita ze znakiem, odpowiadająca odstępowi między wskazywanymi obiektami; wskaźniki wskazujące na kolejne obiekty różnią się o 1. Typ wyniku zależy od implementacji; w standardowym nagłówku <stddef.h> jest on zdefiowany jako ptrdiff_t. Wynik nie jest zdefiniowany, jeśli wskaźniki nie wskazują na elementy tej samej tablicy; jeśli jednak P wskazuje na ostatni element tablicy, to (P+1)–P równa się 1.

A7.8 Operatory przesunięcia

Operatory przesunięcia << i >> są lewostronnie łączne. Ich argumenty muszą być całkowite i podlegają promocji typu całkowitego. Typem wyniku jest promowany typ lewego argumentu. Wynik nie jest zdefiniowany, jeśli prawy argument jest ujemny, a także jeśli jest większy lub równy liczbie bitów w typie lewego argumentu.

wyrażenie-przesunięcia:
wyrażenie-addytywne
wyrażenie-przesunięcia << *wyrażenie-addytywne*
wyrażenie-przesunięcia >> *wyrażenie-addytywne*

Wartością E1<<E2 jest E1 (traktowane jako wzorzec bitowy) przesunięte w lewo o E2 bitów; jeśli nie ma nadmiaru, jest to równoważne pomnożeniu przez 2^{E2}. Wartością E1>>E2 jest E1 przesunięte w prawo o E2 bitów. Jeżeli E1 jest bez znaku lub jest nieujemne, to przesunięcie w prawo jest równoważne z dzieleniem przez 2^{E2}; w pozostałych przypadkach wynik zależy od implementacji.

A7.9 Operatory relacji

Operatory relacji są lewostronnie łączne, ale fakt ten nie jest specjalnie użyteczny: wyrażenie a<b<c jest gramatycznie równoważne z wyrażeniem (a<b)<c, w którym a<b ma wartość 0 lub 1.

wyrażenie-relacyjne:
wyrażenie-przesunięcia
wyrażenie-relacyjne < *wyrażenie-przesunięcia*
wyrażenie-relacyjne > *wyrażenie-przesunięcia*
wyrażenie-relacyjne <= *wyrażenie-przesunięcia*
wyrażenie-relacyjne >= *wyrażenie-przesunięcia*

Wszystkie operatory: < (mniejsze niż), > (większe niż), <= (mniejsze lub równe) i >= (większe lub równe) dają w wyniku 0, jeżeli wskazana relacja jest fałszywa, bądź 1 – jeżeli prawdziwa. Typem wyniku jest int. Argumenty arytmetyczne podlegają zwykłym przekształceniom arytmetycznym. Można porównywać dwa wskaźniki do obiektów tego samego typu (nie licząc kwalifikatorów) – wynik zależy od względnego położenia wskazywanych obiektów w przestrzeni adresowej. Porównywanie wskaźników jest określone tylko dla wskaźników pokazujących fragmenty tego samego obiektu: jeśli dwa wskaźniki pokazują na ten sam prosty obiekt, to są sobie równe; jeśli pokazują na składowe tej samej struktury, to wskaźnik pokazujący składową później zadeklarowaną w strukturze jest uważany za większy; jeśli pokazują

na składowe tej samej unii, to są równe; jeśli wskazują elementy tej samej tablicy, to porównanie wskaźników jest równoważne z porównaniem odpowiednich indeksów tablicy. Jeśli P wskazuje ostatni element tablicy, to P+1 jest większy niż P, chociaż P+1 pokazuje poza tablicę. W pozostałych przypadkach porównywanie wskaźników nie jest zdefiniowane.

> Powyższe reguły liberalizują nieco ograniczenia podane w pierwszym wydaniu książki – dopuszcza się porównywanie wskaźników wskazujących na różne składowe struktur i unii. Reguły te legalizują także porównywanie ze wskaźnikiem wskazującym tuż za koniec tablicy.

A7.10 Operatory przyrównania

wyrażenie-przyrównania:
wyrażenie-relacyjne
wyrażenie-przyrównania == *wyrażenie-relacyjne*
wyrażenie-przyrównania != *wyrażenie-relacyjne*

Operatory == (równe) i != (nie równe) są podobne do operatorów relacji z tym, że mają niższy priorytet. (Zatem a<b == c<d daje w wyniku 1, gdy obie relacje a<b oraz c<d mają tę samą wartość logiczną.)

Do operatorów przyrównania odnoszą się te same reguły, co do operatorów relacji, a ponadto wskaźnik można przyrównać do stałej całkowitej o wartości 0 albo do wskaźnika do typu void. Patrz p. A6.6.

A7.11 Bitowy operator koniunkcji (AND)

wyrażenie-AND:
wyrażenie-przyrównania
wyrażenie-AND & *wyrażenie-przyrównania*

Wynikiem tej operacji jest bitowa koniunkcja argumentów; argumenty podlegają zwykłym przekształceniom arytmetycznym. Operator odnosi się tylko do argumentów całkowitych.

A7.12 Bitowy operator różnicy symetrycznej (XOR)

wyrażenie-XOR:
wyrażenie-AND
wyrażenie-XOR ^ *wyrażenie-AND*

Wynikiem tej operacji jest bitowa różnica symetryczna argumentów; argumenty podlegają zwykłym przekształceniom arytmetycznym. Operator odnosi się tylko do argumentów całkowitych.

A7.13 Bitowy operator alternatywy (OR)

wyrażenie-OR:
wyrażenie-XOR
wyrażenie-OR | *wyrażenie-XOR*

Wynikiem tej operacji jest bitowa alternatywa argumentów; argumenty podlegają zwykłym przekształceniom arytmetycznym. Operator odnosi się tylko do argumentów całkowitych.

A7.14 Operator iloczynu logicznego (logiczne AND)

logiczne-wyrażenie-AND:
wyrażenie-OR
logiczne-wyrażenie-AND && *wyrażenie-OR*

Operator && jest lewostronnie łączny. Wynik jest równy 1, jeżeli oba argumenty są różne od zera; w pozostałych przypadkach jest równy 0. W odróżnieniu od operatora & operator && gwarantuje obliczanie argumentów od lewej strony do prawej: najpierw oblicza się lewy argument, włączając w to wszystkie efekty uboczne, i jeśli jego wartość jest równa zero, to wartością wyrażenia jest 0. W przeciwnym przypadku oblicza się drugi argument i jeśli jego wartość jest równa zero, to wartością wyrażenia jest 0; w przeciwnym przypadku wartością jest 1.

Argumenty nie muszą być tego samego typu, ale muszą mieć typy arytmetyczne lub być wskaźnikami. Wynik jest typu int.

A7.15 Operator sumy logicznej (logiczne OR)

logiczne-wyrażenie-OR:
logiczne-wyrażenie-AND
logiczne-wyrażenie-OR || *logiczne-wyrażenie-AND*

Operator || jest lewostronnie łączny. Wynik jest równy 1, jeżeli którykolwiek z argumentów jest różny od zera; w przeciwnym przypadku jest równy 0. W odróżnieniu od operatora | operator || gwarantuje obliczanie argumentów od lewej strony do prawej: najpierw oblicza się lewy argument, włączając w to wszystkie efekty uboczne, i jeśli jego wartość jest różna od zera, to wartością wyrażenia jest 1. W przeciwnym przy-

padku oblicza się drugi argument i jeśli jego wartość jest różna od zera, to wartością wyrażenia jest 1; w przeciwnym przypadku wartością jest 0.

Argumenty nie muszą być tego samego typu, ale muszą mieć typy arytmetyczne lub być wskaźnikami. Wynik jest typu int.

A7.16 Operator warunkowy

wyrażenie-warunkowe:
logiczne-wyrażenie-OR
logiczne-wyrażenie-OR ? *wyrażenie* : *wyrażenie-warunkowe*

Najpierw oblicza się pierwsze wyrażenie, włączając w to wszystkie efekty uboczne. Jeśli wartość tego wyrażenia jest różna od zera, to wynikiem jest wartość drugiego wyrażenia, a w przeciwnym przypadku trzeciego. Spośród wyrażeń drugiego i trzeciego oblicza się tylko jedno. Jeśli wyrażenia drugie i trzecie mają typy arytmetyczne, to podlegają zwykłym przekształceniom arytmetycznym do wspólnego typu i taki jest typ wyniku. Jeśli oba są typu void albo są strukturami lub uniami tego samego typu, albo są wskaźnikami do tego samego typu, to typem wyniku jest wspólny typ tych wyrażeń. Jeśli jedno jest wskaźnikiem, a drugie stałą 0, to 0 jest przekształcane do typu wskaźnika i taki jest typ wyniku. Jeśli jedno jest wskaźnikiem do void, a drugie innym wskaźnikiem, to ten inny wskaźnik jest przekształcany na wskaźnik do void i taki jest typ wyniku.

Przy rozpatrywaniu typów wskaźników kwalifikatory typu (p. A8.2) obiektów wskazywanych przez te wskaźniki nie odgrywają roli, ale typ wyniku dziedziczy kwalifikatory z obu części wyrażenia warunkowego.

A7.17 Wyrażenia przypisania

Istnieje kilka operatorów przypisania. Wszystkie są prawostronnie łączne.

wyrażenie-przypisania:
wyrażenie-warunkowe
wyrażenie-jednoargumentowe operator-przypisania wyrażenie-przypisania

operator-przypisania: (jeden z)
= *= /= %= += -= <<= >>= &= ^= |=

Dla wszystkich operatorów przypisania lewy argument musi być l-wartością i ta l-wartość musi być modyfikowalna: nie może być tablicą, nie może mieć niekompletnego typu i nie może być funkcją. Jej typ nie może być też kwalifikowany słowem kluczowym const; jeśli l-wartość jest strukturą lub unią, to żadna z jej składowych i,

rekurencyjnie, podskładowych nie może być kwalifikowana słowem const. Typem wyrażenia przypisania jest typ lewego argumentu, a wynikiem – wartość wstawiona do tego argumentu po wykonaniu przypisania.

Przy prostym przypisaniu z operatorem = wartość wyrażenia zastępuje poprzednią wartość obiektu wskazanego przez l-wartość. Musi być spełniony jeden z poniższych warunków: oba argumenty są arytmetyczne i prawy jest przekształcany do typu lewego; oba są strukturami lub uniami tego samego typu; jeden z argumentów jest wskaźnikiem, a drugi wskaźnikiem do typu void; lewy argument jest wskaźnikiem, a drugi wyrażeniem stałym o wartości 0; oba argumenty są wskaźnikami do funkcji lub do obiektów tego samego typu (typy obu wskaźników mogą się różnić jedynie brakiem słów kluczowych const i volatile w deklaracji prawego argumentu).

Wyrażenie o postaci E1 *op*= E2 jest równoważne z wyrażeniem E1 = E1 *op* (E2), z tym tylko, że wyrażenie E1 jest obliczane raz.

A7.18 Operator przecinkowy

wyrażenie:
wyrażenie-przypisania
wyrażenie , wyrażenie-przypisania

Parę wyrażeń oddzielonych przecinkami oblicza się od lewej strony do prawej, przy czym wartość lewego wyrażenia przepada. Typ i wartość wyniku są takie same, jak typ i wartość prawego argumentu. Zanim rozpocznie się obliczanie prawego argumentu, wszystkie efekty uboczne obliczenia lewego argumentu są już zakończone. W kontekście, w którym przecinek ma znaczenie specjalne, na przykład na liście argumentów funkcji (p. A7.3.2) lub liście inicjatorów (p. A8.7), wymaganą jednostką składniową jest wyrażenie przypisania, zatem operator przecinkowy może wystąpić wyłącznie w wyrażeniu ujętym w nawiasy. Na przykład w wywołaniu funkcji

```
f(a, (t=3, t+2), c)
```

występują trzy argumenty; drugi argument ma wartość 5.

A7.19 Wyrażenia stałe

Wyrażenie stałe jest składniowo równoważne z wyrażeniem o ograniczonym repertuarze operatorów.

wyrażenie-stałe:
wyrażenie-warunkowe

Wyrażenia, które sprowadzają się do stałych, są wymagane w kilku kontekstach: po słowie case, jako rozmiar tablicy, jako długość pola bitowego, jako wartość stałej wyliczenia, w inicjatorach i w niektórych wyrażeniach dla preprocesora.

Wyrażenia stałe nie mogą zawierać operatorów przypisania, zwiększania i zmniejszania, wywołań funkcji i operatorów przecinkowych, chyba że występują w argumencie operatora sizeof. Jeśli jest wymagane stałe wyrażenie całkowite, to jego argumenty muszą być stałymi: całkowitymi, wyliczeń, znakowymi lub zmiennopozycyjnymi; w rzutowaniach muszą wystąpić typy całkowite, a każdą stałą zmiennopozycyjną należy zrzutować do liczby całkowitej. Siłą rzeczy są wyeliminowane tablice, adresowanie pośrednie, pobieranie adresu i odwołania do składowych struktur. (Argumenty operatora sizeof mogą być jednak dowolne.)

Więcej swobody dopuszcza się w wyrażeniach stałych dla inicjatorów: argumenty mogą być stałymi dowolnego typu; jednoargumentowy operator adresu & może być stosowany do obiektów zewnętrznych lub statycznych, jak również do zewnętrznych lub statycznych tablic indeksowanych wyrażeniem stałym. Ten operator może być również stosowany niejawnie, przez użycie funkcji lub nieindeksowanych tablic. Inicjatory muszą się sprowadzać albo do wartości stałej, albo do adresu uprzednio zadeklarowanego obiektu zewnętrznego lub statycznego (plus/minus stała).

Mniej swobody jest w wyrażeniach stałych po #if: nie dopuszcza się wyrażeń z operatorem sizeof, stałych wyliczeń i rzutowania. Patrz p. A12.5.

A8 Deklaracje

Deklaracje określają sposób interpretacji identyfikatorów, nie zawsze jednak rezerwują pamięć związaną z identyfikatorami. Deklaracje, które rezerwują pamięć, nazywają się *definicjami*. Deklaracje mają postać:

deklaracja:
    specyfikatory-deklaracji inicjowana-lista-deklaratorów$_{opc}$;

Deklarowane identyfikatory umieszcza się na inicjowanej liście deklaratorów; specyfikatory deklaracji składają się z ciągu specyfikatorów typu i klasy pamięci:

specyfikatory-deklaracji:
    specyfikator-klasy-pamięci specyfikatory-deklaracji$_{opc}$
    specyfikator-typu specyfikatory-deklaracji$_{opc}$
    kwalifikator-typu specyfikatory-deklaracji$_{opc}$

inicjowana-lista-deklaratorów:
inicjowany-deklarator
inicjowana-lista-deklaratorów , inicjowany-deklarator

inicjowany-deklarator:
deklarator
deklarator = inicjator

Deklaratory zostaną omówione później (p. A8.5); deklarator zawiera deklarowaną nazwę. Deklaracja musi zawierać co najmniej jeden deklarator albo jej specyfikator typu musi deklarować etykietkę struktury, etykietkę unii lub składowe wyliczenia; pusta deklaracja jest zabroniona.

A8.1 Specyfikatory klasy pamięci

Specyfikatorami klasy pamięci są:

specyfikator-klasy-pamięci:
auto
register
static
extern
typedef

Znaczenie klas pamięci omówiono w p. A4.

Specyfikatory auto i register nadają deklarowanemu obiektowi automatyczną klasę pamięci i mogą być stosowane wyłącznie wewnątrz funkcji. Takie deklaracje są jednocześnie definicjami i powodują zarezerwowanie pamięci. Deklaracja register jest równoważna z deklaracją auto, ale wskazuje, że deklarowany obiekt będzie często używany. W rejestrach maszyny mieści się tylko kilka obiektów i tylko niektóre ich typy są dozwolone – ograniczenia zależą od implementacji. Mimo to do obiektów zadeklarowanych jako register nie wolno stosować jednoargumentowego operatora adresu & – ani jawnie, ani niejawnie.

> Zasada zabraniająca obliczania adresu obiektu zadeklarowanego jako register, lecz faktycznie występującego jako auto, jest nowa.

Specyfikator static nadaje deklarowanemu obiektowi statyczną klasę pamięci i może być stosowany zarówno wewnątrz, jak i na zewnątrz funkcji. Wewnątrz funkcji ten specyfikator powoduje zarezerwowanie pamięci, oznacza więc definicję. W punkcie A11.2 omówiono znaczenie tego specyfikatora na zewnątrz funkcji.

Wewnątrz funkcji deklaracje ze specyfikatorem extern wskazują, że pamięć dla deklarowanych obiektów będzie zarezerwowana gdzie indziej. W punkcie A11.2 omówiono znaczenie tego specyfikatora na zewnątrz funkcji.

Specyfikator typedef nie rezerwuje pamięci. Zalicza się go do specyfikatorów klasy pamięci wyłącznie dla składniowej wygody; będzie omówiony w p. A8.9.

W deklaracji może wystąpić co najwyżej jeden specyfikator klasy pamięci. Jeśli nie podano żadnego, to obowiązują następujące zasady: dla obiektów zadeklarowanych wewnątrz funkcji przyjmuje się klasę auto; dla funkcji zadeklarowanych wewnątrz funkcji przyjmuje się extern; obiekty i funkcje zadeklarowane na zewnątrz funkcji uważa się za statyczne z łącznością zewnętrzną. Patrz p. A10-A11.

A8.2 Specyfikatory typu

Specyfikatorami typu są:

specyfikator-typu:
void
char
short
int
long
float
double
signed
unsigned
specyfikator-struktury-unii
specyfikator-wyliczenia
nazwa-typedef

Wraz ze specyfikatorem int można użyć jednego ze specyfikatorów long lub short; znaczenie jest takie samo, jakby słowo int było pominięte. Słowo long można podać razem z double. Co najwyżej jeden ze specyfikatorów signed lub unsigned może wystąpić wraz ze słowem int bądź z jego odmianą long lub short, bądź z char. Każdy z nich może wystąpić samodzielnie – wtedy przyjmuje się int. Specyfikator signed przydaje się do wymuszenia znaku arytmetycznego dla obiektów typu char; dla pozostałych typów całkowitych jest zbędny, acz dozwolony.

Oprócz powyższych kombinacji w deklaracji może wystąpić co najwyżej jeden specyfikator typu. Jeśli pominięto specyfikator typu, to przyjmuje się int.

Kwalifikatory typu wskazują specjalne właściwości deklarowanych obiektów.

kwalifikator-typu:
const
volatile

Kwalifikatory mogą wystąpić z każdym specyfikatorem typu. Obiekt const (stały) może być inicjowany, ale nie można mu później przypisać wartości. Dla obiektów volatile (ulotny) nie ma interpretacji niezależnej od implementacji.

Właściwości const i volatile są w ANSI C nowe. Kwalifikator const służy do wskazania obiektów, które można umieścić w pamięci chronionej przed zapisem i, być może, zwiększyć podatność programu na optymalizację. Kwalifikator volatile służy do wyłączenia deklarowanego obiektu z ewentualnej optymalizacji. Na przykład w maszynach z mapowaną pamięcią wejścia-wyjścia wskaźnik do rejestru urządzenia mógłby być zadeklarowany jako wskaźnik do volatile, aby zapobiec usunięciu przez kompilator pozornie zbędnych odwołań do obiektu poprzez ten wskaźnik*. Kompilator może ignorować specyfikatory const, ale powinien sygnalizować jawne próby zmiany wartości obiektów o takim specyfikatorze.

A8.3 Deklaracje struktur i unii

Struktura jest obiektem składającym się z ciągu nazwanych składowych o różnych typach. Unia jest obiektem, który w danej chwili zawiera jedną ze swoich kilku składowych o różnych typach. Specyfikatory struktur i unii mają taką samą postać.

specyfikator-struktury-unii:
struktura-unia identyfikator$_{opc}$ { *lista-deklaracji-składowych* }
struktura-unia identyfikator

struktura-unia:
struct
union

Lista deklaracji struktury jest ciągiem deklaracji składowych struktury lub unii.

* Chodzi o to, że rejestry urządzeń zewnętrznych zmieniają wartości na skutek zdarzeń zewnętrznych, w minimalnym stopniu kontrolowanych przez program, i w związku z tym wymykają się spod ogólnie stosowanych algorytmów optymalizacyjnych. – *Przyp. tłum.*

lista-deklaracji-składowych:
deklaracja-składowych
lista-deklaracji-składowych deklaracja-składowych

deklaracja-składowych:
lista-spec-kwal lista-deklaratorów-składowych ;

lista-spec-kwal:
specyfikator-typu lista-spec-kwal$_{opc}$
kwalifikator-typu lista-spec-kwal$_{opc}$

lista-deklaratorów-składowych:
deklarator-składowej
lista-deklaratorów-składowych , *deklarator-składowej*

Zazwyczaj *deklarator-składowej* jest po prostu deklaratorem elementu struktury lub unii. Składową struktury może też być grupa bitów o podanej długości. Taką składową nazywa się *polem bitowym* lub po prostu *polem*; jej długość jest oddzielona od deklaratora nazwy pola dwukropkiem.

deklarator-składowej:
deklarator
deklarator$_{opc}$: *wyrażenie-stałe*

Specyfikator typu o postaci:

struktura-unia identyfikator { *lista-deklaracji-składowych* }

deklaruje identyfikator jako *etykietkę* struktury lub unii, których budowę opisuje lista. Późniejsze deklaracje na tym samym poziomie lub w zagnieżdżonych blokach mogą odnosić się do tego samego typu przez podanie samej etykietki, bez całej listy deklaracji składowych:

struktura-unia identyfikator

Wystąpienie takiego specyfikatora bez listy, ale z etykietką, dla której brak deklaracji, oznacza *typ niekompletny*. Obiekty z niekompletnym typem struktury lub unii mogą wystąpić w kontekstach, w których ich rozmiar nie ma znaczenia, na przykład w deklaracjach (nie definicjach) przy określaniu wskaźnika lub tworzeniu typu (typedef), ale nigdzie indziej. Ten typ stanie się kompletny po późniejszym napotkaniu specyfikatora z tą samą etykietką, zawierającego listę deklaracji. Nawet w specyfikatorach z listą deklarowany typ struktury lub unii jest niekompletny wewnątrz listy – staje się on kompletny dopiero po napotkaniu klamry } zamykającej specyfikator.

Struktura nie może zawierać składowej o typie niekompletnym. Nie można zatem zadeklarować struktury lub unii zawierającej własne wcielenie. Poza nadaniem nazwy dla typu struktury lub unii, etykietki umożliwiają jednak definiowanie struktur odwo-

łujących się do samych siebie: struktura lub unia może zawierać wskaźnik do swojego wcielenia, można bowiem deklarować wskaźniki do typów niekompletnych.

Bardzo specjalna reguła odnosi się do deklaracji mającej postać:

struktura-unia identyfikator ;

która deklaruje strukturę lub unię, ale nie zawiera listy deklaracji i deklaratorów. Taka deklaracja w danym zasięgu tworzy z identyfikatora etykietkę nowej struktury lub unii (o typie niekompletnym), nawet jeśli ten identyfikator był już w szerszym zasięgu zadeklarowany jako etykietka struktury lub unii (p. A11.1).

> Ta dziwna reguła jest w ANSI C całkiem nowa. Ma ona umożliwić deklarowanie w wewnętrznym zasięgu wzajemnie odwołujących się do siebie struktur, których etykietki mogły być już zadeklarowane w szerszym zasięgu.

Specyfikator struktury lub unii z listą, ale bez etykietki, tworzy typ unikalny; można się do niego bezpośrednio odwołać tylko w deklaracji, której jest częścią.

Nazwy składowych i etykietek nie kolidują ze sobą, a także z innymi zwykłymi zmiennymi. Ta sama nazwa składowej nie może wystąpić dwa razy w tej samej strukturze lub unii, ale może być użyta w innych strukturach i uniach.

> W pierwszym wydaniu tej książki nazwy składowych struktur i unii nie były związane z miejscem ich pochodzenia. Jednak to powiązanie było powszechnie stosowane w kompilatorach na długo przed powstaniem ANSI C.

Składowa struktury lub unii, ale nie pole bitowe, może być obiektem dowolnego typu. Pole (które nie musi mieć deklaratora, może więc nie mieć nazwy) ma typ int, unsigned int lub signed int i jest traktowane jak obiekt całkowity o długości wskazanej w bitach; od implementacji zależy, czy pola typu int są traktowane jak obiekty ze znakiem. Przylegające do siebie pola bitowe struktury pakuje się w zależne od implementacji porcje pamięci w sposób zależny od implementacji. Jeśli kolejne pole już się nie mieści w częściowo wypełnionej porcji pamięci, to może być rozdzielone między porcje lub przeniesione do następnej porcji. Pole bez nazwy o rozmiarze 0 wymusza takie przeniesienie, zatem następne pole zostanie umieszczone na granicy następnej porcji.

> Standard ANSI jeszcze bardziej uzależnia pola bitowe od implementacji, niż reguły podane w pierwszym wydaniu książki. Rozsądnie jest przyjąć zasady językowe dotyczące rozmieszczania pól bitowych jako bezwarunkowo „zależne od implementacji". Struktury z polami bitowymi mogą być stosowane jako przenośny sposób na zmniejszenie rozmiaru pamięci wymaganej dla struktury (prawdopodobnie kosztem zwiększenia rozmiaru kodu i czasu potrzebnego na obsługę pól) lub jako nieprzenośny sposób na opisanie położenia w pamięci na poziomie bitów. W drugim przypadku wymaga się znajomości zasad obowiązujących w danej implementacji.

Adresy składowych struktury wzrastają zgodnie z kolejnością deklaracji tych składowych. Składową struktury, ale nie pole, umieszcza się w pamięci zgodnie z wymaganiami jej typu; zatem w strukturze mogą wystąpić dziury bez nazwy. Jeśli wskaźnik do struktury jest zrzutowany na wskaźnik do typu jej pierwszej składowej, to wynikowy wskaźnik odnosi się do pierwszej składowej.

