Stephen G. Kochan

Język C

Kompendium wiedzy

Wydanie IV

Kompletny przewodnik po języku C!

Podziękowania

Za przygotowanie kolejnych wersji tej książki chciałbym podziękować następującym osobom: Douglasowi McCormickowi, Jimowi Scharfowi, Henry'emu Tabickmanowi, Dickowi Fritzowi, Steve'owi Levy'emu, Tony'emu Ianinno i Kenowi Brownowi. Chcę podziękować też Henry'emu Mullishowi z Uniwersytetu Nowojorskiego za wszystko, czego mnie nauczył o pisaniu, i za wprowadzenie w świat wydawniczy.

Pragnę szczególnie podziękować kilku osobom z wydawnictwa Pearson: Markowi Taberowi, redaktorce prowadzącej Mandie Frank, redaktorce Charlotte Kughen oraz korektorowi merytorycznemu Siddharcie Singhowi. Wreszcie chciałbym podziękować pozostałym pracownikom wydawnictwa Pearson zaangażowanym w ten projekt, nawet jeśli nie współpracowałem z nimi bezpośrednio.

Spis treści

O autorze

Stephen G. Kochan programuje w C już od ponad 30 lat. Jest autorem i współautorem szeregu najlepszych książek o tym języku, takich jak *Programowanie w C*, *Objective-C. Vademecum profesjonalisty* oraz *Zagadnienia z programowania w C*, a także kilku książek o systemie Unix, między innymi: *W głąb systemu Unix* i *Programowanie w powłoce Unixa*.

Autor towarzyszący w czwartym wydaniu

Dean Miller jest pisarzem i redaktorem z ponad 20-letnim doświadczeniem w branży wydawniczej i produktów licencjonowanych. Jest współautorem najnowszych wydań książek *Sams Teach Yourself C in One Hour a Day* i *Sams Teach Yourself Beginning Programming in 24 Hours*.

Wprowadzenie

Język programowania C został stworzony przez Dennisa Ritchiego w laboratoriach AT&T Bell na początku lat 70. ubiegłego wieku. Jednak dopiero pod koniec lat 70. został spopularyzowany i uzyskał powszechne uznanie. Takie opóźnienie wiązało się gównie z brakiem komercyjnych kompilatorów, jedyne były dostępne w laboratoriach Bella. Początkowo wzrost popularności języka C wiązał się z równie szybkim, jeśli nie szybszym, zdobywaniem rynku przez system operacyjny Unix. W tym systemie, także stworzonym w laboratoriach Bell, C stał się „standardowym" językiem programowania. Ponad 90% tego systemu zostało napisane w C!

Niesamowity sukces komputerów IBM PC i ich klonów szybko sprawił, że system MS-DOS stał się najpopularniejszym środowiskiem programowania w języku C. W miarę jak C był popularyzowany w różnych systemach, coraz więcej producentów chciało mieć udział w tym sukcesie i promowało własne kompilatory C. Większość tak tworzonych kompilatorów była oparta na dodatku do pierwszej książki o C, *Język programowania C*, autorstwa Briana Kernighana i Dennisa Ritchiego. Niestety, dodatek ten nie zawierał pełnej i jednoznacznej definicji C, wobec czego poszczególni producenci musieli sami zinterpretować brakujące zagadnienia.

Na początku lat 80. uznano, że język C należy poddać standaryzacji. Amerykański Narodowy Komitet Standaryzacyjny (ANSI — *American National Standards Institute*) jest podmiotem, który zajmuje się tego typu zagadnieniami; tam więc w 1983 roku powstała komisja ANSI C (określana jako X3J11), której zadaniem było stworzenie standardu języka C. W 1989 roku wyniki prac komisji zostały zatwierdzone i w 1990 roku opublikowana została pierwsza oficjalna wersja standardu ANSI C.

Ponieważ język C jest stosowany na całym świecie, w sprawę zaangażowała się też Międzynarodowa Organizacja Standaryzacyjna ISO (*International Standard Organization*). Przyjęła ona stworzony standard jako ISO/IEC 9899:1990. Od tego czasu w języku C wprowadzano zmiany. Najnowszy standard, znany jako ANSI C11 i ISO/IEC 9899: 2011, został opublikowany w 2011 roku. Na tej wersji języka bazuje niniejsza książka.

C jest „językiem wyższego poziomu", który jednak umożliwia użytkownikom pozostanie w bliskim kontakcie ze sprzętem i programowanie na niskim poziomie.

Dzieje się tak dlatego, że język C — mimo iż jest strukturalnym językiem programowania — swój rodowód wyprowadza z programowania systemów operacyjnych, gdzie wymagana jest niesłychana siła i elastyczność.

Ta książka ma nauczyć czytelnika programowania w C. Nie zakładamy w niej żadnej wcześniejszej znajomości C; ma to być pomoc tak dla początkujących, jak i doświadczonych programistów. Osoby mające doświadczenie w programowaniu w innych językach zapewne będą zdziwione, gdyż C jest językiem innym niż wszystkie.

W niniejszej książce opisujemy wszystkie elementy języka C. Omówieniu każdego nowego elementu towarzyszy niewielki, ale *kompletny* przykładowy program pokazujący zastosowanie danej cechy. Stanowi to efekt metody, która jest podstawą tego tekstu, czyli nauczania przez przykłady. Tak jak jeden obraz jest wart tysiąca słów, tak dobrze dobrany program jest wart tysiąca słów opisu. Osoby mające dostęp do komputera z zainstalowanym kompilatorem C zdecydowanie powinny uruchomić każdy prezentowany przykład i porównać uzyskane wyniki z tymi, które zostały opisane w książce. Dzięki temu nie tylko łatwiej nauczą się języka i jego składni, lecz także niejako wejdzie im w krew cały proces wpisywania, kompilowania i uruchamiania programów w C.

W książce wielokrotnie podkreślamy, jak ważna jest czytelność programów. Autor jest zdecydowanym zwolennikiem poglądu, że programy należy tak pisać, aby łatwo było je potem czytać; dotyczy to tak programisty, jak kogokolwiek innego. Po nabyciu pewnego doświadczenia i przy korzystaniu ze zdrowego rozsądku czytelnik stwierdzi, że takie programy niemal zawsze łatwiej pisać, uruchamiać i modyfikować. Co więcej, pisanie czytelnych programów jest naturalnym wynikiem przestrzegania paradygmatu programowania strukturalnego.

Ta książka ma być podręcznikiem, tak więc materiał omawiany w każdym rozdziale opiera się na materiale omawianym wcześniej. Wobec tego najlepiej czytać wszystkie rozdziały po kolei, zdecydowanie nie warto ograniczać się do kartkowania. Ważne jest też, aby przed przejściem do każdego następnego rozdziału przerobić ćwiczenia zamieszczone na końcu rozdziału poprzedniego.

Rozdział 1., omawiający najważniejsze pojęcia dotyczące języków programowania wysokiego poziomu oraz proces kompilacji, został dołączony po to, aby czytelnik na pewno zrozumiał terminologię używaną w dalszej części książki. Od rozdziału 3. będziemy stopniowo wprowadzać czytelnika w arkana języka C. Przed dojściem do rozdziału 15. omówione zostaną wszystkie najważniejsze cechy języka. Rozdział 15. jest poświęcony operacjom wejścia-wyjścia w C, z kolei rozdział 16. to omówienie bardziej zaawansowanych i trudniejszych cech języka C.

W rozdziale 17. pokazano, jak można użyć preprocesora C, aby ułatwić uruchamianie programów. Tam omawiany jest też proces interaktywnego uruchamiania programu i usuwania zeń błędów. Do zilustrowania tego zagadnienia wybrano popularny program uruchomieniowy (*debugger*) — `gdb`.

W ciągu ostatniej dekady w światku programistów wszyscy mówią o programowaniu obiektowym (także OOP — *object-oriented programming*). C nie jest językiem obiektowym.

Jednak istnieje kilka języków obiektowych opartych na C. W rozdziale 18. omawiamy programowanie obiektowe i związane z nim pojęcia. Krótko opisane są też trzy najważniejsze języki obiektowe wywodzące się od C: C++, C# oraz Objective-C.

Dodatek A zawiera podsumowanie wszystkich elementów języka i służy jako materiał referencyjny.

Dodatek B, poświęcony bibliotece standardowej języka C, zawiera opis wielu funkcji tej biblioteki, dostępnych we wszystkich systemach, w których działa język C.

Dodatek C zawiera opis wielu opcji używanych podczas korzystania z kompilatora GNU C — `gcc`.

W dodatku D opisano typowe błędy programistyczne.

W końcu w dodatku E zestawiono tytuły książek i adresy stron internetowych, w których można znaleźć dalsze informacje o języku C.

W książce tej nie zakładamy używania żadnego konkretnego komputera czy systemu operacyjnego. Krótko omawiamy kompilowanie i uruchamianie programów przy użyciu popularnego kompilatora GNU C — `gcc`.

1

Podstawy

W tym rozdziale wprowadzamy pewne podstawowe pojęcia, które trzeba zrozumieć, aby nauczyć się programowania w C. Omawiamy ogólnie programowanie w językach wysokiego poziomu, a także kompilowanie programu napisanego w tych językach.

Programowanie

Komputery to naprawdę bezmyślne maszyny, gdyż robią tylko to, co im się każe. Większość systemów — działając — naprawdę realizuje tylko elementarne zadania. Większość komputerów na przykład potrafi dodać dwie liczby lub sprawdzić, czy liczba jest zerem — ale ponadto niewiele. Podstawowe instrukcje wykonywane przez system komputerowy tworzą to, co nazywamy *zbiorem instrukcji* danego komputera.

Aby za pomocą komputera rozwiązać jakiś problem, trzeba rozwiązanie zapisać w formie instrukcji dla tegoż komputera. *Program* komputerowy to właśnie zbiór instrukcji niezbędnych do jego rozwiązania. Metoda użyta do rozwiązania problemu to *algorytm*. Jeśli na przykład potrzebny jest program sprawdzający, czy liczba jest parzysta czy nieparzysta, zbiór instrukcji rozwiązujący tak postawiony problem tworzy tenże program. Metoda użyta do sprawdzania, czy liczba jest parzysta, to algorytm. Zwykle w celu utworzenia programu rozwiązującego jakiś problem najpierw opracowuje się algorytm, a potem program stanowiący implementację tego algorytmu. Zatem algorytm sprawdzania parzystości można zapisać następująco. Najpierw dzielimy liczbę przez dwa. Jeśli reszta z tego dzielenia to zero, liczba jest parzysta; w przeciwnym wypadku liczba jest nieparzysta. Mając algorytm, możemy zapisać instrukcje stanowiące implementację algorytmu na danym systemie komputerowym. Instrukcje takie należą do jakiegoś konkretnego języka programowania, na przykład Visual Basic, Java, C++ czy C.

Języki wysokiego poziomu

Kiedy powstały komputery, jedynym sposobem ich programowania było użycie zapisu dwójkowego, korzystanie z instrukcji maszynowych i wstawianie wartości bezpośrednio do pamięci. Następnym etapem rozwoju techniki komputerowej były *języki asemblerowe*, które umożliwiły programistom korzystanie z komputera na nieco wyższym poziomie. Zamiast podawać ciągi liczb dwójkowych, w językach asemblerowych programista może różne operacje realizować za pomocą odwołania się do ich nazw, a także może przez nazwy odwoływać się do miejsca w pamięci. Specjalny program, *asembler*, przekształca taki program w z formatu symbolicznego na konkretne instrukcje maszynowe.

Języki asemblerowe traktuje się jak języki niskiego poziomu, bowiem każdej instrukcji takiego języka odpowiada pojedyncza instrukcja maszynowa. Programista musi znać instrukcje konkretnej maszyny, uzyskany program jest zaś *nieprzenośny*, czyli nie zadziała na innym procesorze, jeśli nie zostanie napisany od nowa. Wynika to stąd, że różne procesory mają różne zestawy instrukcji, czyli zestawy instrukcji są zależne od maszyny.

W końcu pojawiły się języki wysokiego poziomu; jednym z pierwszych był FORTRAN (nazwa pochodzi od *FORmula TRANslation* — przekształcanie wyrażeń). Programiści tworzący programy w FORTRAN-ie nie musieli już zaprzątać sobie głowy architekturą danego komputera, operacje tego języka — znacznie bardziej zaawansowane — były wyższego poziomu, a nie miały bezpośredniego przełożenia na instrukcje języka maszynowego. Jednej *instrukcji* języka FORTRAN odpowiadało wiele instrukcji maszynowych.

Standaryzacja składni języka wyższego poziomu spowodowała, że program można było zapisywać w sposób niezależny od maszyny. Wobec tego program po niewielkich zmianach można było uruchomić na każdym komputerze zawierającym obsługę danego języka.

Aby tworzyć języki wysokiego poziomu, trzeba było przygotować specjalne programy przekształcające instrukcje języka wysokopoziomowego na format zrozumiały dla komputera, czyli na instrukcje maszynowe danego komputera. Taki program to *kompilator*.

Systemy operacyjne

Zanim powiemy więcej o kompilatorach, musimy zrozumieć, jaką rolę odgrywa program komputerowy nazywany systemem operacyjnym.

System operacyjny to program, który kontroluje działanie całego komputera. Wszystkie operacje wejścia i wyjścia (w skrócie oznaczane przez I/O) wykonywane przez komputer są cedowane na system operacyjny. System musi też zarządzać zasobami komputera i sterować wykonywaniem programów.

Obecnie jednym z najpopularniejszych systemów komputerowych jest system Unix, stworzony w laboratoriach firmy Bell. Unix jest systemem niezwykłym, bowiem działa na rozmaitych komputerach i ma różne odmiany, takie jak Linux czy Mac OS X. Dawniej systemy operacyjne były związane z pojedynczymi komputerami. Jednak system Unix

został napisany głównie w języku C i jako taki miał niewielkie wymagania dotyczące architektury komputera. Wobec tego został przeniesiony na rozmaite systemy bez zbytniego wysiłku.

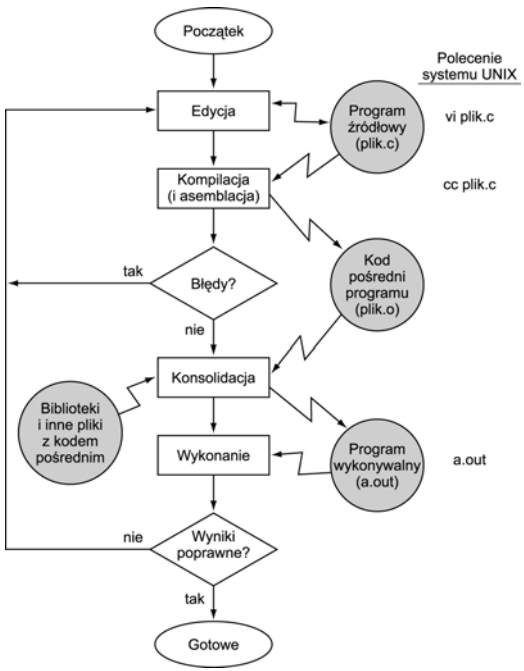
System Microsoft Windows to inny przykład udanego sytemu operacyjnego. System ten działa głównie na procesorach Pentium i procesorach zgodnych z Pentium.

Od niedawna serca użytkowników podbijają systemy operacyjne przeznaczone specjalnie dla urządzeń przenośnych, takich jak telefony komórkowe i tablety. Dwa najpopularniejsze z nich to Android firmy Google i iOS firmy Apple.

Kompilowanie programów

Kompilator to program, który w zasadzie nie różni się od innych programów prezentowanych w niniejszej książce, jest tylko znacznie bardziej złożony. Kompilator analizuje program napisany w jakimś języku i przekształca go na postać, którą można wykonywać w używanym systemie operacyjnym.

Na rysunku 1.1 pokazano etapy związane z wpisywaniem, kompilowaniem i wykonywaniem programu napisanego w języku C oraz typowe polecenia systemu Unix, jakie należy podać, aby taki program uruchomić.



Rysunek 1.1. Wprowadzanie, kompilowanie i wykonywanie programów pisanych w języku C przy użyciu wiersza poleceń

Aby program można było skompilować, najpierw trzeba go wpisać do *pliku*. W różnych systemach odmienne są zasady nazywania plików, ale najczęściej użytkownik decyduje o doborze nazwy. Programy w języku C zwykle są w plikach, które na końcu mają znaki „.c" (nie jest to wymóg, ale powszechnie przyjęta konwencja). Wobec tego przykładową nazwą pliku z programem C może być *prog1.c*.

Program do pliku zapisujemy, stosując edytor tekstowy. W systemie Unix na przykład powszechnie używany jest edytor *vim*. Program w formie takiej, w jakiej zapisujemy go w edytorze, to *program źródłowy*; jest to podstawowa postać tego programu w języku C. Kiedy program źródłowy zostanie zapisany w pliku, można przejść do jego kompilowania.

Proces kompilacji uruchamiamy, wydając w systemie stosowne polecenie. Po podaniu samego polecenia należy wpisać nazwę kompilowanego pliku. W systemie Unix na przykład polecenie uruchamiające kompilację to `cc`. Kiedy używamy popularnego kompilatora GNU C, odpowiednie polecenie to `gcc`.

Wpisanie wiersza:

```
gcc prog1.c
```

spowoduje skompilowanie programu źródłowego z pliku *prog1.c*.

W pierwszym etapie kompilacji wszystkie instrukcje programu źródłowego sprawdzane są pod względem zgodności ze składnią i semantyką użytego języka[1]. Jeśli kompilator wykryje jakieś błędy, użytkownik zostanie o nich poinformowany i dalsza kompilacja zostanie przerwana. Trzeba poprawić błędy w kodzie źródłowym za pomocą edytora, a następnie zacząć kompilację od nowa. Zwykle tego typu błędy wynikają z istnienia wyrażeń z niewłaściwą liczbą wyrazów (błędy składniowe) lub z użycia „niezdefiniowanej" zmiennej (błędy semantyczne).

Kiedy z programu zostaną już usunięte błędy składniowe i semantyczne, kompilator przetwarza wszystkie instrukcje programu i przekłada je na instrukcje „niższego poziomu". W większości systemów oznacza to przekształcanie instrukcji języka na inne instrukcje lub na instrukcje asemblerowe, które realizują dokładnie te same zadania.

Kiedy program zostanie przekształcony w odpowiadający mu program asemblerowy, następnym etapem kompilacji jest przeobrażenie instrukcji asemblerowych w konkretne instrukcje maszynowe. Ten etap może wiązać się z uruchomieniem dodatkowego programu zewnętrznego, *asemblera*, choć nie jest to regułą. W większości systemów asembler jest uruchamiany automatycznie w ramach kompilacji.

Asembler pobiera kolejne instrukcje języka asemblerowego, przekształca je w postać dwójkową i zapisuje jako tak zwany *kod pośredni* w osobnym pliku. Zwykle w systemie Unix pliki te mają nazwę taką samą jak plik źródłowy, ale ich rozszerzenie (ostatnia litera z kropką) to *.o*, a nie *.c*. W środowisku Windows rozszerzeniem zwykle jest *.obj*.

Kiedy mamy już program w formie kodu pośredniego, możemy przystąpić do jego *konsolidowania*. Ten etap po wywołaniu w systemie Unix poleceń `cc` czy `gcc` jest realizowany automatycznie. Celem konsolidacji jest przekształcenie programu do jego

---

[1] Ściślej rzecz biorąc, w języku C kompilator zwykle najpierw sprawdza, czy w programie występują pewne specjalne instrukcje; ten etap kompilacji to preprocesor. Dokładniej omówimy go w rozdziale 13.

ostatecznej postaci, tak aby można było go uruchamiać. Jeśli program wykorzystuje inne programy wcześniej przetworzone przez kompilator, na tym etapie wszystkie one są konsolidowane w całość. Przeszukiwane są też systemowe *biblioteki* programów i odpowiednie ich fragmenty zostają włączone do ostatecznej wersji programu.

Kompilacja i konsolidacja programu często nazywane są łącznie *budowaniem programu*.

Skonsolidowany program, już w formacie wykonywalnym, jest umieszczany w kolejnym pliku i może być uruchamiany, czyli *wykonywany*. W systemie Unix taki plik domyślnie ma nazwę *a.out*; w środowisku Windows zwykle nazwa programu wykonywalnego jest taka sama jak programu źródłowego, ale zamiast rozszerzenia *c* stosowane jest rozszerzenie *exe*.

Aby wykonać skompilowany program, wystarczy wpisać nazwę jego pliku wykonywalnego. Wobec tego polecenie:

```
a.out
```

spowoduje *załadowanie* programu *a.out* do pamięci i jego uruchomienie.

Kiedy program jest wykonywany, realizowane są kolejno wszystkie jego instrukcje. Jeśli program oczekuje danych od użytkownika, zawiesza swoje działanie, póki tych danych nie uzyska. Program może też po prostu czekać na jakieś *zdarzenie*, na przykład na kliknięcie myszą. Wyniki działania programu można zobaczyć w oknie często nazywanym *konsolą*. Wyniki te mogą też zostać przekierowane do wskazanego pliku.

Jeśli wszystko pójdzie dobrze (co rzadko zdarza się za pierwszym razem), program realizuje funkcje, które zostały w nim zakodowane. Jeśli daje wyniki inne od oczekiwanych, trzeba się cofnąć i ponownie przeanalizować logikę programu. Jest to etap *usuwania błędów* lub *uruchamiania programu* (*debugging*), polegający na usunięciu z programu wszystkich błędów (po angielsku takie błędy określa się jako *bug*, co po polsku znaczy *pluskwa*; stąd czasem mowa o *odpluskwianiu* programu). Zwykle usuwanie błędów wymaga dokonania zmian w pierwotnym programie źródłowym. Następnie trzeba powtarzać cały proces kompilacji, konsolidacji i wykonywania programu, aż do uzyskania pożądanych wyników.

Zintegrowane środowiska programistyczne

Poszczególne kroki tworzenia programu w języku C zostały już omówione; pokazaliśmy też typowe polecenia. Często jednak cały cykl edycji, kompilacji, uruchamiania i usuwania błędów z programu jest obsługiwany przez pojedynczą aplikację, zwaną zintegrowanym środowiskiem programistycznym (IDE — *Integrated Development Environment*). IDE to program okienkowy, który ułatwia pracę z dużymi programami, umożliwia edycję plików w osobnych oknach, a także pozwala na kompilację, konsolidację, uruchamianie programów i usuwanie z nich błędów.

Programiści pracujący w systemie Mac OS X mają do dyspozycji rozwijane przez firmę Apple środowisko Xcode. W systemie Windows dobrym przykładem IDE jest Microsoft Visual Studio. Wszystkie IDE znakomicie upraszczają proces tworzenia oprogramowania, więc warto na któreś się zdecydować. Większość środowisk IDE obsługuje — oprócz C — także wiele innych języków programowania, na przykład Objective-C, Java, C# czy C++.

Więcej informacji na temat zintegrowanych środowisk programistycznych znajdziemy w dodatku E.

Interpretery

Zanim skończymy omawianie kompilacji, musimy zwrócić uwagę na jeszcze jedną metodę analizowania i wykonywania programów napisanych w językach wysokiego poziomu. W metodzie tej programy nie są kompilowane, lecz *interpretowane*. Interpreter analizuje i od razu wykonuje kolejne instrukcje programu. Zwykle ułatwia to usuwanie błędów z programu. Jednak języki interpretowane działają wolniej niż ich odpowiedniki kompilowane, gdyż poszczególne instrukcje języka nie są przekształcane w postać niższego poziomu przed wykonaniem programu.

Przykładami języków, które często są interpretowane, a nie kompilowane, są BASIC i JavaScript. Inne przykłady to tak zwana *powłoka* systemu Unix (*shell*) czy język Python. Są też dostępne interpretery języka C.

2

Kompilujemy i uruchamiamy pierwszy program

W tym rozdziale zapoznamy się z językiem C, aby niejako poczuć, na czym polega programowanie. Jak lepiej możemy ocenić język? Tylko oglądając konkretne programy. Rozdział ten jest krótki, ale bardzo treściwy. Oto lista opisanych w nim tematów:

- pisanie pierwszego programu;
- modyfikowanie pierwszego programu tak, aby zmieniał zwracane informacje;
- właściwości funkcji `main()`;
- wysyłanie informacji na wyjście za pomocą funkcji `printf()`;
- zwiększanie czytelności kodu źródłowego przez dodawanie komentarzy.

Na początek wybierzemy dość prosty przykład — program wyświetlający zdanie: „Programowanie to niezła zabawa". Potrzebny kod zawiera program 2.1.

Program 2.1. **Pierwszy program w języku C**

```
#include <stdio.h>

int main (void)
{
    printf ("Programowanie to niezła zabawa.\n");

    return 0;
}
```

W języku C ma znaczenie wielkość liter, nie mają go natomiast sposób podziału programu na wiersze i rozmieszczenie kodu w tych wierszach. Fakt ten należy wykorzystać podczas programowania, aby poprawić czytelność tworzonego kodu. Programiści często używają tabulatora jako wygodnej metody robienia wcięć.

Kompilujemy nasz program

Zajmijmy się dalej naszym pierwszym programem w języku C. Najpierw zapisujemy go do pliku, korzystając z edytora tekstu. Użytkownicy systemu Unix zwykle posługują się programami *vi* czy *emacs*.

Większość kompilatorów języka C za programy w C uznaje pliki mające na końcu kropkę i literę „c". Zatem załóżmy, że nasz program 2.1 jest w pliku *prog1.c*. Teraz program ten musimy skompilować.

Gdy korzystamy z kompilatora GNU C, wystarczy wywołać polecenie `gcc` i jako parametr podać mu nazwę pliku:

```
$ gcc prog1.c
$
```

Podczas korzystania ze standardowego kompilatora systemu Unix używamy polecenia `cc` zamiast `gcc`. Tekst wprowadzany przez użytkownika (czytelnika) jest pogrubiony. Symbol dolara to znak zachęty pojawiający się w czasie kompilowania programu z wiersza poleceń. Możliwa jest zmiana postaci znaku zachęty.

Jeśli podczas wpisywania programu do pliku zrobimy jakieś błędy, kompilator pokaże je po wywołaniu polecenia `gcc`; zwykle wskaże też numery wierszy programu, w których błędy wystąpiły. Jeśli od razu pojawi się kolejny znak zachęty bez informacji o błędach — co pokazano powyżej — oznacza to, że w programie nie ma żadnych błędów.

Kiedy kompilator kompiluje i konsoliduje program, tworzy jego wersję wykonywalną. Gdy korzystamy z kompilatora GNU C lub standardowego kompilatora C, uzyskiwany plik domyślnie ma nazwę *a.out*. W systemie Windows będzie to plik *a.exe*.

Uruchamianie programu

Teraz możemy uruchomić skompilowany program; wystarczy w wierszu poleceń podać jego nazwę[1]:

```
$ a.out
Programowanie to niezła zabawa.
$
```

Programowi wykonywalnemu można też nadać inną nazwę. Robimy to za pomocą opcji –o, po której podajemy żądaną nazwę. Na przykład polecenie:

```
$ gcc prog1.c –o prog1
```

skompiluje program *prog1.c* i utworzy program wykonywalny *prog1*, który możemy wywołać, znowu podając jego nazwę:

```
$ prog1
Programowanie to niezła zabawa.
$
```

[1] Jeśli po wydaniu stosownego polecenia pojawi się komunikat `a.out: No such file or directory`, to prawdopodobnie bieżący katalog nie został umieszczony w ścieżce (zmienna środowiskowa PATH). Można albo ten katalog dodać do PATH, albo bieżący katalog podać bezpośrednio w wywołaniu: `./a.out`.

Analiza naszego pierwszego programu

Teraz przyjrzyjmy się dokładniej naszemu pierwszemu programowi. Pierwszy jego wiersz:

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

wystąpi na początku niemalże każdego programu. Jest tu informacja dla kompilatora, że dalej w programie będzie używana procedura wypisująca wyniki — `printf`. W poświęconym preprocesorowi rozdziale 12. dokładniej opisano działanie podanego wiersza.

Wiersz zawierający napis:

```
int main (void)
```

przekazuje do systemu informację, że nazwa programu to `main` i program ten *zwraca* liczbę całkowitą (napis `int`). Słowo `main` to nazwa specjalna, która wskazuje, *gdzie* ma się zacząć wykonywanie programu. Para nawiasów zaraz za słowem `main` informuje, że mamy do czynienia z funkcją. Słowo kluczowe `void` w tych nawiasach wskazuje, że funkcja `main` nie ma żadnych parametrów. Dokładniej zajmiemy się tym w rozdziale 7., kiedy będziemy omawiali funkcje.

Uwaga

Niektóre zintegrowane środowiska programistyczne automatycznie generują funkcję `main()`. Wówczas jej pierwszy wiersz może wyglądać tak:

```
int main(int argc, char *argv[])
```

Nie ma to wpływu na sposób działania programu, więc na razie się tym nie przejmuj.

Teraz system zna już funkcję `main`, więc możemy dokładnie opisać, co ona robi. W tym celu całą treść funkcji zamykamy w nawiasy klamrowe. Wszystkie instrukcje zamieszczone między tymi nawiasami zostaną uznane za część funkcji `main`. Program 2.1 ma tylko dwie takie instrukcje. Pierwsza zawiera *wywołanie* funkcji `printf`. Do `printf` przekazywany jest parametr (*argument*) będący łańcuchem znakowym:

```
"Programowanie to niezła zabawa.\n"
```

Procedura `printf` to funkcja z biblioteki C, która po prostu wyświetla swoje argumenty na ekranie. Ostatnie dwa znaki w powyższym przykładzie — odwrotny ukośnik (\) oraz litera `n` — tworzą razem *znak nowego wiersza*. Znak ten nakazuje, aby znaki znajdujące się za nim były umieszczane w następnym wierszu. Znak nowego wiersza działa podobnie jak klawisz powrotu karetki w maszynie do pisania. Czy ktoś to jeszcze pamięta?

Wszystkie instrukcje w języku C *muszą* kończyć się średnikiem `;`. Stąd właśnie średnik zaraz po nawiasie zamykającym, po wywołaniu funkcji `printf`.

Ostatnia instrukcja funkcji `main` to:

```
return 0;
```

Nakazuje ona zakończenie wykonywania `main` i zwrócenie do systemu wartości `0`; jest to kod powrotu. Zamiast zera można użyć innej, dowolnej liczby całkowitej.

Zgodnie z powszechnie przyjętą konwencją, zero oznacza, że nie wystąpił żaden błąd. Inne liczby mogą wskazywać rodzaj błędu (na przykład brak pliku). Kod powrotu może być sprawdzany przez inne programy, na przykład przez powłokę systemu Unix, w celu sprawdzenia, czy wykonanie programu się powiodło.

Omówiliśmy już nasz pierwszy program, teraz możemy go zmodyfikować tak, aby dodatkowo wyświetlił napis: „A programowanie w C jest jeszcze lepsze". Wystarczy dodać jeszcze jedno wywołanie funkcji `printf`, jak w programie 2.2. Pamiętajmy, że każda instrukcja języka C musi się kończyć średnikiem.

Program 2.2.

```
#include <stdio.h>

int main (void)
{
    printf ("Programowanie to niezła zabawa.\n");
    printf ("A programowanie w C jest jeszcze lepsze.\n");

    return 0;
}
```

Po wpisaniu, skompilowaniu i uruchomieniu programu 2.2 na ekranie możemy spodziewać się następującego wyniku.

Program 2.2. **Wyniki**

```
Programowanie to niezła zabawa.
A programowanie w C jest jeszcze lepsze.
```

Jak zobaczymy w następnym przykładzie, wcale nie trzeba dwukrotnie wywoływać funkcji `printf`, osobno dla każdego wiersza wyniku. Przeanalizujmy program 2.3 i spróbujmy przewidzieć wyniki jego działania (tylko bez oszukiwania!).

Program 2.3. **Wiele wierszy wynikowych**

```
#include <stdio.h>

int main (void)
{
    printf ("Sprawdzanie...\n..1\n...2\n....3\n");

    return 0;
}
```

Program 2.3. **Wyniki**

```
Sprawdzanie...
..1
...2
....3
```

Wyświetlanie wartości zmiennych

Procedura `printf` jest procedurą najczęściej używaną w całej tej książce. Wynika to stąd, że pozwala ona łatwo i wygodnie pokazywać wyniki działania programu. Nie tylko można wyświetlać zwykłe zdania, lecz także wartości *zmiennych* oraz wyniki obliczeń. Program 2.4 wykorzystuje właśnie procedurę `printf` do wyświetlenia wyników dodania dwóch liczb, 50 i 25.

Program 2.4. **Wyświetlanie wartości zmiennych**

```
#include <stdio.h>

int main (void)
{
    int sum;

    sum = 50 + 25;
    printf ("Suma 50 i 25 to %i\n", sum);

    return 0;
}
```

Program 2.4. **Wyniki**

```
Suma 50 i 25 to 75
```

W pierwszym wierszu programu 2.4 *deklarujemy zmienną* `sum` jako zmienną typu `int`, czyli jako liczbę całkowitą. W C trzeba najpierw zdeklarować zmienną, zanim można będzie jej użyć. Deklaracja przekazuje do kompilatora C informację, jak dana zmienna będzie używana. Dzięki temu kompilator może wygenerować właściwe instrukcje rezerwujące miejsce na zmienną w pamięci oraz pobierające jej wartość. Zmienna zadeklarowana jako `int` może przechowywać tylko liczby całkowite, czyli wartości bez cyfr po przecinku. Przykładami wartości całkowitych są 3, 5, –20 czy 0. Liczby mające cyfry po przecinku, jak 3,14 czy 2,455, czy 27,0, to liczby *zmiennoprzecinkowe*.

Zmienna całkowita `sum` służy do zapisywania wyniku dodawania dwóch liczb, 50 i 25. Umyślnie w programie zostawiono pusty wiersz oddzielający wizualnie deklaracje zmiennych od reszty instrukcji. Jest to kwestia przyjętego stylu pisania. Czasami dodanie pustego wiersza poprawia czytelność programu.

Instrukcja:

```
sum = 50 + 25;
```

wygląda podobnie w większości języków programowania — liczba `50` jest dodawana (symbol `+`) do liczby `25`, wynik zaś jest zapisywany (operator *przypisania*, znak równości) do zmiennej `sum`.

Wywołanie procedury `printf` w programie 2.4 ma dwa *argumenty* ujęte w nawiasy. Argumenty te są rozdzielone przecinkiem. Pierwszy argument to zawsze wyświetlany

łańcuch znaków. Jednak w tym łańcuchu często trzeba pokazać wartości konkretnych zmiennych. Tu chcemy wyświetlić wartość zmiennej `sum` zaraz po napisie:

```
Suma 50 i 25 to
```

Symbol procenta w pierwszym argumencie to znak specjalny, rozpoznawany przez funkcję `printf`. Znak znajdujący się tuż za symbolem procenta opisuje *typ* wartości, jaka ma być w tym miejscu wyświetlona. W powyższym programie litera `i` oznacza wartość całkowitoliczbową[2].

Kiedy tylko funkcja `printf` napotyka znaki `%i` w łańcuchu znakowym, automatycznie wyświetla tam wartość następnego swojego argumentu. W naszym wypadku następnym argumentem jest zmienna `sum`, więc to jej wartość jest wyświetlana.

Spróbujmy teraz przewidzieć wyniki działania programu 2.5.

Program 2.5. **Wyświetlanie wielu wartości**

```
#include <stdio.h>

int main (void)
{
    int value1, value2, sum;

    value1 = 50;
    value2 = 25;
    sum = value1 + value2;
    printf ("Suma %i i %i to %i\n", value1, value2, sum);

    return 0;
}
```

Program 2.5. **Wyniki**

```
Suma 50 i 25 to 75
```

Pierwsza instrukcja programu zawiera deklarację zmiennych `value1`, `value2` i `sum`, wszystkie typu `int`. Równie dobrze można by tę instrukcję zastąpić trzema równoważnymi:

```
int value1;
int value2;
int sum;
```

Po zadeklarowaniu tych trzech zmiennych zmiennej `value1` przypisujemy wartość `50`, `value2` — wartość 25. Następnie jest liczona suma tych dwóch wartości, która zostanie przypisana do zmiennej `sum`.

Wywołanie procedury `printf` ma teraz cztery argumenty. Znowu pierwszy z nich — zwany zwykle *łańcuchem formatującym* — opisuje sposób interpretacji pozostałych

[2] Zauważmy, że także formant `%d` pozwala wyświetlać liczby całkowite. W tej książce konsekwentnie będziemy używali jednak `%i`.

argumentów. Wartość `value1` ma być wyświetlona za napisem „Suma”, miejsce użycia wartości `value2` i `sum` pokazano w formie dwóch wystąpień napisu `%i` w łańcuchu formatującym.

Komentarze

W ostatnim programie z tego rozdziału — programie 2.6 — demonstrujemy użycie *komentarzy*. Instrukcja komentarza służy do dokumentowania programu i poprawy jego czytelności. Komentarze, co zobaczymy w następnym przykładzie, mówią osobom korzystającym z programu — programistom lub osobom odpowiedzialnym za serwisowanie tego programu — o czym myślał programista, kiedy pisał dany kod.

Program 2.6. **Użycie w programie komentarzy**

```
/* Program dodaje dwie liczby całkowite,
   następnie wyświetla wynik tego dodawania.  */

#include <stdio.h>

int main (void)
{
    // Deklaracja zmiennych
    int value1, value2, sum;

    // Przypisanie wartości, wyliczenie ich sumy
    value1 = 50;
    value2 = 25;
    sum = value1 + value2;
    // Pokazanie wyniku
    printf ("Suma %i i %i to %i\n", value1, value2, sum);

    return 0;
}
```

Program 2.6. **Wyniki**

```
Suma 50 i 25 to 75
```

Komentarze w programach C wstawiamy na dwa sposoby. Komentarz może zaczynać się od pary znaków / i *. Jest to *początek* komentarza. Tego typu komentarze muszą być zakończone parą znaków * i /. Wszystkie znaki między /* a */ są uważane za komentarz i pomijane przez kompilator C. Tego typu komentarze są stosowane, kiedy trzeba napisać nieco więcej, na przykład kilka wierszy. Drugi sposób komentowania programów w języku C to użycie pary ukośników — //. Wszystkie znaki za tymi ukośnikami, aż do końca wiersza, są traktowane jako komentarz.

W programie 2.6 użyto czterech osobnych instrukcji komentarza. Poza tym program ten jest identyczny z programem 2.5. Trzeba przyznać, że w tym przykładzie jedynie pierwszy

komentarz, znajdujący się na samym początku, jest naprawdę przydatny (zgadza się, można wstawić do programu tyle komentarzy, że jego czytelność zamiast się poprawić, znacząco spadnie!).

Trudno przecenić wagę celnych komentarzy wstawianych do programu. Wielokrotnie programista wraca do swojego programu po sześciu miesiącach i z przerażeniem stwierdza, że za żadne skarby nie jest w stanie przypomnieć sobie, po co użył jakiejś procedury czy instrukcji. Krótki komentarz wstawiony w odpowiednim miejscu może zaoszczędzić mnóstwa pracy przy analizie kodu i sporo czasu na przypominanie sobie, o co tutaj chodziło.

Warto wyrobić sobie nawyk wstawiania komentarzy do programu podczas jego pisania. Jest po temu wiele powodów. Po pierwsze, znacznie łatwiej komentować program, kiedy dokładnie pamiętamy, o co w nim chodzi; po jakimś czasie trudno sobie przypominać, co mieliśmy na myśli. Po drugie, wstawianie komentarzy na tak wczesnym etapie pozwala z nich skorzystać przy usuwaniu błędów logicznych z programu. Komentarz może nie tylko ułatwić czytanie programu, lecz także przydaje się do znalezienia źródła pomyłki. No i ostatni argument — autorowi tej książki nie zdarzyło się jeszcze spotkać programisty, który lubiłby dokumentować swoje programy. Prawda jest taka, że kiedy kończymy usuwanie błędów z programu, nie myślimy o wstawianiu komentarzy. Wstawianie komentarzy już podczas tworzenia programu nieco ułatwia to zadanie.

Na tym kończymy wstęp do programowania w C. Powinniśmy już orientować się, na czym ta zabawa polega, i samodzielnie napisać proste programy. W następnym rozdziale zajmiemy się pewnymi niuansami cudownie potężnego i elastycznego języka programowania, jakim jest język C. Jednak najpierw należy samodzielnie rozwiązać poniższe ćwiczenia, aby powtórzyć omówiony dotąd materiał.

Ćwiczenia

1. Przepisz i uruchom sześć programów z tego rozdziału. Porównaj uzyskane wyniki z wynikami podanymi w książce.
2. Napisz program, który wyświetli następujące napisy:
   a) W języku C wielkość liter ma znaczenie.
   b) Wykonywanie programu zaczyna się od funkcji main.
   c) Pary nawiasów klamrowych oznaczają instrukcje tworzące procedurę.
   d) Wszystkie instrukcje programu muszą kończyć się średnikami.
3. Jakie wyniki da poniższy program?

```
#include <stdio.h>

int main (void)
{
    printf ("Sprawdzanie...");
    printf ("....1");
    printf ("...2");
```

```
    printf ("..3");
    printf ("\n");

    return 0;
}
```

4. Napisz program odejmujący od 87 wartość 15 i pokazujący wynik wraz ze stosownym opisem.
5. W poniższym programie znajdź błędy składniowe. Następnie przepisz i uruchom poprawiony program, aby upewnić się, że wszystkie błędy zostały zlokalizowane.

```
#include <stdio.h>

int main (Void)
{
    INT sum;
    /* WYLICZ WYNIK
    sum = 25 + 37 – 19
    /* POKAŻ WYNIKI //
    printf ("Odpowiedzią jest %i\n" sum);
    return 0;
}
```

6. Jaki wynik da poniższy program?

```
#include <stdio.h>

int main (void)
{
    int answer, result;

    answer = 100;
    result = answer – 10;
    printf ("Wynik to %i\n", result + 5);

    return 0;
}
```

3

Zmienne, typy danych i wyrażenia arytmetyczne

Największą zaletą programów jest to, że przy ich użyciu można przetwarzać dane. Jednak żeby w pełni wykorzystać ich możliwości, należy dokładnie znać różne dostępne typy danych oraz umieć tworzyć i nazywać zmienne. W języku C dostępnych jest kilka operatorów matematycznych umożliwiających manipulację danymi.
Oto lista tematów opisanych w tym rozdziale:

- typy danych `int`, `float`, `double`, `char` oraz `_Bool`;
- modyfikowanie typów danych przy użyciu słów kluczowych `short`, `long` oraz `long long`;
- zasady tworzenia nazw zmiennych;
- podstawowe operatory i wyrażenia arytmetyczne;
- rzutowanie typów.

Typy danych i stałe

Omówiliśmy już podstawowy typ danych języka C — `int`. Jak pamiętamy, zmienna tego typu może zawierać tylko liczby całkowite, czyli liczby niemające części ułamkowej.

Język C ma jeszcze cztery inne podstawowe typy danych: `float`, `double`, `char` oraz `_Bool`. Zmienna typu `float` może zostać użyta dla liczb zmiennoprzecinkowych. Typ `double` jest podobny do `float`, ale umożliwia zapisanie liczby z około dwukrotnie większą dokładnością. Typ `char` może być wykorzystany do zapisywania pojedynczego znaku, na przykład litery *a*, cyfry *6* czy średnika (więcej na ten temat w dalszej części rozdziału). W końcu typ danych `_Bool` może zawierać jedynie dwie wartości: `0` lub `1`. Zmienne tego typu są używane, kiedy potrzebna jest informacja w rodzaju włączone/wyłączone, tak/nie czy prawda/fałsz. Nazywa się je też czasami wyborami binarnymi.

W języku C liczba, pojedynczy znak lub łańcuch znaków to *stałe*, na przykład liczba 58 to stała wartość całkowitoliczbowa. Łańcuch `"Programowanie w C to niezła zabawa.\n"` to przykład stałego łańcucha znakowego. Wyrażenia, których wszystkie elementy są stałymi, to *wyrażenia stałe*. Wobec tego wyrażenie

```
128 + 7 – 17
```

jest wyrażeniem stałym, gdyż każdy z jego elementów jest wartością stałą. Jeśli jednak `i` zadeklarujemy jako zmienną typu `int`, to wyrażenie:

```
128 + 7 – i
```

nie będzie już wyrażeniem stałym, ponieważ jego wartość zmienia się zależnie od wartości `i`. Jeśli `i` równa się 10, to wartość całego wyrażenia wynosi 125, ale jeśli `i` równa się 200, to wartość całego wyrażenia wynosi –65.

Podstawowy typ danych int

W języku C stała całkowitoliczbowa to jedna lub więcej cyfr. Jeśli przed taką stałą znajduje się znak minus, mamy do czynienia z wartością ujemną. Przykładami całkowitoliczbowych wartości stałych są `158`, `–10` czy `0`. Między cyframi nie wolno wstawiać żadnych spacji, poza tym nie można grupować cyfr za pomocą przecinków ani kropek (zatem nie można napisać `12,000` — zamiast tego trzeba użyć stałej `12000`).

W języku C przewidziano dwa specjalne formaty dotyczące zapisu liczb innych niż dziesiętne. Jeśli pierwszą cyfrą wartości jest `0`, liczba ta jest traktowana jako liczba *ósemkowa*. Wtedy pozostałe cyfry muszą być też cyframi ósemkowymi, czyli należeć do zakresu od 0 do 7. Aby zatem zapisać w C ósemkową wartość `50`, której odpowiada dziesiętne `40`, piszemy `050`. Analogicznie, ósemkowa stała `0177` odpowiada dziesiętnej stałej `127` (1·64+7·8+7). Wartość można wyświetlić jako ósemkową, jeśli w łańcuchu formatującym funkcji `printf` użyjemy formantu `%o`. Wtedy pokazywana jest liczba ósemkowa, ale bez wiodącego zera. Gdy zero jest potrzebne, używamy formantu `%#o`.

Jeśli stała liczba całkowita poprzedzona jest zerem i literą *x* (wielką lub małą), wartość jest traktowana jako liczba szesnastkowa. Zaraz za znakiem *x* znajdują się cyfry szesnastkowe, czyli cyfry od *0* do *9* oraz litery od *a* do *f* (lub od *A* do *F*). Litery reprezentują odpowiednie wartości z zakresu od *10* do *15*. Aby zatem do zmiennej `rgbColor` typu `int` przypisać szesnastkową wartość `FFEF0D`, możemy użyć instrukcji:

```
rgbColor = 0xFFEF0D;
```

Aby wyświetlić wartość szesnastkowo bez wiodących `0x`, z małymi „cyframi” od *a* do *f*, używamy formantu `%x`. Jeśli mają być dodane wiodące `0x`, stosujemy formant `%#x`, taki jak poniżej:

```
printf ("Kolor to %#x\n", rgbColor);
```

Jeśli chcemy uzyskać wielkie „cyfry”, od *A* do *F*, i ewentualnie wiodące `0X`, używamy formantów `%X` oraz `%#X`.

Alokacja pamięci, zakres wartości

Każda wartość, czy to znak, czy liczba całkowita, czy zmiennoprzecinkowa, ma dopuszczalny *zakres* wartości. Zakres ten wiąże się z ilością pamięci przeznaczanej na wartości danego typu. Ogólnie rzecz biorąc, w języku nie zdefiniowano wielkości pamięci na poszczególne typy; zależy to od używanego komputera, więc są to wielkości *zależne od maszyny*. Na przykład liczba typu `int` może mieć 32 lub 64 bity. W programach nigdy nie należy przyjmować założeń dotyczących wielkości danych określonego typu. Istnieją jednak pewne gwarantowane wielkości — wartości danego typu nigdy nie będą zapisywane w mniejszej ilości pamięci niż wielkość gwarantowana, na przykład dla typu `int` są to 32 bity (32 bity to w wielu komputerach „słowo").

Typ zmiennoprzecinkowy float

Zmienna zadeklarowana jako zmienna typu `float` może zostać użyta do przechowywania liczb z częścią ułamkową. Stałe zmiennoprzecinkowe charakteryzują się występowaniem w ich zapisie kropki dziesiętnej. Można pominąć cyfry przed kropką, można pominąć cyfry za kropką, ale nie jednocześnie. Przykładami poprawnych stałych typu `float` są `3.`, `125.8` czy `–.0001`. Jeśli chcemy wyświetlić liczbę zmiennoprzecinkową przy użyciu funkcji `printf`, korzystamy z formantu `%f`.

Stałe zmiennoprzecinkowe mogą być też zapisywane w *notacji naukowej*. Zapis `1.7e4` oznacza wartość $1{,}7\times10^4$. Wartość przed literą `e` to *mantysa*, z kolei za literą `e` to *wykładnik*. Wykładnik może być poprzedzony znakiem plus lub minus i oznacza potęgę 10, przez jaką należy przemnożyć mantysę. Wobec tego w stałej `2.25e–3` wartość mantysy to `2.25`, a wykładnik to `–3`. Stała taka odpowiada wartości 2.25×10^{-3}, czyli 0,00225. Litera `e` oddzielająca mantysę od wykładnika może być wielka lub mała.

Aby wyświetlić wartość w notacji naukowej, w funkcji `printf` używamy formantu `%e`. Formant `%g` powoduje, że sama funkcja `printf` decyduje, czy wartość wyświetli w zwykłym formacie zmiennoprzecinkowym czy w notacji naukowej. Decyzja ta zależy od wykładnika — jeśli jest on mniejszy od –4 lub większy od 5, stosowany jest formant `%e` (czyli notacja naukowa); w przeciwnym razie używany jest formant `%f`.

Jeśli musimy wyświetlać wartości zmiennoprzecinkowe, warto korzystać z formantu `%g`, gdyż daje on najbardziej estetyczne wyniki.

Szesnastkowa stała zmiennoprzecinkowa zaczyna się od `0x` lub `0X`, dalej jest jedna lub więcej cyfr dziesiętnych lub szesnastkowych, potem litera `p` lub `P`, w końcu opcjonalny wykładnik dwójki ze znakiem, na przykład `0x0.3p10` oznacza wartość $3/16\times2^{10}$ `= 0.5`.

Rozszerzony typ double

Typ `double` jest bardzo podobny do typu `float`, ale korzystamy z niego, kiedy zakres wartości typu `float` nie wystarcza. Zmienne zadeklarowane jako zmienne typu `double` mogą mieć blisko dwukrotnie więcej cyfr znaczących niż zmienne typu `float`. W większości komputerów są one zapisywane w 64 bitach.

