Język C

w pigułce

Peter Prinz, Tony Crawford

przekład: Joanna Zatorska,
Leszek Biolik

APN Promise
Warszawa 2016

O'REILLY®

Język C w pigułce

© 2016 APN PROMISE SA

Authorized translation of English edition of
C in a nutshell
ISBN 978-1-491-90475-6
Copyright © 2016 Peter Prinz and Tony Crawford. All rights reserved.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls of all rights to publish and sell the same.

APN PROMISE SA, biuro: ul. Domaniewska 44a, 02-672 Warszawa
tel. +48 22 35 51 600, fax +48 22 35 51 699
e-mail: mspress@promise.pl

Wszystkie prawa zastrzeżone. Żadna część niniejszej książki nie może być powielana ani rozpowszechniana w jakiejkolwiek formie i w jakikolwiek sposób (elektroniczny, mechaniczny), włącznie z fotokopiowaniem, nagrywaniem na taśmy lub przy użyciu innych systemów bez pisemnej zgody wydawcy.

Logo O'Reilly jest zarejestrowanym znakiem towarowym O'Reilly Media, Inc. *Fluent Python*, ilustracja z okładki i powiązane elementy są znakami towarowymi O'Reilly Media, Inc.

Wszystkie inne nazwy handlowe i towarowe występujące w niniejszej publikacji mogą być znakami towarowymi zastrzeżonymi lub nazwami zastrzeżonymi odpowiednich firm odnośnych właścicieli.

Przykłady firm, produktów, osób i wydarzeń opisane w niniejszej książce są fikcyjne i nie odnoszą się do żadnych konkretnych firm, produktów, osób i wydarzeń. Ewentualne podobieństwo do jakiejkolwiek rzeczywistej firmy, organizacji, produktu, nazwy domeny, adresu poczty elektronicznej, logo, osoby, miejsca lub zdarzenia jest przypadkowe i niezamierzone.

APN PROMISE SA dołożyła wszelkich starań, aby zapewnić najwyższą jakość tej publikacji. Jednakże nikomu nie udziela się rękojmi ani gwarancji.
APN PROMISE SA nie jest w żadnym wypadku odpowiedzialna za jakiekolwiek szkody będące następstwem korzystania z informacji zawartych w niniejszej publikacji, nawet jeśli APN PROMISE została powiadomiona o możliwości wystąpienia szkód.

ISBN: 978-83-7541-161-4

Projekt okładki: Karen Montgomery
Ilustracje: Rebecca Demarest

Przekład: Joanna Zatorska, Leszek Biolik
Redakcja: Marek Włodarz
Korekta: Ewa Swędrowska
Skład i łamanie: MAWart Marek Włodarz

Spis treści

Wstęp . . . . . *ix*

Część I: Język

1 Podstawy języka 3

Cechy języka C 3
Struktura programów w języku C 4
Pliki źródłowe 6
Komentarze 7
Zbiory znaków 9
Identyfikatory 15
Jak działa kompilator C 20

2 Typy 25

Typologia 25
Typy całkowite 26
Typy zmiennopozycyjne 33
Zespolone typy zmiennopozycyjne 36
Typy wyliczeniowe 37
Typ void 38
Wyrównanie obiektów w pamięci 40

3 Literały 41

Stałe całkowite 41
Stałe zmiennopozycyjne 42
Stałe znakowe 44
Literały łańcuchowe 48

4 Przekształcenia typów 53

Przekształcenia typów arytmetycznych 54
Przekształcenia typów niearytmetycznych 62
Jawne przekształcenia wskaźników 64

5 Wyrażenia i operatory 71
Jak są przetwarzane wyrażenia 72
Szczegóły dotyczące operatorów 78
Wyrażenia stałe 104

6 Instrukcje 107
Instrukcje wyrażeniowe 107
Instrukcje blokowe 108
Pętle 109
Instrukcje sterujące 114
Przeskoki bezwarunkowe 118

7 Funkcje 123
Definicje funkcji 123
Deklaracje funkcji 131
Jak są wykonywane funkcje 133
Wskaźniki jako argumenty i zwracane wartości 133
Funkcje otwarte 135
Funkcje, które niczego nie zwracają 137
Funkcje rekurencyjne 137
Zmienna liczba argumentów 139

8 Tablice 141
Definiowanie tablic 141
Dostęp do elementów tablicy 143
Inicjalizowanie tablic 144
Łańcuchy 146
Tablice wielowymiarowe 149
Tablice jako argumenty funkcji 151

9 Wskaźniki 155
Deklarowanie wskaźników 155
Operacje ze wskaźnikami 159
Wskaźniki i kwalifikatory typu 163
Wskaźniki do tablic i tablice wskaźników 167
Wskaźniki do funkcji 172

10 Struktury, unie i pola bitowe 175
Struktury 175
Unie 187
Anonimowe struktury i unie 189

Pola bitowe 190

11 Deklaracje 193

Deklaracje obiektów i funkcji 194
Nazwy typów 202
Deklaracje typedef 203
Deklaracje _Static_assert 205
Łączność identyfikatorów 206
Czas trwania obiektów w pamięci 208
Inicjalizacja 209

12 Dynamiczne zarządzanie pamięcią 211

Dynamiczne alokowanie pamięci 212
Cechy zaalokowanej pamięci 214
Zmiana ilości i zwalnianie pamięci 214
Drzewo binarne ogólnego przeznaczenia 216

13 Wejście i wyjście 229

Strumienie 230
Pliki 231
Otwieranie i zamykanie plików 234
Odczytywanie i zapisywanie 236
Bezpośredni dostęp do pliku 257

14 Wielowątkowość 261

Wątki 262
Dostęp do danych współdzielonych 267
Komunikacja między wątkami: zmienne warunkowe 275
Obiekty lokalne wątku i pamięć własna wątku 280

15 Dyrektywy preprocesora 285

Włączanie zawartości plików nagłówkowych 286
Definiowanie i używanie makr 288
Makra generyczne 296
Kompilacja warunkowa 297
Definiowanie numerów wierszy 299
Generowanie komunikatów błędu 300
Dyrektywa #pragma 300
Operator _Pragma 301
Makra predefiniowane 302

Część II: Biblioteka standardowa

16 Nagłówki standardowe 307

Wykorzystywanie nagłówków standardowych 308
Funkcje ze sprawdzaniem zakresów 312
Zawartość nagłówków standardowych 313

17 Rzut oka na funkcje 347

Wejście i wyjście 347
Funkcje matematyczne 349
Klasyfikacja i przekształcanie znaków 357
Przetwarzanie łańcuchów 359
Znaki wielobajtowe 361
Przekształcenia między liczbami i łańcuchami 362
Wyszukiwanie i sortowanie 364
Obsługa bloku pamięci 364
Dynamiczne zarządzanie pamięcią 365
Data i czas 365
Sterowanie procesem 367
Internacjonalizacja 368
Nielokalne skoki 369
Wielowątkowość (C11) 370
Debugowanie 374
Komunikaty błędu 375

18 Funkcje biblioteki standardowej 377

Część III: Podstawowe narzędzia

19 Kompilowanie z wykorzystaniem GCC 745

GNU Compiler Collection 745
Pobieranie i instalowanie GCC 746
Kompilowanie programów C za pomocą GCC 747
Dialekty C 759
Optymalizacja 762
Debugowanie 766
Profilowanie 767
Podsumowanie opcji i zmiennych środowiskowych 767

20 Wykorzystanie ***make*** do budowania programów C 773

Elementy docelowe, zależności i polecenia 774
Makefile 774
Reguły 775
Komentarze 782
Zmienne 783
Cele pozorne 791
Inne atrybuty elementu docelowego 793
Makra 794
Funkcje 795
Dyrektywy 800
Uruchamianie make 804

21 Debugowanie programów C za pomocą GDB 813

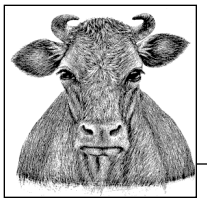
Instalowanie GDB 814
Przykładowa sesja debugowania 814
Uruchamianie GDB 819
Korzystanie z poleceń GDB 825
Analiza stosu 838
Analiza plików zrzutu pamięci w GDB 850

22 Tworzenie programów C za pomocą IDE 853

Środowiska IDE dla C 853
Eclipse dla C/C++ 854
Tworzenie programu C za pomocą Eclipse 856
Debugowanie programu C w Eclipse 860
Więcej informacji na temat środowiska Eclipse 863

Indeks *865*

O autorach *885*



Wstęp

Ta książka jest kompletnym źródłem informacji o języku programowania C i bibliotece czasu wykonania C. Z racji tego, że jest to zbiór informacji „w pigułce", książka ta pretenduje do miana wygodnego i wiarygodnego towarzysza codziennej pracy programistów C. Opisuje wszystkie elementy języka i ilustruje ich użycie za pomocą wielu przykładów.

Obecny opis języka C jest oparty na międzynarodowym standardzie C ISO/IEC 9899:2011 z 2011 roku, znanym szerzej jako C11. Zastępuje on wcześniejszy standard C99, czyli ISO/IEC 9899:1999, oraz poprawki Technical Corrigenda, TC1 z 2001 roku, TC2 z 2004 roku i TC3 z 2007 roku. Pierwszy międzynarodowy standard C, czyli ISO/IEC 9899:1990, został opublikowany w 1990 roku i uzupełniony dodatkiem Normative Addendum 1 (ISO/IEC 9899/AMD1:1995) z 1995 roku. Standard 1990 ISO/IEC odpowiada standardowi ANSI X3.159, który został ratyfikowany pod koniec 1989 roku i jest popularnie nazywany ANSI C lub C89.

Nowe funkcjonalności standardu C z 2011 roku nie są jeszcze w pełni wspierane przez wszystkie kompilatory i implementacje biblioteki standardowej. Dlatego, w tej książce, wszystkie nowości z roku 2011, takie jak wielowątkowość, makra generyczne i nowe funkcje biblioteki standardowej, oznaczyliśmy skrótem C11. Rozszerzenia wprowadzone w standardzie C99 zostały oznaczone skrótem C99.

Ta książka nie jest wstępem do programowania w C. Chociaż opisuje podstawy języka, nie jest zorganizowana, ani napisana jak przewodnik. Czytelnicy, którzy dopiero rozpoczynają przygodę z C, powinni wcześniej przeczytać choć jedną z wielu dostępnych książek, opisujących podstawy języka, lub powinni znać jeden z pokrewnych języków, takich jak Java lub C++.

Organizacja książki

Tak książka jest podzielona na trzy części. Pierwsza część opisuje język C w ścisłym znaczeniu; druga część zawiera omówienie biblioteki standardowej; natomiast trzecia część opisuje proces kompilacji i testowania programów za pomocą popularnych narzędzi z kolekcji oprogramowania GNU.

Część I

Część pierwsza, dotycząca języka C, zawiera rozdziały od 1. do 15. Rozdział 1. opisuje ogólne koncepcje oraz elementy języka, natomiast każdy kolejny jest poświęcony określonemu zagadnieniu, na przykład typom, instrukcjom lub wskaźnikom. Chociaż tematy są opisane w takiej kolejności, że podstawowe koncepcje każdego zagadnienia zostały przedstawione we wcześniejszym rozdziale – na przykład typy są opisane przed wyrażeniami i operatorami, które z kolei poprzedzają opis instrukcji, itd. – czasem zrozumienie wszystkich szczegółów wymaga zapoznania się z zagadnieniami z późniejszych rozdziałów, zgodnie z odwołaniami umieszczonymi w tekście. Na przykład, w rozdziale 5. (opisującym wyrażenia i operatory) konieczne jest zrozumienie pewnych koncepcji dotyczących wskaźników i tablic, chociaż wskaźniki i tablice są szczegółowo opisane dopiero w rozdziałach 8. i 9.

Rozdział 1, „Podstawy języka"

Opisuje cechy języka oraz strukturę i kompilowanie programów C. Ten rozdział wprowadza podstawowe koncepcje, takie jak jednostka tłumaczenia, zestaw znaków i identyfikatory.

Rozdział 2, „Typy"

Zawiera omówienie typów w języku C oraz opisuje typy podstawowe, typ `void` i typy wyliczeniowe.

Rozdział 3, „Literały"

Opisuje stałe liczbowe, stałe znakowe oraz literały łańcuchowe, włącznie z sekwencjami specjalnymi.

Rozdział 4, „Przekształcenia typów"

Opisuje niejawne i jawne przekształcenia typów, włącznie z promocją całkowitą i zwykłymi przekształceniami arytmetycznymi.

Rozdział 5, „Wyrażenia i operatory"

Opisuje przetwarzanie wyrażeń, wszystkie operatory oraz ich operandy.

Rozdział 6, „Instrukcje"

Opisuje instrukcje C, takie jak bloki, pętle i skoki.

Rozdział 7, „Funkcje”

Opisuje definicje funkcji i wywołania funkcji, włącznie z funkcjami rekurencyjnymi i otwartymi.

Rozdział 8, „Tablice”

Opisuje tablice o stałej i zmiennej długości, włącznie z łańcuchami, inicjalizacją tablic i tablicami wielowymiarowymi.

Rozdział 9, „Wskaźniki”

Opisuje definicje i wykorzystanie wskaźników do obiektów i funkcji.

Rozdział 10, „Struktury, unie i pola bitowe”

Opisuje organizację danych w typach zdefiniowanych przez użytkownika.

Rozdział 11, „Deklaracje”

Opisuje ogólną składnię deklaracji, łączność identyfikatorów i czas trwania obiektów.

Rozdział 12, „Dynamiczne zarządzanie pamięcią”

Opisuje funkcje biblioteki standardowej, służące do dynamicznego zarządzania pamięcią, ilustruje ich użycie w przykładowej implementacji drzewa binarnego ogólnego przeznaczenia.

Rozdział 13, „Wejście i wyjście”

Omawia koncepcję wejścia i wyjścia w języku C. Zawiera opis wykorzystania standardowej biblioteki wejścia-wyjścia.

Rozdział 14, „Wielowątkowość”

Opisuje wykorzystanie wielowątkowości w standardzie C11. Znajdziemy tu opis użycia operacji niepodzielnych, komunikacji między wątkami i pamięci własnej wątku.

Rozdział 15, „Dyrektywy preprocesora”

Opisuje definiowanie i wykorzystanie makr, kompilację warunkową i wszystkie pozostałe dyrektywy i operatory preprocesora.

Część II

Część II, zawierająca rozdziały 16., 17. i 18., jest poświęcona standardowej bibliotece C. Zawiera przegląd nagłówków standardowych i zawiera także szczegółowy opis funkcji.

Rozdział 16, „Nagłówki standardowe”

Opisuje zawartość nagłówków i ich wykorzystanie. Nagłówki zawierają wszystkie definicje makr i typów biblioteki standardowej.

Rozdział 17, „Rzut oka na funkcje"

Zawiera przegląd funkcji biblioteki standardowej, zorganizowany według obszarów zastosowań (np. funkcje matematyczne, funkcje przetwarzające daty i czas, itd.).

Rozdział 18, „Funkcje biblioteki standardowej"

Opisuje szczegółowo każdą funkcję w kolejności alfabetycznej i zawiera przykłady ilustrujące wykorzystanie każdej funkcji.

Część III

Trzecia część tej książki, obejmująca rozdziały od 19. do 22., dostarcza niezbędnej wiedzy na temat podstawowych narzędzi programisty C: kompilatora, narzędzia *make* i debugera. Opisane tu narzędzia należą do kolekcji oprogramowania GNU. Na końcu opisane zostało wykorzystanie tych narzędzi w zintegrowanym środowisku programistycznym (IDE) dla języka C. Jako przykład posłużyło środowisko Eclipse IDE.

Rozdział 19, „Kompilowanie z wykorzystaniem GCC"

Opisuje najważniejsze możliwości, oferowane przez popularne kompilatory C.

Rozdział 20, „Wykorzystanie make do budowania programów C"

Opisuje wykorzystanie programu *make* w celu automatyzacji procesu kompilacji dużych programów.

Rozdział 21, „Debugowanie programów C za pomocą GDB"

Opisuje uruchamianie programu pod kontrolą debugera GNU. Pokazuje, jak analizować zachowanie programów w czasie wykonania w celu znalezienia błędów logicznych.

Rozdział 22, „Tworzenie programów C za pomocą IDE"

Opisuje wykorzystanie zintegrowanego środowiska programistycznego (IDE), które umożliwia ujednolicony, wygodny dostęp do wszystkich narzędzi służących do tworzenia programów C.

Dalsza lektura

Oprócz źródeł wspomnianych w tekście książki, istnieje szereg zasobów, z których mogą skorzystać Czytelnicy, zainteresowani bardziej szczegółowymi zagadnieniami technicznymi. Międzynarodowa grupa robocza, zajmująca się standaryzacją C, ma oficjalną stronę internetową, dostępną pod adresem *http://www.open-std.org/jtc1/sc22/wg14*, która zawiera łącza do najnowszej wersji standardu C oraz do bieżących projektów prowadzonych przez grupę.

Czytelnicy zainteresowani nie tylko tym *co* i *jak* osiągnąć w C, lecz także *dlaczego* stosuje się określone mechanizmy, mogą zajrzeć na stronę WG14, zawierającą łącza do szkiców i projektów. Te dokumenty opisują niektóre inspiracje i ograniczenia związane z procesem standaryzacji. Ponadto, Czytelnicy zainteresowani głównie tym, jak C „wyewoluował do obecnej postaci", zainteresują się zapewne artykułem twórcy języka C, Dennisa Ritchiego, zatytułowanym „The Development of the C Language" (*https://www.bell-labs.com/usr/dmr/www/chist.html*). Ten oraz inne dokumenty historyczne są nadal dostępne na stronie laboratorium Bell Labs *https://www.bell-labs.com/usr/dmr/www/index.html.*

Czytelnicy, zainteresowani szczegółami działań matematycznych na liczbach zmiennopozycyjnych, które wykraczają poza zakres C, być może zechcą się zapoznać z gruntownym wstępem autorstwa Davida Goldberga „What Every Computer Scientist Should Know About Floating-Point Arithmetic", dostępnym pod adresem *http://docs.sun.com/source/806-3568/ncg_goldberg.html.*

Konwencje wykorzystywane w tej książce

W tej książce wykorzystywane są następujące konwencje typograficzne:

Kursywa

Stosowana do wyróżnienia nowej terminologii, nazw plików, rozszerzeń plików, adresów URL, katalogów i narzędzi uniksowych.

`Czcionka o stałej szerokości`

Stosowana do oznaczenia wszystkich elementów kodu źródłowego C: słów kluczowych, operatorów, zmiennych, funkcji, makr, typów, parametrów i literałów. Stosowana również do oznaczenia poleceń konsoli, opcji oraz wyniku działania tych poleceń.

`Pogrubiona czcionka o stałej szerokości`

Służy do oznaczenia w przykładowym kodzie omawianych funkcji lub instrukcji. W sesjach kompilatora, programu make i debugera ta czcionka służy do zaznaczenia poleceń, które muszą być wpisane przez użytkownika.

`Czcionka o stałej szerokości z kursywą`

Służy do zaznaczenia parametrów w prototypach funkcji lub wartości zastępczych, które muszą być zastąpione wartościami użytkownika.

Zwykły tekst

Służy do zapisu klawiszy, takich jak Return, Tab i Ctrl.



W ten sposób oznaczana jest wskazówka lub sugestia.



W ten sposób oznaczana jest ogólna uwaga.



W ten sposób oznaczane jest ostrzeżenie.

Korzystanie z przykładowego kodu

Materiały dodatkowe (przykładowy kod, ćwiczenia, itd.) są dostępne pod adresem *https://github.com/oreillymedia/c-in-a-nutshell-2E.*

Celem tej książki jest pomoc programiście w wykonaniu pracy. Ogólnie rzecz biorąc, Czytelnicy mogą wykorzystać kod znajdujący się w tej książce w swoich programach i dokumentacji. Nie ma konieczności kontaktowania się z nami w celu uzyskania zezwolenia, chyba że Czytelnik ma zamiar powielić znaczną część kodu. Na przykład, pisanie programu wykorzystującego kilka fragmentów kodu z tej książki nie wymaga zezwolenia. Sprzedaż lub dystrybucja przykładów z książek wydawnictwa O'Reilly na płytach CD wymaga zezwolenia. Odpowiedź na pytanie, wykorzystująca cytaty z książki wraz z przykładowym kodem, nie wymaga zezwolenia. Włączenie znacznej ilości przykładowego kodu z tej książki do dokumentacji własnego produktu wymaga zezwolenia.

Doceniamy, lecz nie wymagamy odnośników wskazujących źródło. Odnośnik taki powinien zawierać tytuł, autorów, wydawcę i numer ISBN oryginalnego, anglojęzycznego wydania książki. Może to na przykład wyglądać tak: „*C w pigułce*, autorstwa Petera Prinza i Tony'ego Crawforda, wydana przez O'Reilly Media, Inc., 978-1-491-90475-6 ".

Jeśli Czytelnik uważa, że wykorzystanie przez niego przykładowego kodu wykracza poza opisane tu przypadki dopuszczalnego użycia, prosimy o kontakt pod adresem *permissions@oreilly.com*.

Kontakt

Komentarze i pytania dotyczące tej książki należy kierować do wydawnictwa:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (numer dla Stanów Zjednoczonych lub Kanady)
707-829-0515 (numer międzynarodowy lub lokalny)
707-829-0104 (faks)

Przygotowaliśmy stronę internetową przeznaczoną dla tej książki. Można na niej znaleźć erratę, przykłady i inne dodatkowe informaje. Strona ta jest dostępna pod adresem *http://bit.ly/C_Nutshell_2e*.

Aby skomentować lub zadać techniczne pytanie dotyczące tej książki, należy wysłać wiadomość e-mail na adres bookquestions@oreilly.com.

Więcej informacji na temat naszych książek, kursów, konferencji i aktualności można znaleźć na naszej stronie, dostępnej pod adresem *http://www.oreilly.com*.

Podziękowania

Chcielibyśmy obaj podziękować wszystkim pracownikom wydawnictwa O'Reilly za ich wspaniałą pracę nad naszą książką, a w szczególności naszym redaktorom Rachel Roumeliotis i Katie Schooling za ich pomoc w trakcie pracy. Dziękujemy też naszym recenzentom technicznym, Mattowi Crawfordowi, Davidowi Kitabjianowi, Chrisowi LaPre, Johnowi C. Craigowi i Loïcowi Pefferkornowi za ich cenne uwagi krytyczne na temat rękopisu. Jesteśmy wdzięczni naszej redaktor technicznej, Kristen Brown oraz naszemu adiustatorowi Gillianowi McGarvey'emu, za ich zaangażowanie w ostateczny, dobry wygląd naszej książki. Na koniec pragniemy podziękować Jonathanowi Gennickowi za to, że wiele lat temu zapoczątkował ten projekt.

Peter

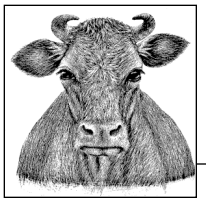
Chciałbym najpierw podziękować Tony'emu za wspaniałą współpracę. Serdecznie dziękuję także wszystkim moim przyjaciołom, za zrozumienie, jakie mi ciągle okazywali, gdy miałem dla nich tak mało czasu. Wreszcie, chciałem dodać, że dedykuję tę książkę moim córkom, Vivian i Jeanette – obydwie są obecnie na informatycznych studiach doktoranckich – które wspierały moje ambicje dotyczące tego projektu.

Tony

Dziękuję Peterowi za przestrzeń, którą mi udostępnił w tym projekcie, a którą mogłem zapełnić.

I

Język



1
Podstawy języka

Ten rozdział przedstawia podstawowe cechy i elementy języka programowania C.

Cechy języka C

C jest proceduralnym językiem programowania ogólnego stosowania. Język C został opracowany w latach 70-tych ubiegłego wieku przez Dennisa Ritchie'go w instytucie AT&T Bell Laboratories w Murray Hill, w stanie New Jersey. Język ten miał umożliwić implementację systemu operacyjnego Unix na dowolnych platformach sprzętowych. Język C nadaje się do tego szczególnie dobrze ze względu na swoje główne cechy:

- Przenośność kodu źródłowego
- Możliwość operowania „blisko maszyny"
- Wydajność

Te cechy pozwoliły programistom systemu Unix napisanie większej części systemu w języku C. Jedynie minimalna ilość kodu obsługi sprzętu specyficzna dla systemu musiała powstać w asemblerze.