Unię można rozumieć jako strukturę, której wszystkie składowe są ulokowane z przesunięciem zerowym względem jej początku i której rozmiar jest dostatecznie duży, aby pomieścić każdą z jej składowych. Jeśli wskaźnik do unii jest zrzutowany do wskaźnika do typu jej pierwszej składowej, to wynikowy wskaźnik odnosi się do tej składowej.

Prostym przykładem deklaracji struktury jest

```
struct tnode {
    char tword[20];
    int count;
    struct tnode *left;
    struct tnode *right;
};
```

Struktura ta zawiera 20-znakową tablicę, wartość całkowitą oraz dwa wskaźniki do podobnych struktur. Przy takiej deklaracji wiersz

```
struct tnode s, *sp;
```

deklaruje s jako strukturę podanego rodzaju oraz sp jako wskaźnik do takiej struktury. Zatem wyrażenie

```
sp->count
```

odnosi się do składowej count w strukturze wskazywanej przez sp; wyrażenie

```
s.left
```

odnosi się do wskaźnika do lewego poddrzewa struktury s, a wyrażenie

```
s.right->tword[0]
```

reprezentuje pierwszy znak składowej tword w prawym poddrzewie s.

Zasadniczo nie można odwoływać się do wartości składowej unii, jeżeli ta wartość nie była wstawiona do unii za pomocą tej samej składowej. Następująca zasada ułatwia jednak korzystanie z unii: jeśli unia składa się z kilku struktur, w których występuje wspólny początkowy ciąg składowych, i w danej chwili unia zawiera jedną z tych struktur, to do początkowej wspólnej części można się odwołać za pomocą dowolnej z tych struktur. Na przykład taki fragment programu jest poprawny:

```
union {
    struct {
        int type;
    } n;
    struct {
        int type;
        int intnode;
    } ni;
    struct {
        int type;
        float floatnode;
    } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
    ... sin(u.nf.floatnode) ...
```

A8.4 Wyliczenia

Wyliczenia są szczególnymi typami o wartościach ze zbioru nazwanych stałych – tzw. wyliczników. Postać specyfikatora wyliczenia zapożyczono od struktur i unii.

specyfikator-wyliczenia:
    enum *identyfikator*$_{opc}$ { *lista-wyliczników* }
    enum *identyfikator*

lista-wyliczników:
    wylicznik
    lista-wyliczników , *wylicznik*

wylicznik:
    identyfikator
    identyfikator = *wyrażenie-stałe*

Identyfikatory na liście wyliczników są deklarowane jako stałe typu int i mogą wystąpić wszędzie tam, gdzie są wymagane stałe*. Jeśli żaden z wyliczników nie występ-

* Z wyjątkiem wyrażeń w #if. – *Przyp. tłum.*

puje z operatorem =, to wartości tych stałych rozpoczynają się od 0 i wzrastają o 1 wraz z kolejnością deklaracji czytanych od lewej strony do prawej. Deklaracja z operatorem = nadaje odpowiedniemu identyfikatorowi wskazaną wartość; wartości kolejnych identyfikatorów wzrastają począwszy od podanej wartości.

W tym samym zasięgu nazwy wyliczników muszą być unikalne i muszą być różne od nazw zwykłych zmiennych, natomiast ich wartości nie muszą być różne.

Rola identyfikatora w specyfikatorze wyliczenia jest taka sama, jak rola etykietki struktury w specyfikatorze struktury – jest nazwą danego wyliczenia. Zasady odnoszące się do specyfikatora wyliczenia z etykietką lub bez i z listą są takie same, jak w przypadku specyfikatorów struktur i unii z tym jednak wyjątkiem, że nie istnieją niekompletne typy wyliczenia – etykietka specyfikatora wyliczenia bez listy wyliczników musi odnosić się do występującego w jej zasięgu specyfikatora z listą.

> Wyliczenia nie występowały w pierwszym wydaniu książki, ale od paru lat są częścią języka C.

A8.5 Deklaratory

Składnia deklaratorów jest następująca:

deklarator:
wskaźnik$_{opc}$ *bezpośredni-deklarator*

bezpośredni-deklarator:
identyfikator
(*deklarator*)
bezpośredni-deklarator [*wyrażenie-stałe*$_{opc}$]
bezpośredni-deklarator (*lista-typów-parametrów*)
bezpośredni-deklarator (*lista-identyfikatorów*$_{opc}$)

wskaźnik:
* *lista-kwalifikatorów-typu*$_{opc}$
* *lista-kwalifikatorów-typu*$_{opc}$ *wskaźnik*

lista-kwalifikatorów-typu:
kwalifikator-typu
lista-kwalifikatorów-typu kwalifikator-typu

Postać deklaratorów przypomina postać wyrażeń z adresowaniem pośrednim, wywołaniem funkcji lub odwołaniem do tablicy; łączność jest taka sama.

A8.6 Znaczenie deklaratorów

Lista deklaratorów pojawia się po ciągu specyfikatorów typu i klasy pamięci. Każdy deklarator zawiera jeden główny identyfikator – ten, który występuje w pierwszej alternatywie produkcji *bezpośredni-deklarator*. Specyfikatory klasy pamięci odnoszą się bezpośrednio do tego identyfikatora, ale jego typ zależy od postaci deklaratora. Deklarator jest zwrotem rozumianym w ten sposób, że jeśli jego identyfikator wystąpi w wyrażeniu w takiej samej postaci, jak w deklaratorze, to będzie oznaczać obiekt o wskazanym typie.

Jeśli rozważać tylko te części specyfikatorów deklaracji (p. A8.2), które dotyczą typu, i konkretny deklarator, to deklarację można zapisać jako „T D", gdzie T jest typem, a D jest deklaratorem. Typ przypisany do identyfikatora w różnych postaciach deklaratora będzie opisywany indukcyjnie przy użyciu tego zapisu.

W deklaracji T D, w której D jest niczym nie ozdobionym identyfikatorem, typem identyfikatora jest T.

W deklaracji T D, w której D ma postać

(D1)

typ identyfikatora w D1 jest taki sam, jak typ D. Nawiasy nie zmieniają typu, ale mogą zmienić rozbiór gramatyczny skomplikowanych deklaratorów.

A8.6.1 Deklaratory wskaźników

W deklaracji T D, w której D ma postać

* *lista-kwalifikatorów-typu*$_{opc}$ D1

a typem identyfikatora w deklaracji T D1 jest „*modyfikator-typu* T", typem identyfikatora w D jest „*modyfikator-typu lista-kwalifikatorów-typu* wskaźnik do T". Kwalifikatory następujące po * odnoszą się tylko do wskaźnika, a nie do wskazywanego przezeń obiektu.

Rozważmy dla przykładu deklarację

```
int *ap[ ];
```

Tutaj ap[ ] pełni rolę D1 (deklaracja „int ap[ ]" – patrz p. A8.6.2 – nadawałaby ap typ „tablica elementów typu int"), lista kwalifikatorów typu jest pusta, a modyfikatorem typu jest „tablica elementów typu". Zatem nasza deklaracja nadaje ap typ „tablica elementów typu wskaźnik do typu int" (lub prościej „tablica wskaźników do typu int").

Jako inny przykład rozważmy deklaracje

```
int i, *pi, *const cpi = &i;
const int ci = 3, *pci;
```

Zadeklarowano tutaj obiekt całkowity i oraz wskaźnik do obiektów całkowitych pi. Wartości stałego wskaźnika cpi nie można zmienić – zawsze będzie wskazywał na to samo miejsce pamięci, chociaż wartość w tym miejscu może być zmieniana. Obiekt całkowity ci jest stały i nie można go zmienić (chociaż można go zainicjować, jak powyżej). Typem pci jest „wskaźnik do const int"; wskaźnik pci można zmieniać tak, by pokazywał w inne miejsce pamięci, ale wartości wskazywanego przezeń obiektu nie można zmienić za pomocą pci.

A8.6.2 Deklaratory tablic

W deklaracji T D, w której D ma postać

D1 [*wyrażenie-stałe*$_{opc}$]

a typem identyfikatora w deklaracji T D1 jest „*modyfikator-typu* T", typem identyfikatora w D jest „*modyfikator-typu* tablica elementów typu T". Jeśli występuje *wyrażenie-stałe*, to musi być typu całkowitego i musi mieć wartość większą niż zero. Jeśli nie podano wyrażenia stałego, opisującego rozmiar tablicy, to tablica ma typ niekompletny.

Tablice można konstruować z typów arytmetycznych, ze wskaźników, struktur i unii lub z innych tablic (tworząc tablice wielowymiarowe). Typ obiektów, z których jest zbudowana tablica, musi być kompletny – nie może być tablicą lub strukturą o typie niekompletnym. Dla tablic wielowymiarowych oznacza to, że można opuścić tylko pierwszy wymiar. Typ obiektu o niekompletnym typie tablicowym jest uzupełniany za pomocą innej, kompletnej deklaracji tego obiektu (p. A10.2) lub za pomocą inicjowania (p. A8.7). Na przykład wiersz

```
float fa[17], *afp[17];
```

deklaruje tablicę o elementach typu float i tablicę wskaźników do obiektów typu float. Wiersz

```
static int x3d[3][5][7];
```

deklaruje statyczną trójwymiarową tablicę elementów całkowitych o rozmiarze 3×5×7. Dokładniej mówiąc, x3d jest tablicą o trzech elementach, z których każdy jest

tablicą o pięciu elementach, z których każdy jest tablicą o siedmiu elementach całkowitych. Każde z wyrażeń: x3d, x3d[i], x3d[i][j] oraz x3d[i][j][k] ma sens w wyrażeniach. Pierwsze trzy są typu „tablica", ostatnie zaś jest typu int. Dokładniej, x3d[i][j] jest tablicą złożoną z siedmiu elementów całkowitych, a x3d[i] jest tablicą złożoną z pięciu tablic o siedmiu elementach całkowitych.

Operację indeksowania tablicy definiuje się w ten sposób, że E1[E2] jest identyczne z *(E1+E2). Zatem, mimo asymetrycznego wyglądu, indeksowanie jest operacją przemienną. Zgodnie z regułami przekształceń, odnoszącymi się do operatora + i do tablic (p. A6.6, A7.1, A7.7), jeśli E1 jest tablicą, a E2 jest całkowite, to E1[E2] wyznacza element o numerze E2 w tablicy E1.

W naszym przykładzie x3d[i][j][k] jest równoważne z *(x3d[i][j] + k). Pierwsze podwyrażenie x3d[i][j] jest przekształcane (zgodnie z p. A7.1) do typu „wskaźnik do tablicy o elementach całkowitych"; dodawanie (zgodnie z p. A7.7) powoduje pomnożenie k przez rozmiar obiektów całkowitych. Wynika to z zasady mówiącej, że tablice umieszcza się w pamięci wierszami (ostatni indeks zmienia się najszybciej) oraz że pierwszy wymiar w deklaracji tablicy pozwala określić wielkość zużytej przez tablicę pamięci, ale przy indeksowaniu nie odgrywa żadnej roli.

A8.6.3 Deklaratory funkcji

W nowym stylu deklaracji funkcji T D, w której D ma postać

D1 (*lista-typów-parametrów*)

a typem identyfikatora w deklaracji T D1 jest „*modyfikator-typu* T", typem identyfikatora w D jest „*modyfikator-typu* funkcja z argumentami *lista-typów-parametrów* zwracająca typ T".

Składnia parametrów ma postać

lista-typów-parametrów:
    lista-parametrów
    lista-parametrów , ...

lista-parametrów:
    deklaracja-parametru
    lista-parametrów , *deklaracja-parametru*

deklaracja-parametru:
    specyfikatory-deklaracji deklarator
    specyfikatory-deklaracji deklarator-abstrakcyjny$_{opc}$

W deklaracji w nowym stylu lista parametrów wskazuje typy parametrów. Szczególnym przypadkiem jest deklarator funkcji bezparametrowej, w którym lista typów parametrów zawiera jedynie słowo kluczowe void. Jeśli lista typów parametrów kończy się wielokropkiem „ , ...", to funkcja może akceptować więcej argumentów, niż jawnie podano; patrz p. A7.3.2.

Typy takich parametrów, jak tablice czy funkcje, są – zgodnie z regułami przekształcania parametrów – zamieniane na wskaźniki; patrz p. A10.1. Jedynym specyfikatorem klasy pamięci, dozwolonym w specyfikatorze deklaracji parametru, jest register. Ten specyfikator jest ignorowany, jeśli deklarator funkcji nie jest nagłówkiem definicji funkcji. Podobnie, jeżeli deklaratory w deklaracji parametrów zawierają identyfikatory, a deklarator funkcji nie rozpoczyna definicji funkcji, to te identyfikatory są natychmiast usuwane. Deklaratory abstrakcyjne, które nie wprowadzają identyfikatorów, będą opisane w p. A8.8.

W starym stylu deklaracji funkcji T D, w której D ma postać

D1 (*lista-identyfikatorów*$_{opc}$)

a typem identyfikatora w deklaracji T D1 jest „*modyfikator-typu* T", typem identyfikatora w D jest „*modyfikator-typu* funkcja o nieokreślonych argumentach zwracająca typ T". Parametry (o ile istnieją) mają postać

lista-identyfikatorów:
identyfikator
lista-identyfikatorów , identyfikator

Jeśli deklaracja nie rozpoczyna definicji funkcji, to w deklaracji funkcji w starym stylu nie można umieścić listy identyfikatorów (p. A10.1). Taka deklaracja nie zawiera żadnych informacji o typach parametrów.

Na przykład deklaracja

```
int f(), *fpi(), (*pfi)();
```

wprowadza funkcję f zwracającą wartość całkowitą, funkcję fpi zwracającą wskaźnik do obiektów całkowitych oraz wskaźnik pfi do funkcji zwracającej wartość całkowitą. Dla żadnej z tych funkcji nie podano typów parametrów – wszystkie deklaracje są w starym stylu.

W deklaracji w nowym stylu:

```
int strcpy(char *dest, const char *source), rand(void);
```

strcpy jest funkcją zwracającą wartość całkowitą i mającą dwa parametry: wskaźnik do znaków i wskaźnik do stałych znaków. Nazwy parametrów są w istocie komentarzami. Druga funkcja, rand, nie ma parametrów i zwraca wartość całkowitą.

> Deklaratory funkcji z prototypami parametrów stanowią najpoważniejszą zmianę w języku, wprowadzoną przez Standard ANSI. Ich przewagą w stosunku do deklaratorów „w starym stylu" z pierwszego wydania książki jest możliwość wykrywania błędów i dopasowywania argumentów w wywołaniach funkcji. Spowodowało to jednak wiele zamieszania i nieporozumień przy ich wprowadzaniu, a także konieczność uwzględniania obu postaci deklaracji funkcji. Zachowanie zgodności z poprzednią postacią spowodowało też zeszpecenie składni, wprowadzając słowo void do deklaracji bezparametrowych funkcji nowego stylu.
>
> Wielokropek „ ,..." służący do deklarowania funkcji ze zmienną liczbą argumentów jest także nowy i wraz z makrami zawartymi w standardowym nagłówku <stdarg.h> formalizuje mechanizm, który w pierwszym wydaniu książki był oficjalnie zakazany, ale nieoficjalnie tolerowany.
>
> Te konstrukcje zostały zapożyczone z języka C++.

A8.7 Inicjowanie

W deklaracji obiektu jego *inicjowany-deklarator* może zawierać wartość początkową deklarowanego identyfikatora. Inicjator, który poprzedza się operatorem =, jest albo wyrażeniem, albo listą inicjatorów zawartą w nawiasach klamrowych. Lista może kończyć się przecinkiem – drobiazg umożliwiający schludne formatowanie programu.

inicjator:
wyrażenie-przypisania
{ *lista-inicjatorów* }
{ *lista-inicjatorów* , }

lista-inicjatorów:
inicjator
lista-inicjatorów , *inicjator*

Dla obiektów i tablic statycznych wszystkie wyrażenia w inicjatorach muszą być wyrażeniami stałymi opisanymi w p. A7.19. Dla obiektów i tablic auto lub register wyrażenia w inicjatorach muszą także być wyrażeniami stałymi, jeśli inicjator jest listą ujętą w klamry. Jeśli jednak inicjator dla obiektu automatycznego jest pojedynczym

wyrażeniem, to nie musi być wyrażeniem stałym, ale musi mieć typ odpowiedni dla przypisania do tego obiektu.

> W pierwszym wydaniu książki nie zezwalano na inicjowanie automatycznych struktur, unii i tablic. Standard ANSI pozwala na to, lecz jeśli inicjator nie jest wyrażeniem prostym, to inicjować je można tylko za pomocą stałych konstrukcji.

Nie zainicjowany jawnie obiekt statyczny jest inicjowany tak, jakby jemu (lub jego składowym) przypisano wartość zero. Początkowa wartość nie zainicjowanego jawnie obiektu automatycznego jest niezdefiniowana.

Inicjatorem dla wskaźnika lub obiektu arytmetycznego jest pojedyncze wyrażenie, być może ujęte w nawiasy klamrowe. Wartość wyrażenia jest wstawiana do obiektu.

Inicjatorem dla struktury jest albo wyrażenie tego samego typu, albo ujęta w nawiasy klamrowe lista inicjatorów dla jej kolejnych składowych. Składowe, które są nie nazwanymi polami bitowymi, są ignorowane i nie są inicjowane. Gdy na liście jest mniej inicjatorów niż składowych struktury, wówczas pozostałe składowe są inicjowane zerami. Nie może być więcej inicjatorów niż składowych.

Inicjatorem dla tablicy jest ujęta w klamry lista inicjatorów dla jej kolejnych elementów. Jeśli nie jest znany rozmiar tablicy, to ten rozmiar wyznacza się na podstawie liczby inicjatorów i typ tablicy staje się kompletny. Jeśli tablica ma ustalony rozmiar, to liczba inicjatorów nie może przekroczyć liczby elementów tablicy; gdy jest ich mniej, wówczas pozostałe elementy są inicjowane zerami.

Specjalnym przypadkiem jest tablica znakowa, która może być inicjowana napisem; kolejne znaki napisu inicjują kolejne elementy tablicy. Podobnie napisem złożonym z rozszerzonych znaków (p. A2.6) można inicjować tablicę o elementach typu wchar_t. Jeśli nie jest znany rozmiar tablicy, to ten rozmiar jest wyznaczony przez liczbę znaków w napisie, włączając w to końcowy znak zerowy; jeśli rozmiar jest ustalony, to liczba znaków w napisie, nie licząc końcowego znaku zerowego, nie może przekroczyć rozmiaru tablicy.

Inicjatorem dla unii jest albo pojedyncze wyrażenie tego samego typu, albo ujęty w klamry inicjator dla jej pierwszej składowej.

> W pierwszym wydaniu książki nie wolno było inicjować unii. Zasada „pierwszej składowej" jest niezręczna, ale nie da się tego uogólnić bez zmiany składni. Oprócz możliwości inicjowania unii (przynajmniej w prymitywny sposób) ta zasada ANSI C określa znaczenie nie zainicjowanych jawnie unii statycznych.

Struktury i tablice są obiektami *złożonymi*. Jeśli obiekt złożony zawiera składowe złożonego typu, to zasady inicjowania odnoszą się rekurencyjnie do tych składowych. Następujące zasady regulują stosowanie nawiasów klamrowych przy inicjowaniu. Jeśli inicjator dla składowej obiektu złożonego, która sama jest obiektem złożonym, rozpoczyna się lewą klamrą, to następująca po klamrze lista inicjatorów rozdzielonych przecinkami określa wartości początkowe składowych tej składowej; jest błędem, gdy inicjatorów jest więcej niż jej składowych. Jeżeli natomiast inicjator dla złożonej składowej nie rozpoczyna się klamrą, to z listy bierze się tylko tyle inicjatorów, ile jest wymaganych dla tej składowej. Pozostałe elementy listy służą do zainicjowania następnej składowej obiektu złożonego.

Oto przykłady:

```
int x[ ] = { 1, 3, 5 };
```

Ponieważ nie podano wymiaru, a lista inicjatorów zawiera trzy elementy, wiersz ten deklaruje i inicjuje zmienną x jako jednowymiarową tablicę o trzech elementach.

```
float y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

To inicjowanie jest całkowicie „sklamrowane": 1, 3 i 5 inicjują pierwszy wiersz tablicy y[0], tzn. elementy y[0][0], y[0][1] i y[0][2]. Podobnie następne wartości inicjują wiersze y[1] i y[2]. Inicjator kończy się za wcześnie, zatem elementy wiersza y[3] zostaną zainicjowane zerami. Ten sam efekt można osiągnąć pisząc

```
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

Inicjator dla y rozpoczyna się lewą klamrą, ale już dla y[0] nie – z listy zostaną więc pobrane trzy początkowe elementy. Podobnie, następne porcje po trzy elementy będą pobierane kolejno dla y[1] i y[2]. Natomiast

```
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

inicjuje tylko pierwszą kolumnę tablicy y (dwuwymiarowej), pozostałe elementy wypełniając zerami.

Na koniec deklaracja

```
char msg[ ] = "Błąd składniowy w wierszu %s\n";
```

wprowadza tablicę znakową, której elementy są inicjowane napisem; w jej rozmiarze uwzględnia się zerowy znak kończący napis.

A8.8 Nazwy typów

Podanie nazwy typu danych jest konieczne w kilku przypadkach: przy wymuszaniu przekształcenia typu przez rzutowanie, przy zadeklarowaniu typów parametrów w deklaratorze funkcji oraz w argumencie operatora sizeof. Do tego celu służy *nazwa-typu*, która w składni pełni rolę deklaracji obiektu żądanego typu z pominiętą nazwą obiektu.

nazwa-typu:
    lista-spec-kwal deklarator-abstrakcyjny$_{opc}$

deklarator-abstrakcyjny:
    wskaźnik
    wskaźnik$_{opc}$ *bezp-deklarator-abstrakcyjny*

bezp-deklarator-abstrakcyjny:
    (*deklarator-abstrakcyjny*)
    bezp-deklarator-abstrakcyjny$_{opc}$ [*wyrażenie-stałe*$_{opc}$]
    bezp-deklarator-abstrakcyjny$_{opc}$ (*lista-typów-parametrów*$_{opc}$)

W deklaratorze abstrakcyjnym można jednoznacznie określić miejsce, w którym mógłby wystąpić identyfikator, gdyby taka konstrukcja była deklaratorem w deklaracji. Nazwany typ jest więc identyczny z typem domniemanego identyfikatora. Na przykład nazwy typów:

```
int
int *
int *[3]
int (*)[ ]
int *( )
int (*[ ]) (void)
```

oznaczają odpowiednio: „typ całkowity", „wskaźnik do obiektów całkowitych", „tablica złożona z trzech wskaźników do elementów całkowitych", „wskaźnik do tablicy

o nieokreślonej liczbie elementów całkowitych", „funkcja o nieokreślonych parametrach zwracająca wskaźnik do obiektów całkowitych" oraz „tablica o nieokreślonej liczbie elementów, którymi są wskaźniki do bezparametrowych funkcji o wartościach całkowitych".

A8.9 Nazwy **typedef**

Deklaracje, w których specyfikatorem klasy pamięci jest typedef, nie deklarują obiektów, natomiast definiują identyfikatory jako nazwy typów. O takich identyfikatorach mówimi, że są nazwami typedef.

nazwa-typedef:
    identyfikator

Deklaracja typedef określa w zwykły sposób (patrz p. A8.6) właściwości typu dla każdej nazwy występującej wśród jej deklaratorów. Po zadeklarowaniu każda nazwa typedef jest składniowo równoważna ze słowem kluczowym specyfikatora typu dla związanego z nią typu.

Na przykład po deklaracjach

```
typedef long Blockno, *Blockptr;
typedef struct { double r, theta; } Complex;
```

wszystkie konstrukcje

```
Blockno b;
extern Blockptr bp;
Complex z, *zp;
```

są poprawnymi deklaracjami. Typem zmiennej b jest long, typem bp jest „wskaźnik do long", typem z jest wskazana struktura, a typem zp jest wskaźnik do takiej struktury.

Deklaracje typedef nie wprowadzają nowych typów, tylko synonimy dla typów, które mogłyby być określone w inny sposób. Zatem w powyższym przykładzie b ma taki sam typ, jak inne obiekty typu long.

Nazwy typedef mogą być ponownie zadeklarowane w węższym zasięgu pod warunkiem, że zbiór specyfikatorów typu nie jest pusty. Na przykład