Jeśli nie określimy inaczej, domyślnie stałe wartości zmiennoprzecinkowe są traktowane w C jako wartości typu `double`; jeśli chcemy mieć wartość stałą typu `float`, na koniec trzeba dodać literę `f` lub `F`, na przykład:

```
12.5f
```

Aby wyświetlić wartość typu `double`, korzystamy z formantów `%f`, `%e` lub `%g` interpretowanych tak samo jak dla wartości typu `float`.

Pojedyncze znaki, typ char

Zmienne typu `char` mogą przechowywać pojedyncze znaki[1]. Stałą znakową tworzymy, zamykając znak w parze apostrofów, na przykład `'a'`, `';'` czy `'0'`. Pierwsza stała odpowiada literze *a*, druga — średnikowi, a trzecia — cyfrze zero; trzeba pamiętać, że nie jest to liczba zero! Nie należy mylić stałych znakowych, czyli pojedynczych znaków ujętych w apostrofy, z łańcuchami znakowymi, ujętymi w cudzysłowy.

Stała znakowa `'\n'` — znak nowego wiersza — to całkiem poprawna stała, choć pozornie wydaje się, że jest niezgodna z podanymi zasadami. Jednak odwrotny ukośnik to w C znak specjalny, który nie jest traktowany jako zwykły znak. Wobec tego kompilator C traktuje napis `'\n'` jako pojedynczy znak, choć zapisywany jest za pomocą dwóch znaków. Istnieją też inne znaki specjalne zaczynające się od odwróconego ukośnika; pełna ich lista znajduje się w dodatku A.

Do pokazywania pojedynczego znaku za pomocą funkcji `printf` używamy formantu `%c`.

Logiczny typ danych, _Bool

Zmienna typu `_Bool` będzie wystarczająco duża, aby zmieścić dwie wartości: `0` i `1`. Dokładna ilość zajmowanej pamięci nie jest określona. Zmienne typu `_Bool` są używane w programach tam, gdzie potrzebny jest warunek logiczny (inaczej boole'owski). Za pomocą tego typu można na przykład wskazać, czy z pliku zostały odczytane wszystkie dane.

Zgodnie z powszechnie przyjętą konwencją `0` oznacza fałsz, a `1` — prawdę. Kiedy zmiennej przypisujemy wartość typu `_Bool`, w zmiennej tej `0` jest zapisywane jako liczba 0, każda zaś wartość niezerowa jako 1.

Aby ułatwić sobie pracę ze zmiennymi typu `_Bool`, używamy standardowego pliku nagłówkowego `<stdbool.h>`, w którym zdefiniowano wartości `bool`, `true` i `false`. Przykład jego zastosowania pokazano w programie 5.10A z rozdziału 5.

W programie 3.1 użyto podstawowych typów danych języka C.

Program 3.1. **Użycie podstawowych typów danych**

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

[1] W dodatku A omawiane są metody zapisywania znaków z rozszerzonego zestawu znaków; wykorzystuje się do tego cytowania, znaki uniwersalne oraz znaki „szerokie".

```
int main (void)
{
    int     integerVar = 100;
    float   floatingVar = 331.79;
    double  doubleVar = 8.44e+11;
    char    charVar = 'W';

    _Bool   boolVar = 0;

    printf ("integerVar = %i\n", integerVar);
    printf ("floatingVar = %f\n", floatingVar);
    printf ("doubleVar = %e\n", doubleVar);
    printf ("doubleVar = %g\n", doubleVar);
    printf ("charVar = %c\n", charVar);

    printf ("boolVar = %i\n", boolVar);

    return 0;
}
```

Program 3.1. ***Wyniki***

```
integerVar = 100
floatingVar = 331.790009
doubleVar = 8.440000e+11
doubleVar = 8.44e+11
charVar = W
boolVar = 0
```

W pierwszej instrukcji programu 3.1 deklarowana jest zmienna `integerVar` jako zmienna całkowitoliczbowa, przypisywana jest jej wartość początkowa 100; zapis ten jest równoważny dwóm osobnym wierszom:

```
int integerVar;
integerVar = 100;
```

W drugim wierszu wyników działania programu mamy wartość zmiennej `floatingVar`, `331.79`, wyświetlaną jako `331.790009`. Dokładna wartość zależy od używanego systemu. Powodem widocznej tu niedokładności jest sposób wewnętrznego zapisywania liczb w pamięci komputera. Większość czytelników zapewne zetknęła się z podobną niedokładnością podczas korzystania z kalkulatora. Dzielenie 1 przez 3 daje na kalkulatorze `.33333333`, ewentualnie może być nieco więcej trójek na końcu. Jest to przybliżenie jednej trzeciej. Teoretycznie trójek powinno być nieskończenie wiele, jednak kalkulator ma na wyświetlaczu tylko określoną liczbę cyfr i stąd bierze się pewna niedokładność. Z taką samą niedokładnością mamy do czynienia w naszym przypadku — pewne liczby zmiennoprzecinkowe nie dają się dokładnie zapisać w komputerze.

Kiedy wyświetlamy wartość zmiennej typu `float` lub `double`, mamy do dyspozycji trzy różne formanty. Format `%f` powoduje wyświetlenie wartości w sposób standardowy. Jeśli nie określimy inaczej, `printf` zawsze wyświetla wartości `float` i `double`, pokazując sześć cyfr części ułamkowej. W dalszej części tego rozdziału pokażemy, jak określać liczbę wyświetlanych cyfr.

Formant `%e` powoduje wyświetlanie wartości `float` i `double` w notacji naukowej. Tutaj także domyślnie wyświetlanych jest sześć cyfr części ułamkowej.

Formant `%g` powoduje, że funkcja `printf` wybiera jeden z dwóch poprzednich formantów, `%f` lub `%e`, i automatycznie usuwa wszelkie końcowe zera. Jeśli nie ma żadnej części ułamkowej, kropka dziesiętna też nie jest wyświetlana.

Przedostatnia instrukcja `printf` korzysta z formantu `%c` i wyświetla pojedynczy znak `'W'`, który przypisaliśmy zmiennej `charVar`. Pamiętajmy — jeśli łańcuch znaków (taki jak pierwszy argument funkcji `printf`) jest zamknięty w podwójny cudzysłów, to stała znakowa musi być ujęta w parę apostrofów.

Ostatnia funkcja `printf` pokazuje, jak można wyświetlić wartość zmiennej `_Bool` przy wykorzystaniu formantu liczb całkowitych — `%i`.

Określniki typu: long, long long, short, unsigned i signed

Jeśli bezpośrednio przed deklaracją typu `int` użyty zostanie określnik `long`, w niektórych systemach zmienna tak utworzona będzie miała większy zakres. Przykładowa deklaracja typu `long int` może wyglądać następująco:

```
long int silnia;
```

Deklarujemy zmienną `silnia` jako liczbę typu `long int`. Tak jak w typach `float` i `double`, konkretna dokładność zmiennej `long` zależy od używanego systemu. Często `int` i `long int` mają taki sam zakres i mogą przechowywać liczby całkowite o wielkości do 32 bitów (2^{31}–1, czyli 2, 147, 483, 647).

Wartości stałe typu `long int` możemy tworzyć, dodając do stałej liczbowej na końcu literę `L` (wielką lub małą). Między liczbą a `L` nie mogą pojawić się żadne spacje. Wobec tego deklaracja:

```
long int liczbaPunktow = 131071100L;
```

powoduje zadeklarowanie zmiennej `liczbaPunktow` typu `long int` z wartością początkową `131 071 100`.

Jeśli funkcja `printf` ma wyświetlić wartość typu `long int`, przed znakiem typu `i`, `o` lub `x` dodajemy modyfikator `l` (litera `l`). Wobec tego używamy formantu `%li`, aby wyświetlić wartość `long int` w systemie dziesiętnym. Aby wyświetlić tę samą wartość ósemkowo, stosujemy `%lo`, a szesnastkowo — `%lx`.

Istnieje jeszcze typ liczb całkowitych `long long`. Instrukcja:

```
long long int maxIloscPamieci;
```

deklaruje liczbę całkowitą o rozszerzonym zakresie. W tym przypadku mamy gwarancję, że wartość będzie miała przynajmniej 64 bity. W łańcuchu formatującym `printf` stosuje się dwie litery `ll` zamiast pojedynczej litery `l`, na przykład `%lli`.

Określnik `long` może też wystąpić przed deklaracją typu `double`:

```
long double deficyt_USA_2004;
```

Stałe typu `long double` zapisujemy jak wszystkie inne stałe zmiennoprzecinkowe, ale na końcu dodajemy literę `l` lub `L`, na przykład:

```
1.234e+7L
```

Aby wyświetlić wartość typu `long double`, używamy modyfikatora `L`. Wobec tego `%Lf` spowoduje wyświetlenie wartości `long double` jako zmiennoprzecinkowej, `%Le` — w notacji naukowej, a `%Lg` nakaże funkcji `printf` wybierać między formantami `%Lf` a `%Le`.

Specyfikator `short` umieszczony przed deklaracją typu `int` nakazuje kompilatorowi C traktować daną zmienną jako wartość całkowitą o zmniejszonym zakresie. Użycie `short` jest uzasadnione przede wszystkim wtedy, gdy chodzi o oszczędność miejsca w pamięci — kiedy program już wymaga bardzo dużo pamięci lub ilość dostępnej pamięci jest znacząco ograniczona.

W niektórych systemach zmienne typu `short int` zajmują połowę miejsca przeznaczonego na zwykłe zmienne `int`. Tak czy inaczej, mamy gwarancję, że typ `short int` zajmie nie mniej niż 16 bitów.

W języku C nie można jawnie zdefiniować stałej typu `short int`. Aby wyświetlić zmienną typu `short int`, przed normalnym oznaczeniem typu całkowitoliczbowego dodajemy literę `h`, czyli używamy formantów `%hi`, `%ho` i `%hx`. Można też użyć dowolnej konwersji wartości `short int`, gdyż i tak zostaną one odpowiednio przekształcone przy przekazywaniu do funkcji `printf`.

Ostatni określnik używany przed typem danych `int` informuje, czy zamierzamy przechowywać jedynie liczby dodatnie. Deklaracja:

```
unsigned int licznik;
```

przekazuje do kompilatora, że zmienna `licznik` zawierać będzie jedynie wartości dodatnie. Ograniczając zakres wartości do liczb dodatnich, zwiększamy dopuszczalny zakres liczb.

Stałą typu `unsigned int` tworzymy, dodając literę `u` (lub `U`) po stałej:

```
0x00ffU
```

Zapisując stałe, możemy łączyć litery `u` (lub `U`) i `l` (lub `L`), zatem

```
20000UL
```

oznacza stałą o wartości 20 000 zapisaną jako typ `unsigned long`.

Stała liczba całkowita, za którą nie występuje żadna z liter `u`, `U`, `l` ani `L` i która nie jest zbyt duża, aby przekroczyć zakres liczb `int`, traktowana jest przez kompilator jako liczba `unsigned int`. Jeśli jest zbyt duża, aby zmieścić się w zakresie liczb `unsigned int`, kompilator traktuje ją jako `long int`. Kiedy także w tym zakresie się nie mieści, liczba traktowana jest jako `unsigned long int`; gdy i to nie wystarcza, traktowana jest jako `long long int` albo ostatecznie jako `unsigned long long int`.

W przypadku deklarowania zmiennych typów `long long int`, `long int`, `short int` oraz `unsigned int` można pominąć słowo kluczowe `int`. Wobec tego zmienną `licznik` typu `unsigned int` możemy zadeklarować, stosując instrukcję:

```
unsigned licznik;
```

Można też deklarować zmienne `char` jako `unsigned`.

Kwalifikator `signed` może zostać użyty, aby jawnie nakazać kompilatorowi traktowanie zmiennej jako wartości ze znakiem. Najczęściej stosuje się go przed deklaracją typu `char`; więcej o typach powiemy jeszcze w rozdziale 13.

Nie warto się przejmować, jeśli dotychczasowy opis określników (specyfikatorów) wydaje się nieco abstrakcyjny. W dalszej części książki wiele zagadnień zilustrujemy konkretnymi przykładami programów. W rozdziale 13. dokładnie zajmiemy się typami danych oraz ich konwersjami.

W tabeli 3.1 zestawiono podstawowe typy danych oraz kwalifikatory.

Tabela 3.1. Podstawowe typy danych

Typ	Przykłady stałych	Formanty funkcji printf
`char`	`'a'`, `'\n'`	`%c`
`_Bool`	`0`, `1`	`%i`, `%u`
`short int`	—	`%hi`, `%hx`, `%ho`
`unsigned short int`	—	`%hu`, `%hx`, `%ho`
`int`	`12`, `–97`, `0xFFE0`, `0177`	`%i`, `%x`, `%o`
`unsigned int`	`12u`, `100U`, `0XFFu`	`%u`, `%x`, `%o`
`long int`	`12L`, `–2001`, `0xffffL`	`%li`, `%lx`, `%lo`
`unsigned long int`	`12UL`, `100ul`, `0xffeeUL`	`%li`, `%lx`, `%lo`
`long long int`	`0xe5e5e5e5LL`, `500ll`	`%lli`, `%llx`, `%llo`
`unsigned long long int`	`12ull`, `0xffeeULL`	`%llu`, `%llx`, `%llo`
`float`	`12.34f`, `3.1e–5f`, `0x1.5p10`, `0x1P–1`	`%f`, `%e`, `%g`, `%a`
`double`	`12.24`, `3.1e–5`, `0x.1p3`	`%f`, `%e`, `%g`, `%a`
`long double`	`12.34l`, `3.1e–5l`	`%Lf`, `%Le`, `%Lg`

Użycie zmiennych

Początkowo programiści musieli pisać swoje programy w binarnych językach obsługiwanych komputerów. Instrukcje maszynowe trzeba było ręcznie kodować w formie liczb dwójkowych i dopiero wtedy można było przekazywać je do komputera. Co więcej, programiści musieli jawnie przypisywać miejsce w pamięci i potem odwoływać się do niego, podając konkretny adres fizyczny.

Obecnie języki programowania pozwalają skoncentrować się na rozwiązywaniu konkretnych problemów, zbędne stało się odwoływanie do kodów maszynowych czy adresów fizycznych w pamięci. Do zapisywania wyników obliczeń i potem odwoływania się

do nich można używać nazw symbolicznych — *nazw zmiennych*. Nazwa zmiennej może być tak dobierana, aby od razu było widać, jakiego typu wartość zawiera dana zmienna.

W rozdziale 2. używaliśmy kilku zmiennych do zapisywania wartości całkowitoliczbowych. W programie 2.4 na przykład zmienna `sum` zawierała wynik dodawania liczb 50 i 25.

Język C pozwala stosować inne typy zmiennych, nie tylko liczby całkowite. Jednak *przed* ich użyciem konieczne jest prawidłowe ich zadeklarowanie. Zmienne mogą zawierać liczby zmiennoprzecinkowe, znaki, a nawet *wskaźniki* do określonych miejsc w pamięci.

Zasady tworzenia nazw zmiennych są proste — nazwa musi zaczynać się literą lub podkreśleniem (_), dalej może być dowolna kombinacja liter (wielkich i małych), podkreśleń oraz cyfr od 0 do 9. Oto przykłady poprawnych nazw zmiennych:

```
sum
pieceFlag
i
J5x7
Liczba_ruchow
_sysflag
```

Niżej podajemy przykłady nazw zmiennych, które są niepoprawne.

`sum$value`	znak dolara, *$*, jest niedopuszczalny
`piece flag`	nie można używać spacji wewnątrz nazwy
`3Spencer`	nazwa zmiennej nie może zaczynać się cyfrą
`int`	`int` to słowo zarezerwowane

Nie można wykorzystać słowa `int` jako nazwy zmiennej, gdyż słowo to ma w języku C specjalne znaczenie, czyli jest to słowo zarezerwowane. Ogólna zasada jest taka, że nie można jako nazwy zmiennej używać żadnej nazwy mającej dla kompilatora C specjalne znaczenie. W dodatku A podano pełną listę nazw zarezerwowanych.

Zawsze trzeba pamiętać, że w języku C ma znaczenie wielkość liter. Wobec tego nazwy zmiennych `sum`, `Sum` i `SUM` odnoszą się do różnych zmiennych. Nazwy zmiennych mogą być dowolnie długie, ale znaczenie mają tylko pierwsze 63 znaki, a w pewnych sytuacjach, opisanych w dodatku A, nawet tylko pierwsze 31 znaków. Zwykle nie zaleca się stosowania długich nazw po prostu dlatego, że wymagają one zbyt wiele pisania. Poniższy zapis jest wprawdzie poprawny:

```
ilePieniedzyZarobilemOdPoczatkuTegoRoku = pieniadzeNaKoniecTegoRoku –
       pieniadzeNaPoczatkuTegoRoku;
```

jednak równoważny mu wiersz:

```
tegoroczniePieniadze = pieniadzeNaKoniec – pieniadzePoczatkowo;
```

jest równie czytelny, a znacznie krótszy.

Dobierając nazwy zmiennych, musimy pamiętać o jednym — nie wolno zanadto folgować swemu lenistwu. Nazwa zmiennej powinna odzwierciedlać jej przeznaczenie. To zalecenie jest oczywiste — zarówno komentarze, jak i dobrze dobrane nazwy zmiennych mogą znakomicie poprawić czytelność programu i ułatwić usuwanie z niego błędów, a dodatkowo

stanowią dokumentację. Ilość wymaganej dokumentacji znacząco się zmniejsza, jeśli program jest czytelny sam w sobie.

Wyrażenia arytmetyczne

W języku C, tak jak chyba w każdym innym języku programowania, znak plus (+) służy do dodawania dwóch wartości, znak minus (–) do odejmowania, gwiazdka (*) to mnożenie, a ukośnik (/) oznacza dzielenie. Operatory te nazywamy *binarnymi* operatorami arytmetycznymi, gdyż działają na dwóch czynnikach.

Widzieliśmy już, jak łatwo w języku C dodawać liczby. Program 3.2 pokazuje jeszcze odejmowanie, mnożenie i dzielenie. Ostatnie dwa działania prowadzą do pojęcia *priorytetu* operatora. Każdy operator języka C ma jakiś priorytet używany do określania, jak należy wyliczać wyrażenie zawierające więcej niż jeden operator — najpierw wyliczane są operatory z wyższym priorytetem. Wyrażenia zawierające operatory o takim samym priorytecie są obliczane od lewej do prawej lub od prawej do lewej — w zależności od tego, jakich operatorów użyto, a dokładniej od ich *łączności*. Pełną listę priorytetów operatorów i zasady ich łączności podano w dodatku A.

Program 3.2. **Użycie operatorów arytmetycznych**

```
// Ilustracja działania różnych operatorów arytmetycznych

#include <stdio.h>

int main (void)
{
    int a = 100;
    int b = 2;
    int c = 25;
    int d = 4;
    int result;

    result = a - b;         // odejmowanie
    printf ("a - b = %i\n", result);

    result = b * c;         // mnożenie
    printf ("b * c = %i\n", result);

    result = a / c;         // dzielenie
    printf ("a / c = %i\n", result);

    result = a + b * c;     // priorytety
    printf ("a + b * c = %i\n", result);

    printf ("a * b + c * d = %i\n", a * b + c * d);

    return 0;
}
```

Program 3.2. **Wyniki**

```
a – b = 98
b * c = 50
a / c = 4
a + b * c = 150
a * b + c * d = 300
```

Po zadeklarowaniu zmiennych całkowitych `a`, `b`, `c`, `d` i `result` program przypisuje zmiennej `result` wynik odejmowania `b` od `a`, następnie wyświetla wartości przy użyciu funkcji `printf`.

Następna instrukcja:

```
result = b * c;
```

powoduje wymnożenie `b` przez `c` i zapisanie iloczynu w zmiennej `result`; wynik znów jest wyświetlany za pomocą funkcji `printf`.

Następnie w programie używany jest operator dzielenia — ukośnik. Wynik dzielenia 100 przez 25, 4, pokazywany jest znowu przy użyciu funkcji `printf`.

W niektórych systemach próba dzielenia przez zero powoduje awaryjne zakończenie wykonywania programu[2]. Jeśli nawet program nie zakończy awaryjnie swojego działania, wyniki uzyskiwane z takich obliczeń są bezwartościowe.

W rozdziale 5. zobaczymy, jak można sprawdzić, czy nie mamy do czynienia z dzieleniem przez zero jeszcze przed samym podzieleniem. Jeśli wiadomo, że dzielnik jest zerem, można podjąć odpowiednie działania i uniknąć dzielenia.

Wyrażenie:

```
a + b * c
```

nie da w wyniku 2550 (102·25), ale odpowiednia funkcja `printf` pokaże 150. Wynika to stąd, że w języku C, tak jak w większości innych języków programowania, istnieją zasady określające kolejność wykonywania działań. W zasadzie wyrażenia są przetwarzane od strony lewej do prawej. Jednak mnożenie i dzielenie mają wyższy priorytet niż dodawanie i odejmowanie, więc wyrażenie:

```
a + b * c
```

zostanie zinterpretowane w języku C jako:

```
a + (b * c)
```

(tak samo jak w normalnej algebrze).

Jeśli chcemy zmienić kolejność wyliczania wyrażeń, możemy użyć nawiasów. Właśnie wyrażenie podane ostatnio jest zupełnie poprawnym wyrażeniem języka C. Wobec tego instrukcja:

```
result = a + (b * c);
```

---

[2] Dzieje się tak w przypadku kompilatora `gcc` w systemie Windows. W systemach Unix program może nie przerwać swojego działania, dając 0 w wyniku dzielenia liczby całkowitej przez zero i „nieskończoność” w przypadku dzielenia przez zero wartości `float`.

może zostać wstawiona do programu 3.2 i ten da taki sam wynik jak poprzednio. Jeśli jednak użyjemy instrukcji:

```
result = (a + b) * c;
```

zmienna `result` będzie miała wartość 2550, gdyż wartość zmiennej `a` (100) zostanie dodana do wartości `b` (2) przed mnożeniem przez `c` (25). Nawiasy mogą też być zagnieżdżane — wtedy wyrażenie wyliczane jest w kolejności od nawiasów najbardziej wewnętrznych. Aby uniknąć pomyłek, zwykle wystarczy sprawdzić, czy liczba nawiasów otwierających równa jest liczbie nawiasów zamykających.

Z ostatniej instrukcji programu 3.2 wynika, że całkiem poprawne jest przekazywanie do funkcji `printf` jako argumentu wyrażenia, bez konieczności przypisywania zmiennej wartości tego wyrażenia. Wyrażenie:

```
a * b + c * d
```

jest wyliczane, zgodnie z podanymi wcześniej zasadami, jako:

```
(a * b) + (c * d)
```

czyli:

```
(100 * 2) + (25 * 4)
```

Funkcja `printf` działa na uzyskanym wyniku, czyli 300.

Arytmetyka liczb całkowitych i jednoargumentowy operator minus

Program 3.3 służy utrwaleniu przekazanej wcześniej wiedzy, pokazuje też zasady arytmetyki całkowitoliczbowej.

Program 3.3. **Dalsze przykłady użycia operatorów arytmetycznych**

```
// Jeszcze trochę wyrażeń arytmetycznych

#include <stdio.h>

int main (void)
{
    int   a = 25;
    int   b = 2;

    float c = 25.0;
    float d = 2.0;

    printf ("6 + a / 5 * b = %i\n", 6 + a / 5 * b);
    printf ("a / b * b = %i\n", a / b * b);
    printf ("c / d * d = %f\n", c / d * d);
    printf ("-a = %i\n", -a);

    return 0;
}
```

Program 3.3. **Wyniki**

```
6 + a / 5 * b = 16
a / b * b = 24
c / d * d = 25.000000
—a = —25
```

Do deklaracji zmiennych typu `int` wstawiono dodatkowe spacje, aby wyrównać wszystkie deklarowane zmienne. Dzięki temu program jest czytelniejszy. Uważni czytelnicy spostrzegli też zapewne, że w pokazywanych dotąd programach każdy operator jest otoczony spacjami. Nie jest to niezbędne, lecz robione w celu poprawienia estetyki programu. Warto dodać kilka spacji, jeśli czytelność programu na tym zyska.

Wyrażenie z pierwszego wywołania funkcji `printf` w programie 3.3 wskazuje, jak istotne są priorytety operatorów. Wyznaczanie wartości tego wyrażenia odbywa się następująco:

1. Dzielenie ma wyższy priorytet od dodawania, więc wartość zmiennej `a` (25) jest dzielona najpierw przez 5. Wynik pośredni wynosi 5.
2. Mnożenie ma wyższy priorytet od dodawania, więc wynik pośredni 5 jest mnożony przez 2, wartość zmiennej `b`, co daje nowy wynik pośredni 10.
3. W końcu wykonywane jest dodawanie 6 i 10, co daje 16 jako ostateczny wynik.

W drugiej instrukcji `printf` pojawia się dodatkowa komplikacja. Można by się spodziewać, że dzielenie `a` przez `b` i następnie pomnożenie przez `b` powinno dać wartość `a`, czyli 25. Jednak uzyskany wynik to 24. Czyżby komputer gdzieś po drodze zgubił jeden bit? Prawdziwa przyczyna jest taka, że obliczenia robione są przy użyciu arytmetyki liczb całkowitych.

Spójrzmy jeszcze raz na deklaracje zmiennych `a` i `b` — obie są typu `int`. Kiedy wyliczane wyrażenie zawiera tylko dwie liczby całkowite, C korzysta z arytmetyki liczb całkowitych, zatem tracone są ewentualne części ułamkowe. Wobec tego, dzieląc wartość zmiennej `a` przez `b`, czyli dzieląc 25 przez 2, uzyskujemy wynik pośredni 12, a *nie* 12.5. Mnożenie tego wyniku pośredniego przez 2 daje ostatecznie 24. Trzeba pamiętać, że dzieląc przez siebie dwie liczby całkowite, zawsze uzyskamy liczbę całkowitą. Ponadto musimy pamiętać, że liczby nie są zaokrąglane, tylko część ułamkowa jest po prostu opuszczana. W efekcie wyniki dzielenia 12.01, 12.5 i 12.99 zostaną zamienione na 12.

W przedostatniej instrukcji `printf` z programu 3.3 widzimy, że uzyskamy wynik zgodny z oczekiwaniami, jeśli te same działania przeprowadzimy na liczbach zmiennoprzecinkowych.

Decyzję o typie zmiennych — `int` lub `float` — trzeba podjąć na podstawie tego, jak zamierzamy tej zmiennej używać. Jeśli jej część ułamkowa będzie zbędna, używamy zmiennych całkowitoliczbowych. Uzyskany program będzie zwykle działał szybciej. Jeśli jednak część ułamkowa będzie potrzebna, wybór jest jasny i pozostaje jedynie pytanie, czy użyć typu `float`, `double` czy `long double`. Odpowiedź zależy od tego, jaka dokładność jest potrzebna, oraz od wielkości przetwarzanych liczb.

W ostatniej instrukcji `printf` wartość zmiennej jest zanegowana przy użycia jednoargumentowego operatora minus. Operator *jednoargumentowy*, zgodnie

ze swoją nazwą, ma tylko jeden argument. Znak minus może pełnić tylko dwie różne funkcje — może być operatorem binarnym (dwuargumentowym) używanym do odejmowania dwóch liczb lub operatorem jednoargumentowym zwracającym przeciwieństwo liczby.

Jednoargumentowy operator minus ma priorytet wyższy od wszystkich innych operatorów arytmetycznych (wyjątkiem jest jednoargumentowy plus, o takim samym priorytecie). Wobec tego wyrażenie:

```
c = –a * b;
```

spowoduje wymnożenie –a przez b. W dodatku A znajduje się tabela z zestawieniem operatorów i ich priorytetów.

Operator modulo

Teraz omówimy operator modulo, oznaczany symbolem procenta — %. Na podstawie programu 3.4 spróbujmy zorientować się, jak działa ten operator.

Program 3.4. **Użycie operatora modulo**

```
// Operator modulo

#include <stdio.h>

int main (void)
{
    int a = 25, b = 5, c = 10, d = 7;

    printf("a = %i, b = %i, c = %i i d = %i\n", a, b, c, d);
    printf ("a %% b = %i\n", a % b);
    printf ("a %% c = %i\n", a % c);
    printf ("a %% d = %i\n", a % d);
    printf ("a / d * d + a %% d = %i\n",
                a / d * d + a % d);

    return 0;
}
```

Program 3.4. **Wyniki**

```
a = 25, b = 5, c = 10 i d = 7
a % b = 0
a % c = 5
a % d = 4
a / d * d + a % d = 25
```

W pierwszej instrukcji w funkcji `main` definiujemy i inicjalizujemy jednocześnie cztery zmienne: `a`, `b`, `c` i `d`.

Dla przypomnienia: przed instrukcjami drukującymi operator modulo pierwsza instrukcja `printf()` drukuje wartości czterech zmiennych zdefiniowanych w programie. Nie jest to konieczne, ale pomaga innym programistom zrozumieć kod. Co się zaś tyczy

pozostałych wywołań funkcji `printf()`, jak już wiemy, funkcja `printf` wykorzystuje znaki znajdujące się za znakiem procentu jako definicje wyświetlanych wartości. Jeśli jednak za jednym znakiem procentu występuje drugi taki sam znak, funkcja `printf` traktuje ten procent jako zwykły znak do wyświetlenia.

Operator `%` zwraca resztę z dzielenia pierwszej wartości przez drugą. W pierwszym przykładzie resztą dzielenia 25 przez 5 jest 0. Jeśli podzielimy 25 przez 10, otrzymamy resztę 5 — widać to w drugim wierszu wyniku. Dzieląc 25 przez 7, otrzymujemy resztę 4, co wynika z trzeciego wiersza.

Ostatni wiersz wynikowy programu 3.4 wymaga pewnego wyjaśnienia. Najpierw zauważmy, że odpowiednia instrukcja została zapisana w dwóch wierszach. W języku C jest to jak najbardziej dopuszczalne. Instrukcja może być przeniesiona do następnego wiersza — podział możliwy jest wszędzie tam, gdzie może wystąpić spacja (wyjątkiem są łańcuchy znakowe, które dokładnie będziemy omawiać w rozdziale 9.). Czasami dzielenie programu na kilka wierszy może być nie tylko przydatne, lecz wprost niezbędne. W programie 3.4 przeniesiona do następnego wiersza część instrukcji `printf` jest wcięta, aby podział instrukcji na części był dobrze widoczny.

Zwróćmy uwagę na wyrażenie obliczane w ostatniej instrukcji. Przypomnijmy, że wszystkie obliczenia na liczbach całkowitych wykonywane są zgodnie z arytmetyką całkowitoliczbową, zatem reszta uzyskiwana z dzielenia dwóch liczb całkowitych jest po prostu odrzucana. Wobec tego dzielenie 25 przez 7, wynikające z wyrażenia `a / d`, daje wynik pośredni 3. Mnożąc tę wartość przez `d`, czyli 7, uzyskamy pośredni wynik 21. Dodając resztę z dzielenia `a` przez `d` (wyrażenie `a % d`), otrzymujemy ostatecznie 25. Nie jest przypadkiem, że jest to ta sama wartość, którą początkowo miała zmienna `a`. Ogólnie rzecz biorąc, wyrażenie:

```
a / b * b + a % b
```

zawsze da wartość `a`, jeśli `a` i `b` są liczbami całkowitymi. Operator modulo może być używany tylko z liczbami całkowitymi.

Operator modulo ma taki sam priorytet jak operatory mnożenia i dzielenia. Oznacza to oczywiście, że wyrażenie:

```
tablica + wartosc % WIELKOSC_TABLICY
```

będzie wyliczane jako:

```
tablica + (wartosc % WIELKOSC_TABLICY)
```

Konwersje między liczbami całkowitymi a zmiennoprzecinkowymi

Aby w języku C pisać dobrze działające programy, trzeba zrozumieć zasady niejawnej konwersji między liczbami zmiennoprzecinkowymi a całkowitymi. W programie 3.5 pokazano niektóre rodzaje konwersji. Warto wiedzieć, że niektóre kompilatory mogą generować ostrzeżenia o realizowanych konwersjach.

Program 3.5. **Konwersje między liczbami całkowitymi a zmiennoprzecinkowymi**

```
// Najprostsze konwersje typów w języku C

# include <stdio.h>

int main (void)
{
    float f1 = 123.125, f2;
    int   i1, i2 = -150;
    char c = 'a';

    i1 = f1;              // konwersja typu float na int
    printf ("%f przypisane zmiennej typu int daje %i\n", f1, i1);

    f1 = i1;              // konwersja typu int na float
    printf ("%i przypisane zmiennej typu float daje %f\n", i2, f1);

    f1 = i2 / 100;        // dzielenie przez siebie dwóch liczb int
    printf ("%i dzielone przez 100 daje %f\n", i2, f1);

    f2 = i2 / 100.0;          // dzielenie liczby int przez float
    printf ("%i dzielone przez 100.0 daje %f\n", i2, f2);

    f2 = (float) i2 / 100;  // operator rzutowania typów
    printf ("(float) %i dzielone przez 100 daje %f\n", i2, f2);

    return 0;
}
```

Program 3.5. **Wyniki**

```
123.125000 przypisane zmiennej typu int daje 123
-150 przypisane zmiennej typu float daje -150.000000
-150 dzielone przez 100 daje -1.000000
-150 dzielone przez 100.0 daje -1.500000
(float) -150 dzielone przez 100 daje -1.500000
```

Kiedy w języku C zmiennej całkowitoliczbowej przypisujemy liczbę zmiennoprzecinkową, część ułamkowa jest odrzucana. Kiedy zatem w powyższym programie do zmiennej `i1` przypisujemy wartość zmiennej `f1`, z liczby 123,125 *odrzucana* jest część ułamkowa, czyli w zmiennej `i1` znajduje się wartość 123. Widać to w pierwszym wierszu wynikowym.

Przypisanie zmiennej całkowitej do zmiennej zmiennoprzecinkowej nie powoduje żadnej zmiany wartości; wartość ta jest po prostu konwertowana przez system i zapisywana w odpowiedniej zmiennej. W drugim wierszu wynikowym widać, że wartość `i2` (–150) została prawidłowo skonwertowana i zapisana w zmiennej `f1` jako `float`.

W następnych dwóch wierszach wynikowych pokazano, o czym trzeba pamiętać przy zapisywaniu wyrażeń arytmetycznych. Po pierwsze, chodzi o arytmetykę całkowitoliczbową, która była już omawiana w tym rozdziale. Kiedy oba operandy wyrażenia są liczbami całkowitymi (czyli chodzi o typy `short`, `unsigned`, `long` i `long long`), działanie jest wykonywane zgodnie z zasadami arytmetyki całkowitoliczbowej.

Wobec tego wszystkie części ułamkowe powstające po dzieleniu są odrzucane, nawet jeśli wynik zapiszemy potem jako wartość zmiennoprzecinkową (jak w naszym programie). Wobec tego, kiedy zmienna całkowitoliczbowa `i2` jest dzielona przez stałą całkowitą `100`, system wykonuje dzielenie całkowitoliczbowe. Wynikiem dzielenia –150 przez 100 jest –1; taka wartość jest zapisywana w zmiennej `f1` typu `float`.

Następne dzielenie powyższego programu zawiera zmienną całkowitoliczbową i stałą zmiennoprzecinkową. W języku C działanie jest traktowane jako zmiennoprzecinkowe, jeśli któryś z argumentów działania jest zmiennoprzecinkowy. Wobec tego dzielenie `i2` przez `100.0` system traktuje jako dzielenie zmiennoprzecinkowe, zatem uzyskujemy wartość –1,5, przypisywaną zmiennej `f1` typu `float`.

Operator rzutowania typów

Ostatnie dzielenie z programu 3.5 mające postać:

```
f2 = (float) i2 / 100; // operator rzutowania typów
```

zawiera nowy operator — operator rzutowania. Operator ten powoduje konwersję zmiennej `i2` na typ `float` na potrzeby pokazanego wyrażenia. Nie wpływa jednak w trwały sposób na zmienną `i2`; jest to jednoargumentowy operator zachowujący się tak jak wszystkie inne operatory. Wyrażenie –a nie ma żadnego trwałego wpływu na wartość `a`, tak samo trwałego wpływu nie ma wyrażenie `(float) a`.

Operator rzutowania typów ma priorytet wyższy niż wszelkie operatory arytmetyczne poza jednoargumentowymi plusem i minusem. Oczywiście w razie potrzeby można użyć nawiasów, aby wymusić pożądaną kolejność obliczeń. Innym przykładem użycia operatora rzutowania jest wyrażenie:

```
(int) 29.55 + (int) 21.99
```

interpretowane w języku C jako:

```
29 + 21
```

— wynika to stąd, że rzutowanie wartości zmiennoprzecinkowych na liczby całkowite polega na odrzuceniu części ułamkowej. Z kolei wyrażenie:

```
(float) 6 / (float) 4
```

da wynik 1.5, podobnie jak wyrażenie:

```
(float) 6 / 4
```

Łączenie działań z przypisaniem

Język C pozwala łączyć działania arytmetyczne z operatorem przypisania; używamy do tego ogólnej postaci *op*=. W zapisie tym *op* to jeden z operatorów arytmetycznych (+, –, *, / i %). Poza tym *op* może być jednym z operatorów działań i przesunięć bitowych, które omówimy później.

Weźmy pod uwagę instrukcję:

```
ilosc += 10;
```

Wynik działania operatora += będzie taki, że znajdujące się po jego prawej stronie wyrażenie zostanie dodane do wyrażenia z jego lewej strony, a wynik zostanie przypisany wyrażeniu z lewej strony operatora. Wobec tego powyższa instrukcja jest równoważna instrukcji:

```
ilosc = ilosc + 10;
```

Wyrażenie:

```
licznik —= 5
```

powoduje odjęcie od zmiennej `licznik` wartości 5 i zapisanie wyniku z powrotem w zmiennej `licznik`; jest równoważne wyrażeniu:

```
licznik = licznik — 5
```

Nieco bardziej skomplikowane jest wyrażenie:

```
a /= b + c
```

które dzieli a przez wyrażenie po prawej stronie; w tym wypadku sumę b i c, a następnie iloraz przypisuje zmiennej a. Dodawanie jest wykonywane jako pierwsze, gdyż operator dodawania ma wyższy priorytet od operatora przypisania. Zresztą wszystkie operatory — poza przecinkiem — mają priorytety wyższe od operatorów przypisania; z kolei wszystkie operatory przypisania mają taki sam priorytet.

Pokazane wcześniej wyrażenie będzie zatem równoważne następującemu:

```
a = a / (b + c)
```

Pokazanych operatorów przypisania używamy z trzech powodów. Po pierwsze, łatwiejsze jest zapisywanie instrukcji programu, gdyż to, co jest po lewej stronie operatora przypisania, nie musi już być powtarzane po jego prawej stronie. Po drugie, wyrażenie tak uzyskane jest łatwiejsze do czytania. Po trzecie, użycie takich operatorów przypisania może przyspieszyć działanie programów, gdyż czasami kompilator generuje nieco mniej kodu do wyliczania wyrażeń.

Typy _Complex i _Imaginary

Zanim przejdziemy do następnego rozdziału, warto odnotować istnienie jeszcze dwóch typów, `_Complex` i `_Imaginary`, używanych do zapisu liczb zespolonych i urojonych.

Obsługę typów `_Complex` i `_Imaginary` wprowadzono w standardzie ANSI C99, chociaż w C11 nie jest ona obowiązkowa. Jeśli chcesz dowiedzieć się, czy dany kompilator obsługuje te typy, zajrzyj do zestawienia typów danych w dodatku A.

Ćwiczenia

1. Przepisz i uruchom pięć programów pokazanych w tym rozdziale. Uzyskane wyniki porównaj z wynikami pokazanymi w tekście.
2. Które z poniższych nazw zmiennych są nieprawidłowe i dlaczego?

```
Int            char      6_05
Calloc         Xx        alpha_beta_routine
floating       _1312     z
ReInitialize   _         A$
```

3. Które z poniższych stałych są nieprawidłowe i dlaczego?

```
123.456     0x10.5    OXOG1
0001        0xFFFF    123L
0Xab05      0L        —597.25
123.5e2     .0001     +12
98.6F       98.7U     17777s
0996        —12E—12   07777
1234uL      1.2Fe—7   15,000
1.234L      197u      100U
0XABCDEFL   0xabcdu   +123
```

4. Napisz program przeliczający 27° ze skali Fahrenheita (*F*) na skalę Celsjusza (*C*). Użyj następującej zależności:

```
C = (F — 32) / 1.8
```

5. Jaki wynik da następujący program?

```
#include <stdio.h>

int main (void)
{
    char c, d;

    c = 'd';
    d = c;
    printf ("d = %c\n", d);

    return 0;
}
```

6. Napisz program wyliczający wartość wielomianu:

$$3x^3–5x^2+6$$

dla $x = 2{,}55$.

7. Napisz program wyznaczający wartość poniższego wyrażenia i pokazujący wyniki (pamiętaj o użyciu zapisu wykładniczego przy wyświetlaniu wyników):

$$(3{,}31\cdot10^{-8}\cdot2{,}01\cdot10^{-7}) / (7{,}16\cdot10^{-6}+2{,}01\cdot10^{-8})$$

8. Aby zaokrąglić liczbę całkowitą `i` do najbliższej wielokrotności innej liczby całkowitej `j`, możesz użyć wzoru:

```
Nastepna_wielokrotnosc = i + j — i % j
```

Aby na przykład zaokrąglić 256 dni do najbliższej liczby dni dzielącej się na pełne tygodnie, mamy `i` = 256 i `j` = 7, więc z powyższego wzoru otrzymujemy:

```
Nastepna_wielokrotnosc   = 256 + 7 – 256 % 7
                         = 256 + 7 – 4
                         = 259
```

9. Napisz program znajdujący najbliższe wielokrotności dla następujących wartości `i` i `j`:

`i`	`j`
365	7
12,258	23
996	4

4

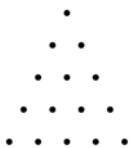
Pętle w programach

Jedną z wielkich zalet komputerów jest możliwość wielokrotnego wykonywania obliczeń. Program w języku C może zawierać kilka konstrukcji specjalnie przeznaczonych do użycia, gdy trzeba coś wykonać więcej niż raz. W tym rozdziale nauczysz się posługiwać następującymi narzędziami:

- instrukcją `for`,
- instrukcją `while`,
- instrukcją `do`,
- instrukcją `break`,
- instrukcją `continue`.

Liczby trójkątne

Gdybyśmy mieli ułożyć 15 kropek w kształcie trójkąta, moglibyśmy utworzyć na przykład wzór taki jak poniżej:



Pierwszy wiersz zawiera jedną kropkę, drugi dwie i tak dalej. Ogólnie liczba kropek potrzebnych do utworzenia trójkąta zawierającego *n* wierszy to suma liczb całkowitych od 1 do *n*. Suma taka nazywana jest *liczbą trójkątną*. Jeśli zaczynamy od jedynki, czwarta liczba trójkątna, czyli 10, jest sumą kolejnych liczb od 1 do 4 (1 + 2 + 3 + 4 = 10).

Załóżmy, że chcemy napisać program wyliczający i pokazujący ósmą liczbę trójkątną. Oczywiście łatwo odpowiednią liczbę wyliczyć w pamięci, ale załóżmy, że potrzebujemy do tego programu w języku C. Program 4.1 zawiera odpowiedni kod.

Program 4.1 dobrze sprawdza się w przypadku wyliczania dość małych liczb trójkątnych, ale co zrobić, jeśli będziemy potrzebować dwusetnej liczby trójkątnej? Bardzo trudno byłoby tak zmodyfikować program 4.1, aby dodawać w nim kolejne liczby całkowite od 1 do 200. Na szczęście istnieje prostsza metoda.

Program 4.1. **Wyliczanie ósmej liczby trójkątnej**

```
// Program wyliczający ósmą liczbę trójkątną.

#include <stdio.h>

int main (void)
{
    int triangularNumber;

    triangularNumber = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8;

    printf ("Ósma liczba trójkątna to %i\n",
                triangularNumber);

    return 0;
}
```

Program 4.1. **Wyniki**

```
Ósma liczba trójkątna to 36
```

Jedną z najważniejszych cech komputera jest możliwość wielokrotnego wykonywania zestawu instrukcji, co nazywamy realizacją *pętli*. Dzięki temu można w zwarty sposób zapisywać powtarzalne działania, które normalnie wymagałyby tysięcy, czy nawet milionów, instrukcji. Język programowania C zawiera trzy różne instrukcje do realizacji pętli. Są to instrukcja `for`, instrukcja `while` oraz instrukcja `do`. Wszystkie te instrukcje szczegółowo opiszemy dalej.

Instrukcja for

Zacznijmy od programu, który wykorzystuje instrukcję `for`. Celem działania programu 4.2 jest wyliczenie dwusetnej liczby trójkątnej. Widzimy tutaj, jak działa instrukcja `for`.

Program 4.2. **Wyliczanie dwusetnej liczby trójkątnej**

```
/* Program wyliczający dwusetną liczbę trójkątną.
   Wstęp do instrukcji for.                    */

#include <stdio.h>

int main (void)
{
    int n, triangularNumber;
```

```
    triangularNumber = 0;
    for ( n = 1; n <= 200; n = n + 1 )
        triangularNumber = triangularNumber + n;

    printf ("Dwusetna liczba trójkątna to %i\n", triangularNumber);

    return 0;
}
```

Program 4.2. **Wyniki**

```
Dwusetna liczba trójkątna to 20100
```

Program 4.2 wymaga pewnego wyjaśnienia. W zasadzie dwusetną liczbę trójkątną wyliczamy tak, jak obliczaliśmy liczbę ósmą w programie 4.1 — sumowane są liczby od 1 do 200. Instrukcja `for` stanowi mechanizm, który pozwala na uniknięcie podawania każdej liczby całkowitej od 1 do 200. W pewnym sensie instrukcji `for` używamy do „wygenerowania" wszystkich potrzebnych liczb.

Ogólna postać instrukcji `for` jest następująca:

```
for ( wyrażenie_początkowe; warunek_pętli; wyrażenie_pętli )
    instrukcja (lub instrukcje)
```

Trzy wyrażenia ujęte w nawiasy — *wyrażenie_początkowe*, *warunek_pętli* oraz *wyrażenie_pętli* — ustalają środowisko, w jakim działa pętla. Pojawiająca się dalej instrukcja (zakończona oczywiście średnikiem) może być dowolną instrukcją języka C, która tworzy treść pętli. Instrukcja ta jest wykonywana tyle razy, na ile wskazują parametry sterujące pętlą.

Pierwszy składnik instrukcji `for`, *wyrażenie_początkowe*, służy do ustalenia wartości początkowych *przed* rozpoczęciem wykonywania pętli. W programie 4.2 ta część instrukcji `for` służy do ustawienia wartości początkowej zmiennej `n` na 1. Jak widać, przypisanie jest wyrażeniem jak najbardziej poprawnym.

Drugi składnik instrukcji `for` to warunek lub warunki, które muszą być spełnione, aby wykonywanie pętli było kontynuowane. Innymi słowy, pętla jest realizowana *tak długo, aż* warunki te zostaną spełnione. Spójrzmy jeszcze raz na program 4.2 — warunkiem pętli jest porównanie:

```
n <= 200
```

Wyrażenie czyta się tak: „`n` jest mniejsze bądź równe 200". Operator „mniejsze bądź równe" (składający się z dwóch znaków, mniejsze „< " i tuż za nim równa się „=") jest jednym z kilku operatorów porównania występujących w języku C. Operatory te służą do sprawdzania, czy zachodzą pewne warunki. Jeśli warunek jest spełniony, odpowiedź jest twierdząca (prawda); jeśli warunek nie jest spełniony, odpowiedź jest przecząca (fałsz).

Operatory porównania

W tabeli 4.1 zestawiono wszystkie operatory porównania występujące w języku C.

Tabela 4.1. Operatory porównania

Operator	Znaczenie	Przykład użycia
==	równe	`liczba == 10`
!=	różne	`flaga != GOTOWE`
<	mniejsze	`a < b`
<=	mniejsze lub równe	`niski <= wysoki`
>	większe	`wskaznik > koniec_listy`
>=	większe lub równe	`j >= 0`

Operatory porównania mają niższy priorytet niż operatory arytmetyczne. Oznacza to, że na przykład poniższe wyrażenie:

```
a < b + c
```

będzie, zgodnie z intuicją, interpretowane jako:

```
a < (b + c)
```

Wyrażenie to będzie prawdziwe (`TRUE`), jeśli wartość `a` będzie mniejsza od sumy wartości `b` i `c`; w przeciwnym wypadku wyrażenie to będzie fałszywe (`FALSE`).

Szczególną uwagę trzeba zwrócić na operator równości — „==”, aby nie mylić go z operatorem przypisania — „=”. Wyrażenie:

```
a == 2
```

sprawdza, czy wartością `a` jest `2`, z kolei wyrażenie:

```
a = 2
```

powoduje przypisanie zmiennej `a` wartości `2`.

Wybór konkretnego operatora zależy od tego, co chcemy sprawdzić, a w pewnych warunkach może też zależeć od osobistych preferencji, na przykład wyrażenie:

```
n <= 200
```

można równie dobrze wyrazić jako[1]:

```
n < 201
```

Wróćmy teraz do naszego przykładu. Treść pętli `for` tworzy instrukcja:

```
triangularNumber = triangularNumber + n;
```

wykonywana wielokrotnie *tak długo, jak warunek jest spełniony*, czyli w naszym wypadku tak długo, jak długo wartość `n` jest mniejsza bądź równa `200`. Nasza instrukcja powoduje

[1] Jest tylko jeden warunek — `n` jest liczbą całkowitą — *przyp. tłum.*

dodanie do zmiennej `triangularNumber` wartości `n` i zapisanie wyniku z powrotem w zmiennej `triangularNumber`.

Kiedy *`warunek_pętli`* przestaje być spełniony, program jest wykonywany dalej od pierwszej instrukcji znajdującej się zaraz za pętlą. W naszym wypadku jest to instrukcja `printf`.

Ostatni element instrukcji `for` to wyrażenie wyliczane po każdym wykonaniu treści pętli. W programie 4.2 to wyrażenie powoduje dodanie jedynki do wartości zmiennej `n`. Wobec tego wartość `n` jest zwiększana o 1 po jej dodaniu do zmiennej `triangularNumber`; zmienia się ona od 1 do 201.