Przodkiem języka C był beztypowy język BCPL (Basic Combined Programming Language), opracowany przez Martina Richardsa, oraz język B, opracowany przez Kena Thompsona, spadkobierca języka BCPL. Nową cechą języka C jest różnorodność dostępnych typów danych: znakowych, liczbowych, tablic, struktur i innych. W 1978 roku Brian Kernighan i Dennis Ritchie opublikowali oficjalny opis języka programowania C. Ich opis, będący de facto pierwszym standardem języka, jest określany po prostu jako

„K&R”[1]. Wysoka przenośność języka C wynika ze zwięzłości podstawowego języka, który jest zależny od sprzętu tylko w niewielu aspektach. Język C nie obsługuje na przykład dostępu do plików, czy też dynamicznego zarządzania pamięcią. Nie istnieją też żadne instrukcje odczytu i wypisywania danych w konsoli. Funkcje służące do wykonania tych zadań zostały zaimplementowane w obszernej bibliotece standardowej języka C.

Taka konstrukcja języka sprawia, że kompilator C jest względnie zwarty i łatwy do przeniesienia na nowe systemy. Ponadto po zainstalowaniu kompilatora w nowym systemie, większość funkcji z biblioteki standardowej nie wymaga żadnych modyfikacji, ponieważ zostały one napisane z wykorzystaniem przenośnego języka C. Dzięki temu kompilatory C są dostępne praktycznie dla każdego system operacyjnego.

Ponieważ język C został zaprojektowany specjalnie na potrzeby programowania systemu, nie należy się dziwić, że obecnie jest stosowany przeważnie w programowaniu systemów wbudowanych. Jednakże wielu programistów wykorzystuje język C jako przenośny, strukturalny język wysokiego poziomu do pisania takich programów, jak wydajne procesory tekstu, aplikacje bazodanowe czy graficzne.

Struktura programów w języku C

Proceduralnymi blokami tworzącymi program C są *funkcje*, które mogą wywoływać kolejne funkcje. W dobrze zaprojektowanym programie każda funkcja ma swoje przeznaczenie. Funkcje zawierają *instrukcje*, które są po kolei wykonywane przez program. Ponadto instrukcje mogą być także pogrupowane w *instrukcje blokowe*, zwane inaczej *blokami*. Programista ma również do dyspozycji gotowe funkcje z biblioteki standardowej. Może także pisać własne funkcje, jeśli żadna ze standardowych funkcji nie spełnia jego potrzeb. Oprócz biblioteki standardowej dostępne są także różnorodne wyspecjalizowane biblioteki, na przykład biblioteki zawierające funkcje graficzne. Jednakże używanie takich niestandardowych bibliotek wiąże się z ograniczeniem przenośności programu do tych systemów, do których biblioteki te zostały przystosowane.

W każdym programie C musi się znaleźć co najmniej jedna funkcja o specjalnej nazwie `main()`. Jest to pierwsza funkcja wykonywana podczas uruchamiania programu. `main()` stanowi najwyższy poziom kontroli programu i może wywoływać inne funkcje jako podprogramy.

Struktura prostego, kompletnego programu C jest przedstawiona w przykładzie 1-1. W dalszej części książki omówimy szczegółowo deklarowanie i wywoływanie funkcji, strumienie wejścia/wyjścia i wiele więcej. W tej części książki chcemy jedynie przedstawić ogólną strukturę kodu źródłowego C. W programie przedstawionym w przykładzie 1-1 zdefiniowane są dwie funkcje – `main()` i `circularArea()`. Funkcja `main()` wywołuje

1 Drugie wydanie, odzwierciedlające standard ANSI C, zostało wydane w Polsce pod tytułem *Język ANSI C*, Brian W. Kernighan i Dennis M. Ritchie, Wydawnictwa Naukowo-Techniczne, Warszawa 1994 (przyp. tłum.).

funkcję `circularArea()`, aby uzyskać pole powierzchni koła o podanym promieniu. Następnie wywołuje funkcję `printf()` z biblioteki standardowej, aby wypisać sformatowany wynik w konsoli.

Przykład 1-1 *Prosty program C*

```
// circle.c: Obliczanie i wyświetlanie pola powierzchni kół

#include <stdio.h>                // Dyrektywa preprocesora
double circularArea( double r );  // Deklaracja funkcji (forma prototypowa)

int main()                        // Początek definicji main()

{
  double radius = 1.0, area = 0.0;
  printf( "    Pola powierzchni kół\n\n" );
  printf( "    Promień          Pole\n"
          "-------------------------\n" );
  area = circularArea( radius );
  printf( "%10.1f     %10.2f\n", radius, area );
  radius = 5.0;
  area = circularArea( radius );
  printf( "%10.1f     %10.2f\n", radius, area );
  return 0;
}

// Funkcja circularArea() oblicza pole powierzchni koła
// Parametr: Promień koła
// Zwracana wartość: Pole koła

double circularArea( double r )     // Początek definicji circularArea()
{
  const double pi = 3.1415926536;   // Pi jest stałą
  return  pi * r * r;
}
```

Wynik:

```
          Pola powierzchni kół
          Promień          Pole
       -------------------------
             1.0           3.14
             5.0          78.54
```

Zauważmy, że kompilator wymaga wcześniejszej *deklaracji* każdej wywoływanej funkcji. Prototyp funkcji `circularArea()` w trzecim wierszu przykładu 1-1 zawiera informacje

niezbędne do skompilowania instrukcji, służącej do wywołania tej funkcji. Prototypy funkcji biblioteki standardowej znajdują się w standardowych plikach nagłówkowych. Ponieważ prototyp funkcji `printf()` znajduje się w pliku nagłówkowym *stdio.h*, *dyrektywa preprocesora* `#include <stdio.h>` deklaruje funkcję pośrednio, nakazując preprocesorowi kompilatora dołączenie zawartości tego pliku (patrz również podrozdział „Jak działa kompilator C" na końcu tego rozdziału).

Funkcje zdefiniowane w programie można dowolnie uporządkować. W przykładzie 1-1 moglibyśmy równie dobrze umieścić funkcję `circularArea()` przed funkcją `main()`. W takim przypadku deklaracja prototypu `circularArea()` byłaby zbędna, ponieważ definicja funkcji jest zarazem jej deklaracją.

Nie można zagnieżdżać definicji funkcji wewnątrz innej funkcji. Wewnątrz bloku funkcji można zdefiniować zmienną lokalną, lecz nie można zdefiniować funkcji lokalnej.

Pliki źródłowe

Na kod źródłowy programu C składają się definicje funkcji, deklaracje globalne i dyrektywy preprocesora. W przypadku małych programów kod źródłowy znajduje się w jednym pliku źródłowym. Większe programy C składają się z kilku plików źródłowych. Ponieważ definicje funkcji zazwyczaj zależą od dyrektyw preprocesora i deklaracji globalnych, wewnętrzna struktura plików źródłowych jest zwykle następująca:

1. Dyrektywy preprocesora
2. Deklaracje globalne
3. Definicje funkcji

Język C wspiera programowanie modułowe, pozwalając na uporządkowanie programu w postaci dowolnej liczby plików nagłówkowych oraz na ich osobną edycję i kompilację. Każdy plik źródłowy zawiera logicznie powiązane funkcje, na przykład funkcje dotyczące interfejsu użytkownika programu. Pliki źródłowe programu C zwykle mają rozszerzenie *.c*.

Przykłady 1-2 i 1-3 przedstawiają ten sam program, który znamy z przykładu 1-1, lecz podzielony na dwa pliki źródłowe.

Przykład 1-2 *Pierwszy plik źródłowy, zawierający funkcję main()*

```
// circle.c: Wyświetla pola powierzchni kół.
// Do obliczeń wykorzystuje plik circlararea.c

#include <stdio.h>
double circularArea( double r );
int main()
{
```

```
 /* ... Jak w przykładzie 1-1... */
}
```

Przykład 1-3 *Drugi plik źródłowy, zawierający funkcję circularArea()*

```
// circlararea.c: Oblicza pola powierzchni kół.
// Wywoływany przez funkcję main() w pliku circle.c

double circularArea( double r )
{
 /* ... Jak w przykładzie 1-1... */
}
```

Jeśli program składa się z kilku plików źródłowych, wówczas w wielu z tych plików należy zadeklarować te same funkcje i zmienne globalne, zdefiniować te same makra i stałe. Te deklaracje i definicje tworzą swego rodzaju nagłówek pliku, który jest mniej więcej jednakowy dla całego programu. Dla uproszczenia i zachowania spójności, można zapisać te informacje tylko raz, w osobnym *pliku nagłówkowym*, a następnie odwołać się do pliku nagłówkowego korzystając z dyrektywy `#include` w każdym pliku źródłowym. Pliki nagłówkowe są zwykle identyfikowane przez przyrostek *.h*. Plik nagłówkowy dołączony do źródłowego pliku C może także dołączać inne pliki.

Każdy plik źródłowy C, wraz ze wszystkimi dołączonymi do niego plikami źródłowymi tworzy *jednostkę tłumaczenia*. Kompilator przetwarza po kolei jednostki tłumaczenia, tłumacząc kod źródłowy na tokeny, czyli najmniejsze jednostki leksykalne, takie jak nazwy zmiennych i operatory. Więcej informacji na ten temat można znaleźć w punkcie „Tokeny" pod koniec tego rozdziału.

Między kolejnymi tokenami może się znajdować dowolna liczba białych znaków, dzięki czemu mamy ogromną dowolność w formatowaniu kodu źródłowego. Nie istnieją żadne reguły dotyczące łamania wierszy, czy wcięć. Korzystając ze spacji, tabulacji i pustych wierszy możemy sformatować kod źródłowy w sposób przyjazny dla człowieka. Taka dowolność nie obejmuje dyrektyw preprocesora. Dyrektywa preprocesora musi się zawsze znajdować w osobnym wierszu, zaś przed znakiem kratki (#) rozpoczynającym wiersz mogą się znaleźć tylko znaki spacji lub tabulacji.

Istnieje wiele różnych konwencji i „firmowych stylów" formatowania kodu źródłowego. Większość z nich opiera się na następujących regułach:

- Każda nowa deklaracja i instrukcja powinna się znaleźć w nowym wierszu.
- Wcięcia pozwalają odzwierciedlić zagnieżdżoną strukturę instrukcji blokowych.

Komentarze

Kod źródłowy należy obficie komentować, aby zapewnić dokumentację programów. W języku C do dyspozycji mamy dwa sposoby umieszczania komentarzy. Pierwszy

to *komentarze blokowe*, zaczynające się znakami `/*` i kończące znakami `*/`, zaś drugi to *komentarze wierszowe*, zaczynające się znakami `//` i kończące znakiem nowego wiersza.

Ograniczniki `/*` i `*/` można stosować na początku i końcu komentarzy jednowierszowych oraz do oznaczenia komentarzy rozciągających się na kilka wierszy. Na przykład w następującym prototypie funkcji wielokropek (`...`) oznacza, że funkcja `open()` przyjmuje trzeci opcjonalny parametr. Użycie opcjonalnego parametru objaśnia komentarz:

```
int open( const char *name, int mode, ... /* int permissions */ );
```

Znaków `//` można użyć do utworzenia komentarzy rozciągających się na cały wiersz lub do utworzenia dwukolumnowego kodu źródłowego, z kodem programu w lewej kolumnie i komentarzami w prawej:

```
    const double pi = 3.1415926536;     // Pi jest stałą
```

Komentarze liniowe zostały oficjalnie dodane do języka C przez standard C99, lecz większość kompilatorów wspierało je jeszcze przed pojawieniem się standardu C99. Są one niekiedy nazywane komentarzami w „stylu C++", choć w rzeczywistości wywodzą się od poprzednika C, czyli języka BCPL.

Znaki `/*` i `//` znajdujące się wewnątrz cudzysłowów ograniczających stałą znakową lub literał łańcuchowy nie rozpoczynają komentarza. Na przykład następująca instrukcja nie zawiera komentarzy:

```
printf( "Komentarze w C rozpoczynają się od /* lub //.\n" );
```

Jedynymi poszukiwanymi przez preprocesor znakami wewnątrz komentarza są znaki jego końca, dlatego zagnieżdżanie komentarzy jest niemożliwe. Jednak można skorzystać ze znaków `/*` i `*/`, aby zakomentować fragment programu zawierający komentarze wierszowe:

```
/* Tymczasowe usunięcie dwóch wierszy:
const double pi = 3.1415926536;     // Pi jest stałą
area = pi * r * r                   // Obliczanie pola powierzchni
Tymczasowe usunięcie do tego miejsca */
```

Jeśli chcemy zakomentować fragment programu zawierający komentarze blokowe, możemy skorzystać z warunkowej dyrektywy preprocesora (opisanej w rozdziale 14.):

```
#if 0
 const double pi = 3.1415926536; /* Pi jest stałą     */
 area = pi * r * r               /* Obliczanie pola powierzchni */
#endif
```

Preprocesor zastąpi każdy komentarz spacją. Sekwencja znaków `min/*max*/Value` przyjmie więc postać dwóch tokenów `min Value`.

Zbiory znaków

Język C odróżnia środowisko, w którym kompilator tłumaczy pliki źródłowe programu (*środowisko tłumaczeniowe*) od środowiska, w którym skompilowany program jest uruchamiany, czyli *środowiska wykonania*. Odpowiednio, C definiuje dwa zbiory znaków: *źródłowy zbiór znaków* zawierający znaki, z których można korzystać w kodzie źródłowym C, oraz *zbiór znaków wykonania programu*, który zawiera znaki interpretowalne przez uruchomiony program. W wielu implementacjach C te dwa zbiory znaków są identyczne. Jeśli się różnią, wówczas kompilator przekształca znaki ze stałych znakowych oraz literałów łańcuchowych w kodzie źródłowym w odpowiednie elementy zbioru znaków wykonania programu.

Każdy z tych zbiorów znaków zawiera *podstawowy zbiór znaków* oraz *znaki dodatkowe*. Język C nie określa znaków dodatkowych, które zwykle zależą od lokalnego języka. Znaki dodatkowe wraz z podstawowym zbiorem znaków tworzą powiększony zbiór znaków.

Podstawowe zbiory źródłowe i wykonania programu zawierają następujące rodzaje znaków:

Litery alfabetu łacińskiego:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Cyfry dziesiętne

```
0 1 2 3 4 5 6 7 8 9
```

29 następujących znaków graficznych

```
! " # % & ' ( ) * + , - . / : ; < = > ? [ \ ] ^ _ { | } ~
```

Pięć białych znaków

Znak spacji, tabulacji poziomej, tabulacji pionowej, nowego wiersza i wysuwu strony

Podstawowy zestaw znaków wykonywalnych zawiera także cztery znaki niedrukowalne: znak *null*, który działa jako znak kończący łańcuch znakowy, *alarm*, *cofanie* i *powrót karetki*. Znaki te można odzwierciedlić w literałach znakowych i łańcuchowych wpisując odpowiednie sekwencje specjalne, rozpoczynające się od ukośnika wstecznego: `\0` reprezentuje znak null, `\a` reprezentuje alarm, `\b` oznacza cofanie, zaś `\r` powrót karetki. Więcej informacji na ten temat można znaleźć w rozdziale 3.

Rzeczywista liczbowa wartość znaku – kod znaku – może się różnić w zależności od implementacji C.

Sam język nakłada jedynie następujące warunki:

- Każdy znak w podstawowym zbiorze znaków musi być reprezentowany przez jeden bajt.
- Znak null jest reprezentowany przez bajt, w którym wszystkie bity mają wartość 0.

- Wartości kolejnych cyfr dziesiętnych następujących po 0 są większe o jeden od wartości odpowiadającej poprzedniej cyfrze.

Znaki rozszerzone i wielobajtowe

C został opracowany w środowisku angielskojęzycznym, w którym dominował zbiór znaków opisany 7-bitowym kodem ASCII. Od tego czasu największą popularność zyskało kodowanie znaków za pomocą bajtów 8-bitowych, lecz oprogramowanie przeznaczone do użytku międzynarodowego musi mieć możliwość reprezentowania większej różnorodności znaków, do których zakodowania nie wystarczy jeden bajt. Ponadto w wielu krajach już od dawna używano różnorodnych schematów wielobajtowych, służących do reprezentowania alfabetów niełacińskich oraz chińskich, japońskich i koreańskich systemów pisma. W 1994 roku, wraz z dostosowaniem do dokumentu „Normative Addendum 1", ISO C wprowadziło dwa standardy reprezentacji większych zbiorów znaków:

- *Znaki rozszerzone*, w których każdy znak w zbiorze ma taką samą liczbę bitów.
- *Znaki wielobajtowe*, w których dowolny znak może być reprezentowany przez jeden lub kilka bajtów, zaś wartość znakowa danej sekwencji bajtów może zależeć od kontekstu w łańcuchu lub strumieniu.



Chociaż C udostępnia obecnie abstrakcyjne mechanizmy manipulowania i przekształcania różnych rodzajów schematów kodowania, sam język nie definiuje i nie określa żadnego schematu kodowania, ani żadnego zbioru znaków, oprócz zbiorów znaków podstawowych i rozszerzonych opisanych w poprzednim podrozdziale. Innymi słowy, sposób kodowania znaków rozszerzonych i wspierane schematy kodowania wielobajtowego zostały pozostawione w gestii poszczególnych implementacji.

Znaki rozszerzone

Począwszy od roku 1994, oprócz typu `char`, język C udostępnia także typ `wchar_t`, czyli *rozszerzony typ znakowy*. Typ ten, zdefiniowany w pliku nagłówkowym *stddef.h*, jest na tyle duży, że pozwala na reprezentację dowolnego elementu z powiększonego zbioru znaków danej implementacji.

Chociaż standard C nie wymaga wsparcia dla zbiorów znaków Unicode, to jednak wiele implementacji do przekształceń znaków rozszerzonych wykorzystuje formaty Unicode, takie jak UTF-16 i UTF-32 (patrz http://www.unicode.org/). Standard Unicode jest w dużej mierze identyczny ze standardem ISO/IEC 10646 i jest nadzbiorem wielu istniejących wcześniej zbiorów znaków, w tym 7-bitowego kodu ASCII. Gdy zaimplementowany jest standard Unicode, typ `wchar_t` składa się z 16 lub 32 bitów, zaś wartość przechowywana

w zmiennej typu `wchar_t` reprezentuje jeden znak Unicode. Na przykład następująca definicja inicjalizuje zmienną `wc` i przypisuje do niej grecką literę *α*.

```
wchar_t wc = '\x3b1';
```

Sekwencja specjalna rozpoczynająca się od `\x` oznacza, że zmienna będzie przechowywać kod znakowy w notacji szesnastkowej - w tym przypadku jest to kod oznaczający małą literę alfa.

W celu uzyskania lepszego wsparcia dla Unicode standard C11 wprowadził dodatkowe rozszerzone typy znakowe `char16_t` i `char32_t`, które zostały zdefiniowane w pliku nagłówkowym *uchar.h* jako bezznakowe typy całkowite. Znaki typu `char16_t` są zakodowane w UTF-16 w implementacjach C, które definiują makro `__STDC_UTF_16__`. Podobnie, w implementacjach definiujących makro `__STDC_UTF_32__`, znaki typu `char32_t` są zakodowane w schemacie UTF-32.

Znaki wielobajtowe

W zbiorach znaków wielobajtowych każdy znak jest zakodowany za pomocą sekwencji jednego lub kilku bajtów. Zarówno znaki z źródłowych zbiorów znaków, jak i zbiorów wykonania programu zajmują tylko jeden bajt, zaś żaden znak wielobajtowy, z wyjątkiem znaku null, nie może zawierać bajtu złożonego z samych bitów 0. Znaki wielobajtowe mogą być wykorzystywane w stałych znakowych, literałach łańcuchowych, identyfikatorach, komentarzach i nazwach plików nagłówkowych. Wiele zbiorów znaków wielobajtowych zaprojektowano z myślą o konkretnym języku, czego przykładem może być zbiór znaków Japanese Industrial Standard (JIS). Zbiór znaków wielobajtowych UTF-8, zdefiniowany przez konsorcjum Unicode (Unicode Consortium), pozwala na zakodowanie wszystkich znaków Unicode. W UTF-8 każdy znak jest reprezentowany przez jeden do czterech bajtów.

Najważniejsza różnica między znakami wielobajtowymi a znakami typu rozszerzonego (czyli znakami typu `wchar_t`, `char16_t` oraz `char32_t`) polega na tym, że wszystkie znaki typu rozszerzonego mają ten sam rozmiar, zaś znaki wielobajtowe są reprezentowane przez różną liczbę bajtów. Taka reprezentacja sprawia, że przetwarzanie łańcuchów znaków wielobajtowych jest bardziej skomplikowane niż łańcuchów znaków typu rozszerzonego. Na przykład, nawet jeśli znak *A* można przedstawić w postaci jednego bajtu, znalezienie go w łańcuchu znaków wielobajtowych wymaga nie tylko zwykłego porównywania bajt po bajcie, ponieważ określony bajt w konkretnym położeniu może wchodzić w skład innego znaku. Jednakże znaki wielobajtowe świetnie się nadają do zapisywania tekstu w plikach (patrz rozdział 13.). Ponadto kodowanie znaków wielobajtowych zależy od architektury systemu, natomiast kodowanie znaków typu rozszerzonego zależy od uporządkowania bajtów w danym systemie: otóż w zależności od systemu, bajty znaku typu rozszerzonego mogą się znajdować w porządku *big-endian* lub *little-endian*.

Przekształcenia

C udostępnia funkcje standardowe pozwalające uzyskać wartość `wchar_t` dowolnego znaku wielobajtowego i przekształcenie dowolnego znaku typu rozszerzonego w reprezentację wielobajtową. Na przykład, jeśli kompilator C używa standardów Unicode UTF-16 i UTF-8, wówczas następujące wywołanie funkcji `wctomb()` (z ang. „wide character to multibyte" – znak typu rozszerzonego w wielobajtowy) pozwala uzyskać wielobajtową reprezentację znaku *α*:

```
wchar_t wc = L'\x3B1';     // Mała grecka litera alfa, a
char mbStr[10] = "";
int nBytes = 0;
nBytes = wctomb( mbStr, wc );
if( nBytes < 0)
 puts("W ustawieniach regionalnych nie znaleziono
       odpowiedniego znaku wielobajtowego.");
```

Po udanym wywołaniu funkcji, tablica `mbStr` będzie zawierać znak wielobajtowy, czyli w tym przypadku sekwencję `"\xCE\xB1"`. Wartość zwracana przez funkcję `wctomb()`, przypisana tutaj do zmiennej `nBytes`, oznacza liczbę bajtów potrzebną do zareprezentowania znaku wielobajtowego, czyli w tym przypadku 2.

Biblioteka standardowa udostępnia również funkcje służące do przekształceń typów `char16_t` i `char32_t`, czyli nowych typów znaków typu rozszerzonego wprowadzonych w C11. Przykładem może być funkcja `c16rtomb()`, która zwraca znak wielobajtowy, odpowiadający podanemu znakowi typu rozszerzonego `char16_t` (patrz podrozdział „Znaki wielobajtowe" w rozdziale 17.).

Uniwersalne nazwy znaków

C wspiera także uniwersalne nazwy znaków, które pozwalają na korzystanie z poszerzonego zbioru znaków niezależnie od kodowania stosowanego w implementacji. Dowolny znak typu rozszerzonego można określić za pomocą *uniwersalnej nazwy znaku*, która odpowiada jego wartości w Unicode w następującej postaci:

`\u`*XXXX*

lub:

`\U`*XXXXXXXX*

gdzie *XXXX* lub *XXXXXXXX* jest pozycją kodu Unicode w zapisie szesnastkowym. W zapisie tym należy używać przedrostka w postaci małej litery `u`, po której następują cztery cyfry szesnastkowe, lub w postaci wielkiej litery `U`, po której następuje dokładnie osiem cyfr szesnastkowych. Jeśli pierwsze cztery cyfry są zerem, wówczas uniwersalna nazwa znaku może być zapisana w postaci `\u`*XXXX* lub `\U0000`*XXXX*.