```
extern Blockno;
```

nie deklaruje ponownie Blockno, ale

```
extern int Blockno;
```

tak.

A8.10 Równoważność typów

Dwie listy specyfikatorów typu są równoważne, jeśli zawierają ten sam zestaw specyfikatorów typu, wliczając w to te specyfikatory, które wynikają z innych (np. samotne long implikuje long int). Struktury, unie i wyliczenia z różnymi etykietkami są różne; także struktury, unie i wyliczenia bez etykietek wprowadzają typy unikalne.

Dwa typy są takie same, jeśli ich deklaratory abstrakcyjne (p. A8.8), ewentualnie po rozwinięciu typów typedef i usunięciu nazw parametrów funkcji, są takie same aż po równoważność list specyfikatorów typu. Rozmiary tablic i typy parametrów funkcji są tu znaczące.

A9 Instrukcje

Jeśli nie powiedziano inaczej, instrukcje są wykonywane sekwencyjnie. Instrukcje wykonuje się w celu osiągnięcia pewnych efektów; instrukcje nie mają wartości. Rozróżnia się kilka grup instrukcji.

instrukcja:
instrukcja-etykietowana
instrukcja-wyrażeniowa
instrukcja-złożona
instrukcja-wyboru
instrukcja-powtarzania
instrukcja-skoku

A9.1 Instrukcje etykietowane

Instrukcje mogą być poprzedzone przedrostkami etykietowymi.

instrukcja-etykietowana:
identyfikator : *instrukcja*
case *wyrażenie-stałe* : *instrukcja*
default : *instrukcja*

Etykieta składająca się z identyfikatora deklaruje ten identyfikator. Jedynym sposobem skorzystania z takiego identyfikatora jest użycie go w instrukcji goto. Zasięgiem tego identyfikatora jest bieżąca funkcja. Etykiety tworzą oddzielną przestrzeń nazw, nie wchodzą więc w konflikt z innymi identyfikatorami, a także nie można ich ponownie zadeklarować. Patrz p. A11.1.

Etykiety przypadków (case) i domyślne (default) są używane w instrukcji switch (p. A9.4). Wyrażenie stałe występujące w etykiecie case musi mieć typ całkowity.

Etykiety jako takie nie wpływają na sterowanie programem.

A9.2 Instrukcja wyrażeniowa

Większość instrukcji to instrukcje wyrażeniowe o postaci

instrukcja-wyrażeniowa:
wyrażenie$_{opc}$;

Większość instrukcji wyrażeniowych to przypisania i wywołania funkcji. Wszystkie efekty uboczne spowodowane przez wyrażenie są zakończone przed rozpoczęciem wykonania następnej instrukcji. Konstrukcja, w której pominięto wyrażenie, nazywa się instrukcją pustą; jest ona często używana do oznaczenia pustej treści instrukcji powtarzania lub do wstawiania etykiet.

A9.3 Instrukcja złożona (blok)

Aby można było użyć kilku instrukcji tam, gdzie oczekuje się jednej, wprowadzono instrukcję złożoną (zwaną również *blokiem*). Treść definicji funkcji jest instrukcją złożoną.

instrukcja-złożona:
{ *lista-deklaracji*$_{opc}$ *lista-instrukcji*$_{opc}$ }

lista-deklaracji:
deklaracja
lista-deklaracji deklaracja

lista-instrukcji:
instrukcja
lista-instrukcji instrukcja

Jeśli dowolny identyfikator z listy deklaracji był zadeklarowany na zewnątrz bloku, to jego poprzednia deklaracja jest przesłaniana na czas wykonywania bloku (patrz p. A11.1); po opuszczeniu bloku jej ważność będzie przywrócona. Dany identyfikator może być tylko raz zadeklarowany w bloku. Ta zasada dotyczy identyfikatorów nale-

żących do tej samej przestrzeni nazw (p. A11); identyfikatory należące do różnych przestrzeni nazw są traktowane jako różne.

Inicjowanie obiektów automatycznych wykonuje się w kolejności ich deklaracji przy każdym wejściu do bloku przez jego początek. Inicjowania nie wykonuje się przy skoku do wnętrza bloku. Inicjowanie obiektów static wykonuje się tylko raz, zanim program rozpocznie działanie.

A9.4 Instrukcje wyboru

Instrukcje wyboru służą do wybrania jednej z kilku ścieżek wykonania programu.

instrukcja-wyboru:
if (*wyrażenie*) *instrukcja*
if (*wyrażenie*) *instrukcja* else *instrukcja*
switch (*wyrażenie*) *instrukcja*

W obu postaciach instrukcji if wyrażenie musi mieć typ arytmetyczny lub wskaźnikowy. Jeśli wartość wyrażenia (po obliczeniu wszystkich efektów ubocznych) jest różna od zera, to wykonuje się pierwszą podinstrukcję. W drugiej postaci, jeżeli wartość wyrażenia jest równa zeru, to wykonuje się drugą podinstrukcję. Niejednoznaczność przyporządkowania else rozstrzyga się, wiążąc else z ostatnią instrukcją if bez else, napotkaną na tym samym poziomie struktury blokowej.

Instrukcja switch powoduje przekazanie sterowania do jednej z szeregu instrukcji, zależnie od wartości wyrażenia (które musi mieć typ całkowity). Podinstrukcja kontrolowana przez instrukcję switch jest w praktyce instrukcją złożoną. Każda instrukcja wewnątrz instrukcji złożonej może być etykietowana jedną lub kilkoma etykietami przypadków (case – p. A9.1). Wyrażenie kontrolne podlega promocji typu całkowitego (p. A6.1), a stałe przypadków są przekształcane do promowanego typu. W jednej instrukcji switch wszystkie stałe przypadków muszą mieć po przekształceniu różne wartości. Z instrukcją switch może być związana co najwyżej jedna etykieta domyślna default. Instrukcje switch mogą być w sobie zagnieżdżone; etykiety case i default są związane z najciaśniej otaczającą je instrukcją switch.

Wykonanie instrukcji switch polega na obliczeniu jej wyrażenia (wraz z efektami ubocznymi) i porównaniu jego wartości z wartościami stałych przypadków. Jeśli wartość jednej ze stałych jest równa wartości wyrażenia, to sterowanie jest przekazywane do instrukcji opatrzonej tą etykietą przypadku. Jeśli żadna ze stałych przypadków nie jest równa wartości wyrażenia i występuje etykieta domyślna default, to sterowanie jest przekazywane do instrukcji opatrzonej tą etykietą. Gdy wszystkie

stałe przypadków są różne od wartości wyrażenia i nie występuje etykieta domyślna, wówczas nie wykonuje się żadnej z podinstrukcji instrukcji switch.

> W pierwszym wydaniu książki dla wyrażenia kontrolnego i stałych przypadków instrukcji switch wymagano typu int.

A9.5 Instrukcje powtarzania (pętle)

Instrukcje powtarzania wskazują iteracyjne wykonywanie fragmentu programu.

instrukcja-powtarzania:
while (*wyrażenie*) *instrukcja*
do *instrukcja* while (*wyrażenie*) ;
for ($wyrażenie_{opc}$; $wyrażenie_{opc}$; $wyrażenie_{opc}$) *instrukcja*

W instrukcjach while i do wykonanie podinstrukcji powtarza się dopóty, dopóki wartość wyrażenia jest różna od zera; wyrażenie musi mieć typ arytmetyczny lub wskaźnikowy. W instrukcji while wartość wyrażenia (po obliczeniu wszystkich efektów ubocznych) jest sprawdzana przed wykonaniem podinstrukcji; w instrukcji do – za każdym razem po wykonaniu podinstrukcji.

W instrukcji for pierwsze podwyrażenie oblicza się tylko raz; jest to część inicjująca stan pętli. Nie ma ograniczeń dotyczących typu tego wyrażenia. Drugie wyrażenie musi mieć typ arytmetyczny lub wskaźnikowy; jest ono obliczane przed każdym przebiegiem pętli i jeśli stanie się zerem, to wykonanie instrukcji for zostanie przerwane. Trzecie wyrażenie jest obliczane po każdym przebiegu pętli i określa zmianę stanu pętli. Nie ma ograniczeń dotyczących typu tego wyrażenia. Efekty uboczne spowodowane wyrażeniami wykonuje się bezpośrednio po obliczeniu każdego z nich. Jeśli podinstrukcja nie zawiera instrukcji continue, to instrukcja

for (*wyrażenie1* ; *wyrażenie2* ; *wyrażenie3*) *instrukcja*

jest równoważna z ciągiem instrukcji

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

Dowolne z tych trzech wyrażeń można pominąć. Pominięcie drugiego wyrażenia jest równoważne z zastąpieniem go stałą różną od zera.

A9.6 Instrukcje skoku

Instrukcje skoku powodują bezwarunkowe przekazanie sterowania do innego miejsca w programie.

instrukcja-skoku:
goto *identyfikator* ;
continue ;
break ;
return *wyrażenie*$_{opc}$;

W instrukcji goto identyfikator musi być etykietą (p. A9.1) umieszczoną w tej samej funkcji. Sterowanie jest przekazywane do instrukcji opatrzonej tą etykietą.

Instrukcja continue może wystąpić tylko w instrukcjach powtarzania. Powoduje przekazanie sterowania do miejsca wznawiania najciaśniej otaczającej ją pętli. Dokładniej mówiąc, w poniższych instrukcjach:

```
while (...) {          do {              for (...) {
  ...                    ...               ...
contin: ;              contin: ;         contin: ;
}                      } while (...);    }
```

instrukcja continue nie występująca w ciaśniej otaczającej pętli jest tym samym, co goto contin.

Instrukcja break może wystąpić tylko w instrukcjach powtarzania i instrukcji switch, i powoduje przerwanie najciaśniej otaczającej ją takiej instrukcji; sterowanie jest przekazywane do instrukcji następnej po przerwanej instrukcji.

Funkcja wraca do miejsca wywołania za pomocą instrukcji return. Jeśli po słowie return występuje wyrażenie, to wartość tego wyrażenia jest przekazywana do miejsca wywołania. Wartość wyrażenia przekształca się (jak w przypisaniu) do typu zwracanego przez funkcję.

Przekroczenie treści funkcji jest równoważne z wykonaniem instrukcji return bez wyrażenia. W obu przypadkach wartość funkcji nie jest określona.

A10 Deklaracje zewnętrzne

Tekst programu przekazany do kompilatora języka C jest jednostką tłumaczenia; jednostka ta składa się z ciągu deklaracji zewnętrznych, które są albo deklaracjami obiektów, albo definicjami funkcji.

jednostka-tłumaczenia:
deklaracja-zewnętrzna
jednostka-tłumaczenia deklaracja-zewnętrzna

deklaracja-zewnętrzna:
definicja-funkcji
deklaracja

Zasięg deklaracji zewnętrznych rozciąga się do końca zawierającej je jednostki tłumaczenia, podobnie jak zasięg deklaracji zawartych w bloku rozciąga się do końca tego bloku. Składnia deklaracji zewnętrznej jest taka sama, jak składnia wszystkich innych deklaracji, jednakże tylko na tym poziomie można podać treść funkcji.

A10.1 Definicje funkcji

Definicja funkcji ma postać

definicja-funkcji:
specyfikatory-deklaracji$_{opc}$ *deklarator lista-deklaracji*$_{opc}$ *instrukcja-złożona*

Dozwolonymi specyfikatorami klasy pamięci w specyfikatorach deklaracji są tylko extern i static; różnice między nimi opisano w p. A11.2.

Funkcja może zwracać wartość typu arytmetycznego, strukturę, unię, wskaźnik lub void; natomiast nie może zwrócić funkcji ani tablicy. Deklarator w deklaracji funkcji musi jawnie wskazywać, że deklarowany identyfikator ma typ funkcyjny, tzn. musi zawierać jedną z poniższych konstrukcji (patrz p. A8.6.3):

bezpośredni-deklarator (*lista-typów-parametrów*)
bezpośredni-deklarator (*lista-identyfikatorów*$_{opc}$)

gdzie *bezpośredni-deklarator* jest identyfikatorem, ewentualnie ujętym w nawiasy. W szczególności nie można zbudować typu funkcyjnego za pomocą typedef.

Pierwsza postać deklaratora definiuje funkcję w nowym stylu; jej parametry wraz z typami są deklarowane na liście typów parametrów; listę deklaracji, następującą po deklaratorze funkcji, należy pominąć. Jeżeli lista typów parametrów nie składa się wyłącznie ze słowa void oznaczającego funkcję bezparametrową, to każdy z deklaratorów na liście typów parametrów musi zawierać identyfikator. Jeśli lista parametrów kończy się wielokropkiem „, ...", to funkcja może być wywołana z liczbą argumen-

tów większą niż liczba parametrów; w odwołaniach do nadliczbowych argumentów należy stosować mechanizm makra va_arg, zdefiniowany w standardowym nagłówku <stdarg.h> i opisany w dodatku B. Funkcje ze zmienną listą argumentów muszą mieć co najmniej jeden nazwany parametr.

Druga postać deklaratora definiuje funkcję w starym stylu; na liście identyfikatorów występują nazwy parametrów, podczas gdy lista deklaracji określa ich typy. Jeśli dla parametru nie podano deklaracji, to przyjmuje się typ int. Lista deklaracji może zawierać deklaracje tylko tych parametrów, które umieszczono na liście parametrów; inicjowanie jest zakazane, a jedynym możliwym specyfikatorem klasy pamięci jest register.

W obu postaciach definicji funkcji przyjmuje się, że parametry są deklarowane na samym początku instrukcji złożonej tworzącej treść funkcji – nie można więc tam ponownie deklarować tych samych identyfikatorów (chociaż można je, jak i inne identyfikatory, deklarować w blokach wewnętrznych). Jeśli parametr jest zadeklarowany jako „tablica elementów *typu*", to tę deklarację zamienia się na „wskaźnik do *typu*"; podobnie, jeśli parametr jest zadeklarowany jako „funkcja zwracająca *typ*", to tę deklarację zamienia się na „wskaźnik do funkcji zwracającej *typ*". Przy wywołaniu funkcji argumenty są przekształcane w miarę potrzeby i wstawiane do parametrów; patrz p. A7.3.2.

> Nowy styl definicji funkcji pojawił się wraz z ANSI C. Nastąpiła również drobna zmiana w promocji typów: w pierwszym wydaniu mówiło się, że parametry zadeklarowane jako float będą rozciągane do typu double. Różnica ta staje się widoczna, gdy wewnątrz funkcji jest generowany wskaźnik do takiego parametru.

Oto pełny przykład definicji funkcji w nowym stylu:

```
int max(int a, int b, int c)
{
    int m;

    m = (a > b) ? a : b;
    return (m > c) ? m : c;
}
```

Tutaj int jest specyfikatorem deklaracji; max(int a, int b, int c) jest deklaratorem funkcji, a { ... } jest blokiem stanowiącym treść funkcji. Odpowiednia definicja funkcji w starym stylu wyglądałaby następująco:

```
int max(a, b, c)
int a, b, c;
{
    /* ... */
}
```

gdzie deklaratorem jest teraz int max(a, b, c), a int a, b, c; jest listą deklaracji parametrów.

A10.2 Deklaracje zewnętrzne

Deklaracje zewnętrzne opisują właściwości obiektów, funkcji i innych identyfikatorów. Pojęcie „zewnętrzne" odnosi się do ich położenia na zewnątrz wszystkich funkcji i nie jest bezpośrednio związane ze słowem kluczowym extern; klasę pamięci dla obiektów zadeklarowanych na zewnątrz można pominąć lub można podać jako extern lub static.

W tej samej jednostce tłumaczenia może wystąpić kilka zewnętrznych deklaracji tego samego identyfikatora pod warunkiem, że są one zgodne pod względem typu i łączności oraz że istnieje co najwyżej jedna definicja tego identyfikatora.

Dwie deklaracje tego samego obiektu lub funkcji uważa się za zgodne ze względu na typ, jeśli spełniają warunki opisane w p. A8.10. Ponadto, jeśli deklaracje się różnią, ponieważ jeden z typów jest niekompletną strukturą, unią lub wyliczeniem (p. A8.3), a drugi jest odpowiednim kompletnym typem z tą samą etykietką, to te typy uważa się za zgodne. Uważa się za zgodne również typy, z których jeden jest niekompletnym, drugi zaś kompletnym typem tablicowym (p. A8.6.2), i które poza tym są identyczne. Wreszcie, jeśli jeden z typów określa funkcję w starym stylu, a drugi funkcję w nowym stylu z deklaracjami parametrów, a poza tym identyczną, to te typy uważa się za zgodne.

Jeśli pierwsza z zewnętrznych deklaracji funkcji lub obiektu zawiera specyfikator static, to identyfikator ma „łączność wewnętrzną"; w przeciwnym przypadku ma „łączność zewnętrzną". Łączność identyfikatorów będzie opisana w p. A11.2.

Zewnętrzna deklaracja obiektu jest jego definicją, jeśli zawiera inicjowanie. Zewnętrzna deklaracja bez inicjowania i bez specyfikatora extern nazywa się *definicją próbną*. Gdy w jednostce tłumaczenia pojawi się definicja obiektu, wówczas jego wszystkie definicje próbne traktuje się jak zbędne deklaracje. Jeśli w jednostce tłumaczenia w ogóle nie wystąpi definicja dla tego obiektu, to jego wszystkie definicje próbne stają się pojedynczą definicją z wartością początkową równą zero.

Każdy obiekt musi mieć dokładnie jedną definicję. Dla obiektów z łącznością wewnętrzną ta zasada odnosi się oddzielnie do każdej jednostki tłumaczenia, ponieważ obiekty z łącznością wewnętrzną są unikalne w ramach jednostki tłumaczenia. Dla obiektów z łącznością zewnętrzną ta zasada odnosi się do całego programu.

> Chociaż w pierwszym wydaniu książki zasadę pojedynczej definicji sformułowano nieco inaczej, była ona w praktyce identyczna z powyższą. W niektórych implementacjach zasadę tę nieco złagodzono, rozszerzając znaczenie definicji próbnej. Alternatywna zasada, obowiązująca zwykle w systemach UNIX i powszechnie traktowana jako rozszerzenie Standardu, mówi, że wszystkie definicje próbne obiektów z łącznością zewnętrzną są rozpatrywane łącznie dla wszystkich jednostek tłumaczenia, nie zaś dla każdej jednostki oddzielnie. Jeśli gdziekolwiek w programie wystąpi definicja obiektu, to jego definicje próbne po prostu stają się deklaracjami; jeśli jednak definicja nie wystąpi, to wszystkie definicje próbne stają się pojedynczą definicją z wartością początkową równą zero.

A11 Zasięg i łączność nazw

Cały program nie musi być tłumaczony w tym samym czasie: tekst źródłowy można trzymać w kilku plikach zawierających jednostki tłumaczenia, a uprzednio przetłumaczone podprogramy można dołączać z bibliotek. Komunikacja między funkcjami programu może być realizowana zarówno przez wywołania, jak i przez manipulowanie danymi zewnętrznymi.

Mamy zatem do rozważenia dwa rodzaje zasięgu nazw. Pierwszy, *leksykalny zasięg* identyfikatora, dotyczy tego fragmentu programu, w którym są znane właściwości identyfikatora. Drugi, związany z obiektami i funkcjami o łączności zewnętrznej, rozstrzyga o powiązaniach między identyfikatorami pochodzącymi z oddzielnie kompilowanych jednostek tłumaczenia.

A11.1 Zasięg leksykalny

Identyfikatory należą do kilku nie związanych ze sobą przestrzeni nazw; ten sam identyfikator może być użyty w kilku różnych kontekstach, nawet w tym samym zasięgu, jeżeli tylko te konteksty wyznaczają różne przestrzenie nazw. Takimi przestrzeniami są: obiekty, funkcje, nazwy typedef i stałe wyliczeń; etykiety; etykietki struktur, unii i wyliczeń; składowe oddzielnie dla każdej struktury i unii.

Te zasady różnią się pod kilkoma względami od zasad opisanych w pierwszym wydaniu tego przewodnika. Poprzednio etykiety nie miały własnej przestrzeni nazw; etykietki struktur i unii, a w niektórych implementacjach również etykietki wyliczeń, należały do odrębnych przestrzeni nazw; włożenie różnego rodzaju etykietek do wspólnej przestrzeni jest nowym ograniczeniem. Najbardziej istotnym odejściem od reguł z pierwszego wydania jest zasada, że każda struktura i unia tworzy oddzielną przestrzeń nazw dla swoich składowych, ta sama nazwa może więc wystąpić w kilku różnych strukturach. Tę zasadę powszechnie stosowano od wielu lat.

Leksykalny zasięg identyfikatora obiektu lub funkcji, występującego w deklaracji zewnętrznej, rozciąga się od końca jego deklaratora do końca jednostki tłumaczenia zawierającej deklarację. Zasięg parametrów definicji funkcji rozpoczyna się z początkiem bloku definiującego tę funkcję i rozciąga na całą funkcję; zasięg parametru w deklaracji funkcji kończy się wraz z końcem deklaratora. Zasięg identyfikatora zadeklarowanego w nagłówku bloku rozciąga się od końca jego deklaratora do końca tego bloku. Zasięgiem etykiety jest cała zawierająca ją funkcja. Zasięg etykietki struktury, unii lub wyliczenia oraz stałej wyliczenia zaczyna się w miejscu jej wystąpienia w specyfikatorze typu i rozciąga na całą jednostkę tłumaczenia (dla deklaracji na poziomie zewnętrznym) lub do końca bloku (dla deklaracji wewnątrz funkcji).

Jawna deklaracja identyfikatora w nagłówku bloku, wliczając w to blok tworzący treść funkcji, zasłania aż do końca bloku wszystkie deklaracje tego samego identyfikatora podane na zewnątrz bloku.

A11.2 Łączność nazw

Wszystkie deklaracje tego samego identyfikatora dla obiektu lub funkcji o łączności wewnętrznej odnoszą się do tej samej rzeczy w całej jednostce tłumaczenia i ten obiekt (lub funkcja) jest unikalny dla całej jednostki tłumaczenia. Wszystkie deklaracje tego samego identyfikatora dla obiektu lub funkcji o łączności zewnętrznej odnoszą się do tej samej rzeczy i ten obiekt (lub funkcja) jest współużywalny w całym programie.

Zgodnie z p. A10.2, gdy pierwsza zewnętrzna deklaracja identyfikatora zawiera specyfikator static, wówczas nadaje ona identyfikatorowi łączność wewnętrzną; w pozostałych przypadkach identyfikator ma łączność zewnętrzną. Jeśli deklaracja identyfikatora wewnątrz bloku nie zawiera specyfikatora extern, to ten identyfikator nie ma łączności i jest unikalny w funkcji. Gdy ta deklaracja zawiera extern i gdy w zasięgu otaczającym blok obowiązuje zewnętrzna deklaracja tego samego identyfikatora, wówczas ten identyfikator ma taką samą łączność, jak w deklaracji zewnętrznej i odnosi się do tego samego obiektu lub funkcji. Jeśli jednak w zasięgu widoczności nie ma żadnej deklaracji zewnętrznej, to łączność tego identyfikatora jest zewnętrzna.

A12 Preprocesor

Preprocesor umożliwia makrogenerację, kompilację warunkową oraz włączanie do programów zawartości wskazanych plików. Wiersze programu rozpoczynające się znakiem # (ewentualnie poprzedzonym odstępami) służą do komunikacji z preprocesorem. Składnia tych wierszy jest niezależna od reszty języka; mogą one wystąpić gdziekolwiek w programie, a ich działanie kończy się (niezależnie od zasad dostępności) wraz z końcem jednostki tłumaczenia. Granice wierszy są istotne – każdy wiersz jest rozpatrywany indywidualnie (ale w p. A12.2 opisano, jak można taki wiersz przedłużyć). Leksemami dla preprocesora są wszystkie leksemy języka lub ciągi znaków tworzące nazwy plików, jak w wierszu #include (p. A12.4). Ponadto każdy znak, którego nie zdefiniowano inaczej, jest traktowany jako leksem, ale w wierszach preprocesora znaczenie białych znaków innych niż odstęp i tabulacja pozioma nie jest określone.

Przebieg preprocesora odbywa się w kilku logicznie po sobie następujących fazach, które w konkretnych implementacjach mogą być połączone:

1. Trzyznakowe sekwencje opisane w p. A12.1 są jako pierwsze zastępowane przez ich odpowiedniki. Jeśli środowisko systemu operacyjnego tego wymaga, to pomiędzy wiersze programu wstawia się znaki nowego wiersza.
2. Wszystkie wystąpienia kombinacji znaku \ i nowego wiersza są usuwane, co prowadzi do sklejenia wierszy (p. A12.2).
3. Program jest przekształcany na ciąg leksemów rozdzielonych odstępami; komentarze zastępuje się pojedynczym odstępem. Następnie wykonuje się instrukcje preprocesora i dokonuje makrorozwinięć (p. A12.3-A12.10).
4. Sekwencje specjalne w stałych znakowych i napisach (p. A2.5.2, A2.6) zastępuje się ich odpowiednikami; następnie skleja się sąsiadujące ze sobą napisy.
5. Wynik jest tłumaczony, a następnie łączony z innymi programami i bibliotekami, by skompletować niezbędne podprogramy i dane oraz by odniesienia do zewnętrznych funkcji i obiektów powiązać z ich definicjami.

A12.1 Sekwencje trzyznakowe

Zbiór znaków stosowanych w programach źródłowych w języku C zawiera się w 7-bitowym zbiorze ASCII, ale jest nadzbiorem zbioru ISO 646-1983 Invariant Code Set. Aby umożliwić napisanie programów w tym ograniczonym zbiorze znaków, wszystkie wystąpienia następujących trzyznakowych sekwencji zastępuje się odpowiednimi pojedynczymi znakami. Odbywa się to przed jakimkolwiek innym przetwarzaniem tekstu programu.

??=	#	??(	[	??<	{
??/	\	??)	]	??>	}
??'	^	??!	|	??-	~

Żadne inne podobne zastąpienia nie występują.

Trzyznakowe sekwencje pojawiły się wraz z ANSI C.

A12.2 Sklejanie wierszy

Wiersze, które kończą się znakiem \, łączy się z następnym wierszem przez usunięcie znaku \ i następującego po nim znaku nowego wiersza. Odbywa się to przed podziałem programu na leksemy.

A12.3 Definicje i rozwinięcia makr

Wiersz sterujący o postaci

define *identyfikator ciąg-leksemów*

zleca preprocesorowi zastępowanie dalszych wystąpień identyfikatora wskazanym ciągiem leksemów; opuszcza się odstępy otaczające ciąg leksemów. Ponowne wystąpienie #define z tym samym identyfikatorem traktuje się jako błędne, jeżeli ciągi leksemów nie są w obu przypadkach identyczne (przy czym wszystkie białe plamy rozdzielające leksemy traktuje się jako równoważne).

Wiersz o postaci

define *identyfikator* (*lista-identyfikatorów*) *ciąg-leksemów*

w którym między pierwszym identyfikatorem a nawiasem otwierającym nie ma odstępu, jest makrodefinicją z parametrami podanymi na liście identyfikatorów. Podobnie jak w poprzednim przypadku, opuszcza się odstępy otaczające ciąg leksemów, a makro może być ponownie zdefiniowane, jeśli w obu definicjach liczba i pisownia parametrów oraz ciągi leksemów są identyczne.

Wiersz sterujący o postaci

undef *identyfikator*

zleca preprocesorowi, aby zapomniał definicję identyfikatora. Zastosowanie #undef do nie zdefiniowanego identyfikatora nie jest błędem.

Jeśli makro zdefiniowano za pomocą drugiej formy #define, to każde późniejsze wystąpienie identyfikatora makra, po którym następują: nawias otwierający (ewentualnie poprzedzony odstępem), ciągi leksemów rozdzielone przecinkami i nawias zamykający, jest wywołaniem makra. Argumentami wywołania makra są ciągi leksemów rozdzielone przecinkami; przecinki zawarte w apostrofach i cudzysłowach oraz chronione przez zagnieżdżone nawiasy nie dzielą argumentów. Podczas kompletowania argumentów występujące w nich makra nie są rozwijane. Liczba argumentów w wywołaniu musi być taka sama, jak liczba parametrów w definicji. Po wyizolowaniu argumentów usuwa się z nich otaczające je odstępy. Następnie, w ciągu leksemów tworzącym makrodefinicję każde wystąpienia identyfikatora parametru – oprócz tych, które są zawarte w cudzysłowach i apostrofach – jest zastępowane wynikowym ciągiem leksemów odpowiedniego argumentu. Jeśli występujący w ciągu leksemów parametr nie jest poprzedzony znakiem # lub nie występuje przed nim lub po nim operator ##, to leksemy argumentu przegląda się w poszukiwaniu innych makrowywołań i rozwija, gdy trzeba, tuż przed zastąpieniem.

Dwa specjalne operatory wpływają na proces rozwijania makr. Po pierwsze, jeżeli w zastępującym ciągu leksemów wystąpienie parametru jest bezpośrednio poprzedzone znakiem #, to identyfikator parametru wraz ze znakiem # zostaną zastąpione odpowiednim argumentem otoczonym znakami cudzysłowu (" "). Każdy ze znaków " i \ występujący na początku, w środku lub na końcu w stałych znakowych i napisach tworzących argument zostanie poprzedzony znakiem \.

Po drugie, gdy w ciągu leksemów tworzącym definicję makra o dowolnej postaci wystąpi operator ##, wówczas – tuż po zastąpieniu parametrów – operator ten wraz z otaczającymi go białymi plamami zostanie usunięty. Powoduje to sklejenie sąsiadujących leksemów i utworzenie nowego leksemu. Jeśli tak utworzony leksem jest niepoprawny lub jeśli jest istotna kolejność wykonywania operatorów ##, to skutek całej operacji nie jest zdefiniowany. Ponadto operator ## nie może wystąpić ani na początku, ani na końcu zastępującego ciągu leksemów.

Niezależnie od postaci makrodefinicji zastępujący ciąg leksemów jest wielokrotnie przeglądany w poszukiwaniu innych tak zdefiniowanych identyfikatorów. Jeżeli jednak identyfikator, zastąpiony już w danym rozwinięciu, znów pojawi się przy ponownym przeglądaniu, to pozostanie w rozwiniętym tekście bez zmiany.

Wynik makrorozwinięcia rozpoczynający się znakiem # nie jest traktowany jak instrukcja preprocesora.

> Szczegóły procesu rozwijania makr są w ANSI C opisane dokładniej niż w pierwszym wydaniu książki. Najistotniejszą zmianą jest dodanie operatorów # i ##, które umożliwiają otaczanie tekstu znakami cudzysłowu i sklejanie napisów. Pewne nowe zasady, szczególnie te, które dotyczą sklejania napisów, są dziwaczne (patrz przykład przy końcu tego punktu).

Mechanizm makrodefinicji przydaje się do definiowania „ważnych stałych", np.