Trzeba też dodać, że ostatnia wartość, jaką przybiera `n` — czyli 201 — już *nie jest dodawana* do zmiennej `triangularNumber`, gdyż pętla kończy się, jak tylko warunek pętli przestaje być spełniony, czyli kiedy tylko `n` osiąga wartość 201.

Instrukcja `for` działa więc następująco.

1. Najpierw wyliczane jest wyrażenie początkowe. Zwykle ustawia ono używaną dalej w pętli zmienną — którą nazywamy zmienną *indeksującą* — na pewną wartość, na przykład 0 lub 1.
2. Sprawdzany jest warunek pętli. Jeśli nie jest spełniony (wyrażenie jest fałszywe), pętla jest kończona. Wykonywana jest instrukcja znajdująca się zaraz za pętlą.
3. Wykonywana jest instrukcja stanowiąca treść pętli.
4. Wyliczane jest wyrażenie pętli. Wyrażenie to jest używane do zmiany wartości zmiennej indeksowej i często polega na dodaniu lub odjęciu od niej jedynki.
5. Następuje powrót do kroku 2.

Pamiętajmy, że warunek pętli jest wyliczany natychmiast po wejściu do pętli, przed pierwszym wykonaniem jej treści. Pamiętajmy też, aby po nawiasie zamykającym pętlę nie umieszczać średnika (to spowodowałoby natychmiastowe zakończenie działania pętli).

Ponieważ program 4.2 tak naprawdę generuje 200 pierwszych liczb trójkątnych, warto pokazać tablicę tych liczb. Aby jednak nie marnować zbyt wiele miejsca, pokażemy tylko 10 początkowych takich liczb. Program 4.3 realizuje właśnie to zadanie.

Program 4.3. **Generowanie tablicy liczb trójkątnych**

```
// Program generujący tablicę liczb trójkątnych

#include <stdio.h>

int main (void)
{
    int n, triangularNumber;

    printf ("TABLICA LICZB TRÓJKĄTNYCH\n\n");
    printf ("n    Suma od 1 do n\n");
    printf ("---  --------------\n");

    triangularNumber = 0;
    for ( n = 1; n <= 10; n = ++1 ) {
```

```
        triangularNumber += n;
        printf (" %i          %i\n"|, n, triangularNumber);
    }

    return 0;
}
```

Program 4.3. **Wyniki**

```
TABLICA LICZB TRÓJKĄTNYCH

 n   Suma od 1 do n
---  --------------
 1            1
 2            3
 3            6
 4            10
 5            15
 6            21
 7            28
 8            36
 9            45
 10            55
```

Zawsze warto dodać kilka instrukcji `printf`, aby program lepiej objaśniał pokazywane wyniki. W programie 4.3 pierwsze trzy instrukcje `printf` służą do pokazania nagłówka oraz etykiet kolumn wyniku. Zauważmy, że pierwsza instrukcja `printf` zawiera dwa znaki nowego wiersza. Jak można oczekiwać, powoduje to nie tylko przejście do nowego wiersza, ale też wstawienie jednego wiersza pustego.

Po wyświetleniu prawidłowych nagłówków program przechodzi do wyliczenia pierwszych 10 liczb trójkątnych. Zmienna `n` służy do rejestracji aktualnej wartości, dla której liczymy sumę od `1` do `n`. Zmienna `triangularNumber` to wartość samej liczby trójkątnej.

Wykonanie instrukcji `for` zaczyna się od ustawienia wartości zmiennej `n` na 1. Jak pamiętamy, instrukcja znajdująca się bezpośrednio za wierszem zawierającym `for` tworzy treść pętli. Ale co się stanie, jeśli będziemy chcieli wykonać nie jedną instrukcję, ale ich grupę? Wystarczy taką grupę instrukcji zamknąć w parę nawiasów klamrowych. Wtedy system traktuje taką grupę, nazywaną *blokiem*, jak pojedynczą instrukcję. Ogólnie rzecz biorąc, wszędzie tam, gdzie w języku C możemy wstawić pojedynczą instrukcję, możemy wstawić też blok instrukcji; musimy tylko pamiętać o otoczeniu go nawiasami klamrowymi.

Wobec tego w programie 4.3 treść pętli tworzą dwie instrukcje — instrukcja dodająca `n` do zmiennej `triangularNumber` oraz instrukcja `printf`. Szczególną uwagę zwróćmy na sposób ich zapisu, jak są wcięte. Odpowiedni układ wierszy sprawia, że łatwo sprawdzić, które instrukcje należą do pętli `for`. Trzeba jednak pamiętać, że różni programiści stosują odmienne sposoby wcinania kodu. Niektórzy wolą pisać:

```
for ( n = 1; n <= 10; n = ++1 )
{
    triangularNumber += n;
    printf (" %i          %i\n"|, n, triangularNumber);
}
```

W tym fragmencie kodu klamrowy nawias otwierający znajduje się w wierszu poniżej `for`; rozmieszczenie nawiasów jest kwestią gustu i nie ma żadnego wpływu na działanie programu.

Następnie liczba trójkątna jest wyliczana przez dodanie wartości zmiennej `n` do poprzedniej liczby trójkątnej. Tym razem wykorzystano operator dodawania jednocześnie z przypisaniem, który omawialiśmy w rozdziale 3. Przypomnijmy, że wyrażenie:

```
triangularNumber += n;
```

jest równoważne wyrażeniu:

```
triangularNumber = triangularNumber + n;
```

Przy pierwszym przejściu pętli `for` „poprzednią" liczbą trójkątną jest zero, więc nowa wartość `triangularNumber`, kiedy `n` jest równe 1, to po prostu wartość `n`, czyli też 1. Dalej wyświetlane są wartości zmiennych `n` i `triangularNumber`, a dookoła nich są wstawiane spacje, aby uzyskać ułożenie tych wartości pod nagłówkami.

Wobec tego, że wykonano treść pętli, wyliczane jest wyrażenie pętli. W tym wypadku wygląda ono nieco dziwnie — jakby ktoś zrobił pomyłkę w pisaniu i zamiast:

```
n = n + 1
```

napisał:

```
++n
```

Jednak wyrażenie `++n` jest jak najbardziej prawidłowym wyrażeniem języka C. Zawiera ono nowy (i raczej specyficzny dla C) *operator inkrementacji*. Operator ten, składający się z dwóch plusów, powoduje dodanie do swojego argumentu jedynki. Operator ten utworzono z uwagi na to, że w programach bardzo często zdarza się dodawanie jedynki. Zatem wyrażenie `++n` jest równoważne wyrażeniu `n = n + 1`. Mogłoby wydawać się, że to drugie jest czytelniejsze, ale po nabyciu wprawy większość osób docenia zwartość `++n`.

Jeśli powiedzieliśmy A, to trzeba powiedzieć B, więc naturalnym oczekiwaniem jest istnienie analogicznego operatora odejmującego 1. Ten operator nazywamy *operatorem dekrementacji* i zapisujemy go za pomocą dwóch minusów: —. Zatem wyrażenie:

```
liczba_obiektow = liczba_obiektow — 1
```

możemy równie dobrze zapisać, stosując operator dekrementacji:

```
—liczba_obiektow
```

Niektórzy programiści wolą dopisywać operatory ++ i — nie przed, ale za nazwami zmiennych, na przykład `n++` czy `liczba_obiektow—`. W przedstawionej przykładowej pętli `for` umiejscowienie tych operatorów jest wyłącznie kwestią gustu, ale w rozdziale 10. opisuję sytuacje, w których to, gdzie znajduje się operator inkrementacji i dekrementacji, ma znaczenie.

Wyrównywanie wyników

W wynikach programu 4.3 nieco irytująca może być jedna rzecz — dziesiąta liczba trójkątna niedokładnie zmieściła się w swojej kolumnie. Wynika to stąd, że liczba 10 zajmuje dwa znaki, a wcześniejsze wartości `n`, od 1 do 9, tylko jeden. Wobec tego wartość 55 została „przepchnięta" o jedno miejsce. Można to poprawić, jeśli instrukcję `printf` z pętli programu 4.3 zastąpimy następującą jej wersją:

```
printf ("%2i          %i\n"|, n, triangularNumber);
```

Uzyskamy teraz taki wynik (zmodyfikowany w ten sposób program oznaczymy jako program 4.3A).

Program 4.3A. **Wyniki**

```
TABLICA LICZB TRÓJKĄTNYCH

 n   Suma od 1 do n
---  --------------
 1            1
 2            3
 3            6
 4            10
 5            15
 6            21
 7            28
 8            36
 9            45
10            55
```

Zmiana w tej instrukcji `printf` polega na wstawieniu *specyfikacji szerokości pola*. Zapis `%2i` przekazuje do funkcji `printf` informację, że chcemy wyświetlić liczbę całkowitą w danym miejscu, a dodatkowo chcemy, aby pokazywana wartość zajmowała dwa znaki. Jeśli normalnie liczba zajęłaby mniej niż dwa znaki (jak liczby od 0 do 9), zostanie wyświetlona ze *spacją wiodącą*; na tym właśnie polega *wyrównanie do prawej strony*.

Korzystając zatem ze specyfikacji szerokości pola w formie `%2i`, możemy zagwarantować, że wartość `n` będzie pokazywana przynajmniej w dwóch kolumnach, a zatem wartości zmiennej `triangularNumber` zostaną prawidłowo wyrównane.

Jeśli wyświetlenie wartości wymaga więcej znaków, niż podano w szerokości pola, funkcja `printf` pomija specyfikację tej szerokości i używa tylu znaków, ilu potrzeba.

Specyfikacji szerokości pola można używać także do wyświetlania wartości niebędących liczbami całkowitymi. Wkrótce pokażemy programy wykorzystujące ten mechanizm.

Dane wejściowe dla programu

Program 4.2 wylicza dwusetną liczbę trójkątną… i na tym koniec. Jeśli natomiast chcemy wyliczyć pięćdziesiątą lub setną liczbę trójkątną, musimy zmienić ten program tak, aby pętla `for` wykonywała się tyle razy, ile trzeba. Należy też zmienić instrukcję `printf`, aby pokazywała właściwy komunikat.

Łatwiej byłoby, gdybyśmy mogli jakoś przekazać programowi, którą liczbę trójkątną chcemy wyliczyć. W języku C możemy taką funkcję zrealizować, korzystając z procedury `scanf`. Procedura ta w swych zasadach jest bardzo podobna do procedury `printf`, ale gdy `printf` pokazuje wartości, to `scanf` umożliwia wpisywanie danych do programu. W programie 4.4 pytamy użytkownika, która liczba trójkątna ma być wyliczona, następnie program ją wylicza i pokazuje uzyskany wynik.

Program 4.4. **Przekazanie danych wejściowych przez użytkownika**

```
#include <stdio.h>

int main (void)
{
    int  n, number, triangularNumber;

    printf ("Jaką liczbę trójkątną chcesz obliczyć? ");
    scanf  ("%i", &number);

    triangularNumber = 0;

    for ( n = 1; n <= number; ++n )
        triangularNumber += n;

    printf ("Liczba trójkątna numer %i to %i\n", number, triangularNumber);

    return 0;
}
```

W wynikach programu 4.4 liczbę wprowadzoną przez użytkownika (100) pogrubiono, aby ją odróżnić od wyników wypisywanych przez program.

Program 4.4. **Wyniki**

```
Jaką liczbę trójkątną chcesz obliczyć? 100
Liczba trójkątna numer 100 to 5050
```

Jak widać, użytkownik wpisał 100. Następnie program wyliczył setną liczbę trójkątną i wyświetlił wynik 5050. Użytkownik równie dobrze mógłby napisać 10, 30 czy inną liczbę i uzyskałby odpowiednią liczbę trójkątną.

Pierwsza instrukcja `printf` z programu 4.4 służy do zadania użytkownikowi pytania o liczbę. Zawsze trzeba przypomnieć, jaką informację program musi otrzymać. Następnie wywoływana jest procedura `scanf`. Jej pierwszy argument to łańcuch formatujący, bardzo podobny do analogicznych łańcuchów z funkcji `printf`. W tym wypadku jednak nie informujemy systemu, jakie wartości będą wyświetlane, ale jakiego typu wartości należy odczytać. Podobnie jak w funkcji `printf`, znaki `%i` wskazują liczby całkowite.

Drugi argument funkcji `scanf` określa, *gdzie* wpisana wartość ma zostać umieszczona. Znak `&` przed zmienną `number` jest niezbędny, ale możemy się nim nie przejmować aż do rozdziału 10. o wskaźnikach, gdzie dokładnie omówimy znaczenie znaku (będącego zresztą operatorem). Jeśli o znaku `&` przed nazwą zmiennej zapomnimy, zachowanie programu będzie nieprzewidywalne i może on zostać przerwany w trybie awaryjnym.

Zatem funkcja `scanf` z programu 4.4 ma odczytać liczbę całkowitą i zapisać ją w zmiennej `number`. Liczba ta oznacza numer liczby trójkątnej, którą użytkownik chce wyliczyć.

Kiedy użytkownik wpisze liczbę i wciśnie klawisz *Enter*, program wylicza żądaną liczbę trójkątną. Robi to tak samo jak w programie 4.2; jedynie zamiast ograniczenia 200 stosowane jest ograniczenie `number`.

Uwaga

Naciśnięcie klawisza *Enter* na klawiaturze numerycznej może nie wystarczyć do wysłania liczby do programu. Do tego celu należy używać klawisza *Enter* znajdującego się w głównej części klawiatury.

Kiedy żądana liczba zostanie wyliczona, program pokazuje wynik i kończy swoje działanie.

Zagnieżdżone pętle for

Program 4.4 umożliwia użytkownikowi wyliczanie dowolnej liczby trójkątnej. Jeśli jednak użytkownik zechce wyliczyć pięć takich liczb, może uruchomić program pięciokrotnie, za każdym razem podając następną liczbę z listy.

Innym sposobem osiągnięcia tego samego celu jest utworzenie programu, który umożliwi wyliczenie wielu liczb trójkątnych (to rozwiązanie jest znacznie bardziej kształcące). W tym celu najlepiej wstawić do programu pętlę, która powtórzy wszystkie obliczenia pięciokrotnie. Wiemy, że pętlę taką można zrealizować za pomocą instrukcji `for`. W programie 4.5 pokazano, jak to zrobić.

Program 4.5. **Użycie zagnieżdżonych pętli for**

```
#include <stdio.h>

int main (void)
{
    int  n, number, triangularNumber, counter;

    for ( counter = 1; counter <= 5; ++counter ) {
        printf ("Jaką liczbę trójkątną chcesz obliczyć? ");
        scanf  ("%i", &number);

        triangularNumber = 0;

        for ( n = 1; n <= number; ++n )
            triangularNumber += n;

        printf ("Liczba trójkątna numer %i to %i\n\n", number, triangularNumber);
    }

    return 0;
}
```

Program 4.5. **Wyniki**

```
Jaką liczbę trójkątną chcesz obliczyć? 12
Liczba trójkątna numer 12 to 78

Jaką liczbę trójkątną chcesz obliczyć? 25
Liczba trójkątna numer 25 to 325

Jaką liczbę trójkątną chcesz obliczyć? 50
Liczba trójkątna numer 50 to 1275

Jaką liczbę trójkątną chcesz obliczyć? 75
Liczba trójkątna numer 75 to 2850

Jaką liczbę trójkątną chcesz obliczyć? 83
Liczba trójkątna numer 83 to 3486
```

Program ten ma dwa poziomy instrukcji `for`. Zewnętrzna instrukcja `for`:

```
for ( counter = 1; counter <= 5; ++counter )
```

wskazuje, że pętla ma być wykonana pięć razy. Wynika to stąd, że zmienna `counter` ma początkowo wartość `1` i jest inkrementowana o 1, *dopóki* jest mniejsza bądź równa `5` (czyli dopóki nie osiągnie wartości 6).

W przeciwieństwie do poprzednich przykładów, zmienna `counter` nie jest używana w żadnym innym miejscu programu. Służy jedynie do zorganizowania pętli. Jednak dlatego, że jest to zmienna, musi zostać w programie zadeklarowana.

Pętla zawiera wszystkie pozostałe instrukcje występujące w programie, co wynika z użytych nawiasów klamrowych. Łatwiej byłoby zrozumieć działanie programu, gdybyśmy go „streścili”:

```
pięć razy
{
    Pobierz od użytkownika liczbę.

    Wylicz żądaną liczbę trójkątną.

    Pokaż wynik.
}
```

Część pętli zapisana wyżej jako *Wylicz żądaną liczbę trójkątną* składa się z ustawienia wartości zmiennej `triangularNumber` na zero oraz z pętli `for` wyliczającej liczbę trójkątną. Tak więc mamy instrukcję `for`, która jest zawarta *wewnątrz* innej pętli `for`. Jest to w języku C konstrukcja jak najbardziej poprawna; mało tego, zagnieżdżanie można kontynuować i można utworzyć dowolną liczbę poziomów!

Kiedy mamy do czynienia z bardziej zaawansowanymi konstrukcjami programistycznymi, takimi jak zagnieżdżone pętle `for`, stosowanie prawidłowych wcięć staje się ogromnie ważne, gdyż umożliwia proste sprawdzenie, co która pętla zawiera. Aby zrozumieć, jak nieczytelny będzie program, jeśli zaniedbamy wcięcia, wystarczy zajrzeć do ćwiczenia 5. z końca tego rozdziału.

Odmiany pętli for

Przy tworzeniu pętli `for` można pozwolić sobie na pewne odstępstwa od standardowej składni. Gdy piszemy pętlę `for`, często mamy ochotę przed rozpoczęciem wykonywania pętli zainicjalizować więcej niż jedną zmienną lub już w pętli wyliczać więcej niż jedno wyrażenie.

Wiele wyrażeń

W dowolnym składniku pętli `for` można umieścić dowolnie wiele wyrażeń, trzeba tylko rozdzielać je przecinkami; na przykład w instrukcji `for` zaczynającej się od:

```
for ( i = 0, j = 0;  i < 10;  ++i )
    ...
```

przed rozpoczęciem wykonywania pętli wartość `i` jest ustawiana na `0` *oraz* wartość `j` jest ustawiana na `0`. Dwa wyrażenia — `i = 0` i `j = 0` — połączone przecinkiem są razem traktowane jako element *`wyrażenie_początkowe`*. Oto inny przykład pętli `for`:

```
for ( i = 0, j = 100; i < 10; ++i, j = j – 10 )
    ...
```

Tutaj ustawiamy dwie zmienne indeksowe — `i` i `j` — pierwsza jest inicjalizowana na *`0`*, druga — na *`100`*. Przy każdym wykonaniu treści pętli wartość zmiennej `i` jest zwiększana o 1, a wartość zmiennej `j` jest zmniejszana o 10.

Pomijanie składników

Czasem w poszczególnych elementach pętli `for` trzeba użyć więcej niż jednego wyrażenia, kiedy indziej zdarza się, że trzeba *pominąć* jeden lub więcej tych elementów. W tym celu wystarczy ten element pominąć i zostawić jedynie średnik. Pomijanie jest najczęściej potrzebne w pętli `for`, kiedy nie ma żadnego wyrażenia inicjalizującego. Wtedy wystarczy, by element *`wyrażenie_początkowe`* pozostał pusty, a zachowany został sam średnik:

```
for ( ; j != 100; ++j )
    ...
```

Taka instrukcja może zostać użyta, jeśli zmienna `j` otrzyma wartość początkową jeszcze przed wykonaniem pętli.

Pętla, w której pominięto *`warunek_pętli`*, stanowi pętlę nieskończoną, która teoretycznie może wykonywać się bez końca. Bywa używana, ale wtedy musi istnieć inna metoda wyjścia z pętli (na przykład instrukcja `return`, `break` lub `goto`; instrukcje te będziemy omawiali później).

Deklarowanie zmiennych

W ramach wyrażenia inicjalizującego (początkowego) pętli `for` można deklarować zmienne. Jest to wykonywane w sposób standardowy dla języka C; na przykład poniższej instrukcji można użyć do utworzenia pętli `for`, w której zmienna `counter` jest jednocześnie definiowana i inicjalizowana przy użyciu wartości 1:

```
for ( int counter = 1; counter <= 5; ++counter )
```

Zmienna `counter` jest dostępna tylko na czas wykonywania pętli `for` (nazywamy ją zmienną *lokalną*) i nie można do niej sięgnąć poza pętlą. Oto inny przykład:

```
for ( int n = 1, triangularNumber = 0; n <= 200; ++n )
    triangularNumber += n;
```

w którym deklarujemy dwie zmienne typu `int` i od razu ustawiamy ich wartości.

Instrukcja while

Instrukcja `while` to następna metoda tworzenia pętli w języku C. Składnia tej powszechnie wykorzystywanej instrukcji jest następująca:

```
while ( wyrażenie )
    instrukcja
```

Znajdujące się w nawiasach *`wyrażenie`* jest wyliczane i jeśli jego wynik jest prawdziwy, wykonywana jest napisana dalej *`instrukcja`*. Po wykonaniu tej instrukcji (lub bloku instrukcji zamkniętych w nawiasy klamrowe) ponownie wyliczane jest wyrażenie i jeśli znowu jest prawdziwe, wykonywana jest *`instrukcja`*. Cały proces powtarza się tak długo, aż w końcu *`wyrażenie`* nie będzie spełnione; wtedy wykonywanie programu jest kontynuowane od instrukcji znajdującej się za *`instrukcją`*.

Program 4.6 pokazuje użycie pętli `while` liczącej po prostu od 1 do 5.

Program 4.6. **Podstawowe zastosowanie instrukcji while**

```
// Program pokazujący najprostsze użycie instrukcji while

#include <stdio.h>

int main (void)
{
    int count = 1;

    while ( count <= 5 ) {
        printf ("%i\n", count);
        ++count;
    }

    return 0;
}
```

Program 4.6. **Wyniki**

```
1
2
3
4
5
```

Program najpierw ustawia wartość zmiennej `count` na 1. Dalej wykonywana jest pętla `while`. Jako że wartość zmiennej `count` jest mniejsza lub równa 5, wykonywane są instrukcje

z treści pętli. Nawiasy klamrowe zamykają razem instrukcję `printf` i zwiększenie wartości zmiennej `count`. Na podstawie wyników działania programu łatwo zauważyć, że pętla wykonała się dokładnie pięć razy, czyli wartość `count` osiąga 6.

Być może zauważyliście, że taki sam efekt możemy uzyskać, stosując instrukcję `for`. Zawsze instrukcję `for` można przekształcić na odpowiadającą jej instrukcję `while` i odwrotnie. Na przykład ogólną postać instrukcji `for`:

```
for ( wyrażenie_początkowe; warunek_pętli; wyrażenie_pętli )
    instrukcja
```

można zapisać jako równoważną instrukcję `while`:

```
wyrażenie_początkowe;
while ( warunek_pętli ) {
    instrukcja
    wyrażenie_pętli;
}
```

Po nabyciu pewnego doświadczenia łatwo będzie „wyczuć", kiedy warto użyć instrukcji `while`, a kiedy `for`.

Ogólna zasada jest taka, że w pętli, która ma się wykonywać z góry ustaloną ilość razy, lepiej zastosować instrukcję `for`. Jeżeli wyrażenie początkowe, wyrażenie pętli i warunek pętli zawierają tę samą zmienną, zwykle lepszym wyborem jest `for`.

Następny program pokazuje inne użycie instrukcji `while`. Program wylicza *największy wspólny dzielnik* dwóch liczb całkowitych. Największy wspólny dzielnik (oznaczany zwykle przez *NWD* lub angielskie *gcd*) dwóch liczb to największa taka liczba, która obie te liczby dzieli bez reszty, na przykład NWD dla liczb 10 i 15 to 5, gdyż jest to największa liczba całkowita dzieląca bez reszty i 10, i 15.

Istnieje pewien sposób, czyli *algorytm*, który pozwala uzyskać NWD dla dwóch dowolnych liczb. Algorytm ten jest oparty na metodzie podanej pierwotnie przez Euklidesa około 300 roku p.n.e., można go zapisać następująco:

Problem: Znaleźć największy wspólny dzielnik dwóch nieujemnych liczb całkowitych *u* i *v*.

Krok 1: Jeśli *v* jest równe 0, działanie algorytmu jest skończone, a NWD jest równy *u*.

Krok 2: Wyliczamy `temp = u % v`, `u = v`, `v = temp` i wracamy do kroku 1.

Nie warto troszczyć się o szczegóły powyższego algorytmu — po prostu uwierzmy, że działa on prawidłowo. Raczej skoncentrujmy się na samym programie znajdującym największy wspólny dzielnik, a nie na analizie samego algorytmu.

Kiedy mamy rozwiązanie problemu znalezienia największego wspólnego dzielnika w formie algorytmu, znacznie łatwiej utworzyć odpowiedni program. Z analizy poszczególnych kroków algorytmu wynika, że krok 2. jest wykonywany wielokrotnie tak długo, aż zmienna `v` będzie równa `0`. Prowadzi to do naturalnej implementacji tego algorytmu w języku C z wykorzystaniem instrukcji `while`.

Program 4.7 znajduje NWD dwóch nieujemnych liczb całkowitych podanych przez użytkownika.

Program 4.7. **Znajdowanie największego wspólnego dzielnika**

```
/* Program znajdujący największy wspólny dzielnik
   dwóch nieujemnych liczb całkowitych.          */

#include <stdio.h>

int main (void)
{
    int u, v, temp;

    printf ("Proszę podać dwie nieujemne liczby całkowite.\n");
    scanf ("%i%i", &u, &v);

    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
    }

    printf ("Największy wspólny dzielnik podanych liczb to %i\n", u);

    return 0;
}
```

Program 4.7. **Wyniki**

```
Proszę podać dwie nieujemne liczby całkowite.
150 35
Największy wspólny dzielnik podanych liczb to 5
```

Program 4.7. **Wyniki (ponowne uruchomienie)**

```
Proszę podać dwie nieujemne liczby całkowite.
1026 405
Największy wspólny dzielnik podanych liczb to 27
```

Para wyrażeń `%i` w wywołaniu funkcji `scanf` mówi, że z klawiatury mają być podane dwie liczby całkowite. Pierwsza z nich zostanie zapisana w zmiennej `u`, druga w zmiennej `v`. Podczas wprowadzania wartości te mogą być rozdzielone jedną lub kilkoma spacjami lub znakami powrotu karetki.

Kiedy wartości zostaną już wpisane z klawiatury i umieszczone w zmiennych `u` i `v`, program wchodzi do pętli `while` w celu wyliczenia największego wspólnego dzielnika. Po wyjściu z pętli `while` wartość zmiennej `u`, która jest NWD `v` i pierwotnej wartości `u`, jest pokazywana na ekranie wraz ze stosownym komunikatem.

Program 4.8 demonstruje inne zastosowanie pętli `while` — służy ono do odwracania cyfr liczby wpisywanej z klawiatury. Jeśli na przykład użytkownik wpisze liczbę 1234, program odwróci jej cyfry i jako wynik pokaże 4321. Aby taki program napisać, najpierw musimy mieć algorytm realizujący tak postawione zadanie. Często analiza tego, jak samodzielnie taki problem rozwiązać, prowadzi do znalezienia potrzebnego algorytmu. Aby odwrócić cyfry liczby, wystarczy po prostu „kolejno odczytywać cyfry od prawej do lewej”. Możemy mieć program „kolejno odczytujący” cyfry liczby; wystarczy utworzyć

procedurę pobierającą kolejne cyfry liczby, zaczynając od skrajnej prawej. Następnie pobrana cyfra może zostać wyświetlona jako następna cyfra odwracanej liczby.

Program 4.8. **Odwracanie cyfr liczby**

```
// Program odwracający cyfry liczby.

#include <stdio.h>

int main (void)
{
    int number, right_digit;

    printf ("Podaj liczbę.\n");
    scanf ("%i", &number);

    while ( number != 0 ) {
        right_digit = number % 10;
        printf ("%i", right_digit);
        number = number / 10;
    }

    printf ("\n");

    return 0;
}
```

Program 4.8. **Wyniki**

```
Podaj liczbę.
13579
97531
```

Można pobrać prawą cyfrę liczby, jeśli weźmiemy resztę z dzielenia tej liczby przez 10. Na przykład 1234 % 10 daje wartość 4, czyli prawą cyfrę liczby 1234. Jest to jednocześnie pierwsza cyfra odwróconej liczby (pamiętajmy, że operator modulo daje resztę z dzielenia jednej liczby całkowitej przez inną). Następną cyfrę można uzyskać tak samo jak pierwszą, ale najpierw trzeba liczbę wyjściową podzielić przez 10 — ale podzielić w sensie dzielenia liczb całkowitych. Zatem 1234/10 daje 123, a 123 % 10 to 3, czyli jest to następna cyfra odwróconej liczby.

Opisaną procedurę kontynuujemy tak długo, aż pobierzemy ostatnią cyfrę. Ostatnia liczba jest pobrana, kiedy wynik ostatniego dzielenia całkowitoliczbowego to 0.

Kolejne cyfry są wyświetlane w chwili ich pobierania. Zauważmy, że nie wstawiliśmy w instrukcji `printf` z pętli `while` znaku nowego wiersza. Dzięki temu wszystkie cyfry są wyświetlane w jednym wierszu. Ostatnie wywołanie funkcji `printf` wstawia jedynie znak nowego wiersza, który powoduje, że kursor przechodzi do następnego wiersza.

Instrukcja do

Obie instrukcje omówione dotąd w tym rozdziale sprawdzają warunek pętli *przed* wykonaniem treści pętli. A zatem, jeśli warunek nie będzie spełniony, treść pętli może nie wykonać się ani razu. Czasami podczas tworzenia programu przydaje się sprawdzenie warunku dopiero na *końcu* pętli, a nie na początku. Oczywiście język C zawiera stosowną konstrukcję umożliwiającą takie działanie. Jest to instrukcja `do`. Składnia tej instrukcji jest następująca:

```
do
    instrukcja
while ( wyrażenie_pętli );
```

Wykonanie instrukcji `do` odbywa się następująco. Najpierw wykonywana jest *`instrukcja`*. Następnie wyliczane jest *`wyrażenie_pętli`* znajdujące się w nawiasach. Jeśli jest ono prawdziwe, pętla jest kontynuowana i ponownie wykonywana jest *`instrukcja`*. Pętla ta jest wykonywana tak długo, jak długo wyrażenie pozostaje prawdziwe. Kiedy wyrażenie przestaje być spełnione, pętla kończy swoje działanie i normalnie wykonywana jest instrukcja znajdująca się za pętlą.

Instrukcja `do` stanowi swego rodzaju odwrotność instrukcji `while`, gdyż warunek pętli umieszcza się na końcu, a nie na początku.

Pamiętać trzeba, że pętla `do` w przeciwieństwie do pętli `for` i `while` gwarantuje, że jej treść zostanie wykonana przynajmniej raz.

W programie 4.8 do odwracania cyfr liczby użyto instrukcji `while`. Cofnijmy się do tego programu i zastanówmy się, co się stanie, jeśli wpiszemy 0 zamiast 13759. Pętla `while` nie zostanie nigdy wykonana i jedynym wynikiem będzie pusty wiersz (wynikający z wyświetlenia znaku nowego wiersza w drugiej instrukcji `printf`). Jeśli użyjemy instrukcji `do` zamiast `while`, mamy gwarancję, że pętla wykona się przynajmniej raz, więc zawsze wyświetlona zostanie przynajmniej jedna cyfra. Program 4.9 to poprawiona wersja programu 4.8.

Program 4.9. **Poprawiony program odwracający cyfry podanej liczby**

```
// Program odwracający cyfry liczby.

#include <stdio.h>

int main (void)
{
    int number, right_digit;

    printf ("Podaj liczbę.\n");
    scanf ("%i", &number);

    do {
        right_digit = number % 10;
        printf ("%i", right_digit);
        number = number / 10;
    }
```

```
    while ( number != 0 );

    printf ("\n");

    return 0;
}
```

Program 4.9. **Wyniki**

```
Podaj liczbę.
13579
97531
```

Program 4.9. **Wyniki (ponowne uruchomienie)**

```
Podaj liczbę.
0
0
```

Jak widać, jeśli użytkownik poda liczbę 0, program także odpowie prawidłowo i wyświetli 0.

Instrukcja break

Czasami podczas wykonywania pętli pożądane jest opuszczenie tej pętli w przypadku zajścia jakiegoś warunku (na przykład pojawienia się błędu lub zbyt wczesnego natknięcia się na koniec danych). W takich sytuacjach używa się instrukcji `break`. Instrukcja ta powoduje natychmiastowe przerwanie przez program właśnie wykonywanej pętli, niezależnie od jej rodzaju: `for`, `while` czy `do`. Pozostałe instrukcje z pętli są pomijane, kończy się też cała pętla. Program jest wykonywany od instrukcji znajdującej się za pętlą.

Jeśli instrukcja `break` zostanie wykonana w pętlach zagnieżdżonych, przerwana będzie jedynie pętla najbardziej wewnętrzna, w której wystąpiła `break`.

Format instrukcji `break` jest bardzo prosty, zawiera ona słowo kluczowe `break` i średnik:

```
break;
```

Instrukcja continue

Instrukcja `continue` jest podobna do `break`, ale nie przerywa pętli. Instrukcja ta powoduje kontynuowanie wykonania pętli, w której wystąpiła. Kiedy wykonana zostanie instrukcja `continue`, wszelkie instrukcje znajdujące się w pętli *za* `continue` są pomijane, ale wykonywanie pętli jest normalnie kontynuowane.

Instrukcja `continue` najczęściej jest używana do pominięcia grupy instrukcji z pętli, jeśli spełniony jest jakiś warunek. Jeśli warunek nie jest spełniony, pętla ma się wykonywać normalnie. Składnia instrukcji `continue` jest bardzo prosta:

```
continue;
```

Nie należy używać instrukcji `break` i `continue`, póki nie nabierze się praktyki w pisaniu pętli i w normalnym ich kończeniu. Zbyt łatwo nadużyć tych instrukcji, co może powodować, że programy będą nieczytelne.

Teraz znamy już podstawowe konstrukcje pętli dostępne w języku C, możemy więc przejść do następnej grupy instrukcji, służących do podejmowania decyzji. Temu jest poświęcony rozdział 5. Najpierw jednak warto rozwiązać poniższe ćwiczenia, aby utrwalić sobie wiedzę o pętlach.

Ćwiczenia

1. Przepisz i uruchom dziewięć programów pokazanych w tym rozdziale. Uzyskane wyniki porównaj z wynikami pokazanymi w tekście.
2. Napisz program generujący i wyświetlający tablicę wartości n i n^2 dla wartości całkowitych n od 1 do 10. Pamiętaj o właściwych nagłówkach kolumn.
3. Liczby trójkątne możemy też generować, korzystając ze wzoru:

   ```
   triangularNumber = n (n + 1) / 2
   ```

   dla całkowitych wartości n. Na przykład dziesiąta liczba trójkątna, czyli 55, może być wygenerowana przez podstawienie do powyższego wzoru $n = 10$. Napisz program generujący tablicę liczb trójkątnych, wykorzystujący właśnie powyższy wzór. Niech program wygeneruje co piątą liczbę trójkątną z zakresu od 5 do 50 (czyli 5, 10, 15, …, 50).
4. Silnia liczby całkowitej n, zapisywana jako $n!$, to iloczyn kolejnych liczby całkowitych od 1 do n. Na przykład 5! liczy się następująco:

   ```
   5! = 5·4·3·2·1 = 120
   ```

   Napisz program generujący i pokazujący tablicę od 1! do 10!.
5. Poniższy program w języku C jest jak najbardziej poprawny, ale nie zatroszczono się w nim o układ kodu. Jak widać, trudno go czytać (możecie wierzyć lub nie, ale program można uczynić jeszcze bardziej nieczytelnym!). Korzystając z programów przykładowych z tego rozdziału, doprowadź poniższy program do bardziej czytelnej postaci. Następnie wprowadź program do komputera i uruchom go.

   ```
   #include <stdio.h>
   int main(void){
   int n, two_to_the_n;
   printf("TABLICA POTĘG DWÓJKI\n\n");
   printf(" n    2 do n-tej\n");
   two_to_the_n=1;
   for(n=0;n<=10;++n){
   printf("%2i       %i\n",n,two_to_the_n); two_to_the_n*=2;}
   return 0;}
   ```

6. Znak minus umieszczony przed specyfikacją szerokości pola powoduje, że pole będzie *wyrównane do lewej strony*. Zastąp odpowiednią instrukcję `printf` z programu 4.2 instrukcją poniższą, uruchom program i porównaj uzyskane wyniki.

```
printf ("%-2i        %i\n", n, triangularNumber);
```

7. Kropka umieszczona przed specyfikacją szerokości pola ma specjalne znaczenie. Spróbuj je odgadnąć, wpisując i uruchamiając poniższy program. Poeksperymentuj, wpisując na żądanie programu różne wartości.

```
#include <stdio.h>

int main (void)
{
    int dollars, cents, count;

    for ( count = 1; count <= 10; ++count) {
        printf ("Podaj ilość dolarów: ");
        scanf ("%i", &dollars);
        printf ("Podaj ilość centów: ");
        scanf ("%i", &cents);
        printf ("$%i.%.2i\n\n", dollars, cents);
    }
    return 0;
}
```

8. Program 4.5 umożliwia użytkownikowi podanie jedynie pięciu różnych liczb. Zmodyfikuj ten program tak, aby użytkownik mógł podać, ile liczb trójkątnych będzie liczonych.
9. Przepisz programy od 4.2 do 4.5, zastępując wszystkie wystąpienia instrukcji `for` odpowiednią instrukcją `while`. Uruchom każdy z nich, aby sprawdzić, czy obie wersje dają identyczne wyniki.
10. Co by się stało, gdybyś w programie 4.8 podał liczbę ujemną? Sprawdź to w praktyce.
11. Napisz program wyliczający sumę cyfr liczby całkowitej. Na przykład suma cyfr liczby 2155 to 2+1+5+5, czyli 13. Program powinien przyjmować dowolne liczby całkowite podawane przez użytkownika.

5

Podejmowanie decyzji

W rozdziale 4. poznaliśmy podstawową cechę komputera, czyli gotowość do wielokrotnego powtarzania ciągu instrukcji. Inną ogromnie ważną cechą komputera jest możliwość podejmowania decyzji. Widzieliśmy ten proces w czasie sprawdzania warunku pętli. Bez takiej możliwości program nigdy nie byłby w stanie „wyjść" z pętli i stale powtarzałby te same instrukcje, teoretycznie w nieskończoność (stąd zresztą nazwa: *pętla nieskończona*).

Język programowania C zawiera też kilka innych konstrukcji umożliwiających podejmowanie decyzji. W tym rozdziale opiszemy:

- instrukcję `if`,
- instrukcję `switch`,
- operator wyboru.

Instrukcja if

W języku C podstawową metodą podejmowania decyzji jest użycie ogólnej instrukcji `if`. Pełna postać tej instrukcji ma formę:

```
if ( wyrażenie )
    instrukcja
```

Wyobraźmy sobie, że zdanie „Jeśli nie będzie padało, pójdę popływać" mamy przetłumaczyć na język C. Korzystając z instrukcji `if` (samo słowo *if* oznacza *jeśli*), możemy napisać:

```
if ( nie będzie padało )
    pójdę popływać
```

Instrukcja `if` służy do sterowania wykonywaniem instrukcji (lub grupy instrukcji ujętych w nawiasy klamrowe) w zależności od spełnienia bądź niespełnienia pewnych warunków. Pójdę popływać, jeśli nie będzie padało. Analogicznie w programie:

```
if ( liczba > OGRANICZENIE )
    printf ("Przekroczono nałożone ograniczenie.\n");
```

instrukcja `printf` jest wykonywana tylko wtedy, gdy zmienna `liczba` ma wartość większą niż `OGRANICZENIE`; w przeciwnym razie instrukcja `printf` jest pomijana.

Konkretny program pomoże dokładniej zorientować się, o co chodzi. Załóżmy, że potrzebny jest program przyjmujący podaną przez użytkownika liczbę całkowitą i następnie pokazujący wartość bezwzględną tej liczby. Prostym sposobem obliczania wartości bezwzględnej jest zanegowanie liczby, jeśli jest ona mniejsza od zera. Zaistnienie frazy „jeśli jest ona mniejsza od zera" znamionuje konieczność podjęcia decyzji. W programie 5.1 decyzja jest zapisana w formie instrukcji `if`.

Program 5.1. **Wyliczanie bezwzględnej wartości liczby całkowitej**

```
// Program wyliczający wartość bezwzględną liczby całkowitej.

#include <stdio.h>

int main (void)
{
    int number;

    printf ("Podaj liczbę: ");
    scanf ("%i", &number);

    if ( number < 0 )
        number = -number;

    printf ("Wartość bezwzględna to %i\n", number);

    return 0;
}
```

Program 5.1. **Wyniki**

```
Podaj liczbę: -100
Wartość bezwzględna to 100
```

Program 5.1. **Wyniki (ponowne uruchomienie)**

```
Podaj liczbę: 2000
Wartość bezwzględna to 2000
```

Program został uruchomiony dwukrotnie, aby sprawdzić, czy prawidłowo działa. Oczywiście, aby zwiększyć zaufanie do niego, należałoby go jeszcze kilkakrotnie uruchomić, ale teraz sprawdziliśmy przynajmniej dwie możliwości podejmowanej decyzji.

Użytkownik widzi komunikat, a wprowadzona liczba jest zapisywana w zmiennej `number`. Program sprawdza, czy wartość `number` jest mniejsza od zera, jeśli tak, jej wartość jest negowana. Jeśli wartość zmiennej `number` nie jest mniejsza od zera, instrukcja negująca jest pomijana — przecież nie chcemy negować liczb dodatnich. W końcu wartość bezwzględna zmiennej `number` jest wyświetlana i na tym działanie programu się kończy.

Spójrzmy na program 5.2, w którym też użyto instrukcji `if`. Wyobraźmy sobie, że mamy listę ocen, dla których chcemy obliczyć średnią. Jednak niezależnie od liczenia

średniej chcemy wyznaczyć liczbę ocen negatywnych. Na potrzeby tego przykładu założymy, że ocena jest negatywna, jeśli jest mniejsza niż 65.

Program 5.2. **Wyliczanie średniej zestawu ocen oraz zliczanie ocen negatywnych**

```
/* Program wylicza średnią dla zestawu ocen oraz zlicza, ile
   ocen jest negatywnych.  */

#include <stdio.h>

int main (void)
{
    int      numberOfGrades, i, grade;
    int      gradeTotal = 0;
    int      failureCount = 0;
    float    average;

    printf ("Ile ocen zamierzasz wprowadzić? ");
    scanf ("%i", &numberOfGrades);

    for ( i = 1; i <= numberOfGrades; ++i ) {
        printf ("Podaj ocenę nr %i: ", i);
        scanf ("%i", &grade);

        gradeTotal = gradeTotal + grade;
        if ( grade < 65 )
            ++failureCount;
    }

    average = (float) gradeTotal / numberOfGrades;

    printf ("\nŚrednia ocen = %.2f\n", average);
    printf ("Liczba ocen negatywnych = %i\n", failureCount);

    return 0;
}
```

Program 5.2. **Wyniki**

```
Ile ocen zamierzasz wprowadzić? 7
Podaj ocenę nr 1: 93
Podaj ocenę nr 2: 63
Podaj ocenę nr 3: 87
Podaj ocenę nr 4: 65
Podaj ocenę nr 5: 62
Podaj ocenę nr 6: 88
Podaj ocenę nr 7: 76

Średnia ocen = 76.29
Liczba ocen negatywnych = 2
```

Konieczność zliczania ocen negatywnych wskazuje, że trzeba podjąć decyzję, czy dana ocena jest negatywna. Znowu użyjemy instrukcji `if`.

Zmienna `gradeTotal`, służąca do sumowania kolejnych ocen w miarę ich wprowadzania przez użytkownika, ma początkowo wartość `0`. Liczba ocen negatywnych jest zapisywana w zmiennej `failureCount`, także początkowo ustawianej na `0`. Zmienna `average` jest typu `float`, gdyż średnia zbioru liczb całkowitych wcale nie musi być liczbą całkowitą.

Program prosi użytkownika o podanie liczby ocen, odpowiedź jest zapisywana w zmiennej `numberOfGrades`. Następnie przygotowywana jest pętla, której jeden przebieg pozwala obsłużyć jedną ocenę. W pierwszej części pętli użytkownik powinien podać ocenę. Ocena jest umieszczana w zmiennej `grade`.

Wartość zmiennej `grade` jest dodawana do `gradeTotal`, następnie program sprawdza, czy podana ocena jest oceną negatywną. Jeśli tak, zwiększana jest o 1 wartość zmiennej `failureCount`. Potem pętla jest powtarzana dla następnej oceny z listy.

Kiedy wszystkie oceny zostaną wprowadzone i zsumowane, program wylicza średnią. W pierwszej chwili wydawałoby się, że wystarczy wyrażenie:

```
average = gradeTotal / numberOfGrades;
```

Przypomnijmy jednak, że w takim wypadku zastosowana byłaby arytmetyka całkowitoliczbowa, więc część ułamkowa ilorazu zostałaby odrzucona. Wynika to stąd, że zastosowane byłoby dzielenie liczb całkowitych, gdyż liczbami całkowitymi jest *zarówno* dzielna, jak i dzielnik.

Problem ten można rozwiązać na różne sposoby. Jednym z nich jest zadeklarowanie zmiennej `numberOfGrades` lub `gradeTotal` jako zmiennej typu `float`. Wtedy dzielenie zostanie wykonane bez odrzucania części ułamkowej. Jednak obie te zmienne służą do zapisywania liczb całkowitych i zadeklarowanie ich jako `float` zaciemniłoby ich przeznaczenie. Tak nie należy postępować.

Drugie rozwiązanie, zastosowane w naszym programie, polega na jawnej *konwersji* jednej z wartości na zmiennoprzecinkową. Użyto operatora rzutowania `(float)`. Zmienna `gradeTotal` jest rzutowana na typ `float` jeszcze przed dzieleniem, więc mamy do czynienia z dzieleniem zmiennoprzecinkowym i uzyskamy także część ułamkową wyniku.

Po wyliczeniu pokazujemy na ekranie średnią z dwoma miejscami po przecinku. Jeśli w instrukcji `printf` bezpośrednio przed znakiem określającym format (`f` lub `e`) zostanie dodana kropka dziesiętna oraz liczba (oba te znaki są nazywane *modyfikatorami dokładności*), pokazywana wartość zostanie zaokrąglona do podanej liczby miejsc dziesiętnych. Wobec tego w programie 5.2 modyfikator dokładności `.2` powoduje, że zmienna `average` zostanie wyświetlona z dwoma miejscami dziesiętnymi.

Kiedy program wyświetli liczbę ocen negatywnych, jego działanie się zakończy.

Zwróć uwagę, że jeśli użytkownik wpisze 0 jako liczbę ocen do zarejestrowania, program wygeneruje dziwne wyniki typu NaN (*Not a Number*) lub jeszcze coś innego — to zależy od systemu i sposobu postępowania w nim z przypadkami dzielenia przez zero. Pewnie zastanawiasz się, po co ktoś miałby uruchamiać program do rejestrowania wyników testów, skoro nie ma nic do wpisania, ale jednak tego typu konstrukcje sprawdzające błędy powinno się dodawać do programów.

Konstrukcja if-else

Kiedy ktoś prosi o sprawdzenie, czy dana liczba jest parzysta, większość osób sprawdza jedynie ostatnią cyfrę. Jeśli jest to 0, 2, 4, 6 lub 8, stwierdza, że liczba jest parzysta. Inaczej liczba jest uznawana za nieparzystą. W komputerze, zamiast kontrolować ostatnią cyfrę, łatwiej sprawdzić, czy liczba jest podzielna bez reszty przez 2. Jeśli tak, liczba jest parzysta. Jeśli nie — jest nieparzysta.

Widzieliśmy już użycie operatora modulo „%" do sprawdzania, jaka jest reszta z dzielenia. Jest to zatem doskonały operator do kontroli, czy liczba dzieli się przez 2 bez reszty. Jeśli reszta z takiego dzielenia wynosi 0, liczba jest parzysta. W przeciwnym razie jest nieparzysta.

Spójrzmy na program 5.3. Sprawdza on, czy liczba jest parzysta, czy nie, i wyświetla stosowny komunikat.

Program 5.3. **Sprawdzanie, czy liczba jest parzysta**

```
// Program sprawdza, czy liczba jest parzysta, czy nie.

#include <stdio.h>

int main (void)
{
    int number_to_test, remainder;

    printf ("Podaj liczbę, którą chcesz sprawdzić: ");
    scanf ("%i", &number_to_test);

    remainder = number_to_test % 2;

    if ( remainder == 0 )
        printf ("Podana liczba jest parzysta.\n");

    if ( remainder != 0 )
        printf ("Podana liczba jest nieparzysta.\n");

    return 0;
}
```

Program 5.3. **Wyniki**

```
Podaj liczbę, którą chcesz sprawdzić: 2455
Podana liczba jest nieparzysta.
```

Program 5.3. **Wyniki (ponowne uruchomienie)**

```
Podaj liczbę, którą chcesz sprawdzić: 1210
Podana liczba jest parzysta.
```

Kiedy użytkownik poda liczbę, wyznaczana jest reszta z dzielenia przez 2. Pierwsza instrukcja `if` służy do sprawdzenia, czy reszta z dzielenia jest zerem. Jeśli tak, wyświetlany jest komunikat: „Podana liczba jest parzysta".

Druga instrukcja `if` sprawdza, czy reszta z dzielenia nie jest zerem, i jeśli ten warunek jest spełniony, wyświetlany jest komunikat: „Podana liczba jest nieparzysta”.

Jeśli spełniony zostanie warunek pierwszej instrukcji `if`, warunek drugiej musi być niespełniony… i odwrotnie. Przypomnijmy sobie dyskusję z początku tego podrozdziału — jeśli liczba dzieli się bez reszty przez dwa, jest parzysta; w przeciwnym razie jest nieparzysta.