Uniwersalne nazwy znaków można stosować w identyfikatorach, stałych znakowych i literałach tekstowych. Jednakże nie można ich używać do reprezentowania znaków w podstawowym zbiorze znaków. Gdy określamy znak za pomocą jego uniwersalnej nazwy, kompilator przechowuje go w zbiorze znaków wykorzystywanych przez implementację. Na przykład, jeśli w programie wykorzystywany jest zbiór znaków wykonywalnych ISO 8859-7 (grecki 8-bitowy), wówczas następująca definicja zainicjalizuje zmienną `alpha` kodem `\xE1`:

```
char alpha = '\u03B1';
```

Jednakże, jeśli wykorzystywany jest zbiór znaków wykonywalnych UTF-16, wówczas należy zdefiniować zmienną w postaci znaku typu rozszerzonego:

```
wchar_t alpha = '\u03B1';  // lub char16_t alpha = u'\u03B1';
```

W tym przypadku wartość kodu znakowego przypisana do zmiennej `alpha` ma postać szesnastkowego kodu 3B1, takiego samego, jak uniwersalna nazwa znaku.



Nie wszystkie kompilatory wspierają uniwersalne nazwy znaków.

Dwuznaki i trójznaki

W języku C dostępne są alternatywne reprezentacje dla kilku znaków interpunkcyjnych, które nie są dostępne na niektórych klawiaturach. Sześć z nich to *dwuznaki*, czyli tokeny dwuznakowe, które reprezentują znaki przedstawione w tabeli 1-1.

Tabela 1-1 *Dwuznaki*

Dwuznak	Odpowiednik
`<:`	`[`
`:>`	`]`
`<%`	`{`
`%>`	`}`
`%:`	`#`
`%:%:`	`##`

Jeśli sekwencje te wchodzą w skład stałych znakowych lub literałów łańcuchowych, wówczas nie są interpretowane jako dwuznaki. W pozostałych przypadkach zachowują się dokładnie jak reprezentowane przez siebie tokeny jednoznakowe. Następujące przykładowe

fragmenty kodu są dokładnie równoważne i pozwalają uzyskać identyczny wynik. Z wykorzystaniem dwuznaków:

```
int arr<::> = <% 10, 20, 30 %>;
printf( "Drugim elementem w tablicy jest <%d>.\n", arr<:1:> );
```

Bez dwuznaków:

```
int arr[] = { 10, 20, 30 };
printf( "Drugim elementem w tablicy jest <%d>.\n", arr[1] );
```

Wynik:

```
Drugim elementem w tablicy jest <20>.
```

W języku C dostępne są także *trójznaki*, czyli reprezentacje trzyznakowe, z których każda rozpoczyna się od dwóch znaków zapytania. Trzeci znak określa, który znak interpunkcyjny jest reprezentowany przez trójznak, co uwidacznia tabela 1-2.

Tabela 1-2 *Trójznaki*

Trójznak	Odpowiednik
??(	[
??)	]
??<	{
??>	}
??=	#
??/	\
??!	|
??'	^
??-	~

Trójznaki pozwalają na napisanie dowolnego programu w C z wykorzystaniem jedynie znaków dostępnych w zdefiniowanym w 1991 roku standardzie ISO/IEC 646, odpowiadającym 7-bitowemu ASCII. Preprocesor kompilatora zastępuje trójznak odpowiednikiem jednoznakowym w pierwszym etapie kompilacji. Oznacza to, że trójznaki, w odróżnieniu od dwuznaków, są tłumaczone na swoje jednoznakowe odpowiedniki niezależnie od miejsca wystąpienia, nawet jeśli znajdą się w stałych znakowych, literałach tekstowych, komentarzach i dyrektywach preprocesora. W poniższym przykładzie drugi i trzeci znak zapytania zostanie zinterpretowany przez preprocesor jako początek trójznaku:

```
printf("Cancel???(y/n) ");
```

W tym przypadku w wyniku działania preprocesora uzyskamy wiersz kodu w niepożądanej postaci:

```
printf("Cancel?[y/n) ");
```

Jeśli zależy nam na użyciu jednej z tych trójznakowych sekwencji i chcemy uniknąć jej zinterpretowania jako trójznaku, możemy zapisać znaki zapytania za pomocą sekwencji specjalnych:

```
printf("Cancel\?\?\?(y/n) ");
```

Jeśli po dowolnych dwóch znakach zapytania następuje znak inny, niż zaprezentowane w tabeli 1-2, wówczas sekwencja nie jest traktowana jak trójznak i nie jest przetwarzana.



Oprócz dwuznaków i trójznaków dostępny jest jeszcze inny substytut znaków interpunkcyjnych. Dostępny jest za pośrednictwem pliku nagłówkowego *iso646.h*, który zawiera makra definiujące alternatywne reprezentacje operatorów logicznych i bitowych języka C, takich jak and dla `&&` i `xor` dla `^`. Szczegółowe omówienie zawiera rozdział 16.

Identyfikatory

Termin *identyfikator* odnosi się do nazw zmiennych, funkcji, makr, struktur i innych obiektów zdefiniowanych w programie C. Identyfikatory mogą zawierać następujące znaki:

- Litery z podstawowego zestawu znaków, `a-z` i `A-Z`. Wielkość liter w identyfikatorach ma znaczenie.
- Znak podkreślenia `_`.
- Cyfry dziesiętne `0-9`, chociaż nie mogą one stanowić pierwszego znaku identyfikatora.
- Uniwersalne nazwy znaków, które reprezentują litery i cyfry obecne w innych językach.

Dozwolone znaki uniwersalne są przedstawione w dodatku D do standardu C i odpowiadają znakom zdefiniowanym w standardzie ISO/IEC TR 10176, z pominięciem znaków z podstawowego zestawu znaków.

Identyfikatory mogą także zawierać znaki wielobajtowe. Jednakże to dana implementacja C odpowiada za dokładne określenie dozwolonych znaków wielobajtowych oraz odpowiadające im uniwersalne nazwy znaków.

Poniżej przestawione są 44 *zastrzeżone* słowa kluczowe języka C. Każde z nich ma specjalne znaczenie dla kompilatora i nie można ich używać jako identyfikatorów:

auto	extern	short	while
break	float	signed	_Alignas
case	for	sizeof	_Alignof
char	goto	static	_Atomic
const	if	struct	_Bool
continue	inline	switch	_Complex
default	int	typedef	_Generic
do	long	union	_Imaginary
double	register	unsigned	_Noreturn
else	restrict	void	_Static_assert
enum	return	volatile	_Thread_local

Dozwolone są na przykład następujące identyfikatory:

```
x dollar Break error_handler scale64
```

Następujące identyfikatory są niedozwolone:

```
1st_rank switch y/n x-ray
```

Jeśli kompilator wspiera uniwersalne nazwy znaków, wówczas przykładem dozwolonego identyfikatora jest także litera *α*, która może stanowić nazwę zmiennej:

```
double α = 0.5;
```

Edytor kodu źródłowego mógłby zapisać znak α w pliku źródłowym jako uniwersalny znak `\u03B1`.

Definiując identyfikatory w swoich programach należy pamiętać, że wiele identyfikatorów jest używanych przez standardową bibliotekę C. Należą do nich nazwy funkcji z biblioteki standardowej. Nazw tych nie można wykorzystywać do definiowania własnych funkcji czy zmiennych globalnych. Więcej informacji na ten temat można znaleźć w rozdziale 16.

Kompilator udostępnia predefiniowany identyfikator `__func__` (zwróćmy uwagę, że są tu cztery znaki podkreślenia!), który użyty w dowolnej funkcji pozwala uzyskać stałą łańcuchową, zawierającą nazwę tej funkcji. Może się to przydać do logowania lub debugowania wyniku, na przykład:

```
#include <stdio.h>
int test_func( char *s )
{
  if( s == NULL)
  {
    fprintf( stderr,
        "%s: otrzymano argument w postaci wskaźnika pustego\n",
        __func__ );
    return -1;
```

```
    }
    /* ... */
}
```

W tym przykładzie przekazanie do funkcji wskaźnika pustego wygeneruje następujący komunikat błędu:

```
test_func: otrzymano argument w postaci wskaźnika pustego
```

Identyfikatory mogą mieć dowolną długość. Jednakże większość kompilatorów uwzględnia jedynie ograniczoną liczbę znaczących znaków w identyfikatorze. Innymi słowy, kompilator może nie odróżnić dwóch identyfikatorów rozpoczynających się długą identyczną sekwencją znaków. Aby spełnić standardy C, kompilator musi traktować jako znaki znaczące pierwsze 31 znaków w nazwach funkcji i zmiennych globalnych (czyli w identyfikatorach o zewnętrznych powiązaniach) oraz co najmniej pierwsze 63 znaki we wszystkich pozostałych identyfikatorach.

Przestrzenie nazw identyfikatorów

Wszystkie identyfikatory należą do jednej z następujących czterech kategorii, tworzących osobne *przestrzenie nazw*:

- Nazwy etykiet.
- Znaczniki identyfikujące struktury, unie i typy wyliczeniowe.
- Nazwy członków struktury lub unii. Każda struktura lub unia tworzy osobną przestrzeń nazw swoich członków.
- Wszystkie pozostałe identyfikatory, które są zwane *zwykłymi identyfikatorami*.

Identyfikatory należące do różnych przestrzeni nazw mogą być identyczne. Nie ma tu ryzyka wystąpienia konfliktów. Innymi słowy, te same nazwy można wykorzystywać do określania różnych obiektów, o ile są one różnych rodzajów. Na przykład kompilator może odróżniać zmienne od etykiet noszących te same nazwy. Podobnie, tę samą nazwę możemy nadać strukturze, elementowi struktury i zmiennej, jak na poniższym przykładzie:

```
struct pin { char pin[16];  /* ... */ };
_Bool check_pin( struct pin *pin )
{
  int len = strlen( pin->pin );
  /* ... */
}
```

Pierwszy wiersz w przykładzie definiuje strukturę identyfikowaną przez znacznik `pin`. Struktura ta zawiera tablicę znaków o nazwie `pin`. W drugim wierszu znajduje się parametr funkcji `pin`, który jest wskaźnikiem do struktury właśnie zdefiniowanego typu. Wyrażenie `pin->pin`, znajdujące się w czwartym wierszu, oznacza członka struktury

wskazywanego przez parametr funkcji. Przestrzeń nazw identyfikatora jest jednoznacznie wyznaczana na podstawie kontekstu, w którym się on pojawia. Jednakże warto zadbać o to, aby wszystkie identyfikatory w programie były unikalne, aby ułatwić korzystanie z kodu.

Zakres identyfikatora

Zakres identyfikatora odnosi się do tej części jednostki tłumaczenia, w której identyfikator ma znaczenie. Innymi słowy, zakres identyfikatora jest tą częścią programu, która „widzi" ten identyfikator. Rodzaj zakresu zawsze zależy od miejsca, w którym deklarujemy identyfikator (za wyjątkiem etykiet, które zawsze mają zakres funkcji). Istnieją cztery rodzaje zakresów:

Zakres pliku

Jeśli zadeklarujemy identyfikator poza wszystkimi blokami i listą parametrów, wówczas ma on zakres pliku. Z identyfikatora możemy korzystać w dowolnym miejscu po deklaracji, aż do końca jednostki tłumaczenia.

Zakres blokowy

Identyfikatory zadeklarowane wewnątrz bloku, z wyjątkiem etykiet, mają zakres blokowy. Z takiego identyfikatora można korzystać jedynie od miejsca deklaracji do końca najmniejszego bloku zawierającego deklarację. Najmniejszy zawierający blok jest zwykle, lecz nie zawsze, treścią definicji funkcji. Począwszy od wersji C99 deklaracje nie muszą być umieszczane przed wszystkimi instrukcjami bloku funkcji. Nazwy parametrów w nagłówku funkcji również mają zakres blokowy i są dostępne w odpowiednim bloku funkcji.

Zakres prototypu funkcji

Nazwy parametrów w prototypie funkcji mają zakres prototypu funkcji. Ponieważ te nazwy parametrów nie mają znaczenia poza samym prototypem, przydają się jedynie jako komentarze. Można je też pominąć. Więcej informacji na ten temat można znaleźć w rozdziale 7.

Zakres funkcji

Zakresem etykiety jest zawsze blok funkcji, w której znajduje się ta etykieta, nawet jeśli jest ona zdefiniowana wewnątrz zagnieżdżonych bloków. Innymi słowy możemy użyć instrukcji `goto`, aby przejść do etykiety z dowolnego miejsca wewnątrz funkcji zawierającej etykietę (jednak przechodzenie do bloków zagnieżdżonych nie jest dobrym pomysłem – więcej informacji na ten temat można znaleźć w rozdziale 6).

Generalnie, zakres identyfikatora rozpoczyna się *po* jego deklaracji. Wyjątkiem od tej reguły są nazwy lub znaczniki typu struktury, unii i wyliczeń oraz nazwy stałych wyliczeń. Ich zakres rozpoczyna się natychmiast po ich pojawieniu się *w* deklaracji, dlatego można się do nich ponownie odwoływać w samej deklaracji (struktury i unie zostaną omówione

szczegółowo w rozdziale 10; typy wyliczeniowe są omówione w rozdziale 2). Na przykład w następującej deklaracji struktury, ostatni członek struktury, `next` jest wskaźnikiem do właśnie deklarowanej struktury:

```
struct Node { /* ... */  struct Node *next; }; // Definicja typu
                                               // strukturalnego
void printNode( const struct Node *ptrNode);   // Deklaracja funkcji
int printList( const struct Node *first )    // Początek definicji funkcji
{
  struct Node *ptr = first;

  while( ptr != NULL ) {
    printNode( ptr );
    ptr = ptr->next;
  }
}
```

W tym fragmencie kodu identyfikatory `Node`, `next`, `printNode` i `printList` mają zakres pliku. Parametr `ptrNode` ma zakres prototypu funkcji, zaś zmienne `first` i `ptr` mają zakres blokowy.

Identyfikator można wykorzystać ponownie w nowej deklaracji zagnieżdżonej wewnątrz istniejącego zakresu, nawet jeśli nowy identyfikator znajduje się w tej samej przestrzeni nazw. W takiej sytuacji nowa deklaracja musi mieć zakres blokowy lub prototypu funkcji, zaś blok lub prototyp funkcji musi być prawdziwym podzbiorem zewnętrznego zakresu. W takich przypadkach nowa deklaracja tego samego identyfikatora *ukrywa* zewnętrzną deklarację, dzięki czemu zmienna lub funkcja zadeklarowana w zewnętrznym bloku nie jest *widoczna* w wewnętrznym zakresie. Dozwolone są na przykład poniższe deklaracje:

```
double x;               // Deklaracja zmiennej x o zakresie plikowym
long calc( double x );  // Deklaracja nowej zmiennej x o zakresie prototypu
                        // funkcji

int main()
{
  long x = calc( 2.5 ); // Deklaracja zmiennej x typu long o zakresie
                        // blokowym
  if( x < 0 )           // Tutaj x dotyczy zmiennej typu long
  { float x = 0.0F;     // Deklaracja nowej zmiennej x typu float
                        // o zakresie blokowym
    /*...*/
  }
  x *= 2;               // Tutaj x ponownie dotyczy zmiennej typu long
  /*...*/
}
```

W tym przykładzie zmienna x typu long zadeklarowana w funkcji main() ukrywa globalną zmienną x typu double. W ten sposób nie istnieje bezpośredni dostęp do zmiennej x typu double z poziomu funkcji main(). Ponadto, w bloku warunkowym, który jest zależny od instrukcji if, x odnosi się do nowo zadeklarowanej zmiennej typu float, która z kolei ukrywa zmienną x typu long.

Jak działa kompilator C

Po napisaniu kodu źródłowego za pomocą edytora tekstu można uruchomić kompilator C, który przetłumaczy go na kod maszynowy. Kompilator operuje na *jednostce tłumaczenia*, zawierającej plik źródłowy i wszystkie pliki nagłówkowe dołączone za pomocą dyrektywy #include. Jeśli kompilator nie napotka żadnych błędów w jednostce tłumaczenia, wygeneruje *plik obiektowy* zawierający odpowiedni kod maszynowy. Pliki obiektowe można zwykle rozpoznać na podstawie rozszerzenia *.o* lub *.obj*. Ponadto kompilator może także wygenerować kod asemblera (patrz rozdział 19).

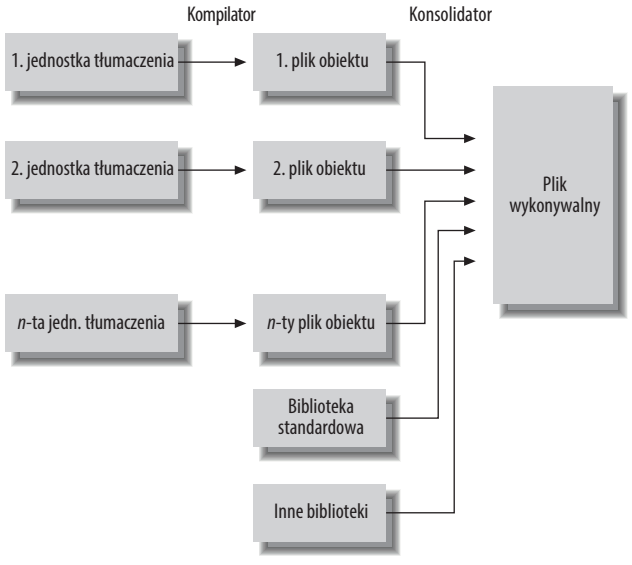
Pliki źródłowe są zwane także *modułami*. Biblioteka, na przykład standardowa biblioteka C, zawiera skompilowane, łatwo dostępne moduły zawierające funkcje standardowe.

Kompilator tłumaczy każdą jednostkę tłumaczenia programu C – czyli każdy plik źródłowy wraz ze wszystkimi dołączonymi do niego plikami nagłówkowymi – w osobny plik obiektowy. Następnie kompilator uruchamia konsolidator (*linker*), który łączy pliki obiektowe i wszystkie wykorzystywane funkcje bibliotek w *plik wykonywalny*. Rysunek 1-1 przedstawia proces kompilacji i konsolidacji programu składającego się z kilku plików źródłowych i bibliotek. Plik wykonywalny zawiera także wszystkie informacje dla docelowego systemu operacyjnego, potrzebne do załadowania i uruchomienia programu.

Etapy tłumaczenia kompilatora C

Proces kompilacji odbywa się w ośmiu logicznych krokach. Konkretny kompilator może połączyć ze sobą niektóre z tych kroków, o ile nie wpłynie to na wynik końcowy. Kroki te są następujące:

1. Z pliku źródłowego odczytywane są znaki i jeśli to konieczne, są one przekształcane w znaki ze źródłowego zbioru znaków. Jeśli znaki oznaczające koniec wiersza w pliku źródłowym są inne niż znaki nowego wiersza, wówczas są nim zastępowane. Podobnie, wszystkie sekwencje trójznaków są zastępowane swoimi jednoznakowymi odpowiednikami (jednakże dwuznaki są pozostawiane bez zmian; nie są przekształcane w swoje jednoznakowe odpowiedniki).
2. W każdym miejscu, w którym po ukośniku wstecznym znajduje się znak nowego wiersza, preprocesor usuwa obydwa znaki. Ponieważ koniec wiersza kończy dyrektywę preprocesora, ten krok pozwala na umieszczenie ukośnika wstecznego na końcu wiersza w celu kontynuacji dyrektywy, na przykład definicji makra, w kolejnym wierszu.



Rysunek 1-1 *Od pliku źródłowego do pliku wykonywalnego*



Każdy plik źródłowy, który nie jest całkowicie pusty, musi się kończyć znakiem nowego wiersza.

3. Każdy plik źródłowy jest dzielony na tokeny preprocesora (patrz następny punkt „Tokeny") oraz sekwencje białych znaków. Każdy komentarz jest traktowany jak jeden znak spacji.
4. Wykonywane są dyrektywy preprocesora i rozwijane są wywołania makr.



Kroki od 1. do 4. są przeprowadzane na każdym pliku dołączonym za pomocą dyrektywy `#include`. Gdy tylko kompilator przetworzy dyrektywy preprocesora, usuwa je z kopii roboczej pliku źródłowego.

5. Znaki i sekwencje specjalne obecne w stałych znakowych i literałach tekstowych są przekształcane w odpowiadające im znaki ze zbioru znaków rozszerzonych.
6. Sąsiadujące literały tekstowe są łączone w jeden łańcuch.
7. Proces właściwej kompilacji. Kompilator analizuje sekwencję tokenów i generuje odpowiedni kod maszynowy.

8. Konsolidator przetwarza odniesienia do zewnętrznych obiektów i funkcji i generuje plik wykonywalny. Jeśli moduł odnosi się do zewnętrznych obiektów lub funkcji, które nie są zdefiniowane w żadnej jednostce tłumaczenia, konsolidator pobiera je z biblioteki standardowej lub innej określonej biblioteki. Zewnętrzne obiekty i funkcje muszą być zdefiniowane w programie tylko jeden raz.

W większości kompilatorów preprocesor jest osobnym programem albo kompilator udostępnia opcje pozwalające przeprowadzić samo wstępne przetwarzanie (kroki od 1. do 4. z powyższej listy). W ten sposób możemy zweryfikować, czy dyrektywy preprocesora dają pożądany wynik. Więcej praktycznych informacji dotyczących procesu kompilacji można znaleźć w rozdziale 19.

Tokeny

Tokenem może być słowo kluczowe, identyfikator, stała, literał łańcuchowy lub symbol. Symbole w C składają się z jednego lub kilku znaków interpunkcyjnych i działają jak operatory, dwuznaki lub mają znaczenie składniowe. Na przykład średnik kończy prostą instrukcję, a nawiasy { } wydzielają instrukcję blokową. Poniższa przykładowa instrukcja C składa się z pięciu tokenów:

```
printf("Witaj, świecie.\n");
```

Poszczególne tokeny to:

```
printf
(
"Witaj, świecie.\n"
)
;
```

Tokeny interpretowane przez preprocesor są przetwarzane w trzecim etapie tłumaczenia. Różnią się tylko w niewielkim stopniu od tokenów interpretowanych przez kompilator w siódmym etapie tłumaczenia:

- Wewnątrz dyrektywy `#include` preprocesor rozpoznaje dodatkowe tokeny <*nazwapliku*> i "*nazwapliku*".
- Podczas etapu wstępnego przetwarzania stałe znakowe i literały tekstowe nie zostały jeszcze przekształcone ze znaków źródłowych w znaki ze zbioru poszerzonego.
- W przeciwieństwie do samego kompilatora, preprocesor nie odróżnia stałych całkowitych od stałych zmiennopozycyjnych.

Podczas przetwarzania plików źródłowych na tokeny, kompilator (lub preprocesor) zawsze stosuje następującą zasadę: każdy kolejny nie-biały znak musi zostać dołączony do odczytywanego tokena, chyba że prowadziłoby to do przekształcenia prawidłowego

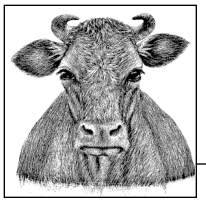
tokena w nieprawidłowy. Ta zasada rozwiązuje wszystkie wątpliwości w następującym przykładzie:

```
a+++b
```

Ponieważ pierwszy znak + nie może wchodzić w skład identyfikatora, czy słowa kluczowego rozpoczynającego się od litery a, rozpoczyna się tworzenie nowego tokena. Drugi znak + po dołączeniu do pierwszego tworzy poprawny token – operator inkrementacji – lecz trzeci znak + już nie. Dlatego wyrażenie to musi zostać odczytane następująco:

```
a ++ + b
```

Więcej informacji na temat kompilowania programów C można znaleźć w rozdziale 19.



2
Typy

Programy muszą mieć możliwość przechowywania i przetwarzania różnych rodzajów danych, na przykład liczb całkowitych czy zmiennopozycyjnych, w odmienny sposób. Dlatego też kompilator musi wiedzieć, jaki rodzaj danych reprezentuje dana wartość.

W języku C termin *obiekt* dotyczy obszaru pamięci, którego zawartość może reprezentować wartości. Obiekty, które mają nazwę, są także określane jako zmienne. Typ obiektu wpływa na ilość miejsca zajmowanego w pamięci i sposób kodowania możliwych wartości. Na przykład ten sam wzór bitów może reprezentować całkowicie różne liczby całkowite w zależności od tego, czy obiekt danych jest interpretowany jako liczba *ze znakiem* – dodatnia lub ujemna – czy też *bez znaku*, gdy nie może reprezentować wartości ujemnych.