```
#define TABSIZE 100
int table[TABSIZE];
```

Wiersz

```
#define ABSDIFF(a, b) ((a)>(b) ? (a)-(b) : (b)-(a))
```

definiuje makro zwracające wartość absolutną różnicy jego argumentów. W przeciwieństwie do funkcji, która robi to samo, argumenty makra i zwracana wartość mogą być dowolnego typu arytmetycznego lub nawet mogą być wskaźnikami. Jednak argumenty (a te mogą powodować efekty uboczne) są tu obliczane dwa razy – raz przy porównaniu i drugi raz przy produkowaniu wyniku.

Przy definicji

```
#define tempfile(dir) #dir "/%s"
```

wywołanie `tempfile(/usr/tmp)` daje w wyniku

```
"/usr/tmp" "/%s"
```

co następnie zostanie sklejone w jeden napis. Przy definicji

```
#define cat(x, y)      x ## y
```

wywołanie `cat(var,123)` produkuje leksem `var123`. Natomiast wynik wywołania `cat(cat(1,2),3)` jest nieokreślony: obecność operatora ## zapobiega rozwinięciu argumentów wewnętrznego wywołania. Zatem w efekcie powstanie ciąg leksemów

```
cat ( 1 , 2 )3
```

w którym leksem `)3` (wynik sklejenia ostatniego leksemu pierwszego argumentu z pierwszym leksemem drugiego) jest niepoprawny.

Po wprowadzeniu makra drugiego poziomu

```
#define xcat(x,y) cat(x,y)
```

wszystko to działa bardziej przyzwoicie: `xcat(xcat(1, 2), 3)` rzeczywiście tworzy leksem `123`, ponieważ w samym `xcat` nie występuje operator ##.

Podobnie `ABSDIFF(ABSDIFF(a,b),c)` zgodnie z oczekiwaniem tworzy wynik w pełni rozwinięty.

A12.4 Włączanie plików

Wiersz sterujący o postaci

include *<nazwa-pliku>*

zleca wstawienie w jego miejsce całej zawartości pliku o podanej nazwie. Wśród znaków nazwy pliku nie może wystąpić znak > i znak nowego wiersza, a skutek tej instrukcji jest nieokreślony, jeśli wystąpi któryś ze znaków ", ', \ lub para znaków /*. Wskazany plik jest poszukiwany w kilku miejscach zależnych od implementacji.

W przypadku wiersza sterującego o postaci

include *"nazwa-pliku"*

plik jest najpierw poszukiwany w miejscu związanym z oryginalnym plikiem źródłowym (wyrażenie celowo zależne od implementacji), a gdy to się nie powiedzie – w tych samych miejscach, co w poprzednim przypadku. Skutek użycia w nazwie pliku znaków ', \ oraz pary znaków /* jest nadal nieokreślony, ale znak > jest dozwolony.

Na koniec, w wierszu o postaci

include *ciąg-leksemów*

różnej od obu poprzednich, ciąg leksemów jest rozwijany w zwykły sposób; w wyniku musi powstać jedna z postaci <...> lub "...", która zostanie zinterpretowana tak, jak opisano powyżej.

Włączanie plików za pomocą #include może być zagnieżdżone.

A12.5 Kompilacja warunkowa

Fragmenty programu mogą być kompilowane warunkowo, zgodnie z poniższą schematyczną składnią:

kompilacja-warunkowa:
    wiersz-if tekst części-elif część-else$_{opc}$ #endif

wiersz-if:
    # if *wyrażenie-stałe*
    # ifdef *identyfikator*
    # ifndef *identyfikator*

części-elif:
    wiersz-elif tekst
    części-elif$_{opc}$

wiersz-elif:
    # elif *wyrażenie-stałe*

część-else:
    wiersz-else tekst

wiersz-else:
    # else

Każdy z takich wierszy preprocesora (*wiersz-if*, *wiersz-elif*, *wiersz-else* i wiersz zawierający #endif) pojawia się w oddzielnym wierszu programu. Wyrażenia stałe, występujące w #if i w następujących #elif, są obliczane kolejno, aż do napotkania wyrażenia z niezerową wartością. Tekst następujący po takich wierszach z zerową wartością opuszcza się. Tekst następujący po takim wierszu z „pozytywną'' wartością wyrażenia jest włączany do programu. Przez „tekst'' rozumie się tutaj dowolny materiał, łącznie z tymi wierszami preprocesora, które nie są częścią danej konstrukcji warunkowej; tekst ten może być pusty. Jeśli już znaleziono „pozytywny'' wiersz #if lub #elif i obsłużono jego tekst, to kolejne wiersze #elif i #else wraz z ich tekstami są pomijane. Jeśli wszystkie wyrażenia są równe zero i występuje wiersz #else, to obsługuje się tekst następujący po #else. Tekst kontrolowany przez nieaktywne (negatywne) ramiona konstrukcji warunkowej ignoruje się, ale przegląda w poszukiwaniu zagnieżdżonych konstrukcji warunkowych.

Wyrażenia stałe występujące w #if i #elif są przedmiotem zwykłych makrorozwinięć. Ponadto, każde wyrażenie mające postać

    defined *identyfikator*

lub

    defined (*identyfikator*)

przed rozwijaniem makr zastępuje się przez 1L – jeśli identyfikator jest zdefiniowany w preprocesorze, lub przez 0L – jeśli nie jest. Wszystkie identyfikatory pozostałe w takim wyrażeniu po makrorozwinięciach zastępuje się przez 0L. Na koniec, wszystkie stałe całkowite traktuje się tak, jakby miały przyrostek L, zatem całą arytmetykę przeprowadza się na liczbach długich lub długich bez znaku.

Wynikowe wyrażenie stałe (p. A7.19) ma ograniczenia: musi być całkowite i nie może zawierać operatora sizeof, rzutowania i stałych wyliczeń.

Wiersze sterujące o postaci

    #ifdef *identyfikator*
    #ifndef *identyfikator*

odpowiadają wierszom

if defined *identyfikator*
if ! defined *identyfikator*

> Wiersz sterujący #elif nie występował w pierwszym wydaniu, chociaż był dostępny w kilku preprocesorach. Operator preprocesora defined jest także nowy.

A12.6 Numeracja wierszy

Dla potrzeb innych preprocesorów, które generują programy w języku C, wiersz mający jedną z postaci

line *stała "nazwa-pliku"*
line *stała*

zleca kompilatorowi, aby dla celów diagnostycznych przyjął, że następny źródłowy wiersz będzie miał numer podany stałą, a nazwą bieżącego pliku źródłowego będzie *nazwa-pliku*. Jeśli pominięto nazwę pliku (wraz ze znakami cudzysłowu), to pamiętana poprzednia nazwa pliku pozostanie nie zmieniona. Makra występujące w takich wierszach są rozwijane przed interpretacją instrukcji.

A12.7 Generowanie błędów

Wiersz sterujący o postaci

error $ciąg\text{-}leksemów_{opc}$

zleca preprocesorowi wypisanie komunikatu diagnostycznego zawierającego podany ciąg leksemów.

A12.8 Instrukcja **pragma**

Wiersz sterujący o postaci

pragma $ciąg\text{-}leksemów_{opc}$

zleca preprocesorowi podjęcie akcji zależnej od implementacji. Nieznana akcja jest ignorowana.

A12.9 Pusta instrukcja preprocesora

Wiersz zawierający jedynie znak

```
#
```

nie ma żadnego skutku.

A12.10 Nazwy zdefiniowane w preprocesorze

Istnieje kilka nazw, które są zdefiniowane w preprocesorze i służą do produkowania specjalnych informacji. Definicji tych nazw, jak również operatora **defined** (występującego w wyrażeniach preprocesora), nie można odwołać ani zmienić.

__LINE__ Dziesiętna stała całkowita zawierająca numer bieżącego wiersza programu źródłowego.

__FILE__ Stała napisowa zawierająca nazwę tłumaczonego pliku.

__DATE__ Stała napisowa zawierająca datę tłumaczenia programu; jej format – "Mmm dd rrrr".

__TIME__ Stała napisowa zawierająca czas tłumaczenia programu; jej format – "gg:mm:ss".

__STDC__ Stała 1. Z zamierzenia identyfikator ten powinien być zdefiniowany z wartością 1 jedynie w implementacjach dostosowanych do standardu.

> Wiersze sterujące **#error** i **#pragma** pojawiły się wraz z ANSI C; nazwy zdefiniowane w preprocesorze są nowe, ale niektóre były już dostępne w kilku implementacjach.

A13 Gramatyka

Poniższa gramatyka jest podsumowaniem składni podanej w poprzednich punktach tego dodatku. Zawartość jest dokładnie taka sama, ale zmieniono kolejność reguł składniowych.

W gramatyce występują nie zdefiniowane symbole terminalne: *stała-całkowita*, *stała-znakowa*, *napis* i *stała-wyliczenia*; słowa i symbole zapisane pismem **specjalnym** są podanymi dosłownie symbolami terminalnymi. Gramatykę tę można mechanicznie przekształcić do tekstu akceptowalnego przez automatyczne generatory analizatorów składniowych. Oprócz dodania oznaczenia wymaganego dla alternatyw w produkcjach składniowych, należy również rozwinąć konstrukcję „jeden z" i (jeśli tego wy-

maga generator analizatorów składniowych) powtórzyć te produkcje, które zawierają symbol $_{opc}$ – raz z opcjonalną konstrukcją i raz bez niej. Z jedną dodatkową zmianą, polegającą na usunięciu produkcji *nazwa-typedef: identyfikator* i zdefiniowaniu symbolu *nazwa-typedef* jako symbolu terminalnego, gramatyka ta jest akceptowalna przez generator analizatorów składniowych YACC. Gramatyka ta zawiera tylko jeden konflikt, spowodowany niejednoznacznością if-else.

jednostka-tłumaczenia:
    deklaracja-zewnętrzna
    jednostka-tłumaczenia deklaracja-zewnętrzna

deklaracja-zewnętrzna:
    definicja-funkcji
    deklaracja

definicja-funkcji:
    specyfikatory-deklaracji$_{opc}$ *deklarator lista-deklaracji*$_{opc}$ *instrukcja-złożona*

deklaracja:
    specyfikatory-deklaracji inicjowana-lista-deklaratorów$_{opc}$;

lista-deklaracji:
    deklaracja
    lista-deklaracji deklaracja

specyfikatory-deklaracji:
    specyfikator-klasy-pamięci specyfikatory-deklaracji$_{opc}$
    specyfikator-typu specyfikatory-deklaracji$_{opc}$
    kwalifikator-typu specyfikatory-deklaracji$_{opc}$

specyfikator-klasy-pamięci: jeden z
    auto register static extern typedef

specyfikator-typu: jeden z
    void char short int long float double signed unsigned
    specyfikator-struktury-unii specyfikator-wyliczenia nazwa-typedef

kwalifikator-typu: jeden z
    const volatile

specyfikator-struktury-unii:
    struktura-unia identyfikator$_{opc}$ { *lista-deklaracji-składowych* }
    struktura-unia identyfikator

struktura-unia: jeden z
    struct union

lista-deklaracji-składowych:
    deklaracja-składowych
    lista-deklaracji-składowych deklaracja-składowych

inicjowana-lista-deklaratorów:
    inicjowany-deklarator
    inicjowana-lista-deklaratorów , inicjowany-deklarator

inicjowany-deklarator:
    deklarator
    deklarator = inicjator

deklaracja-składowych:
    lista-spec-kwal lista-deklaratorów-składowych ;

lista-spec-kwal:
    specyfikator-typu lista-spec-kwal$_{opc}$
    kwalifikator-typu lista-spec-kwal$_{opc}$

lista-deklaratorów-składowych:
    deklarator-składowej
    lista-deklaratorów-składowych , deklarator-składowej

deklarator-składowej:
    deklarator
    deklarator$_{opc}$: *wyrażenie-stałe*

specyfikator-wyliczenia:
    enum *identyfikator*$_{opc}$ { *lista-wyliczników* }
    enum *identyfikator*

lista-wyliczników:
    wylicznik
    lista-wyliczników , wylicznik

wylicznik:
    identyfikator
    identyfikator = wyrażenie-stałe

deklarator:
wskaźnik$_{opc}$ *bezpośredni-deklarator*

bezpośredni-deklarator:
identyfikator
(*deklarator*)
bezpośredni-deklarator [*wyrażenie-stałe*$_{opc}$]
bezpośredni-deklarator (*lista-typów-parametrów*)
bezpośredni-deklarator (*lista-identyfikatorów*$_{opc}$)

wskaźnik:
* *lista-kwalifikatorów-typu*$_{opc}$
* *lista-kwalifikatorów-typu*$_{opc}$ *wskaźnik*

lista-kwalifikatorów-typu:
kwalifikator-typu
lista-kwalifikatorów-typu kwalifikator-typu

lista-typów-parametrów:
lista-parametrów
lista-parametrów , ...

lista-parametrów:
deklaracja-parametru
lista-parametrów , *deklaracja-parametru*

deklaracja-parametru:
specyfikatory-deklaracji deklarator
specyfikatory-deklaracji deklarator-abstrakcyjny$_{opc}$

lista-identyfikatorów:
identyfikator
lista-identyfikatorów , *identyfikator*

inicjator:
wyrażenie-przypisania
{ *lista-inicjatorów* }
{ *lista-inicjatorów* , }

lista-inicjatorów:
inicjator
lista-inicjatorów , *inicjator*

nazwa-typu:
lista-spec-kwal deklarator-abstrakcyjny$_{opc}$

deklarator-abstrakcyjny:
wskaźnik
wskaźnik$_{opc}$ *bezp-deklarator-abstrakcyjny*

bezp-deklarator-abstrakcyjny:
(*deklarator-abstrakcyjny*)
bezp-deklarator-abstrakcyjny$_{opc}$ [*wyrażenie-stałe*$_{opc}$]
bezp-deklarator-abstrakcyjny$_{opc}$ (*lista-typów-parametrów*$_{opc}$)

nazwa-typedef:
identyfikator

instrukcja:
instrukcja-etykietowana
instrukcja-wyrażeniowa
instrukcja-złożona
instrukcja-wyboru
instrukcja-powtarzania
instrukcja-skoku

instrukcja-etykietowana:
identyfikator : *instrukcja*
case *wyrażenie-stałe* : *instrukcja*
default : *instrukcja*

instrukcja-wyrażeniowa:
wyrażenie$_{opc}$;

instrukcja-złożona:
{ *lista-deklaracji*$_{opc}$ *lista-instrukcji*$_{opc}$ }

lista-instrukcji:
instrukcja
lista-instrukcji instrukcja

instrukcja-wyboru:
if (*wyrażenie*) *instrukcja*
if (*wyrażenie*) *instrukcja* else *instrukcja*
switch (*wyrażenie*) *instrukcja*

instrukcja-powtarzania:
while (*wyrażenie*) *instrukcja*
do *instrukcja* while (*wyrażenie*) ;
for ($wyrażenie_{opc}$; $wyrażenie_{opc}$; $wyrażenie_{opc}$) *instrukcja*

instrukcja-skoku:
goto *identyfikator* ;
continue ;
break ;
return $wyrażenie_{opc}$;

wyrażenie:
wyrażenie-przypisania
wyrażenie , *wyrażenie-przypisania*

wyrażenie-przypisania:
wyrażenie-warunkowe
wyrażenie-jednoargumentowe operator-przypisania wyrażenie-przypisania

operator-przypisania: jeden z
= *= /= %= += -= <<= >>= &= ^= |=

wyrażenie-warunkowe:
logiczne-wyrażenie-OR
logiczne-wyrażenie-OR ? *wyrażenie* : *wyrażenie-warunkowe*

wyrażenie-stałe:
wyrażenie-warunkowe

logiczne-wyrażenie-OR:
logiczne-wyrażenie-AND
logiczne-wyrażenie-OR || *logiczne-wyrażenie-AND*

logiczne-wyrażenie-AND:
wyrażenie-OR
logiczne-wyrażenie-AND && *wyrażenie-OR*

wyrażenie-OR:
wyrażenie-XOR
wyrażenie-OR | *wyrażenie-XOR*

wyrażenie-XOR:
wyrażenie-AND
wyrażenie-XOR ^ *wyrażenie-AND*

wyrażenie-AND:
wyrażenie-przyrównania
wyrażenie-AND & *wyrażenie-przyrównania*

wyrażenie-przyrównania:
wyrażenie-relacyjne
wyrażenie-przyrównania == *wyrażenie-relacyjne*
wyrażenie-przyrównania != *wyrażenie-relacyjne*

wyrażenie-relacyjne:
wyrażenie-przesunięcia
wyrażenie-relacyjne < *wyrażenie-przesunięcia*
wyrażenie-relacyjne > *wyrażenie-przesunięcia*
wyrażenie-relacyjne <= *wyrażenie-przesunięcia*
wyrażenie-relacyjne >= *wyrażenie-przesunięcia*

wyrażenie-przesunięcia:
wyrażenie-addytywne
wyrażenie-przesunięcia << *wyrażenie-addytywne*
wyrażenie-przesunięcia >> *wyrażenie-addytywne*

wyrażenie-addytywne:
wyrażenie-multyplikatywne
wyrażenie-addytywne + *wyrażenie-multyplikatywne*
wyrażenie-addytywne – *wyrażenie-multyplikatywne*

wyrażenie-multyplikatywne:
wyrażenie-rzutowania
wyrażenie-multyplikatywne * *wyrażenie-rzutowania*
wyrażenie-multyplikatywne / *wyrażenie-rzutowania*
wyrażenie-multyplikatywne % *wyrażenie-rzutowania*

wyrażenie-rzutowania:
wyrażenie-jednoargumentowe
(*nazwa-typu*) *wyrażenie-rzutowania*

wyrażenie-jednoargumentowe:
wyrażenie-przyrostkowe
++ *wyrażenie-jednoargumentowe*
-- *wyrażenie-jednoargumentowe*
operator-jednoargumentowy wyrażenie-rzutowania
sizeof *wyrażenie-jednoargumentowe*
sizeof (*nazwa-typu*)

operator-jednoargumentowy: jeden z
& * + - ~ !

wyrażenie-przyrostkowe:
wyrażenie-proste
wyrażenie-przyrostkowe [*wyrażenie*]
wyrażenie-przyrostkowe (*lista-argumentów*$_{opc}$)
wyrażenie-przyrostkowe . *identyfikator*
wyrażenie-przyrostkowe -> *identyfikator*
wyrażenie-przyrostkowe ++
wyrażenie-przyrostkowe --

wyrażenie-proste:
identyfikator
stała
napis
(*wyrażenie*)

lista-argumentów:
wyrażenie-przypisania
lista-argumentów , *wyrażenie-przypisania*

stała:
stała-całkowita
stała-znakowa
stała-zmiennopozycyjna
stała-wyliczenia

Poniższa gramatyka podsumowuje budowę wierszy sterujących preprocesora, nie nadaje się jednak do mechanicznej analizy. Zawiera symbol *tekst*, który oznacza zwykły tekst programu, bezwarunkowe wiersze sterujące preprocesora lub pełne warunkowe konstrukcje preprocesora.

wiersz-sterujący:
define *identyfikator ciąg-leksemów*
define *identyfikator* (*lista-identyfikatorów*) *ciąg-leksemów*
undef *identyfikator*
include <*nazwa-pliku*>
include "*nazwa-pliku*"
include *ciąg-leksemów*
line *stała* "*nazwa-pliku*"
line *stała*
error *ciąg-leksemów*$_{opc}$
pragma *ciąg-leksemów*$_{opc}$
#
kompilacja-warunkowa

kompilacja-warunkowa:
wiersz-if tekst części-elif część-else$_{opc}$ # endif

wiersz-if:
if *wyrażenie-stałe*
ifdef *identyfikator*
ifndef *identyfikator*

części-elif:
wiersz-elif tekst
części-elif$_{opc}$

wiersz-elif:
elif *wyrażenie-stałe*

część-else:
wiersz-else tekst

wiersz-else:
else

DODATEK B BIBLIOTEKA STANDARDOWA

Niniejszy dodatek zawiera podsumowanie wiadomości o bibliotece zdefiniowanej przez standard ANSI. Biblioteka standardowa nie jest częścią samego języka, natomiast pochodzą z niej deklaracje funkcji oraz definicje typów i makr oferowanych przez standardowe środowisko programów w C. W poniższym opisie opuściliśmy kilka funkcji, które mają ograniczone zastosowanie lub łatwo dają się wyprowadzić z innych; pominęliśmy wielobajtowe znaki; pominęliśmy również kwestie regionalne, tzn. właściwości zależne od miejscowego języka, narodowości czy kultury.

Funkcje, typy i makra należące do standardowej biblioteki są zdefiniowane w standardowych *plikach nagłówkowych* (*nagłówkach*):

<assert.h>	<float.h>	<math.h>	<stdarg.h>	<stdlib.h>
<ctype.h>	<limits.h>	<setjmp.h>	<stddef.h>	<string.h>
<errno.h>	<locale.h>	<signal.h>	<stdio.h>	<time.h>

Dostęp do pliku nagłówkowego realizuje instrukcja preprocesora