Przy pisaniu programu potrzeba opisania „przeciwnego razu” jest tak powszechna, że niemal wszystkie współczesne języki programowania posiadają specjalną konstrukcję składniową. W C jest to konstrukcja zwana `if-else`, której ogólna postać jest następująca:

```
if ( wyrażenie )
    instrukcja 1
else
    instrukcja 2
```

Konstrukcja `if-else` stanowi rozszerzenie ogólnej postaci instrukcji `if`. Jeśli podane w instrukcji `if` wyrażenie jest prawdziwe, wykonywana jest pierwsza instrukcja, *`instrukcja 1`*. W przeciwnym wypadku wykonywana jest *`instrukcja 2`*. Zawsze wykonywana jest jedna z instrukcji, ale nigdy obie.

W programie 5.3 możemy użyć konstrukcji `if-else`, zastępując nią dwie osobne instrukcje `if`. Użycie takiej konstrukcji zmniejsza stopień złożoności programu i poprawia jego czytelność, co widać w programie 5.4.

Program 5.4. **Poprawiony program sprawdzający, czy liczba jest parzysta**

```
// Program sprawdza, czy liczba jest parzysta, czy nie. (wersja 2)

#include <stdio.h>

int main (void)
{
    int number_to_test, remainder;

    printf ("Podaj liczbę, którą chcesz sprawdzić: ");
    scanf ("%i", &number_to_test);

    remainder = number_to_test % 2;

    if ( remainder == 0 )
        printf ("Podana liczba jest parzysta.\n");
    else
        printf ("Podana liczba jest nieparzysta.\n");

    return 0;
}
```

Program 5.4. **Wyniki**

```
Podaj liczbę, którą chcesz sprawdzić: 1234
Podana liczba jest parzysta.
```

Program 5.4. **Wyniki (ponowne uruchomienie)**

```
Podaj liczbę, którą chcesz sprawdzić: 6551
Podana liczba jest nieparzysta.
```

Pamiętajmy, że podwójny znak równości „==" sprawdza równoważność, natomiast pojedynczy znak równości to operator przypisania. Zapominanie o tym rozróżnieniu i używanie operatora przypisania w instrukcji `if` może być przyczyną mnóstwa poważnych problemów.

Złożone warunki porównania

Instrukcje `if` używane dotąd wykorzystują proste porównywanie dwóch liczb. W programie 5.1 porównywaliśmy wartość zmiennej `number` z liczbą 0, w programie 5.2 — wartość zmiennej `grade` z liczbą `65`. Czasami przydatne lub nawet konieczne staje się przeprowadzenie bardziej skomplikowanych porównań. Załóżmy na przykład, że w programie 5.2 chcemy zliczyć nie oceny negatywne, ale oceny z zakresu od 70 do 79. Wtedy nie wystarczy porównać wartości zmiennej `grade` z jedną liczbą, ale z dwiema: 70 i 79.

Język C zapewnia mechanizmy niezbędne do wykonania tego typu porównań. *Złożony warunek porównania* to po prostu jedno proste porównanie lub kilka połączonych albo operatorem *logicznym AND*, albo *logicznym OR*. Operatory te są zapisywane odpowiednio znakami `&&` lub `||` (dwie pionowe kreski). Na przykład instrukcja:

```
if ( grade >= 70  &&  grade <= 79 )
    ++grades_70_to_79;
```

powoduje zwiększenie wartości zmiennej `grades_70_to_79` tylko wtedy, gdy wartość zmiennej `grade` jest większa lub równa `70` *oraz* mniejsza lub równa `79`. Analogicznie instrukcja:

```
if ( index < 0  ||  index > 99 )
    printf ("Błąd – indeks spoza zakresu\n");
```

powoduje wykonanie instrukcji `printf`, jeśli wartość zmiennej `index` jest mniejsza od `0` lub większa od `99`.

W języku C złożone warunki porównania mogą składać się na wyjątkowo złożone wyrażenia. Język ten umożliwia programiście praktycznie dowolne tworzenie zestawień. Ta elastyczność często bywa nadużywana — im wyrażenia prostsze, tym łatwiej je czytać i usuwać z nich błędy.

Tworząc złożone warunki porównania, musimy używać nawiasów, aby zwiększać czytelność wyrażeń i unikać problemów związanych z nieprawidłowym zapamiętaniem priorytetów. W celu zwiększenia czytelności można też wykorzystać dodatkowe spacje, które wstawione dookoła operatorów `&&` i `||` wizualnie oddzielają je od łączonych z nimi wyrażeń.

Aby zilustrować użycie złożonych warunków porównania, pokażemy praktyczny przykład sprawdzający, czy dany rok jest przestępny. Rok jest przestępny, jeśli jest podzielny przez 4. Jeśli jednak jednocześnie jest podzielny przez 100, *nie* jest przestępny — chyba że jest podzielny przez 400.

Zastanówmy się, jak przygotować sprawdzenie tego warunku. Najpierw możemy wyliczyć reszty z dzielenia roku przez 4, 100 i 400; wartości te przypiszemy odpowiednio zmiennym `rem_4`, `rem_100` i `rem_400`. Następnie możemy porównywać te zmienne, aby sprawdzić, czy spełniony jest warunek przestępności roku.

Jeśli przekształcimy podaną wcześniej definicję roku przestępnego, możemy powiedzieć, że rok jest przestępny, jeśli jest podzielny przez 4, ale nie przez 100, lub jest podzielny przez 400. Warto na chwilę zastanowić się nad tym zdaniem, aby nabrać przekonania, że jest ono równoważne poprzedniej definicji. Teraz już względnie łatwo można zakodować naszą definicję i zmienić ją w program:

```
if ( (rem_4 == 0  &&  rem_100 != 0) != 0) ||  rem_400 == 0 )
   printf ("Podany rok jest przestępny.\n");
```

Nawiasy wokół wyrażenia:

```
rem_4 == 0  &&  rem_100 != 0
```

nie są niezbędne, gdyż wyrażenie byłoby obliczane tak samo i bez nich. Jeśli przed naszym sprawdzeniem dodamy kilka instrukcji deklaracji zmiennych oraz umożliwimy użytkownikowi wprowadzenie roku, który ma być sprawdzony, mamy gotowy program 5.5, sprawdzający, czy rok jest przestępny.

Program 5.5. **Sprawdzanie, czy dany rok jest przestępny**

```
// Program sprawdzający, czy dany rok jest przestępny.

#include <stdio.h>

int main (void)
{
    int year, rem_4, rem_100, rem_400;

    printf ("Podaj rok, który chcesz sprawdzić: ");
    scanf ("%i", &year);

    rem_4 = year % 4;
    rem_100 = year % 100;
    rem_400 = year % 400;

    if ( (rem_4 == 0  &&  rem_100 != 0) != 0) ||  rem_400 == 0 )
       printf ("Podany rok jest przestępny.\n");
    else
        printf ("Nie, ten rok nie jest przestępny.\n");

    return 0;
}
```

Program 5.5. **Wyniki**

```
Podaj rok, który chcesz sprawdzić: 1955
Nie, ten rok nie jest przestępny.
```

Program 5.5. **Wyniki (ponowne uruchomienie)**

```
Podaj rok, który chcesz sprawdzić: 2000
Podany rok jest przestępny.
```

Program 5.5. **Wyniki (ponowne uruchomienie)**

```
Podaj rok, który chcesz sprawdzić: 1800
Nie, ten rok nie jest przestępny.
```

Powyższe przykłady pokazują, że rok niepodzielny przez 4 (1955) nie jest przestępny, że jest przestępny rok podzielny przez 400 (2000) i rok podzielny przez 100, ale niepodzielny przez 400 (1800), nie jest przestępny. Aby skończyć testowanie programu, powinniśmy sprawdzić jeszcze rok podzielny przez 4, ale niepodzielny przez 100. To ćwiczenie pozostawiamy już czytelnikom.

Jak wspomnieliśmy wcześniej, język C zapewnia niesamowitą elastyczność tworzenia wyrażeń. Przykładowo, w poprzednim programie nie musieliśmy wyliczać wyników pośrednich, `rem_4`, `rem_100` i `rem_400`, a obliczenia zostały wykonane bezpośrednio w instrukcji `if`, jak poniżej:

```
if ( ( year % 4 == 0  &&  year % 100 != 0 )  || year % 400 == 0 )
```

Użycie dodatkowych spacji między operatorami poprawia czytelność powyższego wyrażenia. Jeśli zdecydujemy się usunąć spacje i zbędne nawiasy, to samo wyrażenie przybierze postać:

```
if(year%4==0&&year%100!=0)||year%400==0)
```

Wyrażenie to jest jak najbardziej poprawne i zadziała identycznie (czy kto chce wierzyć, czy nie) jak wyrażenie poprzednie. Jak widać, dodatkowe spacje znakomicie poprawiają czytelność złożonych wyrażeń.

Zagnieżdżone instrukcje if

Jeśli w ogólnej postaci instrukcji `if` wyrażenie w nawiasach jest prawdziwe, wykonywana jest znajdująca się dalej instrukcja. Nie ma żadnych przeszkód, aby tą instrukcją była kolejna instrukcja `if`, tak jak poniżej:

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Twój ruch\n");
```

Jeśli wartość zmiennej `gameIsOver` jest zerem, wykonywana jest następna instrukcja, czyli kolejna instrukcja `if`. Ta instrukcja porównuje wartość zmiennej `playerToMove` z wartością `YOU`. Jeśli obie wartości są równe, wyświetlany jest komunikat: „Twój ruch”. Zatem instrukcja `printf` jest wykonywana tylko wtedy, gdy zmienna `gameIsOver` jest równa 0 *oraz* zmienna `playerToMove` jest równa `YOU`. Całą tę instrukcję równie dobrze można by zresztą zapisać następująco:

```
if ( gameIsOver == 0  &&  playerToMove == YOU )
printf ("Twój ruch\n");
```

Bardziej praktycznym przykładem „zagnieżdżonych” instrukcji `if` jest sytuacja, kiedy do poprzedniego przykładu dodamy frazę `else`:

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Twój ruch\n");
    else
        printf ("Mój ruch\n");
```

Wykonanie tego programu przebiega podobnie jak poprzednio. Jeśli jednak zmienna `gameIsOver` ma wartość zero, a `playerToMove` nie ma wartości `YOU`, wykonywana jest fraza `else` i wyświetlany komunikat: „Mój ruch”. Jeśli zmienna `gameIsOver` nie jest równa zeru, cała wewnętrzna instrukcja `if`, wraz z frazą `else`, jest całkowicie pomijana.

Zauważmy, że pokazana fraza `else` jest związana z instrukcją `if` sprawdzającą zmienną `playerToMove`, a nie `gameIsOver`. Fraza `else` zawsze jest związana z ostatnią instrukcją `if`, która nie ma tej frazy.

Możemy pójść krok dalej i dodać do zewnętrznej instrukcji `if` z poprzedniego przykładu frazę `else`. Fraza ta będzie użyta, kiedy wartość zmiennej `gameIsOver` nie będzie zerem.

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Twój ruch\n");
    else
        printf ("Mój ruch\n");
else
    printf ("Koniec gry\n");
```

Prawidłowo zastosowane wcięcia ułatwiają zrozumienie sposobu działania skomplikowanych instrukcji.

Oczywiście, jeśli nawet użyjemy wcięć, aby zasugerować, jakiego działania oczekujemy, może się zdarzyć, że nasze oczekiwania będą rozbieżne ze sposobem interpretacji instrukcji przez kompilator. Jeśli na przykład z poprzedniego przykładu usuniemy pierwszą frazę `else`:

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Twój ruch\n");
else
    printf ("Koniec gry\n");
```

całość instrukcji *nie* będzie interpretowana tak, jak wynikałoby ze sposobu jej sformatowania. Instrukcja ta zostanie zinterpretowana następująco:

```
if ( gameIsOver == 0 )
    if ( playerToMove == YOU )
        printf ("Twój ruch\n");
    else
        printf ("Koniec gry\n");
```

gdyż istniejąca fraza `else` zostanie sparowana z pierwszą instrukcją `if` bez `else`. Można użyć nawiasów klamrowych, żeby wymusić pożądaną interpretację instrukcji — aby na przykład sparować frazę `else` z zewnętrzną instrukcją `if`. Pokazane nawiasy powodują „zamknięcie” instrukcji `if`.

```
if ( gameIsOver == 0 ) {
    if ( playerToMove == YOU )
        printf ("Twój ruch\n");
}
else
    printf ("Koniec gry\n");
```

Powyższy kod zadziała już zgodnie z oczekiwaniami i komunikat: „Koniec gry” będzie pokazywany, kiedy zmienna `gameIsOver` będzie miała wartość niezerową.

Konstrukcja else if

Widzieliśmy już, jak używa się instrukcji `else` do obsługi wykluczających się warunków — liczba albo jest parzysta, albo nieparzysta; rok albo jest przestępny, albo nie jest przestępny. Jednak w programach nie wszystkie decyzje są takie proste. Rozważmy program wyświetlający –1, jeśli podana przez użytkownika liczba jest mniejsza od zera; wyświetlający 0, jeżeli podaną liczbą jest zero, i wyświetlający 1, gdy podana liczba jest większa od zera. Opisane zachowanie to *de facto* opis funkcji *signum*. Oczywiście trzeba wykonać trzy sprawdzenia: czy liczba jest ujemna, dodatnia, czy też jest zerem. Nasza prosta konstrukcja `if-else` nie wystarczy. Można by oczywiście użyć trzech osobnych instrukcji `if`, ale jest to rozwiązanie mało ogólne, szczególnie jeśli sprawdzane warunki nie zawsze się wykluczają.

Opisaną sytuację można rozwiązać, jeśli we frazie `else` dodamy następną instrukcję `if`. Za `else` mogą znajdować się dowolne, byle poprawne, instrukcje języka C, więc logiczne wydaje się, że może to być kolejna instrukcja `if`. Wobec tego ogólnie można napisać:

```
if ( wyrażenie 1 )
    instrukcja 1
else
    if ( wyrażenie 2 )
        instrukcja 2
    else
        instrukcja 3
```

W ten sposób uzyskujemy podejmowanie decyzji nie w logice dwuwartościowej, ale w logice trójwartościowej. Możemy dodawać kolejne instrukcje `if` we frazach `else` tak, że ostatecznie nasze decyzje mogą wykorzystywać logikę *n*-wartościową.

Pokazana powyżej konstrukcja jest często używana; nazywa się ją konstrukcją `else if`, ale jej formatowanie najczęściej jest nieco inne niż to, które pokazaliśmy:

```
if ( wyrażenie 1 )
    instrukcja 1
else if ( wyrażenie 2 )
    instrukcja 2
else
        instrukcja 3
```

Takie formatowanie poprawia czytelność instrukcji i bardziej unaocznia, że podejmujemy decyzję, mając do wyboru trzy różne możliwości.

Program 5.6 pokazuje użycie konstrukcji `else if` do zaimplementowania opisanej wyżej funkcji *signum*.

Program 5.6. **Implementacja funkcji signum**

```
// Program implementujący funkcję signum.

#include <stdio.h>

int main (void)
{
    int number, sign;

    printf ("Proszę podać liczbę: ");
    scanf ("%i", &number);

    if ( number < 0 )
        sign = -1;
    else if ( number == 0 )
        sign = 0;
    else        // Musi być dodatnia
        sign = 1;
    printf ("Signum = %i\n", sign);

    return 0;
}
```

Program 5.6. **Wyniki**

```
Proszę podać liczbę: 1121
Signum = 1
```

Program 5.6. **Wyniki (ponowne uruchomienie)**

```
Proszę podać liczbę: -158
Signum = -1
```

Program 5.6. **Wyniki (ponowne uruchomienie)**

```
Proszę podać liczbę: 0
Signum = 0
```

Jeśli podana liczba jest mniejsza od zera, zmienna `sign` otrzymuje wartość –1; jeśli podana liczba jest zerem, `sign` otrzymuje wartość 0; w przeciwnym wypadku liczba musi być większa od zera, więc zmienna `sign` otrzymuje wartość 1.

Program 5.7 analizuje znak podany przez użytkownika i klasyfikuje go jako literę (od *a* do *z* lub od *A* do *Z*), cyfrę (od *0* do *9*) lub znak specjalny (wszystkie inne). Aby odczytać z terminala pojedynczy znak, w funkcji `scanf` używamy formantu `%c`.

Program 5.7. **Ustalenie rodzaju pojedynczego znaku podanego z terminala**

```
// Program ustala rodzaj pojedynczego znaku podanego z terminala.

#include <stdio.h>

int main (void)
{
    char c;

    printf ("Podaj znak:\n");
    scanf ("%c", &c);

    if ( (c >= 'a'  &&  c <= 'z') || (c >= 'A'  &&  c <= 'Z') )
        printf ("Podany znak jest literą.\n");
    else if ( c >= '0'  &&  c <= '9' )
        printf ("Podany znak to cyfra.\n");
    else
        printf ("Podano znak specjalny.\n");

    return 0;
}
```

Program 5.7. **Wyniki**

```
Podaj znak:
&
Podano znak specjalny.
```

Program 5.7. **Wyniki (ponowne uruchomienie)**

```
Podaj znak:
8
Podany znak to cyfra.
```

Program 5.7. **Wyniki (ponowne uruchomienie)**

```
Podaj znak:
B
Podany znak jest literą.
```

Najpierw ustalamy, czy wartość zmiennej c typu `char` jest literą. W tym celu sprawdzamy, czy jest to mała litera lub czy jest to litera wielka. Małe litery znajdujemy, korzystając z wyrażenia:

```
( c >= 'a'  &&  c <= 'z' )
```

Wyrażenie to jest prawdziwe, jeśli wartość zmiennej c mieści się w zakresie od `'a'` do `'z'`. Wielkie litery są sprawdzane za pomocą wyrażenia:

```
( c >= 'A'  &&  c <= 'Z' )
```

które jest prawdziwe, jeśli wartość zmiennej c mieści się w zakresie od 'A' do 'Z'. Oba podane porównania zadziałają poprawnie na komputerach wykorzystujących na poziomie maszynowym kodowanie ASCII[1].

Jeśli w zmiennej c będzie litera, wykona się kod z pierwszej instrukcji if i wyświetlony zostanie komunikat: „Podany znak jest literą". Jeśli pierwszy warunek nie będzie spełniony, wykonana zostanie fraza else if, która sprawdza, czy znak jest cyfrą. Zauważmy, że porównujemy tutaj zmienną c ze *znakami* '0' i '9', a *nie* z liczbami 0 i 9. Znaki od '0' do '9' odczytywane z terminala to co innego niż liczby od 0 do 9. Jeśli w danym systemie znaki są kodowane zgodnie z ASCII, znakowi '0' odpowiada liczba 48, znakowi '1' liczba 49 i tak dalej.

Jeśli podany znak jest cyfrą, pokazywany jest komunikat: „Podany znak to cyfra". W przeciwnym razie c nie jest ani literą, ani liczbą, więc wykonywana jest ostatnia fraza else i widzimy napis: „Podano znak specjalny"[2]. Na tym wykonywanie programu się kończy.

Trzeba tu odnotować, że choć scanf odczytuje tylko jeden znak, po podaniu tego znaku trzeba jeszcze wcisnąć klawisz *Enter*. Zawsze, kiedy odczytujemy dane z terminala, program ich nie otrzyma, póki nie wciśniemy klawisza *Enter*.

Załóżmy, że chcemy teraz napisać program, który umożliwi użytkownikowi wpisywanie prostych wyrażeń w formie:

```
liczba  operator  liczba
```

Program następnie zinterpretuje wyrażenie i wyświetli wynik z dokładnością do dwóch miejsc po przecinku. Operatory, jakie będziemy rozpoznawać, to standardowe operatory dodawania, odejmowania, mnożenia i dzielenia. Do sprawdzenia, jakie działanie ma być wykonane, program 5.8 używa wielu instrukcji if z wieloma frazami else if.

Program 5.8. **Interpretacja prostych wyrażeń**

```
/* Program interpretujący proste wyrażenia w postaci
        liczba  operator  liczba            */

#include <stdio.h>

int main (void)
{
    float value1, value2;
    char  operator;

    printf ("Podaj wyrażenie.\n");
    scanf ("%f %c %f", &value1, &operator, &value2);

    if ( operator == '+')
        printf ("%.2f\n", value1 + value2);
```

---

[1] Aby uniezależnić się od wewnętrznego sposobu reprezentacji danych, lepiej byłoby użyć bibliotecznych funkcji islower i isupper. W tym celu do programu trzeba dołączyć dyrektywę #include <ctype.h>. Na razie tego nie robimy, bo podany przykład służy jedynie do zilustrowania pewnych technik programowania.

[2] Właśnie z uwagi na sposób, w jaki kodowane są znaki w systemie ASCII, polskie znaki diakrytyczne: ą, Ą, ć, Ć i tak dalej, będą traktowane jako znaki specjalne — *przyp. tłum.*

```
    else if ( operator == '-')
        printf ("%.2f\n", value1 - value2);
    else if ( operator == '*')
        printf ("%.2f\n", value1 * value2);
    else if ( operator == '/')
        printf ("%.2f\n", value1 / value2);

    return 0;
}
```

Program 5.8. **Wyniki**

```
Podaj wyrażenie.
123.5 + 59.3
182.80
```

Program 5.8. **Wyniki (ponowne uruchomienie)**

```
Podaj wyrażenie.
198.7 / 26
7.64
```

Program 5.8. **Wyniki (ponowne uruchomienie)**

```
Podaj wyrażenie.
89.3 * 2.5
223.25
```

W wywołaniu funkcji `scanf` do zmiennych `value1`, `operator` i `value2` wczytywane są trzy wartości. Wartość zmiennoprzecinkowa może być wczytana przy użyciu formantu `%f`, tak samo jak w przypadku wypisywania liczb zmiennoprzecinkowych. W ten sposób do zmiennej `value1` wczytujemy pierwszy czynnik działania.

Następnie chcemy wczytać operator. Jest on pojedynczym znakiem (`'+'`, `'-'`, `'*'` lub `'/'`); wczytujemy go do zmiennej znakowej `operator` i używamy formantu `%c`. Spacje występujące w łańcuchu formatującym pozwalają użytkownikowi na wprowadzenie w dane miejsce dowolnej liczby białych znaków. Dzięki temu czynniki mogą być oddzielane od reszty wyrażenia. Gdybyśmy podali łańcuch formatujący `"%f%c%f"`, użytkownik nie mógłby między liczbami a operatorem wpisać żadnej spacji. Kiedy funkcja `scanf` odczytuje formant `%c`, wczytuje następny znak, *nawet jeśli jest on spacją*. Jednak trzeba też zauważyć, że z drugiej strony funkcja `scanf` *zawsze* pomija wiodące spacje, gdy wczytuje liczby. Wobec tego łańcuch formatujący `"%f %c%f"` sprawdziłby się równie dobrze jak łańcuch, którego użyliśmy w naszym przykładzie.

Po wczytaniu drugiego argumentu i zapisaniu go do zmiennej `value2` program sprawdza, czy zmienna `operator` zawiera jeden z czterech dopuszczalnych operatorów. Jeśli tak, wyniki obliczeń są pokazywane za pomocą odpowiedniej instrukcji `printf`. Na tym kończy się wykonywanie programu.

Teraz należałoby w kilku słowach ocenić nasz programu. Wprawdzie wykonuje on to, czego od niego oczekujemy, ale nie można go uznać za ukończony, gdyż w ogóle nie są

w nim brane pod uwagę pomyłki robione przez użytkownika. Co na przykład będzie, jeśli użytkownik pomyłkowo jako operator poda pytajnik „?”? Program po prostu przejdzie przez wszystkie instrukcje `if` i nie wyświetli ostrzeżenia, że podano nieprawidłowe wyrażenie.

Inną przeoczoną kwestią jest sytuacja, kiedy użytkownik jako dzielnik poda zero. Wiemy, że w C nie wolno dzielić przez zero, więc program powinien sprawdzić, czy nie zachodzi taka sytuacja.

Próba przewidzenia, gdzie program może zawieść lub dać niepożądane wyniki, oraz podjęcie stosownych działań zaradczych jest obowiązkowym elementem odpowiedzialnego programowania. Często wykonanie dostatecznie wielu testów pozwoli znaleźć potencjalnie niebezpieczne fragmenty programu. Jednak to nie wszystko. W nawyk musi wejść zadawanie sobie cały czas pytania „Co się stanie, jeśli…” oraz wstawianie do programu kodu, który pozwoli ominąć wszelkie rafy.

Program 5.8A to zmodyfikowana wersja programu 5.8, uwzględniająca dzielenie przez zero i podanie niewłaściwego operatora.

Program 5.8A. **Poprawiony program interpretujący proste wyrażenia**

```
/* Program interpretujący proste wyrażenia w postaci
        liczba  operator  liczba            */

#include <stdio.h>

int main (void)
{
    float value1, value2;
    char  operator;

    printf ("Podaj wyrażenie.\n");
    scanf ("%f %c %f", &value1, &operator, &value2);

    if ( operator == '+')
        printf ("%.2f\n", value1 + value2);
    else if ( operator == '-')
        printf ("%.2f\n", value1 - value2);
    else if ( operator == '*')
        printf ("%.2f\n", value1 * value2);
    else if ( operator == '/')
        if ( value2 == 0 )
            printf ("Dzielenie przez zero.\n");
        else
            printf ("%.2f\n", value1 / value2);
    else
        printf ("Nieznany operator.\n");

    return 0;
}
```

Program 5.8A. **Wyniki**

```
Podaj wyrażenie.
123.5 + 59.3
182.80
```

Program 5.8A. **Wyniki (ponowne uruchomienie)**

```
Podaj wyrażenie.
198.7 / 0
Dzielenie przez zero.
```

Program 5.8A. **Wyniki (ponowne uruchomienie)**

```
Podaj wyrażenie.
125 $ 28
Nieznany operator.
```

Kiedy podany operator to ukośnik, czyli dzielenie, wykonywana jest dodatkowa kontrola, czy zmienna `value2` ma wartość równą zero. Jeśli tak, wyświetlany jest stosowny komunikat. Jeśli nie, wykonywane jest dzielenie i widzimy jego wynik. Szczególną uwagę trzeba zwrócić na zagnieżdżenie instrukcji `if` i sparowanie z nimi fraz `else`.

Fraza `else` na końcu programu wyłapuje wszystkie przypadki, które nie zakwalifikowały się nigdzie wcześniej. Jeśli zatem wartość zmiennej `operator` nie pasuje do żadnego z czterech oczekiwanych znaków, wykonywana jest właśnie fraza `else`, w wyniku czego pokazywany jest napis: „Nieznany operator”.

Instrukcja switch

Łańcuch instrukcji `if-else`, z jakim mieliśmy do czynienia w ostatnim programie, kiedy wartość zmiennej porównywaliśmy z kolejnymi stałymi i innymi wartościami, jest tak powszechnie spotykany, że zdefiniowano odpowiadającą mu specjalną instrukcję. Instrukcja ta nazywa się `switch` i jej ogólna postać jest następująca:

```
switch ( wyrażenie )
{
    case wartość1:
        instrukcja
        instrukcja
            ...
        break;
    case wartość2:
        instrukcja
        instrukcja
            ...
        break;
    case wartośćn:
        instrukcja
        instrukcja
            ...
        break;
    default:
        instrukcja
        instrukcja
            ...
        break;
}
```

Ujęte w nawiasy *wyrażenie* jest porównywane kolejno z wartościami *wartość1*, *wartość2*, …, *wartośćn*, które muszą być zwykłymi stałymi lub wyrażeniami stałymi. Jeśli znaleziona zostanie wartość równa wartości *wyrażenia*, wykonywane są instrukcje występujące przy tej wartości. Zauważmy, że jeśli instrukcji tych jest wiele, nie musimy ujmować ich w nawiasy klamrowe.

Instrukcja `break` oznacza koniec danego przypadku i powoduje zakończenie wykonywania instrukcji `switch`. Trzeba ją wstawiać do każdego przypadku; w przeciwnym razie program będzie wykonywał instrukcje odpowiadające następnym wartościom.

Specjalna fraza, `default`, jest używana, kiedy *wyrażenie* nie pasuje do żadnej z podanych wcześniej wartości. Logicznie fraza ta odpowiada ostatniej frazie `else` z naszego poprzedniego programu. Zresztą pokazaną powyżej ogólną postać instrukcji `switch` możemy wyrazić, korzystając z instrukcji `if`:

```
if ( wyrażenie == wartość1 )
{
    instrukcja
    instrukcja
        ...
}
else if ( wyrażenie == wartość2 )
{
    instrukcja
    instrukcja
        ...
}
    ...
else if ( wyrażenie == wartośćn )
{
    instrukcja
    instrukcja
        ...
}
else
{
    instrukcja
    instrukcja
        ...
}
```

Możemy teraz przekształcić zestaw instrukcji `if` z programu 5.8A na odpowiadającą mu instrukcję `switch` — w ten sposób powstanie program 5.9.

Program 5.9. **Poprawiony program interpretujący proste wyrażenia, 2. wersja**

```
/* Program interpretujący proste wyrażenia w postaci
        liczba  operator  liczba          */

#include <stdio.h>

int main (void)
{
    float value1, value2;
```

```
    char  operator;

    printf ("Podaj wyrażenie.\n");
    scanf ("%f %c %f", &value1, &operator, &value2);

    switch (operator)
    {
        case '+':
            printf ("%.2f\n", value1 + value2);
            break;
        case '—':
            printf ("%.2f\n", value1 — value2);
            break;
        case '*')
            printf ("%.2f\n", value1 * value2);
            break;
        case '/')
            if ( value2 == 0 )
                printf ("Dzielenie przez zero.\n");
            else
                printf ("%.2f\n", value1 / value2);
            break;
        default:
            printf ("Nieznany operator.\n");
            break;
    }

    return 0;
}
```

Program 5.9. **Wyniki**

```
Podaj wyrażenie.
178.99 — 326.8
—147.81
```

Po wczytaniu wyrażenia wartość zmiennej `operator` jest porównywana z kolejnymi stałymi z instrukcji `switch`. Kiedy zostanie znalezione dopasowanie, wykonywane są instrukcje odpowiedniego przypadku. Instrukcja `break` powoduje przekazanie sterowania poza instrukcję `switch`, gdzie już kończy się wykonywanie całego programu. Jeśli wartość zmiennej `operator` nie odpowiada żadnemu ze zdefiniowanych przypadków, wykorzystywana jest fraza `default`, która wyświetla komunikat: „Nieznany operator”.

Instrukcja `break` we frazie `default` tak naprawdę jest zbędna, gdyż za tą frazą w instrukcji `switch` nie ma już żadnych instrukcji. Niemniej jednak dobrym zwyczajem programistycznym jest wstawianie `break` do każdego przypadku.

Pisząc instrukcję `switch`, musimy pamiętać, że nie mogą istnieć dwie takie same wartości dla różnych przypadków. Można jednak z danym zestawem instrukcji powiązać więcej niż jeden przypadek. W tym celu wystarczy wyliczyć wszystkie interesujące nas wartości (ze słowem kluczowym `case` i dwukropkiem), a potem podać wspólne instrukcje. Przykładowo poniższa instrukcja `switch` powoduje wykonanie instrukcji `printf` z mnożeniem `value1` przez `value2` wtedy, kiedy zmienna `operator` zawiera gwiazdkę lub małą literę `x`.

```
switch (operator)
{
        ...
    case '*':
    case 'x':
        printf ("%.2f\n", value1 * value2);
        break;
         ...
}
```

Zmienne logiczne

Często początkującym programistom przychodzi napisać program generujący tablicę *liczb pierwszych*. Dla przypomnienia podaję, że dodatnia liczba całkowita *p* jest liczbą pierwszą, jeśli nie dzieli się bez reszty przez żadną inną liczbę całkowitą poza jednością i samą sobą. Ustalono, że pierwsza liczba pierwsza to 2. Druga liczba pierwsza to 3, które nie dzieli się bez reszty przez żadną liczbę całkowitą poza 1 i 3, z kolei 4 *nie* jest liczbą pierwszą, gdyż dzieli się bez reszty przez 2.

Tablicę liczb pierwszych można generować na kilka sposobów. Jeśli chcemy wygenerować liczby pierwsze nieprzekraczające 50, najprostszym algorytmem jest sprawdzenie dla każdej liczby *p*, czy dzieli się przez liczby całkowite od 2 do p–1. Jeśli przez jakąś liczbę się podzieli, *p* nie jest liczbą pierwszą. Takie podejście reprezentuje program 5.10.

Program 5.10. **Generowanie tablicy liczb pierwszych**

```
// Program generujący tablicę liczb pierwszych.

#include <stdio.h>

int main (void)
{
    int    p, d;
    _Bool  isPrime;

    for ( p = 2;  p <= 50;  ++p ) {
        isPrime = 1;

        for ( d = 2;  d < p;  ++d )
            if ( p % d == 0 )
                isPrime = 0;

            if ( isPrime != 0 )
                printf ("%i  ", p);
    }

    printf ("\n");
    return 0;
}
```

Program 5.10. **Wyniki**

```
2  3  5  7  11  13  17  19  23  29  31  37  41  43  47
```

W programie 5.10 warto zwrócić uwagę na kilka rzeczy. Zewnętrzna instrukcja `for` przygotowuje pętlę działającą na liczbach od 2 do 50. Zmienna pętli — `p` — odpowiada aktualnie sprawdzanej liczbie. Pierwsza instrukcja w pętli przypisuje zmiennej `isPrime` wartość 1. Zastosowanie tej zmiennej zaraz wyjaśnię.

Druga, wewnętrzna pętla dzieli liczbę `p` przez liczby całkowite od 2 do `p`–1. W tej pętli sprawdzamy, czy reszta z dzielenia `p` przez `d` jest zerem. Jeśli tak, `p` nie może być liczbą pierwszą, gdyż dzieli się przez liczbę inną od 1 i samej siebie. Aby zaznaczyć, że `p` przestaje być kandydatem na liczbę pierwszą, ustawiamy wartość zmiennej `isPrime` na 0.

Kiedy kończy się wykonywanie pętli wewnętrznej, sprawdzana jest wartość zmiennej `isPrime`. Jeśli nie jest ona zerem, oznacza to, że nie znaleziono żadnej liczby całkowitej dzielącej `p` bez reszty, wobec tego `p` musi być liczbą pierwszą — zatem ją wyświetlamy.

Jak widać, zmienna `isPrime` przyjmuje jedną z dwóch wartości — 0 lub 1. Innych możliwości nie ma. Dlatego właśnie zadeklarowaliśmy ją jako zmienną typu `_Bool`. Zmienna ta ma wartość 1 tak długo, jak długo `p` jest kandydatem na liczbę pierwszą. Kiedy znajdujemy jej inny dzielnik, zmieniamy wartość na 0. Często tak używane zmienne nazywane są *flagami*. Zwykle flaga może mieć tylko jedną z dwóch wartości. Co więcej, flaga jest później sprawdzana — czy jest ustawiona, czy nie — i na podstawie tego sprawdzenia podejmowane są pewne działania.

W języku C ustawieniu flagi odpowiada wartość 1, jej wygaszeniu — 0. Wobec tego w programie 5.10, kiedy w pętli ustawiamy wartość `isPrime` na 1, zaznaczamy, że „`p` jest liczbą pierwszą”. Jeśli w trakcie wykonywania pętli wewnętrznej znajdziemy dzielnik `p`, wartość `isPrime` ustawiamy na 0, co oznacza, że `p` nie jest liczbą pierwszą.

Nie jest rzeczą przypadku, że wartość 1 zwykle oznacza prawdę lub włączenie flagi, a 0 oznacza fałsz lub wyłączenie. Odpowiada to ustawieniu pojedynczego bitu w komputerze. Kiedy bit jest ustawiony (włączony), jego wartością jest 1. Kiedy bit jest wyzerowany (wyłączony), ma wartość 0. Jednak w języku C za takim przypisaniem wartości logicznych przemawia jeszcze jeden, ważniejszy nawet argument. Wiąże się on ze sposobem traktowania w tym języku prawdy i fałszu.

Na początku tego rozdziału mówiliśmy, że jeśli warunki z instrukcji `if` są spełnione, program wykonuje dalsze instrukcje. Ale co oznacza owo „spełnienie”? W języku C spełnienie warunków oznacza wartość niezerową. Wobec tego instrukcja:

```
if ( 100 )
    printf ("To zdanie zostanie wyświetlone.\n");
```

powoduje wykonanie pokazanej instrukcji `printf` — warunek instrukcji `if`, tutaj 100, nie jest zerem, więc jest spełniony.

We wszystkich programach z tego rozdziału korzystamy z założenia, że wartość niezerowa oznacza spełnienie warunków, a zerowa ich niespełnienie. Kiedy więc używane wyrażenie porównania jest prawdziwe, otrzymuje wartość 1. Jeśli wyrażenie nie jest prawdziwe, otrzymuje wartość 0. Wobec tego interpretacja instrukcji:

```
if ( number < 0 )
    number = –number;
```

odbywa się następująco:

1. Interpretowane jest wyrażenie porównujące `number < 0`. Jeśli warunek jest spełniony, czyli zmienna `number` ma wartość mniejszą od zera, wartością całego wyrażenia jest 1. W przeciwnym wypadku wartością wyrażenia jest 0.
2. Instrukcja `if` sprawdza, jaki wynik dała interpretacja wyrażenia. Jeśli wynik ten nie jest zerem, wykonywana jest następna instrukcja. W przeciwnym razie instrukcja ta jest pomijana.

Powyższa analiza dotyczy także interpretacji wyrażeń z pętli `for`, `while` i `do`. Interpretacja złożonych wyrażeń, takich jak:

```
while ( char != 'e'  &&  count != 80 )
```

odbywa się tak samo. Jeśli spełnione są oba podane warunki, wynikiem całości jest 1. Jeśli któryś z warunków nie jest spełniony, wynikiem jest 0. Później sprawdzana jest wartość całego wyrażenia. Jeśli jest to 0, działanie pętli `while` jest przerywane. Jeśli nie jest to 0, wykonywanie pętli jest kontynuowane.

Wróćmy teraz do programu 5.10 i do flag. W języku C jak najbardziej poprawny jest następujący sposób sprawdzania, czy flaga jest ustawiona:

```
if ( isPrime )
```

Jest to równoważne użyciu wyrażenia:

```
if ( isPrime != 0 )
```

Aby sobie ułatwić sprawdzanie, czy flaga jest wyłączona, można użyć operatora *negacji logicznej* — `!`. W wyrażeniu:

```
if ( ! isPrime )
```

operatora tego używamy do sprawdzenia, czy zmienna `isPrime` jest ustawiona na fałsz (czyli całe wyrażenie należy czytać jako „jeśli nie zachodzi `isPrime`”). Ogólnie rzecz biorąc, wyrażenie:

```
! wyrażenie
```

stanowi negację wyrażenia *`wyrażenie`*. Jeśli *`wyrażenie`* jest zerem, jego negacja logiczna jest jedynką. Jeśli *`wyrażenie`* daje wartość niezerową, jego negacja jest zerem.

Operator negacji logicznej wykorzystujemy do przełączania wartości flagi, na przykład:

```
myMove = ! myMove;
```

Zgodnie z intuicją, operator ten ma taki sam priorytet jak jednoargumentowy operator minus, czyli ma wyższy priorytet niż wszelkie binarne operatory arytmetyczne i operatory porównania. Aby zatem sprawdzić, czy wartość zmiennej `x` jest nie mniejsza od wartości zmiennej `y`, piszemy:

```
! ( x < y )
```

Nawiasy są konieczne, aby zapewnić właściwą kolejność obliczeń. Można by oczywiście równie dobrze napisać:

```
x >= y
```

W rozdziale 3. mówiliśmy o pewnych specjalnych wartościach zdefiniowanych w języku, używanych do pracy z wyrażeniami logicznymi. Są one typu `bool`, ich wartości to `true` i `false`. Aby posłużyć się nimi, trzeba włączyć plik nagłówkowy `<stdbool.h>`. Program 5.10A stanowi odmianę programu 5.10, w której wykorzystano ten typ danych i właśnie te wartości.

Program 5.10A. **Zmodyfikowany program generujący tablicę liczb pierwszych**

```
// Program generujący tablicę liczb pierwszych.

#include <stdio.h>
#include <stdbool.h>

int main (void)
{
    int   p, d;
    bool  isPrime;

    for ( p = 2;  p <= 50;  ++p ) {
        isPrime = true;

        for ( d = 2;  d < p;  ++d )
            if ( p % d == 0 )
                isPrime = false;

            if ( isPrime != false )
                printf ("%i  ", p);
    }

    printf ("\n");
    return 0;
}
```

Program 5.10A. **Wyniki**

```
2  3  5  7  11  13  17  19  23  29  31  37  41  43  47
```

Jak widać, po włączeniu do programu pliku nagłówkowego `<stdbool.h>` możemy deklarować zmienne typu `bool` zamiast `_Bool`. Jest to kwestia wyłącznie kosmetyczna, gdyż łatwiej pisać `bool`, poza tym taka nazwa typu jest bliższa konwencji nazywania typów języka C: `int`, `float` czy `char`.

Operator wyboru

W języku C chyba najbardziej niezwykłym operatorem jest operator *wyboru*. W przeciwieństwie do innych, jedno i dwuargumentowych operatorów, ten jest trójargumentowy. Oznacza się go za pomocą dwóch symboli, pytajnika — `?` — i dwukropka — `:`. Pierwszy argument jest umieszczany przed pytajnikiem, drugi między pytajnikiem a dwukropkiem, a trzeci za dwukropkiem.

Ogólna postać operatora wyboru jest następująca:

```
warunek ? wyrażenie1 : wyrażenie2
```

gdzie *warunek* to wyrażenie, zwykle porównania, które jest interpretowane jako pierwsze. Jeśli *warunek* jest prawdziwy, interpretowane jest *wyrażenie1* i ono staje się wynikiem całej operacji. Jeśli *warunek* jest fałszywy (czyli jest zerem), interpretowane jest *wyrażenie2* i to ono staje się wynikiem całej operacji.

Operator wyboru jest najczęściej używany do przypisania zmiennej jednej z dwóch wartości, w zależności od spełnienia pewnego warunku. Załóżmy na przykład, że mamy zmienne całkowitoliczbowe `x` i `s`. Jeśli zmiennej `s` chcemy przypisać wartość –1 dla `x` mniejszego od zera lub wartość x^2 w przeciwnym wypadku, to możemy napisać:

```
s = ( x < 0 ) ? –1 : x * x;
```

Przy interpretacji takiego wyrażenia najpierw sprawdzany jest warunek `x < 0`. Dookoła warunku umieszcza się nawiasy, aby poprawić czytelność całego wyrażenia. Zwykle nie jest to konieczne, gdyż priorytet operatora wyboru jest bardzo niski — niższy od wszystkich innych operatorów poza przypisaniem i przecinkiem.

Jeśli wartość zmiennej `x` jest mniejsza od zera, wyliczane jest wyrażenie znajdujące się za pytajnikiem. Jest to po prostu stała –1, która jest przypisywana zmiennej `s`, jeśli `x` jest mniejsze niż zero.

Jeśli wartość `x` nie jest mniejsza od zera, wyliczane jest wyrażenie za dwukropkiem i jego wartość jest przypisywana do zmiennej `s`. Jeśli zatem `x` jest większe lub równe zeru, zmiennej `s` przypisywana jest wartość `x * x`, czyli x^2.

Oto następny przykład użycia operatora wyboru — zmiennej `maxValue` przypisywana jest większa z wartości `a` i `b`:

```
maxValue = ( a > b ) ? a : b;
```

Jeśli wyrażenie użyte po dwukropku (odpowiednik frazy `else`) zawiera inny operator wyboru, można uzyskać efekt frazy `else if`. Na przykład funkcję *signum* z programu 5.6 możemy zapisać w jednym wierszu, korzystając z dwóch operatorów wyboru:

```
sign = ( number < 0 ) ? –1 : (( number == 0 ) ? 0 : 1);
```

Jeśli wartość zmiennej `number` jest mniejsza od zera, `sign` otrzymuje wartość –1. Gdy `number` ma wartość `0`, `sign` też otrzymuje wartość `0`. Jeśli żaden z dotychczasowych warunków nie jest spełniony, `sign` otrzymuje wartość `1`. Nawiasy dookoła części „else" powyższego wyrażenia nie są konieczne. Wynika to stąd, że operator wyboru jest łączony od prawej strony do lewej, czyli wielokrotnie użyty w jednym wyrażeniu, na przykład:

```
e1 ? e2 : e3 ? e4 : e5
```

jest grupowany od prawej do lewej, więc pokazane wyrażenie zostanie zinterpretowane jako:

```
e1 ? e2 : ( e3 ? e4 : e5 )
```

Operator wyboru nie musi być używany koniecznie po prawej stronie przypisania; można go też wykorzystać wszędzie tam, gdzie dozwolone jest użycie jakiegokolwiek wyrażenia. Oznacza to, że można wyświetlić znak zmiennej `number` bez uprzedniego przypisywania tego znaku innej zmiennej, tak jak w poniższej instrukcji `printf`:

```
printf ("Signum = %i\n", ( number < 0 ) ? –1 : ( number == 0 ) ? 0 : 1);
```

Operator wyboru jest bardzo przydatny w *makrach* preprocesora C. Szczegółami zajmiemy się w rozdziale 12.

Na tym kończymy omawianie decyzji w języku C. W rozdziale 6. zapoznamy się z bardziej złożonymi typami danych. *Tablica* to naprawdę użyteczne narzędzie, używane w bardzo wielu programach pisanych w C. Zanim przejdziemy dalej, utrwalimy materiał z tego rozdziału, wykonując poniższe ćwiczenia.

Ćwiczenia

1. Przepisz i uruchom dwanaście programów pokazanych w tym rozdziale. Uzyskane wyniki porównaj z wynikami pokazanymi w tekście. Poeksperymentuj z tymi programami, podając wartości inne, niż pokazano w książce.
2. Napisz program proszący użytkownika o podanie dwóch liczb całkowitych. Sprawdź, czy pierwsza z liczb jest podzielna przez drugą, i wyświetl odpowiedni komunikat.
3. Napisz program przyjmujący od użytkownika dwie liczby całkowite. Pokaż wynik dzielenia pierwszej liczby przez drugą z dokładnością do trzech miejsc po przecinku. Pamiętaj o ochronie przed dzieleniem przez zero.
4. Napisz program działający jako prosty kalkulator „drukujący”. Program powinien umożliwiać użytkownikowi wprowadzanie wyrażeń w postaci:

   ```
   liczba operator
   ```

   Program ma „rozumieć” następujące operatory:

   ```
   +   –   *   /   S   E
   ```

   Operator `S` nakazuje programowi wstawienie do „akumulatora” podanej liczby.

   Operator `E` nakazuje programowi zakończenie obliczeń.

   Działania arytmetyczne dotyczą akumulatora jako pierwszego argumentu i podanej liczby jako drugiego argumentu. Oto przykładowy „wydruk” działania programu:

   ```
   Początek obliczeń
   10 S            Ustaw akumulator na 10
   = 10.000000     Zawartość akumulatora
   ```

```
2 /             Dzielenie przez 2
= 5.000000      Zawartość akumulatora
55 –            Odjęcie 55
–50.000000
100.25 S        Ustawienie akumulatora na 100.25
= 100.250000
4 *             Mnożenie przez 4
= 401.000000
0 E             Koniec działania programu
= 401.000000
Koniec obliczeń.
```

Sprawdzaj, czy program nie dzieli przez zero; wskaż nieznane operatory.

5. Przygotowaliśmy program 5.9, odwracający kolejność cyfr liczby całkowitej podanej z terminala. Program ten jednak nie zadziała poprawnie, jeśli zostanie mu podana liczba ujemna. Sprawdź, co się stanie w takim wypadku, i zmodyfikuj program tak, aby prawidłowo obsługiwał liczby ujemne. Jeśli na przykład użytkownik poda liczbę –8645, program powinien odpowiedzieć i napisać 5468–.

6. Napisz program pobierający od użytkownika liczbę całkowitą i podający polskie nazwy poszczególnych cyfr. Jeśli zatem użytkownik napisze 932, program ma odpowiedzieć:

   ```
   dziewięć trzy dwa
   ```

   Pamiętajmy o wyświetleniu napisu „zero”, jeśli użytkownik wpisze samo 0. (Uwaga! To jest trudne ćwiczenie).

7. Program 5.10 ma kilka niedoskonałości. Pierwszą z nich jest sprawdzanie liczb parzystych. Jasne, że żadna liczba parzysta większa od 2 nie może być liczbą pierwszą, więc liczby parzyste nie powinny być w ogóle sprawdzane; tak samo liczby parzyste, poza 2, nie powinny być traktowane jako potencjalne dzielniki. Wewnętrzna pętla `for` jest niewydajna, gdyż wartość zmiennej `p` jest dzielona *zawsze* przez wszystkie wartości od 2 do `p`–1. Można tego uniknąć, jeśli w pętli `for` na bieżąco będziemy sprawdzać, jaką wartość ma zmienna `isPrime`. To pozwoli zatrzymać działanie pętli `for`, jeśli zostanie już znaleziony jakiś dzielnik. Zmodyfikuj program 5.10 tak, aby uwzględniał dwa opisane udoskonalenia. Uruchom program, aby sprawdzić poprawność jego działania. (Uwaga! W rozdziale 6. zobaczymy, jak można jeszcze sprawniej generować liczby pierwsze).

6

Tablice

Język C umożliwia definiowanie uporządkowanych zbiorów danych nazywanych *tablicami*. W tym rozdziale powiemy, jak należy definiować tablice i jak ich używać. Następne rozdziały poszerzą naszą wiedzę o tablicach; pokażemy, jak się mają do funkcji, struktur, łańcuchów znakowych i wskaźników. Jednak najpierw musisz poznać podstawowe zagadnienia dotyczące tablic, a konkretnie:

- tworzenie prostych tablic;
- inicjowanie tablic;
- praca z tablicami znaków;
- zastosowanie słowa kluczowego `const`;
- implementowanie tablic wielowymiarowych;
- tworzenie tablic o zmiennym rozmiarze.

