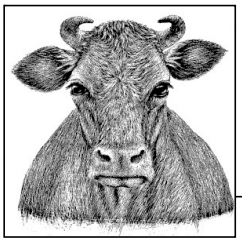


Wstęp

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.

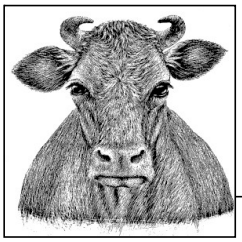
Peter

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.

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()*

```
// circulararea.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).

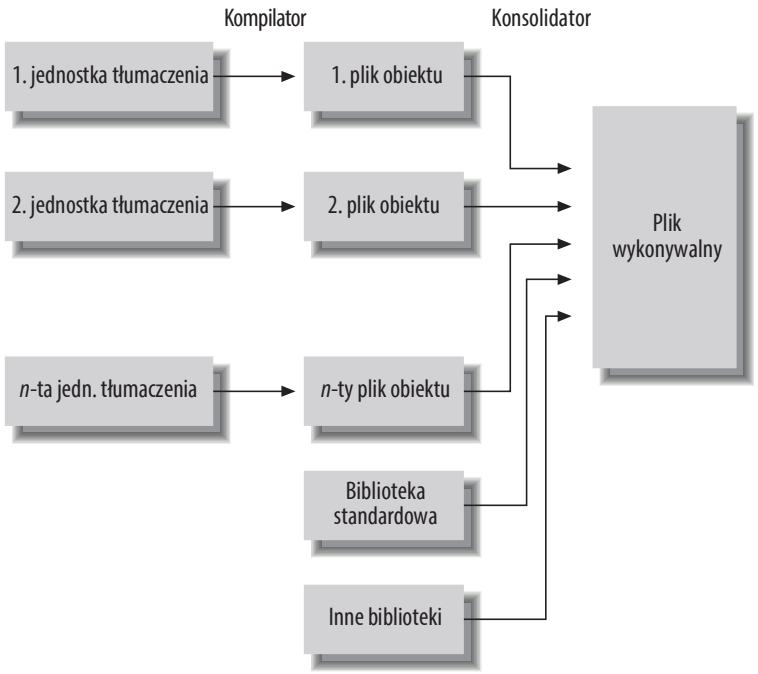
Pliki źródłowe są zwane także *modułami*. Biblioteka, na przykład standardowa biblioteka C, zawiera skompilowane, łatwo dostępne moduły zawierające funkcje standardowe.

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

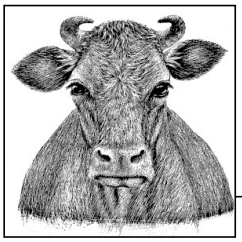
tokena w nieprawidłowy. Ta zasada rozwiązuje wszystkie wątpliwości w następującym przykładzie:

```
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.