```
#include <nagłówek>
```

Nagłówki mogą być dołączane w dowolnej kolejności i dowolną liczbę razy. Nagłówek musi być dołączony na zewnątrz jakiejkolwiek zewnętrznej deklaracji czy definicji i przed użyciem czegokolwiek, co jest w nim zdefiniowane. Nagłówek nie musi być plikiem źródłowym*.

Identyfikatory zewnętrzne rozpoczynające się od znaku podkreślenia są zarezerwowane na użytek biblioteki; to samo dotyczy wszelkich innych identyfikatorów rozpoczynających się od znaku podkreślenia i wielkiej litery lub dwóch znaków podkreślenia.

* Niektóre kompilatory pozwalają na wstępne przetłumaczenie nagłówków i włączanie ich do programów już w pewnej postaci pośredniej. – *Przyp. tłum.*

B1 Wejście i wyjście: nagłówek <stdio.h>

Funkcje wejścia i wyjścia, typy i makra zdefiniowane w <stdio.h> stanowią prawie jedną trzecią zawartości biblioteki.

Strumień jest źródłem lub celem danych i może być skojarzony z dyskiem lub innym urządzeniem zewnętrznym. Biblioteka obsługuje strumienie tekstowe i binarne, jakkolwiek w niektórych systemach operacyjnych, w szczególności w systemie Unix, są one identyczne. Strumień tekstowy jest ciągiem wierszy; każdy wiersz zawiera zero lub więcej znaków i kończy się znakiem '\n'. W pewnych środowiskach programów strumienie tekstowe mogą wymagać przekształcenia do jakiejś innej postaci (np. przez zastąpienie znaków '\n' znakami powrotu karetki i nowego wiersza). Strumień binarny jest ciągiem nie przetworzonych bajtów reprezentujących wewnętrzne dane programu, mającym tę właściwość, że zapisany i ponownie w tym samym systemie odczytany, pozostaje bez zmian.

Strumień wiąże się z plikiem lub urządzeniem za pomocą *otwarcia*; to połączenie przerywa się *zamykając* strumień. Otwarcie pliku udostępnia wskaźnik do obiektu typu FILE (plik), w którym gromadzi się wszelkie informacje niezbędne do obsługi strumienia. Tam, gdzie to nie będzie prowadzić do nieporozumień, będziemy wymiennie używali określeń „wskaźnik do pliku" i „strumień".

Gdy program rozpoczyna działanie, trzy strumienie są już otwarte: stdin (standardowe wejście), stdout (standardowe wyjście) i stderr (standardowe wyjście błędów).

B1.1 Operacje na plikach

Następujące funkcje operują na plikach (typ size_t jest typem wartości całkowitej bez znaku, otrzymywanej za pomocą operatora sizeof):

FILE *fopen(const char *filename, const char *mode)

Funkcja fopen otwiera wskazany argumentem filename plik i zwraca strumień lub NULL, jeśli próba otwarcia się nie powiedzie. Argument mode określa rodzaj dostępu do pliku; poniższe zestawienie zawiera jego dopuszczalne wartości:

"r"	otwórz plik tekstowy do czytania;
"w"	otwórz plik tekstowy do pisania; skasuj poprzednią zawartość, jeśli istnieje;
"a"	dopisuj; otwórz lub utwórz plik tekstowy do dopisywania na końcu pliku;
"r+"	otwórz plik tekstowy do aktualizacji (tj. czytania i pisania);
"w+"	utwórz plik tekstowy do aktualizacji; skasuj poprzednią zawartość, jeśli istnieje;
"a+"	dopisuj; otwórz lub utwórz plik tekstowy do aktualizacji, dopisując na końcu pliku.

Aktualizacja oznacza możliwość jednoczesnego pisania do i czytania z tego samego pliku; między czytaniem i pisaniem (lub odwrotnie) należy wywołać funkcję fflush lub funkcje pozycjonujące plik. Jeśli w argumencie mode po początkowej literze występuje litera b, np. "rb" lub "w+b", oznacza to plik binarny. Nazwy plików są ograniczone do FILENAME_MAX znaków. Jednocześnie można otworzyć co najwyżej FOPEN_MAX plików.

FILE *freopen(const char *filename, const char *mode, FILE *stream)
Funkcja freopen otwiera wskazany plik (filename) przy wskazanym rodzaju dostępu (mode) i wiąże z nim strumień. Funkcja zwraca wartość stream lub NULL w przypadku błędu. Funkcji freopen zwykle używa się do zmiany plików związanych ze strumieniami stdin, stdout i stderr.

int fflush(FILE *stream)
Dla strumieni wyjściowych funkcja fflush powoduje wypisanie wszelkich zbuforowanych, lecz jeszcze nie wypisanych, danych; dla strumieni wejściowych efekt nie jest zdefiniowany. Funkcja zwraca EOF w przypadku błędu pisania, a zero w pozostałych przypadkach. Wywołanie fflush(NULL) wypisuje dane ze wszystkich strumieni wyjściowych.

int fclose(FILE *stream)
Funkcja fclose wypisuje wszystkie nie wypisane dane związane ze strumieniem stream, kasuje wszystkie nie przeczytane dane z buforów wejściowych, zwalnia automatycznie przydzielone bufory i na koniec zamyka plik. Funkcja zwraca EOF w przypadku jakiegoś błędu, a zero w pozostałych przypadkach.

int remove(const char *filename)
Funkcja remove usuwa wskazany plik tak, że późniejsza próba otworzenia go zakończy się niepowodzeniem. Funkcja zwraca wartość różną od zera, jeśli usunięcie pliku się nie powiedzie.

int rename(const char *oldname, const char *newname)
Funkcja rename zmienia nazwę pliku (z oldname na newname); zwraca wartość różną od zera, jeśli ta zmiana się nie powiedzie.

FILE *tmpfile(void)
Funkcja tmpfile tworzy plik tymczasowy z rodzajem dostępu "wb+"; plik ten zostanie automatycznie usunięty po zamknięciu lub po normalnym zakończeniu programu. Funkcja zwraca strumień lub NULL, gdy nie jest w stanie utworzyć pliku.

char *tmpnam(char s[L_tmpnam])
Wywołanie tmpnam(NULL) tworzy nazwę (ciąg znaków), która nie jest nazwą żadnego z istniejących plików, i zwraca wskaźnik do tej nazwy w wewnętrznej, sta-

tycznej tablicy. Wywołanie tmpnam(s) wstawia tę nazwę do argumentu s i zwraca wartość s. W tablicy s musi być miejsce przynajmniej dla L_tmpnam znaków. Funkcja tmpnam przy każdym wywołaniu tworzy inną nazwę; gwarantuje się co najwyżej TMP_MAX różnych nazw podczas wykonywania programu. Zwróć uwagę na to, że funkcja tworzy nazwę, a nie plik.

int setvbuf(FILE *stream, char *buf, int mode, size_t size)
Funkcja setvbuf kontroluje buforowanie danych strumienia; musi być wywołana przed czytaniem, pisaniem lub jakąkolwiek inną operacją. Wartość _IOFBF argumentu mode oznacza pełne buforowanie, _IOLBF – buforowanie wierszy pliku tekstowego, a _IONBF – brak buforowania. Jeśli argument buf jest różny od NULL, to buforem będzie wskazany obszar; w przeciwnym przypadku bufor zostanie przydzielony. Argument size określa rozmiar bufora. W przypadku błędu funkcja setvbuf zwraca wartość różną od zera.

void setbuf (FILE *stream, char *buf)
Jeśli argument buf jest równy NULL, to buforowanie strumienia stream zostanie wyłączone. W przeciwnym przypadku setbuf jest równoważne
(void) setvbuf(stream, buf, _IOFBF, BUFSIZ).

B1.2 Formatowane wyjście

Funkcje z rodziny printf realizują formatowane przekształcanie danych wyjściowych.

int fprintf(FILE *stream, const char *format, ...)
Funkcja fprintf, pod kontrolą argumentu format, przekształca i wypisuje pozostałe argumenty do strumienia wyjściowego stream. Wartość zwracana przez funkcję równa się liczbie wypisanych znaków lub jest liczbą ujemną w przypadku błędu.

Format zawiera znaki dwojakiego rodzaju: zwykłe znaki, które są bezpośrednio kopiowane do strumienia, i specyfikacje przekształcenia, które wskazują sposób przekształcenia i wypisania kolejnych argumentów funkcji. Specyfikacja przekształcenia rozpoczyna się znakiem % i kończy znakiem przekształcenia. Między % i znakiem przekształcenia mogą wystąpić (w podanej kolejności):

- Modyfikatory (w dowolnej kolejności), które wpływają na postać wyniku:
  - – dosunięcie przekształconego argumentu do lewego krańca jego pola;
  - \+ wypisanie liczby zawsze ze znakiem;
  - *odstęp* poprzedzenie wyniku znakiem odstępu, jeśli jego pierwszym znakiem nie jest plus lub minus;

0 dla przekształceń liczbowych: uzupełnienie liczby wiodącymi zerami do pełnego rozmiaru pola;

alternatywna postać wyniku: dla formatu o pierwszą wypisaną cyfrą będzie 0; dla formatu x lub X wynik różny od zera poprzedza się 0x lub 0X; dla formatów e, E, f, g i G wypisana liczba zawsze będzie miała kropkę dziesiętną (dla g i G nieznaczące zera nie zostaną usunięte).

- Liczba określająca minimalny rozmiar pola. Przekształcony argument zostanie wypisany w polu o rozmiarze nie mniejszym od wskazanego (jeśli to konieczne – o większym). Jeśli przekształcony argument jest krótszy od rozmiaru pola, to zostanie dopełniony z lewej strony (lub z prawej, jeśli zlecono dosunięcie w lewo) do pełnego rozmiaru pola. Znakiem wypełniającym jest zwykle odstęp, chyba że polecono dopełnianie zerami – wówczas jest nim znak 0.

- Kropka oddzielająca rozmiar pola od precyzji.

- Liczba (precyzja) określająca: maksymalną liczbę wypisywanych znaków tekstu; liczbę cyfr wypisywanych po kropce dziesiętnej w specyfikacjach e, E lub f; liczbę cyfr znaczących w specyfikacjach g lub G; minimalną liczbę cyfr wypisywanych dla argumentu całkowitego (żądany rozmiar pola osiąga się przez dopisanie wiodących zer).

- Modyfikator długości h, l (litera l) lub L. Litera h wskazuje, że odpowiedni argument jest typu short lub unsigned short ; litera l wskazuje, że argument jest typu long lub unsigned long; litera L wskazuje, że argument jest typu long double.

Zamiast liczby określającej rozmiar pola lub precyzję (lub obie) można podać znak *. W tym przypadku żądaną wartość oblicza się na podstawie następnego argumentu (lub następnych argumentów) funkcji. Argument ten musi mieć typ int.

Znaki przekształcenia i ich znaczenie podano w tabl. B-1. Jeśli znak następujący po znaku % nie jest znakiem przekształcenia, to skutek takiej specyfikacji nie jest zdefiniowany.

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

Wywołanie printf(...) jest równoważne z wywołaniem fprintf(stdout,...).

int sprintf(char *s, const char *format, ...)

Funkcja sprintf działa tak samo, jak printf, przy czym wynik (zakończony znakiem '\0') jest zapisywany do tablicy s; tablica s musi być dostatecznie duża, żeby pomieścić wynik. Zwracana przez funkcję wartość (liczba wypisanych znaków) nie uwzględnia znaku '\0'.

Tablica B-1. Przekształcenia funkcji **printf**

Znak	Typ argumentu	Przekształcany do postaci
d, i	int	liczby dziesiętnej ze znakiem.
o	int	liczby ósemkowej bez znaku (bez wiodącego zera).
x, X	int	liczby szesnastkowej bez znaku (bez wiodącego 0x lub 0X) z użyciem liter abcdef dla 0x i ABCDEF dla 0X.
u	int	liczby dziesiętnej bez znaku.
c	int	pojedynczego znaku po przekształceniu do typu unsigned char.
s	char *	Znaki tekstu są wypisywane aż do napotkania znaku '\0' lub liczba wypisanych znaków osiągnie wskazaną precyzję.
f	double	liczby dziesiętnej [−]*mmm.ddd*, gdzie liczbę cyfr *d* określa precyzja. Domyślną precyzją jest 6; przy precyzji 0 opuszcza się kropkę dziesiętną.
e, E	double	liczby dziesiętnej [−]*m.dddddd*e±*xx* lub [−]*m.dddddd*E±*xx*, gdzie liczbę cyfr *d* określa precyzja. Domyślną precyzją jest 6; przy precyzji 0 opuszcza się kropkę dziesiętną.
g, G	double	Jeśli wykładnik potęgi jest mniejszy niż −4 lub większy lub równy precyzji, to stosuje się specyfikację %e lub %E; w przeciwnym przypadku %f. Nie wypisuje się nie znaczących zer i zbędnej kropki dziesiętnej.
p	void *	wskaźnika (reprezentacja zależy od implementacji).
n	int *	Liczbę dotychczas wypisanych znaków w tym wywołaniu funkcji printf *zapisuje się* do odpowiedniego argumentu. Nie ma żadnego przekształcenia argumentu.
%		Nie ma żadnego przekształcania argumentu – zostanie wypisany znak %.

vprintf(const char *format, va_list arg)
vfprintf(FILE *stream, const char *format, va_list arg)
vsprintf(char *s, const char *format, va_list arg)

Funkcje vprintf, vfprintf i vsprintf działają tak samo, jak odpowiednie funkcje printf z tym, że zamiast listy argumentów o zmiennej długości podaje się argument arg, który został zainicjowany wywołaniem makra va_start i, być może, wywołaniami va_arg (patrz: rozważania nad <stdarg.h> w p. B7).

B1.3 Formatowane wejście

Funkcje z rodziny scanf realizują formatowane przekształcanie danych wejściowych.

int fscanf(FILE *stream, const char *format, ...)

Funkcja fscanf czyta strumień stream i pod kontrolą argumentu format przypisuje przekształcone wartości kolejnym argumentom. *Każdy z tych argumentów musi być*

wskaźnikiem. Funkcja kończy działanie, gdy zinterpretuje cały format. Funkcja zwraca EOF, jeśli przed jakimkolwiek przekształceniem napotka koniec pliku lub nastąpi jakiś błąd; w przeciwnym przypadku zwraca liczbę przekształconych i przypisanych danych wejściowych.

Format zwykle zawiera specyfikacje przekształceń, które sterują interpretacją wejściowego ciągu znaków. W formacie mogą wystąpić:

- Odstępy i tabulacje, które są ignorowane;
- Zwykłe znaki (nie %) spodziewane jako następne „czarne'' znaki w strumieniu wejściowym;
- Specyfikacje przekształceń złożone ze znaku %, opcjonalnego znaku * wstrzymującego przypisanie, opcjonalnej liczby określającej maksymalny rozmiar pola, opcjonalnej litery h, l lub L określającej rozmiar odpowiedniego argumentu oraz ze znaku przekształcenia.

Specyfikacja przekształcenia steruje przekształceniem kolejnego pola wejściowego. Zwykle wynik jest przypisywany zmiennej wskazanej odpowiednim argumentem. Jeśli jednak przypisanie ma być wstrzymane, jak np. w %*s, to dane pole wejściowe będzie po prostu pominięte – nie wykona się żadne przypisanie. Pole wejściowe jest zdefiniowane jako ciąg czarnych znaków; rozciąga się ono albo do najbliższej „białej plamy'' (ciąg białych znaków), albo – jeśli podano rozmiar pola – na odległość wskazaną tym rozmiarem. Wynika z tego, że funkcja scanf w poszukiwaniu swoich pól będzie przekraczać granice wierszy, ponieważ znak nowego wiersza jest białym znakiem. (Do białych znaków należą: znak odstępu, tabulacji, nowego wiersza, powrotu karetki, tabulacji pionowej i znak nowej strony.)

Znak przekształcenia określa sposób interpretacji najbliższego pola wejściowego. Odpowiedni argument musi być wskaźnikiem. Dozwolone znaki przekształcenia przedstawiono w tabl. B-2.

Znaki przekształcenia d, i, n, o, u i x można poprzedzić literą h, jeśli argument jest wskaźnikiem do obiektów typu short , a nie int, lub literą l, jeśli argument jest wskaźnikiem do obiektów typu long. Znaki przekształcenia e, f i g mogą być poprzedzone literą l, jeśli na liście argumentów znajduje się wskaźnik do obiektów typu double, a nie float, lub literą L dla typu long double.

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

Wywołanie scanf(...) jest równoważne z wywołaniem fscanf(stdin,...).

int sscanf(char *s, const char *format, ...)

Funkcja sscanf odpowiada funkcji scanf z tym, że wejściowe znaki są pobierane z tablicy znakowej s.

Tablica B-2. Przekształcenia funkcji **scanf**

Znak	Dana wejściowa	Typ argumentu
d	Liczba całkowita dziesiętna.	int *
i	Liczba całkowita; może być w postaci ósemkowej (wiodące zero) lub szesnastkowej (wiodące 0x lub 0X).	int *
o	Liczba całkowita ósemkowa (z wiodącym zerem lub bez).	int *
u	Liczba całkowita dziesiętna bez znaku.	unsigned int *
x	Liczba całkowita szesnastkowa (z wiodącym 0x lub 0X lub bez).	int *
c	Znaki; kolejne wejściowe znaki zostaną wstawione do wskazanej tablicy. Liczbę wstawionych znaków określa rozmiar pola (domyślnie 1). Nie dodaje się znaku '\0'. W tym przypadku nie obowiązuje zwykłe pomijanie białych znaków; aby przeczytać najbliższy czarny znak użyj %1s.	char *
s	Ciąg czarnych znaków (bez cudzysłowów). Tablica podana jako argument musi być dostatecznie duża, aby pomieścić cały ciąg razem z dodanym na końcu znakiem '\0'.	char *
e, f, g	Liczba zmiennopozycyjna. Wejściowy format dla liczb zmiennopozycyjnych zawiera opcjonalny znak, ciąg cyfr ewentualnie rozdzielony kropką dziesiętną oraz opcjonalny wykładnik potęgi, składający się z litery E lub e i liczby całkowitej ewentualnie ze znakiem.	float *
p	Wartość wskaźnika w takiej postaci, w jakiej jest wypisywany przez printf("%p").	void *
n	Wpisuje do argumentu liczbę dotychczas przeczytanych znaków w tym wywołaniu funkcji. Nie czyta żadnych wejściowych znaków. Nie zwiększa licznika przeczytanych pól.	int *
[...]	Odpowiada najdłuższemu niepustemu ciągowi wejściowych znaków ze zbioru znaków zawartych między nawiasami. Na końcu jest dopisywany znak '\0'. Napis []...] włącza znak] do zbioru.	char *
[^...]	Odpowiada najdłuższemu niepustemu ciągowi wejściowych znaków *nie* należących do zbioru znaków zawartych między nawiasami. Na końcu jest dopisywany znak '\0'. Napis [^]...] włącza znak] do zbioru.	char *
%	Znak %. Nie ma żadnego przypisania.	

B1.4 Funkcje realizujące wejście i wyjście znakowe

int fgetc(FILE *stream)

Funkcja fgetc czyta ze strumienia stream następny znak i zwraca jego wartość potraktowaną jako unsigned char (i przekształconą do int); w przypadku napotkania końca pliku albo wystąpienia błędu funkcja zwraca EOF.

char *fgets(char *s, int n, FILE *stream)
Funkcja fgets czyta co najwyżej n–1 znaków i wstawia do tablicy s. Funkcja przerywa czytanie po napotkaniu znaku nowego wiersza; znak ten także wstawia do tablicy. Cały tekst jest zakończony znakiem '\0'. Funkcja zwraca s lub NULL w przypadku napotkania końca pliku albo błędu.

int fputc(int c, FILE *stream)
Funkcja fputc wpisuje znak c (przekształcony do unsigned char) do strumienia stream. Zwraca wypisany znak lub EOF w przypadku błędu.

int fputs(const char *s, FILE *stream)
Funkcja fputs wypisuje tekst zawarty w tablicy s (nie musi zawierać '\n') do strumienia stream. Zwraca liczbę nieujemną lub EOF w przypadku błędu.

int getc(FILE *stream)
Działanie getc jest równoważne z działaniem funkcji fgetc, przy czym jeśli getc zrealizowano jako makro, to argument stream może być obliczany więcej niż jeden raz.

int getchar(void)
Wywołanie getchar jest równoważne z wywołaniem getc(stdin).

char *gets(char *s)
Funkcja gets wczytuje z wejścia następny wiersz i wpisuje go do tablicy s. Końcowy znak nowego wiersza zastępuje znakiem '\0'. Funkcja zwraca s lub NULL w przypadku napotkania końca pliku albo wystąpienia błędu.

int putc(int c, FILE *stream)
Działanie putc jest równoważne z działaniem funkcji fputc, czy czym jeśli putc zrealizowano jako makro, to argument stream może być obliczany więcej niż jeden raz.

int putchar(int c)
Wywołanie putchar(c) jest równoważne z wywołaniem putc(c, stdout).

int puts(const char *s)
Funkcja puts wypisuje tekst zawarty w tablicy s oraz znak nowego wiersza do strumienia stdout. Zwraca liczbę nieujemną lub EOF w przypadku błędu.

int ungetc(int c, FILE *stream)
Funkcja ungetc wycofuje znak c (przekształcony do unsigned char) z powrotem do strumienia stream; znak ten zostanie przeczytany ponownie przy najbliższym

czytaniu ze strumienia. Dla każdego strumienia można wycofać tylko jeden znak. Nie można wycofać znacznika końca pliku EOF. Funkcja zwraca wycofany znak lub EOF w przypadku błędu.

B1.5 Funkcje realizujące bezpośrednie wejście i wyjście

size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)

Funkcja fread wczytuje ze strumienia stream do tablicy ptr co najwyżej nobj obiektów o rozmiarze size. Funkcja zwraca liczbę przeczytanych obiektów; ta liczba może być mniejsza od żądanej. Do zbadania stanu strumienia należy użyć funkcji feof i ferror.

size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)

Funkcja fwrite wypisuje nobj obiektów o rozmiarze size, pochodzących z tablicy ptr, do strumienia stream. Zwraca liczbę wypisanych obiektów, która w przypadku błędu jest mniejsza niż nobj.

B1.6 Funkcje wyznaczające pozycję w pliku