Załóżmy, że mamy zbiór ocen `grades`, który mamy wczytać do komputera; załóżmy też, że potem chcemy te oceny jakoś przetwarzać, porządkować je rosnąco, wyliczać średnią czy znajdować medianę. W programie 5.2 wyliczaliśmy średnią zbioru ocen, po prostu dodając kolejne oceny w miarę ich wczytywania. Jeśli jednak chcemy na przykład oceny uporządkować, musimy zrobić coś więcej. Jeśli mamy przetwarzać zbiór ocen `grades`, szybko stwierdzamy, że nie możemy tego robić, póki nie będziemy znali wszystkich ocen ze zbioru. Wobec tego, korzystając z opisanej poprzednio techniki, możemy wczytywać wszystkie oceny i zapisywać je do kolejnych zmiennych, na przykład tak:

```
printf ("Podaje ocenę 1\n");
scanf ("%i", &grade1);
printf ("Podaje ocenę 2\n");
scanf ("%i", &grade2);
    ...
```

Kiedy podane zostaną oceny, możemy je układać. W tym celu wystarczy użyć zestawu instrukcji `if` porównujących oceny tak, aby znaleźć ocenę najmniejszą, potem drugą od końca i tak dalej, aż do oceny najwyższej. Jeśli faktycznie napiszemy taki program, szybko stwierdzimy, że już dla list o rozsądnej wielkości (rozsądną wielkością jest na przykład

10 ocen) uzyskany program jest długi i skomplikowany. Jednak nie wszystko jest stracone — wykorzystajmy tablice.

Definiowanie tablicy

Możemy zdefiniować zmienną grades, która nie będzie zawierała *pojedynczej* oceny, ale cały *zestaw ocen*. Każdy element takiego zestawu może być wskazany przez liczbę nazywaną *indeksem*. W zapisie matematycznym numerowane zmienne zapisujemy przez x_p — oznacza to *i*-ty element *x* z danego zbioru, natomiast w języku C analogiczny zapis wygląda następująco:

```
x[i]
```

Wobec tego wyrażenie

```
grades[5]
```

będzie się odnosiło do piątego elementu tablicy grades. Jednak elementy tablic są numerowane od zera, więc:

```
grades[0]
```

oznacza pierwszy element tablicy (z tego powodu łatwiej mówić o zerowym, a nie o pierwszym elemencie).

Poszczególne elementy tablicy mogą być używane wszędzie tam, gdzie wykorzystujemy zwykłe zmienne, na przykład możemy zwykłej zmiennej przypisać wartość z tablicy za pomocą instrukcji:

```
g = grades[50];
```

Instrukcja taka pobiera wartość z elementu grades[50] i przypisuje ją zmiennej g. Jeśli zatem i jest zmienną całkowitoliczbową, instrukcja:

```
g = grades[i];
```

pobiera z tablicy grades element numer i i przypisuje go zmiennej g. Jeśli zatem w danej chwili i jest równe 7, zmiennej g zostanie przypisana wartość grades[7].

Wartość możemy zapisać w elemencie tablicy, po prostu umieszczając odwołanie do tego elementu po lewej stronie operatora przypisania. Instrukcja:

```
grades[100] = 95;
```

powoduje zapisanie wartości 95 w tablicy grades, w komórce numer 100. Instrukcja:

```
grades[i] = g;
```

powoduje przypisanie komórce numeru i wartości zmiennej g.

Możliwość zapisywania w pojedynczej tablicy całych zestawów powiązanych ze sobą danych ułatwia tworzenie zwartych i wydajnych programów. Korzystając na przykład z dodatkowej zmiennej jako indeksu, możemy łatwo przejrzeć wszystkie elementy tablicy. Wobec tego pętla for w postaci:

```
for ( i = 0;  i < 100;  ++i )
    sum += grades[i];
```

powoduje przejrzenie pierwszych 100 elementów tablicy `grades` (elementy o indeksach od 0 do 99) oraz dodanie wartości każdej oceny do zmiennej `sum`. Po skończeniu działania tej pętli zmienna `sum` zawiera sumę pierwszych stu wartości z tablicy `grades` (zakładamy, że przed wejściem do pętli zmienna `sum` została ustawiona na zero).

Używając tablic, musimy pamiętać, że pierwszy element tablicy ma indeks zero, a ostatni jest o jeden mniejszy od liczby elementów.

Do wskazywania poszczególnych elementów tablicy można używać — oprócz stałych liczb całkowitych — wyrażeń dających w wyniku liczbę całkowitą. Jeśli zatem zmienne `low` i `high` są zmiennymi typu `int`, instrukcja:

```
next_value = sorted_data[(low + high) / 2];
```

powoduje przypisanie zmiennej `next_value` wartości elementu tablicy, którego indeks jest równy `(low + high) / 2`. Jeśli na przykład `low` ma wartość `1`, a `high` wartość `9`, zmiennej `next_value` zostanie przypisany element `sorted_data[5]`. Jeśli `low` ma wartość `1`, a `high` wartość `10`, zmiennej `next_value` zostanie przypisany element `sorted_data[5]`, gdyż całkowitoliczbowe dzielenie 11 przez 2 daje 5.

Tak jak zwykłe zmienne, tak i tablice muszą być przed użyciem zadeklarowane. Deklaracja tablicy oznacza zadeklarowanie typu elementów tej tablicy (na przykład `int`, `float` czy `char`) oraz podanie maksymalnej liczby elementów, które w tablicy się zmieszczą (kompilator C potrzebuje tej ostatniej informacji, aby zdecydować, ile pamięci trzeba zarezerwować na tablicę).

Przykładowo deklaracja:

```
int grades[100];
```

powoduje utworzenie tablicy zawierającej 100 liczb całkowitych. Indeksy tej tablicy mogą być liczbami od `0` do `99`. Jednak to programista jest odpowiedzialny za używanie tylko prawidłowych indeksów, gdyż język C tego nie sprawdza. Wobec tego użycie indeksu `150` w odniesieniu do powyższej tablicy `grades` może wcale nie spowodować błędu, ale zwykle wywoła niepożądane i nieprzewidywalne zachowanie programu.

Aby zadeklarować tablicę `averages` z 200 liczbami zmiennoprzecinkowymi, piszemy:

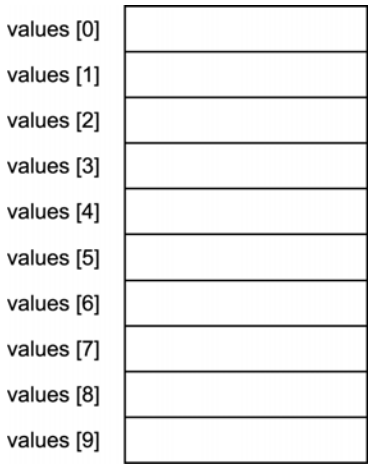
```
float average[200];
```

Deklaracja taka powoduje zarezerwowanie dostatecznej ilości pamięci na 200 liczb typu `float`. Analogicznie deklaracja:

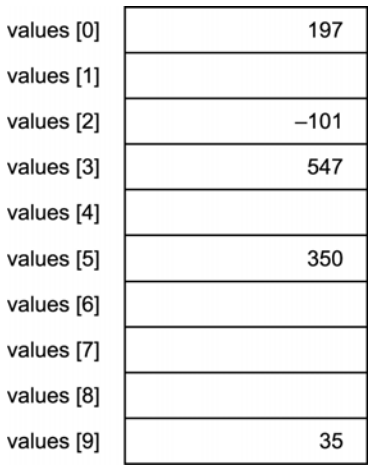
```
int values[10];c
```

powoduje zarezerwowanie miejsca na tablicę `values` mogącą przechowywać do 10 liczb całkowitych. Aby lepiej zdać sobie sprawę, jak jest rezerwowana pamięć, wystarczy popatrzeć na rysunek 6.1.

Te elementy tablicy, które zadeklarowano jako `int`, `float` czy `char`, można wykorzystywać tak samo jak zwykłe zmienne odpowiednich typów. Możemy im przypisywać wartości, wyświetlać ich wartości, dodawać je, odejmować i tak dalej. Jeśli zatem wykonamy pokazane poniżej instrukcje, tablica `values` będzie zawierała dane takie jak na rysunku 6.2.



Rysunek 6.1. Ułożenie tablicy values w pamięci



Rysunek 6.2. Tablica values z ustawionymi niektórymi komórkami

```
int values[10];

values[0] = 197;
values[2] = –100;
values[5] = 350;
values[3] = values[0] + values[5];
values[9] = values[5] / 10;
––values[2];
```

Pierwsza instrukcja przypisania powoduje umieszczenie w `values[0]` wartości 197. Analogicznie instrukcje druga i trzecia umieszczają w elementach `values[2]` i `values[5]` wartości –100 i 350. Następna instrukcja dodaje zawartość `values[0]` (czyli 197)

do zawartości `values[5]` (czyli 350); wynik (547) jest zapisywany w `values[3]`. W następnej instrukcji wartość z `values[5]` (350) jest dzielona przez 10; wynik jest zapisywany w `values[9]`. Ostatnia instrukcja zmniejsza wartość elementu `values[2]`, czyli zmienia tę wartość z –100 na –101.

Powyższe instrukcje zostały włączone do programu 6.1. Pętla `for` przechodzi po wszystkich elementach tablicy i pokazuje ich wartości.

Program 6.1. **Użycie tablicy**

```
#include <stdio.h>

int main (void)
{
    int  values[10];
    int  index;

    values[0] = 197;
    values[2] = -100;
    values[5] = 350;
    values[3] = values[0] + values[5];
    values[9] = values[5] / 10;
    --values[2];

    for ( index = 0;  index < 10;  ++index )
        printf ("values[%i] = %i\n", index, values[index]);

    return 0;
}
```

Program 6.1. **Wyniki**

```
values[0] = 197
values[1] = -2
values[2] = -101
values[3] = 547
values[4] = 4200224
values[5] = 350
values[6] = 4200326
values[7] = 4200224
values[8] = 8600872
values[9] = 35
```

Zmienna `index` przybiera wartości od 0 do 9, ponieważ ostatni element tablicy ma numer zawsze o jeden mniejszy od liczby elementów — z uwagi na istnienie elementu zerowego. W ogóle nie przypisaliśmy wartości pięciu elementom tablicy (o numerach 1, 4 i od 6 do 8); wartości pokazywane dla tych komórek są przypadkowe. Wprawdzie program pokazuje zera, jednak wartość niezainicjalizowanej zmiennej lub komórki tablicy jest nieokreślona. Z tego powodu nie wolno nigdy czynić założeń dotyczących wartości komórki tablicy, której nie nadano wartości.

Użycie tablic jako liczników

Teraz zajmijmy się nieco bardziej praktycznym przykładem. Załóżmy, że robimy ankietę telefoniczną, aby poznać opinie o pewnym programie telewizyjnym, i prosimy każdego ankietowanego o ocenienie go w skali od 1 do 10. Po przepytaniu 5000 osób mamy 5000 liczb. Teraz chcemy przeanalizować wyniki. Jedna z najbardziej potrzebnych danych to tablica z rozkładem ocen — chcemy wiedzieć, ile osób oceniło program na 1, ile na 2 i tak dalej, aż do 10.

Ręczne przeglądanie wszystkich ocen — choć możliwe — byłoby uciążliwe. Poza tym czasem odpowiedzi nie ograniczają się do dziesięciu opcji (weźmy pod uwagę choćby uwzględnienie wieku respondenta). Zatem chcemy przygotować program zliczający odpowiedzi z poszczególnymi ocenami. W pierwszym odruchu moglibyśmy zdefiniować dziesięć różnych liczników, na przykład `rating_1` do `rating_10`, a następnie zwiększać ich wartość po napotkaniu oceny odpowiadającej danemu licznikowi. Jednak znowu może być więcej możliwych odpowiedzi, a wtedy takie rozwiązanie staje się niewygodne w użyciu. Poza tym użycie tablicy jest rozwiązaniem znacznie „czystszym".

Możemy przygotować tablicę liczników, dajmy na to `ratingCounters`, a potem będziemy zwiększać odpowiednie liczniki podczas podawania następnych odpowiedzi. Aby nie marnować papieru, w programie 6.2 założymy, że mamy do czynienia z jedynie 20 odpowiedziami. Zresztą dobrą praktyką jest przetestowanie programu na niewielkiej próbce danych, a dopiero potem analizowanie kompletu. W przypadku dużego zbioru danych znacznie trudniej znaleźć błędy i usunąć je.

Program 6.2. **Użycie tablicy liczników**

```
#include <stdio.h>

int main (void)
{
    int ratingCounters[11], i, response;

    for ( i = 1;  i <= 10;  ++i )
        ratingCounters[i] = 0;

    printf ("Podawaj odpowiedzi\n");
    for ( i = 1;  i <= 20;  ++i ) {
        scanf ("%i", &response);

        if ( response < 1 || response > 10 )
            printf ("Nieprawidłowa odpowiedź: %i\n", response);
        else
            ++ratingCounters[response];
    }

    printf ("\n\nOcena    Liczba odpowiedzi\n");
    printf ("------ -------------------\n");

    for ( i = 1;  i <= 10;  ++i )
        printf ("%4i%14i\n", i, ratingCounters[i]);

    return 0;
}
```

Program 6.2. **Wyniki**

```
Podawaj odpowiedzi
6
5
8
3
9
6
5
7
15
Nieprawidłowa odpowiedź: 15
5
5
1
7
4
10
5
5
6
8
9


Ocena    Liczba odpowiedzi
------ -------------------
   1              1
   2              0
   3              1
   4              1
   5              6
   6              3
   7              2
   8              2
   9              2
  10              1
```

Tablicę `ratingCounters` zdefiniowaliśmy tak, aby miała 11 elementów. Można zapytać, po co właściwie 11 elementów, skoro możliwych ocen jest 10? Chodzi o sposób zliczania odpowiedzi w ramach poszczególnych grup ocen. Każda odpowiedź jest liczbą od 1 do 10, program zlicza te odpowiedzi, zwiększając wartość odpowiedniej komórki tablicy (sprawdza najpierw, czy ocena mieści się w zakresie od 1 do 10). Jeśli na przykład mamy ocenę 5, o jeden zwiększamy wartość komórki `ratingCounters[5]`. Gdy korzystamy z tej techniki, liczbę osób, które oceniły program na 5, mamy w komórce `ratingCounters[5]`.

Teraz chyba powinna być jasna przyczyna użycia 11 elementów zamiast 10. Najwyższa możliwa ocena to 10, więc musimy mieć komórkę `ratingCounters[10]`, a zatem elementów musi być 11, o jeden więcej niż największy indeks. Nie ma oceny 0, więc komórka `ratingCounters[0]` nigdy nie zostanie użyta. Kiedy pętla `for` inicjalizuje i pokazuje zawartość tablicy, zmienna sterująca pętli `i` przybiera wartości od 1, zatem element `ratingCounters[0]` jest pomijany.

Można by też użyć tabeli zawierającej jedynie 10 elementów. Wtedy każdą odpowiedź podaną przez użytkownika — `response` — wpisywalibyśmy do komórki `ratingCounters[response – 1]`. Dzięki temu `ratingCounters[0]` zawierałoby liczbę ocen na 1, `ratingCounters[1]` liczbę ocen na 2 i tak dalej. Jest to rozwiązanie równie dobre jak poprzednie. Nie użyliśmy go dlatego, że bardziej naturalne jest zapisywanie liczby ocen *n* w komórce `ratingCounters[n]`.

Generowanie ciągu Fibonacciego

Przeanalizujmy program 6.3, generujący 15 pierwszych elementów ciągu *Fibonacciego*, i spróbujmy powiedzieć, jakie uzyskamy wyniki. Jak powiązane są poszczególne liczby z tablicy?

Program 6.3. **Generowanie ciągu Fibonacciego**

```
// Program generujący 15 pierwszych elementów ciągu Fibonacciego
#include <stdio.h>

int main (void)
{
    int  Fibonacci[15], i;

    Fibonacci[0] = 0;     // z definicji
    Fibonacci[1] = 1;     // jak wyżej

    for ( i = 2;  i < 15;  ++i )
        Fibonacci[i] = Fibonacci[i–2] + Fibonacci[i–1];

    for ( i = 0;  i < 15;  ++i )
        printf ("%i\n", Fibonacci[i]);

    return 0;
}
```

Program 6.3. **Wyniki**

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
```

Pierwsze dwa elementy ciągu Fibonacciego, oznaczane przez F_0 i F_1, z definicji są równe odpowiednio 0 i 1. Każda następna liczba tego ciągu — F_i — jest sumą dwóch liczb poprzednich, F_{i-2} i F_{i-1}. Zatem F_2 wyliczamy, dodając do siebie F_0 i F_1. W powyższym programie zapisujemy to, wyliczając `Fibonacci[2]` jako sumę `Fibonacci[0]` i `Fibonacci[1]`. Obliczenia te wykonywane są w pętli `for`, która wylicza od F_2 do F_{14} (czyli od `Fibonacci[2]` do `Fibonacci[14]`).

Ciągi Fibonacciego są ważne w matematyce i teorii algorytmów. Ciąg ten powstał w związku z „problemem królików" — jeśli mamy na początek parę królików i zakładamy, że każda para daje nową parę co miesiąc, że nowa para królików ma potomstwo na koniec swojego drugiego miesiąca życia i że króliki żyją wiecznie, należy przewidzieć, ile królików będzie na koniec roku. Odpowiedź na tak postawione pytanie bazuje na tym, że na koniec n-tego miesiąca będzie łącznie F_{n+2} par królików. Wobec tego, zgodnie z tabelą uzyskaną z programu 6.3, na koniec 12. miesiąca będzie 377 par królików.

Zastosowanie tablic do generowania liczb pierwszych

Teraz nadszedł czas, by wrócić do naszego programu z rozdziału 5. generującego liczby pierwsze. Zastanówmy się, jak możemy użyć tablic, aby program ten udoskonalić. W programie 5.10A dzieliliśmy badaną liczbę przez kolejne liczby całkowite od 2 do liczby o jeden mniejszej od liczby badanej, aby stwierdzić, czy liczba jest pierwsza. W ćwiczeniu 7. z rozdziału 5. zauważyliśmy dwie niedoskonałości programu łatwe do poprawienia. Jednak nawet po tych zmianach program nadal działał mało wydajnie. Wprawdzie tego typu niedoskonałość niewiele znaczy, kiedy mamy 50 liczb, to przy szukaniu liczb pierwszych do 100 000 problem narasta.

Udoskonalona metoda generowania liczb pierwszych opiera się na spostrzeżeniu, że liczba jest pierwsza, jeśli nie dzieli się bez reszty przez żadną inną liczbę pierwszą. Wynika to stąd, że każdą liczbę złożoną (czyli niepierwszą) można zapisać jako iloczyn liczb pierwszych; na przykład 20 ma czynniki pierwsze 2, 2 i 5. Korzystając z tego spostrzeżenia, możemy przygotować lepszy program wskazujący liczby pierwsze. Program będzie badał, czy dana liczba jest pierwsza, za pomocą sprawdzenia, czy dzieli się ona bez reszty przez którąkolwiek wcześniej znalezioną liczbę pierwszą. Wzmianka o liczbach „wcześniej znalezionych" od razu nasuwa na myśl skorzystanie z tablicy. W tablicy tej będziemy przechowywali wszystkie kolejno generowane liczby pierwsze.

Optymalizując program dalej, łatwo możemy wykazać, że każda liczba złożona n w swoim rozkładzie musi mieć liczbę mniejszą lub równą pierwiastkowi kwadratowemu z n. Wobec tego wystarczy sprawdzać podzielność badanej liczby jedynie przez liczby pierwsze nie większe od jej pierwiastka kwadratowego.

Program 6.4 wykorzystuje wszystkie te wnioski do wygenerowania liczb pierwszych nieprzekraczających 50.

Program 6.4. **Poprawiony program generujący liczby pierwsze, 2. wersja**

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

// Zmodyfikowany program generujący liczby pierwsze

int main (void)
{
    int  p, i, primes[50], primeIndex = 2;
    bool isPrime;

    primes[0] = 2;
    primes[1] = 3;

    for ( p = 5;  p <= 50;  p = p + 2 ) {
        isPrime = true;

        for ( i = 1;  isPrime  &&  p / primes[i] >= primes[i]; ++i )
            if ( p % primes[i] == 0 )
                isPrime = false;

        if ( isPrime == true ) {
            primes[primeIndex] = p;
            ++primeIndex;
        }
    }

    for ( i = 0;  i < primeIndex;  ++i )
        printf ("%i  ", primes[i]);

    printf ("\n");

    return 0;
}
```

Program 6.4. **Wyniki**

```
2  3  5  7  11  13  17  19  23  29  31  37  41  43  47
```

W wewnętrznej pętli `for` używane jest wyrażenie:

```
p / primes[i] >= primes[i]
```

sprawdzające, czy wartość `p` nie przekracza pierwiastka liczby `primes[i]`. Ograniczenie to wynika bezpośrednio z powyższej analizy.

Program 6.4 zaczyna się od zapisania liczb 2 i 3 jako pierwszych dwóch liczb pierwszych w tablicy `primes`. Tablica ta zawiera 50 elementów, choć jasne jest, że nie będziemy potrzebować ich aż tylu. Zmienna `primeIndex` początkowo ma wartość 2, czyli wskazuje pierwsze wolne miejsce w tablicy `primes`. Następnie mamy pętlę `for` przechodzącą przez liczby nieparzyste od 5 do 50. Kiedy wartość logiczna `isPrime` zostanie ustawiona na `true`, wchodzimy do kolejnej pętli `for`. Pętla ta dzieli liczbę `p` przez wszystkie wygenerowane dotąd liczb pierwsze z tablicy `primes`. Indeks `i` zaczyna działanie od 1, gdyż nie trzeba sprawdzać podzielności badanych liczb przez `primes[0]` (czyli przez 2). Po prostu nasz

program nie traktuje liczb parzystych jako potencjalnie pierwszych. W tej pętli sprawdzamy, czy p dzieli się bez reszty przez primes[i] i jeśli tak, ustawiamy isPrime na false. Pętla for działa tak długo, jak długo isPrime ma wartość true *oraz* wartość primes[i] nie przekracza pierwiastka kwadratowego z p.

Po wyjściu z pętli for sprawdzamy flagę isPrime i na tej podstawie decydujemy, czy wartość p należy wstawić do tablicy primes jako następną liczbę pierwszą.

Kiedy zbadamy już wszystkie wartości p, program pokazuje wszystkie liczby pierwsze z tablicy primes. Wartość zmiennej indeksującej i przechodzi od 0 do primeIndex – 1, gdyż primeIndex zawsze wskazuje *następną* wolną komórkę tablicy primes.

Inicjalizowanie tablic

Tak jak można inicjalizować zmienne w chwili ich deklaracji, tak samo można przypisać wartości początkowe elementom tablicy. Dokonujemy tego po prostu przez podanie wartości kolejnych komórek, od pierwszej poczynając. Wartości te rozdzielamy przecinkami, a całość zamykamy w nawiasy klamrowe.

Instrukcja:

```
int counters[5] = { 0, 0, 0, 0, 0 };
```

deklaruje tablicę counters zawierającą pięć wartości typu int oraz inicjalizuje wszystkie komórki tej tablicy na zero. Analogicznie instrukcja:

```
int integers[5] = { 0, 1, 2, 3, 4 };
```

ustawia integers[0] na 0, integers[1] — na 1, integers[2] — na 2 i tak dalej.

Podobnie inicjalizuje się tablice znaków; instrukcja:

```
char letters[5] = {'a', 'b', 'c', 'd', 'e' };
```

definiuje tablicę znakową letters i inicjalizuje jej pięć elementów znakami 'a', 'b', 'c', 'd' oraz 'e'.

Nie trzeba inicjalizować całej tablicy. Jeśli podamy mniej wartości, niż tablica ma komórek, zainicjalizowana zostanie tylko odpowiednia część komórek. Pozostałe wartości zostaną ustawione na zero, więc deklaracja:

```
float sample_data[500] = { 100.0, 300.0, 500.5 };
```

inicjalizuje pierwsze trzy wartości tablicy sample_data na 100.0, 300.0 i 500.5, pozostałe 497 elementów otrzymuje wartość zero.

Zamykając numer elementu w nawiasach kwadratowych, możemy inicjalizować tylko wybrane komórki tablicy. Na przykład:

```
float sample_data[500] = { [2] = 500.5, [1] = 300.0, [0] = 100.0 };
```

inicjalizuje tablicę sample_data tak samo jak instrukcja poprzednia.

Z kolei instrukcje:

```
int x = 1233;
int a[10] = { [9] = x + 1, [2] = 3, [1] = 2, [0] = 1 };
```

definiują 10-elementową tablicę i inicjalizują jej ostatni element za pomocą wartości `x + 1` (czyli `1234`), a pierwsze trzy elementy odpowiednio przy użyciu wartości `1`, `2` i `3`.

Niestety, C nie zawiera żadnego skrótowego mechanizmu służącego do inicjalizowania elementów tablicy. Wobec tego nie można nakazać inicjalizacji wielu komórek jedną wartością; gdybyśmy na przykład musieli ustawić wszystkie 500 wartości tablicy `sample_data` na 1, trzeba by tę jedynkę napisać 500 razy. W takim wypadku lepiej inicjalizację przeprowadzajmy programowo, korzystając z pętli `for`.

Program 6.5 pokazuje oba omówione rodzaje inicjalizacji tablic.

Program 6.5. **Inicjalizacja tablic**

```
#include <stdio.h>

int main (void)
{
    int array_values[10] = { 0, 1, 4, 9, 16 };
    int i;

    for ( i = 5;  i < 10;  ++i )
        array_values[i] = i * i;

    for ( i = 0;  i < 10;  ++i )
        printf ("Array_values[%i] = %i\n", i, array_values[i]);

    return 0;
}
```

Program 6.5. **Wyniki**

```
array_values[0] = 0
array_values[1] = 1
array_values[2] = 4
array_values[3] = 9
array_values[4] = 16
array_values[5] = 25
array_values[6] = 36
array_values[7] = 49
array_values[8] = 64
array_values[9] = 81
```

W deklaracji tablicy `array_values` pierwszych pięć elementów inicjalizujemy kwadratami ich numerów (na przykład element `3` ma wartość 3^2, czyli 9). Pierwsza pętla `for` pokazuje tę samą inicjalizację wykonaną właśnie w pętli programowej. Pętla ta ustawia elementy od `5` do `9` na kwadraty numerów tych elementów. Druga pętla `for` po prostu przebiega przez wszystkie 10 elementów w celu wyświetlenia ich wartości.

Tablice znakowe

Program 6.6 ma po prostu pokazać, jak używa się tablic znakowych. W tym programie warto jednak zwrócić uwagę na jedną rzecz — jaką?

Program 6.6. **Tablice znakowe**

```
#include <stdio.h>

int main (void)
{
    char word[] = { 'H', 'e', 'l', 'l', 'o', '!' };
    int  i;

    for ( i = 0;  i < 6;  ++i )
        printf ("%c", word[i]);

    printf ("\n");

    return 0;
}
```

Program 6.6. **Wyniki**

```
Hello!
```

W powyższym programie najważniejsza jest deklaracja tablicy znakowej `word`. Nie jest w niej podana liczba elementów. Język C pozwala na takie definiowanie tablicy; w takim wypadku wielkość tej tablicy jest wyznaczana automatycznie na podstawie ilości podanych wyrażeń inicjalizujących. Program 6.6 ma sześć wartości inicjalizujących tablicę `word`, więc kompilator niejawnie ustali wielkość tej tablicy na 6.

Takie rozwiązanie dobrze się sprawdza tak długo, jak długo inicjalizujemy wszystkie elementy tablicy. Jeśli nie, wymiar musimy podać jawnie.

Jeśli używamy w inicjalizacji numerów indeksów, jak poniżej:

```
float sample_data[] = { [0] = 1.0, [49] = 100.0, [99] = 200.0 };
```

wielkość tablicy jest określona największym indeksem. Wtedy tablica `sample_data` będzie zawierała 100 elementów, gdyż największy podany indeks to 99.

Użycie tablic do zamiany podstawy liczb

W następnym programie pokazujemy użycie tablic liczb całkowitych i tablic znakowych. Chodzi o utworzenie programu konwertującego liczby dodatnie w zapisie dziesiętnym na ich odpowiedniki w innych systemach, o podstawie nieprzekraczającej 16. Jako dane wejściowe podajemy konwertowaną liczbę i podstawę docelowego systemu liczbowego. Wtedy program przekształca liczbę i podaje wynik.

Pierwszym etapem jest przygotowanie algorytmu konwertującego liczbę z systemu o podstawie 10 na inny system. Algorytm generujący cyfry z wynikowej liczby możemy ustalać następująco — cyfrę uzyskujemy, biorąc liczbę modulo podstawa. Następnie liczbę dzielimy przez podstawę, odrzucamy część ułamkową i cały proces powtarzamy tak długo, aż liczba będzie równa zeru.

Opisana procedura podaje cyfry od ostatniej (prawej). Spójrzmy na przykładzie, jak to działa. Załóżmy, że chcemy przekształcić liczbę z systemu dziesiętnego na binarny. W tabeli 6.1 pokazano kolejne etapy procesu przekształcania.

Tabela 6.1. Przekształcanie liczby z systemu dziesiętnego na dwójkowy

Liczba	Liczba modulo 2	Liczba / 2
10	0	5
5	1	2
2	0	1
1	1	0

Wynikiem konwersji jest zatem 1010 — wystarczy od dołu do góry odczytać cyfry z kolumny „Liczba modulo 2".

Aby napisać program wykonujący potrzebną konwersję, musimy wziąć pod uwagę kilka spraw. Po pierwsze, algorytm generujący cyfry od końca nie jest zbyt elegancki. Nie będziemy przecież kazali czytać użytkownikowi cyfry od prawej do lewej ani od dołu do góry. Musimy zastosować tutaj jakąś poprawkę. Zamiast wyświetlać każdą wygenerowaną cyfrę, będziemy cyfry te zapisywali w tablicy. Kiedy skończymy konwersję, zawartość tablicy możemy pokazać w pożądanej kolejności.

Po drugie, musimy zdawać sobie sprawę, że program ma konwertować liczby na systemy o podstawach do 16. Wobec tego uzyskiwane cyfry z zakresu od 10 do 15 musimy wyświetlać, korzystając z liter od *A* do *F*. Tutaj właśnie potrzebna będzie tablica znakowa.

Spójrzmy na program 6.7. Występuje w nim nowy modyfikator `const`, używany do zmiennych, których wartości nie zmieniają się w programie.

Program 6.7. **Konwersja dodatnich liczb całkowitych na inne systemy liczbowe**

```
// Program konwertuje dodatnie liczby całkowite na inne systemy liczbowe

#include <stdio.h>

int main (void)
{
    const char baseDigits[16] = {
           '0', '1', '2', '3', '4', '5', '6', '7',
           '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
    int      convertedNumber[64];
    long int numberToConvert;
    int      nextDigit, base, index = 0;

    // pobieramy liczbę i podstawę systemu

    printf ("Jaką liczbę konwertować? ");
    scanf ("%ld", &numberToConvert);
    printf ("Podstawa? ");
    scanf ("%i", &base);
```

```
    // konwersja na system o wskazanej podstawie

    do {
        convertedNumber[index] = numberToConvert % base;
        ++index;
        numberToConvert = numberToConvert / base;
    }
    while ( numberToConvert != 0 );

    // prezentacja wyniku – od końca

    printf ("Przekształcona liczba = ");

    for (—index;  index >= 0;  —index ) {
        nextDigit = convertedNumber[index];
        printf ("%c", baseDigits[nextDigit]);
    }

    printf ("\n");
    return 0;
}
```

Program 6.7. **Wyniki**

```
Jaką liczbę konwertować? 10
Podstawa? 2
Przekształcona liczba = 1010
```

Program 6.7. **Wyniki (ponowne uruchomienie)**

```
Jaką liczbę konwertować? 128362
Podstawa? 16
Przekształcona liczba = 1F56A
```

Kwalifikator const

Kompilator pozwala do deklaracji zmiennych, których wartości nie będą w programie zmieniane, dołączyć kwalifikator `const`. Można w ten sposób przekazać do kompilatora informację, że przez cały czas działania programu zmienna taka będzie miała *stałą* wartość. Jeśli po inicjalizacji próbujemy zmiennej zadeklarowanej jako `const` przypisać wartość, zwiększyć lub zmniejszyć jej wartość, kompilator może wygenerować komunikat o błędzie, choć specyfikacja języka C tego nie wymusza. Jednym z powodów stosowania atrybutu `const` jest umożliwienie kompilatorowi umieszczenia takiej zmiennej w pamięci przeznaczonej tylko do odczytu (normalnie instrukcje programu są umieszczane w pamięci tylko do odczytu).

Oto przykład użycia atrybutu `const`:

```
const double pi = 3.141592654;
```

Deklarujemy zmienną `pi` o stałej wartości. Kompilator wie, że wartość tej zmiennej nie będzie ulegała zmianie. Jeśli dalej w programie napiszemy coś takiego:

```
pi = pi / 2;
```

kompilator gcc wyświetli ostrzeżenie o treści:

```
foo.c:16: warning: assignment of read-only variable 'pi'
```[1]

Wróćmy do programu 6.7. Tablica znaków `baseDigits` zawiera 16 cyfr, jakie mogą być użyte w wyświetlanych liczbach. Tablicę tę deklarujemy ze słowem kluczowym `const`, gdyż jej zawartość po inicjalizacji nie będzie już zmieniana. Zauważmy, że deklaracja tablicy jako stałej jednocześnie zwiększa czytelność programu.

Tablica `convertedNumber` ma zawierać co najwyżej 64 cyfry, czyli pozwala zapisać największą możliwą wartość typu `long int` przy najmniejszej podstawie (czyli 2) na niemalże wszystkich komputerach. Zmienna `numberToConvert` jest typu `long int`, aby w razie potrzeby można było konwertować dość duże liczby. W końcu zmienna `base` (opisująca podstawę systemu liczbowego, który nas interesuje) oraz zmienna `index` (indeks tablicy `convertedNumber`) są typu `int`.

Po tym, jak użytkownik poda liczby, które mają być konwertowane, oraz podstawę (zauważmy, że funkcja `scanf`, jeśli ma wczytać liczbę typu `long int`, używa formantu `%ld`), program wchodzi do pętli konwertującej — `do`. Wybrano właśnie taką pętlę, aby w tablicy `convertedNumber` pojawiła się przynajmniej jedna cyfra — nawet jeśli konwertowane jest zero.

W pętli wyliczane jest `numberToConvert` modulo `base` — tak wyznaczana jest następna cyfra. Cyfra ta jest zapisywana w tablicy `convertedNumber`, indeks tej tablicy — `index` — jest inkrementowany o jeden. Po podzieleniu `numberToConverted` przez podstawę (`base`) sprawdzane jest spełnienie warunków pętli `do`. Jeśli `numberToConvert` jest równe zeru, pętla kończy się. W przeciwnym wypadku pętla jest ponownie wykonywana w celu określenia następnej cyfry wyniku.

Kiedy skończy się wykonywanie pętli, wartość zmiennej `index` zawiera liczbę cyfr przekonwertowanej liczby. Jako że zmienna ta jest inkrementowana o jeden w każdym przebiegu pętli `do`, w pętli `for` jej wartość jest początkowo zmniejszana o jeden. Zadaniem pętli `for` jest wyświetlenie wyniku. Pętla `for` przechodzi po tablicy `convertedNumber` *od końca*, aby cyfry były prawidłowo uporządkowane.

Każda cyfra z tablicy `convertedNumber` jest kolejno przypisywana zmiennej `nextDigit`. Aby prawidłowo wyświetlić liczby od *10* do *15* jako litery od *A* do *F*, przeszukiwana jest tablica `baseDigits`; jako indeks używana jest zmienna `nextDigit`. Dla cyfr od *0* do *9* odpowiednie miejsce w tablicy `baseDigits` zawiera znaki od `'0'` do `'9'` (pamiętajmy, że znaki te są czymś innym niż cyfry *0* – *9*). Komórki tablicy od 10. do 15. zawierają znaki od `'A'` do `'F'`. Jeśli zatem wartością `nextDigit` jest na przykład 10, pokazywany jest znak `baseDigits[10]`, czyli `'A'`. Jeśli z kolei `nextDigit` ma wartość 8, pokazywany jest znak `baseDigits[8]`, czyli `'8'`.

Kiedy wartość zmiennej `index` jest mniejsza od zera, działanie pętli `for` kończy się. Program wyświetla znak nowego wiersza i kończy swoje działanie.

---

[1] `foo.c:16: ostrzeżenie: przypisano wartość zmiennej 'pi' o wartości stałej` — *przyp. tłum.*

Warto jeszcze dodać, że łatwo można uniknąć pośredniego kroku, czyli przypisywania wartości `convertedNumber[index]` zmiennej `nextDigit`; wystarczy w wywołaniu funkcji `printf` bezpośrednio użyć pierwszego wyrażenia jako indeksu tablicy `baseDigits`, czyli w procedurze `printf` należy podać wyrażenie:

```
baseDigits[ convertedNumber[index] ]
```

Oczywiście takie wyrażenie jest nieco trudniejsze do zrozumienia niż dwa osobne wyrażenia w naszym programie.

Trzeba zaznaczyć, że omawiany program jest niedoskonały. Nie kontrolujemy, czy zmienna `base` ma wartość od *2* do *16*. Jeśli użytkownik jako podstawę poda `0`, dzielenie w pętli `do` będzie dzieleniem przez zero, czego trzeba bezwzględnie unikać. Dodatkowo program wpadnie w pętlę nieskończoną, jeśli użytkownik poda jako bazę `1`, gdyż `numberToConvert` nigdy nie osiągnie wartości zero. Jeśli użytkownik poda podstawę większą od 16, mogą zostać przekroczone granice tablicy `baseDigits`. Jest to kolejny, typowy błąd w programach C, których język nie sprawdza, bo to nasz obowiązek.

W rozdziale 7. napiszemy ten program ponownie z uwzględnieniem powyższych uwag. Jednak teraz zajmijmy się ciekawym rozszerzeniem pojęcia tablicy.

Tablice wielowymiarowe

Tablice omawiane dotąd były tablicami liniowymi (wektorami), czyli wszystkie miały jeden wymiar. Język C pozwala na używanie tablic o dowolnej liczbie wymiarów. W tym podrozdziale zajmiemy się tablicami dwuwymiarowymi.

Jednym z najbardziej naturalnych zastosowań tablic dwuwymiarowych jest macierz. Weźmy pod uwagę macierz 4×5 z tabeli 6.2.

Tabela 6.2. Macierz 4×5

10	5	–3	17	82
9	0	0	8	–7
32	20	1	0	14
0	0	8	7	6

W matematyce do wskazywania elementów macierzy zwykle używa się podwójnych indeksów. Jeśli zatem powyższą macierz oznaczymy przez *M*, oznaczenie $M_{i,j}$ odnosić się będzie do elementu z *i*-tego wiersza i *j*-tej kolumny. Wartość *i* należy do zakresu od *1* do *4*, wartość *j* — do zakresu od *1* do *5*. Oznaczenie $M_{3,2}$ wskaże wartość `20` z 3. wiersza i 2. kolumny macierzy. Analogicznie $M_{4,5}$ dotyczy 4. wiersza i 5. kolumny, czyli wartości 6.

W języku C można analogicznie odwoływać się do elementów tablic dwuwymiarowych, ale wobec tego, że indeksy tablic zaczynają się od zera, pierwszą kolumną macierzy jest kolumna 0. Powyższa macierz będzie miała wiersze i kolumny pokazane w tabeli 6.3.

Tabela 6.3. Macierz 4×5 w języku C

Kolumna (j) Wiersz (i)	0	1	2	3	4
0	10	5	–3	17	82
1	9	0	0	8	–7
2	32	20	1	0	14
3	0	0	8	7	6

Gdy w matematyce piszemy $M_{i,j}$, w języku C to samo zapisujemy jako:

```
M[i][j]
```

Pamiętajmy, że pierwszy indeks to numer wiersza, drugi — numer kolumny. Zatem instrukcja:

```
sum = M[0][2] + M[2][4];
```

spowoduje dodanie wartości z wiersza 0, kolumny 2. (czyli –3) do wartości z wiersza 2., kolumny 4. (czyli 14), wynik zaś — 11 — zostanie wpisany do zmiennej `sum`.

Tablice dwuwymiarowe deklarujemy tak samo jak tablice jednowymiarowe, zatem:

```
int M[4][5];
```

powoduje zadeklarowanie tablicy `M` jako tablicy dwuwymiarowej mającej 4 wiersze i 5 kolumn, razem 20 elementów. Każdy element tablicy zawiera liczbę całkowitą.

Tablice dwuwymiarowe można inicjalizować analogicznie jak ich odpowiedniki jednowymiarowe. Gdy podajemy wartości inicjalizujące, robimy to wiersz po wierszu. Pary nawiasów klamrowych rozdzielają dane służące do inicjalizacji poszczególnych wierszy. Aby zatem zdefiniować i zainicjalizować tablicę `M` z elementami z tabeli 6.3, używamy instrukcji:

```
int M[4][5] = {
                 { 10,  5, –3, 17, 82 },
                 {  9,  0,  0,  8, –7 },
                 { 32, 20,  1,  0, 14 },
                 {  0,  0,  8,  7,  6 }
               };
```

Zwróćmy szczególną uwagę na składnię powyższej instrukcji. Po każdym nawiasie klamrowym kończącym wiersz występuje przecinek — wyjątkiem jest ostatni wiersz. Użycie wewnętrznych par nawiasów jest opcjonalne. Jeśli nie zostaną wpisane, inicjalizacja będzie się odbywać wiersz po wierszu. Wobec tego powyższą instrukcję można by zapisać następująco:

```
int M[4][5] = { 10, 5, –3, 17, 82, 9, 0, 0, 8, –7, 32,
                20, 1, 0, 14, 0, 0, 8, 7, 6 };
```

Tak jak w tablicach jednowymiarowych, nie trzeba inicjalizować całej tablicy. Instrukcja typu:

```
int M[4][5] = {
                 { 10,  5, -3 },
                 {  9,  0,  0 },
                 { 32, 20,  1 },
                 {  0,  0,  8 }
              };
```

zainicjalizuje pierwsze trzy elementy każdego wiersza macierzy. Pozostałe wartości będą równe zeru. Zauważmy, że tym razem nawiasy wewnętrzne *są wymagane*. Bez nich zainicjalizowane byłyby dwa pierwsze wiersze i dwa pierwsze elementy wiersza trzeciego (warto sprawdzić to samemu).

W inicjalizacji można także używać indeksów, tak jak w tablicach jednowymiarowych. Wobec tego deklaracja:

```
int matrix[4][3] = {  [0][0] = 1, [1][1] = 5, [2][2] = 9 };
```

powoduje zainicjalizowanie wskazanymi wartościami trzech elementów tablicy `matrix`. Pozostałe elementy domyślnie zostaną ustawione na zero.

Tablice o zmiennej wielkości[2]

W tej części rozdziału omówimy cechę języka C, która umożliwia użycie tablic bez konieczności określania z góry ich stałej wielkości.

W przykładach z tego rozdziału widzieliśmy deklarowanie tablic określonej wielkości. W języku C wielkość tablicy może być zmienna. Program 6.3 wylicza 15 pierwszych wyrazów ciągu Fibonacciego. A co zrobić, gdybyśmy chcieli wyliczyć 100, lub nawet 500, tych wyrazów? Albo co będzie, gdybyśmy chcieli, aby to użytkownik podał interesującą go liczbę tych wyrazów? Przeanalizujmy program 6.8, w którym pokazano jedno z możliwych rozwiązań opisanego problemu.

Program 6.8. **Generowanie liczb Fibonacciego przy użyciu tablic o zmiennej wielkości**

```
// Generowanie liczb Fibonacciego przy użyciu tablic o zmiennej wielkości

#include <stdio.h>

int main (void)
{
    int i, numFibs;

    printf ("Ile liczb Fibonacciego potrzebujesz (od 1 do 75)? ");
    scanf ("%i", &numFibs);
```

[2] W standardzie ANSI C11 obsługa tablic o zmiennej wielkości jest nieobowiązkowa. Aby dowiedzieć się, czy dany kompilator je obsługuje, należy zajrzeć do jego dokumentacji.

```
    if (numFibs < 1 || numFibs > 75) {
        printf ("Podano nieprawidłową ilość!\n");
        return 1;
    }

    unsigned long long int   Fibonacci[numFibs];

    Fibonacci[0] = 0;          // z definicji
    Fibonacci[1] = 1;          // jak wyżej

    for ( i = 2;  i < numFibs;  ++i )
        Fibonacci[i] = Fibonacci[i-2] + Fibonacci[i-1];

    for ( i = 0;  i < numFibs;  ++i )
        printf ("%llu  ", Fibonacci[i]);

    printf ("\n");

    return 0;
}
```

Program 6.8. **Wyniki**

```
Ile liczb Fibonacciego potrzebujesz (od 1 do 75)? 50
0  1  1  2  3  5  8  13  21  34  55  89  144  233  377  610  987  1597  2584
4181  6765  10946  17711  28657  46368  75025  121393  196418  317811  514229
832040  1346269  2178309  3524578  5702887  9227465  14930352  24157817
39088169  63245986  102334155  165580141  267914296  433494437  701408733
1134903170  1836311903  2971215073  4807526976  7778742049
```

Program 6.8. ma kilka miejsc, które warto omówić. Po pierwsze, deklarujemy zmienne `i` i `numFibs`. Druga zmienna jest używana do zapisu żądanej liczby wyrazów ciągu Fibonacciego. Zauważmy, że program sprawdza, czy podana liczba mieści się w żądanym zakresie; jest to dobra praktyka programistyczna. Jeśli wartość jest spoza zakresu (czyli jest mniejsza od 1 lub większa od 75), program pokazuje komunikat i zwraca wartość 1, kończąc swoje działanie. Wykonanie w pokazanym miejscu instrukcji `return` powoduje zakończenie działania programu, pozostałe instrukcje nie są w ogóle wykonywane. Jak wspomniano w rozdziale 2., zwrócenie przez program wartości niezerowej zgodnie z powszechnie przyjętą konwencją oznacza jakiś błąd; może to być sprawdzone przez inny program.

Kiedy użytkownik poda liczbę wyrazów, mamy instrukcję:

```
unsigned long long int   Fibonacci[numFibs];
```

Tablica `Fibonacci` ma `numFibs` elementów. Jest to tablica o zmiennej wielkości, gdyż jej rozmiar jest ustalany przez zmienne, a nie stałe wyrażenie. Jak wspomniano wcześniej, każda zmienna może być deklarowana w dowolnym miejscu programu, byle tylko przed pierwszym użyciem. Choć więc nasza deklaracja pojawia się gdzie indziej niż inne deklaracje, to jest jak najbardziej poprawna. Zwykle jednak za dobry styl programowania uważa się grupowanie deklaracji zmiennych w jednym miejscu, gdyż czytającemu kod ułatwia to potem sprawdzanie typów zmiennych.

Ciąg Fibonacciego bardzo szybko rośnie, więc jego elementy zadeklarowaliśmy jako typ `long long int`. W ramach ćwiczenia sprawdźmy, jaką największą liczbę tego typu możemy zapisać w swoim komputerze w zmiennej typu `long long int`.

Reszta programu jest już oczywista — wyliczane są żądane wyrazy ciągu Fibonacciego, następnie są one pokazywane użytkownikowi. Na tym działanie programu się kończy.

Do alokacji pamięci dla tablic podczas wykonywania programu często stosuje się technikę zwaną *dynamiczną alokacją pamięci*. Oznacza to użycie funkcji `malloc` i `calloc` z biblioteki standardowej języka C. Temat ten omawiamy szczegółowo w rozdziale 16.

Jak widać, tablice to naprawdę użyteczne narzędzie programistyczne, dostępne w prawie wszystkich językach programowania. Przykładowy program wykorzystujący wielowymiarowe tablice pokażemy w rozdziale 7., w którym zaczynamy omawianie jednego z najważniejszych pojęć języka C — *funkcji*. Zanim przejdziemy dalej, spróbujmy jednak zrobić ćwiczenia.

Ćwiczenia

1. Przepisz i uruchom osiem programów pokazanych w tym rozdziale. Uzyskane wyniki porównaj z wynikami pokazanymi w tekście. Poeksperymentuj z tymi programami, podając wartości inne, niż pokazano w książce.
2. Zmodyfikuj program 6.1 tak, aby elementy tablicy `values` były początkowo ustawiane na zero. Inicjalizacji dokonaj w pętli `for`.
3. Program 6.2 pozwala wprowadzić tylko 20 odpowiedzi. Zmodyfikuj program tak, aby można było używać dowolnie wielu odpowiedzi. Niech użytkownik nie musi zliczać w ogóle ankiet; program powinien traktować wartość `999` jako koniec danych. (Podpowiedź! Do wyjścia z pętli można użyć instrukcji `break`).
4. Napisz program wyliczający średnią 10 liczb zmiennoprzecinkowych z tablicy.
5. Jaki wynik da poniższy program?

```
#include <stdio.h>

int main (void)
{
    int number[10] = { 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    int i, j;

    for ( j = 0;  j < 10;  ++j )
        for ( i = 0;  i < j;  ++i )
            numbers[j] += numbers[i];

    for ( j = 0;  j < 10;  ++j )
        printf ("%i ", numbers[j]);

    printf ("\n");

    return 0;
```

6. Do wygenerowania ciągu Fibonacciego nie trzeba wykorzystywać tablic. Wystarczy użyć trzech zmiennych: dwóch na poprzednie wyrazy ciągu, trzeciej na wyraz bieżący. Zmodyfikuj program 6.3 tak, aby nie używać w nim tablicy. Skoro nie ma już tablicy, to wyniki trzeba wyświetlać na bieżąco.
7. Liczby pierwsze możemy generować, korzystając z algorytmu nazywanego *sitem Eratostenesa*. Algorytm ten pokazujemy poniżej. Napisz program implementujący ten algorytm. Niech program znajduje liczby pierwsze do $n = 150$. Jak się ma ten algorytm do innych metod wyznaczania liczb pierwszych, omawianych w tym rozdziale?

   Algorytm „Sito Eratostenesa"

   Wyświetla wszystkie liczby pierwsze od 1 do n

   Krok 1: Definiujemy tablicę liczb całkowitych *P*. Wszystkie elementy P_i ustawiamy na 0, 2 <= *i* <= *n*.

   Krok 2: Ustawiamy *i* na 2.

   Krok 3: Jeśli $i > n$, algorytm się kończy.

   Krok 4: Jeśli P_i jest równe 0, *i* jest liczbą pierwszą.

   Krok 5: Dla wszystkich dodatnich liczb całkowitych *j* takich, że $i \times j \leq n$, ustawiamy $P_{i \times j}$ na 1.

   Krok 6: Do *i* dodajemy 1 i przechodzimy do kroku 3.
8. Sprawdź, czy używany przez Ciebie kompilator obsługuje tablice o zmiennej wielkości. Jeśli tak, przetestuj je przy użyciu niewielkiego własnoręcznie napisanego programu.