Typologia

Typy dostępne w C można sklasyfikować następująco:

- Typy podstawowe
  - Standardowe i rozszerzone typy całkowite
  - Zmiennopozycyjne typy rzeczywiste i zespolone
- Typy wyliczeniowe
- Typ `void`
- Typy pochodne
  - Wskaźniki
  - Tablice
  - Struktury

- Unie
- Typy funkcyjne

Typy podstawowe i typy wyliczeniowe należą do *typów arytmetycznych*. Typy arytmetyczne i wskaźniki są nazywane *typami skalarnymi*. Natomiast tablice i struktury są zwane *typami agregacyjnymi* (unie nie są uważane za typy agregacyjne, ponieważ tylko jeden członek unii może w określonym momencie przechowywać wartość). *Typ funkcyjny* opisuje interfejs funkcji; oznacza to, że określa typ wartości zwracanej przez funkcję i może również określać typy wszystkich parametrów przekazywanych do funkcji podczas jej wywoływania.

Wszystkie pozostałe typy opisują obiekty. Ten opis może, lecz nie musi dotyczyć także rozmiaru obiektu w pamięci. W tym przypadku typ jest nazywany *typem obiektowym*; w przeciwnym razie jest to *typ niekompletny*. Przykładem niekompletnego typu może być zadeklarowana zewnętrznie zmienna tablicowa:

```
extern float fArr[ ];     // Deklaracja zewnętrzna
```

Ten wiersz deklaruje zmienną tablicową `fArr`, której elementy są typu `float`. Jednak, ponieważ nie podano tutaj rozmiaru tablicy, `fArr` jest typu niekompletnego. Jeśli definicja globalnej tablicy `fArr` w innej lokalizacji programu – na przykład w innym pliku źródłowym – nadaje jej określony rozmiar, to ta deklaracja pozwoli na użycie tablicy w bieżącym zakresie (więcej informacji na temat deklaracji zewnętrznych można znaleźć w rozdziale 11).



Ten rozdział opisuje typy podstawowe, wyliczeniowe i typ `void`. Typy pochodne zostaną omówione w rozdziałach od 7 do 10.

Niektóre typy są określane przez sekwencję kilku słów kluczowych, na przykład `unsigned short`. W tym przypadku słowa kluczowe można zapisać w dowolnym porządku. Jednakże zwykle stosuje się określony porządek słów kluczowych, którego przestrzegamy w tej książce.

Typy całkowite

Istnieje pięć typów całkowitych ze znakiem. Większość z nich można określić za pomocą kilku synonimów, które zostały przedstawione w tabeli 2-1.

Tabela 2-1 *Standardowe typy całkowite ze znakiem*

Typ	Synonimy
`signed char`	
`int`	`signed`, `signed int`
`short`	`short int`, `signed short`, `signed short int`
`long`	`long int`, `signed long`, `signed long int`
`long long` *(C99)*	`long long int`, `signed long long`, `signed long long int`

Każdy z pięciu typów całkowitych ze znakiem przedstawionych w tabeli 2-1 posiada odpowiednik w postaci typu bez znaku, który zajmuje tę samą ilość pamięci i ma to samo wyrównanie. Innymi słowy, jeśli kompilator przypisuje obiektom typu `signed int` adresy na bajcie parzystym, wówczas obiekty typu `unsigned int` są także wyrównywane do adresów parzystych. Typy bez znaków są przedstawione w tabeli 2-2.

Tabela 2-2 *Standardowe typy całkowite bez znaku*

Typ	Synonimy
`_Bool`	`bool` (zdefiniowany w *stdbool.h*)
`unsigned char`	
`unsigned int`	`unsigned`
`unsigned short`	`unsigned short int`
`unsigned long`	`unsigned long int`
`unsigned long long`	`unsigned long long int`

Standard C99 wprowadził typ całkowity bez znaku `_Bool`, który służy do reprezentowania wartości logicznych. Logiczna wartość *true* jest kodowana jako 1, zaś *false* jako 0. Jeśli dołączymy do programu plik nagłówkowy *stdbool.h*, możemy używać także identyfikatorów `bool`, `true` i `false`, które są znajome programistom C++. Makro `bool` jest synonimem typu `_Bool`, zaś `true` i `false` są stałymi symbolicznymi równymi 1 i 0.

Do standardowych typów całkowitych należy również typ `char`. Jednakże w zależności od kompilatora jednowyrazowa nazwa typu `char` jest synonimem typu `signed char` lub `unsigned char`. Ponieważ wybór pozostawiono implementacji, typy `char`, `signed char` i `unsigned char` są formalnie różnymi typami.



Jeśli program wykorzystuje typ `char`, mogący przechowywać wartości ujemne lub większe niż 127, należy raczej korzystać z typu `signed char` lub `unsigned char`.

Na zmiennych znakowych można dokonywać działań arytmetycznych. To do programisty należy decyzja, czy program będzie interpretował zmienną typu `char` jako kod znaku, czy inaczej. Na przykład następujący krótki program w jednym miejscu traktuje wartość zmiennej `ch` typu `char` jak liczbę całkowitą, a w innym jak znak:

```
char ch = 'A';                // Zmienna typu char
printf("Znak %c ma kod znakowy %d.\n", ch, ch);
for ( ; ch <= 'Z'; ++ch )
  printf("%2c", ch);
```

W instrukcji `printf()`, wartość `ch` jest najpierw traktowana jak znak, który ma zostać wyświetlony, a następnie jak liczbowa wartość kodu znaku. Również pętla `for` traktuje `ch` w wyrażeniu `++ch` jako liczbę całkowitą, zaś w wywołaniu funkcji `printf()` jako znak. W systemach wykorzystujących 7-bitowy kod ASCII, lub jego rozszerzenie, wynik działania powyższego kodu będzie następujący:

```
Znak A ma kod znakowy 65.
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Wartość typu `char` zawsze zajmuje jeden bajt – innymi słowy, funkcja `sizeof(char)` zawsze zwróci wartość 1 – zaś bajt ten zawiera co najmniej osiem bitów. Każdy znak z podstawowego zbioru znaków może być reprezentowany za pomocą obiektu `char` jako liczba dodatnia.

Standard C definiuje tylko *minimalne* rozmiary przechowywania pozostałych typów standardowych: typ `short` zajmuje co najmniej dwa bajty, `long` ma co najmniej cztery bajty, zaś `long long` co najmniej osiem bajtów. Ponadto, mimo że typy całkowite mogą mieć rozmiary większe od minimalnych, to ich implementacja musi spełniać następujący warunek:

```
sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)
```

Typ `int` jest typem całkowitym, który jest najlepiej przystosowany do architektury docelowego systemu - takim, którego rozmiar i format bitowy odpowiadają rejestrowi CPU.

Wewnętrznie typ całkowity jest reprezentowany za pomocą kodu binarnego. Typy ze znakiem mogą być reprezentowane binarnie w postaci *kodu znak-moduł*, *kodu uzupełnień do jedności* lub *kodu uzupełnień do dwóch*. Najpopularniejszą reprezentacją jest kod uzupełnień do dwóch. Wartości nieujemne typu ze znakiem mieszczą się w zakresie wartości odpowiadających wartościom typu bez znaku, zaś binarna reprezentacja wartości nieujemnych jest taka sama w przypadku typów ze znakiem i bez znaku. Tabela 2-3 przedstawia różne interpretacje wzorców bitowych dla typów całkowitych ze znakiem i bez znaku.

Tabela 2-3 *Binarne reprezentacje 16-bitowych liczb całkowitych ze znakiem i bez znaku*

Kod binarny	Wartość dziesiętna typu unsigned int	Wartość dziesiętna typu signed int, kod uzupełnień do jedności	Wartość dziesiętna typu signed int, kod uzupełnień do dwóch
00000000 00000000	0	0	0
00000000 00000001	1	1	1
00000000 00000010	2	2	2
…			
01111111 11111111	32767	32767	32767
10000000 00000000	32768	-32767	-32768
10000000 00000001	32769	-32766	-32767
…			
11111111 11111110	65534	-1	-2
11111111 11111111	65535	-0	-1

Tabela 2-4 przedstawia rozmiary i zakresy wartości standardowych typów całkowitych.

Tabela 2-4 *Popularne rozmiary zajmowane przez standardowe typy całkowite i ich zakresy wartości*

Typ	Rozmiar	Wartość minimalna	Wartość maksymalna
`char`	(taki sam jak typu `signed char` lub `unsigned char`)		
`unsigned char`	jeden bajt	0	255
`signed char`	jeden bajt	-128	127
`int`	dwa lub cztery bajty	-32768 lub -2147483648	32767 lub 2147483647
`unsigned int`	dwa lub cztery bajty	0	65535 lub 4294967295
`short`	dwa bajty	-32768	32767

ciąg dalszy na stronie następnej

Tabela 2-4 *Popularne rozmiary zajmowane przez standardowe typy całkowite i ich zakresy wartości*

Typ	Rozmiar	Wartość minimalna	Wartość maksymalna
`unsigned short`	dwa bajty	0	65535
`long`	cztery bajty	-2147483648	2147483647
`unsigned long`	cztery bajty	0	4294967295
`long long` (C99)	osiem bajtów	-9223372036854775808	9223372036854775807
`unsigned long long` (C99)	osiem bajtów	0	18446744073709551615

W poniższym przykładzie każda ze zmiennych typu `int`, `iIndex` i `iLimit`, zajmuje na komputerze 32-bitowym cztery bajty:

```
int iIndex,          // Definicja dwóch zmiennych typu int
  iLimit = 1000;     // inicjalizacja drugiej zmiennej.
```

Aby uzyskać dokładny rozmiar typu lub zmiennej, należy użyć operatora `sizeof`. Wyrażenie

```
sizeof(typ)
```

zwróci rozmiar typu o podanej nazwie, zaś

```
sizeof wyrażenie
```

zwróci rozmiar typu podanego wyrażenia w postaci liczby bajtów wyrażonej wartością typu `size_t`. Typ `size_t` jest zdefiniowany w plikach *stddef.h*, *stdio.h* i innych plikach nagłówkowych jako typ całkowity bez znaku, na przykład jako `unsigned long`. Jeśli operand jest wyrażeniem, rozmiar dotyczy typu zwracanego przez wyrażenie. W poprzednim przykładzie wartość `sizeof(int)` powinna być identyczna jak wartość `sizeof(iIndex)`, a mianowicie 4. Ponieważ `iIndex` jest wyrażeniem, a nie typem, możemy opuścić nawiasy wokół `iIndex`.

Zakresy wartości typów całkowitych swojego kompilatora można znaleźć w pliku nagłówkowym *limits.h*, w którym zdefiniowane są makra, takie jak `INT_MIN`, `INT_MAX`, `UINT_MAX` i inne (patrz rozdział 16). Program w przykładzie 2-1 wykorzystuje te makra do wyświetlenia wartości minimalnych i maksymalnych typów `char` i `int`.

Przykład 2-1 *Zakresy wartości typów char i int*

```
// limits.c: Wyświetla zakresy wartości typów char i int.
// ------------------------------------------------
#include <stdio.h>
#include <limits.h>     // Zawiera makra CHAR_MIN, INT_MIN, itd.
int main()
```

```
{
  printf("Rozmiary i zakresy wartości typów char i int\n\n");
  printf("Typ char jest %s.\n\n", CHAR_MIN < 0 ? "ze znakiem" :
         "bez znaku");
  printf(" Typ   Rozmiar (w bajtach)   Minimum       Maksimum\n"
         "-------------------------------------------------\n");
  printf(" char %8zu %20d %15d\n", sizeof(char), CHAR_MIN, CHAR_MAX );
  printf(" int  %8zu %20d %15d\n", sizeof(int), INT_MIN, INT_MAX );
  return 0;
}
```


W operacjach arytmetycznych na liczbach całkowitych mogą wystąpić przepełnienia. Przepełnienie następuje, gdy wynik operacji wykracza poza zakres wartości danego typu. W operacjach arytmetycznych z udziałem typów całkowitych bez znaku przepełnienia są ignorowane. W terminologii matematycznej oznacza to, że wynik operacji na liczbach całkowitych bez znaku jest równy reszcie z dzielenia przez wyrażenie *UTYPE_MAX* + 1, gdzie *UTYPE_MAX* jest maksymalną wartością, jaką może reprezentować zmienna tego typu bez znaku. Na przykład następujące dodawanie doprowadzi do przepełnienia:

```
unsigned int ui = UINT_MAX;
ui += 2;                        // Wynik: 1
```

C definiuje takie zachowanie jedynie dla typów całkowitych bez znaku. W przypadku wszystkich pozostałych typów wynik przepełnienia pozostaje nieokreślony. Przepełnienie może być na przykład zignorowane lub może wywołać sygnał przerywający działanie programu, o ile nie zostanie przechwycone.

Typy całkowite zdefiniowane w nagłówkach standardowych

Nagłówki biblioteki standardowej definiują liczne typy całkowite o specjalnym przeznaczeniu, takie jak `wchar_t`, który reprezentuje znaki typu rozszerzonego. Te typy są zadeklarowane za pomocą instrukcji `typedef`, czyli są synonimami standardowych typów całkowitych (patrz podrozdział „Deklaracje typedef” w rozdziale 11.).

Typy `ptrdiff_t`, `size_t` i `wchar_t` są zdefiniowane w pliku nagłówkowym *stddef.h* (i w innych); typy `char16_t` i `char32_t` są zdefiniowane w pliku nagłówkowym *uchar.h*. W pliku nagłówkowym *stdint.h* zdefiniowano typy całkowite specjalnego przeznaczenia o określonej liczbie bitów, w wersjach ze znakiem i bez znaku. Zostały one opisane w kolejnym punkcie.

Ponadto, w nagłówku *stdint.h* zdefiniowane są także makra udostępniające wartości maksymalne i minimalne wszystkich typów całkowitych zdefiniowanych w bibliotece standardowej. Na przykład `SIZE_MAX` jest równy największej wartości, jaką można przechowywać w zmiennej typu `size_t`. Szczegółowe omówienie wspomnianych tutaj typów i odpowiadających im makr można znaleźć w rozdziale 16.

Typy całkowite o określonej liczbie bajtów (C99)

Rozmiar typu całkowitego jest zdefiniowany w postaci liczby bitów wykorzystywanych do przedstawienia wartości, wraz z bitem znaku. Zwykle jest to 8, 16, 32 lub 64 bity. Na przykład typ `int` ma co najmniej 16 bitów.

W standardzie C99 plik nagłówkowy *stdint.h* definiuje typy całkowite, spełniające warunek określonej liczby bitów. Są one wymienione w tabeli 2-5. Typy, których nazwy rozpoczynają się od litery `u`, są bez znaku. Implementacje standardu C99 nie muszą zapewniać obecności typów oznaczonych w tabeli jako opcjonalne.

Tabela 2-5 *Typy całkowite o zdefiniowanym rozmiarze*

Typ	Znaczenie	Implementacja
`int`*N*`_t` `uint`*N*`_t`	Typ całkowity o wielkości równej dokładnie *N* bitów	Opcjonalna
`int_least`*N*`_t` `uint_least`*N*`_t`	Typ całkowity o wielkości równej co najmniej *N* bitów	Wymagana dla *N* = 8, 16, 32, 64
`int_fast`*N*`_t` `uint_fast`*N*`_t`	Najszybszy do przetwarzania typ o wielkości co najmniej *N* bitów	Wymagana dla *N* = 8, 16, 32, 64
`intmax_t` `uintmax_t`	Największy zaimplementowany typ całkowity	Wymagana
`intptr_t` `uintptr_t`	Typ całkowity o wielkości wystarczającej do przechowywania wartości wskaźnika	Opcjonalna

Na przykład `int_least64_t` i `uint_least64_t` są typami całkowitymi o wielkości co najmniej 64 bitów. Jeśli zdefiniowany jest typ opcjonalny ze znakiem (bez przedrostka `u`), wówczas wymagana jest implementacja odpowiednika bez znaku (z przedrostkiem `u`) i odwrotnie. Następujący przykład definiuje i inicjalizuje tablicę, której elementy są typu `int_fast32_t`:

```
#define ARR_SIZE 100
int_fast32_t arr[ARR_SIZE];     // Definicja tablicy arr
                                // o elementach typu int_fast32_t

for ( int i = 0; i < ARR_SIZE; ++i )
  arr[i] = (int_fast32_t)i;     // Inicjalizacja każdego elementu
```

Typy przedstawione w tabeli 2-5 są zwykle zdefiniowane w postaci synonimów istniejących typów standardowych. Na przykład plik *stdint.h* jednego z kompilatorów C zawiera wiersz:

```
typedef signed char    int_fast8_t;
```

Ta deklaracja definiuje nowy typ `int_fast8_t` (najszybszy 8-bitowy typ całkowity ze znakiem), który jest odpowiednikiem typu `signed char`.

Ponadto implementacja może także definiować rozszerzone typy całkowite, takie jak `int24_t` lub `uint_least128_t`.

Typy `intN_t` ze znakiem mają specyficzną cechę. Otóż muszą mieć reprezentację binarną w postaci kodu uzupełnień do dwóch. W wyniku tego, ich minimalna wartość wynosi $-2N-^{1}$, zaś ich wartość maksymalna wynosi $2N-^{1}-1$.

Równie łatwo jest uzyskać zakresy wartości typów zdefiniowanych w pliku *stdint.h*. W tym samym pliku nagłówkowym zdefiniowane są makra definiujące największe i najmniejsze wartości. Nazwy makr mają postać nazw typów zapisanych wielkimi literami, zaś przyrostek `_t` (od słowa *typ*) jest zastąpiony przyrostkiem `_MAX` lub `_MIN` (patrz rozdział 16). Na przykład następująca definicja inicjalizuje zmienną `i64`, przypisując do niej najmniejszą możliwą wartość:

```
int_least64_t i64 = INT_LEAST64_MIN;
```

Plik nagłówkowy *inttypes.h* dołącza plik nagłówkowy *stdint.h* i udostępnia także specyfikatory rozszerzonych typów całkowitych, z których można korzystać w wywołaniach funkcji `printf()` i `scanf()` (patrz rozdział 16).

Typy zmiennopozycyjne

C zawiera także specjalne typy liczbowe, które mogą reprezentować liczby niecałkowite z kropką dziesiętną w dowolnym miejscu. Standardowe typy zmiennopozycyjne służące do obliczeń na liczbach rzeczywistych są następujące:

`float`

Dla zmiennych o pojedynczej precyzji

`double`

Dla zmiennych o podwójnej precyzji

`long double`

Dla zmiennych o rozszerzonej precyzji

Wartość zmiennopozycyjną można przechowywać jedynie z zachowaniem ograniczonej precyzji, która jest określana przez format binarny służący do jej reprezentowania oraz od ilości pamięci wykorzystywanej do jej przechowywania. Precyzja jest wyrażana jako liczba cyfr znaczących. Na przykład „precyzja sześciu cyfr dziesiętnych" lub „precyzja sześciocyfrowa" oznacza, że reprezentacja binarna typu jest wystarczająca do przechowywania liczby rzeczywistej składającej się z sześciu cyfr dziesiętnych. Innymi słowy, ponowna konwersja do sześciocyfrowej liczby dziesiętnej zwróci sześć oryginalnych cyfr. Położenie kropki dziesiętnej nie ma znaczenia, zaś początkowe i końcowe zera nie

są wliczane do wspomnianych sześciu cyfr. Zarówno 123456000, jak i 0,00123456 można przechowywać w typie o sześciocyfrowej precyzji.

W języku C operacje arytmetyczne na liczbach zmiennopozycyjnych są zazwyczaj przeprowadzane wewnętrznie z podwójną lub większą precyzją. Zmiennoprzecinkowa precyzja używana wewnętrznie przez daną implementację jest sygnalizowana wartością makra FLT_EVAL_METHOD, zdefiniowaną w nagłówku *float.h*. Na przykład, jeśli makro FLT_EVAL_METHOD ma wartość 1, następujące działanie jest obliczane z wykorzystaniem typu `double`:

```
float height = 1.2345, width = 2.3456;  // Zmienne typu float mają
                                        // pojedynczą precyzję
double area = height * width;           // Właściwe obliczenia są
                                        // przeprowadzane z podwójną
                                        // (lub większą) precyzją.
```

Jeśli przypiszemy wynik do zmiennej typu `float`, wartość zostanie odpowiednio zaokrąglona. Więcej informacji na temat obliczeń z wykorzystaniem typów zmiennopozycyjnych można znaleźć w opisie pliku *math.h* w rozdziale 16.

C definiuje tylko minimalne wymagania dotyczące rozmiaru i binarnego formatu typów zmiennopozycyjnych. Jednakże powszechnie używa się formatu standardu IEC 60559, zdefiniowanego przez komisję International Electrotechnical Commission (IEC) w 1989 roku. Dotyczy on działań arytmetycznych na binarnych liczbach zmiennopozycyjnych. Standard ten opiera się na standardzie IEEE 754, opracowanym przez instytut Electrical and Electronics Engineers w 1985 roku. Jeśli zdefiniowane jest makro `__STDC_IEC_559__`, oznacza to, że kompilator wspiera standard zmiennoprzecinkowy IEC. Tabela 2-6 przedstawia zakresy wartości zapisane w notacji dziesiętnej i precyzję rzeczywistych typów zmiennoprzecinkowych, zgodnie ze standardem IEC 60559.

Tabela 2-6 *Rzeczywiste typy zmiennopozycyjne*

Typ	Rozmiar	Zakres wartości	Najmniejsza wartość dodatnia	Precyzja
`float`	4 bajty	±3.4E+38	1.2E-38	6 cyfr
`double`	8 bajtów	±1.7E+308	2.3E-308	15 cyfr
`long double`	10 bajtów	±1.1E+4932	3.4E-4932	19 cyfr

Plik nagłówkowy *float.h* definiuje makra, pozwalające nam używać tych wartości i zawiera inne szczegóły dotyczące binarnej reprezentacji liczb rzeczywistych w programach. Makra `FLT_MIN`, `FLT_MAX` i `FLT_DIG` określają zakres wartości i precyzję typu `float`. Odpowiednie makra dla typów `double` i `long double` rozpoczynają się od przedrostków `DBL_` i `LDBL_`. Szczegóły dotyczące tych makr i binarnej reprezentacji liczb zmiennopozycyjnych można znaleźć w opisie pliku *float.h* w rozdziale 16.

Program w przykładzie 2-2 rozpoczyna się od wypisania typowych wartości dla typu `float`, a następnie prezentuje błąd zaokrąglania, wynikający z przechowywania liczby zmiennopozycyjnej w zmiennej typu `float`.

Przykład 2-2 *Kod ilustrujący precyzję typu float*

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

int main()
{
  puts("\nCechy typu float\n");

  printf("Rozmiar: %d bajtów\n"
         "Najmniejsza wartość dodatnia: %E\n"
         "Największa wartość dodatnia: %E\n"
         "Precyzja: %d cyfr dziesiętnych \n",
         sizeof(float), FLT_MIN, FLT_MAX, FLT_DIG);

  puts("\nPrzykład precyzji zmiennopozycyjnej:\n");
  double d_var = 12345.6;       // Zmienna typu double.
  float f_var = (float)d_var;   // Inicjalizacja zmiennej typu float
                                // wartością zmiennej d_var.
  printf("Liczba zmiennopozycyjna       "
         "%18.10f\n", d_var);
  printf("została przypisana do zmiennej\n"
         "typu float jako wartość       "
         "%18.10f\n", f_var);
  printf("Błąd zaokrąglenia wynosi      "
         "%18.10f\n", d_var - f_var);

  return 0;
}
```

Ostatnia część tego programu zwykle wygeneruje następujący wynik:

```
Liczba zmiennopozycyjna       12345.6000000000
została przypisana do zmiennej
typu float jako wartość   12345.5996093750
Błąd zaokrąglenia wynosi      0.0003906250
```

W tym przykładzie najbliższą reprezentowalną wartością dla liczby 12345,6 jest 12345,5996093750. Nie wygląda to jak zaokrąglona wartość w notacji dziesiętnej, lecz w przypadku wewnętrznej interpretacji binarnej typu zmiennopozycyjnego *jest* to wartość możliwa do zapisania, zaś 12345,60 nią nie jest.

Zespolone typy zmiennopozycyjne

Standard C99 wspiera obliczenia matematyczne na liczbach zespolonych. Standard z 1999 roku wprowadził zespolone typy zmiennopozycyjne i rozszerzył bibliotekę matematyczną o zespolone funkcje arytmetyczne. Te funkcje są deklarowane w pliku nagłówkowym *complex.h* i należą do nich na przykład funkcje trygonometryczne `csin()`, `ctan()` itd. (patrz rozdział 16).

W standardzie C11 wsparcie dla liczb zespolonych jest opcjonalne. Jeśli implementacja nie zawiera pliku nagłówkowego *complex.h*, może być w niej zdefiniowane makro `__STDC_NO_COMPLEX__`.