int fseek(FILE *stream, long offset, int origin)

Funkcja fseek wyznacza pozycję w strumieniu stream; następne czytanie lub pisanie będzie odnosić się do danych rozpoczynających się od nowej pozycji. Dla plików binarnych nowa pozycja wypada w miejscu oddalonym o offset znaków od punktu odniesienia origin, który może mieć wartości: SEEK_SET (początek pliku), SEEK_CUR (bieżąca pozycja) lub SEEK_END (koniec pliku). Dla plików tekstowych wartość offset musi być równa zeru lub wartości zwróconej przez funkcję ftell (w tym przypadku origin musi się równać SEEK_SET). Funkcja fseek zwraca wartość różną od zera w przypadku błędu.

long ftell(FILE *stream)

Funkcja ftell zwraca wartość bieżącej pozycji dla strumienia stream lub –1L w przypadku błędu.

void rewind(FILE *stream)

Wywołanie rewind(fp) jest równoważne z ciągiem wywołań
fseek(fp, 0L, SEEK_SET); clearerr(fp).

int fgetpos(FILE *stream, fpos_t *ptr)

Funkcja fgetpos zapamiętuje bieżącą pozycję strumienia stream w miejscu wskazanym przez *ptr. Z tej wartości można później skorzystać w funkcji fsetpos. Typ

fpos_t jest odpowiednim typem obiektu do przechowania takiej wartości. W przypadku błędu funkcja fgetpos zwraca wartość różną od zera.

int fsetpos(FILE *stream, const fpos_t *ptr)
Funkcja fsetpos ustawia bieżącą pozycję strumienia stream według wartości zapamiętanej przez funkcję fgetpos w miejscu wskazanym przez *ptr. W przypadku błędu funkcja fsetpos zwraca wartość różną od zera.

B1.7 Obsługa błędów

Wiele funkcji bibliotecznych ustawia znaczniki stanu dla strumieni po napotkaniu końca pliku lub gdy wystąpi błąd. Te znaczniki mogą być ustawiane i sprawdzane jawnie. Dodatkowo, wartością wyrażenia całkowitego errno (zadeklarowanego w <errno.h>) może być numer błędu, który bliżej określa ostatnio napotkany błąd.

void clearerr(FILE *stream)
Funkcja clearerr kasuje znaczniki końca pliku i błędu dla strumienia stream.

int feof(FILE *stream)
Funkcja feof zwraca wartość różną od zera w przypadku, gdy jest ustawiony znacznik końca pliku dla strumienia stream.

int ferror(FILE *stream)
Funkcja ferror zwraca wartość różną od zera w przypadku, gdy jest ustawiony znacznik błędu dla strumienia stream.

void perror(const char *s)
Wywołanie perror(s) wypisuje tekst z tablicy s i zależny od implementacji komunikat o błędzie odpowiadający wartości errno. Działanie funkcji można przedstawić jako:

fprintf(stderr, "%s: %s\n", s, "*komunikat o błędzie*")

Patrz również: strerror w p. B3.

B2 Klasyfikowanie znaków: nagłówek <ctype.h>

Nagłówek <ctype.h> zawiera deklaracje funkcji służących do klasyfikowania znaków. We wszystkich funkcjach argument c jest typu int o wartości albo równej EOF, albo wyrażalnej jako unsigned char, a wynik jest typu int. Funkcje zwracają wartość

różną od zera (prawda), jeśli argument c spełnia podany warunek albo należy do wskazanej klasy znaków; w przeciwnym przypadku zwracają zero.

isalnum(c)	prawdą jest albo isalpha(c), albo isdigit(c)
isalpha(c)	prawdą jest albo isupper(c), albo islower(c)
iscntrl(c)	znak kontrolny
isdigit(c)	cyfra dziesiętna
isgraph(c)	znak drukowalny za wyjątkiem odstępu
islower(c)	mała litera
isprint(c)	znak drukowalny łącznie z odstępem
ispunct(c)	znak drukowalny za wyjątkiem odstępu, liter i cyfr
isspace(c)	odstęp, nowa strona, nowy wiersz, powrót karetki, tabulator, pionowy tabulator
isupper(c)	wielka litera
isxdigit(c)	cyfra szesnastkowa

W siedmiobitowym zbiorze znaków ASCII znaki drukowalne mieszczą się w przedziale między 0x20 (' ') i 0x7E ('~'); do znaków kontrolnych należą znaki z przedziału między 0 (NUL) i 0x1F (US) oraz znak 0x7F (DEL).

Dwie dodatkowe funkcje służą do zmiany wielkości liter:

int tolower(int c)	zamienia c na małą literę
int toupper(int c)	zamienia c na wielką literę

Jeśli c jest wielką literą, to funkcja tolower(c) zwróci odpowiednią małą literę; w przeciwnym przypadku zwróci c. Jeśli c jest małą literą, to funkcja toupper(c) zwróci odpowiednią wielką literę; w przeciwnym przypadku zwróci c.

B3 Operacje na tekstach: nagłówek <string.h>

W nagłówku <string.h> są zdefiniowane dwie grupy funkcji operujących na tekstach. Do pierwszej grupy należą funkcje, których nazwy rozpoczynają się od str, do drugiej grupy – od mem. Z wyjątkiem funkcji memmove, skutek kopiowania obiektów, które częściowo się pokrywają, nie jest zdefiniowany. Argumenty funkcji porównujących są traktowane jak tablice obiektów typu unsigned char.

W poniższym zestawieniu argumenty s i t są typu char *, argumenty cs i ct są typu const char *, argument n jest typu size_t, a argument c jest obiektem typu int przekształconym do typu char.

char *strcpy(s,ct)	kopiuje tekst z ct do s łącznie ze znakiem '\0'; zwraca s
char *strncpy(s,ct,n)	kopiuje co najwyżej n znaków tekstu z ct do s; jeżeli ct ma mniej niż n znaków, dopełnia s znakami '\0'; zwraca s
char *strcat(s,ct)	dopisuje znaki tekstu z ct na koniec tekstu w s; zwraca s
char *strncat(s,ct,n)	dopisuje co najwyżej n znaków tekstu ct na koniec tekstu w s; tekst w s kończy znakiem '\0'; zwraca s
int strcmp(cs,ct)	porównuje teksty zawarte w cs i ct; zwraca wartość < 0 dla cs<ct, wartość 0 dla cs==ct lub wartość > 0 dla cs>ct
int strncmp(cs,ct,n)	porównuje co najwyżej n znaków tekstów zawartych w cs i ct; zwraca wartość < 0 dla cs<ct, wartość 0 dla cs==ct lub wartość > 0 dla cs>ct
char *strchr(cs,c)	zwraca wskaźnik do pierwszego wystąpienia znaku c w tekście cs lub NULL, jeśli ten znak nie występuje
char *strrchr(cs,c)	zwraca wskaźnik do ostatniego wystąpienia znaku c w tekście cs lub NULL, jeśli ten znak nie występuje
size_t strspn(cs,ct)	zwraca długość przedrostka w tekście cs składającego się ze znaków występujących w ct
size_t strcspn(cs,ct)	zwraca długość przedrostka w tekście cs składającego się ze znaków *nie* występujących w ct
char *strpbrk(cs,ct)	zwraca wskaźnik do pierwszego wystąpienia w tekście cs któregokolwiek ze znaków tekstu ct lub NULL, jeśli żaden ze znaków ct nie występuje w cs
char *strstr(cs,ct)	zwraca wskaźnik do pierwszego wystąpienia tekstu ct w tekście cs lub NULL, jeśli ct nie występuje w cs
size_t strlen(cs)	zwraca długość tekstu cs
char *strerror(n)	zwraca wskaźnik do (zależnego od implementacji) tekstu komunikatu odpowiadającego błędowi o numerze n
char *strtok(s,ct)	wyszukuje w tekście s ciągi znaków przedzielone znakami z ct – patrz niżej

Szereg wywołań funkcji strtok dzieli tekst zawarty w s na ciągi znaków rozdzielone znakami z tekstu ct. Za pierwszym razem funkcję woła się z argumentem s różnym od NULL. Funkcja znajduje w s pierwszy ciąg złożony ze znaków nie należących do ct; ogranicza go, zastępując następny znak w s znakiem '\0', i zwraca wskaźnik do początku tego ciągu. Każde następne wywołanie funkcji, tym razem z argumentem s równym NULL, zwróci kolejny taki ciąg, rozpoczynając szukanie za końcem poprzednio znalezionego ciągu. Funkcja strtok zwróci NULL, gdy nie znajdzie więcej ciągów. Przy każdym wywołaniu argument ct może być inny.

Funkcje mem... służą do operowania na obiektach traktowanych jak tablice znakowe; z założenia mają one działać bardzo sprawnie. W poniższym zestawieniu argumenty

s i t są typu void *, argumenty cs i ct są typu const void *, argument n jest typu size_t, a argument c jest obiektem typu int przekształconym do typu unsigned char.

void *memcpy(s,ct,n)	kopiuje n znaków z obiektu ct do obiektu s i zwraca s
void *memmove(s,ct,n)	robi to samo, co memcpy, z tym że działa również wtedy, kiedy obiekty zachodzą na siebie
int memcmp(cs,ct,n)	porównuje początkowe n znaków zawartych w obiektach cs i ct; zwraca taką samą wartość, jak strcmp
void *memchr(cs,c,n)	zwraca wskaźnik do pierwszego wystąpienia znaku c w obiekcie cs lub NULL, jeśli c nie występuje wśród początkowych n znaków cs
void *memset(s,c,n)	wstawia znak c do początkowych n znaków obiektu s i zwraca s

B4 Funkcje matematyczne: nagłówek <math.h>

Nagłówek <math.h> zawiera deklaracje funkcji i makr matematycznych.

Makra EDOM i ERANGE, zdefiniowane (w <errno.h>) jako różne od zera stałe całkowite, są używane do sygnalizowania błędów dziedziny i zakresu operacji matematycznych; stała HUGE_VAL jest dodatnią wartością typu double. *Błąd dziedziny* występuje wtedy, kiedy argument funkcji nie należy do dziedziny, dla której funkcja została zdefiniowana. W przypadku błędu dziedziny zmienna errno otrzymuje wartość EDOM, a wynik funkcji zależy od implementacji. *Błąd zakresu* występuje wówczas, gdy wartość funkcji nie może być wyrażona jako double. Jeśli wynik jest nadmiarowy, to funkcja zwraca HUGE_VAL z właściwym znakiem i do errno wstawia ERANGE. Jeśli wynik jest niedomiarowy, to funkcja zwraca zero; od implementacji zależy, czy w tym przypadku zmienna errno otrzyma wartość ERANGE.

W poniższym zestawieniu argumenty x i y są typu double, a argument n jest typu int. Wszystkie funkcje zwracają wartość typu double. Wartości kątów dla funkcji trygonometrycznych wyraża się w radianach.

sin(x)	sinus x
cos(x)	cosinus x
tan(x)	tangens x
asin(x)	$\sin^{-1}(x)$ w przedziale $[-\pi/2, \pi/2]$, $x \in [-1, 1]$
acos(x)	$\cos^{-1}(x)$ w przedziale $[0, \pi]$, $x \in [-1, 1]$
atan(x)	$\tan^{-1}(x)$ w przedziale $[-\pi/2, \pi/2]$
atan2(y,x)	$\tan^{-1}(y/x)$ w przedziale $[-\pi, \pi]$
sinh(x)	sinus hiperboliczny x
cosh(x)	cosinus hiperboliczny x

tanh(x)	tangens hiperboliczny x
exp(x)	funkcja wykładnicza e^x
log(x)	logarytm naturalny: $\ln(x)$, $x > 0$
log10(x)	logarytm o podstawie 10: $\log_{10}(x)$, $x > 0$
pow(x,y)	x^y Błąd zakresu wystąpi, gdy x=0 i y≤0 lub gdy x<0 i y nie jest całkowite.
sqrt(x)	$\sqrt{x}$, $x \geq 0$
ceil(x)	najmniejsza liczba całkowita nie mniejsza niż x; wynik typu double
floor(x)	największa liczba całkowita nie większa niż x; wynik typu double
fabs(x)	wartość bezwzględna $\|x\|$
ldexp(x,n)	$x \cdot 2^n$
frexp(x, int *exp)	Rozdziela x na znormalizowaną część ułamkową z przedziału [1/2, 1] i wykładnik potęgi 2; funkcja zwraca część ułamkową, a wykładnik potęgi wstawia do *exp; jeśli x jest równe zero, to obie części wyniku są równe zero.
modf(x, double *ip)	Rozdziela x na część całkowitą i ułamkową, obie z tym samym znakiem co x; część całkowitą wstawia do *ip i zwraca część ułamkową
fmod(x,y)	zmiennopozycyjna reszta z dzielenia x/y, z tym samym znakiem, co x; jeśli y jest równe zero, to wynik zależy od implementacji.

B5 Funkcje narzędziowe: nagłówek <stdlib.h>

Nagłówek <stdlib.h> zawiera deklaracje funkcji służących do przekształcania liczb, przydzielania pamięci i innych podobnych zadań.

double atof(const char *s)

Funkcja atof przekształca tekst zawarty w s na wartość typu double; jest równoważna z strtod(s,(char**)NULL).

int atoi(const char *s)

Przekształca tekst zawarty w s na wartość typu int; jest równoważna z (int) strtol(s, (char**)NULL, 10).

long atol(const char *s)

Przekształca tekst zawarty w s na wartość typu long; jest równoważna z strtol(s, (char**)NULL, 10).

double strtod(const char *s, char **endp)

Funkcja strtod przekształca początkowe znaki tekstu s na wartość typu double (wiodące białe znaki są ignorowane). Jeśli argument endp jest różny od NULL, funkcja wstawia pod wskazany adres wskaźnik do dalszej, nie przekształconej części tekstu s. W przypadku nadmiaru funkcja zwraca HUGE_VAL z właściwym znakiem; w przypadku niedomiaru funkcja zwraca zero. W obu przypadkach wstawia ERANGE do zmiennej errno.

long strtol(const char *s, char **endp, int base)

Funkcja strtol przekształca początkowe znaki tekstu s na wartość typu long (wiodące białe znaki są ignorowane). Jeśli argument endp jest różny od NULL, funkcja wstawia pod wskazany adres wskaźnik do dalszej, nie przekształconej części tekstu s. Jeśli wartość argumentu base (podstawa) zawiera się w przedziale między 2 i 36, to przekształcenia dokonuje się przy założeniu, że tekst wejściowy zawiera liczbę zapisaną przy tej podstawie. Gdy base równa się zero, wówczas podstawa zależy od postaci liczby zapisanej w s: jeśli liczba rozpoczyna się znakiem 0, za podstawę przyjmuje się 8; jeśli rozpoczyna się od 0x lub 0X, za podstawę przyjmuje się 16; w pozostałych przypadkach przyjmuje się 10. Niezależnie od podstawy, litery reprezentują „cyfry" między 10 a base–1; przy podstawie 16 dopuszcza się przedrostek 0x lub 0X. W przypadku nadmiaru funkcja zwraca stałą LONG_MAX lub LONG_MIN (w zależności od znaku wyniku), a errno otrzymuje wartość ERANGE.

unsigned long strtoul(const char *s, char **endp, int base)

Funkcja strtoul działa tak samo, jak strtol, przy czym wynik jest typu unsigned long, a w przypadku błędu zwracaną wartością jest ULONG_MAX.

int rand(void)

Funkcja rand zwraca pseudolosową liczbę całkowitą z przedziału między 0 i RAND_MAX. Wartość RAND_MAX wynosi co najmniej 32767.

void srand(unsigned int seed)

Argument seed funkcji srand jest zarodkiem dla nowego ciągu liczb pseudolosowych. Początkowy zarodek jest równy 1.

void *calloc(size_t nobj, size_t size)

Funkcja calloc zwraca wskaźnik do obszaru pamięci przeznaczonego dla tablicy złożonej z nobj elementów, każdy o rozmiarze size. Funkcja zwraca NULL, jeśli to polecenie nie może być wykonane. Obszar jest inicjowany zerami.

void *malloc(size_t size)

Funkcja malloc zwraca wskaźnik do obszaru pamięci przeznaczonego dla obiektu o rozmiarze size. Funkcja zwraca NULL, jeśli to polecenie nie może być wykonane. Obszar nie jest inicjowany.

void *realloc(void *p, size_t size)

Funkcja realloc zmienia rozmiar obiektu wskazanego przez p na wartość określoną przez size. Zawartość obiektu nie ulegnie zmianie w jego początkowej części o rozmiarze równym mniejszemu z rozmiarów: starego i nowego. Jeśli nowy rozmiar jest większy, dodatkowy obszar pamięci nie jest inicjowany. Funkcja zwraca wskaźnik do nowego obszaru lub NULL, jeśli to polecenie nie może być wykonane; w tym przypadku zawartość *p nie ulega zmianie.

void free(void *p)

Funkcja free zwalnia obszar pamięci wskazany przez p; nie robi nic, jeżeli p równa się NULL. Argument p musi być wskaźnikiem do obszaru uprzednio przydzielonego przez jedną z funkcji: calloc, malloc lub realloc.

void abort(void)

Funkcja abort powoduje nienormalne zakończenie programu, podobnie jak raise(SIGABRT).

void exit(int status)

Funkcja exit powoduje normalne zakończenie programu. Funkcje zarejestrowane przez atexit są wywoływane w kolejności odwrotnej do ich rejestracji, otwarte pliki są aktualizowane, a otwarte strumienie – zamykane. Sterowanie wraca do środowiska programu. Od implementacji zależy to, jak wartość argumentu status jest przekazywana środowisku; przyjmuje się jednak, że wartość równa zero oznacza szczęśliwe zakończenie programu. Można także użyć wartości EXIT_SUCCESS (sukces) i EXIT_FAILURE (porażka).

int atexit(void (*fcn) (void))

Funkcja atexit rejestruje funkcję fcn, którą należy wykonać przy normalnym zakończeniu programu. Jeśli rejestracja nie jest możliwa, funkcja atexit zwraca wartość różną od zera.

int system(const char *)

Funkcja system przekazuje tekst zawarty w s jako polecenie do wykonania przez środowisko programu. Jeśli s jest równe NULL, funkcja zwraca wartość różną od zera wówczas, gdy w środowisku istnieje interpretator poleceń. Jeśli s jest różne od NULL, zwracana przez funkcję wartość zależy od implementacji.

char *getenv(const char *name)

Funkcja getenv zwraca tekst zawarty w zmiennej środowiskowej wskazanej przez nazwę name lub NULL, jeśli taka zmienna nie istnieje. Szczegóły zależą od implementacji.

void *bsearch(const void *key, const void *base,
size_t n, size_t size,
int (*cmp) (const void *keyval, const void *datum))

Funkcja bsearch przegląda tablicę base[0]...base[n–1], złożoną z elementów o rozmiarze size, w poszukiwaniu elementu odpowiadającego obiektowi *key. Funkcja wskazana przez cmp musi zwracać wartość ujemną, gdy jej pierwszy argument (klucz szukania) jest mniejszy od drugiego (element tablicy), wartość zero – gdy jest równy, i wartość większą od zera – gdy jest większy. Elementy tablicy base muszą być uporządkowane rosnąco. Funkcja bsearch zwraca wskaźnik do znalezionego elementu lub NULL, jeśli taki element nie istnieje.

void qsort(void *base, size_t n, size_t size,
int (*cmp) (const void *, const void *))

Funkcja qsort porządkuje w rosnącej kolejności tablicę base[0]...base[n–1] złożoną z elementów o rozmiarze size. Funkcja porównująca wskazana przez cmp jest taka sama, jak w funkcji bsearch.

int abs(int n)

Funkcja abs zwraca wartość bezwzględną argumentu n o typie int.

long labs(long n)

Funkcja labs zwraca wartość bezwzględną argumentu n o typie long.

div_t div(int num, int denom)

Funkcja div oblicza część całkowitą i resztę z dzielenia num/denom. Wyniki są wstawiane do składowych quot i rem struktury typu div_t; składowe są typu int.

ldiv_t ldiv(long num, long denom)

Funkcja ldiv oblicza część całkowitą i resztę z dzielenia num/denom. Wyniki są wstawiane do składowych quot i rem struktury typu ldiv_t; składowe są typu long.

B6 Diagnostyka: nagłówek <assert.h>

Makro assert stosuje się do diagnozowania programów:

void assert(int *wyrażenie*)

Jeśli przy wykonaniu

assert(*wyrażenie*)

wartość wyrażenia jest równa zero, to makro assert wypisze do stderr komunikat w następującej postaci:

Assertion failed: *wyrażenie*, file *nazwa_pliku*, line *nnn*

a następnie wywoła funkcję abort w celu zakończenia programu. Nazwa pliku źródłowego i numer wiersza (*nnn*) pochodzą z wbudowanych makr preprocesora: __FILE__ i __LINE__.

Jeśli przed włączeniem nagłówka <assert.h> do pliku źródłowego zdefiniowano makro NDEBUG, to makro assert jest ignorowane.

B7 Zmienne listy argumentów: nagłówek <stdarg.h>

Nagłówek <stdarg.h> zawiera narzędzia do obsługi listy argumentów funkcji o nieznanej liczbie elementów o nieznanych typach.

Przypuśćmy, że *ostatni_par* jest ostatnim wymienionym z nazwy parametrem funkcji f, która ma zmienną liczbę argumentów. Wewnątrz funkcji f należy zadeklarować zmienną ap typu va_list, która będzie wskazywać kolejne argumenty:

va_list ap;

Zmienną ap trzeba zainicjować za pomocą makra va_start przed użyciem którego-kolwiek z nienazwanych argumentów funkcji:

va_start(va_list ap, *ostatni_par*);

Od tej pory każde wywołanie makra va_arg udostępni wartość, która ma typ i wartość kolejnego nienazwanego argumentu funkcji; wskaźnik ap zostanie przy tym tak zmodyfikowany, aby następne użycie makra va_arg udostępniło następny argument:

typ va_arg(va_list ap, *typ*);

Makro

void va_end(va_list ap);

musi być wywołane raz, po obsłużeniu argumentów, ale przed opuszczeniem funkcji f.

B8 Skoki odległe: nagłówek <setjmp.h>

Deklaracje zawarte w <setjmp.h> dostarczają mechanizmu pozwalającego ominąć normalny ciąg wywołań funkcji i powrotów, zwykle po to, aby umożliwić bezpośredni powrót z głęboko zagnieżdżonych wywołań funkcji.

int setjmp(jmp_buf env)

Funkcja setjmp zachowuje w env informacje o stanie programu, wykorzystywane później przez funkcję longjmp. Bezpośrednio wywołana funkcja setjmp zwraca zero; wartość różną od zera otrzymuje się w wyniku późniejszego wywołania funkcji longjmp. Wywołania funkcji setjmp mogą wystąpić tylko w niektórych kontekstach – w zasadzie tylko w testach instrukcji if, switch i pętli, a i tu w niezbyt skomplikowanych wyrażeniach relacyjnych.