7

Funkcje

W języku C wszystkie dobrze napisane programy mają wspólny fundament — *funkcje*. Używaliśmy ich w każdym z programów napisanych dotąd. Przykładami funkcji są choćby procedury `printf` i `scanf`. W każdym programie występuje funkcja o nazwie `main`. Można by zatem zapytać, po co to całe zamieszanie? Otóż funkcje stanowią mechanizm, który ułatwia pisanie, czytanie, zrozumienie, poprawianie, modyfikowanie i serwisowanie programów. Wydaje się, że coś, co daje nam takie atuty, godne jest choćby niewielkiej owacji. Dlatego ten rozdział jest pełen bardzo ważnych informacji, a konkretnie zawiera następujące tematy:

- podstawowe wiadomości o funkcjach;
- zmienne lokalne, globalne, automatyczne i statyczne;
- posługiwanie się tablicami jedno- i wielowymiarowymi w funkcjach;
- zwracanie danych przez funkcje;
- wykonywanie programu od góry do dołu za pomocą funkcji;
- wywoływanie funkcji z innych funkcji oraz funkcje rekurencyjne.

Definiowanie funkcji

Najpierw musimy zrozumieć, czym jest funkcja, a dopiero potem będziemy zastanawiać się, jak jej najlepiej używać. Wróćmy do naszego pierwszego programu (program 2.1), wyświetlającego napis: „Programowanie to niezła zabawa”.

```
#include <stdio.h>

int main (void)
{
    printf ("Programowanie to niezła zabawa.\n");

    return 0;
}
```

Oto funkcja printMessage, która wykonuje to samo co nasz program:

```
void printMessage (void)
{
    printf ("Programowanie to niezła zabawa.\n");
}
```

Różnice między funkcją printMessage a funkcją main z programu 2.1 występują tylko w pierwszym i ostatnim wierszu. Pierwszy wiersz definicji funkcji informuje kompilator kolejno o następujących cechach funkcji. Oto one:

1. Kto może funkcję wywołać (dokładniej omówimy to w rozdziale 14.).
2. Jakiego typu wartość funkcja zwraca.
3. Nazwa funkcji.
4. Argumenty (inaczej parametry) funkcji.

Pierwszy wiersz definicji funkcji printMessage przekazuje do kompilatora informację, że funkcja nie zwraca żadnej wartości (pierwsze słowo kluczowe void), nazywa się printMessage i nie ma żadnych argumentów (drugie słowo kluczowe void). Więcej o słowie void dowiemy się wkrótce.

Jest oczywiste, że nadawanie funkcjom znaczących nazw jest równie ważne, jak nadawanie znaczących nazw zmiennym. Prawidłowy dobór nazw w dużej mierze decyduje o czytelności programu.

Pamiętamy z rozdziału 2., że main to nazwa specjalna, rozpoznawana przez system C; wykonywanie programu zawsze zaczyna się od tak nazwanej funkcji. *Zawsze* trzeba zdefiniować funkcję main. Aby powyższy fragment kodu był pełnoprawnym programem, dodajmy do niego na koniec funkcję main. Będzie to program 7.1.

Program 7.1. **Funkcje w języku C**

```
#include <stdio.h>

void printMessage (void)
{
    printf ("Programowanie to niezła zabawa.\n");
}

int main (void)
{
    printMessage ();

    return 0;
}
```

Program 7.1. **Wyniki**

```
Programowanie to niezła zabawa.
```

Program 7.1 zawiera *dwie* funkcje: `printMessage` oraz `main`. Wykonywanie programu zawsze zaczyna się od funkcji `main`. W tej funkcji pojawia się instrukcja:

```
printMessage ();
```

Zapis taki oznacza, że w tym miejscu wykonywana jest funkcja `printMessage`. Pusta para nawiasów służy do poinformowania kompilatora, że `printMessage` jest funkcją i nie są jej przekazywane żadne argumenty (przekazywane parametry muszą być zgodne z parametrami w definicji funkcji). Kiedy wywoływana jest funkcja, sterowanie programem jest przekazywane bezpośrednio do niej. W funkcji `printMessage` instrukcja `printf` jest wykonywana po to, by wyświetlić komunikat: „Programowanie to niezła zabawa”. Po wyświetleniu tej wiadomości działanie `printMessage` się kończy (zaznaczone jest to zamykającym nawiasem klamrowym), program *wraca* do funkcji `main`, do miejsca wywołania `printMessage`. W funkcji `printMessage` można użyć instrukcji `return` w postaci:

```
return;
```

Wobec tego, że `printMessage` nie zwraca żadnej wartości, `return` też nie ma żadnej wartości. Instrukcja ta jest opcjonalna, bo jeśli nie ma `return`, to po dojściu do końca funkcji program zachowa się tak, jakby było tam `return` bez parametrów. Tak więc użycie bądź nie instrukcji `return` nie wpływa na zachowanie się funkcji `printMessage`.

Jak wspomnieliśmy wcześniej, sam pomysł wywoływania funkcji nie jest nowy. Procedury `printf` i `scanf` to też funkcje. Jednak ich definicji nie pisaliśmy, gdyż są one częścią standardowej biblioteki języka C. Kiedy za pomocą funkcji `printf` wyświetlamy komunikat lub wyniki działania programu, sterowanie programem jest przekazywane do funkcji `printf`, która realizuje swoje zadania i zwraca sterowanie do miejsca wywołania. Program jest dalej wykonywany od miejsca znajdującego się tuż za wywołaniem `printf`.

Spróbujmy teraz odpowiedzieć, jakie wyniki da program 7.2.

Program 7.2. **Wywoływanie funkcji**

```
#include <stdio.h>

void printMessage (void)
{
    printf ("Programowanie to niezła zabawa.\n");
}

int main (void)
{
    printMessage ();
    printMessage ();

    return 0;
}
```

Program 7.2. **Wyniki**

```
Programowanie to niezła zabawa.
Programowanie to niezła zabawa.
```

Wykonywanie powyższego programu zaczyna się od funkcji `main`, która zawiera dwa wywołania funkcji `printMessage`. Po pierwszym wywołaniu sterowanie jest przekazywane do `printMessage`, funkcja ta pokazuje komunikat: „Programowanie to niezła zabawa.”, później następuje powrót do funkcji `main`. Po powrocie wykonywane jest drugie wywołanie `printMessage`, które działa tak samo jak pierwsze. Po powrocie z `printMessage` wykonywanie programu kończy się.

Teraz ostatni przykład z funkcją `printMessage`. Spróbujmy przewidzieć, jakie wyniki da program 7.3.

Program 7.3. **Wywoływanie funkcji jeszcze raz**

```
#include <stdio.h>

void printMessage (void)
{
    printf ("Programowanie to niezła zabawa.\n");
}

int main (void)
{
    int i;

    for ( i = 1;  i <= 5;  ++i )
        printMessage ();

    return 0;
}
```

Program 7.3. **Wyniki**

```
Programowanie to niezła zabawa.
Programowanie to niezła zabawa.
Programowanie to niezła zabawa.
Programowanie to niezła zabawa.
Programowanie to niezła zabawa.
```

Parametry i zmienne lokalne

Kiedy wywoływana jest funkcja `printf`, zawsze podajemy jej jedną lub więcej wartości; pierwsza wartość to łańcuch formatujący, pozostałe to dane, które program ma wyświetlić. Wartości te, nazywane *parametrami* lub *argumentami*, znakomicie zwiększają przydatność i elastyczność funkcji. W przeciwieństwie do naszej procedury `printMessage`, która przy każdym wywołaniu wyświetla ten sam komunikat, funkcja `printf` wyświetla to, czego od niej zażądamy.

Można definiować funkcje mające parametry. W rozdziale 4. tworzyliśmy programy obliczające liczby trójkątne. Tym razem zdefiniujemy funkcję generującą liczbę trójkątną, `calculateTriangularNumber`. Parametrem tej funkcji będzie numer liczby trójkątnej, którą chcemy wyliczyć. Funkcja wyliczy żądaną liczbę i pokaże wynik na ekranie. Program 7.4 zawiera odpowiednią funkcję oraz funkcję `main` służącą do jej testowania.

Program 7.4. **Wyliczanie n-tej liczby trójkątnej**

```
// Funkcja wyliczająca n-tą liczbę trójkątną

#include <stdio.h>

void calculateTriangularNumber (int n)
{
    int i, triangularNumber = 0;

    for ( i = 1;  i <= n;  ++i )
        triangularNumber += i;

    printf ("Trójkątna liczba numer %i to %i\n", n, triangularNumber);
}

int main (void)
{
    calculateTriangularNumber (10);
    calculateTriangularNumber (20);
    calculateTriangularNumber (50);

    return 0;
}
```

Program 7.4. **Wyniki**

```
Trójkątna liczba numer 10 to 55
Trójkątna liczba numer 20 to 210
Trójkątna liczba numer 50 to 1275
```

Deklaracja prototypu funkcji

Funkcja `calculateTriangularNumber` wymaga pewnych wyjaśnień. Pierwszy wiersz tej funkcji:

```
void calculateTriangularNumber (int n)
```

to *deklaracja prototypu funkcji*, która przekazuje do kompilatora informację, że `calculateTriangularNumber` jest funkcją niezwracającą żadnej wartości (słowo kluczowe `void`), mającą pojedynczy parametr `n` typu `int`. Nazwa argumentu (*parametru formalnego*) oraz sama nazwa funkcji są tworzone analogicznie jak nazwy zmiennych, co opisywaliśmy w rozdziale 3. Oczywiście nazwy te powinny być takie, aby jasne było, do czego dany parametr służy.

Po zdefiniowaniu nazw parametrów formalnych możemy odwoływać się do nich w całym ciele funkcji.

Początek definicji funkcji oznacza się otwierającym nawiasem klamrowym. Chcemy wyliczyć `n`-tą liczbę trójkątną, więc musimy mieć zmienną do wyliczenia pośredniego tej liczby. Potrzebna jest też zmienna działająca jako indeks pętli. Zmienne `triangularNumber` i `i` są właśnie tymi zmiennymi; obie są typu `int`. Zmienne te definiujemy i inicjalizujemy tak samo, jak definiowaliśmy i inicjalizowaliśmy zmienne w procedurze `main` we wcześniejszych programach.

Automatyczne zmienne lokalne

Zmienne definiowane wewnątrz funkcji nazywamy *automatycznymi zmiennymi lokalnymi*, gdyż są tworzone automatycznie przy każdym wywołaniu funkcji, a ich wartości są lokalne dla funkcji. Wartość zmiennej lokalnej może być użyta w funkcji, w której zmienną zdefiniowano. Nie można jej użyć w innej funkcji. Jeśli wewnątrz funkcji nadajemy zmiennej wartość początkową, wartość ta będzie przypisana zmiennej przy *każdym* wywołaniu funkcji.

Gdy w funkcji definiujemy zmienną lokalną, formalnie należałoby używać słowa kluczowego `auto`, na przykład:

```
auto int i, triangularNumber = 0;
```

Jednak kompilator C zakłada, że wszelkie zmienne zdefiniowane w funkcji są automatycznymi zmiennymi lokalnymi, słowa `auto` rzadko się używa; w tej książce też z niego zrezygnujemy.

Wróćmy do naszego przykładowego programu. Kiedy zostaną zdefiniowane zmienne lokalne, funkcja wylicza liczbę trójkątną i pokazuje na ekranie wynik swojego działania. Zamykający nawias klamrowy kończy definicję funkcji.

W procedurze `main`, w pierwszym wywołaniu `calculateTriangularNumber` jako parametr przekazujemy wartość 10. Sterowanie jest przekazywane bezpośrednio do naszej funkcji, 10 zaś *staje się wartością parametru formalnego*. Funkcja wylicza 10. liczbę trójkątną i pokazuje wynik.

Przy następnym wywołaniu `calculateTriangularNumber` jako parametr przekazywana jest wartość 20. Podobnie jak poprzednio, 20 staje się wartością `n` w ramach funkcji, wyliczana jest 20. liczba trójkątna i pokazywany jest wynik.

Teraz zajmijmy się przykładem, w którym funkcja będzie miała więcej niż jeden argument. Zmieńmy program wyliczający największy wspólny dzielnik (program 4.7), wydzielając zeń funkcję. Dwoma argumentami tej funkcji będą dwie liczby, dla których chcemy wyliczyć NWD — powstanie program 7.5.

Program 7.5. **Zmodyfikowany program znajdujący największy wspólny dzielnik**

```
/* Funkcja znajdująca największy wspólny dzielnik
   dwóch nieujemnych liczb całkowitych.                */

#include <stdio.h>

void gcd (int u, int v)
{
    int temp;

    printf ("NWD dla %i i %i to ", u, v);

    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
    }
```

```
    printf ("%i\n", u);
}

int main (void)
{
    gcd (150, 35);
    gcd (1026, 405);
    gcd (83, 240);

    return 0;
}
```

Program 7.5. **Wyniki**

```
NWD dla 150 i 35 to 5
NWD dla 1026 i 405 to 27
NWD dla 83 i 240 to 1
```

Funkcja `gcd` ma dwa parametry typu `int`. W funkcji do tych parametrów odwołujemy się przez nazwy parametrów formalnych — `u` i `v`. Po zadeklarowaniu zmiennej `temp` typu `int` program pokazuje wartości parametrów `u` i `v` wraz ze stosownym komunikatem. Dalej funkcja wylicza i pokazuje największy wspólny dzielnik obu liczb.

Można by się zastanawiać, po co w funkcji `gcd` są dwa wywołania funkcji `printf`. Wartości `u` i `v` trzeba wyświetlić *przed* wejściem do pętli `while`, gdyż ich wartości w tej pętli są zmieniane. Gdybyśmy czekali aż do zakończenia wykonywania pętli, pokazywane wartości `u` i `v` nie miałyby nic wspólnego z wartościami początkowymi, przekazanymi do funkcji. Innym rozwiązaniem mogłoby być przypisanie przed wejściem do pętli wartości `u` i `v` dwóm innym zmiennym, a potem pokazanie tych dwóch dodatkowych zmiennych wraz z wartością `u` (największym wspólnym dzielnikiem) w pojedynczym wywołaniu funkcji `printf`, już po pętli `while`.

Zwracanie wyników funkcji

Funkcje z programów 7.4 i 7.5 wykonują pewne proste obliczenia i pokazują wyniki tych obliczeń na ekranie. Jednak nie zawsze chcemy wyświetlać taki wynik. Język C udostępnia wygodny mechanizm *zwracania* przez funkcję wartości do miejsca wywołania. To nic nowego, gdyż we wcześniejszych programach korzystaliśmy z tego mechanizmu w funkcji `main`. Ogólna składnia odpowiedniego kodu jest prosta:

`return` *`wyrażenie`*`;`

Instrukcja ta nakazuje funkcji zwrócenie wartości *`wyrażenia`* do procedury wywołującej. Niektórzy programiści otaczają wyrażenie nawiasami; to kwestia stylu programowania, ale nie jest to konieczne.

Sama instrukcja `return` to za mało. Kiedy deklarujemy funkcję, musimy też zadeklarować *typ wartości zwracanej przez funkcję*. Typ ten podajemy *przed* nazwą funkcji. We wszystkich

dotychczasowych przykładach funkcji `main` podawaliśmy typ `int` jako typ wartości zwracanej. Jeśli z kolei deklaracja funkcji zaczyna się od:

```
float kmh_to_mph (float km_speed)
```

to mamy do czynienia z funkcją `kmh_to_mph` mającą jeden parametr `km_speed` typu `float`, zwracającą wartość zmiennoprzecinkową. Analogicznie:

```
int gcd (int u, int v)
```

definiuje funkcję `gcd` z argumentami `u` i `v` typu `int`, zwracającą wartość typu `int`. Możemy zmodyfikować program 7.5 tak, aby funkcja `gcd` nie wyświetlała największego wspólnego dzielnika, ale zwracała go do funkcji `main` — tak powstanie program 7.6.

Program 7.6. **Znajdowanie największego wspólnego dzielnika i zwracanie wyniku**

```
/* Funkcja znajdująca największy wspólny dzielnik
   dwóch nieujemnych liczb całkowitych i zwracająca wynik.  */

#include <stdio.h>

int gcd (int u, int v)
{
    int temp;

    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
    }

    return u;
}

int main (void)
{
    int result;

    result = gcd (150, 35);
    printf ("NWD dla 150 i 35 to %i\n", result);

    result = gcd (1026, 405);
    printf ("NWD dla 1026 i 405 to %i\n", result);

    printf ("NWD dla 83 i 240 to %i\n", gcd (83, 240));

    return 0;
}
```

Program 7.6. **Wyniki**

```
NWD dla 150 i 35 to 5
NWD dla 1026 i 405 to 27
NWD dla 83 i 240 to 1
```

Kiedy funkcja gcd wyliczy już największy wspólny dzielnik, wykonywana jest instrukcja:

```
return u;
```

Powoduje ona zwrócenie wartości u, czyli największego wspólnego dzielnika, do procedury wywołującej naszą funkcję.

Pewnie niektórzy już się zastanawiają, co można zrobić z wartością zwróconą przez wywoływaną procedurę. Jak widać w funkcji main, w pierwszych dwóch przypadkach wartość ta jest zapisywana w zmiennej result. Ściślej rzecz biorąc, instrukcja:

```
result = gcd (150, 35);
```

nakazuje wywołanie funkcji gcd z parametrami 150 i 35, a wynik przez tę funkcję zwrócony zapisuje w zmiennej result.

Wynik zwracany przez funkcję nie musi być przypisywany żadnej zmiennej, co widać w ostatnim wywołaniu funkcji main. Tutaj wartość zwrócona przez wywołanie:

```
gcd (83, 240)
```

jest przekazywana bezpośrednio do funkcji printf, która wartość tę wyświetla.

W opisany sposób funkcja języka C może zwrócić tylko jedną wartość. Język C, w przeciwieństwie do niektórych innych języków, nie odróżnia procedur (podprocedur) od funkcji. W C są tylko funkcje, które mogą ewentualnie zwracać wartość. Jeśli w deklaracji pominięty zostanie typ zwracanej wartości, kompilator zakłada, że zwracana jest wartość typu int. Niektórzy programiści korzystają z tego i pomijają deklarację typu int jako deklarację wartości zwracanej. Jednak jest to zła praktyka i nie należy jej stosować. Kiedy funkcja zwraca wartość, trzeba pamiętać o typie tej wartości w nagłówku funkcji — choćby w celu poprawienia czytelności programu. W ten sposób zawsze można zidentyfikować nagłówek funkcji nie tylko według nazwy funkcji oraz liczby i typów jej parametrów, lecz także według typu wartości zwracanej.

Jak wspominaliśmy wcześniej, deklaracja funkcji poprzedzona słowem kluczowym void przekazuje do kompilatora informację, że funkcja nie zwraca żadnej wartości. Próba użycia w takiej wartości zwracanego wyrażenia powoduje zgłoszenie przez kompilator błędu. Oto przykład. Ponieważ funkcja calculateTriangularNumber z programu 7.4 nie zwraca wartości, przed jej nazwą w definicji umieściliśmy słowo kluczowe void. Próba użycia tej funkcji tak, jakby zwracała ona jakąś wartość:

```
number = calculateTriangularNumber (20);
```

powoduje zgłoszenie błędu przez kompilator.

W pewnym sensie typ danych void oznacza brak typu danych. Tego typu funkcja zadeklarowana jako funkcja typu void nie ma wartości i nie może zostać użyta w wyrażeniu tak, jakby wartość zwracała.

W rozdziale 5. napisaliśmy program wyliczający i pokazujący wartość bezwzględną podanej liczby. Teraz napiszemy funkcję obliczającą wartość bezwzględną przekazanego jej parametru i zwracającą wynik. Zamiast używać liczb całkowitych, jak to robiliśmy w programie 5.1, tym razem użyjmy parametru i wartości zwracanej zmiennoprzecinkowej, typu float — tak oto powstanie program 7.7.

Program 7.7. **Wyliczanie wartości bezwzględnej**

```
// Funkcja wyliczająca wartość bezwzględną

#include <stdio.h>

float absoluteValue (float x)
{
    if ( x < 0 )
        x = -x;

    return x;
}

int main (void)
{
    float  f1 = -15.5, f2 = 20.0, f3 = -5.0;
    int    i1 = -716;
    float result;

    result = absoluteValue (f1);
    printf ("result = %.2f\n", result);
    printf ("f1 = %.2f\n", f1);

    result = absoluteValue (f2) + absoluteValue (f3);
    printf ("result = %.2f\n", result);

    result = absoluteValue ( (float) i1 );
    printf ("result = %.2f\n", result);

    result = absoluteValue (i1);
    printf ("result = %.2f\n", result);

    printf ("%.2f\n", absoluteValue (-6.0) / 4 );

    return 0;
}
```

Program 7.7. **Wyniki**

```
result = 15.50
f1 = -15.50
result = 25.00
result = 716.00
result = 716.00
1.50
```

Funkcja `absoluteValue` jest dość prosta. Parametr formalny — `x` — jest porównywany z zerem. Jeśli jest mniejszy od zera, wartość tego parametru jest mnożona przez minus jeden. Wynik zwracany jest do miejsca wywołania za pomocą odpowiedniej instrukcji `return`.

W funkcji `main` sprawdzającej działanie funkcji `absoluteValue` warto zwrócić uwagę na kilka ciekawych opcji. W pierwszym wywołaniu tej funkcji przekazywana jest wartość zmiennej `f1`, początkowo ustawionej na `–15.5`. W samej funkcji wartość ta przypisywana jest zmiennej `x`. Warunek instrukcji `if` jest spełniony, więc brane jest przeciwieństwo tej

wartości, czyli x otrzymuje wartość 15.5. W następnej instrukcji wartość x jest zwracana do procedury main, gdzie przypisywana jest jej zmienna result, następnie wartość ta jest wyświetlana.

Kiedy w funkcji absoluteValue zmianie ulega wartość x, nie wpływa to w żaden sposób na wartość zmiennej f1. Podczas przekazywania f1 do funkcji absoluteValue, *wartość ta jest automatycznie kopiowana* do parametru formalnego x. Wobec tego jakiekolwiek zmiany wartości x wpływają tylko na tę wartość x, a nie wpływają na f1. Wynika to z drugiego wywołania printf, gdzie pokazywana jest niezmieniona wartość f1. Trzeba to dobrze zrozumieć — wewnątrz funkcji nie można zmienić wartości zmiennej przekazanej funkcji jako argument, gdyż operuje się na kopiach tej zmiennej.

Następne dwa wywołania funkcji absoluteValue pokazują, jak można użyć wyniku zwracanego przez funkcję w wyrażeniu arytmetycznym. Wartość bezwzględna f2 jest dodawana do wartości bezwzględnej f3, suma jest przypisywana zmiennej result. Czwarte wywołanie funkcji absoluteValue prowadzi do stwierdzenia, że typ argumentu przekazanego funkcji powinien być zgodny z typem argumentu zadeklarowanego wewnątrz funkcji. Funkcja absoluteValue spodziewa się otrzymać jako parametr wartość zmiennoprzecinkową, zmienna całkowita i1 jest najpierw rzucana na typ float, dopiero wtedy wykonywane jest wywołanie. Jeśli pominęlibyśmy to rzutowanie, kompilator zrobiłby je sam, gdyż jest poinformowany, jakiego typu wartości spodziewa się funkcja absoluteValue (widać to w piątym wywołaniu absoluteValue). Jednak lepiej samemu robić rzutowanie, zamiast polegać na konwersjach wykonywanych przez system.

Ostatnie wywołanie funkcji absoluteValue pokazuje, że zasady wyliczania wyrażeń arytmetycznych dotyczą także wartości zwracanych przez funkcje. Ponieważ wartość zwracana przez naszą funkcję to float, kompilator traktuje dzielenie jako dzielenie liczby zmiennoprzecinkowej przez liczbę całkowitą. Jak pamiętamy, jeśli jeden z operandów takiego dzielenia jest typu float, dzielenie jest wykonywane w arytmetyce zmiennoprzecinkowej. Zgodnie z tą zasadą, podzielenie wartości bezwzględnej –6.0 przez 4 daje wynik 1,5.

Mamy już funkcję obliczającą wartość bezwzględną liczby i możemy używać jej w innych programach, tam gdzie będzie potrzebna. Przykładem takiego zastosowania jest właśnie następny program, czyli program 7.8.

Nic, tylko wywoływanie i wywoływanie...

Dzisiaj, kiedy kalkulatory są tak powszechnie dostępne jak zegarki, wyciąganie pierwiastka kwadratowego z jakiejś liczby nie stanowi żadnego problemu. Jednak przed laty studentów uczono ręcznych metod obliczania przybliżeń pierwiastków. Jedną z takich metod, która stosunkowo dobrze nadaje się do użycia w komputerze, jest *technika iteracyjna Newtona-Raphsona*. W programie 7.8 napiszemy funkcję obliczającą pierwiastek kwadratowy, która wykorzysta tę właśnie technikę do znajdowania przybliżonej wartości pierwiastka.

Metodę Newtona-Raphsona można opisać następująco — zaczynamy od próby odgadnięcia wartości pierwiastka. Im bliżej prawidłowej wartości trafimy, tym mniej obliczeń będziemy musieli wykonać później. Załóżmy jednak, że nie jesteśmy zbyt dobrzy w zgadywaniu i początkowo zawsze obstawiamy wartość 1.

Liczbę, której pierwiastek chcemy wyznaczyć, dzielimy przez pierwszą próbę odgadnięcia pierwiastka (dalej będziemy ją oznaczali jako zmienną `guess`) i dodajemy wynik do zmiennej `guess`. Ten wynik dzielimy przez 2. Następna próba odgadnięcia to wynik, który przed chwilą uzyskaliśmy. Liczba, której pierwiastek chcemy znaleźć, jest dzielona przez nową wartość, ta nowa wartość jest dodawana do wyniku i wszystko jest dzielone przez 2. Ten wynik staje się nową wartością odgadniętą (zmienna `guess`), po czym przechodzimy do kolejnej iteracji.

Nie chcemy naszego procesu iteracyjnego ciągnąć w nieskończoność, musimy wiedzieć, kiedy go przerwać. Kolejne odgadywane wartości są coraz bliższe prawdziwej wartości pierwiastka kwadratowego, możemy ustalić granicę, która wyznacza satysfakcjonującą nas dokładność. Różnica między odgadniętą wartością a samą liczbą może być porównana z ustaloną granicą, zwykle oznaczaną przez `epsilon` (ε). Jeśli różnica jest mniejsza od ε, uzyskaliśmy już pożądaną dokładność i cały proces możemy zakończyć.

Całą opisaną powyżej procedurę można zapisać w formie algorytmu.

Metoda Newtona-Raphsona służąca do wyliczania pierwiastka kwadratowego z liczby x

Krok 1: Ustalamy wartość zmiennej `guess` na 1.

Krok 2: Jeśli $|guess^2 - x| < \varepsilon$, przechodzimy do kroku 4.

Krok 3: Ustawiamy wartość `guess` na `(x/guess+guess)/2` i wracamy do kroku 2.

Krok 4: Wartość zmiennej `guess` jest przybliżeniem pierwiastka kwadratowego.

W kroku 2. koniecznie trzeba porównywać z ε *wartość bezwzględną* różnicy między $guess^2$ a x, gdyż wartość `guess` może zbliżać się do pierwiastka kwadratowego x z dowolnej strony.

Teraz, kiedy mamy gotowy algorytm znajdowania pierwiastka kwadratowego, dość łatwo możemy przygotować funkcję wyliczającą ten pierwiastek. W naszej funkcji jako wartość ε arbitralnie wybraliśmy `0.00001`. Gotowy program ma numer 7.8.

Program 7.8. **Wyliczanie pierwiastka kwadratowego**

```
// Funkcja wyliczająca pierwiastek kwadratowy z danej liczby

#include <stdio.h>

float absoluteValue (float x)
{
    if ( x < 0 )
        x = -x;
    return (x);
}
```

```
// Funkcja wyliczająca pierwiastek kwadratowy

float squareRoot (float x)
{
    const float epsilon = .00001;
    float       guess   = 1.0;

    while ( absoluteValue (guess * guess – x) >= epsilon )
        guess = ( x / guess + guess ) / 2.0;

    return guess;
}

int main (void)
{
    printf ("squareRoot (2.0) = %f\n", squareRoot (2.0));
    printf ("squareRoot (144.0) = %f\n", squareRoot (144.0));
    printf ("squareRoot (17.5) = %f\n", squareRoot (17.5));

    return 0;
}
```

Program 7.8. **Wyniki**

```
squareRoot (2.0) = 1.414216
squareRoot (144.0) = 12.000000
squareRoot (17.5) = 4.183300
```

W różnych systemach konkretne wartości mogą wykazywać nieznaczne różnice w ostatnich cyfrach wyniku.

Powyższy przykład wymaga gruntownej analizy. Najpierw definiowana jest funkcja `absoluteValue`. Jest to ta sama funkcja, której używaliśmy w programie 7.7.

Dalej mamy funkcję `squareRoot`. Ma ona jeden parametr `x` i zwraca wartość typu `float`. W treści tej funkcji korzystamy z dwóch zmiennych lokalnych — `epsilon` i `guess`. Wartości `epsilon` używamy do określenia, kiedy zakończyć proces iteracyjny; ustawiamy ją na `0.00001`. Zmiennej `epsilon` można nadać jeszcze mniejszą wartość — im mniejsza, tym dokładniejszy wynik, ale też dłużej trwa jego obliczanie. Początkowo wartość `guess` to `1.0`. Oba przypisania wartości wykonywane są przy każdym wywołaniu funkcji.

Po zadeklarowaniu zmiennych lokalnych przygotowujemy pętlę `while`, która jest wykonywana tak długo, jak długo bezwzględna różnica między `guess²` a `x` jest większa lub równa `epsilon`. Wyliczane jest wyrażenie:

```
guess * guess – x
```

i jego wynik jest przekazywany do funkcji `absoluteValue`. Wynik zwracany z `absoluteValue` jest porównywany ze zmienną `epsilon`. Jeśli wartość ta jest większa lub równa `epsilon`, żądana dokładność wyliczania pierwiastka nie została jeszcze osiągnięta. Wtedy wykonuje się następna iteracja pętli i liczona jest nowa wartość `guess`.

Kiedy wartość `guess` dostatecznie przybliży się do faktycznego pierwiastka, działanie pętli `while` kończy się. W tej chwili uzyskana wartość `guess` jest zwracana do miejsca

wywołania. W pętli `main` ta zwrócona wartość jest przekazywana do funkcji `printf` i wyświetlana.

Uważni czytelnicy zauważyli zapewne, że *obie* funkcje — `absoluteValue` i `squareRoot` — mają parametr formalny `x`. Kompilator niczego tutaj nie pomyli i odróżni od siebie obie te wartości.

Każda funkcja ma własny zestaw parametrów formalnych. Zatem parametr formalny `x` w funkcji `absoluteValue` nie ma nic wspólnego z parametrem formalnym `x` w funkcji `squareRoot`.

To samo dotyczy zmiennych lokalnych. W wielu z nich można deklarować tak samo nazywające się zmienne lokalne. Kompilator C nie myli poszczególnych zmiennych, gdyż każda z nich może być użyta tylko w funkcji, w której została zdefiniowana. To samo można powiedzieć inaczej — *zakres* zmiennej lokalnej to funkcja, w której taka zmienna została zdefiniowana (w rozdziale 10. poświęconym wskaźnikom okaże się, że język C zapewnia mechanizm pozwalający pośrednio sięgać do zmiennych lokalnych spoza funkcji).

Teraz już wiemy, że kiedy do funkcji `absoluteValue` przekazujemy wartość `guess²–x` i przypisujemy ją parametrowi formalnemu `x`, przypisanie to w żadnej mierze *nie wpływa* na wartość `x` w funkcji `squareRoot`.

Deklarowanie zwracanych typów, typy argumentów

Jak wspominaliśmy wcześniej, kompilator C zakłada, że domyślnie funkcja zwraca wartość typu `int`. Ściślej rzecz biorąc, kiedy wywoływana jest funkcja, kompilator zakłada, że funkcja ta zwraca wartość typu `int`, chyba że zachodzi jedna z poniższych sytuacji:

1. Funkcja została zdefiniowana w programie przed wywołaniem tej funkcji.
2. Wartość zwracana przez funkcję została *zadeklarowana* przed wywołaniem funkcji.

W programie 7.8 zdefiniowano funkcję `absoluteValue`, zanim funkcja ta zostanie wywołana w funkcji `squareRoot`. Kiedy więc dochodzi do wywołania, kompilator „wie”, że funkcja `absoluteValue` zwraca wartość typu `float`. Gdyby funkcja `absoluteValue` została zdefiniowana *po* funkcji `squareRoot`, po natknięciu się na wywołanie `absoluteValue` kompilator założyłby, że funkcja ta zwraca liczbę typu `int`. Większość kompilatorów C wychwytuje tego typu błąd i pokazuje odpowiedni komunikat diagnostyczny.

Aby można było zdefiniować funkcję `absoluteValue` po funkcji `squareRoot` (lub nawet w innym pliku — zobacz rozdział 14.), trzeba *zadeklarować* typ wyniku funkcji `absoluteValue` przed jej wywołaniem. Deklarację taką można zamieścić w funkcji `squareRoot` lub poza wszelkimi innymi funkcjami. W tym drugim wypadku zwykle deklarację taką umieszczamy na samym początku programu.

Deklaracji funkcji używamy nie tylko do zadeklarowania typu zwracanej wartości, lecz także do poinformowania kompilatora, ile funkcja ma argumentów i jakie są ich typy.

Aby zadeklarować `absoluteValue` jako funkcję zwracającą wartość `float` i mającą jeden parametr typu `float`, używamy deklaracji:

```
float absoluteValue (float);
```

Jak widać, w nawiasach trzeba podać typ parametru, a nie należy podawać nazwy. Można zresztą podać dowolną nazwę „tymczasową”:

```
float absoluteValue (float x);
```

Nazwa ta nie musi być taka sama jak w definicji funkcji; kompilator i tak ją ignoruje.

Aby w deklaracji funkcji uniknąć wszelkich możliwych błędów, możemy skopiować pierwszy wiersz definicji funkcji i wkleić go jako deklarację, nie zapominając przy tym o dodaniu średnika.

Jeśli funkcja nie ma parametrów, między nawiasami należy użyć słowa kluczowego `void`. Jeśli funkcja nie zwraca żadnej wartości, też można to zapisać, aby potem uniknąć nieporozumień przy jej wywoływaniu:

```
void calculateTriangularNumber (int n);
```

Jeśli funkcja ma zmienną liczbę parametrów (przykładem mogą być choćby funkcje `printf` czy `scanf`), trzeba o tym poinformować kompilator. Deklaracja:

```
int printf (char *format, ...);
```

informuje kompilator, że `printf` jako pierwszy swój parametr pobiera *wskaźnik* (więcej na ten temat w dalszej części rozdziału), a potem może być dowolnie wiele innych parametrów — wynika to z zapisu `...`. Funkcje `printf` i `scanf` zdefiniowano w specjalnym pliku nagłówkowym — *stdio.h*. Dlatego właśnie na początku każdego programu umieszczamy wiersz:

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

Bez tego kompilator założyłby, że `printf` i `scanf` mają stałą liczbę parametrów, wobec czego wygenerowany zostałby nieprawidłowy kod.

Przy wywołaniu funkcji kompilator automatycznie konwertuje przekazane parametry na odpowiednie typy, ale tylko wtedy, gdy wcześniej wystąpiła definicja lub deklaracja tej funkcji.

Oto pewne przypomnienia i sugestie dotyczące funkcji:

1. Pamiętajmy, że kompilator domyślnie zakłada, iż funkcja zwraca wartość typu `int`.
2. Definiując funkcję, która zwraca wartość `int`, i tak musimy to zapisać.
3. Definiując funkcję niezwracającą żadnej wartości, jako typ zwracany musimy podać `void`.
4. Kompilator konwertuje przekazane parametry na odpowiednie typy tylko wtedy, gdy funkcja zostanie wcześniej zdefiniowana lub zadeklarowana.
5. Aby nasz kod był możliwie mało podatny na błędy, należy deklarować wszystkie funkcje, nawet jeśli są definiowane jeszcze przed swoim wywołaniem (być może kiedyś zechcemy je przenieść w inne miejsce, lub nawet do innego pliku).

Sprawdzanie parametrów funkcji

Wyciąganie pierwiastka kwadratowego z liczby ujemnej powoduje, że tracimy kontakt z liczbami rzeczywistymi i lądujemy w królestwie liczb urojonych. Co się stanie, jeśli liczbę ujemną przekażemy funkcji `squareRoot`? W przypadku zastosowania metody Newtona-Raphsona wyliczenia będą rozbieżne, to znaczy wartość `guess` w kolejnych iteracjach nie będzie zbliżała się do prawidłowej wartości pierwiastka. Wobec tego warunek zakończenia pętli `while` *nigdy* nie zostanie spełniony, zatem program wpadnie w pętlę nieskończoną. Wykonywanie programu trzeba będzie awaryjnie przerwać, czyli wpisać odpowiednie polecenie lub wcisnąć specjalną kombinację klawiszy (jak choćby *Ctrl+C*).

Oczywiście w tym wypadku poprawienie programu jest niezbędne. Moglibyśmy wprawdzie cały problem scedować na procedurę wywołującą naszą funkcję i żądać, aby jako parametr nigdy nie była przekazywana wartość ujemna. W pierwszej chwili może to się wydawać rozsądnym rozwiązaniem, ale jednak ma poważne wady. Napiszemy program korzystający z funkcji `squareRoot`, ale zapomnimy sprawdzić, czy przekazywana wartość nie jest ujemna i program wpadnie w nieskończoną pętlę.

Znacznie rozsądniej byłoby w samej funkcji `squareRoot` sprawdzać wartość przekazanego parametru, zamiast narzucać takie wymagania sposobowi użycia funkcji. Dzięki temu funkcja byłaby „chroniona” w każdym programie, w jakim zostałaby użyta. Wobec tego warto sprawdzić wartość parametru `x` w samej funkcji i następnie (opcjonalnie) wyświetlić komunikat, jeśli parametr będzie ujemny. Wtedy funkcja może od razu przerwać swoje działanie, bez wykonania jakichkolwiek obliczeń. Aby funkcji wywołującej zasygnalizować, że coś nie zadziałało, należy zwrócić wartość, której nasza funkcja normalnie nigdy nie zwraca[1].

Oto zmodyfikowana funkcja `squareRoot`, sprawdzająca swój argument oraz zawierająca deklarację prototypu funkcji `absoluteValue`:

```
/* Funkcja wyliczająca pierwiastek kwadratowy podanej liczby.
   Jeśli przekazany zostanie parametr ujemny, pokazywany jest
   komunikat i zwracana jest wartość –1.0.                  */

float squareRoot (float x)
{
    const float epsilon = .00001;
    float guess   = 1.0;
    float absoluteValue (float x);

    if ( x < 0 )
    {
        printf ("Przekazano ujemny parametr do funkcji squareRoot.\n");
        return –1.0;
    }
```

[1] Funkcja obliczająca pierwiastek kwadratowy ze standardowej biblioteki C nazywa się `sqrt` i jeśli podany zostanie jej ujemny parametr, zwraca *błąd dziedziny* (*domain error*). Konkretna wartość tego błędu zależy od implementacji. W niektórych systemach można taką wartość wyświetlać — pokazuje się jako `nan`, czyli nie-liczba (od angielskiego „not a number”).

```
    while ( absoluteValue (guess * guess – x) >= epsilon )
        guess = ( x / guess + guess ) / 2.0;

    return guess;
}
```

Jeśli powyższa funkcja otrzyma parametr o ujemnej wartości, pokaże stosowny komunikat i zwróci wartość –1.0. Jeśli parametr nie będzie miał wartości ujemnej, wykonywane jest obliczenie pierwiastka kwadratowego tak samo jak poprzednio.

Jak widać w powyższym programie (zresztą to samo można zobaczyć w ostatnim przykładzie z rozdziału 6.), funkcja może mieć więcej niż jedną instrukcję `return`. Kiedy wykonywana jest instrukcja `return`, sterowanie natychmiast jest przekazywane do funkcji wywołującej i nie jest wykonywana już żadna instrukcja występująca za `return`. Powoduje to, że `return` jest idealną instrukcją dla funkcji niezwracających żadnych wartości. W tym wypadku, jak już wspomnieliśmy, instrukcja ta ma prostszą postać:

```
return;
```

gdyż nie trzeba niczego zwracać. Jeśli oczywiście funkcja ma zwrócić jakąś wartość, nie można użyć powyższej postaci.

Programowanie z góry na dół

Skoro mamy funkcje, które wywołują inne funkcje i tak dalej, uzyskujemy ostatecznie dobre, strukturalne programy. W procedurze `main` programu 7.8 funkcja `squareRoot` jest wywoływana wielokrotnie. Wszystkie szczegóły związane z konkretnymi obliczeniami pierwiastka kwadratowego są zamknięte w samej funkcji, a nie w `main`. Wobec tego funkcję można wywoływać jeszcze przed napisaniem tworzących ją instrukcji; trzeba tylko zadbać o zadeklarowanie parametrów funkcji i podanie wartości przez nią zwracanej.

Kiedy później przechodzimy do kodowania funkcji `squareRoot`, mamy tę samą sytuację — *programowanie z góry na dół*, czyli możemy wywołać funkcję `absoluteValue`, nie troszcząc się jeszcze o sposób jej działania. Wystarczy wiedzieć, że *potrafimy* taką funkcję napisać.

Opisana technika programowania ułatwia nie tylko pisanie programów, lecz także ich czytanie. Czytelnik programu 7.8 podczas analizy funkcji `main` łatwo stwierdzi, że program wylicza i wyświetla pierwiastki kwadratowe trzech liczb. Nie trzeba przedzierać się przez szczegóły wyliczania pierwiastka. Jeśli czytelnik jest bardziej zainteresowany szczegółami, może przeanalizować kod funkcji `squareRoot`. W tej funkcji tak samo można potraktować funkcję `absoluteValue` — nie trzeba wiedzieć, jak ona działa, a i tak można zrozumieć działanie `squareRoot`. Jeśli będzie to potrzebne, analizą treści `absoluteValue` możemy zająć się później.

Funkcje i tablice

Tak jak przekazujemy zwykłe zmienne i wartości, tak samo do funkcji możemy przekazywać jako parametry całe tablice. Aby zapisać w funkcji pojedynczy element tablicy (robiliśmy tak w rozdziale 6., kiedy używaliśmy funkcji `printf` do pokazywania elementów tablicy), element ten jak zwykle podajemy w wywołaniu funkcji. Aby zatem obliczyć pierwiastek kwadratowy z komórki `averages[i]` i przypisać wynik zmiennej `sq_root_result`, używamy instrukcji:

```
sq_root_result = squareRoot (averages[i]);
```

Dla samej funkcji `squareRoot` to, że przekazano nie zwykłą zmienną, ale komórkę tablicy, nie ma specjalnego znaczenia. Wartość komórki, tak jak w przypadku innych zmiennych, jest kopiowana do zmiennej odpowiadającej parametrowi formalnemu.

Przekazywanie do funkcji całej tablicy to już całkiem inna historia. Aby przekazać do funkcji tablicę, wystarczy przekazać jedynie jej nazwę, *bez żadnych indeksów*. Jeśli na przykład zadeklarowaliśmy `gradeScores` jako 100-elementową tablicę, wyrażenie:

```
minimum (gradeScores)
```

powoduje przekazanie do funkcji `minimum` stu elementów z tablicy `gradeScores`. Z drugiej strony oczywiście, funkcja `minimum` musi spodziewać się przekazania całej tablicy — wynika to z deklaracji parametrów formalnych. Zatem funkcja `minimum` może mieć postać:

```
int minimum (int values[100])
{
    ...
    return minValue;
}
```

Z tej deklaracji wynika, że funkcja `minimum` zwraca wartość typu `int`, a jako parametr przyjmuje 100-elementową tablicę danych `int`. Wszystkie odwołania do elementów tablicy przekazanej jako parametr formalny `values` są jednocześnie odwołaniami do elementów tablicy przekazywanej. Jeśli na przykład mamy powyższe wywołanie funkcji `minimum` i powyższą definicję tej funkcji, odwołanie do `values[4]` faktycznie stanowić będzie odwołanie do `gradeScores[4]`.

Pierwszym programem, który jako parametr przyjmie tablicę, będzie funkcja `minimum` znajdująca najmniejszą wartość z tablicy z dziesięcioma wartościami typu `int`. Funkcja ta wraz z funkcją `main` przygotowującą tablicę została pokazana w programie 7.9.

Program 7.9. **Znajdowanie wartości minimalnej w tablicy**

```
// Funkcja znajdująca wartość minimalną w tablicy

#include <stdio.h>

int minimum (int values[10])
{
    int minValue, i;

    minValue = values[0];
```

```
    for ( i = 1;  i < 10;  ++i )
        if ( value[i] < minValue )
            minValue = values[i];

    return minValue;
}

int main (void)
{
    int scores[10], i, minScore;
    int minimum (int values[10]);

    printf ("Podaj 10 wyników\n");

    for ( i = 0;  i < 10;  ++i )
        scanf ("%i", &scores[i]);

    minScore = minimum (scores);
    printf ("Najmniejszy wynik to %i\n", minScore);

    return 0;
}
```

Program 7.9. **Wyniki**

```
Podaj 10 wyników
69
97
65
87
69
86
78
67
92
90

Najmniejszy wynik to 65
```

W funkcji `main` najpierw rzuca się w oczy deklaracja prototypu funkcji `minimum`. Z deklaracji tej wynika, że `minimum` zwraca wartość typu `int`, a jako parametry pobiera 10 liczb typu `int`. Pamiętajmy, że deklaracja w tym wypadku nie jest konieczna, gdyż funkcja `minimum` jest zdefiniowana wcześniej, nim zostanie użyta w `main`. Jednak nie ryzykujmy — będziemy deklarować wszystkie funkcje, których użyjemy.

Po zdefiniowaniu tablicy `scores` prosimy użytkownika o podanie 10 wartości. Każde wywołanie `scanf` umieszcza jedną liczbę w `scores[i]`, przy czym `i` zmienia się od 0 do 9. Kiedy zostaną już podane wszystkie wartości, wywoływana jest funkcja `minimum` z tablicą `scores` jako parametrem.

Nazwa parametru formalnego `values` służy do wskazywania poszczególnych elementów z tej tablicy wewnątrz naszej funkcji. Parametr ten został zadeklarowany jako tablica dziesięciu liczb typu `int`. Zmienna lokalna `minValue` służy do przechowywania najmniejszej wartości w tablicy i początkowo jest ustawiana na `values[0]`. Pętla `for`

przechodzi po pozostałych elementach tablicy, porównując je kolejno z `minValue`. Jeśli któryś element `values[i]` jest mniejszy od `minValue`, mamy nowe minimum. Tę nową wartość przypisujemy zmiennej `minValue` i kontynuujemy przeglądanie dalszych elementów tablicy.

Kiedy pętla `for` skończy swoje działanie, do procedury wywołującej zwracana jest wartość `minValue`, która zostanie przypisana zmiennej `minScore`, a następnie wyświetlona.

Mając taką ogólną funkcję `minimum`, za jej pomocą możemy odnajdywać najmniejszy element z dowolnej, 10-elementowej tablicy liczb typu `int`. Gdybyśmy mieli pięć różnych tablic, z których każda zawierałaby po 10 liczb, wystarczyłoby wywołać pięć razy funkcję `minimum` i mielibyśmy już wartości najmniejsze każdej z tablic. Dodatkowo równie łatwo możemy zdefiniować funkcje realizujące inne zadania, na przykład znajdujące wartość największą, medianę, średnią i tak dalej.

Jeśli definiujemy małe, niezależne funkcje wykonujące dobrze określone zadania, możemy korzystać z nich do budowania zadań bardziej złożonych oraz używać ich w innych podobnych programach, na przykład moglibyśmy zdefiniować funkcję `statystyka`, która jako parametr otrzymywałaby tablicę i ewentualnie mogłaby wywoływać funkcję `srednia`, funkcję `odchylenieStandardowe` i tak dalej, gromadząc w ten sposób parametry statystyczne przekazanej tablicy. Tego typu działanie jest kluczem do tworzenia programów łatwych do napisania, zrozumienia, modyfikowania i serwisowania.

Oczywiście nasza ogólna funkcja `minimum` nie jest ogólna w tym sensie, że działa jedynie na tablicach 10-elementowych. Jednak problem ten stosunkowo łatwo rozwiązać. Można zwiększyć wszechstronność naszej funkcji, jeśli tylko liczbę elementów tablicy także będziemy przekazywać jako parametr. W deklaracji funkcji można wtedy pominąć liczbę elementów przy definiowaniu parametru formalnego opisującego tablicę. Kompilator C i tak pomija tę część deklaracji; kompilatorowi wystarczy wiedza, że przekazywana jest tablica, natomiast liczba jej elementów nie ma znaczenia.

Program 7.10 to poprawiona wersja programu 7.9, gdzie funkcja `minimum` znajduje najmniejszą wartość w tablicy liczb `int` dowolnej długości.

Program 7.10. **Poprawiona funkcja znajdująca w tablicy najmniejszą wartość**

```
// Funkcja znajdująca w tablicy najmniejszą wartość

#include <stdio.h>

int minimum (int values[], int numberOfElements)
{
    int minValue, i;

    minValue = values[0];

    for ( i = 1;  i < numberOfElements;  ++i )
        if ( values[i] < minValue )
            minValue = values[i];

    return minValue;
}
```

```
int main (void)
{
    int array1[5] = {157, –28, –37, 26, 10 };
    int array2[7] = { 12, 45, 1, 10, 5, 3, 22 };
    int minimum (int values[], int numberOfElements);

    printf (minimum array1: %i\n", minimum (array1, 5));
    printf (minimum array2: %i\n", minimum (array2, 7));

    return 0;
}
```

Program 7.10. **Wyniki**

```
minimum array1: –37
minimum array2: 1
```

Tym razem funkcja `minimum` ma dwa parametry — pierwszy to tablica, której najmniejsza wartość nas interesuje, a drugi to liczba elementów w tej tablicy. Otwierający i zamykający nawias kwadratowy występują bezpośrednio za nazwą `values` w nagłówku funkcji, przez co służą do poinformowania kompilatora, że `values` jest tablicą. Jak powiedzieliśmy wcześniej, kompilator nie musi znać rozmiaru tej tablicy.