Liczba zespolona *z* może być reprezentowana w postaci współrzędnych kartezjańskich jako $z = x + y \times i$, gdzie *x* i *y* są liczbami rzeczywistymi, zaś *i* jest *jednostką urojoną*, zdefiniowaną w postaci równania $i^2 = -1$. Liczba *x* jest nazywana częścią rzeczywistą, a liczba *y* urojoną częścią liczby *z*.

W języku C liczba zespolona jest reprezentowana przez parę wartości zmiennopozycyjnych dla części rzeczywistych i urojonych. Obydwie części są tego samego typu, czy to będzie `float`, `double` czy `long double`. Odpowiednie trzy zmiennopozycyjne typy zespolone są następujące:

- `float _Complex`
- `double _Complex`
- `long double _Complex`

Każdy z tych typów ma ten sam rozmiar i wyrównanie jak tablica dwóch elementów typu `float`, `double` lub `long double`.

Plik nagłówkowy *complex.h* definiuje makra `complex` i `I`. Makro `complex` jest synonimem słowa kluczowego `_Complex`. Makro `I` reprezentuje jednostkę urojoną *i* i jest typu `const float _Complex`:

```
#include <complex.h>
// ...
double complex z = 1.0 + 2.0 * I;
z *= I;    // Obróć z wokół punktu początkowego o 90°
           // w kierunku przeciwnym do wskazówek zegara.
```

C11 udostępnia również makra `CMPLX`, `CMPLXF` i `CMPLXL`, pozwalające tworzyć liczby zespolone na podstawie części rzeczywistej i urojonej. Na przykład liczba złożona `CMPLXL (1.0, 2.0)` jest równa liczbie z zdefiniowanej w powyższym przykładzie i jest typu `double complex`. Podobnie makra `CMPLXF` i `CMPLXL` zwracają liczby złożone typu `float complex` i `long double complex`. Implementacja może także zawierać następujące typy służące do reprezentowania *czystych liczb urojonych*: `float imaginary`, `double imaginary` i `long double imaginary`.

Typy wyliczeniowe

Typy wyliczeniowe są typami całkowitymi definiowanymi w programie. Definicja typu wyliczeniowego rozpoczyna się od słowa kluczowego `enum`, po którym może się znajdować identyfikator wyliczenia. Typ ten zawiera listę możliwych wartości wraz z nazwą:

```
enum [identyfikator] { lista wyliczenia };
```

W poniższym przykładzie zdefiniowany jest typ wyliczeniowy `enum color`:

```
enum color { black, red, green, yellow, blue, white=7, gray };
```

Identyfikator `color` jest *znacznikiem* tego typu wyliczeniowego. Identyfikatory z tej listy – `black`, `red`, itd. – są *stałymi wyliczenia* i są typu `int`. Możemy z nich korzystać w dowolnym miejscu ich zakresu ważności – na przykład jako stałych `case` w instrukcji `switch`.

Każda stała wyliczenia danego typu wyliczeniowego reprezentuje konkretną wartość, określoną pośrednio przez jej pozycję na liście lub bezpośrednio poprzez inicjalizację wartością wyrażenia stałego. Stała bez inicjalizacji ma wartość 0, jeśli znajduje się na pierwszej pozycji listy, lub wartość wcześniejszej stałej plus jeden. Zgodnie z tymi zasadami stałe wymienione we wcześniejszym przykładzie mają wartości 0, 1, 2, 3, 4, 7 i 8.

Wewnątrz zakresu typu wyliczeniowego możemy korzystać z tego typu w deklaracjach:

```
enum color bgColor = blue,         // Definiuje dwie zmienne
           fgColor = yellow;       // typu enum color.
void setFgColor( enum color fgc ); // Deklaracja funkcji za pomocą
                                   // parametru typu enum color.
```

Typ wyliczeniowy zawsze odpowiada jednemu ze standardowych typów całkowitych. Dzięki temu programy C mogą wykonywać zwykle operacje na zmiennych typów wyliczeniowych. W zależności od wartości zdefiniowanych dla stałych wyliczeniowych kompilator może wybrać odpowiedni typ całkowity. W poprzednim przykładzie do przedstawienia wszystkich wartości typu wyliczeniowego `enum color` wystarczyłby typ `char`.

Różne stałe w wyliczeniu mogą mieć tę samą wartość:

```
enum { OFF, ON, STOP = 0, GO = 1, CLOSED = 0, OPEN = 1 };
```

Jak widać w powyższym przykładzie, definicja typu wyliczeniowego nie musi zawsze zawierać znacznika. Ominięcie znacznika jest sensowne tylko wtedy, gdy chcemy jedynie zdefiniować stałe, bez definiowania żadnej zmiennej danego typu. Ten sposób definiowania stałych całkowitych jest zalecany bardziej niż korzystanie z długiej listy dyrektyw `#define`, jako że typ wyliczeniowy udostępnia kompilatorowi nazwy stałych wraz z ich wartościami liczbowymi. Te nazwy dają na przykład ogromną przewagę podczas debugowania.

Typ void

Specyfikator typu `void` oznacza brak wartości. W związku z tym nie można zdefiniować zmiennych czy stałych tego typu. Typ `void` można wykorzystywać na potrzeby opisane w kolejnych punktach.

void w deklaracjach funkcji

Funkcja, która nie zwraca wartości, jest typu `void`. Na przykład funkcja standardowa `perror()` jest definiowana przez prototyp:

```
void perror( const char * );
```

Słowo kluczowe `void` na liście parametrów prototypu funkcji oznacza, że funkcja nie przyjmuje paramentów:

```
FILE *tmpfile( void );
```

W rezultacie, jeśli spróbujemy wywołać funkcję za pomocą wyrażenia `tmpfile("name.tmp")`, kompilator wygeneruje komunikat błędu. Gdyby funkcja została zadeklarowana bez słowa `void` na liście parametrów, wówczas kompilator nie posiadałby wystarczających informacji o parametrach funkcji i nie miałby możliwości określenia, czy wywołanie funkcji jest prawidłowe.

Wyrażenia typu void

Wyrażenie `void` nie zwraca wartości. Wyrażeniem typu `void` jest na przykład wywołanie funkcji nie zwracającej żadnej wartości:

```
char filename[ ] = "memo.txt";
if ( fopen( filename, "r" ) == NULL )
  perror( filename );             // Wyrażenie typu void.
```

Operacja rzutowania `(void)`*`wyrażenie`* jawnie odrzuca wartość wyrażenia, na przykład wartość zwracaną przez funkcję:

```
(void)printf("Nie potrzebuję wartości zwracanej przez tę funkcję!\n");
```

Wskaźniki do typu void

Wskaźnik typu `void *` reprezentuje adres obiektu, lecz nie jego typ. Tego rodzaju pseudo-beztypowych wskaźników można używać głównie do deklarowania funkcji, które operują na różnych typach argumentów wskaźnikowych lub które zwracają wskaźnik „wielorakiego przeznaczenia". Przykładem są standardowe funkcje zarządzania:

```
void *malloc( size_t size );
void *realloc( void *ptr, size_t size );
void free( void *ptr );
```

Jak obrazuje przykład 2-3, wartość typu void możemy przypisać do wskaźnika innego typu lub odwrotnie, bez konieczności jawnego przekształcenia.

Przykład 2-3 *Korzystanie z typu void*

```
// usingvoid.c: Prezentuje użycie typu void
// ----------------------------------------------------
#include <stdio.h>
#include <time.h>
#include <stdlib.h>  // Udostępnia prototypy następujących funkcji
                     // void srand( unsigned int seed );
                     // int rand( void );
                     // void *malloc( size_t size );
                     // void free( void *ptr );
                     // void exit( int status );

enum { ARR_LEN = 100 };

int main()
{
  int i,                      // Uzyskujemy nieco miejsca w pamięci.
    *pNumbers = malloc(ARR_LEN * sizeof(int));

  if ( pNumbers == NULL )
  {
    fprintf(stderr, "Niewystarczająca ilość pamięci.\n");
    exit(1);
  }

  srand( (unsigned)time(NULL) );   // Inicjalizacja generatora
                                   // liczb losowych.

  for ( i=0; i < ARR_LEN; ++i )
    pNumbers[i] = rand() % 10000;  // Zachowaj kilka liczb losowych.

  printf("\n%d liczb losowych między 0 i 9999:\n", ARR_LEN );
  for ( i=0; i < ARR_LEN; ++i )        // Pętla wynikowa:
  {
    printf("%6d", pNumbers[i]);        // Wypisz jedną liczbę w
                                       // iteracji pętli
    if ( i % 10 == 9 ) putchar('\n');  // i nowy wiersz po każdych
  }                                    // 10 liczbach
```

```
    free( pNumbers );                    // Zwolnij miejsce w pamięci.
    return 0;
}
```

Wyrównanie obiektów w pamięci

Każdy kompletny typ obiektu nakłada na obiekty tego typu określone *wyrównanie*. Innymi słowy, typ określa rodzaj adresów w pamięci, w których mogą być przechowywane obiekty tego typu. Mogą to być wszystkie adresy, tylko parzyste adresy, lub tylko adresy podzielne przez cztery, i tak dalej. Wyrównanie typu jest wyrażane za pomocą liczby bajtów równej minimalnej odległości między obiektami tego typu w pamięci. Określone wartości wyrównań typów mogą się różnić w zależności od implementacji, lecz zawsze są one liczbami całkowitymi będącymi potęgą liczby 2, czyli 1, 2, 4, 8, i tak dalej. Wyrównanie o wartości większej od wyrównania innego typu jest określane jako *bardziej rygorystyczne*.

Standard C11 udostępnia operator `_Alignof`, służący do wyznaczenia wyrównania danego typu, oraz `_Alignas` służący do określenia wyrównania w definicji obiektu.

Operator `_Alignof`, podobnie jak operator `sizeof`, zwraca stałą wartość typu `size_t`, czyli typu całkowitego ze znakiem, zdefiniowanego w pliku *stddef.h* i innych plikach nagłówkowych. Na przykład poniższe wyrażenie zwraca wyrównanie typu `int`, które zwykle wynosi 4:

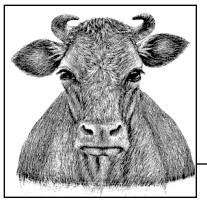
```
_Alignof(int)
```

Wartość wyrównania mniejsza lub równa `_Alignof(max_align_t)` jest nazywana wyrównaniem fundamentalnym. Wszystkie typy fundamentalne – czyli typy podstawowe i wskaźnikowe – mają wyrównanie fundamentalne. Typ `max_align_t` jest zdefiniowany w pliku nagłówkowym *stddef.h* i jego wyrównanie jest wspierane w każdym kontekście, włącznie z dynamiczną alokacją pamięci. Ponadto implementacja może także wspierać wyrównania większe niż `_Alignof(max_align_t)`, które są określane mianem *wyrównań rozszerzonych*.

Obiekt zdefiniowany za pomocą specyfikatora `_Alignas` może mieć bardziej rygorystyczne wyrównanie niż wymagane przez swój typ. Argument specyfikatora `_Alignas` może być wyrażeniem stałej całkowitej, której wartością jest poprawne wyrównanie lub typ, jak w poniższych przykładach:

```
_Alignas(4) short var;      // Definiuje zmienną var typu short
                            // o wyrównaniu czterobajtowym.
_Alignas(double) float x;   // Definiuje zmienną x typu float
                            // o wyrównaniu typu double.
```

Forma `_Alignas(type)` jest synonimem zapisu `_Alignas (_Alignof(type))`. Plik nagłówkowy *stdalign.h* definiuje `alignof` i `alignas` jako synonimy dla `_Alignof` i `_Alignas`. Dzięki temu w programie dołączającym nagłówek *stdalign.h* zamiast zapisu `_Alignas(int)` możemy użyć `alignas(int)`.



3
Literały

Literał w kodzie źródłowym języka C to token, oznaczający ustaloną wartość, która może być liczbą całkowitą, liczbą zmiennopozycyjną, znakiem lub łańcuchem. Typ literału jest determinowany przez jego wartość i sposób zapisu.

Omawiane tu literały różnią się od *literałów złożonych*, wprowadzonych przez standard C99. Literały złożone są zwykłymi obiektami modyfikowalnymi, podobnymi do zmiennych. Pełen opis literałów złożonych i specjalnego operatora, służącego do ich utworzenia, można znaleźć w rozdziale 5.

Stałe całkowite

Stałą całkowitą można wyrazić w postaci zwykłej dziesiętnej liczby całkowitej lub liczby w notacji ósemkowej lub szesnastkowej. Zamierzoną notację należy określić za pomocą przedrostka.

Stała dziesiętna rozpoczyna się od cyfry niezerowej. Na przykład `255` jest stałą dziesiętną o wartości 255 i podstawie 10.

Liczba rozpoczynająca się od cyfry zero jest interpretowana jako *stała ósemkowa*. Notacja ósemkowa (lub o podstawie osiem) wykorzystuje jedynie cyfry od 0 do 7. Na przykład liczba `047` jest prawidłową stałą ósemkową reprezentującą liczbę $4 \times 8 + 7$ i jest odpowiednikiem stałej dziesiętnej `39`. Stała dziesiętna `255` jest równa stałej ósemkowej `0377`.

Stała szesnastkowa rozpoczyna się od przedrostka `0x` lub `0X`. Cyfry szesnastkowe od A do F mogą być zapisywane małymi lub wielkimi literami. Na przykład liczby `0xff`, `0Xff`, `0xFF`, i `0XFF` reprezentują tę samą stałą szesnastkową, która jest odpowiednikiem stałej dziesiętnej `255`.

Typ stałych ma duże znaczenie, ponieważ definiowane przez nas stałe całkowite zostaną w pewnym momencie wykorzystane w wyrażeniach i deklaracjach. Typ stałej jest determinowany w chwili definiowania jej wartości. Stałe całkowite, takie jak wspomniane w przykładach, zwykle są typu `int`. Jednak jeśli wartość stałej całkowitej wykracza poza zakres typu `int`, stała musi mieć typ pozwalający na deklarację wyższych wartości. W tym przypadku kompilator wybierze z hierarchii typów pierwszy, który będzie wystarczająco pojemny, aby przechować jej wartość. W przypadku stałych dziesiętnych hierarchia typów wygląda następująco:

```
int, long, long long
```

W przypadku stałych szesnastkowych, hierarchia typów wygląda tak:

```
int, unsigned int, long, unsigned long, long long, unsigned long long
```

Na przykład w systemie 16-bitowym, stała dziesiętna `50000` jest typu `long`, ponieważ `int` może przechowywać wartości do 32767 ($2^{15} - 1$).

Na typ stałych w programie można wpłynąć w jawny sposób za pomocą przyrostków. Stała o przyrostku `l` lub `L` jest typu `long` (a jeśli to konieczne – większego, zgodnie z wspomnianą nieco wcześniej hierarchią). Podobnie, stała o przyrostku `ll` lub `LL` jest co najmniej typu `long long`. Przyrostek `u` lub `U` pozwala zdefiniować stałą o typie bez znaku. Przyrostki `long` i `unsigned` można ze sobą łączyć. W tabeli 3-1 zebrano kilka przykładów.

Tabela 3-1 *Przykłady stałych z przyrostkami*

Stała całkowita	Typ
0x200	`int`
512U	`unsigned int`
0L	`long`
0Xf0fUL	`unsigned long`
0777LL	`long long`
0xAAAllu	`unsigned long long`

Stałe zmiennopozycyjne

Stałe zmiennopozycyjne można zapisać zarówno w notacji dziesiętnej, jak i szesnastkowej. Notacje te są opisane w kolejnych punktach.

Dziesiętne stałe zmiennopozycyjne

Zwykła stała zmiennopozycyjna składa się z sekwencji cyfr dziesiętnych zawierających kropkę dziesiętną[2]. Wartość można także zaprezentować za pomocą mnożenia przez potęgę 10, jak w notacji naukowej: potęga 10 jest reprezentowana przez wykładnik poprzedzony literą `e` lub `E`. Stała zmiennopozycyjna zapisana przy użyciu wykładnika nie musi zawierać kropki dziesiętnej. W tabeli 3-2 zebrano kilka przykładów dziesiętnych stałych zmiennopozycyjnych.

Tabela 3-2 *Przykłady dziesiętnych stałych zmiennopozycyjnych*

Stała zmiennopozycyjna	Wartość
10.0	10
2.34E5	2.34×10^5
67e-12	$67.0 \times 10{-}^{12}$

Kropka może się także znaleźć na początku lub na końcu liczby. Dozwolone są więc liczby `10.` i `.234E6`. Jednakże liczba `10` bez kropki dziesiętnej będzie stałą całkowitą, a nie stałą zmiennopozycyjną.

Stała zmiennopozycyjna ma domyślny typ `double`. Natomiast jeśli dołączymy do niej przyrostek `F` lub `f`, utworzymy stałą typu `float`. Przyrostek `L` lub `l` pozwala utworzyć stałą typu `long double`, jak w poniższym przykładzie:

```
 float  f_var = 123.456F;            // Inicjalizacja zmiennej float
long double ld_var = f_var * 987E7L; // Inicjalizacja zmiennej
                                     // long double wynikiem mnożenia
                                     // o podwójnej precyzji.
```

Szesnastkowe stałe zmiennopozycyjne

W standardzie C99 wprowadzono *szesnastkowe* stałe zmiennopozycyjne, które mają kluczową przewagę nad dziesiętnymi liczbami zmiennopozycyjnymi: jeśli określimy wartość stałej w notacji szesnastkowej, będzie ona przechowywana w pamięci w binarnym formacie zmiennopozycyjnym komputera, z uniknięciem błędów zaokrągleń. Natomiast w reprezentacji wewnętrznej wartości „liczb zaokrąglonych", zapisanych w notacji dziesiętnej – na przykład 0.1 – części ułamkowe mogą być okresowe (zaokrąglanie liczb zmiennopozycyjnych można prześledzić w przykładzie 2-2).

Szesnastkowa stała zmiennopozycyjna składa się z przedrostka `0x` lub `0X`, sekwencji cyfr szesnastkowych z opcjonalną kropką dziesiętną (którą w tym przypadku moglibyśmy nazwać „kropką szesnastkową") i wykładnika potęgi o podstawie dwa. Wykładnik jest

2 Kropka dziesiętna w kodzie powinna być zapisana za pomocą kropki, a nie przecinka, który jest stosowany w tekście przy polskich ustawieniach językowych (przyp. tłum.).

liczbą *dziesiętną* wprowadzaną przez literę `p` lub `P`. Na przykład stała `0xa.fP-10` jest równa liczbie $(10 + 15/16) \times 2^{-10}$ (nie 2^{-16}) w notacji dziesiętnej. Odpowiednikami w zapisie tej samej wartości stałej są `0xA.Fp-10`, `0x5.78p-9`, `0xAFp-14` i `0x.02BCp0`. Każda różnica o 1 w wykładniku mnoży lub dzieli ułamek szesnastkowy przez 2, a każde przesunięcie kropki szesnastkowej odpowiada mnożeniu lub dzieleniu przez 16 lub 2^4.

W szesnastkowej stałej zmiennopozycyjnej musimy zawsze zapisać wykładnik, nawet jeśli jego wartość wynosi zero. Jest to niezbędne, aby odróżnić przyrostek `F` oznaczający typ (po wykładniku) od cyfry szesnastkowej `F` (z lewej strony kropki szesnastkowej). Gdyby wykładnik nie był wymagany, to przykładowa stała `0x1.0f` mogłaby reprezentować liczbę 1.0 typu `float` albo liczbę 1 + 15/256 domyślnego typu `double`, a taka niejednoznaczność jest niedopuszczalna.

Podobnie jak dziesiętne stałe zmiennopozycyjne, szesnastkowe stałe zmiennopozycyjne są domyślnie typu `double`. Aby utworzyć stałą typu `float`, należy dołączyć przyrostek `F` lub `f`, zaś stałą typu `long double` tworzy przyrostek `L` lub `l`.

Stałe znakowe

Stała znakowa składa się z co najmniej jednego znaku, ujętego w pojedyncze cudzysłowy. Oto kilka przykładów:

```
'a'   'XY'   '0'   '*'
```

Stałe znakowe mogą zawierać wszystkie znaki ze źródłowego zbioru znaków, z wyjątkiem znaku pojedynczego cudzysłowu `'`, znaku ukośnika wstecznego `\` oraz znaku nowego wiersza. Znaki te można zapisać za pomocą sekwencji specjalnych:

```
'\''   '\\'   '\n'
```

W piątym kroku tłumaczenia (patrz podrozdział „Jak działa kompilator C” w rozdziale 1) znaki i sekwencje specjalne w stałych znakowych są przekształcane w odpowiadające im znaki ze zbioru znaków wykonania programu. Wszystkie sekwencje specjalne, które mogą być używane w stałych znakowych, są opisane w dalszym punkcie „Sekwencje specjalne”.

Rozszerzone stałe znakowe są stałymi znakowymi zdefiniowanymi za pomocą jednego z przedrostków `L`, `u` i `U`. Mają one inny typ i zakres wartości od stałych znakowych zdefiniowanych bez przedrostka.

Typy i wartości stałych znakowych

Stałe znakowe, które nie są znakami rozszerzonymi, są typu `int`. Jeśli stała znakowa zawiera *jeden* znak, który może być reprezentowany przez pojedynczy bajt w zbiorze znaków wykonania programu, wówczas ma ona wartość kodu tego znaku. Na przykład stała `'a'` w kodzie ASCII lub ISO 8859-1 ma dziesiętną wartość 97. We wszystkich pozostałych

przypadkach, szczególnie jeśli stała znakowa zawiera więcej niż jeden znak, wartość stałej znakowej może się zmieniać w zależności od kompilatora.

Następujący fragment kodu sprawdza, czy odczytany znak jest cyfrą z przedziału od 1 do 5 włącznie:

```
#include <stdio.h>
int c = 0;

/* ... */
c = getchar();                              // Odczyt znaku.
if ( c != EOF && c > '0' && c < '6' )       // Porównanie wejścia do
                                            // stałych znakowych.
{
  /* Ten blok jest wykonywany, jeśli użytkownik wpisze cyfrę od 1  do 5. */
}
```


Jeśli typ `char` jest ze znakiem, wartość stałej znakowej może być także ujemna, ponieważ wartość ta jest wynikiem przekształcenia kodu znaku z `char` do `int`. Weźmy na przykład ISO 8859-1, który jest powszechnie używanym zbiorem znaków 8-bitowych, znanym również pod nazwą ISO Latin 1 lub zbiorem znaków ANSI. W tym zbiorze znaków, symbol waluty funt szterling `£` w kodzie szesnastkowym na wartość A3:

```
int c = '\xA3';                    // Symbol funta szterlinga
printf("Znak: %c     Kod: %d\n", c, c);
```

Jeśli zbiorem znaków wykonania programu jest ISO 8859-1, zaś typ `char` jest ze znakiem, wówczas instrukcja `printf` z poprzedniego przykładu wygeneruje następujący wynik:

```
Znak: £     Kod: -93
```

W programie wykorzystującym znaki, które nie są reprezentowane przez pojedynczy bajt, możemy korzystać z rozszerzonych stałych znakowych. Stała znaku rozszerzonego jest zapisywana za pomocą jednego z przedrostków `L`, `u` lub `U`. Przedrostek determinuje typ stałej znakowej, zgodnie z tabelą 3-3.

Tabela 3-3 *Typy stałych znakowych*

Przedrostek	Przykłady	Typ
brak	`a, \t`	`int`
`L`	`L'a', L'\u0100'`	`wchar_t` (zdefiniowany w stddef.h)
`u`	`U'a', u'\x3b3'`	`char16_t` (zdefiniowany w uchar.h)
`U`	`U'a', U'\u27FA'`	`char32_t` (zdefiniowany w uchar.h)

Wartość znaku rozszerzonego, który zawiera *jeden* znak wielobajtowy, możliwy do reprezentacji w zestawie znaków wykonania programu, jest równa kodowi odpowiadającemu

mu znakowi rozszerzonemu. Jest to wartość, która zostałaby zwrócona dla znaku rozszerzonego przez funkcję standardową `mbtowc()` (ang. *„multibyte to wide character" – znak wielobajtowy na rozszerzony*) lub przez `mbrtoc16()` lub `mbrtoc32()`, w zależności od typu rozszerzonej stałej znakowej.

W standardzie C11 wprowadzono typy Unicode `char16_t` i `char32_t` oraz odpowiadające im funkcje przekształceń. Znaki typu `char16_t` są kodowane w UTF-16, jeśli dana implementacja definiuje makro `__STDC_UTF_16__`. Podobnie, znaki typu `char32_t` są kodowane w UTF-32, jeśli implementacja definiuje makro `__STDC_UTF_32__`.