```
if (setjmp(env) == 0)
        /* wchodzi tu przy bezpośrednim wywołaniu */
else
        /* wchodzi tu po wywołaniu longjmp */
```

void longjmp(jmp_buf env, int val)

Funkcja longjmp przywraca stan programu zapamiętany w zmiennej env przez ostatnie wywołanie funkcji setjmp, a wykonanie programu wznawia się w taki sposób, jakby właśnie wykonano funkcję setjmp, która wróciła z wartością val różną od zera. Funkcja zawierająca wywołanie setjmp nie może się zakończyć przed wywołaniem longjmp. Dostępne obiekty mają takie wartości, jakie miały w chwili wywołania longjmp, z tym że wartości zmiennych automatycznych nie-volatile nie są zdefiniowane, jeśli uległy zmianie po wykonaniu setjmp.

B9 Sygnały: nagłówek <signal.h>

Nagłówek <signal.h> dostarcza mechanizmów do obsługi zdarzeń wyjątkowych, które mogą zajść podczas wykonywania programu. Takimi zdarzeniami są np. sygnał przerwania, nadesłany z zewnątrz programu, lub błąd w wykonaniu programu.

void (*signal(int sig, void (*handler)(int)))(int)

Funkcja signal rozstrzyga, w jaki sposób będą obsługiwane nadchodzące sygnały. Jeśli argument handler ma wartość SIG_DFL, podejmuje się domyślną akcję zależną od implementacji; jeśli ma wartość SIG_IGN, to sygnał będzie ignorowany; w pozostałych przypadkach wywołuje się wskazaną przez niego funkcję z podanym rodzajem sygnału. Oto niektóre z możliwych sygnałów:

SIGABRT	nienormalne zakończenie programu, np. jako skutek wywołania funkcji abort;
SIGFPE	błąd arytmetyczny, np. dzielenie przez zero lub nadmiar;
SIGILL	zła postać funkcji, np. nielegalna instrukcja;
SIGINT	ingerencja użytkownika, np. przerwanie;
SIGSEGV	nielegalne odwołanie do pamięci, np. sięgnięcie poza przydzielony obszar;
SIGTERM	żądanie zakończenia przesłane do programu.

Funkcja signal zwraca poprzednią wartość argumentu handler dla danego sygnału lub SIG_ERR w przypadku błędu.

Jeśli teraz wystąpi sygnał sig, to najpierw przywraca się domyślną akcję dla tego sygnału, a następnie wywołuje wskazaną funkcję obsługi tak, jakby nastąpiło wywołanie (*handler)(sig). Po powrocie z funkcji obsługi program będzie wznowiony w miejscu, w którym nastąpiło przerwanie sygnałem.

Początkowy stan sygnałów zależy od implementacji.

int raise(int sig)

Funkcja raise wysyła do programu sygnał sig; w przypadku niepowodzenia zwraca wartość różną od zera.

B10 Obsługa daty i czasu: nagłówek <time.h>

W nagłówku <time.h> znajdują się deklaracje typów i funkcji służących do obsługi daty i czasu. Niektóre funkcje operują *czasem lokalnym*, który może być różny od czasu kalendarzowego na skutek np. różnicy stref czasowych. Typy clock_t i time_t są typami arytmetycznymi reprezentującymi czas, a struktura struct tm mieści w sobie składniki czasu kalendarzowego:

int tm_sec;	sekundy, które upłynęły po minucie (0, 61)
int tm_min;	minuty, które upłynęły po godzinie (0, 59)
int tm_hour;	godziny, które upłynęły od północy (0, 23)
int tm_mday;	dzień miesiąca (1, 31)
int tm_mon;	miesiące, które upłynęły od stycznia (0, 11)
int tm_year;	lata, które upłynęły od 1900 r.
int tm_wday;	dni, które upłynęły od niedzieli (0, 6)
int tm_yday;	dni, które upłynęły od 1 stycznia (0, 365)
int tm_isdst;	znacznik letniej zmiany czasu

Znacznik tm_isdst jest dodatni, gdy obowiązuje czas letni; zero – gdy nie obowiązuje, i ujemny – w przypadku braku takiej informacji.

clock_t clock(void)
Funkcja clock zwraca czas procesora wykorzystany przez program od rozpoczęcia działania lub −1, gdy ta informacja jest niedostępna. Wyrażenie clock()/CLOCKS_PER_SEC daje czas w sekundach.

time_t time(time_t *tp)
Funkcja time zwraca aktualny czas kalendarzowy lub −1, gdy ta informacja jest niedostępna. Jeśli argument tp jest różny od NULL, zwracana wartość jest również wstawiana do *tp.

double difftime(time_t time2, time_t time1)
Funkcja difftime zwraca różnicę time2 – time1 wyrażoną w sekundach.

time_t mktime(struct tm *tp)
Funkcja mktime przekształca lokalny czas zawarty w strukturze *tp na czas kalendarzowy w takiej samej postaci, jakiej używa funkcja time. Wartości składowych struktury muszą się zawierać w podanych wyżej granicach. Funkcja zwraca czas kalendarzowy lub −1, jeśli nie może dokonać przekształcenia.

Następne cztery funkcje zwracają wskaźnik do statycznego obiektu, który może ulec zmianie na skutek innych wywołań tych funkcji.

char *asctime(const struct tm *tp)
Funkcja asctime przekształca czas zapisany w strukturze *tp na tekst o postaci:
Sun Jan 3 15:14:13 1988\n\0

char *ctime(const time_t *tp)
Funkcja ctime przekształca czas kalendarzowy *tp na czas lokalny; jest równoważna z wywołaniem asctime(localtime(tp)).

struct tm *gmtime(const time_t *tp)
Funkcja gmtime przekształca czas kalendarzowy *tp na Skoordynowany Czas Uniwersalny (UTC); zwraca NULL, jeśli UTC nie jest dostępny. Nazwa gmtime ma znaczenie historyczne.

struct tm *localtime(const time_t *tp)
Funkcja localtime przekształca czas kalendarzowy *tp na czas lokalny.

size_t strftime(char *s, size_t smax, const char *fmt, const struct tm *tp)
Funkcja strftime przekształca datę i czas, zawarte w strukturze *tp, na tekst i zapisuje w miejsce wskazane przez s. Przekształcenie odbywa się na podstawie formatu fmt, analogicznego do formatu w funkcji printf. Zwykłe znaki (łącznie z końcowym '\0') są kopiowane do s. Każde wystąpienie specyfikacji przekształcenia

%c jest zastępowane wartościami odpowiednimi dla miejscowych zwyczajów (zgodnie z poniższym opisem). Do s wstawia się co najwyżej smax znaków. Funkcja strftime zwraca liczbę znaków w s (nie licząc '\0') lub zero, jeśli wyprodukowała więcej niż smax znaków. W formacie mogą wystąpić następujące specyfikacje przekształcenia:

%a	skrócona nazwa dnia tygodnia
%A	pełna nazwa dnia tygodnia
%b	skrócona nazwa miesiąca
%B	pełna nazwa miesiąca
%c	miejscowa postać daty i czasu
%d	dzień miesiąca (01-31)
%H	godzina (zegar 24-godzinny) (00-23)
%I (litera I)	godzina (zegar 12-godzinny) (01-12)
%j	dzień roku (001-366)
%m	miesiąc (01-12)
%M	minuta (00-59)
%p	miejscowy odpowiednik AM (przed południem) i PM (po południu)
%S	sekunda (00-61)
%U	numer tygodnia w roku (niedziela jest pierwszym dniem tygodnia) (0-53)
%w	dzień tygodnia (0-6, niedziela jest 0)
%W	numer tygodnia w roku (poniedziałek jest pierwszym dniem tygodnia) (0-53)
%x	miejscowa postać daty
%X	miejscowa postać czasu
%y	rok bez wieku (00-99)
%Y	rok z wiekiem (np. 1993)
%Z	nazwa strefy czasowej, jeśli jest zdefiniowana
%%	znak %

B11 Ograniczenia implementacji: nagłówki <limits.h> i <float.h>

Nagłówek <limits.h> zawiera definicje stałych określających rozmiary typów całkowitych. Podane niżej wartości są dopuszczalnymi wartościami minimalnymi; w użyciu mogą być większe wartości.

CHAR_BIT	8	bitów w char
CHAR_MAX	UCHAR_MAX *lub* SCHAR_MAX	maksymalna wartość char
CHAR_MIN	0 *lub* SCHAR_MIN	minimalna wartość char

INT_MAX	+32767	maksymalna wartość int
INT_MIN	–32767	minimalna wartość int
LONG_MAX	+2147483647	maksymalna wartość long
LONG_MIN	–2147483647	minimalna wartość long
SCHAR_MAX	+127	maksymalna wartość signed char
SCHAR_MIN	–127	minimalna wartość signed char
SHRT_MAX	+32767	maksymalna wartość short
SHRT_MIN	–32767	minimalna wartość short
UCHAR_MAX	255	maksymalna wartość unsigned char
UINT_MAX	65535	maksymalna wartość unsigned int
ULONG_MAX	4294967295	maksymalna wartość unsigned long
USHRT_MAX	65535	maksymalna wartość unsigned short

Stałe wymienione poniżej (podzbiór <float.h>) odnoszą się do arytmetyki zmiennopozycyjnej. Jeśli jest podana wartość, reprezentuje ona minimalną wartość odnośnej wielkości. Każda implementacja określa sobie właściwe wartości.

FLT_RADIX	2	podstawa reprezentacji wykładnika potęgi, np. 2, 16
FLT_ROUNDS		rodzaj zaokrąglania przy dodawaniu zmiennopozycyjnym
FLT_DIG	6	liczba cyfr dziesiętnych precyzji
FLT_EPSILON	1E–5	najmniejsza liczba x taka, że $1{,}0 + x \neq 1{,}0$
FLT_MANT_DIG		liczba cyfr podstawy FLT_RADIX w mantysie
FLT_MAX	1E+37	maksymalna liczba zmiennopozycyjna
FLT_MAX_EXP		maksymalne n takie, że da się wyrazić FLT_RADIX^n–1
FLT_MIN	1E–37	minimalna znormalizowana liczba zmiennopozycyjna
FLT_MIN_EXP		minimalne n takie, że 10^n jest liczbą znormalizowaną
DBL_DIG	10	liczba cyfr dziesiętnych precyzji
DBL_EPSILON	1E–9	najmniejsza liczba x taka, że $1{,}0 + x \neq 1{,}0$
DBL_MANT_DIG		liczba cyfr podstawy FLT_RADIX w mantysie
DBL_MAX	1E+37	maksymalna liczba zmiennopozycyjna double
DBL_MAX_EXP		maksymalne n takie, że da się wyrazić FLT_RADIX^n–1
DBL_MIN	1E–37	minimalna znormalizowana liczba zmiennopozycyjna double
DBL_MIN_EXP		minimalne n takie, że 10^n jest liczbą znormalizowaną

DODATEK C PODSUMOWANIE ZMIAN

Od czasu opublikowania pierwszego wydania tej książki definicja języka C uległa zmianom. W większości były to rozszerzenia oryginalnego języka sformułowane tak, aby zachować zgodność z dotychczasową praktyką; w niektórych przypadkach usunięto niejednoznaczności oryginalnej definicji; niektóre modyfikacje zmieniły dotychczasową praktykę. Wiele nowych udogodnień ogłoszono w dokumentach towarzyszących kompilatorom udostępnianym przez AT&T i następnie adaptowano przez innych dostawców kompilatorów języka C. Komitet ANSI, standaryzując później język, uwzględnił większość tych zmian, a także wprowadził inne znaczące modyfikacje. Część tych zmian przewidziano w kilku komercyjnych kompilatorach nawet przed ukazaniem się oficjalnego standardu języka C.

Niniejszy dodatek podsumowuje różnice między językiem zdefiniowanym w pierwszym wydaniu tej książki, a spodziewanym ostatecznym Standardem. Omówiono tu jedynie sam język, pominięto zaś otoczenie programów i bibliotekę; chociaż te są ważną częścią Standardu, nie ma wiele do porównywania, ponieważ w pierwszym wydaniu nie próbowano narzucać wymagań dotyczących otoczenia i biblioteki.

- Aparat preprocesora został w Standardzie zdefiniowany dokładniej niż w pierwszym wydaniu książki i został rozszerzony: wprowadzono nowe operatory do łączenia słów (##) i tworzenia napisów (#); wprowadzono nowe instrukcje preprocesora, jak #elif i #pragma; jawnie dopuszczono ponowne deklaracje makra za pomocą tego samego ciągu leksemów; nie zastępuje się już parametrów wewnątrz napisów. Kontynuacja za długich wierszy za pomocą znaku \ jest dopuszczalna wszędzie, nie tylko w napisach i makrodefinicjach. Patrz p. A12.
- Minimalna znacząca długość lokalnych identyfikatorów została rozszerzona do 31 znaków; najmniejsza znacząca długość zewnętrznych identyfikatorów pozostaje ograniczona do 6 znaków z utożsamianiem małych i wielkich liter (wiele implementacji dopuszcza więcej znaków).
- Sekwencje trzyznakowe rozpoczynające się od ?? pozwalają wyrazić znaki nie występujące w niektórych zbiorach znaków. Wpowadzono takie sekwencje dla zna-

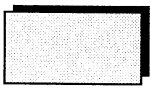
ków # \ ^ [ ] { } | ~ (patrz p. A12.1). Zauważ, że wprowadzone sekwencje trzyznakowe mogą zmienić znaczenie stałych napisowych zawierających ciąg ??.

- Wprowadzono nowe słowa kluczowe (void, const, volatile, signed, enum). Usunięto martwe słowo kluczowe entry.
- Zdefiniowano nowe sekwencje specjalne do użytku w stałych znakowych i napisowych. Nie jest zdefiniowany efekt użycia \ ze znakiem nie należącym do zatwierdzonego zestawu sekwencji specjalnych. Patrz p. A2.5.2.
- Przez wszystkich faworyzowana mała zmiana: 8 i 9 nie są cyframi ósemkowymi.
- Wprowadzono większą liczbę przyrostków dla oznaczenia typu stałych: U lub L dla całkowitych i F lub L dla zmiennopozycyjnych. Wysubtelniono również zasady rozstrzygania o typach stałych bez przyrostka (p. A2.5).
- Sąsiadujące ze sobą stałe napisowe są sklejane.
- Wprowadzono notację dla napisów i stałych znakowych zawierających znaki z rozszerzonego zbioru znaków; patrz p. A2.6.
- Za pomocą słów kluczowych signed i unsigned można jawnie deklarować, czy stałe znakowe i innych typów mają znak arytmetyczny. Usunięto zwrot long float jako synonim do double, można natomiast używać long double do zadeklarowania dodatkowej precyzji wielkości zmiennopozycyjnych.
- Od jakiegoś czasu stosuje się zwrot unsigned char. W Standardzie wprowadzono słowo kluczowe signed, by jawnie podkreślać, że obiekty typu char i innych typów arytmetycznych mają znak.
- Od kilku lat w większości implementacji jest dostępne słowo kluczowe void. Standard wprowadza void * jako ogólny typ wskaźnikowy; poprzednio rolę tę pełnił char *. Jednocześnie ustanowiono jawne zasady dotyczące mieszania wskaźników i liczb całkowitych lub wskaźników różnych typów bez użycia operatora rzutowania.
- Standard ustala minimalne zakresy dla wartości typów arytmetycznych i wyznacza specjalne nagłówki (<limit.h> i <float.h>), które mają zawierać charakterystyki poszczególnych implementacji.
- Wyliczenia nie były znane za czasów pierwszego wydania tej książki.
- Standard adoptuje z języka C++ pojęcie kwalifikatora typu, np. const (p. A8.2).
- Stałe napisowe już nie mogą być modyfikowane i można je umieszczać w pamięci chronionej przed zapisem.
- „Zwykłe przekształcenia arytmetyczne" uległy zmianie. Zamiast „wśród typów całkowitych zawsze zwycięża unsigned; w operacjach zmiennopozycyjnych uży-

waj double" mówi się teraz „promuj do najmniejszego, ale wystarczająco obszernego typu". Patrz p. A6.5.

- Stare operatory przypisania, jak =+, zniknęły całkowicie. Ponadto operatory przypisania są pojedynczymi leksemami; w poprzednim wydaniu książki były one parami leksemów i mogły być rozdzielone odstępami.
- Unieważniono prawo kompilatorów do traktowania operatorów łącznych matematycznie jako łączne obliczeniowo.
- Wprowadzono jednoargumentowy operator + dla symetrii z jednoargumentowym –.
- Wskaźnik do funkcji może być używany jako nazewnik funkcji bez jawnego stosowania operatora *. Patrz p. A7.3.2.
- Struktury mogą być przypisywane, przekazywane funkcjom i zwracane przez funkcje.
- Dopuszcza się stosowanie operatora adresu do tablic; wynikiem jest wskaźnik do tablicy.
- W pierwszym wydaniu książki operator sizeof zwracał typ int; wiele późniejszych implementacji zmieniło to na unsigned. Standard wyraźnie uzależnia ten typ od implementacji, ale wymaga, aby był on (pod nazwą size_t) zdefiniowany w standardowym nagłówku (<stddef.h>). Podobna zmiana dotyczy typu różnicy wskaźników (ptrdiff_t). Patrz p. A7.4.8 i A7.7.
- Operator adresu & nie może odnosić się do obiektów zadeklarowanych jako register, nawet jeśli w danej implementacji ten obiekt nie jest trzymany w rejestrze.
- Typem wyrażenia przesunięcia jest typ jego lewego argumentu; prawy argument nie może mieć wpływu na typ wyniku. Patrz p. A7.8.
- Standard zezwala na tworzenie wskaźnika pokazującego tuż za koniec tablicy i pozwala używać go w obliczeniach i relacjach; patrz p. A7.7.
- Standard wprowadza (zapożyczając od C++) notację deklaracji prototypu funkcji ustalającego typy parametrów oraz określa jawne oznaczanie funkcji ze zmienną liczbą argumentów i aprobowany sposób postępowania z nimi. Patrz p. A7.3.2, A8.6.3, B7. Starsza notacja jest (z pewnymi ograniczeniami) nadal akceptowana.
- Puste deklaracje, które nie mają deklaratorów i nie deklarują przynajmniej struktury, unii lub wyliczenia, są przez Standard zabronione. Z drugiej strony, deklaracja tylko z etykietką struktury lub unii zmienia deklarację tej etykietki, nawet jeśli przedtem jej zasięg był szerszy.
- Deklaracje zewnętrzne bez jakichkolwiek specyfikatorów i kwalifikatorów (po prostu goły deklarator) są zabronione.

- Deklaracje extern zawarte wewnątrz bloku mogły być w niektórych implementacjach eksportowane do reszty pliku. W Standardzie jest jasno powiedziane, że zasięgiem takich deklaracji jest tylko zawierający je blok.
- Zasięg parametrów rozciąga się na całą instrukcję złożoną, tworzącą treść funkcji; zmienne zadeklarowane na najwyższym poziomie w funkcji nie mogą zatem zasłonić parametrów.
- Przestrzenie nazw dla identyfikatorów zostały nieco zmienione. W Standardzie wszystkie etykietki są zaliczane do wspólnej klasy; wprowadzono także nową klasę etykiet; patrz p. A11.1. Nazwy składowych struktur i unii są związane z tymi strukturami i uniami. (Od pewnego czasu było to powszechną praktyką.)
- Unie mogą być inicjowane; inicjator odnosi się do pierwszej składowej.
- Automatyczne struktury, unie i tablice mogą być inicjowane, choć w ograniczonym zakresie.
- Tablice znakowe z jawnym rozmiarem mogą być inicjowane napisem o dokładnie takiej samej liczbie znaków (znak '\0' jest milcząco pomijany).
- Wyrażenie kontrolne i wyrażenia przypadków w instrukcji switch mogą być dowolnego typu całkowitego.



SKOROWIDZ

A

abort, funkcja standardowa 337
abs, funkcja standardowa 338
acos, funkcja standardowa 334
addpoint, funkcja 177
addtree, funkcja 190
adres 17, 51, 129, 130
–, arytmetyka 51, 139
afree, funkcja 139
alloc, funkcja 139, 191
analizator składniowy, program 168
ANSIC, standard 13, 18, 50, 60, 99, 130, 173, 203, 225, 251
a.out, plik wykonywalny 24, 104
argc, licznik argumentów programu 157
argumenty funkcji 19, 25, 30, 49, 51, 58 73, 94, 107, 208, 231, 266, 301
– –, przekazywanie przez referencję 51
– –, przekazywanie przez wartości 51, 132, 145, 177, 212, 266
– –, przekształcenia 30, 32, 73, 266
– –, tablice 52, 138
– –, wskaźniki 52, 133
– –, zmienna liczba 208, 290
– makrowywołania 126
– wywołania funkcji 103
– – programu 157
argv, tablica argumentów programu 157, 204
arytmetyka 19, 23
– liczb całkowitych 30
– – zmiennopozycyjnych 30, 344
– modulo 2^n 62, 257
– na adresach 51, 139
– – wskaźnikach 130, 137, 139, 143, 162, 186, 246, 271
– uzupełnieniowa do 2 80, 95

A

ASCII, zbiór znaków 41, 63, 70, 305, 332
asctime, funkcja standardowa 342
asin, funkcja standardowa 334
asm, słowo kluczowe 253
assert, makro standardowe 338
assert.h, nagłówek standardowy 321, 338
atan, funkcja standardowa 334
atan2, funkcja standardowa 224, 334
atexit, funkcja standardowa 337
atof, funkcja 92, 104
atof, funkcja standardowa 335
atoi, funkcja 69, 91, 106
atoi, funkcja standardowa 335
atol, funkcja standardowa 335
atrybuty pliku 239
auto, klasa pamięci 56, 119, 256, 278

B

białe znaki 44, 211, 327, 332
biblioteka standardowa 13, 19, 25, 30, 36, 47, 58, 64, 73, 92, 101, 104, 114, 124, 140, 198, 203, 220, 225, 231, 233, 321
binsearch, funkcja 87, 182, 186
bitcount, funkcja 79
bitowy wzorzec bajtu 37, 63, 71
blok 84, 120, 121, 256, 296, 301
błąd dziedziny 334
– zakresu 334
błędy 125
break, instrukcja 89, 96, 299
bsearch, funkcja standardowa 338

C

calloc, funkcja standardowa 223, 249, 336
case, przypadek w switch 88, 295, 297

cat, program 214, 216
cc, kompilator C 24, 103
ceil, funkcja standardowa 335
char, typ danych 28, 37, 45, 61, 69, 71, 76, 130, 153, 257, 279
clearerr, funkcja standardowa 331
clock, funkcja standardowa 342
clock_t, nazwa typu 341
close, funkcja systemowa 231
closedir, funkcja 243
const, kwalifikator typu 67, 258, 279
continue, instrukcja 90, 298, 299
copy, funkcja 52, 54, 55, 58
cos, funkcja standardowa 104, 224, 334
cosh, funkcja standardoowa 334
cp, program 230
creat, funkcja systemowa 229, 230
ctime, funkcja standardowa 342
ctype.h, nagłówek standardowy 70, 127, 184, 205, 222, 321, 331

D

day_of_year, funkcja 152
dcl, funkcja 168
dcl, program 166
decyzje wielowariantowe 46, 85, 88
default, przypadek w switch 88, 295, 297
#define, instrukcja preprocesorowa 35, 65, 128, 306
defined, wyrażenie preprocesora 128, 310
definicja próbna 302
– typu 174
definicje funkcji 19, 48, 49, 99, 102, 289, 300
– pól bitowych 201
– tablic 116
– zmiennych 56, 58, 103, 125
deklaracja... (zmienna liczba argumentów) 208
– rekurencyjna 188
deklaracje 28, 66, 277-290
– funkcji 19, 49, 99, 131, 266, 288
– parametrów funkcji 48
– struktur 174, 280
– tablic 45, 152, 287
– unii 280
– wskaźników 131, 286
– zmiennych 28, 56, 58, 66, 71, 107, 115, 120, 125
deklaratory 167, 285
deskryptor pliku 226, 229, 233
dev_t, nazwa typu 239
difftime, funkcja standardowa 342
dirdcl, funkcja 168
dirent.h, nagłówek 238
dirwalk, funkcja 240
div, funkcja standardowa 338
div_t, nazwa typu 338
do-while, instrukcja pętli 94, 96, 298
double, typ danych 28, 39, 61, 71, 104, 258, 279
drzewo binarne 187
– wywodu 168
dummy, funkcja atrapa 102
dystrybutor pamięci 139, 191, 244
dzielenie całkowite 30, 67

E

EBCDIC, zbiór znaków 70
echo, program 157
EDOM, makro standardowe 334
efekty uboczne obliczeń 83, 127, 263, 296, 308
#elif, instrukcja preprocesora 128, 310
else, część instrukcji if 43, 46, 87, 297
#else, instrukcja preprocesora 128, 310
#endif, instrukcja preprocesora 128, 310
enum, typ danych 65, 200, 253
EOF, znacznik końca pliku 37, 114, 204
ERANGE, makro standardowe 334
errno 331
errno.h, nagłówek standardowy 321, 331
error, funkcja 231
etykietki struktur 174, 214, 281, 303
– unii 281, 303
etykiety 97, 295
–, zasięg 97
exit, funkcja standardowa 218, 337
exp, funkcja standardowa 224, 335
extern, klasa pamięci 56, 115, 257, 278, 302

F

fabs, funkcja standardowa 224, 335
fclose, funkcja standardowa 217, 231, 323
fcntl.h, nagłówek systemowy 229
feof, funkcja standardowa 219, 331
ferror, funkcja standardowa 219, 331
fflush, funkcja standardowa 323
fgetc, funkcja standardowa 328
fgetpos, funkcja standardowa 330
fgets, funkcja standardowa 219, 329

__FILE__, makro preprocesora 339
FILE, struktura pliku 214, 322
filecopy, funkcja 217
_fillbuf, funkcja 234
find, program 159, 160
float.h, nagłówek standardowy 62, 321, 343, 344
float, typ danych 28, 39, 61, 69, 71, 258, 279
floor, funkcja standardowa 335
_flushbuf, funkcja 234
fmod, funkcja standardowa 335
fopen, funkcja standardowa 214, 229, 234, 322
for, instrukcja pętli 33, 34, 40, 71, 90, 96, 298
format 55, 206, 210, 324, 326
Forth 108
Fortran 24, 41, 47, 51, 56, 91, 107
fortran, słowo kluczowe 253
fpos_t, nazwa typu 331
fprintf, funkcja standardowa 216, 218, 231, 324
fputc, funkcja standardowa 329
fputs, funkcja standardowa 219, 329
fread, funkcja standardowa 330
free, funkcja standardowa 140, 223, 248, 337
freopen, funkcja standardowa 217, 323
frexp, funkcja standardowa 335
fscanf, funkcja standardowa 216, 326
fseek, funkcja standardowa 233, 237, 330
fsetpos, funkcja standardowa 331
fsize, funkcja 240
fsize, program 240
fstat, funkcja systemowa 242
ftell, funkcja standardowa 330
funkcje 17, 23, 24, 28, 47, 99, 258, 288
–, argumenty 19, 25, 30, 49, 51, 58, 73, 94, 107, 208, 231, 266, 301
–, – strukturowe 176
–, definicje 19, 48, 49, 99, 102, 289, 300
–, deklaracje 19, 49, 99, 131, 266, 288
– numeryczne 104
–, parametry 19, 48, 51, 106, 115, 119, 138, 154, 266, 289, 300
–, prototypy 49, 73, 106, 125, 163, 186, 197, 227, 266
– rekurencyjne 122, 168, 190, 241
– rozpraszające 194
– statyczne 118
–, wartości 48, 51, 54, 103, 104
–, wywołania 25, 51, 56, 83, 108, 163, 265, 301
–, zmienna liczba argumentów 231, 301
fwrite funkcja standardowa 330

G

gebits, funkcja 78
generator analizatorów składniowych YACC 313
getc, funkcja standardowa 215, 329
getc, makro 234
getch, funkcja 112
getchar, funkcja standardowa 36, 113, 127, 204, 216, 228, 329
getenv, funkcja standardowa 338
getint, funkcja 134
getline, funkcja 52, 58, 101, 114, 220
getop, funkcja 112
gets, funkcja standardowa 329
gettoken, funkcja 170
getword, funkcja 183
gmtime, funkcja standardowa 342
goto, instrukcja 97, 295, 299

H

hash, funkcja 194
HUGE_VAL, nadmiar 334, 336

I

identyfikatory (zob. nazwy)
if, instrukcja 41, 43, 71, 87, 297
#if, instrukcja preprocesora 128, 183, 309
#ifdef, instrukcja preprocesora 129, 309
#ifndef, instrukcja preprocesora 129, 309
#include, instrukcja preprocesora 25, 58, 118, 321
indeksowanie 52, 137
– tablic 45, 69, 136, 153, 265, 288
inicjator 66, 144, 155, 290
inicjowanie 99, 121, 290, 297
– pętli 34, 91
– struktur 175, 292
– tablic 45, 122, 154, 155, 291
– unii 200, 291