Formalny parametr `numberOfElements` zastępuje teraz stałą 10 i pełni funkcję górnego ograniczenia pętli `for`. Wobec tego teraz pętla `for` przechodzi przez tablicę, od `values[1]` po ostatni element tej tablicy, `values[numberOfElements–1]`.

W funkcji `main` tworzone są dwie tablice — `array1` i `array2` — zawierające odpowiednio pięć i siedem elementów.

W pierwszym wywołaniu funkcji `printf` funkcja `minimum` jest wywoływana z parametrami `array1` i 5. Drugi parametr ustala liczbę elementów w tablicy `array1`. Funkcja `minimum` określa wartość najmniejszą w tablicy, a zwrócony wynik, czyli –37, jest następnie pokazywany na monitorze. Za drugim razem funkcja `minimum` jest wywoływana z parametrami `array2` oraz liczbą elementów tej tablicy. Funkcja ta do `printf` przekazuje odpowiedź 1.

Operatory przypisania

Przeanalizujmy program 7.11 i spróbujmy *przed* sprawdzeniem odpowiedzi przewidzieć, jakie da wyniki.

Program 7.11. **Zmiana elementów tablicy przez funkcję**

```
#include <stdio.h>

void multiplyBy2 (float array[], int n)
{
    int i;

    for ( i = 0;  i < n;  ++i )
        array[i] *= 2;
}
```

```
int main (void)
{
    float floatVals[4] = { 1.2f, –3.7f, 6.2f, 8.55f };
    int   i;
    void  multiplyBy2 (float array[], int n);

    multiplyBy2 (floatVals, 4);

    for ( i = 0;  i < 4;  ++i )
        printf ("%.2f   ", floatVals[i]);

    printf ("\n");

    return 0;
}
```

Program 7.11. **Wyniki**

```
2.40   –7.40   12.40   17.10
```

Niewątpliwie podczas analizy programu 7.11 większość czytelników zwróciła uwagę na zapis:

```
array[i] *= 2;
```

Efektem działania operatora „razy równa się" jest wymnożenie wyrażenia z lewej strony operatora przez wyrażenie ze strony prawej i *zapisanie wyniku z powrotem w zmiennej z lewej strony operatora*. Wobec tego powyższa instrukcja jest równoważna instrukcji:

```
array[i] = array[i] * 2;
```

Najważniejsze w powyższym programie jest jednak to, że funkcja `multiplyBy2` naprawdę *zmienia* wartości w tablicy `floatVals`. Nie stoi to bynajmniej w sprzeczności z dotychczasowymi twierdzeniami, że funkcja nie może zmieniać wartości swoich parametrów. Już tłumaczymy dlaczego.

Nasz przykładowy program pokazuje najważniejsze rozróżnienie, o którym musimy pamiętać, omawiając parametry w formie tablic — jeśli funkcja zmienia wartość w elemencie tablicy, zmienia oryginalną tablicę przekazaną jako parametr. Zmiana ta pozostaje obowiązująca nawet po zakończeniu wykonywania się funkcji i po powrocie sterowania do miejsca wywołania.

Powód, dla którego tablica zachowuje się inaczej niż zwykła zmienna lub element tablicy (ich wartości funkcja *nie może zmieniać*), jest godny objaśnienia. Jak wspominaliśmy wcześniej, kiedy wywoływana jest funkcja, przekazywane jej parametry są kopiowane do odpowiednich parametrów formalnych. Tutaj nic się nie zmienia. Jednak cała zawartość tablicy *nie* jest kopiowana do tablicy z parametrów formalnych. Zamiast tego funkcja otrzymuje informację, *gdzie* w pamięci należy szukać danej tablicy. Wszelkie zmiany wykonane na parametrze formalnym będącym tablicą automatycznie przenoszą się na oryginalną tablicę. Wobec tego, kiedy funkcja kończy swoje działanie, zmiany nadal obowiązują.

Pamiętajmy, że to, co teraz powiedzieliśmy, dotyczy zawsze całych tablic, a nigdy ich elementów, które bywają samodzielnie przekazywane do funkcji i wtedy ich wartości są kopiowane. Tych wartości nie można trwale zmienić. W rozdziale 10. omówimy to dokładniej.

Sortowanie tablic

Aby jeszcze lepiej ukazać, że funkcja może zmieniać wartości w tablicy, która została przekazana do niej w parametrze, utworzymy funkcję sortującą tablice liczb całkowitych. Proces sortowania był już dokładnie analizowany przez naukowców zajmujących się informatyką, gdyż często spotyka się go na co dzień. Powstało szereg doskonałych algorytmów sortujących, które pozwalają na minimalizowanie czasu działania i zużycia pamięci. Celem naszej książki nie jest prezentowanie najnowocześniejszych algorytmów, utworzymy funkcję `sort` korzystającą z dość prostego algorytmu w celu uporządkowania tablicy *w kolejności rosnącej*. Sortowanie tablicy w kolejności rosnącej oznacza ustawienie jej elementów od najmniejszego do największego. Na koniec sortowania wartość najmniejsza znajduje się w pierwszej komórce tabeli, największa w ostatniej, a pozostałe są ułożone kolejno między nimi.

Jeśli chcemy posortować rosnąco tablicę *n*-elementową, możemy porównywać kolejne elementy. Zaczynamy na przykład od elementu pierwszego i drugiego. Jeśli pierwszy jest większy od drugiego, zamieniamy je miejscami w tablicy.

Następnie porównujemy pierwszy element tablicy (o którym wiemy, że jest mniejszy od drugiego) z trzecim elementem. Jeśli pierwsza wartość jest większa od trzeciej, zamieniamy te wartości miejscami. W przeciwnym razie pozostawiamy je bez zmian. Teraz mamy najmniejszy z trzech pierwszych elementów w pierwszej komórce tablicy.

Jeśli opisany proces powtórzymy dla wszystkich pozostałych elementów z tablicy — porównamy pierwszy element z każdym następnym i podmienimy ich wartości, jeśli pierwszy jest większy od któregoś z dalszych — na koniec całego procesu najmniejsza wartość z całej tablicy znajdzie się w pierwszej komórce tej tablicy.

Jeśli to samo zrobimy z drugim elementem tablicy — czyli porównamy go z elementem trzecim, potem z czwartym i tak dalej oraz będziemy zamieniali miejscami elementy ułożone w złej kolejności — na koniec w drugiej komórce tablicy będziemy mieli drugą najmniejszą wartość.

Teraz już powinno być jasne, jak w opisany sposób można posortować całą tablicę. Proces kończy się, kiedy porównujemy przedostatni element tablicy z elementem ostatnim i w razie potrzeby zamieniamy je miejscami. W tej chwili cała tablica jest posortowana rosnąco.

Poniższy algorytm dokładniej opisuje proces sortowania. Zakładamy, że sortujemy tablicę *a* mającą *n* elementów.

Prosty algorytm sortowania przez zamianę miejsc

Krok 1: Ustawiamy *i* na 0.

Krok 2: Ustawiamy *j* na *i*+1.

Krok 3: Jeśli $a[i] > a[j]$, wartości te zamieniamy miejscami.

Krok 4: Ustawiamy j na $j+1$. Jeśli $j < n$, przechodzimy do kroku 3.

Krok 5: Ustawiamy i na $i+1$. Jeśli $i < n-1$, przechodzimy do kroku 2.

Krok 6: Tablica a jest posortowana rosnąco.

Program 7.12 stanowi implementację opisanego algorytmu; odpowiednia funkcja to `sort`, ma ona dwa parametry — sortowaną tablicę i liczbę elementów tej tablicy.

Program 7.12. **Sortowanie tablicy liczb całkowitych**

```
// Program sortujący rosnąco tablicę liczb całkowitych

#include <stdio.h>

void sort (int a[], int n)
{
    int i, j, temp;

    for ( i = 0;  i < n - 1;  ++i )
        for ( j = i + 1;  j < n;  ++j )
            if ( a[i] > a[j] ) {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
}

int main (void)
{
    int i;
    int array[16] = {34, -5, 6, 0, 12, 100, 56, 22,
                     44, -3, -9, 12, 17, 22, 6, 11 };
    void sort (int a[], int n);

    printf ("Tablica przed sortowaniem:\n");

    for ( i = 0;  i < 16;  ++i )
        printf ("%i ", array[i]);

    sort (array, 16);

    printf ("\n\nTablica po sortowaniu:\n");

    for ( i = 0;  i < 16;  ++i )
        printf ("%i ", array[i]);

    printf ("\n");

    return 0;
}
```

Program 7.12. **Wyniki**

```
Tablica przed sortowaniem:
34 –5 6 0 12 100 56 22 44 –3 –9 12 17 22 6 11

Tablica po sortowaniu:
–9 –5 –3 0 6 6 11 12 12 17 22 22 34 44 56 100
```

Funkcja `sort` realizuje opisany wcześniej algorytm, korzystając z zagnieżdżonych pętli `for`. Pętla pierwsza przechodzi przez elementy od pierwszego do przedostatniego (czyli `a[n–2]`). Dla każdego z tych elementów wykonywana jest druga pętla, która zaczyna swoje działanie od elementu znajdującego się za elementem wybranym przez pętlę zewnętrzną i przechodzi do ostatniego elementu tablicy.

Jeśli elementy nie są we właściwej kolejności (czyli `a[i]` jest większe od `a[j]`), są zamieniane miejscami. Zmienna `temp` służy jako miejsce chwilowego przechowania wartości, która bierze udział w takiej zamianie.

Kiedy kończy się działanie obu pętli `for`, tablica jest posortowana rosnąco, więc kończy się wykonywanie naszej funkcji.

W funkcji `main` zmienną `array` definiujemy jako tablicę 16 liczb całkowitych i tyloma liczbami ją inicjalizujemy. Następnie program wyświetla zawartość tej tablicy, po czym wywołuje funkcję `sort` i przekazuje jako parametry zmienną `array` i liczbę elementów tej tablicy — 16. Kiedy nasza funkcja skończy swoje działanie, program ponownie wyświetla zawartość tablicy `array`. Jak widać w wynikach, funkcja faktycznie sortuje wartości z tablicy.

Funkcja `sort` z programu 7.12 jest dość prosta. Ceną tej prostoty jest czas wykonywania się programu. Jeśli trzeba będzie sortować bardzo dużą tablicę, zawierającą tysiące liczb, nasza funkcja będzie działała długo. Jeśli któryś z czytelników stanie przed takim problemem, powinien skorzystać z bardziej zaawansowanych rozwiązań. Klasycznym źródłem takich algorytmów jest książka *The Art of Computer Programming. Volume 3: Sorting and Searching* autorstwa Donalda E. Knutha, wydana przez Addison-Wesley[2].

Tablice wielowymiarowe

Do funkcji można przekazać element tablicy wielowymiarowej tak samo jak każdą zwykłą zmienną czy element tablicy jednoelementowej. Zatem instrukcja:

```
squareRoot (matrix[i][j]);
```

wywołuje funkcję `squareRoot` i jako parametr przekazuje wartość `matrix[i][j]`.

Cała tablica wielowymiarowa może być przekazana do funkcji tak samo jak tablica jednowymiarowa — po prostu podaje się jej nazwę. Jeśli na przykład zmienna `wyniki_pomiarow` zostanie zadeklarowana jako dwuwymiarowa tablica liczb całkowitych, instrukcja:

---

[2] W standardowej bibliotece C jest funkcja `qsort`, której można używać do sortowania tablic z danymi dowolnego typu. Jednak aby jej użyć, trzeba zrozumieć działanie wskaźników na funkcje, czym zajmiemy się w rozdziale 10.

```
mnozenieSkalarne (wyniki_pomiarow, stala);
```

pozwala wywołać funkcję mnożącą każdy element przekazanej tablicy przez liczbę stałą. Oznacza to oczywiście, że funkcja może zmienić wartości znajdujące się w tablicy `wyniki_pomiarow`. Tutaj także obowiązują wszystkie zasady dotyczące tablic jednowymiarowych — przypisanie w funkcji elementowi tablicy wartości będącej parametrem formalnym powoduje trwałą zmianę wartości w tablicy przekazanej do tej funkcji.

Deklarując tablice jednowymiarowe jako parametry formalne, nie musimy podawać ich rozmiaru. Wystarczy po prostu użyć pary nawiasów kwadratowych, aby zaznaczyć, że dany parametr jest tablicą. Dla tablic wielowymiarowych jest już inaczej. Przy tablicach dwuwymiarowych można pominąć liczbę wierszy, ale *trzeba* podać liczbę kolumn tablicy. Wobec tego deklaracje:

```
int array_values[100][50]
```

oraz

```
int array_values[][50]
```

są poprawne, jeśli zostaną użyte jako deklaracje parametru formalnego `array_values` mającego 100 wierszy i 50 kolumn. Jednak deklaracje:

```
int array_values[100][]
```

oraz

```
int array_values[][]
```

już nie są poprawne, gdyż nie podano w nich liczby kolumn.

W programie 7.13 definiujemy funkcję `scalarMultiply` mnożącą dwuwymiarową tablicę liczb całkowitych przez wartość skalarną — liczbę całkowitą. Załóżmy na potrzeby tego przykładu, że tablica ma wymiary 3×5. Funkcja `main` dwukrotnie wywołuje funkcję `scalarMultiply`. Po każdym wywołaniu do procedury `displayMatrix`, wyświetlającej zawartość tablicy, przekazywana jest tablica, którą chcemy pokazać. Zwróćmy szczególną uwagę na zagnieżdżone pętle `for` użyte w funkcjach `scalarMultiply` i `displayMatrix` do przechodzenia przez wszystkie elementy tablicy dwuwymiarowej.

Program 7.13. **Użycie tablic wielowymiarowych i funkcji**

```
#include <stdio.h>

int main (void)
{
    void  scalarMultiply (int matrix[3][5], int scalar);
    void  displayMatrix (int matrix[3][5]);
    int   sampleMatrix[3][5] =
          {
              {  7, 16, 55, 13, 12 },
              { 12, 10, 52,  0,  7 },
              { -2,  1,  2,  4,  9 }
          };

    printf ("Macierz oryginalna:\n");
```

```
    displayMatrix (sampleMatrix);

    scalarMultiply (sampleMatrix, 2);

    printf ("\nMacierz wymnożona przez 2:\n");
    displayMatrix (sampleMatrix);

    scalarMultiply (sampleMatrix, -1);

    printf ("\nNastępnie wymnożona przez -1:\n");
    displayMatrix (sampleMatrix);

    return 0;
}

// Funkcja mnożąca tablicę 3 x 5 przez skalar

void scalarMultiply (int matrix[3][5], int scalar)
{
    int row, column;

    for ( row = 0;  row < 3;  ++row )
       for ( column = 0;  column < 5;  ++column )
           matrix[row][column] *= scalar;
}

void displayMatrix (int matrix[3][5])
{
    int row, column;

    for ( row = 0;  row < 3;  ++row) {
        for ( column = 0;  column < 5;  ++column )
            printf ("%5i", matrix[row][column]);
        printf ("\n");
    }
}
```

Program 7.13. **Wyniki**

```
Macierz oryginalna:
     7  16  55  13  12
    12  10  52   0   7
    -2   1   2   4   9
Macierz wymnożona przez 2:
    14  32 110  26  24
    24  20 104   0  14
    -4   2   4   8  18

Następnie wymnożona przez -1:
    -14  -32 -110  -26  -24
    -24  -20 -104    0  -14
      4   -2   -4   -8  -18
```

Funkcja `main` zawiera definicję macierzy `sampleValues` i wywołanie funkcji `displayMatrix` wyświetlającej początkowe wartości tej macierzy. W funkcji `displayMatrix` mamy zagnieżdżone instrukcje `for`. Zewnętrzna pętla przechodzi po wierszach macierzy, więc

wartość zmiennej sterującej `row` zmienia się od 0 do 2. Dla każdej wartości `row` wykonywana jest wewnętrzna pętla `for`, która przechodzi po kolumnach, a wartość jej zmiennej sterującej — `column` — zmienia się od 0 do 4.

Instrukcja `printf` pokazuje wartość zawartą w podanym wierszu i kolumnie, korzysta przy tym ze znaków formatujących `%5i`, aby zapewnić, że wartości będą prawidłowo wyrównane. Kiedy wewnętrzna pętla `for` zakończy swoje działanie, czyli pokazany zostanie cały wiersz macierzy, wyświetlany jest znak nowego wiersza, aby następny wiersz macierzy pokazać w nowym wierszu na ekranie.

Pierwsze wywołanie funkcji `scalarMultiply` mnoży tablicę `sampleMatrix` przez 2. Wewnątrz tej funkcji realizowane są zagnieżdżone pętle `for`, które przechodzą po każdym elemencie tablicy. Element `matrix[row][column]` jest mnożony przez wartość zmiennej `scalar` za pomocą operatora `*=`. Kiedy funkcja kończy swoje działanie i przekazuje sterowanie do `main`, znowu wywoływana jest funkcja `displayMatrix` wyświetlająca aktualną zawartość tablicy `sampleMatrix`. Program pokazuje, że faktycznie wszystkie elementy tablicy zostały wymnożone przez 2.

Po raz drugi funkcja `scalarMultiply` jest wywoływana w celu wymnożenia nowych elementów tablicy `sampleMatrix` przez –1. Ostatnie wywołanie `displayMatrix` ponownie pokazuje zmienioną zawartość tablicy i na tym kończy się działanie programu.

Wielowymiarowe tablice o zmiennej długości a funkcje

Korzystając z istnienia w języku C tablic o zmiennej długości, możemy napisać funkcje tak, by mogły przyjmować jako parametry tablice wielowymiarowe o różnej wielkości. Program 7.13 można zmodyfikować tak, aby funkcje `scalarMultiply` i `displayMatrix` przyjmowały jako parametry tablice mające dowolną liczbę wierszy i kolumn. Tak zmodyfikowany program to program 7.14.

Program 7.14. **Wielowymiarowe tablice o zmiennej wielkości**

```
#include <stdio.h>

int main (void)
{
    void  scalarMultiply (int nRows, int nCols,
                          int matrix[nRows][nCols], int scalar);
    void  displayMatrix (int nRows, int nCols, int matrix[nRows][nCols]);
    int   sampleMatrix[3][5] =
          {
              {  7, 16, 55, 13, 12 },
              { 12, 10, 52,  0,  7 },
              { -2,  1,  2,  4,  9 }
          };

    printf ("Macierz oryginalna:\n");
    displayMatrix (3, 5, sampleMatrix);

    scalarMultiply (3, 5, sampleMatrix, 2);
    printf ("\nMacierz wymnożona przez 2:\n");
    displayMatrix (3, 5, sampleMatrix);
```

```
    scalarMultiply (3, 5, sampleMatrix, -1);
    printf ("\nNastępnie wymnożona przez -1:\n");
    displayMatrix (3, 5, sampleMatrix);

    return 0;
}

// Funkcja mnożąca macierz przez skalar

void scalarMultiply (int nRows, int nCols,
                     int matrix[nRows][nCols], int scalar)
{
    int row, column;

    for ( row = 0;  row < nRows;  ++row )
       for ( column = 0;  column < nCols;  ++column )
           matrix[row][column] *= scalar;
}

void displayMatrix (int nRows, int nCols, int matrix[nRows][nCols])
{
    int row, column;

    for ( row = 0;  row < nRows;  ++row) {
        for ( column = 0;  column < nCols;  ++column )
            printf ("%5i", matrix[row][column]);
        printf ("\n");
    }
}
```

Program 7.14. **Wyniki**

```
Macierz oryginalna:
     7  16  55  13  12
    12  10  52   0   7
    -2   1   2   4   9

Macierz wymnożona przez 2:
    14  32 110  26  24
    24  20 104   0  14
    -4   2   4   8  18

Następnie wymnożona przez -1:
    -14  -32 -110  -26  -24
    -24  -20 -104    0  -14
      4   -2   -4   -8  -18
```

Deklaracja funkcji `scalarMultiply` wygląda teraz następująco:

```
void scalarMultiply (int nRows, int nCols, int matrix[nRows][nCols], int scalar)
```

Liczbę wierszy i kolumn macierzy — odpowiednio `nRows` i `nCols` — trzeba podawać w wywołaniu jako parametry *przed* samą macierzą, aby kompilator znał wartości tych parametrów jeszcze przed natknięciem się na deklarację macierzy w liście parametrów. Gdybyśmy próbowali zamiast tego zapisać:

```
void scalarMultiply (int matrix[nRows][nCols], int nRows, int nCols, int scalar)
```

otrzymalibyśmy komunikat błędu kompilacji, gdyż kompilator nie znałby wartości `nRows` i `nCols` w chwili interpretacji deklaracji `matrix`.

Jaki widać, program 7.13A daje takie same wyniki jak program 7.13. Mamy zatem dwie funkcje — `scalarMultiply` i `displayMatrix` — które działają na macierzach dowolnej wielkości. Jest to istotna zaleta użycia tablic o zmiennej wielkości.

Zmienne globalne

W tym rozdziale zobaczyliśmy już wiele zasad programowania w C. Nadszedł czas, by powiązać je teraz w całość oraz poznać nowe zasady. Weźmy pod uwagę program 6.7, który przekształcał całkowite liczby dodatnie na liczby zapisane w innym systemie liczbowym, i przepiszmy go tak, aby zastosować w nim funkcje. W tym celu musimy podzielić program na dwie logiczne części. Jeśli spojrzymy na ten program, to zobaczymy, że w funkcji `main` można wyróżnić trzy części, wyróżnikami będą komentarze. Wynika z nich, że program wykonuje trzy rzeczy — pobiera liczbę i podstawę od użytkownika, przekształca liczbę na inny system liczbowy oraz wyświetla wyniki.

Możemy zdefiniować trzy funkcje wykonujące te trzy rzeczy. Pierwsza funkcja — `getNumberAndBase` — poprosi użytkownika o podanie konwertowanej liczby i podstawy systemu oraz wczyta te wartości. Wprowadzimy tu też pewne udoskonalenie w porównaniu z programem 6.7 — jeśli użytkownik poda bazę mniejszą od 2 lub większą od 16, program wyświetli stosowny komunikat i ustawi podstawę na 10. Dzięki temu program po prostu ponownie wyświetli tę samą liczbę, którą poda użytkownik (można by też poprosić użytkownika o ponowne podanie prawidłowej podstawy, jednak zostawimy to jako ćwiczenie dla czytelnika).

Druga funkcja to `convertNumber`. Pobiera ona wartość podaną przez użytkownika i przekształca na żądany system liczbowy, cyfry wyniku zapisuje w tablicy `convertedNumber`.

Trzecia, ostatnia funkcja — `displayConvertedNumber` — pobiera cyfry z tablicy `convertedNumber` i wyświetla je we właściwej kolejności. Przy każdej cyfrze wykonywane jest przeszukanie tablicy `baseDigits` tak, aby pokazać właściwy znak odpowiadający danej cyfrze.

Zdefiniowane przez nas trzy funkcje komunikują się, korzystając ze *zmiennych globalnych*. Jak wspominaliśmy już wcześniej, jedną z zasadniczych cech zmiennych lokalnych jest dostępność ich tylko w tej funkcji, w której są zdefiniowane. Jak można się spodziewać, ograniczenie to nie dotyczy zmiennych globalnych. Wartość zmiennej globalnej jest dostępna w *dowolnej* funkcji w całym programie.

Podstawowa różnica między deklaracją zmiennej globalnej a deklaracją zmiennej lokalnej jest taka, że pierwsza z nich jest deklarowana *poza* jakąkolwiek funkcją. To właśnie wynika z globalnego charakteru takiej zmiennej — nie należy ona do żadnej funkcji. *Dowolna* funkcja w programie może do takiej zmiennej sięgnąć i w razie potrzeby zmienić jej wartość.

W programie 7.15 zdefiniowano cztery zmienne globalne. Każda z nich jest używana przynajmniej w dwóch funkcjach programu. Ponieważ tablica `baseDigits` oraz zmienna `nextDigit` są używane wyłącznie przez funkcję `displayConvertedNumber`, *nie* są one globalne, lecz lokalne dla funkcji `displayConvertedNumber`.

Zmienne globalne definiuje się na początku programu. Nie należą one do żadnej konkretnej funkcji, więc są globalne, czyli można się do nich odwołać z dowolnej funkcji w programie.

Program 7.15. **Konwersja dodatnich liczb całkowitych do innego systemu liczbowego**

```
// Program konwertujący dodatnie liczby całkowite do innych systemów liczbowych

#include <stdio.h>

int      convertedNumber[64];
long int numberToConvert;
int      base;
int      digit = 0;

void getNumberAndBase (void)
{
    printf ("Konwertowana liczba? ");
    scanf ("%li", &numberToConvert);

    printf ("Podstawa? ");
    scanf ("%i", &base);

    if ( base < 2  ||  base > 16 ) {
        printf ("Nieprawidłowa podstawa – musi mieścić się między 2 a 16\n");
        base = 10;
    }
}

void convertNumber (void)
{
    do {

        convertedNumber[digit] = numberToConvert % base;
        ++digit;
        numberToConvert /= base;
    }
    while { numberToConvert != 0 );
}

void displayConvertedNumber (void)
{
    const char baseDigits[16] =
            { '0', '1', '2', '3', '4', '5', '6', '7',
              '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
    int   nextDigit;

    printf ("Przekonwertowana liczba = ");

    for ( —digit;  digit >= 0;  —digit ) {
        nextDigit = convertedNumber[digit];
```

```
        printf ("%c", baseDigits[nextDigit]);
    }

    printf ("\n");
}

int main (void)
{
    void getNumberAndBase (void), convertNumber (void),
         displayConvertedNumber (void);

    getNumberAndBase ();
    convertNumber ();
    displayConvertedNumber ();

    return 0;
}
```

Program 7.15. **Wyniki**

```
Konwertowana liczba? 100
Podstawa? 8
Przekonwertowana liczba = 144
```

Program 7.15. **Wyniki (ponowne uruchomienie)**

```
Konwertowana liczba? 1983
Podstawa? 0
Nieprawidłowa podstawa — musi mieścić się między 2 a 16
Przekonwertowana liczba = 1983
```

Zauważmy, jak bardzo staranny dobór nazw funkcji pozwala zrozumieć program 7.15. Jeśli przeczytamy i przetłumaczymy treść procedury `main`, otrzymamy opis działania całego programu — pobieramy liczbę i podstawę, konwertujemy liczbę i wyświetlamy przekonwertowaną liczbę. Znacząca poprawa czytelności tego programu w porównaniu z programem z rozdziału 6. jest wynikiem podzielenia programu na kilka funkcji, które realizują proste, jasno określone zadania. Zauważmy, że nawet nie potrzebujemy w funkcji `main` komentarzy opisujących działanie programu — nazwy funkcji mówią same za siebie.

Podstawowym zastosowaniem zmiennych globalnych jest sytuacja, kiedy wiele funkcji musi korzystać z tej samej zmiennej. Zamiast przekazywać wartość zmiennej do każdej funkcji z osobna, funkcja sama odwołuje się do tej zmiennej. Rozwiązanie takie ma jednak poważną wadę. Skoro funkcja jawnie odwołuje się do konkretnej zmiennej globalnej, ogólność tej funkcji jest ograniczona — przy każdym wywołaniu tej funkcji trzeba zagwarantować, że potrzebna zmienna globalna, o ustalonej nazwie, istnieje.

Na przykład funkcja `convertNumber` z programu 7.15 zadziała prawidłowo, jeśli liczba konwertowana jest w zmiennej `numberToConvert`, a podstawa — w zmiennej `base`. Dodatkowo zdefiniowane muszą być zmienna `digit` oraz tablica `convertedNumber`. Omawiana funkcja byłaby znacznie elastyczniejsza, gdyby otrzymywała potrzebne dane w parametrach.

Użycie zmiennych globalnych pozwala wprawdzie ograniczyć liczbę parametrów, które trzeba przekazać funkcji, ale ceną za to jest utrata ogólności funkcji i czasami zmniejszenie czytelności programu. To zmniejszenie czytelności bierze się stąd, że jeśli używamy w funkcji zmiennych globalnych, to sprawdzenie nagłówka funkcji nic o tym nie mówi. Poza tym wywołanie takiej funkcji nie informuje użytkownika, jakie parametry są funkcji potrzebne i jakie wartości funkcja zwraca.

Niektórzy programiści stosują konwencję polegającą na poprzedzaniu nazw zmiennych globalnych literą „g". Przykładowo deklaracje zmiennych z programu 7.15 mogłyby wyglądać następująco:

```
int      gConvertedNumber[64];
long int gNumberToConvert;
int      gBase;
int      gDigit = 0;
```

Powodem korzystania z takiej właśnie konwencji jest ułatwienie „wyłapania" zmiennych globalnych podczas czytania programu, na przykład instrukcja:

```
nextMove = gCurrentMove + 1;
```

od razu sugeruje, że `nextMove` jest zmienną lokalną, a `gCurrentMove` — zmienną globalną. W ten sposób czytelnik od razu zna zakres zmiennych użytych w tym wyrażeniu i wie, gdzie szukać potrzebnych deklaracji.

Jeszcze jedno o zmiennych globalnych. Mają one wartość domyślną — zero. Wobec tego w przypadku deklaracji globalnej:

```
int gData[100];
```

na początku programu wszystkie elementy tablicy `gData` w ilości 100 są ustawiane na zero.

Pamiętajmy zatem — jeśli zmienne globalne mają wartość początkową zero, zmienne lokalne nie mają domyślnej wartości początkowej i wobec tego trzeba je inicjalizować jawnie.

Zmienne automatyczne i statyczne

Kiedy w funkcji deklarujemy zmienną lokalną, jak w przypadku zmiennych `guess` i `epsilon` w funkcji `squareRoot`:

```
float squareRoot (float x)
{
    const float epsilon = .00001;
    float guess = 1.0;
        ...
}
```

deklarujemy ją *automatycznie*. Przypomnijmy, że zmienne takie można poprzedzić słowem kluczowym `auto`, ale jest ono opcjonalne. Zmienna automatyczna jest tworzona w chwili każdego wywołania funkcji `squareRoot`. Kiedy działanie tej funkcji się kończy, zmienne te „znikają". Cały proces odbywa się automatycznie, stąd nazwa.

Lokalne zmienne automatyczne mogą mieć wartości początkowe, tak jak powyższe zmienne `guess` i `epsilon`. Jako wartości początkowej zmiennej automatycznej można użyć dowolnego, poprawnego wyrażenia języka C. Wartość tego wyrażenia jest obliczana *każdorazowo* przy wywołaniu funkcji. Skoro zmienne automatyczne znikają po zakończeniu wykonywania funkcji, wartość takiej zmiennej też znika. Innymi słowy, wartość zmiennej automatycznej *na pewno* nie będzie istniała przy następnym wywołaniu tej samej funkcji.

Jeśli przed deklaracją zmiennej umieścimy słowo kluczowe `static`, mamy całkiem inną sytuację. Słowo `static` (dosłownie *statyczny*) w języku C nie odnosi się do ładunków elektrycznych, ale do braku ruchu. Takie właśnie są zmienne statyczne — nie pojawiają się i nie znikają, kiedy funkcja jest wywoływana i kiedy kończy swoje działanie. Wobec tego wartość zmiennej statycznej po zakończeniu działania funkcji jest taka sama także później, przy ponownym wywołaniu tej funkcji.

Zmienne statyczne są inaczej inicjalizowane. Statyczna zmienna lokalna jest inicjalizowana *tylko raz*, kiedy zaczyna się wykonywanie całego programu. Nie jest już inicjalizowana podczas wywoływania funkcji, w której ją zdefiniowano. Co więcej, wartość zmiennej statycznej *musi* być stałą lub wyrażeniem stałym. Zmienne statyczne, w przeciwieństwie do zmiennych automatycznych, mają wartość początkową — zero.

W funkcji `auto_static`, zdefiniowanej jak poniżej:

```
void auto_static (void)
{
    static int staticVar = 100;
        ...
}
```

zmienna `staticVar` jest inicjalizowana tylko raz na 100, kiedy zaczyna się wykonanie całego programu. Aby ustawić tę zmienną na wartość 100 przy każdym wywołaniu funkcji, trzeba zrobić jawne przypisanie:

```
void auto_static (void)
{
    static int staticVar = 100;

    staticVar = 100;
        ...
}
```

Oczywiście tego typu wielokrotna inicjalizacja zmiennej `staticVar` stoi w jawnej sprzeczności z celem stosowania zmiennych statycznych.

Program 7.16 powinien pomóc w zrozumieniu pojęcia zmiennych automatycznych i zmiennych statycznych.

Program 7.16. **Ilustracja użycia zmiennych statycznych i zmiennych automatycznych**

```
// Program pokazuje użycie zmiennych statycznych i automatycznych

#include <stdio.h>

void auto_static (void)
{
    int        autoVar = 1;
```

```
    static int staticVar = 1;

    printf("automatyczna = %i, statyczna = %i\n", autoVar, staticVar);

    ++autoVar;
    ++staticVar;
}

int main (void)
{
    int  i;
    void auto_static (void);

    for ( i = 0;  i < 5;  ++i )
        auto_static ();

    return 0;
}
```

Program 7.16. **Wyniki**

```
automatyczna = 1, statyczna = 1
automatyczna = 1, statyczna = 2
automatyczna = 1, statyczna = 3
automatyczna = 1, statyczna = 4
automatyczna = 1, statyczna = 5
```

W funkcji `auto_static` zadeklarowano dwie zmienne lokalne. Pierwsza z nich — `autoVar` — to zmienna automatyczna typu `int` o wartości początkowej 1. Druga zmienna — `staticVar` — jest zmienną statyczną, także o wartości początkowej 1. Funkcja wywołuje procedurę `printf`, aby pokazać wartości obu tych zmiennych. Następnie zmienne są zwiększane o 1, a wykonywanie funkcji kończy się.

Procedura `main` tworzy pętlę wywołującą funkcję `auto_static` pięciokrotnie. Wyniki działania programu 7.16 pokazują różnicę między dwoma klasami zmiennych. Wartość zmiennej automatycznej stale wynosi 1. Wynika to stąd, że zmienna ta jest inicjalizowana przy każdym wywołaniu funkcji, która ją zawiera. Z kolei wartość zmiennej statycznej stale rośnie, od 1 do 5. Dzieje się tak dlatego, że zmienna ta uzyskuje wartość 1 tylko raz, kiedy zaczyna się wykonywanie programu. Potem wartość tej zmiennej jest zachowywana między kolejnymi wywołaniami funkcji.

Decyzja o tym, czy zmienna ma być statyczna czy automatyczna, zależy od przeznaczenia tej zmiennej. Jeśli powinna ona zachowywać swoją wartość między kolejnymi wywołaniami zawierającej ją funkcji (niech na przykład funkcja zlicza, ile razy została wywołana), używamy zmiennej statycznej. Jeśli funkcja używa zmiennej, której wartość jest ustawiona raz i potem nigdy się nie zmienia, też można użyć zmiennej statycznej, gdyż w ten sposób unikamy obciążenia związanego z wielokrotnym inicjalizowaniem tej zmiennej. Kwestie wydajności stają się szczególnie ważne, kiedy mamy do czynienia z tablicami.

Jednak jeśli wartość zmiennej lokalnej musi być inicjalizowana przy każdym wywołaniu funkcji, użycie zmiennej automatycznej jest naturalne.

Funkcje rekurencyjne

Język C obsługuje funkcje *rekurencyjne*. Funkcje takie pozwalają skutecznie rozwiązywać problemy przez ich upraszczanie. Zwykle stosowane są tam, gdzie rozwiązanie problemu można opisać przez rozwiązywanie podproblemów. Przykładem takiego zastosowania może być analiza wyrażeń z zagnieżdżonymi nawiasami. Innym typowym zastosowaniem jest przeszukiwanie i sortowanie struktur danych, takich jak *drzewa* i *listy*.

Funkcje rekurencyjne zwykle omawia się na przykładzie liczenia silni. Przypomnijmy, że silnia pewnej dodatniej liczby całkowitej *n*, zapisywana jako *n*!, to iloczyn wszystkich liczb od 1 do *n*. Liczba 0 jest wyjątkiem, gdyż jej silnia z definicji równa jest 1. Aby zatem obliczyć 5!, liczymy, co następuje:

```
5! = 5·4·3·2·1
   = 120
```

natomiast

```
6! = 6·5·4·3·2·1
   = 720
```

Porównując sposób liczenia 5! i 6!, stwierdzamy, że 6! jest sześciokrotnie większe od 5!, czyli *6! = 6·5!*. Ogólnie rzecz biorąc, silnia dowolnej liczby dodatniej *n*! większej od zera jest równa *n* razy silnia *n*–1:

$$n! = n \cdot (n-1)!$$

Jeśli wyrazimy *n*! za pomocą (*n*–1)!, mówimy o definicji *rekurencyjnej*. Można utworzyć funkcję liczącą silnię zgodnie z powyższą definicją rekurencyjną, tak jak w programie 7.17.

Program 7.17. **Rekurencyjne wyliczanie silni**

```
#include <stdio.h>

int main (void)
{
    unsigned int  j;
    unsigned long int factorial (unsigned int n);

    for ( j = 0;  j < 11;  ++j )
        printf ("%2u! = %lu\n", j, factorial (j));

    return 0;
}

// Funkcja rekurencyjna wyliczająca silnię dodatnich liczb całkowitych

unsigned long int factorial (unsigned int n)
{
    unsigned long int result;

    if ( n == 0 )
        result = 1;
    else
```

```
        result = n * factorial (n - 1);

    return result;
}
```

Program 7.17. **Wyniki**

```
0!  = 1
1!  = 1
2!  = 2
3!  = 6
4!  = 24
5!  = 120
6!  = 720
7!  = 5040
8!  = 40320
9!  = 362880
10! = 3628800
```

Funkcja `factorial` jest rekurencyjna dlatego, że wywołuje samą siebie. Kiedy funkcja ta zostanie wywołana w celu obliczenia silni liczby 3, wartość parametru formalnego `n` zostanie ustawiona na 3. Nie jest to zero, więc zostanie wykonana instrukcja:

```
result = n * factorial (n - 1);
```

co spowoduje wyliczenie:

```
result = 3 * factorial (2));
```

Wyrażenie to mówi, że wywołana ma zostać funkcja `factorial`, aby wyliczyć silnię 2. Wobec tego mamy mnożenie przez 3 wartości zwróconej przez `factorial(2)`.

Wprawdzie wywołujemy ponownie tę samą funkcję, ale trzeba zdawać sobie sprawę, że jest to jakby wywołanie funkcji całkiem osobnej. Zawsze, kiedy w języku C wywołuje się funkcję — rekurencyjną lub nie — funkcja ta otrzymuje własny zestaw zmiennych lokalnych i parametrów formalnych. Wobec tego zmienna lokalna `result` i parametr formalny `n` istniejący w chwili wywołania `factorial` w celu wyliczenia 3! są różne od zmiennej `result` i parametru `n` używanego przy wywołaniu wykonywanym w celu obliczenia 2!.

Kiedy wartość `n` jest równa 2, funkcja `factorial` wykonuje instrukcję:

```
result = n * factorial (n - 1);
```

interpretowaną jako:

```
result = 2 * factorial (1);
```

Znowu pozostaje do obliczenia mnożenie 2 przez silnię 1, natomiast wywołuje się funkcję `factorial` w celu policzenia silni 1.

Kiedy `n` jest równe 1, funkcja `factorial` ponownie wykonuje instrukcję:

```
result = n * factorial (n - 1);
```

interpretowaną jako:

```
result = 1 * factorial (0);
```

Kiedy wywołamy funkcję `factorial`, aby obliczyć silnię zera, funkcja ta ustawi `result` na 1 i zakończy swoje działanie, pozwalając na wykonanie wszystkich wcześniej zawieszonych mnożeń. Wobec tego do funkcji wywołującej (akurat także `factorial`) zwracany jest wynik wywołania `factorial(1)`, czyli 1. Wartość ta jest mnożona przez 2 i zapisywana w zmiennej `result` jako wartość `factorial(2)`. W końcu zwrócona wartość 2 jest mnożona przez 3, co kończy wykonywanie czekającego `factorial(3)`. Uzyskana wartość 6 jest przekazywana zwrotnie do funkcji wywołującej, jest tam wyświetlana za pomocą funkcji `printf`.

Podsumowując, ciąg działań związanych z wyliczaniem `factorial(3)` możemy obliczyć przy użyciu następujących działań:

```
factorial (3) = 3 * factorial (2)
              = 3 * 2 * factorial (1)
              = 3 * 2 * 1 * factorial (0)
              = 3 * 2 * 1 * 1
              = 6
```

Warto byłoby prześledzić działanie funkcji `factorial` z ołówkiem w ręku. Załóżmy, że najpierw funkcja ta jest wywoływana w celu obliczenia silni 4. Należy podać wartości zmiennych `n` i `result` przy każdym wywołaniu funkcji `factorial`.

Na tym kończymy omawianie funkcji i zmiennych w tym rozdziale. Funkcje to użyteczne narzędzie języka C. Trudno przecenić możliwość strukturyzacji programu za pomocą niewielkich, jasno określonych funkcji. W dalszej części książki funkcje będą używane bardzo intensywnie. W tej chwili należy ponownie przejrzeć tematy nie do końca zrozumiałe w tym rozdziale. Wykonanie ćwiczeń pozwoli utrwalić wiedzę.

Ćwiczenia

1. Przepisz i uruchom siedemnaście programów pokazanych w tym rozdziale. Uzyskane wyniki porównaj z wynikami pokazanymi w tekście.
2. Zmodyfikuj program 7.4 tak, aby wartość `triangularNumber` była zwracana przez funkcję. Następnie wróć do programu 4.5 i zmodyfikuj go tak, aby wywoływał nową wersję funkcji `calclulateTriangularNumber`.
3. Zmodyfikuj program 7.8 tak, aby wartość `epsilon` była przekazywana do funkcji jako parametr. Pobaw się z innymi wartościami `epsilon`, aby zobaczyć, jak wpływają one na uzyskiwany wynik pierwiastkowania.
4. Zmodyfikuj program 7.8 tak, aby wartość zmiennej `guess` była pokazywana w każdym przebiegu pętli `while`. Zwróć uwagę, jak szybko uzyskiwany wynik zdąża do pierwiastka kwadratowego. Jakie wnioski możesz wyciągnąć o liczbie iteracji, pierwiastkowanej liczbie i początkowej wartości odgadywanego pierwiastka?

5. Warunki użyte do kończenia pętli w funkcji `squareRoot` w programie 7.8 nie nadają się do obliczania pierwiastków bardzo dużych i bardzo małych liczb. Zamiast porównywać *różnicę* wartości `x` i wartości `guess`^{2}, program powinien porównywać *stosunek* dwóch wartości do jedności. Im ten stosunek jest bliżej jedności, tym dokładniejsze jest przybliżenie pierwiastka kwadratowego. Zmodyfikuj program 7.8 tak, aby użyć nowego warunku zakończenia pętli.

6. Zmodyfikuj program 7.8 tak, aby funkcja `squareRoot` przyjmowała parametr podwójnej precyzji i zwracała wartość podwójnej precyzji. Upewnij się, że wartość `epsilon` zostanie zmieniona tak, aby uwzględnić używanie liczb podwójnej precyzji.

7. Napisz funkcję podnoszącą liczbę całkowitą do dodatniej potęgi całkowitej. Niech to będzie funkcja `x_to_the_n` mająca dwa parametry całkowitoliczbowe, `x` i `n`. Funkcja zwracać będzie wartość typu `long int`, stanowiącą wynik obliczenia x^n.

8. Równanie w postaci:

   $$ax^2+bx+c = 0$$

   to równanie *kwadratowe*. Wartości *a*, *b* i *c* to stałe. Wobec tego:

   $$4x^2-17x-15 = 0$$

   to równanie kwadratowe, gdzie $a = 4$, $b = -17$, a $c = -15$. Wartości x spełniające dane równanie kwadratowe, nazywane *pierwiastkami* tego równania, możemy wyliczyć, podstawiając wartości *a*, *b* i *c* do następujących wzorów:

   $$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

   Jeśli wartość b^2-4ac, nazywana *wyznacznikiem*, jest mniejsza od zera, pierwiastki x_1 i x_2 są liczbami zespolonymi.

   Napisz program rozwiązujący równania kwadratowe. Program powinien umożliwiać użytkownikowi podanie wartości *a*, *b* i *c*. Jeśli wyznacznik jest mniejszy od zera, należy wyświetlić komunikat informujący, że pierwiastki podanego równania są liczbami zespolonymi. Jeśli nie, program powinien wyliczyć i pokazać oba pierwiastki równania. (Uwaga! Skorzystaj z funkcji `squareRoot`, którą utworzyliśmy w tym rozdziale).

9. Najmniejsza wspólna wielokrotność (NWW) dwóch dodatnich liczb całkowitych *u* i *v* to najmniejsza dodatnia liczba całkowita, która jest podzielna bez reszty przez *u* i przez *v*. Wobec tego NWW dla 15 i 10, czyli NWW(15, 10), to 30, gdyż 30 to najmniejsza liczba całkowita podzielna przez 15 i przez 10. Napisz funkcję `lcm` (od angielskiej nazwy najmniejszej wspólnej wielokrotności, *least common multiple*) mającą dwa parametry całkowitoliczbowe, zwracającą NWW dla tych parametrów. Korzystając z największego wspólnego dzielnika, oblicz najmniejszą wspólną wielokrotność za pomocą wywołania funkcji `gcd` z programu 7.6. Zależność między NWW a NWD jest następująca:

   $\mathrm{NWW}(u, v) = uv/\mathrm{NWD}(u, v)$, gdzie $u, v \geq 0$.

10. Napisz funkcję `prime` zwracającą 1, jeśli jej parametr jest liczbą pierwszą, i 0 w przeciwnym wypadku.
11. Napisz funkcję `arraySum` mającą dwa parametry — tablicę liczb całkowitych oraz liczbę elementów w tej tablicy. Funkcja ta ma zwracać jako swój wynik sumę wszystkich elementów przekazanej tablicy.
12. Macierz *M* mającą *i* wierszy i *j* kolumn można *przetransponować* w macierz *N* mającą *j* wierszy i *i* kolumn przez zwykłe przypisanie $N_{a,b} = M_{b,a}$ dla wszystkich możliwych indeksów *a* i *b*.
    a) Napisz funkcję `transposeMatrix`, która jako parametr pobierze macierze 4×5 oraz 5×4. Niech ta funkcja zapisze transponowaną macierz 4×5 w przekazanej macierzy 5×4. Napisz funkcję `main` testującą `transposeMatrix`.
    b) Korzystając z tablic o zmiennej długości, przepisz funkcję `transposeMatrix` z ćwiczenia 12.a tak, aby jako parametry pobierała też ilość wierszy i kolumn macierzy, a następnie transponowała macierz o podanych wymiarach.
13. Zmodyfikuj funkcję `sort` z programu 7.12, dodając jej trzeci parametr wskazujący, czy tablica ma być sortowana rosnąco czy malejąco. Zmodyfikuj następnie algorytm `sort` tak, aby prawidłowo sortował tablicę w żądanej kolejności.
14. Przepisz funkcje z ostatnich czterech ćwiczeń tak, aby zamiast parametrów używały zmiennych globalnych, na przykład program z poprzedniego ćwiczenia powinien sortować tablicę globalną.
15. Zmodyfikuj program 7.14 tak, aby ponownie prosił użytkownika o podanie podstawy docelowego systemu liczbowego, jeśli podstawa podana wcześniej jest nieprawidłowa. Zmodyfikowany program powinien ponawiać swoje żądania tak długo, aż otrzyma prawidłową odpowiedź.
16. Zmodyfikuj program 7.14 tak, aby użytkownik mógł konwertować dowolnie wiele liczb całkowitych. Niech program kończy swoje działanie, kiedy jako liczba konwertowana podane zostanie 0.

8

Struktury

W rozdziale 6. omówiliśmy tablice, które pozwalają grupować w całość elementy tego samego typu. Aby odwołać się do elementu tablicy, wystarczy wskazać tablicę i indeks potrzebnego elementu.

Język C zawiera inne narzędzie do grupowania elementów. Są to struktury, którymi zajmiemy się w tym rozdziale. Jak się przekonamy, są to potężne narzędzia, które wykorzystuje się w bardzo wielu programach w języku C.

W tym rozdziale znajdują się podstawowe wiadomości na temat struktur danych:

- definiowanie struktur,
- przekazywanie struktur do funkcji,
- tablice struktur,
- struktury tablic.

Podstawowe wiadomości o strukturach

Załóżmy, że mamy zapisać w programie datę — na przykład 25 września 2015 roku. Potem użyjemy jej w nagłówku wydruku z programu albo coś będziemy na jej podstawie obliczać. Naturalną metodą byłoby przypisanie dnia zmiennej `day`, miesiąca zmiennej całkowitoliczbowej o nazwie `month`, a roku zmiennej `year`. Zatem świetnie nadawałyby się do tego instrukcje

```
int day = 25, month = 9, year = 2015;
```

Jest to całkiem dobre rozwiązanie. Załóżmy jednak, że potrzebna jest nam w programie data nabycia jakiegoś towaru. Możemy analogicznie zdefiniować kolejne trzy zmienne: `purchaseDay`, `purchaseMonth` i `purchaseYear`. Kiedy potrzebna będzie nam data zakupu, skorzystamy z tych trzech zmiennych.

Korzystając z opisanej metody, dla każdej interesującej nas daty musimy jednak śledzić trzy osobne zmienne, które w istocie powinny być ze sobą logicznie powiązane. Znacznie lepiej byłoby, gdybyśmy mogli te zmienne jakoś połączyć w trójki. Do tego właśnie w języku C służą struktury.