Wartość stałej znakowej składającej się z kilku znaków, na przykład `L'xy'`, jest nieokreślona. Aby zagwarantować przenośność kodu, należy się upewnić, że w programie nie ma zależności od takich stałych znakowych o określonej wartości.

Sekwencje specjalne

Sekwencje specjalne rozpoczynają się od znaku ukośnika wstecznego `\` i reprezentują jeden znak. Sekwencje specjalne pozwalają na reprezentowanie dowolnego znaku w stałej znakowej i literale tekstowym, włącznie ze znakami niedrukowalnymi i znakami, które w innym przypadku miałyby znaczenie specjalne, na przykład `'` i `"`. Tabela 3-4 przedstawia sekwencje specjalne rozpoznawane w języku C.

Tabela 3-4 *Sekwencje specjalne*

Sekwencje specjalne	Wartość znaku	Akcja na urządzeniu wyjściowym
`\'`	Pojedynczy cudzysłów (')	Wypisuje znak.
`\"`	Podwójny cudzysłów (")	
`\?`	Znak zapytania (?)	
`\\`	Ukośnik wsteczny (\)	
`\a`	Alarm	Generuje sygnał dźwiękowy lub wizualny.
`\b`	Cofanie	Przenosi aktywną pozycję o jeden znak wstecz.
`\f`	Znak nowej strony	Przenosi aktywną pozycję na początek następnej strony.
`\n`	Znak nowego wiersza	Przenosi aktywną pozycję na początek następnego wiersza.
`\r`	Znak powrotu karetki	Przenosi aktywną pozycję na początek bieżącego wiersza.

Tabela 3-4 *Sekwencje specjalne*

Sekwencje specjalne	Wartość znaku	Akcja na urządzeniu wyjściowym
`\t`	Tabulacja pozioma	Przenosi aktywną pozycję na początek następnego punktu tabulacji poziomej.
`\v`	Tabulacja pionowa	Przenosi aktywną pozycję na początek następnego punktu tabulacji pionowej.
`\o`, `\oo` lub `\ooo` (gdzie `o` jest cyfrą ósemkową)	Znak o określonym kodzie ósemkowym	Wypisuje znak.
`\xh[h…]` (gdzie `h` jest cyfrą szesnastkową)	Znak o określonym kodzie szesnastkowym	
`\uhhhh\Uhhhhhhhh`	Znak o określonej uniwersalnej nazwie znaku	

Aktywna pozycja w tabeli dotyczy pozycji, w której urządzenie wyjściowe wypisuje kolejny znak, czyli na przykład jest to pozycja kursora w konsoli. Zachowanie urządzenia wyjściowego nie jest zdefiniowane w następujących przypadkach: jeśli sekwencja specjalna `\b` (cofanie) znajduje się na początku wiersza; jeśli `\t` (tabulacja) znajduje się na końcu wiersza; lub jeśli `\v` (tabulacja pionowa) znajduje się na końcu strony.

Zgodnie z tabelą 3-4 uniwersalne nazwy znaków są również uważane za sekwencje specjalne. Uniwersalne nazwy znaków pozwalają na określenie dowolnego znaku z rozszerzonego zbioru znaków, niezależnie od używanego kodowania. Więcej informacji można znaleźć w punkcie „Uniwersalne nazwy znaków” w rozdziale 1.

Dowolny kod znaku z zakresu wartości typu `unsigned char` – lub dowolnego kodu rozszerzonego z zakresu wartości `wchar_t` – możemy też określić, używając *ósemkowych* i *szesnastkowych sekwencji specjalnych*, pokazanych w tabeli 3-5.

Tabela 3-5 *Przykłady ósemkowych i szesnastkowych sekwencji specjalnych*

Ósemkowe	Szesnastkowe	Opis
`\0`	`\x0`	Znak null.
`\033`	`\x1B`	Znak kontrolny `ESC` („escape”).
`\33`		
`\376`	`\xfe`	Znak o kodzie dziesiętnym 254.
`\417`	`\x10f`	Niedozwolony, ponieważ numeryczna wartość jest poza zakresem typu `unsigned char`.

Tabela 3-5 *Przykłady ósemkowych i szesnastkowych sekwencji specjalnych*

Ósemkowe	Szesnastkowe	Opis
`L'\417'`	`L'\x10f'`	Tak lepiej! To teraz stała rozszerzona; jej typ to `wchar_t`.
-	`L'\xF82'`	Inna rozszerzona stała znakowa.
-	`U'\x222B'`	Rozszerzona stała znakowa typu `char32_t`.

Nie istnieje ósemkowy odpowiednik ostatnich stałych przedstawionych w tabeli, ponieważ ósemkowe sekwencje specjalne mogą zawierać maksymalnie trzy cyfry ósemkowe. Z tego samego powodu rozszerzona stała znakowa `L'\3702'` zostanie zinterpretowana jako składająca się z dwóch znaków `L'\370'` i `L'2'`.

Literały łańcuchowe

Literał łańcuchowy składa się z sekwencji znaków (i/lub sekwencji specjalnych) ujętych w podwójne cudzysłowy. Przykład:

```
"Witaj świecie!\n"
```

Poszczególne znaki literału łańcuchowego podlegają takim samym regułom, jak wartości znaków w stałych znakowych. Literały łańcuchowe mogą zawierać wszystkie znaki wielobajtowe ze źródłowego zbioru znaków. Jedynym wyjątkiem są znaki podwójnych cudzysłowów ", znak ukośnika wstecznego \ oraz znak nowego wiersza, które muszą być reprezentowane przez sekwencje specjalne. Na przykład wszystkie ukośniki wsteczne w ścieżkach do plików w systemie Windows muszą być zapisane znakami \\. Następująca instrukcja `printf` wygeneruje dźwięk alarmu, a następnie wypisze informację o ścieżce do katalogu dokumentów, ujętej w cudzysłowy, przy czym specyfikator `%s` zostanie zastąpiony literałem łańcuchowym wskazywanym przez argument `doc_path`:

```
char doc_path[128] = ".\\share\\doc";   // Czyli ".\share\doc"
printf("\aSprawdź dokumentację w katalogu \"%s\"\n", doc_path);
```

Literał łańcuchowy jest statyczną tablicą elementów typu `char`, zawierającą kody znaków, po których następuje koniec łańcucha, czyli znak null `\0` (patrz również rozdział 8.). Pusty łańcuch `""` zajmuje w pamięci dokładnie jeden bajt, który przechowuje kończący znak null. Znaki, które nie mogą być reprezentowane przez jeden bajt, są przechowywane jako znaki wielobajtowe.

Jak przedstawiono w poprzednim przykładzie, literał łańcuchowy może posłużyć do zainicjalizowania tablicy elementami typu `char`. Literał łańcuchowy może także służyć do inicjalizacji wskaźnika do typu `char`:

```
char *pStr = "Witaj, świecie!";      // pStr wskazuje na pierwszy znak 'W'
```

W tego typu inicjalizacji literał łańcuchowy, tak samo jak nazwa tablicy, reprezentuje adres pierwszego elementu.

W przykładzie 3-1 tablica `error_msg` zawiera trzy wskaźniki do typu `char`. Do każdego z nich jest przypisywany adres pierwszego znaku literału tekstowego.

Przykład 3-1 *Przykładowa funkcja error_exit()*

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

void error_exit(unsigned int error_n)  // Wypisuje ostatni komunikat
                                       // błędu i kończy
{                                      // działanie programu.
  char * error_msg[ ] = { "Nieznany kod błędu.\n",
                          "Brak pamięci.\n",
                          "Niedozwolony dostęp do pamięci.\n" };
  unsigned int arr_len = sizeof(error_msg)/sizeof(char *);
  if ( error_n >= arr_len )
    error_n = 0;
  fputs( error_msg[error_n], stderr );
  exit(1);
}
```

Standard C11 udostępnia nowy przedrostek `u8`, który pozwala na zdefiniowanie literału łańcuchowego UTF-8. Znaki wielobajtowe w tablicy typu char, zdefiniowanej przez literał łańcuchowy UTF-8, są zakodowane w UTF-8. Literał łańcuchowy w postaci `u8"..."` nie różni się więc od literału łańcuchowego bez przedrostka, o ile domyślnym kodowaniem implementacji znaków wielobajtowych jest UTF-8.

Podobnie jak w przypadku rozszerzonych stałych znakowych, korzystając z przedrostków `L`, `u` lub `U` można definiować literały łańcuchowe składające się ze znaków rozszerzonych. W ten sposób definiujemy tak zwane *rozszerzone literały łańcuchowe*, które zawierają tablicę znaków rozszerzonych zakończoną znakiem o wartości `0`. Typ elementów w tablicy determinuje przedrostek.

Rozszerzony literał łańcuchowy jest definiowany za pomocą przedrostka `L`:

```
L"Oto rozszerzony literał łańcuchowy."
```

To wyrażenie definiuje statyczną tablicę zakończoną znakiem null, zawierającą elementy typu `wchar_t`. Tablica jest inicjalizowana poprzez przekształcenie znaków wielobajtowych literału łańcuchowego na znaki rozszerzone w ten sam sposób, w jaki dokonałaby tego funkcja standardowa `mbstowcs()` (ang. *„multibyte string to wide character string” – łańcuch wielobajtowy na łańcuch znaków rozszerzonych*).

Przedrostki `u` i `U` wprowadzone w standardzie C11 oznaczają tablicę statyczną znaków rozszerzonych typu `char16_t` lub `char32_t`. Znaki wielobajtowe w tych rozszerzonych

literałach łańcuchowych są w niejawny sposób przekształcane w znaki rozszerzone w kolejnych wywołaniach funkcji `mbrtoc16()` lub `mbrtoc32()`.

W następującym przykładzie `\u03b1` jest uniwersalną nazwą znaku α (greckiej litery alfa), zaś `wprintf()` jest rozszerzoną wersją funkcji `printf()`, która formatuje i wypisuje łańcuchy znaków rozszerzonych:

```
double angle_alpha = 90.0/3;
wprintf( L"Kąt \u03b1 ma %lf stopni.\n", angle_alpha );
```

Jeśli dowolny znak wielobajtowy lub sekwencja specjalna w literale łańcuchowym nie ma reprezentacji w zbiorze znaków wykonania programu, wartość literału łańcuchowego jest nieokreślona, czyli jest zależna od kompilatora.

Preprocesor kompilatora łączy sąsiadujące literały łańcuchowe – czyli te, które są rozdzielone tylko białym znakiem – w jeden łańcuch. Jak przedstawia następujący przykład, dzięki temu mechanizmowi można w prosty sposób zwiększyć czytelność łańcucha dzieląc go na kilka wierszy:

```
#define PRG_NAME "EasyLine"
char msg[ ] = "Instalacja " PRG_NAME
                " została ukończona.";
```

Gdyby dowolny z tych literałów miał przedrostek, wówczas powstały łańcuch byłby traktowany jak literał łańcuchowy o takim samym przedrostku. To, czy literały tekstowe o różnych przedrostkach mogą być łączone, zależy od implementacji kompilatora.

Inny sposób dzielenia literału łańcuchowego na kilka wierszy polega na umieszczeniu znaku ukośnika wstecznego na końcu wiersza, jak na poniższym przykładzie:

```
char info[] =
"To jest literał łańcuchowy podzielony na\
 kilka wierszy kodu źródłowego.\nJeszcze jeden wiersz:\n\
wystarczy, tutaj łańcuch się kończy.";
```

Łańcuch jest kontynuowany *od początku* następnego wiersza. Wszelkie znaki spacji z lewej strony, jak spacja przed słowem `kilka` w powyższym przykładzie, wchodzą w skład literału łańcuchowego. Ponadto zdefiniowany tutaj literał łańcuchowy zawiera dokładnie dwa znaki nowego wiersza. Jeden znajduje się tuż przed słowem `Jeszcze`, a kolejny tuż przed słowem `wystarczy`. Innymi słowy, są to tylko te znaki, które zostały w jawny sposób zapisane jako `\n`.

Kompilator interpretuje sekwencje specjalne przed połączeniem sąsiadujących łańcuchów (patrz punkt „Etapy tłumaczenia kompilatora C” w rozdziale 1.). W wyniku tego, dwa następujące literały łańcuchowe tworzą jeden łańcuch znaków rozszerzonych, który rozpoczyna się od dwóch znaków `'\xA7'` i `'2'`:

```
L"\xA7" L"2 i tak dalej"
```

Jednakże, jeśli łańcuch zostanie zapisany jednym ciągiem w postaci `L"\xA72 i tak dalej"`, wówczas pierwszym znakiem w łańcuchu będzie znak rozszerzony `'\xA72'`.

Chociaż C nie zabrania rygorystycznie modyfikowania literałów łańcuchowych, nie należy tego próbować. W następującym przykładzie druga instrukcja jest próbą zastąpienia pierwszego znaku w łańcuchu:

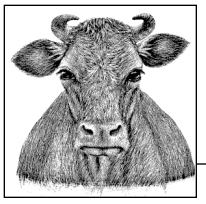
```
char *p = "dom";          // Inicjalizacja wskaźnika do  char.
*p = 'm';        // To nie jest dobry pomysł!
```

Ta instrukcja nie jest przenośna i w niektórych systemach wygeneruje błąd czasu wykonania. Po pierwsze, kompilator, traktując literał łańcuchowy jak stałą, może go umieścić w pamięci tylko do odczytu, zatem próba zapisu spowoduje błąd. Po drugie, jeśli program wykorzystuje kilka identycznych literałów łańcuchowych, kompilator może je przechowywać w tej samej lokalizacji, dlatego modyfikacja jednego z nich spowoduje nieoczekiwany rezultat podczas dostępu do drugiego.

Jednakże jeśli korzystamy z literałów łańcuchowych do zainicjalizowania *zmiennej tablicowej*, jej zawartość możemy modyfikować:

```
char s[] = "house";       // Inicjalizacja tablicy typu char.
s[0] = 'm';               // Teraz tablica zawiera napis "mouse".
```

Korzystając z odpowiedniego rozszerzonego literału łańcuchowego, można w ten sam sposób inicjalizować tablice z elementami typu `wchar_t`, `char16_t` lub `char32_t`.



4
Przekształcenia typów

W języku C w jednej operacji można uwzględniać operandy różnych typów. Można na przykład wykonać następujące operacje:

```
double dVar = 2.5;   // Definicja zmiennej dVar typu double.
dVar *= 3;           // Mnożenie dVar przez stałą całkowitą.
if ( dVar < 10L )    // Porównanie dVar ze stałą typu long.
{ /* ... */ }
```

Gdy operandy są różnych typów, przed wykonaniem operacji kompilator próbuje przekształcić je w taki sam typ. Ponadto w pewnych sytuacjach musimy umieścić w programie instrukcje dotyczące przekształceń. W procesie tym wartość zwracana w wyrażeniu jest przekształcana w nowy typ. Może to być typ `void` (co oznacza, że wartość wyrażenia została odrzucona – patrz punkt „Wyrażenia typu void" w rozdziale 2.) lub typ skalarny – czyli typ arytmetyczny lub wskaźnikowy. Na przykład wskaźnik do struktury można przekształcić w inny typ wskaźnikowy. Jednakże rzeczywistej wartości struktury nie można przekształcić w strukturę innego typu.

Gdy typy operandów są niezgodne lub gdy wywołujemy funkcję z wykorzystaniem argumentu, którego typ nie pasuje do odpowiedniego parametru funkcji, kompilator stosuje *niejawne* przekształcenia typu. W razie konieczności program dokonuje niejawnego przekształcenia typu także podczas inicjalizowania zmiennych lub przypisywania do nich wartości. Jeśli niezbędne przekształcenie jest niemożliwe, kompilator generuje komunikat błędu.

Za pomocą *operatora rzutowania* można *jawnie* przekształcić wartości z jednego typu na inny (patrz rozdział 5.):

```
(nazwa_typu) wyrażenie
```

Zastosowanie operatora rzutowania w poniższym przykładzie powoduje, że dzielenie jednej zmiennej całkowitej przez inną jest przeprowadzane jak operacja na liczbach zmiennopozycyjnych:

```
int sum = 22, count = 5;
double mean = (double)sum / count;
```

Ponieważ operator rzutowania ma wyższy priorytet niż dzielenie, najpierw wartość zmiennej `sum` w tym przykładzie jest przekształcana do typu `double`. Przed wykonaniem dzielenia kompilator musi niejawnie przekształcić dzielnik, czyli wartość `count`, do tego samego typu.

Operatora rzutowania należy używać zawsze, gdy istnieje ryzyko utraty informacji, jak na przykład w przypadku przekształceń z typu `int` w `unsigned int`. Jawne rzutowanie pozwala uniknąć ostrzeżeń ze strony kompilatora i sygnalizuje innym programistom istnienie przekształceń typów w programie. Na przykład użycie jawnego rzutowania do typu `void` przy odrzucaniu wartości funkcji służy jako przypomnienie, że w ten sposób możemy zignorować informacje o błędach generowanych przez funkcję.

Jednakże, aby zobrazować niejawne przekształcenia typu przeprowadzane przez kompilator, w przykładach przedstawionych w tym rozdziale operatora rzutowania używamy tylko, gdy jest to niezbędne.

Przekształcenia typów arytmetycznych

Przekształcenia typów są zwykle możliwe pomiędzy dowolnymi dwoma typami arytmetycznymi i w razie konieczności są niejawnie przeprowadzane przez kompilator. W wyniku przekształcenia wartość wyrażenia zostanie zachowana, jeśli tylko nowy typ może ją reprezentować. Nie zawsze jest to możliwe. Na przykład gdy przekształcamy wartość ujemną do typu bez znaku lub gdy przekształcamy ułamek zmiennopozycyjny typu `double` do typu `int`, nowy typ po prostu nie może reprezentować wartości oryginalnej. W tych przypadkach kompilator zwykle generuje ostrzeżenia.

Hierarchia typów

Gdy operandy arytmetyczne są różnych typów, niejawna konwersja typów zależy od *rangi przekształceń*. Ranking typów powstaje na podstawie następujących reguł:

- Dwa różne typy całkowite bez znaku mają inną rangę przekształceń. Jeśli jeden typ składa się z większej liczby bajtów, wówczas ma wyższą rangę.
- Każdy typ całkowity ze znakiem ma taką samą rangę jak odpowiednik bez znaku. Typ `char` ma tę samą rangę, jak `signed char` i `unsigned char`.
- Rangi standardowych typów całkowitych wyglądają następująco:

  ```
  _Bool < char < short < int < long < long long
  ```

- Dowolny standardowy typ całkowity ma wyższą rangę niż rozszerzony typ całkowity tej samej wielkości (rozszerzone typy całkowite zostały opisane w punkcie „Typy całkowite o określonej liczbie bajtów (C99)" w rozdziale 2).
- Każdy typ wyliczeniowy ma taką samą rangę, jak jego odpowiednik typu całkowitego (patrz „Typy wyliczeniowe" w rozdziale 2).
- Rangi typów zmiennopozycyjnych wyglądają następująco:

  ```
  float < double < long double
  ```
- Typ zmiennopozycyjny o najniższej randze, czyli `float`, ma wyższą rangę niż dowolny typ całkowity.
- Każdy zespolony typ zmiennopozycyjny ma tę samą rangę, co jego część rzeczywista i urojona.

Promocja typu całkowitego

W dowolnych wyrażeniach zamiast operandu typu `int` lub `unsigned int` można zawsze użyć wartości, której typ ma niższą rangę niż typ `int`. Jako operand całkowity może także służyć pole bitowe (pola bitowe są omówione w rozdziale 10.). W tych przypadkach kompilator stosuje *promocję typu całkowitego*. Dowolny operand, którego ranga jest niższa niż ranga typu `int`, jest automatycznie przekształcany do typu `int`, o ile typ `int` może reprezentować wszystkie wartości oryginalnego typu operandu. Jeśli typ `int` jest niewystarczający, operand jest przekształcany do typu `unsigned int`.

Promocja liczb całkowitych zawsze zachowuje wartość operandu. Oto kilka przykładów:

```
char c = '?';
unsigned short var = 100;

if ( c < 'A' )    // Stała znakowa 'A' jest typu int: wartość
                  // c jest niejawnie promowana do typu int w celu
                  // przeprowadzenia porównania.

var = var + 1;    // Zanim zostanie przeprowadzone dodawanie, wartość
                  // var jest promowana do typu int lub unsigned int.
```

W ostatniej instrukcji, przed wykonaniem dodawania kompilator dokonuje promocji wartości pierwszego składnika, czyli wartości zmiennej `var`, do typu `int` lub `unsigned int`. Jeśli `int` i `short` mają ten sam rozmiar, co jest możliwe na komputerach 16-bitowych, wówczas typ `int` ze znakiem jest niewystarczający do przechowywania wszystkich możliwych wartości zmiennej `var` typu `unsigned short`. W tym przypadku wartość `var` jest promowana do typu `unsigned int`.

Zwykłe przekształcenia arytmetyczne

Zwykłe przekształcenia arytmetyczne są niejawnymi przekształceniami dokonywanymi automatycznie dla większości operatorów, jeśli operandy są odmiennych typów arytmetycznych. Celem zwykłych przekształceń arytmetycznych jest odnalezienie *wspólnego typu rzeczywistego* dla wszystkich operandów i wyniku operacji.

Zwykłe przekształcenia arytmetyczne są przeprowadzane niejawnie dla następujących operatorów:

- Operatorów arytmetycznych przyjmujących dwa operandy: `*`, `/`, `%`, `+` i `-`
- Operatorów porównania i przyrównania: `<`, `<=`, `>`, `>=`, `==` i `!=`
- Operatorów binarnych: `&`, `|` i `^`
- Operatora trójargumentowego `?:` (w przypadku drugiego i trzeciego operandu).

Z wyjątkiem operatorów porównania i przyrównania, wspólny typ rzeczywisty uzyskany w wyniku zwykłego przekształcenia arytmetycznego jest zwykle identyczny jak typ wyniku. Jednakże jeśli jeden lub kilka operandów jest złożonego typu zmiennopozycyjnego, wówczas wynik również będzie miał złożony typ zmiennopozycyjny.

Zwykłe przekształcenia arytmetyczne są stosowane według następujących reguł:

1. Jeśli dowolny z operandów jest typu zmiennopozycyjnego, wówczas operand o niższej randze jest przekształcany do typu o tej samej randze, co typ drugiego operandu. Typy rzeczywiste są jednak przekształcane tylko do typów rzeczywistych, zaś typy zespolone tylko do typów zespolonych.

   Innymi słowy, jeśli jeden z operandów ma zespolony typ zmiennopozycyjny, zwykłe przekształcenie arytmetyczne dopasowuje tylko typ rzeczywisty, na którym oparty jest operand. Kilka przykładów:

```
#include <complex.h>
// ...
short n = -10;
double x = 0.5, y = 0.0;
float _Complex f_z = 2.0F + 3.0F * I;
double _Complex d_z = 0.0;

y  = n * x;       // Wartość n jest przekształcana do
                  // typu double.
d_z = f_z + x;    // Tylko wartość f_z jest przekształcana
                  // do typu double _Complex.
                  // Wynik operacji też jest typu
                  // double _Complex.
f_z = f_z / 3;    // Stała wartość 3 jest przekształcana
                  // do typu float.
```

```
d_z = d_z - f_z;  // Wartość f_z jest przekształcana do
                  // typu double _Complex.
```

2. Jeśli obydwa operandy są typu całkowitego, najpierw przeprowadzana jest promocja liczb całkowitych na obydwu operandach. Jeśli po promocji typy operandów nadal się różnią, przekształcenia odbywają się następująco:
   - Jeśli jeden z operandów jest typu *T bez znaku*, którego ranga jest co najmniej tak wysoka jak typu drugiego operandu, wówczas drugi operand jest przekształcany do typu *T*.
   - W przeciwnym razie, jeden operand jest typu *T ze znakiem*, którego ranga jest wyższa niż ranga typu drugiego operandu. Drugi operand jest przekształcany do typu *T* tylko wtedy, gdy typ *T* może reprezentować wszystkie wartości poprzedniego typu. Jeśli nie, wówczas obydwa operandy są przekształcane do typu bez znaku, który odpowiada typowi *T* ze znakiem.

Następujący kod zawiera kilka przykładów:

```
int i = -1;
unsigned int limit = 200U;
long n = 30L;

if ( i < limit )
  x = limit * n;
```

W tym przykładzie, aby dokonać porównania warunku w instrukcji `if`, wartość `i`, czyli −1, musi być najpierw przekształcona do typu `unsigned int`. Wynikiem jest duża liczba dodatnia. W systemie 32-bitowym liczba ta jest równa $2^{32} - 1$, zaś w dowolnym systemie jest większa niż `limit`. Dlatego warunek `if` jest fałszywy.

W ostatnim wierszu przykładu wartość zmiennej limit jest przekształcana do typu zmiennej `n`, czyli `long`, o ile wartość zakresu typu `long` zawiera cały zakres wartości typu `unsigned int`. W przeciwnym razie – na przykład jeśli zarówno typy `int` i `long` mają rozmiar 32 bitów – wówczas obydwie mnożne są przekształcane do typu `unsigned long`.

Zwykłe przekształcenia arytmetyczne zachowują wartość operandu z wyjątkiem następujących przypadków:

- Gdy liczba całkowita o dużej wartości jest przekształcana do typu zmiennopozycyjnego, precyzja typu docelowego może być niewystarczająca do dokładnego reprezentowania liczby.
- Wartości ujemne są poza zakresem wartości typów bez znaku.

W tych dwóch przypadkach wartości przekraczające zakres lub precyzję typu docelowego są przekształcane zgodnie z zasadami opisanymi w punkcie „Wyniki przekształceń typów arytmetycznych” w dalszej części tego rozdziału.

Inne niejawne przekształcenia typu

Kompilator automatycznie przekształca wartości arytmetyczne również w następujących przypadkach:

- W przypisaniach i inicjalizacjach wartość operandu z prawej strony jest zawsze przekształcana do typu operandu z lewej strony.
- W wywołaniach funkcji argumenty są przekształcane do typów odpowiadających im parametrów. Jeśli parametry nie zostały zadeklarowane, wówczas przeprowadzana jest *promocja domyślnych argumentów*: na argumentach całkowitych przeprowadzana jest promocja liczb całkowitych, zaś argumenty typu `float` są promowane do typu `double`.
- W instrukcjach `return`, wartość wyrażenia `return` jest przekształcana do typu zwracanego przez funkcję.

W przypisaniu złożonym, takim jak `x += 2.5`, najpierw zwykłym przekształceniom arytmetycznym poddawane są wartości obydwu operandów, następnie wynik operacji arytmetycznej przekształcany jest do typu lewego operandu, jak w prostym przypisaniu. Jeśli typ operandu z lewej strony jest niewystarczający do reprezentowania wartości prawego operandu, większość kompilatorów wygeneruje ostrzeżenie. Oto kilka przykładów:

```
#include <math.h>     // Deklaruje funkcję double sqrt( double ).

int   i = 7;
float x = 0.5;        // Wartość stałej jest przekształcana z typu
                      // double do typu float.

i = x;                // Wartość x jest przekształcana z typu float do int.

x += 2.5;             // Przed dodawaniem wartość x jest przekształcana
                      // do typu double. Następnie, wartość sumy jest
                      // przekształcana do typu float, przed przypisaniem
                      // do x.

x = sqrt( i );        // Oblicza pierwiastek kwadratowy z i:
                      // Argument jest przekształcany z typu int do
                      // typu double; przed przypisaniem do x
                      // zwracana wartość jest przekształcana
                      // z typu double do typu float.

long my_func()
{
  /* ... */
  return 0;           // Stała 0 jest przekształcana do typu long,
                      // czyli typu zwracanego przez funkcję.
}
```

Wyniki przekształceń typów arytmetycznych

Ponieważ różne typy mają różne przeznaczenie, rozmiary i ograniczenia, przekształcenie wartości jednego typu w inny zwykle wymaga zastosowania reguł specjalnych, uwzględniających te szczegóły. Ogólnie rzecz biorąc, dokładny wynik przekształcenia typu zależy przede wszystkim od cech typu docelowego.

Przekształcenia do typu _Bool

Do typu `_Bool` można przekształcić dowolną wartość dowolnego typu skalarnego. Jeśli wartość skalarna jest równa 0, wynikiem będzie 0, czyli `false`; natomiast jeśli wartość jest niezerowa, wynikiem będzie 1, czyli `true`. Ponieważ wskaźnik pusty jest uznawany za równy zero, to po przekształceniu do typu `_Bool` otrzymuje wartość `false`.

Przekształcenia do typów całkowitych bez znaku innych niż _Bool

Wartości całkowite są zawsze zachowywane, o ile mieszczą się w zakresie nowego typu bez znaku – innymi słowy, jeśli ich wartość mieści się w przedziale od 0 do U*typ*_MAX, gdzie U*typ*_MAX jest największą wartością, jaką może reprezentować typ `unsigned` *typ*.

Jeśli wartość wykracza poza zakres nowego typu bez znaku, w wyniku przekształcenia uzyskamy wartość powstałą przez dodanie lub odjęcie jedynki (U*typ*_MAX + 1) tyle razy, aż wynik znajdzie się w zakresie nowego typu. Następujący przykład ilustruje przypisywanie wartości ujemnych do typu całkowitego bez znaku:

```
#include <limits.h>        // Definiuje makra USHRT_MAX, UINT_MAX, itd.
unsigned short  n = 1000;  // Wartość 1000 jest w zakresie typu
                           // unsigned short;
n = -1;                    // wartość -1 musi być przekształcona.
```

Aby dostosować wartość ze znakiem – 1 do zmiennej o typie bez znaku, program niejawnie dodaje do tej wartości USHRT_MAX + 1, aż wynik znajdzie się w zakresie typu. Ponieważ −1 + (USHRT_MAX + 1) = USHRT_MAX, ostatnia instrukcja w poprzednim przykładzie jest odpowiednikiem `n = USHRT_MAX;`.

W przypadku dodatnich wartości całkowitych, w wyniku wielokrotnego odejmowania (U*typ*_MAX + 1), aż do osiągnięcia wartości mieszczącej się w zakresie nowego typu, uzyskamy liczbę równą reszcie z dzielenia przez (U*typ*_MAX + 1), jak w poniższym przykładzie:

```
#include <limits.h>        // Definiuje makra USHRT_MAX, UINT_MAX, itd.
unsigned short  n = 0;
n = 0xFEDCBA;              // Wartość jest poza zakresem typu
                           // unsigned short.
```

Jeśli typ `unsigned short` ma 16 bitów, wówczas jego wartość maksymalna USHRT_MAX wynosi FFFF w notacji szesnastkowej. Gdy wartość FEDCBA jest przekształcana do typu

`unsigned short`, wynik ma tę samą wartość, jak reszta z dzielenia przez liczbę szesnastkową 10000 (czyli `USHRT_MAX` + 1), która jest zawsze równa lub mniejsza od FFFF. W tym przypadku wartość przypisana do zmiennej `n` to liczba szesnastkowa DCBA.

Aby przekształcić zmiennopozycyjną liczbę całkowitą do typu całkowitego bez znaku lub ze znakiem, kompilator odrzuca część ułamkową. Jeśli pozostała część całkowita znajduje się poza zakresem nowego typu, wynik przekształcenia jest nieokreślony. Na przykład:

```
double x = 2.9;
unsigned long n = x;        // Ułamkowa część x zostanie utracona.
unsigned long m = round(x); // Jeśli x jest nieujemna, wynik tej
                            // operacji jest równoważny z m = x + 0.5;
```

Podczas inicjalizacji zmiennej `n` w tym przykładzie, wartość `x` jest przekształcana z typu `double` do `unsigned long` poprzez odrzucenie wartości ułamkowej 0,9. Do `n` zostanie przypisana część całkowita 2. Podczas inicjalizacji zmiennej `m`, funkcja C99 `round()` zaokrągla wartość `x` do najbliższej wartości całkowitej (większej lub mniejszej) i zwraca wartość typu `double`. Część ułamkowa wynikowej wartości typu `double` – w tym przypadku 3.0 – jest wobec tego równa zero, zanim zostanie odrzucona w wyniku przekształcenia typu podczas przypisywania do zmiennej `m`.

Gdy liczba zespolona jest przekształcana do typu całkowitego bez znaku, najpierw odrzucana jest część urojona. Wynikowa wartość zmiennopozycyjna jest przekształcana, jak opisano wcześniej. Na przykład:

```
#include <limits.h>         // Definiuje makra takie jak UINT_MAX.
#include <complex.h>        // Definiuje makra takie jak stała
                            // urojona I.
unsigned int  n = 0;
float _Complex  z = -1.7 + 2.0 * I;

n = z;                     // W tym przypadku wynik jest taki sam,
                           // jak w przypadku n = -1;
                           // Wartość wynikowa n wynosi UINT_MAX.
```

Część urojona `z` zostaje odrzucona pozostawiając rzeczywistą wartość zmiennopozycyjną – 1,7. Następnie odrzucana jest również wartość ułamkowa liczby zmiennopozycyjnej. Pozostała część całkowita – 1 jest przekształcana w `unsigned int` poprzez dodanie `UINT_MAX + 1` i ostatecznie wartość przypisana do `n` jest równa `UINT_MAX`.

Przekształcenia do typów całkowitych ze znakiem

Problem przekraczania zakresu wartości typu całkowitego może także dotyczyć sytuacji, gdy wartość jest przekształcana z typu całkowitego, czy to ze znakiem, czy bez znaku, do innego typu całkowitego ze znakiem; na przykład jeśli wartość jest przekształcana

z typu long lub unsigned int do typu int. Wynik takiego przepełnienia podczas przekształceń do typu całkowitego ze znakiem, w przeciwieństwie do przekształceń do typów całkowitych bez znaku, zależy od danej implementacji.

Większość kompilatorów odrzuca najwyższe bity w reprezentacji binarnej oryginalnej wartości i interpretuje najniższe bity zgodnie z nowym typem. Jak obrazuje poniższy przykład, w wyniku takiej strategii przekształceń istniejący wzorzec bitowy typu unsigned int jest interpretowany jak wartość int ze znakiem:

```
#include <limits.h>          // Definiuje makra takie jak UINT_MAX
int i = UINT_MAX;            // Wynik: i = -1 (kod uzupełnień do
                             // dwójki)
```

Jednakże w zależności od kompilatora próby takich przekształceń mogą skutkować także wygenerowaniem sygnału informującego program o przepełnieniu zakresu wartości.

Gdy liczba rzeczywista lub złożona liczba zmiennopozycyjna jest przekształcana do typu całkowitego ze znakiem, obowiązują te same reguły, jak w przypadku przekształceń do typu całkowitego bez znaku, opisanych w poprzednim punkcie.

Przekształcenia do typów rzeczywistych zmiennopozycyjnych

Nie wszystkie wartości można dokładnie reprezentować za pomocą typu zmiennopozycyjnego. Na przykład, chociaż zakres wartości typu float obejmuje zakres typów long i long long, float charakteryzuje się precyzją tylko do sześciu cyfr dziesiętnych. Dlatego niektórych wartości long nie można przechowywać bezpośrednio w obiekcie float. Wynikiem takiego przekształcenia jest następna mniejsza lub następna większa możliwa do reprezentowania wartość, co pokazuje następujący przykład:

```
long  l_var = 123456789L;
float f_var = l_var;          // Bezpośrednio przekształca wartość
                              // long we float.

printf("Błąd zaokrąglenia (f_var - l_var) wynosi %f\n", f_var - l_var);
```

Zwróćmy uwagę, że odejmowanie w ostatnim wierszu przykładu, podobnie jak cała arytmetyka zmiennopozycyjna, jest przeprowadzane co najmniej z precyzją double. Typowy wynik uzyskany po uruchomieniu tego kodu jest następujący:

```
Błąd zaokrąglenia (f_var - l_var;) wynosi 3.000000
```

Dowolna wartość typu zmiennopozycyjnego może być dokładnie reprezentowana za pomocą innego typu zmiennopozycyjnego o większej precyzji. Dlatego, gdy wartość double jest przekształcana w typ long double lub gdy wartość float jest przekształcana w double lub long double, wartość jest dokładnie zachowana. Jednakże przy przekształceniach z typu o większej precyzji do typu o mniejszej precyzji przekształcana wartość może wykraczać poza zakres nowego typu. Jeśli wartość przekracza zakres typu docelowego, wynik

przekształcenia jest nieokreślony. Jeśli wartość mieści się w zakresie typu docelowego, lecz nie jest dokładnie reprezentowalna z precyzją typu docelowego, wówczas wynikiem jest następna mniejsza lub większa reprezentowalna wartość. Program w przykładzie 2-2 ilustruje błąd zaokrągleń spowodowany przez przekształcenie do typu zmiennopozycyjnego o mniejszej precyzji.

Gdy liczba zespolona jest przekształcana w rzeczywisty typ zmiennopozycyjny, wówczas część urojona jest po prostu odrzucana, zaś wynikiem jest część rzeczywista liczby zespolonej, która może być dalej przekształcona w typ docelowy, zgodnie z opisem w tym punkcie.

Przekształcenia do zespolonych typów zmiennopozycyjnych

Gdy liczba całkowita lub rzeczywista liczba zmiennopozycyjna jest przekształcana w typ zespolony, część rzeczywista wyniku jest uzyskiwana poprzez przekształcenie wartości do odpowiedniego rzeczywistego typu zmiennopozycyjnego, zgodnie z opisem w poprzednim punkcie. Część urojona wynosi zero.

Gdy liczba zespolona jest przekształcana do innego typu zespolonego, części rzeczywiste i urojone są przekształcane osobno, zgodnie z regułami dla rzeczywistych typów zmiennopozycyjnych.

```
#include <complex.h>        // Definiuje makra takie jak
                            // stała urojona I
double _Complex dz = 2;
float _Complex fz = dz + I;
```

Podczas pierwszej z powyższych inicjalizacji, zanim stała całkowita 2 będzie mogła być przypisana do zmiennej `dz`, jest niejawnie przekształcana w typ `double _Complex`. Do zmiennej `dz` zostanie przypisana wartość $2.0 + 0.0 \times \mathtt{I}$.

Podczas inicjalizacji zmiennej `fz`, dwie części wartości `dz` typu `double _Complex` są przekształcane (po operacji dodawania) w typ `float`, w wyniku czego część rzeczywista zmiennej `fz` jest równa `2.0F`, a część urojona `1.0F`.

Przekształcenia typów niearytmetycznych

Wskaźniki i nazwy tablic i funkcji także podlegają pewnym niejawnym i jawnym przekształceniom typów. Struktury i unie nie podlegają przekształceniom, chociaż wskaźniki do nich mogą być przekształcane z innych i do innych typów wskaźników.

Oznaczniki tablicy i funkcji

Oznacznik tablicy lub funkcji jest dowolnym wyrażeniem typu tablicowego lub funkcyjnego. W większości przypadków kompilator przekształca wyrażenie typu tablicowego, takie

jak nazwa tablicy, we wskaźnik do pierwszego elementu tablicy. Wyrażenie tablicowe *nie* jest przekształcane we wskaźnik tylko w następujących przypadkach:

- Gdy tablica jest operandem operatora `sizeof`,
- Gdy tablica jest operandem operatora adresu `&`,
- Gdy tablica typu `char`, `wchar_t`, `char16_t` lub `char32_t` jest inicjalizowana literałem łańcuchowym.

Poniższy przykład demonstruje niejawne przekształcenia oznaczników tablicy we wskaźniki i wykorzystuje specyfikator `%p` do wypisania wartości wskaźnika:

```
#include <stdio.h>

int *iPtr = 0;                      // Wskaźnik do typu int,
                                    // zainicjalizowany wartością 0.
int iArray[] = { 0, 10, 20 };       // Zainicjalizowana tablica
                                    // elementów typu int.

int array_length = sizeof(iArray) / sizeof(int);   // Liczba elementów:
                                                   // w tym przypadku 3

printf("Tablica rozpoczyna się od adresu %p.\n", iArray);
*iArray = 5;                        // Odpowiednik iArray[0] = 5;
iPtr = iArray + array_length - 1;   // wskazuje ostatni element
                                    // tablicy iArray:
                                    // odpowiednik zapisu
                                    // iPtr = &iArray[array_length-1];
printf("Ostatnim elementem tablicy jest %d.\n", *iPtr);
```

Podczas inicjalizacji zmiennej `array_length` w tym przykładzie wyrażenie `sizeof(iArray)` zwraca rozmiar całej tablicy, a nie rozmiar wskaźnika. Ten sam identyfikator `iArray` jest niejawnie przekształcany we wskaźnik w pozostałych trzech instrukcjach, w których występuje:

- Jako argument w pierwszym wywołaniu funkcji `printf()`.
- Jako operand operatora dereferencji `*`.
- W operacjach arytmetycznych na wskaźniku i przypisaniu do zmiennej `iPtr` (patrz też punkt „Modyfikowanie i porównywanie wskaźników” w rozdziale 9.).

Nazwy tablic znakowych są wykorzystywane jako wskaźniki w operacjach na łańcuchach, jak w poniższym przykładzie:

```
#include <stdio.h>
#include <string.h>     // Deklaruje size_t strlen( const char *s )
```

```
char msg[80] = "Jestem literałem łańcuchowym."; // Inicjalizuje
                                                // tablicę char.
printf("Łańcuch ma długość %d znaków.\n", strlen(msg));
                                                // Odpowiedź: 29.
printf("Tablica o nazwie msg ma długość %d bajtów.\n", sizeof(msg));
                                                // Odpowiedź: 80.
```

W wywołaniu funkcji `strlen(msg)` w tym przykładzie, identyfikator tabeli `msg` jest niejawnie przekształcany we wskaźnik do pierwszego elementu tablicy. Typ tego wskaźnika jest zgodny z typem parametru funkcji `const char *`. Wewnętrznie, funkcja `strlen()` po prostu wyznacza liczbę znaków począwszy od tego adresu, aż do pierwszego znaku null, który kończy łańcuch.

Podobnie, dowolne wyrażenie oznaczające funkcję, takie jak nazwa funkcji, może być niejawnie przekształcone we wskaźnik do funkcji. Ponownie, to przekształcenie nie jest wykonywane, jeśli wyrażenie jest operandem operatora adresu `&`. Operator `sizeof` nie może być używany z operandem typu funkcyjnego. Następujący przykład obrazuje niejawne przekształcenie nazw funkcji we wskaźniki (program inicjalizuje tablicę wskaźników do funkcji, a następnie wywołuje funkcje w pętli).

```
#include <stdio.h>
void func0() { puts("To jest funkcja func0(). "); } //Dwie funkcje
void func1() { puts("To jest funkcja func1(). "); }
/* ... */
void (*funcTable[2])(void) = { func0, func1 }; // Tablica dwóch wskaźników
                                               // do funkcji zwracających
                                               // void.
for ( int i = 0; i < 2; ++i )                  // Używa licznika pętli jako
  funcTable[i]();                              // indeksu tablicy.
```

Jawne przekształcenia wskaźników

Aby przekształcić wskaźnik z jednego typu w inny, zwykle należy skorzystać z jawnego rzutowania. W pewnych przypadkach kompilator pozwala na przekształcenia niejawne. Te przypadki zostaną opisane w podrozdziale „Niejawne przekształcenia wskaźników" w dalszej części rozdziału. Wskaźniki można jawnie przekształcić w liczby całkowite i odwrotnie.

Wskaźniki do obiektów

Dowolny wskaźnik do obiektu – czyli wskaźnik do kompletnego lub niekompletnego typu obiektu – można niejawnie przekształcić we wskaźnik do obiektu innego typu. W swoim programie należy się upewnić, że wybór przekształconego wskaźnika ma sens.

Oto przykład:

```
float  f_var = 1.5F;
long *l_ptr = (long *)&f_var;     // Inicjalizacja wskaźnika do
                                  // typu long za pomocą adresu
                                  // zmiennej f_var.
double *d_ptr = (double *)l_ptr;  // Inicjalizacja wskaźnika do
                                  // typu double za pomocą tego
                                  // samego adresu.

// W systemie, w którym sizeof(float) jest równy sizeof(long):

printf( "%zu bajtów reprezentujących %f, w kodzie szesnastkowym: 0x%lX\n",
  sizeof(f_var), f_var, *l_ptr );

// Użycie przekształconego wskaźnika w przypisaniu może nam
// przysporzyć kłopotów:

/*  *d_ptr = 2.5;  */   // Nie próbuj tego! W lokalizacji f_var
                        // nie ma miejsca na wartość double!
*(float *)d_ptr = 2.5;  // OK: przechowuje w tym miejscu
                        // wartość float.
```

Jeśli po przekształceniu wskaźnik do obiektu nie ma wyrównania wymaganego przez nowy typ, wynik użycia wskaźnika jest nieokreślony. We wszystkich pozostałych przypadkach przekształcenie wartości wskaźnika z powrotem do oryginalnego typu wskaźnika gwarantuje zwrócenie odpowiednika oryginalnego wskaźnika. Jeśli przekształcimy dowolny typ wskaźnika do obiektu we wskaźnik do dowolnego typu `char` (`char`, `signed char`, lub `unsigned char`), wynikiem będzie wskaźnik do pierwszego bajtu obiektu. Pierwszy bajt odnosi się w tym miejscu do bajtu o najniższym adresie, niezależnie od struktury porządku bajtów w systemie. Poniższy przykład wykorzystuje to do wypisania szesnastkowej reprezentacji struktury:

```
#include <stdio.h>
struct Data {
             short id;
             double val;
            };

struct Data myData = { 0x123, 77.7 };  // Inicjalizacja struktury.

unsigned char *cp = (unsigned char *)&myData; // Wskaźnik do
                                              // pierwszego
                                              // bajtu struktury.

printf( "%p: ", cp );                         // Wypisuje adres początkowy.
```

```
for ( int i = 0; i < sizeof(myData); ++i )    // Wypisuje każdy bajt
  printf( "%02X ", *(cp + i) );               // struktury w postaci
                                              // szesnastkowej.
putchar( '\n' );
```

Ten przykład wygeneruje następujący wynik:

```
0xbffffd70: 23 01 00 00 00 00 00 00 CD CC CC CC CC 6C 53 40
```

Dwa pierwsze bajty wyniku, czyli `23 01`, świadczą o tym, że kod został uruchomiony w systemie o kolejności bajtów w formie little-endian. Otóż bajt o najniższym adresie w strukturze `myData` był ostatnim bajtem znaczącym członka struktury `id` typu `short`.

Wskaźniki do funkcji

Typ funkcji zawsze zawiera zwracany przez nią typ i może także zawierać typy parametrów. Wskaźnik do danej funkcji można zawsze jawnie przekształcić we wskaźnik do funkcji innego typu. W poniższym przykładzie, instrukcja `typedef` definiuje nazwę typu „funkcji przyjmującej jeden parametr typu `double` i zwracającej wartość typu `double`”:

```
#include <math.h>                 // Deklaruje sqrt() i pow().
typedef double (func_t)(double); // Definiuje typ o nazwie func_t.
func_t *pFunc = sqrt;             // Wskaźnik do func_t,
                                  // zainicjalizowany adresem sqrt().
double y = pFunc( 2.0 );          // Prawidłowe wywołanie funkcji z
                                  // wykorzystaniem wskaźnika.

printf( "Pierwiastek kwadratowy z 2 wynosi %f.\n", y );

pFunc = (func_t *)pow;            // Zmiana wartości wskaźnika na adres
                                  // funkcji pow().
/*  y = pFunc( 2.0 );  */         // Nie próbuj tego: pow() przyjmuje
                                  // dwa argumenty.
```

W tym przykładzie do wskaźnika do funkcji `pFunc` są przypisywane adresy funkcji innego typu. Jednakże, jeśli program używa wskaźnika do wywołania funkcji, której definicja nie pasuje dokładnie do typu wskaźnika funkcji, wówczas zachowanie programu będzie nieokreślone.

Niejawne przekształcenia wskaźników

Kompilator niejawnie przekształca pewne typy wskaźników. Przypisanie, wyrażenia warunkowe wykorzystujące operatory równości `==` i `!=` oraz wywołania funkcji wiążą się z niejawnym przekształceniem wskaźnika w trzech przypadkach, opisanych poniżej w osobnych punktach. Te trzy przypadki niejawnego przekształcania wskaźnika to:

- Dowolny typ wskaźnika do obiektu można niejawnie przekształcić we wskaźnik do `void` i odwrotnie.
- Dowolny wskaźnik do danego typu można niejawnie przekształcić we wskaźnik do tego samego typu, lecz oznaczonego większą liczbą kwalifikatorów typu.
- *Stała wskaźnika pustego* może być niejawnie przekształcona w dowolny typ wskaźnika.

Wskaźniki do typu void

Wskaźniki do typu `void` – czyli wskaźniki typu `void *` – są wykorzystywane jako wskaźniki „różnorodnego przeznaczenia", reprezentujące adres dowolnego obiektu, niezależnie od jego typu. Wskaźnik do typu `void` zwraca na przykład funkcja `malloc()` (patrz przykład 2-3). Przed uzyskaniem dostępu do bloku pamięci wskaźnik do typu `void` musi być zawsze przekształcony we wskaźnik do obiektu.

Przykład 4-1 demonstruje więcej przypadków użycia wskaźników do typu `void`. Program sortuje tablicę za pomocą funkcji standardowej `qsort()`, zadeklarowanej w pliku nagłówkowym *stdlib.h*. Jej prototyp jest następujący:

```
void qsort( void *array, size_t n, size_t element_size,
int (*compare)(const void *, const void *) );
```

Funkcja `qsort()` sortuje tablicę z wykorzystaniem algorytmu quick-sort w porządku rosnącym, począwszy od adresu *`array`*. Zakłada się, że tablica ma *`n`* elementów o rozmiarze *`element_size`*.

Czwarty parametr, czyli *`compare`*, jest wskaźnikiem do funkcji, wywoływanej przez `qsort()` w celu porównania dwóch elementów tablicy. Adresy tych dwóch porównywanych elementów są przekazywane do funkcji jako parametry wskaźnikowe. Zwykle funkcja dokonująca porównania musi być zdefiniowana przez programistę. Musi ona zwracać wartość, która jest mniejsza, równa lub większa od 0, informując w ten sposób, czy pierwszy element jest mniejszy, równy, czy większy od drugiego.

Przykład 4-1 *Funkcja porównująca przeznaczona dla qsort()*

```
#include <stdlib.h>
#define ARR_LEN 20
/*
 * Funkcja służąca do porównań dowolnych dwóch elementów typu float,
 * do wykorzystania jako wywołanie zwrotne w funkcji qsort().
 * Argumenty są przekazywane za pomocą wskaźnika.
 *
 * Zwraca: -1 jeśli pierwszy element jest mniejszy niż drugi;
 *          0 jeśli elementy są równe;
 *          1 jeśli pierwszy element jest większy niż drugi.
```

```
 */
int  floatcmp( const void* p1, const void* p2 )
{
  float x = *(float *)p1,
        y = *(float *)p2;
  return (x < y) ? -1 : ((x == y) ? 0 : 1);
}
/*
 * Funkcja main() sortuje tablicę typu float.
 */
int main()
{
  /* Dynamiczna alokacja miejsca dla tablicy: */
  float *pNumbers = malloc( ARR_LEN * sizeof(float) );
  /* ... Obsługa błędów, inicjalizacja elementów tablicy... */
  /* Sortowanie tablicy: */
  qsort( pNumbers, ARR_LEN, sizeof(float), floatcmp );
  /* ... Korzystanie z posortowanej tablicy ... */
  return 0;
}
```

W przykładzie 4-1 funkcja `malloc()` zwraca `void *`, który jest niejawnie przekształcany we `float *` podczas przypisania do `pNumbers`. Pierwszy argument w wywołaniu funkcji `qsort()`, czyli `pNumbers`, jest niejawnie przekształcany z typu `float *` do typu `void *`, a nazwa funkcji `floatcmp` jest niejawnie interpretowana jako wskaźnik do funkcji. Na końcu, funkcja `qsort()` wywołuje funkcję `floatcmp()`, przekazując do niej argumenty typu `void *`, czyli „uniwersalnego" typu wskaźnikowego. Argumenty te są następnie jawnie przekształcane do typu `float *`, przed ich dereferencją w celu inicjalizacji zmiennych `float`.

Wskaźniki do typów obiektów z kwalifikatorami

W języku C dostępne są następujące kwalifikatory typu: `const`, `volatile` i `restrict` (więcej informacji na temat tych kwalifikatorów można z należć w rozdziale 11.). Na przykład, jeśli to konieczne, kompilator niejawnie przekształca dowolny wskaźnik do typu `int` we wskaźnik typu `const int`. Jeśli nie chcemy dodawać kwalifikatora, ale raczej go usunąć, musimy skorzystać z jawnego przekształcania typu, jak w poniższym przykładzie:

```
int n = 77;

const int *ciPtr = 0;  // Wskaźnik do const int.
                       // Sam wskaźnik nie jest stałą!

ciPtr = &n;            // Niejawnie przekształca adres do typu
                       // const int *.
```

```
n = *ciPtr + 3;        // OK: taki sam efekt jak n = n + 3;

*ciPtr *= 2;           // Błąd: nie można zmienić obiektu, do którego
                       // odwołuje się wskaźnik do typu const int.

*(int *)ciPtr *= 2;    // OK: Jawne przekształcenie wskaźnika we
                       // wskaźnik do typu int nie będącego stałą.
```

Przedostania instrukcja w tym przykładzie obrazuje, dlaczego wskaźniki do typów ze specyfikatorem `const` są czasem nazywane *wskaźnikami tylko do odczytu*: chociaż można modyfikować wartości wskaźników, to nie można ich wykorzystywać do modyfikacji obiektów, które wskazują.

Stałe wskaźnika pustego

Stała wskaźnika pustego (*null pointer*) jest stałą całkowitą o wartości 0 lub stałą wartością całkowitą 0 rzutowaną jako wskaźnik do typu `void`. Makro `NULL` jest zdefiniowane w plikach nagłówkowych *stdlib.h*, *stdio.h* i innych jako wskaźnik pusty. Następujący przykład obrazuje, jak wykorzystać makro `NULL` jako stałej wskaźnikowej do zainicjalizowania wskaźników, a nie liczby całkowitej zero, czy znaku null:

```
#include <stdlib.h>
long *lPtr = NULL;      // Inicjalizacja wartością NULL: wskaźnik
                        // nie jest gotowy do użycia.

/* ... przeprowadzane tu operacje mogą przypisać adres obiektu do lPtr...
*/
if ( lPtr != NULL )
{
  /* ... użyj lPtr tylko jeśli jego wartość została zmieniona i jest różna
od NULL ... */
}
```

Gdy przekształcamy stałą wskaźnika pustego w inny typ wskaźnika, wynik jest nazywany *wskaźnikiem pustym*. Wzorzec bitowy wskaźnika pustego nie zawsze jest zerem. Jednakże, gdy przyrównamy wskaźnik pusty do zera, do wartości `NULL` lub do innego wskaźnika pustego, wynikiem będzie zawsze `true`. Natomiast odwrotnie, porównanie wskaźnika pustego do dowolnego prawidłowego wskaźnika do obiektu lub funkcji zawsze zwróci `false`.

Przekształcenia między wskaźnikiem i typami całkowitymi

Wskaźnik można zawsze jawnie przekształcić w typ całkowity i odwrotnie. Wynik takiego przekształcenia zależy od kompilatora i powinien być spójny ze strukturą adresowania w systemie, na którym jest uruchomiony program. Przekształcenia między wskaźnikiem a typami całkowitymi mogą być przydatne w programowaniu systemów i są niezbędne,

gdy programy muszą mieć dostęp do określonych adresów fizycznych, jak ROM czy rejestry typu MMIO (ang. *„memory-mapped I/O”*).

Gdy przekształcamy wskaźnik w typ całkowity, którego zakres jest niewystarczający do reprezentowania wartości wskaźnika, wynik operacji jest nieokreślony. Natomiast odwrotnie, przekształcenie typu całkowitego w typ wskaźnikowy nie zawsze zwraca prawidłowy wskaźnik. Plik *stdint.h* może opcjonalnie definiować typy całkowite `intptr_t` (ze znakiem) i `uintptr_t` (bez znaku). Do jednego z tych typów można przekształcić dowolny prawidłowy wskaźnik, zaś przy dalszym przekształceniu z powrotem do wskaźnika gwarantowane jest uzyskanie oryginalnego wskaźnika. Dlatego jeśli plik *stdint.h* definiuje te typy, należy ich używać zawsze, gdy tylko zamierzamy dokonywać przekształceń między wskaźnikami i typami całkowitymi.

Oto kilka przykładów:

```
float x = 1.5F, *fPtr = &x; // Zmienna float, i wskaźnik do niej.

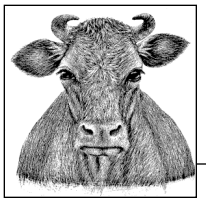
// Zapisanie wartości wskaźnika jako typu całkowitego:
unsigned long long adr_val = (unsigned long long)fPtr;

// Lub, jeśli dołączony jest plik stdint.h i jeśli zdefiniowany
// jest typ  uintptr_t:
uintptr_t adr_val = (uintptr_t)fPtr;

/*
 * Na komputerach PC z procesorem Intel x86 PC z systemem DOS,
 * blok danych BIOS-u rozpoczyna się od adresu 0x0040:0000.
 * Pierwsze dwubajtowe słowo w tym miejscu zawiera adres I/O
 * portu szeregowego COM1.
 * (Kompilacja z wykorzystaniem modelu pamięci typu "large"
 * systemu DOS.)
 */

unsigned short *biosPtr = (unsigned short *)0x400000L;
unsigned short com1_io = *biosPtr;  // Pierwsze słowo zawiera
                                    // adres I/O portu COM1.
printf( "COM1 ma adres ... I/O %Xh.\n", com1_io );
```

Ostatnie trzy instrukcje pozwalają uzyskać informacje o konfiguracji sprzętowej z tablicy danych systemowych, zakładając, że środowisko operacyjne umożliwia programom dostęp do tego obszaru w pamięci. W programie dla systemu DOS, skompilowanym z wykorzystaniem modelu „large” wskaźniki są 32-bitowe i przechowują adres segmentu w starszych 16 bitach oraz przesunięcie w młodszych 16 bitach (zwykle zapisywane jest to w postaci *`segment:przesunięcie`*). Dlatego wskaźnik `biosPtr` w powyższym przykładzie można zainicjalizować za pomocą stałej typu `long int`.



5
Wyrażenia i operatory

Wyrażenie składa się z sekwencji stałych, identyfikatorów i operatorów, które program przetwarza wykonując wskazane operacje. Celem wyrażenia w programie może być uzyskanie wynikowej *wartości* lub uzyskanie *efektów ubocznych* operacji lub obydwu (patrz punkt „Efekty uboczne i punkty sekwencji" w dalszej części tego rozdziału).

Wyrażeniem jest również pojedyncza stała, literał tekstowy lub identyfikator obiektu czy funkcji. Takie proste wyrażenie lub, ujęte w nawiasy, bardziej złożone wyrażenie jest nazywane *wyrażeniem pierwotnym*. Standard C11 wprowadził kolejny rodzaj wyrażeń pierwotnych zwanych *selekcją generyczną*, opisaną w kolejnym podrozdziale.

Każde wyrażenie ma swój typ. Typ wyrażenia odpowiada typowi wartości zwracanej w wyniku przetworzenia wyrażenia. Wyrażenie, które nie zwraca żadnej wartości, jest typu `void`. W tabeli 5-1 wypisano kilka prostych przykładów wyrażeń (zakładamy, że zmienna `a` została zadeklarowana jako typ `int`, a `z` jest zmienną typu `float _Complex`).

Tabela 5-1 *Wyrażenia przykładowe*

Wyrażenie	Typ
`'\n'`	`int`
`a + 1`	`int`
`a + 1.0`	`double`
`a < 77.7`	`int`
`"Literał łańcuchowy."`	`char *`
`abort()`	`void`
`sqrt(2.0)`	`double`

Tabela 5-1 *Wyrażenia przykładowe*

Wyrażenie	Typ
`z / sqrt(2.0)`	`double _Complex`

Jak widać w ostatnim przykładzie z tabeli 5-1, wyrażenia złożone są tworzone za pomocą operatora, którego operandami są wyrażenia. Same operandy mogą być wyrażeniami pierwotnymi lub złożonymi. Na przykład mnożnikiem w mnożeniu może być wywołanie funkcji. Podobnie, argumenty w wywołaniu funkcji mogą mieć postać wyrażenia złożonego z kilku operatorów, jak w tym przykładzie:

```
2.0 * sin( 3.14159 * fAngleDegrees/180.0 )
```

Jak są przetwarzane wyrażenia

Zanim zaczniemy szczegółowo rozważać poszczególne operatory, w tym podrozdziale wyjaśnimy kilka podstawowych zasad, które pomogą nam zrozumieć, jak są przetwarzane wyrażenia w języku C.

Z pewnością duże znaczenie w przetwarzaniu wyrażeń złożonych mają *priorytet* i *łączność* operatorów, lecz nie mniej ważna w zrozumieniu działania programu C jest *selekcja generyczna*, *L-wartości* i *punkty sekwencji*.

Selekcje generyczne (C11)

Selekcja generyczna jest wyrażeniem pierwotnym, które wybiera wyrażenie z listy w zależności od typu innego wyrażenia. Selekcja ma miejsce podczas kompilacji. Ten mechanizm pozwala programistom C na pisanie makr generycznych, takich jak te dostępne w nagłówku *tgmath.h*, które obsługują funkcje matematyczne i zostały wprowadzone w wersji C99 standardu. Plik *tgmath.h* udostępnia na przykład sześć różnych funkcji obliczających pierwiastek kwadratowy, trzy dla typów rzeczywistych `float`, `double` i `long double` oraz trzy dla odpowiedników typów złożonych. W programie dołączającym nagłówek *tgmath.h* można użyć makra generycznego `sqrt(`*`x`*`)` do automatycznego wywołania tej funkcji, która pasuje do typu *x*.

Zapis selekcji generycznej rozpoczyna się od nowego słowa kluczowego `_Generic`, po którym następują nawiasy zawierające wyrażenie kontrolne i listę asocjacji generycznych:

```
_Generic( wyrażenie, asocjacja generyczna 1
[, asocjacja generyczna 2, ...] )
```

Asocjacja generyczna ma postać

```
nazwa typu : wyrażenie
```

lub

```
default : wyrażenie
```

Asocjacja `default` jest opcjonalna i może się pojawić na liście tylko raz. Nazwy typów muszą przedstawiać odrębne, wzajemnie niekompatybilne typy. Typy niekompletne i typy tablic o zmiennej długości są niedozwolone.

Wyrażeniem kontrolnym *`wyrażenie`* nie jest przetwarzanie, lecz jego typ jest porównywany z nazwami typów na liście asocjacji. Jeśli typ wyrażenia kontrolnego jest zgodny z jedną z nazw typów, wówczas kompilator wybiera wyrażenie powiązane z tym typem na liście. Jeśli na liście nie ma nazwy zgodnego z typu, wówczas wybierane jest wyrażenie z asocjacji `default`. Jeśli lista nie zawiera typu zgodnego, ani asocjacji domyślnej, wówczas kompilator wygeneruje komunikat błędu.

Typ i wartość selekcji generycznej są zgodne z wyrażeniem wynikowym, a podczas wykonywania programu przetwarzane jest tylko wyrażenie wynikowe. Poniżej znajduje się prosty przykład:

```
_Generic( 1.0, int: "int", double: "double",
               default: "ani int, ani double")
```

Wynikiem tej selekcji jest literał tekstowy `"double"`, ponieważ `1.0` jest typu `double`. Selekcje generyczne są używane przede wszystkim do definiowania makr generycznych, jak w poniższym przykładzie:

```
#define typeOf(x) _Generic((x), int: "int", double: "double", \
                                default: "ani int, ani double")
```

Po tej definicji, wywołanie makra `typeOf('A')` zwróci `"int"`, ponieważ stała znakowa w języku C jest typu `int`. Jednakże wywołanie `typeOf(var)` zwróci napis `"ani int, ani double"`, jeśli zmienna `var` jest typu `unsigned int` lub `const double`, ponieważ te typy nie są zgodne z żadnym typem na liście selekcji generycznej, zawierającej nazwy typów `int` i `double`. Inny, bardziej przydatny przykład makra generycznego, napisanego z wykorzystaniem selekcji generycznej, jest przedstawiony w rozdziale 15. „Dyrektywy preprocesora".

L-wartości

L-wartość jest wyrażeniem odnoszącym się do obiektu. Najprostszym przykładem jest nazwa zmiennej. Pierwsza litera „L" początkowo oznaczała „left" (ang. *z lewej*). Ponieważ L-wartość odnosi się do obiektu, może się znajdować z lewej strony operatora przypisania, jak w *`wyrażeniezlewej = wyrażeniezprawej`*[3]. Inne wyrażenia – te które reprezentują

3 Standard C uznaje tę etymologię, lecz proponuje, aby litera L w L-wartości dotyczyła „lokalizatora", ponieważ L-wartość zawsze odnosi się do lokalizacji w pamięci. Standard unika używania terminu R-wartość, preferując określenie „wartość nie będąca L-wartością".

wartość, lecz nie odnoszą się do obiektu – są zwane przez analogię R-wartościami. *R-wartość* jest wyrażeniem, które może się znajdować z prawej (ang. *right*) strony operatora przypisania, lecz nie z lewej strony. Przykładem mogą być stałe i wyrażenia arytmetyczne.

L-wartość może być zawsze przełożona jako adres odpowiadającego obiektu, chyba że obiekt jest polem bitowym lub zmienną zadeklarowaną za pomocą klasy przypisania *register* (patrz punkt „Specyfikatory klasy przydziału" w rozdziale 11.). Do operatorów zwracających L-wartość należy operator indeksowania `[]` oraz operator adresowania pośredniego `*`, co obrazują przykłady w tabeli 5-2 (zakładamy, że zmienna `array` jest zadeklarowana jako tablica, a `ptr` jako zmienna wskaźnikowa).

Tabela 5-2 *Wyrażenia wskaźnikowe i tablicowe mogą być L-wartościami*

Wyrażenie	L-wartość?
`array[1]`	Tak; element tablicowy jest obiektem o określonej lokalizacji.
`&array[1]`	Nie; pozycja obiektu nie jest obiektem o określonej lokalizacji.
`ptr`	Tak; zmienna wskaźnikowa jest obiektem o określonej lokalizacji.
`*ptr`	Tak; wskaźnik wskazuje także obiekt o określonej lokalizacji.
`ptr+1`	Nie; dodawanie zwraca wartość nowego adresu, lecz nie obiekt.
`*ptr+1`	Nie; dodawanie zwraca nową wartość arytmetyczną, lecz nie obiekt.

Obiekt może być zadeklarowany jako stała. W tym przypadku nie można go umieścić z lewej strony przypisania, nawet jeśli jest to L-wartość, co obrazuje poniższy przykład:

```
int a = 1;
const int b = 2, *ptr = &a;
b = 20;        // Błąd: zmienna b jest zadeklarowana jako const int.
*ptr = 10;     // Błąd: zmienna ptr jest zadeklarowana jako wskaźnik
               // do const int.
```

W tym przykładzie wszystkie wyrażenia `a`, `b`, `ptr` i `*ptr` są L-wartościami.

Jednakże `b` i `*ptr` są stałymi L-wartościami. Ponieważ zmienna `ptr` jest zadeklarowana jako wskaźnik do `const int`, nie można jej użyć do zmodyfikowania wskazywanego przez nią obiektu. Pełne rozważania na temat deklaracji można znaleźć w rozdziale 11.

Operand z lewej strony przypisania, podobnie jak dowolny operand operacji inkrementacji i dekrementacji `++` i `--`, nie tylko muszą być L-wartością lecz także *modyfikowalną L-wartością*. Modyfikowalna L-wartość jest L-wartością, która nie jest zadeklarowana z kwalifikatorem `const` (patrz punkt „Kwalifikatory typu" w rozdziale 11.) i nie jest typu tablicowego. Jeśli modyfikowalna L-wartość odnosi się do obiektu będącego strukturą lub unią, żaden z jej elementów nie może być zadeklarowany ani bezpośrednio, ani pośrednio, jako typ `const`.

Efekty uboczne i punkty sekwencji

Przetwarzanie wyrażenia może skutkować nie tylko zwróceniem wartości, ale także innymi zmianami w środowisku wykonywalnym, zwanymi *efektami ubocznymi*. Przykłady takich zmian obejmują modyfikacje wartości zmiennej lub strumieni wejścia lub wyjścia.

Podczas wykonywania programu istnieje kilka określonych punktów, w których pojawiły się już wszystkie efekty uboczne jednego wyrażenia, ale nie zaszły jeszcze żadne efekty następnego wyrażenia. Takie punkty w programie są zwane *punktami sekwencji*. Pomiędzy dwoma kolejnymi punktami sekwencji może dochodzić do częściowego przetwarzania wyrażeń w dowolnej kolejności. Dlatego programista musi pamiętać, aby pomiędzy dwoma kolejnymi punktami sekwencji nie modyfikować żadnego obiektu więcej niż jeden raz. Oto przykład:

```
int i = 1;          // OK.
i = i++;            // Źle: dwie modyfikacje i; zachowanie jest
                    // nieokreślone.
```

Ponieważ operacje przypisania i inkrementacji w ostatniej instrukcji mogą zajść w dowolnej kolejności, wartość `i` jest nieokreślona. Podobnie, w wyrażeniu `f()+g()`, gdzie `f()` i `g()` są dwiema funkcjami, C nie określa, które wywołanie funkcji zostanie wykonane jako pierwsze. To programista musi się upewnić, że wyniki takich wyrażeń nie są zależne od kolejności wywołania. Inny przykład:

```
int i = 0, array[ ] = { 0, 10, 20 };
// ...
array[i] = array[++i];          // źle: zachowanie nieokreślone.
array[i] = array[i + 1]; ++i;   // OK: modyfikacje oddzielone
                                // punktem sekwencji.
```

Najważniejsze punkty sekwencji występują:

- Po przetworzeniu wszystkich argumentów w wywołaniu funkcji i przed przekazaniem kontroli do instrukcji wewnątrz funkcji.
- Na końcu wyrażenia, które nie wchodzi w skład większego wyrażenia. Do takich *pełnych wyrażeń* należą wyrażenia w instrukcjach wyrażeniowych (patrz podrozdział „Instrukcje wyrażeniowe" w rozdziale 6.), wszystkie trzy wyrażenia kontrolne w instrukcji `for`, warunki w instrukcjach `if` lub `while`, wyrażenia w instrukcjach `return` i inicjalizatory.
- Po przetworzeniu pierwszego operandu każdego z następujących operatorów:
  - `&&` (logiczne AND)
  - `||` (logiczne OR)
  - `?:` (operator warunkowy)
  - `,` (operator przecinkowy)

Dlatego wyrażenie `++i < 100 ? f(i++) : (i = 0)` jest dozwolone, gdyż między pierwszą modyfikacją `i` i jedną z kolejnych modyfikacji istnieje punkt sekwencji.

Priorytet i łączność operatora

Wyrażenie może zawierać kilka operatorów. W tym przypadku to *priorytet* operatorów określa, która część wyrażenia jest traktowana jako operand każdego operatora. Na przykład, zgodnie ze zwyczajowymi regułami arytmetycznymi, operatory `*`, `/` i `%` mają wyższy priorytet w wyrażeniu, niż operatory `+` i `-`. Na przykład następujące wyrażenie:

```
a – b * c
```

jest odpowiednikiem `a – (b * c)`. Jeśli zamierzamy inaczej pogrupować operandy, musimy użyć nawiasów:

```
(a – b) * c
```

Jeśli dwa operatory w wyrażeniu mają ten sam priorytet, to ich *łączność* określa, czy zostaną zgrupowane z operandami w kolejności od lewej do prawej, czy od prawej do lewej. Na przykład operatory arytmetyczne są łączone z operandami od lewej do prawej, zaś z operatorami przypisania od prawej do lewej, jak pokazano w tabeli 5-3. Tabela 5-4 zawiera listę priorytetów i łączności wszystkich operatorów języka C, w kolejności priorytetu.

Tabela 5-3 *Grupowanie operatorów*

Wyrażenie	Łączność	Grupowanie
`a / b % c`	Od lewej do prawej	`(a / b) % c`
`a = b = c`	Od prawej do lewej	`a = (b = c)`

Tabela 5-4 *Priorytet i łączność operatorów*

Priorytet	Operatory	Łączność

1.	Operatory przyrostkowe:
	`[] () . -> ++ --`
	`(`*`nazwa typu`*`){`*`lista`*`}`
		Od lewej do prawej

2.	Operatory jednoargumentowe:
	`++ --`
	`! ~ + – * &`
	`sizeof _Alignof`
		Od prawej do lewej

3.	Operator rzutowania: `(`*`nazwa typu`*`)`	Od prawej do lewej
4.	Operatory multiplikatywne: `* / %`	Od lewej do prawej
5.	Operatory dodawania: `+ -`	Od lewej do prawej