– wskaźników 141, 155, 291
– zmiennych 51, 66, 120, 121, 290
– – automatycznych 56, 66, 121, 291
– – rejestrowych 121
– – statycznych 66, 121, 291
– – zewnętrznych 66, 116, 121
ino_t, nazwa typu 239, 242
install, funkcja 193, 195
instrukcja 18, 24, 43, 46, 84, 295

instrukcja etykietowana 295
– powtarzania (zob. pętle)
– przypisania 29, 122
– pusta 40, 296
– skoku 97, 295, 299
– sterująca 84
– wyboru 295
– wyrażeniowa 84, 296
– złożona 84, 120, 296
instrukcje preprocesora 125-129, 305-311
– –, #define 35, 65, 128, 306
– –, #elif 128, 310
– –, #else 128, 310
– –, #endif 128, 310
– –, #error 311
– –, #if 128, 183, 309
– –, #ifdef 129, 309
– –, #ifndef 129, 309
– –, #inlude 25, 58, 118, 309, 321
– –, #line 311
– –, #pragma 311
– –, #undef 127, 228, 306
int, typ danych 28, 37, 39, 45, 50, 61, 71, 76, 103, 104, 257, 279
isalnum, funkcja standardowa 222, 332
isalnum, makro standardowe 184
isalpha, funkcja standardowa 222, 332
isalpha, makro standardowe 184
iscntrl, funkcja standardowa 332
isdigit, funkcja standardowa 222, 332
isgraph, funkcja standardowa 332
islower, funkcja standardowa 222, 332
isprint, funkcja standardowa 322
ispunct, funkcja standardowa 332
isspace, funkcja standardowa 222, 332
isspace, makro standardowe 184
isupper, funkcja standardowa 222, 332
isxdigit, funkcja standardowa 332
itoa, funkcja 94

J

Jednostka leksykalna 251
– tłumaczenia 251, 299

K

klasy pamięci 56, 118, 120, 256, 278
– –, auto 56, 119, 256, 278
– –, extern 56, 115, 257, 278, 302
klasy pamięci, register 119, 131, 256, 278
– –, static 56, 118, 120, 256, 278, 302
kod znaku 41, 63, 65, 70, 257
kolejność obliczania wyrażeń 43, 82, 93, 263
komentarze 28, 50, 59, 252
kompilacja warunkowa 99, 305, 309
kontrola poprawności 106
– zgodności typów 191
korzeń drzewa 188
kwalifikatory typów 61, 67, 258, 262, 279

L

labs, funkcja standardowa 338
ldexp, funkcja standardowa 335
ldiv, funkcja standardowa 338
leksem 170, 305
limits.h, nagłówek standardowy 62, 257, 321, 343
#line, instrukcja preprocesora 311
__LINE__, makro preprocesora 339
locale.h, nagłówek standardowy 321
localtime, funkcja standardowa 342
log, funkcja standardowa 224, 335
log10, funkcja standardowa 224, 335
long, typ danych 28, 39, 61, 62, 72, 76, 130, 279
long double, typ danych 60, 62, 71, 258
longjmp, funkcja standardowa 340
LONG_MAX, stała symboliczna 336
LONG_MIN, stała symboliczna 336
lookup, funkcja 193, 194
lower, funkcja 70
lower, program 205
lseek, funkcja systemowa 232
l-wartość 259

Ł

Łańcuch opisów 193
łączność identyfikatorów (nazw) 257, 302, 304
– –, wewnętrzna 257, 302, 304
– –, zewnętrzna 257, 302, 304
– operatorów 43, 67, 81, 263

M

main, główna funkcja programu 24, 49
makepoint, funkcja 176
makra 205, 216
–, wywołanie 307
–, rozwijanie 307

makrodefinicja 126, 306
makrogeneracja 305
makrorozwinięcie 126, 306
malloc, funkcja standardowa 140, 191, 223, 246, 337
maski 200
math.h, nagłówek standardowy 72, 73, 224, 321, 334
memchr, funkcja standardowa 334
memcmp, funkcja standardowa 334
memcpy, funkcja standardowa 334
memmove, funkcja standardowa 334
memset, funkcja standardowa 334
minprintf, funkcja 208
mktime, funkcja standardowa 342
modf, funkcja standardowa 335
month_day, funkcja 152
month_name, funkcja 155
morecore, funkcja 248
MS-DOS, system operacyjny 226, 237

N

nagłówki 19, 58, 62, 117, 231
– standardowe 321
– –, <assert.h> 321, 338
– –, <ctype.h> 70, 127, 184, 205, 222, 321, 331
– –, <errno.h> 321, 331
– –, <float.h> 62, 321, 343, 344
– –, <limits.h> 62, 257, 321, 343
– –, <locale.h> 321
– –, <math.h> 72, 73, 224, 321, 334
– –, <setjmp.h> 321, 340
– –, <signal.h> 321, 340
– –, <stdarg.h> 208, 321, 339
– –, <stddef.h> 143, 183, 321
– –, <stdio.h> 25, 37, 125, 127, 204, 228, 233, 321, 322
– –, <stdlib.h> 104, 192, 321, 335
– –, <string.h> 65, 146, 159, 221, 321, 332
– –, <time.h> 321, 341
napisy 18, 25, 64, 155, 255
nazwy 60, 126, 174, 252, 264
– funkcji 61, 106
– parametrów 49
– plików 125, 214, 229
– stałych symbolicznych 60
– symboliczne 35
– tablic 137
– typów 196, 270, 293
nazwy wewnętrzne 252
– zewnętrzne 61, 252
– zmiennych 35, 48, 60, 97, 256
NULL, pusty wskaźnik 141, 261
numcmp, funkcja 164

O

obsługa stosu 109, 147
odwołania systemowe 225
Odwrotna Notacja Polska 108
ogniwo łańcucha 194
ograniczenia dotyczące położenia w pamięci 191, 245, 262
opcje wywołania programu 159
open, funkcja systemowa 229
opendir, funkcja 238, 242
operator adresowania pośredniego * 82, 131, 154, 268
– adresu & 82, 131, 133, 137, 268
– bitowej alternatywy (OR) | 77, 82, 274
– – koniunkcji (AND) & 77, 82, 273
– – różnicy sysmetrycznej (XOR) ^ 77, 82, 273
– bitowego dopełnienia jedynkowego ~ 77, 269
– – przesunięcia w lewo << 77, 272
– – przesunięcia w prawo >> 77, 272
– dodawania + 67, 81, 271
– dzielenia / 67, 81, 271
– dzielenia modulo % 67, 81, 271
– iloczynu logicznego (AND) && 43, 68, 77, 82, 274
– minus jednoargumentowy – 67, 81, 269
– mnożenia * 67, 81, 270
– negacji logicznej (NOT) ! 69, 269
– odejmowania – 67, 81, 271
– plus jednoargumentowy + 67, 81, 269
– przecinkowy , 82, 93, 276
– przypisania *op*= 38, 41, 68, 78, 275
– przyrównania równe == 41, 68, 273
– – nie równe != 37, 38, 68, 273
– relacji mniejsze lub równe <= 68, 272
– – mniejsze niż < 68, 272
– – większe lub równe >= 68, 272
– – większe niż > 68, 272
– rozmiaru sizeof 82, 269
– rzutowania (*typ*) 73, 270
– składowej . 81, 175
– sumy logicznej (OR) || 43, 68, 77, 82, 274
– trzyargumentowy 80
– warunkowy ?: 80, 275

operator wskaźnikowy składowej -> 81, 179
– wywołania funkcji () 81, 265
– zmniejszenia -- 39, 74, 267, 268
– zwiększenia ++ 39, 74, 267, 268
operatory 60, 67, 74, 76, 78, 268
– addytywne 271, 268
– arytmetyczne 67, 132
– bitowe 76, 273, 274
– dwuargumentowe 67, 71
– jednoargumentowe 67, 69, 77, 268
–, kolejność obliczeń 43, 68, 81
– logiczne 68
–, łączność 43, 67, 81, 263
– multyplikatywne 270
– preprocesora 127, 128, 307
– –, tworzenie napisów 127, 307
– –, sklejanie argumentów makro 128, 307
–, priorytety 38, 67, 81, 82, 154, 179, 263
– przedrostkowe 39, 74, 268
–, przemienność 111
– przesunięcia 77, 272
– przypisania 38, 43, 68, 78, 268, 275, 276
– przyrostkowe 39, 74, 267
– przyrównania 68, 82, 273
– relacji 68, 272
organizacja pamięci 130
otwieranie pliku 214, 226, 229, 322
oznacznik funkcji 265, 267

P

pamięć statyczna 118
parametry funkcji 19, 48, 51, 58, 66, 106, 115, 119, 120, 138, 154, 266, 300
Pascal 24, 41, 47, 51, 56, 91, 94, 107, 119, 173, 198
perror, funkcja standardowa 331
pętle 29, 34, 40, 71, 90, 96, 298
–, inicjowanie 34, 91
–, przyrost 34, 91
–, treść 29, 34, 38, 95
–, warunek 29, 32, 34, 40, 91
plik wykonywalny a.out 24, 104
pliki 36, 214, 322
–, atrybuty 239
–, binarne 215, 323
–, dopisywanie danych 232
–, dostęp sekwencyjny 214, 232
–, – swobodny 232
–, kopiowanie 36, 227, 230
pliki nagłówkowe 19, 58, 62, 117, 321
–, obsługa 214, 225, 322
–, otwieranie 214, 226, 229, 322
–, położenie 238
– pośrednie 103
–, przewijanie 232
–, rodzaj dostępu 214, 229, 233
–, skorowidz 237
– standardowe 215, 217, 226
– tekstowe 215
–, tworzenie 215, 226, 229, 230
–, uprawnienia 215, 230
–, wskaźniki 214, 217, 233, 322
–, wstawiania 125, 129
–, zamykanie 217, 231, 323
– źródłowe 18, 24, 48, 54, 58, 99, 103, 115, 125, 205
pola bitowe 201, 202, 281
– –, kolejność bitów 202
pole wejściowe 211
– wyjściowe 206
położenie w pamięci 191, 200, 223, 245, 262, 272
pop, funkcja 109
porządkowanie metodą drzew binarnych 187
– – sąsiednich zamian 92
– – „quicksort" 123, 148, 151
– – Shell-sort 92, 148
Postscript 108
potok 204, 215, 218
pow, funkcja standardowa 47, 335
power, funkcja 47, 50
powielanie bitu znaku 71, 72, 234, 254
#pragma, instrukcja preprocesora 311
precyzja 32, 206, 325
preprocesor 18, 99, 125, 183, 197, 305
printd, funkcja 123
printf, funkcja standardowa 25, 30, 39, 205, 206, 208, 231, 325
priorytety operatorów 38, 67, 81, 154, 179, 263
program 24, 103, 251
–, analizator składniowy 168
–, argumenty wywołania 157
– cat 214, 216
– cp 230
– date 222
– dcl 166
– dir 237
–, dystrybutor pamięci 139, 191, 244
– echo 157
– find 159, 160

program fsize 240
– grep 100, 158
–, kalkulator 105, 108, 212
–, kopiowanie danych 36, 38, 227, 230
– lower 205
– ls 237
–, porządkowanie wierszy 149, 162
– undcl 171
– wc 42
–, wpisywanie najdłuższego wiersza 53, 56
–, wyszukiwanie wzorca 100, 101, 158, 159, 160
–, zestawienie temperatur 27, 31, 33
–, zliczanie słów 42, 189
–, – – kluczowych 180, 184
–, – wierszy 40
–, – znaków 39, 44, 88, 89
promocja typu 71, 72, 259, 260, 266
prototypy funkcji 49, 58, 73, 106, 125, 163, 186, 197, 227,266
przekształcenia typów 30, 32, 69, 71, 99, 259
– –, argumenty funkcji 30, 32, 73, 266
– – arytmetyczne 45, 71, 260
– – automatyczne 69, 72
– – całkowite 259
– –, promowanie 71, 259, 260, 266
– –, przypisania 72, 275
– –, reguły 69, 71, 72, 81, 260
– –, wskaźniki 261
– –, wymuszanie 73
przełączanie celu danych 204, 214, 226
– źródła danych 204, 214, 226
przenośność oprogramowania 19, 71, 77, 202, 203, 206, 237, 244, 248
przestrzeń nazw 297, 303
przeszukiwanie liniowe 191
– rozproszone 193
przypisanie 29, 32, 37, 41, 43, 122, 275
ptinrect, funkcja 177
ptrdiff-t, nazwa typu 143, 198, 271
push, funkcja 109
putc, funkcja standardowa 215, 329
putc, makro standardowe 234
putchar, funkcja standardowa 36, 127, 204, 216, 329
puts, funkcja standardowa 219, 329

Q

qsort, funkcja 124, 151, 163
qsort, funkcja standardowa 124, 338

R

raise, funkcja standardowa 341
rand, funkcja standardowa 74, 224, 336
RAND_MAX, stała symboliczna 336
read, funkcja systemowa 226
readdir, funkcja 242, 243
readlines, funkcja 150
realloc, funkcja standardowa 337
register, klasa pamięci 119, 131, 256, 278
rekurencja 122
remove, funkcja standardowa 231, 323
rename, funkcja standardowa 323
reprezentacja wewnętrzna cyfr 45
– – znaków 41
return, instrukcja 49, 54, 89, 103, 299
reverse, funkcja 93
rewind, funkcja standardowa 330
rodzaj dostępu do pliku 214
rozbiór gramatyczny deklaracji 168
rozmiar danych 61
– pola 201
– struktury 186
– tablicy 183
rozszerzenia języka 125
równoważność typów 295
rzut 270
rzutowanie 73, 223, 270

S

sbrk, funkcja systemowa 248
scanf, funkcja standardowa 30, 134, 210, 327
sekwencje specjalne 26, 41, 63, 254
– –, \a znak alarmu 63, 254
– –, \b znak cofania 26, 63
– –, \f znak nowej strony 63, 254
– –, \n znak nowego wiersza 26, 41, 63, 254, 322
– –, \r znak powrotu karetki 63, 254
– –, \t znak tabulacji poziomej 26, 30, 63, 254
– –, \v znak tabulacji pionowej 63, 254
– –, \\ znak \ 26, 63, 254
– –, \? znak zapytania 63, 254
– –, \' znak apostrofu 63, 254
– –, \" znak cudzysłowu 26, 63, 254
– –, *ooo* liczba ósemkowa 63, 254
– –, *xhh* liczba szesnastkowa 63, 254
– trzyznakowe 305
setbuf, funkcja standardowa 324
setjmp, funkcja standardowa 340

setjmp.h, nagłówek standardowy 321, 340
setvbuf, funkcja standardowa 324
shell, interpretator poleceń 226
shellsort, funkcja 95
short, typ danych 28, 61, 71, 76, 130, 279
signal, funkcja standardowa 340
signal.h, nagłówek standardowy 321, 340
signed, typ danych 62, 71, 279
sin, funkcja standardowa 104, 224, 334
sinh, funkcja standardowa 334
sizeof, operator rozmiaru 82, 143, 183, 187, 269
size_t, nazwa typu 143, 183, 198, 270, 322
skalowanie, arytmetyka na adresach 142
sklejanie argumentów 128
– napisów 60, 64, 127, 307
– wierszy 306
składnia 50, 255, 312
– deklaracji 131
– wyrażenia 131
składowe struktur 174, 266, 280
– unii 199, 267, 280
skorowidz plików 237
słowa kluczowe 61, 252
słowo maszynowe 201
– –, kolejność bitów 202
sort, program 148
specyfikacje przekształceń 30, 32, 39, 206, 210, 324, 327
sprintf, funkcja standardowa 208, 231, 325
sqrt, funkcja standardowa 73, 104, 224, 335
squeeze, funkcja 75
srand, funkcja standardowa 224, 336
sscanf, funkcja standardowa 210, 327
stałe 23, 32, 60, 62, 253
– bez znaku liczby 60
– całkowite 45, 62, 253
– – ósemkowe 62, 253
– – szesnastkowe 60, 62, 253
– dziesiętne 253
– napisowe 25, 41, 54, 60, 63, 64, 143, 255
– symboliczne 35, 37, 43, 204
– wieloznakowe 254
– wyliczeń 65, 255
– zmiennopozycyjne 32, 62, 255
– znakowe 60, 63, 254
– – rozszerzone 254
standard ANSI 13, 18, 30, 60, 99, 130, 173, 203, 225, 251
stat, funkcja systemowa 231, 239
stat, struktura systemowa 239
static, klasa pamięci 58, 118, 120, 256, 278, 302
stdarg.h, nagłówek standardowy 208, 321, 339
stddef.h, nagłówek standardowy 143, 183, 321
stderr, wskaźnik pliku standardowego 215, 217, 322
stdin, wskaźnik pliku standardowego 215, 217, 322
stdio.h, nagłówek standardowy 25, 37, 125, 127, 204, 228, 233, 321, 322
stdlib.h, nagłówek standardowy 104, 192, 321, 335
stdout, wskaźnik pliku standardowego 215, 217, 322
stos 108, 140, 147
strcat, funkcja standardowa 76, 221, 333
strchr, funkcja standardowa 221, 333
strcmp, funkcja standardowa 146, 221, 333
strcpy, funkcja standardowa 144, 221, 333
strcspn, funkcja standardowa 333
strdup, funkcja 190, 192
strerror, funkcja standardowa 333
strftime, funkcja standardowa 342
strindex, funkcja 101
string.h, nagłówek standardowy 65, 146, 159, 221, 321, 332
strlen, funkcja standardowa 64, 138, 142, 221, 333
strncat, funkcja standardowa 147, 221, 333
strncmp, funkcja standardowa 147, 221, 333
strncpy, funkcja standardowa 147, 221, 333
strpbrk, funkcja standardowa 76, 333
strrchr, funkcja standardowa 221, 333
strspn, funkcja standardowa 333
strstr, funkcja standardowa 101, 159, 333
strtod, funkcja standardowa 336
strtok, funkcja standardowa 333
strtol, funkcja standardowa 92, 333
strtoul, funkcja standardowa 336
struct, słowo kluczowe 174, 280
struct, typ danych 174
struktura blokowa 120
– programu 29, 47, 92, 97, 108
struktury 17, 28, 99, 173, 258, 267
–, argumenty funkcji 176
–, deklaracje 174, 280
–, etykietki 174, 214, 281, 303
–, inicjowanie 175, 292
–, pola bitowe 201, 281
–, przekazywanie przez wartość 177
–, przypisania 173, 176
– rekurencyjne 124, 189

struktury, składowe 174, 267, 280
– zagnieżdżone 175, 199
strumień binarny 322
– danych 205
– tekstowy 322
– wyjściowy 206
– znaków 36, 40, 203, 215
styl programowania 15, 29, 34, 38, 46, 50, 59, 83, 90, 91, 98, 121, 187
swap, funkcja 124, 133, 151, 165
switch, instrukcja 46, 88, 96, 110, 297
syscalls.h, nagłówek 227
sys/dir.h, nagłówek systemowy 242
sys/file.h, nagłówek systemowy 229
sys/stat.h, nagłówek systemowy 239
system, funkcja standardowa 222, 337
system operacyjny Unix 13, 15, 17, 19, 24, 42, 100, 103, 148, 158, 215, 222, 225, 229, 248, 322
system plików 225, 230, 238

Ś

średnik, ogranicznik instrukcji 84

T

tablice 17, 28, 44, 52, 64, 66, 130, 258, 287
–, automatyczne 99
–, definicje 116
–, deklaracje 45, 152, 287
–, indeksowanie 45, 69, 136, 153, 265, 288
–, inicjowanie 45, 122, 154, 155, 291
–, położenie w pamięci 144
–, przeglądanie 193
–, rozmiar 183
–, – domyślny 122, 155, 291
– statyczne 155
– struktur 180
– wielowymiarowe 152, 187
–, wskaźniki 51, 136, 154
– wskaźników 148
– znakowe 52, 122, 143
talloc, funkcja 190, 192
tan, funkcja standardowa 334
tanh, funkcja standardowa 335
time, funkcja standardowa 342
time.h, nagłówek standardowy 321, 341
time_t, nazwa typu 341
tłumaczenie programu 24, 48, 78, 99, 103, 106, 114, 125, 251, 303
tłumaczenie programu oddzielne 99, 303
– programów starszych 106
tmpfile, funkcja standardowa 323
tmpnam, funkcja standardowa 323
tolower, funkcja standardowa 205, 222, 332
toupper, funkcja standardowa 222, 332
treeprint, funkcja 191
treść pętli 29, 38, 40
trim, funkcja 96
typedef, nazywanie typów 166, 196, 278, 294
typy danych 17, 28, 37, 54, 60, 239, 279
– – arytmetyczne 60, 258
– – całkowite 37, 60, 61, 62, 258
– – niekompletne 281, 287
– – pochodne 17, 258
– – podstawowe 28, 60, 61, 257
– –, synonimy 196, 294
– –, unikalne 282
– – wyliczeniowe 258
– – zmiennopozycyjne 258

U

ukrywanie nazw 118
ULONG_MAX, stała symboliczna 336
undcl, program 171
#undef, instrukcja procesora 127, 228, 306
ungetc, funkcja standardowa 114, 222, 329
ungetch, funkcja 112
unie 17, 28, 198, 258, 267
–, deklaracje 198, 280
–, etykietki 281, 303
–, inicjowanie 200, 291
–, przypisywanie 200
–, składowe 267, 280
union, słowo kluczowe 280
Unix Berkeley (BSD), system operacyjny 229
Unix, system operacyjny 13, 15, 17, 19, 24, 42, 100, 103, 148, 158, 215, 222, 225, 229, 248, 322
Unix System V, system operacyjny 229, 238, 242
Unix Version 7, system operacyjny 238, 242
unlink, funkcja systemowa 231
unsigned char, typ danych 62, 228, 257
unsigned, typ danych 62, 71, 77, 79, 257, 279

V

va_arg, makro standardowe 209, 326, 339
va_end, makro standardowe 209, 339

va_list, typ danych 209, 339
va_start, makro standardowe 209, 231, 326, 339
void*, wskaźnik do typu void 130, 143, 163, 262
void, typ danych 54, 130, 192, 258, 262, 279, 289, 300
volatile, kwalifikator typu 258, 280
vfprintf, funkcja standardowa 231, 326
vprintf, funkcja standardowa 231, 326
vsprintf, funkcja standardowa 231, 326

W

wartość rozproszenia 193
wartości logiczne 43, 71
warunek 29, 32, 40, 46, 91, 94
– pętli 32, 34, 40, 91
– zatrzymania 29, 94
wchar_t, nazwa typu 254
wejście formatowane 210
wejście-wyjście 18, 23, 30, 35, 203, 225
węzły w drzewie binarnym 187
while, instrukcja pętli 29, 33, 40, 71, 90, 96, 298
wiersze 36, 40, 52, 203
włączanie plików 305, 309
write, funkcja systemowa 226
writelines, funkcja 150
wskaźnik pliku 214, 233
– pusty NULL 141, 261
wskaźniki 17, 28, 51, 130, 136, 258, 286
–, arytmetyka 130, 137, 139, 143, 162, 186, 246, 271
–, deklaracje 131, 286
– do funkcji 162
– do plików 322
– do struktur 178, 184
– do tablic 136, 154
– do typu void 130, 143, 163, 262
– do wskaźników 148, 158
–, inicjowanie 141, 155, 291
– ogólne 263
wstawianie plików 125, 129
wyjście formatowane 206
wykrywanie błędów 50
wyliczenia 19, 65, 258, 284
wyliczniki 255, 284
wyrażenia 17, 34, 38, 43, 60, 81, 263
– arytmetyczne 69
wyrażenia całkowite 45
–, kolejność obliczeń 43, 82, 93, 263
– logiczne 43, 69, 71, 153
– proste 264
– przypisania 17, 37, 43, 79, 275
– przyrostkowe 265
– relacyjne 69, 71
– stałe 64, 88, 121, 175, 276
– warunkowe 80, 91
– wrostkowe 108
wyszukiwanie metodą bisekcji 87, 181
wywołanie funkcji 25, 51, 56, 83, 108, 163, 265, 301
wzorzec bitowy 37, 63, 71

Z

zakres danych 28
zamiana deklaracji C na opis 166
zamykanie plików 231
zasięg deklaracji 300
– etykiety 97
– leksykalny 303
– nazwy 99, 114, 126, 303
– zmiennych 55, 108, 118
zbiór ISO 646-1983 Invariant Code Set 305
– znaków 41, 45, 305
– – ASCII 41, 63, 70, 305, 332
– – EBCDIC 70
– – rozszerzony 254
zgodność typów 302
zmienne 18, 23, 24, 28, 60, 66, 103, 107, 118, 119
– automatyczne 56, 66, 108, 115, 120
– definiowane 107
– globalne 18, 56
– lokalne 48, 55, 61, 115
– rejestrowe 119
– statyczne 55, 66, 119
– tymczasowe 51, 111
– wewnętrzne 18, 119
– zewnętrzne 18, 55, 58, 66, 103, 107
zmniejszanie o jeden 267
znacznik końca pliku 37, 114, 204
znak liczby 60, 71, 76
– pusty \0 54, 64, 254
znaki przekształceń 206, 211, 324, 327
zwiększanie o jeden 267

WYDAWNICTWA NAUKOWO-TECHNICZNE
00-048 Warszawa, Mazowiecka 2/4
TEL./FAX 826-82-93

Zamawiam za zaliczeniem pocztowym następujące książki:

• *Arnold K., Gosling J.*: **Java™**	42,00 zł	... egz.
• *Banachowski L., Diks K., Rytter W.*: **Algorytmy i struktury danych**	28,00 zł	... egz.
• *Ben-Ari M.*: **Podstawy programowania współbieżnego i rozproszonego**	18,00 zł	... egz.
• *Beynon-Davies P.*: **Inżynieria systemów informacyjnych**	27,50 zł	... egz.
• *Beynon-Davies P.*: **Systemy baz danych**	35,00 zł	... egz.
• *Białasiewicz J.T.*: **Falki i aproksymacje**	36,00 zł	... egz.
• *Boone B.*: **Java™ dla programistów C i C++**	38,00 zł	... egz.
• *Comer D.E.*: **Sieci komputerowe i intersieci**	45,00 zł	... egz.
• *Comer D.E.*: **Sieci komputerowe TCP/IP T.1 – Zasady, protokoły i architektura**	52,00 zł	... egz.
• *Cormen T.M., Leiserson Ch.E., Rivest R.L.*: **Wprowadzenie do algorytmów**	117,00 zł	... egz.
• *Coulouris G., Dollimore J., Kindberg T.*: **Systemy rozproszone. Podstawy i projektowanie**	79,00 zł	... egz.
• *Date C.J., Darwen H..*: **SQL. Omówienie standardu języka**	78,00 zł	... egz.
• *Drozdek A.*: **Wprowadzenie do kompresji danych**	21,50 zł	... egz.
• *Drozdek A., Simon D.L.*: **Struktury danych w języku C**	34,00 zł	... egz.
• *Goldberg D.E.*: **Algorytmy genetyczne i ich zastosowania**	24,00 zł	... egz.
• *Kiciak P.*: **Podstawy modelowania krzywych i powierzchni. Zastosowania w grafice komputerowej**	31,50 zł	... egz.
• *Lausen G., Vossen G.*: **Obiektowe bazy danych**	40,00 zł	... egz.
• *Ledgard H.F.*: **Mała księga programowania obiektowego**	20,00 zł	... egz.
• *Lippman S.B.*: **Model obiektu w C++**	28,00 zł	... egz.
• *Magid L.J.*: **Mała księga komputerów PC** (Książka wyjaśnia wszystko, co powinien wiedzieć na początku użytkownik komputera – począwszy od kupna komputera, a skończywszy na używaniu oprogramowania)	19,50 zł	... egz.
• *Majidimehr A.H.*: **Optymalizacja systemu UNIX**	44,00 zł	... egz.
• *Meyers S.*: **Język C++ bardziej efektywny**	28,00 zł	... egz.
• *Osowski S.*: **Sieci neuronowe w ujęciu algorytmicznym** (+ dyskietka)	33,00 zł	... egz.
• *Plauger P.J.*: **Biblioteka standardowa C++**	65,00 zł	... egz.
• *Poe V., Klauer P., Brobst S.*: **Tworzenie hurtowni danych**	39,00 zł	... egz.
• **Przewodnik administratora systemu UNIX®**. Praca zbiorowa	130,00 zł	... egz.
• *Pugh K.*: **Unix® dla użytkownika DOS-u**	19,00 zł	... egz.
• *Ribenboim P.*: **Mała księga wielkich liczb pierwszych**	20,50 zł	... egz.
• *Robertson J., Robertson S.*: **Pełna analiza systemowa**	75,00 zł	... egz.
• *Schneier B.*: **Ochrona poczty elektronicznej. Jak chronić prywatność korespondencji w sieci internet?**	39,00 zł	... egz.
• *Silberschatz A., Galvin P.B.*: **Podstawy systemów operacyjnych**	70,00 zł	... egz.
• *Southerton A., Perkins E.C., Jr.*: **Słownik poleceń systemów Unix i X**	43,00 zł	... egz.
• *Stallings W.*: **Ochrona danych w sieci i intersieci**	38,00 zł	... egz.
• *Stallings W.*: **Organizacja i architektura systemu komputerowego. Projektowanie systemu a jego wydajność**	60,00 zł	... egz.

- *Stevens W.R.*: **Programowanie zastosowań sieciowych w systemie Unix** 42,00 zł ... egz.
- *Stevens W.R.*: **UNIX. Programowanie usług sieciowych. T. 1 – API: gniazda i XTI** 75,50 zł ... egz.
- *Stroustrup B.*: **Projektowanie i rozwój języka C++** 35,00 zł ... egz.
- *Weiskamp K.*: **Borland Pascal 7.0** 34,00 zł ... egz.
- *Wieczorkowski R., Zieliński R.*: **Komputerowe generatory liczb losowych** 11,00 zł ... egz.
- *Yourdon E., Argila C.*: **Analiza obiektowa i projektowanie** 43,00 zł ... egz.

Seria: KLASYKA INFORMATYKI

- *Harel D.*: **Rzecz o istocie informatyki. Algorytmika** 52,00 zł ... egz.
- *Stroustrup B.*: **Język C++** 132,00 zł ... egz.
- *Wirth N.*: **Algorytmy + struktury danych = programy** 61,00 zł ... egz.

Seria: LUDZIE, KOMPUTERY, INFORMACJA

- *Brooks F.P.Jr.*: **Mityczny osobomiesiąc. Eseje o inżynierii oprogramowania** 45,00 zł ... egz.
- *Yourdon E.*: **Marsz ku klęsce. Przewodnik dla projektanta systemów** 35,00 zł ... egz.

Zamówienie na wybrane książki proszę wysłać pod adresem:
Dział Handlowy Wydawnictw Naukowo-Technicznych
Skrytka pocztowa 359, 00-950 Warszawa

Zamówienia przyjmujemy również za pośrednictwem poczty elektronicznej.
Nasz adres: marketing@wnt.com.pl

Zamawiający:

Imię i nazwisko . . . . . .

Adres . . . . . .

. . . . . .

NIP . . . . . .

Przy zakupie książek o wartości przekraczającej 200,00 zł udzielamy 10% rabatu.

Zapraszamy do naszej księgarni internetowej – **www.wnt.com.pl**

Powyższe ceny obowiązują do wyczerpania nakładu danego tytułu

Wyrażam zgodę (nie wyrażam) na wykorzystanie przez WNT moich danych osobowych do celów marketingowych (głównie wysyłanie ofert reklamowych)

....

data podpis