Struktura na daty

Możemy zdefiniować w języku C strukturę `date` mającą trzy składniki odpowiadające dniowi, miesiącowi i rokowi. Składnia takiej definicji jest dość prosta:

```
struct date
{
    int day;
    int month;
    int year;
};
```

Z powyższej definicji wynika, że struktura `date` ma trzy *pola*: `day`, `month` i `year`, opisujące odpowiednio dzień, miesiąc i rok. W pewnym sensie taka definicja tworzy nowy typ zmiennych, którego można dalej używać podczas definiowania zmiennych, na przykład

```
struct date today;
```

Można też zadeklarować zmienną `purchaseDate` jako zmienną takiego samego typu:

```
struct date purchaseDate;
```

Można też po prostu połączyć obie definicje w jednym wierszu:

```
struct date today, purchaseDate;
```

W przeciwieństwie do zmiennych typów `int`, `float` czy `char`, do obsługi struktur trzeba stosować specjalną składnię. Do pola struktury sięgamy, podając nazwę zmiennej, dalej kropkę i nazwę pola. Na przykład aby ustawić pole `day` w zmiennej `today` na 25, piszemy

```
today.day = 25;
```

Odnotujmy, że niedopuszczalne jest stosowanie spacji między nazwą zmiennej a kropką i między kropką a nazwą pola. Aby ustawić pole `year` zmiennej `today` na 2015, można użyć wyrażenia

```
today.year = 2015;
```

W końcu, aby sprawdzić, czy pole `month` ma wartość 12, stosuje się instrukcje typu:

```
if ( today.month == 12 )
    nextMonth = 1;
```

Spróbujmy ustalić wynik działania poniższej instrukcji:

```
if (today.month == 1  &&  today.day == 1 )
    printf ("Szczęśliwego Nowego Roku!\n");
```

Program 8.1 łączy omówione dotąd zagadnienia w konkretnym programie.

Program 8.1. **Ilustracja użycia struktur**

```
// Program pokazujący użycie struktur

#include <stdio.h>

int main (void)
{
```

```
    struct date
    {
        int  day;
        int  month;
        int  year;
    };

    struct date  today;

    today.day = 25;
    today.month = 9;
    today.year = 2015;

    printf ("Dzisiaj jest %.2i.%i.%i.\n", today.year % 100,
             today.month, today.day);

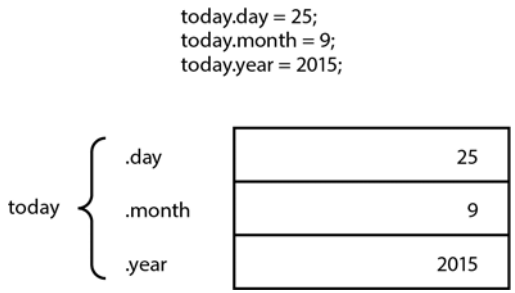
    return 0;
}
```

Program 8.1. **Wyniki**

```
Dzisiaj jest 15.9.25.
```

Pierwsza instrukcja w funkcji `main` zawiera definicję struktury `date` mającej trzy zmienne całkowite: `day`, `month` i `year`. W drugiej instrukcji deklarujemy zmienną `today` typu `struct date`. Pierwsza instrukcja opisuje kompilatorowi C postać struktury, natomiast nie jest na nią rezerwowana żadna pamięć. Druga instrukcja deklaruje zmienną typu `struct date`, wobec czego ta już *powoduje* zarezerwowanie pamięci na trzy liczby całkowite wchodzące w skład struktury. Trzeba dobrze zrozumieć różnicę między definiowaniem struktury a deklarowaniem zmiennych danego typu strukturalnego.

Po zadeklarowaniu zmiennej `today` program przechodzi do przypisania wartości wszystkim trzem polom zmiennej `today`, jak to pokazano na rysunku 8.1.



Rysunek 8.1. Przypisywanie wartości polom struktury

Po zrobieniu przypisań wartości z pól struktury są pokazywane instrukcją `printf`. Przed przekazaniem roku do `printf` wyliczana jest reszta z dzielenia `today.year`, dzięki czemu rok pokazywany jest jako 15. Przypomnijmy, że łańcuch formatujący `%.2i` mówi, że wyświetlana będzie dwucyfrowa liczba całkowita wypełniana z przodu zerami. Dzięki

temu, jeśli końcówka będzie mieścić się w przedziale od 01 do 09, ostatnie dwie cyfry roku zawsze będą prawidłowo wyświetlane.

Użycie struktur w wyrażeniach

Kiedy wylicza się wyrażenia, pola struktur są traktowane tak samo jak zwykłe zmienne. Wobec tego dzielenie całkowitej liczby należącej do struktury przez inną liczbę całkowitą jest dzieleniem całkowitoliczbowym, jak w wyrażeniu

```
century = today.year / 100 + 1;
```

Załóżmy, że chcemy napisać prosty program przyjmujący jako dane wejściowe aktualną datę i pokazujący użytkownikowi datę jutrzejszą. W pierwszej chwili wydaje się, że jest to proste. Można użytkownika poprosić o podanie daty, a następnie obliczyć datę jutrzejszą za pomocą kilku instrukcji:

```
tomorrow.day   = today.day;
tomorrow.month = today.month;
tomorrow.year  = today.year;
```

Faktycznie, powyższe instrukcje wystarczają w przypadku większości dat, ale nieprawidłowo obsłużone będą sytuacje, kiedy:

1. data dzisiejsza wypada na koniec miesiąca,
2. data dzisiejsza wypada na koniec roku (czyli jeśli jest to 31 grudnia).

Aby sprawdzić, czy dana data jest końcem miesiąca, można przygotować tablicę liczb całkowitych zawierającą informację o liczbie dni w poszczególnych miesiącach. Wtedy sprawdzając komórkę odpowiadającą danemu miesiącowi uzyskujemy liczbę dni tego miesiąca. Instrukcja

```
int daysPerMonth[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

definiuje tablicę `daysPerMonth` zawierającą 12 liczb całkowitych. Dla każdego miesiąca `i` komórka `daysPerMonth[i – 1]` zawiera liczbę dni w tym miesiącu. Zatem liczba dni kwietnia, czwartego miesiąca roku, to `daysPerMonth[3]`, czyli 30. Można byłoby też zadeklarować tablicę 13-elementową, w której `daysPerMonth[i]` zawiera liczbę dni miesiąca `i`. Wtedy możliwy jest bezpośredni dostęp do tablicy według odpowiedniego numeru miesiąca, a nie według miesiąca minus 1. Decyzja o tym, czy użyć 12 czy 13 elementów tablicy, jest kwestią wyłącznie osobistych preferencji.

Teraz, jeśli stwierdzimy, że dzisiejsza data wypada na koniec miesiąca, datę jutrzejszą możemy ustalić, dodając 1 do numeru miesiąca, a dzień ustawiając na 1.

Aby rozwiązać drugi ze wspomnianych powyżej problemów, trzeba sprawdzić, czy dzisiejsza data nie wypada na koniec miesiąca i czy miesiącem tym jest 12. Wtedy jutrzejszy miesiąc i dzień ustawiamy na 1, a rok zwiększamy o 1.

Program 8.2 prosi użytkownika o wprowadzenie aktualnej daty, wyświetla datę jutrzejszą i pokazuje wynik.

Program 8.2. **Ustalanie jutrzejszej daty**

```
// Program wyznaczający jutrzejszą datę

#include <stdio.h>

int main (void)
{
    struct date
    {
        int  day;
        int  month;
        int  year;
    };
    struct date today, tomorrow;

    const int daysPerMonth[12] = { 31, 28, 31, 30, 31, 30,
                                   31, 31, 30, 31, 30, 31 };

    printf ("Podaj dzisiejszą datę (dd mm rrrr): ");
    scanf ("%i%i%i", &today.day, &today.month, &today.year);

    if ( today.day != daysPerMonth[today.month - 1] ) {
        tomorrow.day = today.day + 1;
        tomorrow.month = today.month;
        tomorrow.year = today.year;
    }
    else if ( today.month == 12 ) {     // koniec roku
        tomorrow.day = 1;
        tomorrow.month = 1;
        tomorrow.year = today.year + 1;
    }
    else {                              // koniec miesiąca
        tomorrow.day = 1;
        tomorrow.month = today.month + 1;
        tomorrow.year = today.year;
    }

    printf ("Jutrzejsza data to %i.%i.%.2i.\n", tomorrow.day,
                tomorrow.month, tomorrow.year % 100);

    return 0;
}
```

Program 8.2. **Wyniki**

```
Podaj dzisiejszą datę (dd mm rrrr): 17 12 2013
Jutrzejsza data to 18.12.13.
```

Program 8.2. **Wyniki (ponowne uruchomienie)**

```
Podaj dzisiejszą datę (dd mm rrrr): 31 12 2014
Jutrzejsza data to 1.1.15.
```

Program 8.2. **Wyniki (ponowne uruchomienie)**

```
Podaj dzisiejszą datę (dd mm rrrr): 28 2 2012
Jutrzejsza data to 1.3.12.
```

Jeśli przyjrzymy się dokładnie wynikom działania programu, szybko zauważymy, że wkradł się błąd: po 28 lutego 2012 roku następuje 1 marca 2012 zamiast 29 lutego 2012. Program „zapomniał" o latach przestępnych! Problemem tym zajmiemy się za chwilę. Najpierw musimy przeanalizować program i jego logikę.

Po definicji struktury `date` deklarowane są dwie zmienne typu `struct date`: `today` i `tomorrow`. Następnie program prosi użytkownika o podanie aktualnej daty. Trzy podane liczby całkowite są umieszczane w polach `today.day`, `today.month` i `today.year`. Następnie porównując `today.day` z `daysPerMonth[today.month - 1]`, sprawdzamy, czy dany dzień wypada na koniec miesiąca. Jeśli nie, datę jutrzejszą wyliczamy, zwiększając po prostu dzień o 1, natomiast miesiąc i rok pozostawiamy bez zmian.

Jeśli aktualna data wypada na koniec miesiąca, program sprawdza, czy mamy do czynienia z końcem roku. Jeśli miesiąc jest równy 12, mamy 31 grudnia, więc datą jutrzejszą jest 1 stycznia następnego roku. Jeśli miesiąc nie jest równy 12, data jutrzejsza to pierwszy dzień następnego miesiąca (tego samego roku).

Kiedy jutrzejsza data zostanie wyliczona, pokazujemy ją za pomocą funkcji `printf` i na tym działanie programu się kończy.

Funkcje i struktury

Teraz możemy zająć się problemem, który wykryliśmy w omawianym programie. Nasz program zakłada, że luty zawsze ma 28 dni, więc zapytany o dzień po 28 lutego, zawsze odpowie, że jest to 1 marca. Musimy zrobić dodatkowy test uwzględniający lata przestępne. Jeśli rok jest przestępny, a miesiąc to luty, liczba dni w tym miesiącu to 29. We wszystkich innych przypadkach wystarcza nam dotychczasowa tablica `daysPerMonth`.

Dobrym sposobem na uwzględnienie opisanych zmian w programie 8.2 byłoby przygotowanie funkcji `numberOfDays` określającej liczbę dni w miesiącu. Funkcja ta sprawdzałaby, czy rok jest przestępny, i odczytywałaby wartość z tablicy `daysPerMonth`. W funkcji `main` wystarczy zmienić instrukcję `if` porównującą `today.day` z `daysPerMonth[today.month - 1]`. Teraz chcemy porównywać wartość `today.day` z wartością zwracaną przez funkcję `numberOfDays`.

Przeanalizujmy starannie program 8.3, patrząc, co jest przekazywane funkcji `numberOfDays` jako parametr.

Program 8.3. **Poprawiona wersja programu ustalającego jutrzejszą datę**

```
// Program wyznaczający jutrzejszą datę

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

```
struct date
{
    int  day;
    int  month;
    int  year;
};

int main (void)
{
    struct date today, tomorrow;
    int numberOfDays (struct date d);

    printf ("Podaj dzisiejszą datę (dd mm rrrr): ");
    scanf ("%i%i%i", &today.day, &today.month, &today.year);

    if ( today.day != numberOfDays (today) ) {
        tomorrow.day = today.day + 1;
        tomorrow.month = today.month;
        tomorrow.year = today.year;
    }
    else if ( today.month == 12 ) {     // koniec roku
        tomorrow.day = 1;
        tomorrow.month = 1;
        tomorrow.year = today.year + 1;
    }
    else {                              // koniec miesiąca
        tomorrow.day = 1;
        tomorrow.month = today.month + 1;
        tomorrow.year = today.year;
    }

    printf ("Jutrzejsza data to %i.%i.%.2i.\n", tomorrow.day,
                tomorrow.month, tomorrow.year % 100);

    return 0;
}

// Funkcja określająca liczbę dni w miesiącu

int numberOfDays (struct date d)
{
    int   days;
    bool  isLeapYear (struct date d);
    const int daysPerMonth[12] = { 31, 28, 31, 30, 31, 30,
                                   31, 31, 30, 31, 30, 31 };

    if ( isLeapYear (d) == true  &&  d.month == 2 )
        days = 29;
    else
        days = daysPerMonth[d.month - 1];

    return days;
}

// Funkcja sprawdzająca, czy rok jest przestępny

bool isLeapYear (struct date d)
{
```

```
    bool  leapYearFlag;

    if ( (d.year % 4 == 0  &&  d.year % 100 != 0)  ||
                d.year % 400 == 0 )
        leapYearFlag = true;     // rok jest przestępny
    else
        leapYearFlag = false;    // rok nie jest przestępny


    return leapYearFlag;
}
```

Program 8.3. **Wyniki**

```
Podaj dzisiejszą datę (dd mm rrrr): 28 2 2016
Jutrzejsza data to 29.2.16.
```

Program 8.3. **Wyniki (ponowne uruchomienie)**

```
Podaj dzisiejszą datę (dd mm rrrr): 28 2 2014
Jutrzejsza data to 1.3.14.
```

Pierwsze, co zwraca uwagę w powyższym programie, to zdefiniowanie struktury `date` poza jakąkolwiek funkcją, na samym początku programu. Dzięki temu definicja ta jest znana w całym pliku. Definicje struktur zachowują się bardzo podobnie do definicji zmiennych — jeśli struktura jest zdefiniowana w jakiejś funkcji, tylko ta funkcja wie o jej istnieniu. Wtedy mamy do czynienia z *lokalną* definicją struktury. Jeśli zdefiniujemy strukturę poza funkcjami, definicja taka jest *globalna*. Globalna definicja struktury sprawia, że dowolne zmienne definiowane gdziekolwiek dalej w programie (wewnątrz funkcji lub poza nimi) mają typ takiej właśnie struktury.

W funkcji `main` mamy deklarację prototypu funkcji:

```
int numberOfDays (struct date d);
```

informującą kompilator, że funkcja `numberOfDays` zwraca liczbę całkowitą i że ma ona jeden parametr typu `struct date`.

Zamiast porównywać wartość `today.day` z `daysPerMonth[today.month – 1]`, jak to robiliśmy w poprzednim programie, używamy instrukcji:

```
if ( today.day != numberOfDays (today) )
```

Jak widać, jako parametr przekazujemy strukturę `today`. W funkcji `numberOfDays` musimy mieć odpowiednią deklarację, która poinformuje system, że argumentem powinna być struktura:

```
int numberOfDays (struct date d)
```

Tak jak w przypadku zwykłych zmiennych, a w przeciwieństwie do tablic, wszelkie zmiany robione wewnątrz funkcji na wartościach ze struktury przekazanej jako parametr nie mają wpływu na strukturę oryginalną. Wpływają jedynie na kopię struktury utworzoną w chwili wywołania funkcji.

Funkcja `numberOfDays` zaczyna od sprawdzenia, czy dany rok jest przestępny i czy miesiącem jest luty. Wywołujemy tutaj kolejną funkcję — `isLeapYear`. Za chwilę tą funkcją zajmiemy się dokładniej. Na razie z instrukcji `if` w postaci:

```
if ( isLeapYear (d) == true  &&  d.month == 2 )
```

możemy się domyślać, że jeśli funkcja `isLeapYear` zwróci `true`, rok jest przestępny; jeśli rok nie jest przestępny, funkcja ta zwróci `false`. Wnioski te są zbieżne z tym, co mówiliśmy o zmiennych logicznych w rozdziale 6. Przypomnijmy, że standardowy plik nagłówkowy `<stdbool.h>` zawiera definicje wartości `bool`, `true` i `false`; plik ten włączyliśmy na początku programu 8.3.

Jeśli chodzi o cytowaną powyżej instrukcję `if`, godnym odnotowania jest jeszcze dobór nazwy funkcji — `isLeapYear`. Taka nazwa czyni naszą instrukcją wyjątkowo czytelną i od razu sugeruje, że funkcja zwraca wartość typu tak/nie.

Wróćmy teraz do programu. Jeśli okazuje się, że mamy do czynienia z lutym w roku przestępnym, zmienna `days` uzyskuje wartość 29. W przeciwnym wypadku wartość tej zmiennej odczytujemy z odpowiedniej komórki tablicy `daysPerMonth`, korzystając jak zwykle z miesiąca jako indeksu. Wartość zmiennej `days` jest zwracana do funkcji `main`, gdzie dalsze działanie programu odbywa się tak samo, jak w programie 8.2.

Funkcja `isLeapYear` jest dość prosta: sprawdzamy rok w strukturze `date` przekazanej jako parametr i jeśli jest on przestępny, zwracamy `true`; jeśli nie jest przestępny, zwracamy `false`.

W ramach ćwiczenia poprawiającego sposób organizacji programu cały proces wyznaczania jutrzejszej daty wydzielmy do odrębnej funkcji. Możemy ją nazwać `dateUpdate` i przekazywać jej jako parametr datę dzisiejszą. Funkcja wyliczy datę następnego dnia i *zwróci* tę nową datę. Program 8.4 pokazuje realizację takiego programu.

Program 8.4. **Poprawiona wersja programu ustalającego jutrzejszą datę (wersja 2.)**

```
// Program wyznaczający jutrzejszą datę

#include <stdio.h>
#include <stdbool.h>

struct date
{
    int  day;
    int  month;
    int  year;
};

// Funkcja wyliczająca jutrzejszą datę

struct date  dateUpdate (struct date  today)
{
    struct date tomorrow;
    int numberOfDays (struct date d);

    if ( today.day != numberOfDays (today) ) {
        tomorrow.day = today.day + 1;
        tomorrow.month = today.month;
```

```
        tomorrow.year = today.year;
    }
    else if ( today.month == 12 ) {     // koniec roku
        tomorrow.day = 1;
        tomorrow.month = 1;
        tomorrow.year = today.year + 1;
    }
    else {                              // koniec miesiąca
        tomorrow.day = 1;
        tomorrow.month = today.month + 1;
        tomorrow.year = today.year;
    }

    return tomorrow;
}

// Funkcja określająca liczbę dni w miesiącu

int numberOfDays (struct date d)
{
    int   days;
    bool  isLeapYear (struct date d);
    const int daysPerMonth[12] = { 31, 28, 31, 30, 31, 30,
                                   31, 31, 30, 31, 30, 31 };

    if ( isLeapYear (d) == true  &&  d.month == 2 )
        days = 29;
    else
        days = daysPerMonth[d.month - 1];

    return days;
}

// Funkcja sprawdzająca, czy rok jest przestępny

bool isLeapYear (struct date d)
{
    bool  leapYearFlag;

    if ( (d.year % 4 == 0  &&  d.year % 100 != 0)  ||
                d.year % 400 == 0 )
        leapYearFlag = true;     // rok jest przestępny
    else
        leapYearFlag = false;    // rok nie jest przestępny

    return leapYearFlag;
}

int main (void)
{
    struct date  dateUpdate (struct date  today);
    struct date  thisDay, nextDay;

    printf ("Podaj dzisiejszą datę (dd mm rrrr): ");
    scanf ("%i%i%i", &thisDay.day, &thisDay.month,
                     &thisDay.year);
```

```
    nextDay = dateUpdate (thisDay);

    printf ("Jutrzejsza data to %i.%i.%.2i.\n", nextDay.day,
            nextDay.month, nextDay.year % 100);

    return 0;
}
```

Program 8.4. **Wyniki**

```
Podaj dzisiejszą datę (dd mm rrrr): 28 2 2016
Jutrzejsza data to 29.2.16.
```

Program 8.4. **Wyniki (ponowne uruchomienie)**

```
Podaj dzisiejszą datę (mm dd rrrr): 22 2 2015
Jutrzejsza data to 23.2.15.
```

W funkcji `main` mamy instrukcję:

```
nextDay = dateUpdate (thisDay);
```

pokazującą możliwość przekazania do funkcji struktury i jednocześnie zwrócenia struktury. Funkcja `dateUpdate` ma deklarację, z której wynika, że funkcja ta zwraca wartość typu `struct date`. W funkcji tej mamy ten sam kod, który w programie 8.3 był w funkcji `main`. Funkcje `numberOfDays` i `isLeapYear` pozostają niezmienione.

Czytelnicy powinni się upewnić, że dobrze rozumieją hierarchię wywołań funkcji w powyższym programie. Funkcja `main` wywołuje `dateUpdate`, ta z kolei wywołuje `numberOfDays`, a ta wywołuje `isLeapYear`.

Struktura na czas

Załóżmy, że w programie musimy przechowywać różne czasy wyrażone jako godziny, minuty i sekundy. Mamy już strukturę `date` pomagającą nam zorganizować dzień, miesiąc i rok, naturalne wydaje się użycie analogicznej struktury `time` na godziny, minuty i sekundy. Definicja takiej struktury jest dość prosta:

```
struct time
{
    int  hour;
    int  minutes;
    int  seconds;
};
```

Większość komputerów używa czasu 24-godzinnego. Dzięki temu nie trzeba dodawać, czy chodzi o godziny przedpołudniowe czy popołudniowe, jak to ma miejsce w przypadku czasu 12-godzinnego. Godziny liczymy od 0 o północy co 1, aż do 23, czyli jedenastej wieczorem.

Właściwie wszystkie komputery mają wbudowany nigdy niewyłączany zegar wewnętrzny. Służy on do różnych celów: informowania użytkownika, która jest godzina, uruchamiania programów o żądanej godzinie, rejestrowania, o której godzinie zaszło jakieś zdarzenie,

i tak dalej. Zwykle z zegarem systemowym współpracuje przynajmniej jeden program. Programy takie mogą być na przykład uruchamiane co sekundę, aby uaktualnić czas zapisany gdzieś w pamięci.

Załóżmy, że chcemy zrealizować program opisany powyżej, czyli aktualizujący co sekundę czas. Jeśli przez chwilę się nad tym zastanowimy, stwierdzimy, że mamy problem bardzo podobny jak w przypadku aktualizacji daty o jeden dzień.

Tak jak z określaniem następnego dnia wiążą się pewne wymagania, tak samo mamy pewne wymagania dotyczące aktualizacji czasu. W szczególności trzeba uwzględnić następujące przypadki brzegowe:

1. Jeśli liczba sekund osiąga 60, trzeba sekundy ustawić na 0, a zwiększyć o 1 liczbę minut.
2. Jeśli liczba minut osiąga 60, trzeba minuty ustawić na 0, a zwiększyć o 1 liczbę godzin.
3. Jeśli liczba godzin osiąga 24, trzeba ustawić godziny, minuty i sekundy na 0[1].

Program 8.5 wykorzystuje funkcję `timeUpdate` mającą za parametr aktualny czas i zwracającą czas przesunięty o jedną sekundę.

Program 8.5. **Aktualizacja czasu o sekundę**

```
// Program aktualizujący czas o sekundę

#include <stdio.h>

struct time
{
    int  hour;
    int  minutes;
    int  seconds;
};

int main (void)
{
    struct time  timeUpdate (struct time now);
    struct time  currentTime, nextTime;

    printf ("Podaj czas (gg:mm:ss): ");
    scanf ("%i:%i:%i", &currentTime.hour,
           &currentTime.minutes, &currentTime.seconds);

    nextTime = timeUpdate (currentTime);

    printf ("Zaktualizowany czas to %.2i:%.2i:%.2i\n", nextTime.hour,
            nextTime.minutes, nextTime.seconds );

    return 0;
```

[1] W Europie w tego typu programach trzeba pamiętać jeszcze o uwzględnieniu zmiany czasu dwa razy do roku. Powoduje to oczywiście, że nie możemy operować na czasie w całkowitym oderwaniu od daty — *przyp. tłum.*

```
}

// Funkcja aktualizująca czas o sekundę

struct time timeUpdate (struct time now)
{
    ++now.seconds;
    if ( now.seconds == 60 ) {       // następna minuta
        now.seconds = 0;
        ++now.minutes;

        if ( now.minutes == 60 ) {   // następna godzina
            now.minutes = 0;
            ++now.hour;

            if ( now.hour == 24 )    // północ
                now.hour = 0;
        }
    }

    return now;
}
```

Program 8.5. **Wyniki**

```
Podaj czas (gg:mm:ss): 12:23:55
Zaktualizowany czas to 12.23.56
```

Program 8.5. **Wyniki (ponowne uruchomienie)**

```
Podaj czas (gg:mm:ss): 16:12:59
Zaktualizowany czas to 16:13:00
```

Program 8.5. **Wyniki (ponowne uruchomienie)**

```
Podaj czas (gg:mm:ss): 23:59:59
Zaktualizowany czas to 00:00:00
```

Funkcja `main` prosi użytkownika o podanie czasu, `scanf` odczytuje podane wartości, korzystając z łańcucha formatującego:

```
"%i:%i:%i"
```

Podanie znaków nieformatujących, takich jak `':'`, informuje funkcję `scanf`, że dokładnie takie znaki mają pojawić się w danych wejściowych. Wobec tego łańcuch formatujący z programu 8.5 mówi, że mają być podane trzy liczby całkowite oddzielone od siebie dwukropkami. W rozdziale 15., poświęconym operacjom wejścia i wyjścia w języku C, dowiemy się, jak za pomocą wartości zwracanych przez `scanf` sprawdzić, czy użytkownik podał prawidłowe dane.

Kiedy podany zostanie czas, program wywoła funkcję `timeUpdate`, przekazując jej `currentTime` jako parametr. Wynik zwrócony przez tę funkcję zostanie przypisany zmiennej `nextTime` typu `struct time`, a następnie wyświetlony przez funkcję `printf`.

Funkcja `timeUpdate` zaczyna swoje działanie od „popchnięcia” czasu ze zmiennej `now` o jedną sekundę. Następnie sprawdzane jest, czy liczba sekund osiągnęła wartość 60. Jeśli tak, sekundy są zerowane, a liczba minut jest zwiększana o 1. Potem sprawdzane jest, czy liczba minut osiągnęła 60; jeśli tak, minuty są zerowane, a liczba godzin jest powiększana o 1. Jeśli spełnione były oba dotychczasowe warunki, sprawdzane jest, czy liczba godzin wynosi 24. Jeśli tak, jest dokładnie północ i wtedy liczbę godzin ustawiamy na 0. W końcu funkcja zwraca wartość zmiennej `now`, która teraz zawiera zaktualizowany czas.

Inicjalizowanie struktur

Inicjalizowanie struktur przypomina inicjalizowanie tablic — po prostu podaje się wartości poszczególnych pól w parze nawiasów klamrowych, wartości te rozdziela się przecinkami.

Aby zainicjalizować zmienną `today` typu `struct date` na 2 lipca 2015 roku, używamy instrukcji:

```
struct date today = { 2, 7, 2015 };
```

Instrukcja:

```
struct time this_time = { 3, 29, 55 };
```

definiuje zmienną `this_time` typu `struct time` i inicjalizuje ją wartością 3:29:55. Tak jak w przypadku innych zmiennych, jeśli `this_time` jest zmienną lokalną, jest inicjalizowana przy każdym wejściu do funkcji. Jeśli jest zmienną statyczną (poprzedzona zostanie słowem kluczowym `static`), będzie inicjalizowana tylko raz, podczas uruchomienia programu. W obu wypadkach wartości podawane w nawiasach klamrowych muszą być wyrażeniami stałymi.

Tak jak w przypadku inicjalizacji tablic, można podać mniej wartości niż struktura ma pól. Na przykład instrukcja:

```
struct time time1 = { 12, 10 };
```

ustawia `time1.hour` na 12, `time1.minutes` na 10, ale nie podaje wartości początkowej dla pola `time1.seconds`. W takiej sytuacji wartość początkowa tego ostatniego pola jest nieokreślona.

Można też podać nazwy pól w liście inicjalizacyjnej. Robi się to, korzystając ze składni:

```
.pole = wartość
```

W ten sposób można inicjalizować pola w dowolnej kolejności albo inicjalizować tylko wybrane pola. Na przykład:

```
struct time time1 = { .hour = 12, .minutes = 10 };
```

ustawia pola zmiennej `time1` tak samo jak w poprzednim przykładzie. Instrukcja:

```
struct date today = { .year = 2015 };
```

ustawia tylko pole `year` zmiennej `today` na 2015.

Literały złożone

W jednej instrukcji można przypisać strukturze więcej niż jedną wartość — wykorzystuje się do tego *literały złożone*. Na przykład: jeśli założymy, że zmienna `today` została zadeklarowana jako `struct date`, przypisanie pól `today`, takie samo jak w programie 8.1, można zrealizować instrukcją:

```
today = {struct date} { 25, 9, 2015 };
```

Zauważmy, że taka instrukcja może pojawić się w dowolnym miejscu programu: nie jest to deklaracja. Operator rzutowania typu informuje kompilator o typie wyrażenia, w tym wypadku `struct date`; za nim podawane są we właściwej kolejności wartości poszczególnych pól struktury. Wartości te są podawane tak samo jak przy inicjalizacji struktury.

Można też podać wartości, korzystając z zapisu *`.pole`*:

```
today = {struct date} { .day = 25, .month = 9, .year = 2015 };
```

Zaletą korzystania z takiego zapisu jest to, że wartości można podawać w dowolnej kolejności. Jeśli nazwy pól nie są jawnie podane, wartości trzeba podawać w takiej kolejności, w jakiej zadeklarowano je w strukturze.

Oto przykład pokazujący funkcję `dateUpdate` z programu 8.4 przepisaną tak, aby skorzystać z literałów złożonych:

```
// Funkcja wyliczająca jutrzejszą datę, wykorzystująca literały złożone

struct date dateUpdate { struct date today}
{
    struct date tomorrow;
    int numberOfDays {struct date d};

    if ( today.day != numberOfDays (today) )
        tomorrow = (struct date) { today.day + 1, today.month, today.year };
    else if ( today.month == 12 )     // koniec roku
        tomorrow = (struct date) { 1, 1, today.year + 1 };
    else                              // koniec miesiąca
        tomorrow = (struct date) { today.month + 1, 1, today.year };

    return tomorrow;
}
```

To, czy w swoich programach będziemy używać literałów złożonych, zależy tylko od osobistych preferencji. W pokazanym przykładzie użycie takich literałów poprawia czytelność funkcji `dateUpdate`.

Literały złożone mogą być też używane wszędzie tam, gdzie można używać wyrażeń ze strukturami. Jak najbardziej poprawny (choć całkiem niepraktyczny) jest zapis:

```
nextDay = dateUpdate ((struct date) { 11, 5, 2015} );
```

Funkcja `dateUpdate` oczekuje parametru typu `struct date`; dokładnie takiego typu jest literał złożony, który przekazujemy funkcji.

Tablice struktur

Widzieliśmy już, jak bardzo przydatne są struktury do logicznego grupowania powiązanych ze sobą wartości. Na przykład użycie struktury `time` pozwala nam mieć jedną zmienną tam, gdzie normalnie potrzebowaliśmy trzech zmiennych. Aby w programie zapisać 10 różnych godzin, potrzebujemy 10 zmiennych zamiast 30.

Jeszcze lepszą metodą przechowywania 10 różnych godzin byłoby połączenie dwóch potężnych technik języka C: struktur i tablic. Język C nie ogranicza tablic do typów prostych: jak najbardziej poprawne jest zdefiniowanie *tablicy struktur*. Na przykład instrukcja

```
struct time experiments[10];
```

definiuje tablicę `experiments` zawierającą 10 elementów. Każdy element tablicy jest typu `struct time`. Analogicznie definicja:

```
struct date birthdays[15];
```

opisuje tablicę `birthdays` zawierającą 15 elementów typu `struct date`. Odwołanie się do poszczególnych pól struktury w tablicy też jest normalnym działaniem. Aby ustawić drugie urodziny z tablicy `birthdays` na 8 sierpnia 1986 roku, można użyć trzech instrukcji:

```
birthdays[1].day   = 8;
birthdays[1].month = 8;
birthdays[1].year  = 1986;
```

Aby przekazać funkcji `checkTime` całą strukturę `time` z `experiments[4]`, wskazujemy normalnie komórkę tablicy:

```
checkTime (experiments[4]);
```

W deklaracji funkcji `checkTime` musi być naturalnie przewidziany parametr typu `struct time`:

```
void checkTime (struct time t0)
{
    .
    .
    .
}
```

Inicjalizacja tablic zawierających struktury odbywa się analogicznie do inicjalizacji tablic wielowymiarowych. Wobec tego instrukcja:

```
struct time runTime [5] =
    {  {12, 0, 0},  {12, 30, 0},  {13, 15, 0}  };
```

ustawia pierwsze trzy czasy w tablicy `runTime` odpowiednio na 12:00:00, 12:30:00 i 13:15:00. Wewnętrzne pary nawiasów klamrowych są opcjonalne, więc powyższą instrukcję można byłoby też zapisać jako:

```
struct time runTime [5] =
    {  12, 0, 0,  12, 30, 0,  13, 15, 0  };
```

Instrukcja:

```
struct time runTime[5] =
    { [2] = {12, 0, 0} };
```

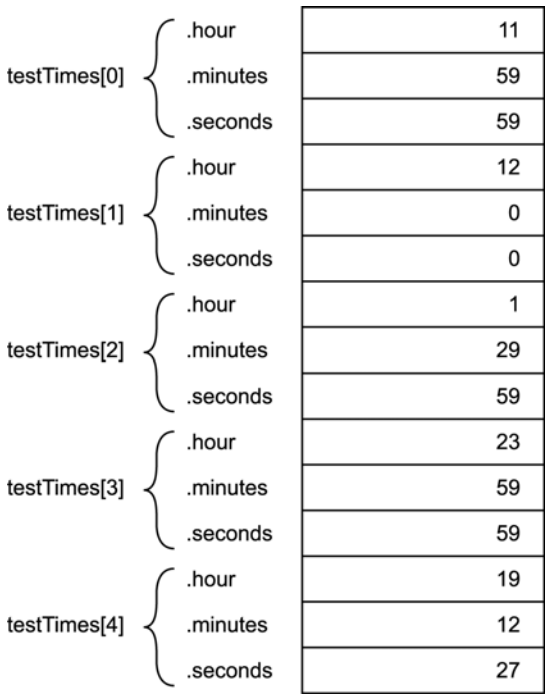
inicjalizuje trzeci element tablicy podaną wartością, z kolei instrukcja:

```
static struct time runTime[5] = { [1].hour = 12, [1].minutes = 30 };
```

ustawia godziny i minuty drugiego elementu tablicy `runTime` odpowiednio na 12 i 30.

Program 8.6 przygotowuje tablicę struktur na czas o nazwie `testTimes`, a następnie wywołuje funkcję `timeUpdate` z programu 8.5.

W programie 8.6 tablica `testTimes` zawiera pięć różnych czasów. Elementy tej tablicy otrzymują wartości początkowe odpowiednio 11:59:59, 12:00:00, 1:29:59, 23:59:59 oraz 19:12:27. Rysunek 8.2 pomoże zrozumieć, jak naprawdę przedstawia się tablica `testTimes` w pamięci komputera. Do poszczególnych struktur `time` z tablicy `testTimes` sięgamy, korzystając z indeksów od 0 do 4. Poszczególne pola (`hour`, `minutes` i `seconds`) są dostępne przez dodanie kropki i dalej nazwy pola.



Rysunek 8.2. Tablica testTimes w pamięci

Program 8.6 dla każdego elementu z tablicy `testTimes` pokazuje czas w tym elemencie zawarty, wywołuje funkcję `timeUpdate` z programu 8.5 i wyświetla zaktualizowany czas.

Program 8.6. **Ilustracja użycia tablic struktur**

```
// Program pokazujący użycie tablic struktur

#include <stdio.h>

struct time
{
    int  hour;
    int  minutes;
    int  seconds;
};

int main (void)
{
    struct time  timeUpdate (struct time now);
    struct time  testTimes[5] =
        {  { 11, 59, 59 }, { 12, 0, 0 }, { 1, 29, 59 },
           { 23, 59, 59 }, { 19, 12, 27 }};
    int i;

    for ( i = 0;  i < 5;  ++i ) {
        printf ("Czas to %.2i:%.2i:%.2i", testTimes[i].hour,
                testTimes[i].minutes, testTimes[i].seconds);

        testTimes[i] = timeUpdate (testTimes[i]);

        printf ( " ...sekundę później to %.2i:%.2i:%.2i\n",
            testTimes[i].hour, testTimes[i].minutes, testTimes[i].seconds);
    }

    return 0;
}

struct time timeUpdate (struct time now)
{
    ++now.seconds;

    if ( now.seconds == 60 ) { // następna minuta
        now.seconds = 0;
        ++now.minutes;

    if ( now.minutes == 60 ) { // następna godzina
        now.minutes = 0;
        ++now.hour;

        if ( now.hour == 24 ) // północ
            now.hour = 0;
        }
    }

    return now;
}
```

Program 8.6. **Wyniki**

```
Czas to 11:59:59 ...sekundę później to 12:00:00
Czas to 12:00:00 ...sekundę później to 12:00:01
Czas to 01:29:59 ...sekundę później to 01:30:00
Czas to 23:59:59 ...sekundę później to 00:00:00
Czas to 19:12:27 ...sekundę później to 19:12:28
```

Tablica struktur to potężna i bardzo ważna konstrukcja języka C. Przed przejściem dalej czytelnik powinien być przekonany, że dobrze to pojęcie sobie przyswoił.

Struktury zawierające inne struktury

Język C zapewnia niesamowitą wprost elastyczność w definiowaniu struktur. Na przykład można zdefiniować strukturę, której niektóre pola znowu są strukturami, albo strukturę, której pola są tablicami.

Widzieliśmy, jak można logicznie zgrupować miesiąc, dzień i rok w strukturę `date`. Widzieliśmy też, jak zgrupować godzinę, minuty i sekundy w strukturę `time`. W niektórych zastosowaniach konieczne może być logiczne zgrupowanie w jedno daty i czasu. Na przykład może być nam potrzebna lista zdarzeń, które wystąpiły danego dnia, o danej godzinie.

Z wcześniejszych uwag wynika, że możemy wygodnie połączyć ze sobą datę i czas. Wystarczy w języku C zdefiniować nową strukturę, na przykład `dateAndTime`, zawierającą dwa pola: na datę i na czas.

```
struct dateAndTime
{
    struct date  sdate;
    struct time  stime;
};
```

Pierwsze pole naszej struktury jest typu `struct date`, nazywa się ono `sdate`. Drugie pole jest typu `struct time` i nazywa się `stime`. Definicja struktury `dateAndTime` wymaga, aby struktury `date` i `time` zostały zdefiniowane wcześniej, by kompilator je znał w chwili analizowania definicji `dateAndTime`.

Można teraz definiować zmienne typu `struct dateAndTime`:

```
struct dateAndTime  event;
```

Aby odwołać się do struktury `date` w zmiennej `event`, używamy zwykłej składni:

```
event.sdate
```

Możemy zatem wywołać funkcję `dateUpdate`, przekazując tę datę jako parametr, i przypisać wynik funkcji do tej samej struktury:

```
event.sdate = dateUpdate (event.sdate);
```

To samo można zrobić ze strukturą `time` znajdującą się w strukturze `dateAndTime`:

```
event.stime = timeUpdate (event.stime);
```

Aby odwołać się do poszczególnych pól ze struktur wewnętrznych, dołączamy na koniec kolejną kropkę i nazwę pola:

```
event.sdate.month = 10;
```

Taka instrukcja ustawia pole `month` struktury `date` znajdującej się w strukturze `event` na październik. Instrukcja:

```
++event.stime.seconds;
```

dodaje jeden do pola `seconds` w strukturze `time`.

Zmienna `event` może być inicjalizowana następująco:

```
struct dateAndTime event =
        { { 1, 2, 2014 }, { 3, 30, 0 } };
```

W ten sposób ustawiamy datę w zmiennej `event` na 1 lutego 2014 roku, a czas na 3:30:00.

Oczywiście możemy użyć w inicjalizacji nazw pól:

```
struct dateAndTime event =
        { { .day = 1, .month = 2, .year = 2015 },
          { .hour = 3, .minutes = 30, .seconds = 0 }
        };
```

Można też oczywiście utworzyć tablicę struktur `dateAndTime`:

```
struct dateAndTime events[100];
```

Tablica `events` zawiera 100 elementów typu `struct dateAndTime`. Do czwartej wartości typu `dateAndTime` z tej tablicy możemy odwołać się jak zwykle przez `events[3]`, a *i*-tą datę z tablicy możemy przekazać do funkcji `dateUpdate`, korzystając z wywołania:

```
events[i].sdate = dateUpdate (events[i].sdate);
```

Aby ustawić pierwszy czas w tablicy na południe, możemy użyć kilku przypisań:

```
events[0].stime.hour    = 12;
events[0].stime.minutes = 0;
events[0].stime.seconds = 0;
```

Struktury zawierające tablice

Jak sugeruje powyższy śródtytuł, można definiować struktury zawierające tablice. Jednym z najczęstszych zastosowań takiej konstrukcji jest umieszczanie w strukturze tablic znakowych. Załóżmy na przykład, że chcemy zdefiniować strukturę `month` mającą jako pola liczbę dni miesiąca oraz 3-znakowy skrót nazwy miesiąca. Oto odpowiednia definicja:

```
struct month
{
    int   numberOfDays;
    char  name[3];
};
```

W ten sposób zdefiniowaliśmy strukturę `month`, która ma pole całkowitoliczbowe `numberOfDays` oraz pole znakowe `name`. Pole `name` tak naprawdę jest trzyznakową tablicą. Zmienne typu `struct month` definiujemy normalnie:

```
struct month  aMonth;
```

Aby ustawić zmienną `aMonth` tak, by opisywała styczeń, możemy zrobić następujące przypisania:

```
aMonth.numberOfDays = 31;
aMonth.name[0] = 'S';
aMonth.name[1] = 't';
aMonth.name[2] = 'y';
```

Taką samą inicjalizację można zrobić instrukcją

```
struct month  aMonth = { 31, { 'S', 't', 'y' } };
```

Możemy pójść jeszcze krok dalej i ustawić 12 struktur reprezentujących miesiące w tablicę opisującą miesiące całego roku:

```
struct month months[12];
```

Program 8.7 pokazuje użycie tablicy `months`. Jego zadaniem jest po prostu zainicjowanie wartości w tablicy i wyświetlenie tych wartości.

Program 8.7. **Ilustracja użycia struktur i tablic**

```
// Program ilustrujący użycie struktur i tablic

#include <stdio.h>

int main (void)
{
    int  i;

    struct month
    {
        int   numberOfDays;
        char  name[3];
    };

    const struct month  months[12] =
     { { 31, {'S', 't', 'y'} }, { 28, {'L', 'u', 't'} },
       { 31, {'M', 'a', 'r'} }, { 30, {'K', 'w', 'i'} },
       { 31, {'M', 'a', 'j'} }, { 30, {'C', 'z', 'e'} },
       { 31, {'L', 'i', 'p'} }, { 31, {'S', 'i', 'e'} },
       { 30, {'W', 'r', 'z'} }, { 31, {'P', 'a', 'ź'} },
       { 30, {'L', 'i', 's'} }, { 31, {'G', 'r', 'u'} } };

    printf ("Mies.    Liczba dni\n");
    printf ("-----    --------------\n");

    for ( i = 0;  i < 12;  ++i )
        printf (" %c%c%c           %i\n",
            months[i].name[0], months[i].name[1],
```

```
            months[i].name[2], months[i].numberOfDays);

    return 0;
}
```

Program 8.7. **Wyniki**

```
Mies.    Liczba dni
-----    --------------
 Sty          31
 Lut          28
 Mar          31
 Kwi          30
 Maj          31
 Cze          30
 Lip          31
 Sie          31
 Wrz          30
 Paź          31
 Lis          30
 Gru          31
```

Niektórym czytelnikom łatwiej może być zrozumieć działanie programu na podstawie analizy rysunku 8.3.

Jak na tym rysunku widać, wyrażenie:

```
months[0]
```

odnosi się do *całej* struktury `month` z pierwszej pozycji tablicy `months`. Typem tego wyrażenia jest `struct month`. Wobec tego, przekazując `months[0]` do funkcji jako parametr, musimy mieć gwarancję, że odpowiedni parametr formalny funkcji też jest typu `struct month`.

Dalej, wyrażenie:

```
months[0].numberOfDays
```

dotyczy pola `numberOfDays` struktury `month` znajdującej się w `months[0]`. Typem tego wyrażenia jest `int`. Wyrażenie:

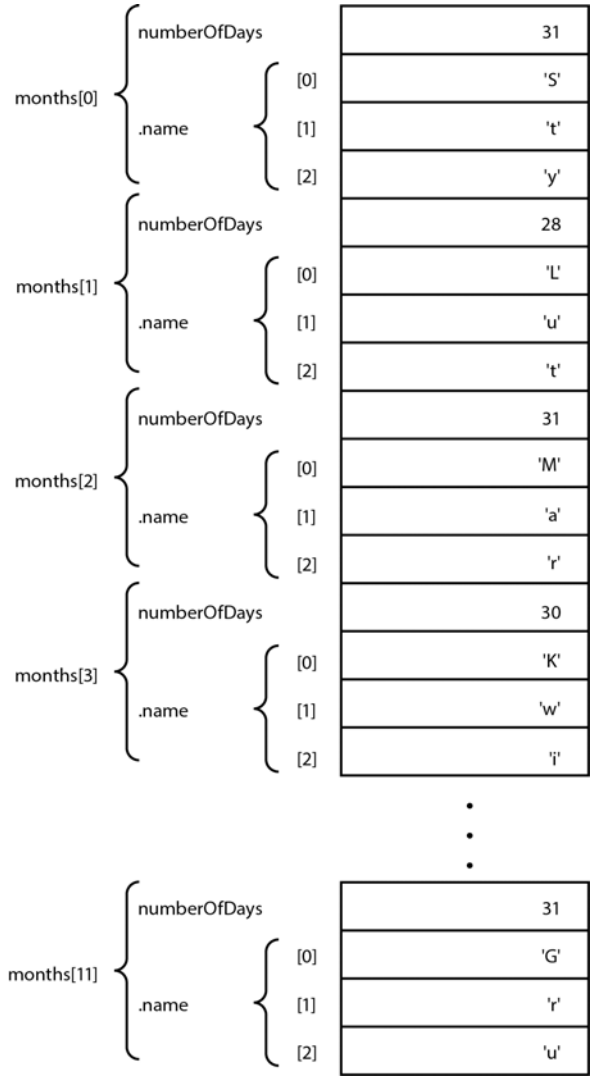
```
months[0].name
```

dotyczy tablicy trzyznakowej `name` w strukturze `month` w tablicy `months[0]`. Jeśli wyrażenie takie przekażemy jako parametr funkcji, odpowiedni parametr formalny tej funkcji musi być tablicą typu `char`.

W końcu wyrażenie:

```
months[0].name[0]
```

dotyczy pierwszego znaku tablicy `name` znajdującej się w `months[0]` (czyli znaku `'S'`).



Rysunek 8.3. Tablica months

Wersje struktur

Wiemy już, jak elastycznymi konstrukcjami są struktury. Można zadeklarować zmienną typu strukturalnego już w chwili definiowania odpowiedniej struktury. Wystarczy podać nazwę lub nazwy zmiennych przed średnikiem kończącym definicję struktury. Na przykład instrukcja:

```
struct date
{
    int  day;
    int  month;
    int  year;
} todaysDate, purchaseDate;
```

definiuje strukturę `date` *oraz* od razu definiuje zmienne `todaysDate` i `purchaseDate` typu `struct date`. Można tym zmiennym normalnie przypisywać wartości inicjujące. Instrukcja:

```
struct date
{
    int  day;
    int  month;
    int  year;
} todaysDate = { 11, 1, 2015 };
```

definiuje strukturę `date`, zmienną `todaysDate` z wartością początkową 11 stycznia 2015 roku.

Jeśli wszystkie zmienne danego typu strukturalnego są definiowane w chwili definicji samej struktury, można nazwę struktury całkowicie pominąć:

```
struct
{
    int  day;
    int  month;
    int  year;
}  dates[100];
```

Taka instrukcja definiuje zmienną `dates` będącą 100-elementową tablicą. Każdy element struktury ma trzy pola całkowitoliczbowe: `month`, `day` i `year`. Nie podaliśmy nazwy struktury, więc aby ponownie zdefiniować nową zmienną takiego typu strukturalnego, trzeba ponownie zdefiniować samą strukturę.

Widzieliśmy, jak struktury pozwalają grupować dane. Widzieliśmy też, jak łatwo buduje się tablice struktur i jak się ich używa w połączeniu z funkcjami. W następnym rozdziale dowiemy się, jak używa się tablic znakowych nazywanych też łańcuchami znakowymi. Przed przejściem dalej należy wykonać poniższe ćwiczenia.

Ćwiczenia

1. Przepisz i uruchom siedem programów pokazanych w tym rozdziale. Porównaj uzyskane wyniki z wynikami pokazanymi w tekście.
2. W niektórych zastosowaniach, szczególnie w aplikacjach finansowych, trzeba wyliczać, ile czasu upłynęło między dwiema datami. Na przykład liczba dni między 2 lipca 2015 roku a 16 lipca 2015 roku to 14. Ile jednak dni upłynęło między 8 sierpnia 2014 a 22 lutym 2015 roku? Tego typu obliczenia wymagają nieco pracy.

Na szczęście istnieje wzór na wyliczanie liczby dni między dwiema datami. Wylicza się wartość *N* dla każdej z obu dat, następnie bierze się różnicę tych *N*. Samo *N* liczy się następująco:

```
N = 1461 · f(rok, miesiąc) / 4 + 153 · g(miesiąc) / 5 + dzień
```

gdzie:

```
f(rok, miesiąc =    rok − 1          jeśli miesiąc <= 2
                    rok              w przeciwnym wypadku

g(miesiąc) =    miesiąc + 13     jeśli miesiąc <= 2
                miesiąc + 1      w przeciwnym wypadku
```

Zastosujmy podany wzór na przykład do wyliczenia liczby dni między 8 sierpnia 2004 roku a 22 lutym 2005 roku. Wyliczamy najpierw wartości N_1 i N_2 dla obu dat:

```
N₁   = 1461 · f(2004, 8) / 4 + 153 · g(8) / 5 + 3
     = (1461 · 2004) / 4 + (153 · 9) / 5 + 3
     = 2 927 844 / 4 + 1 377 / 5 + 3
     = 731 961 + 275 + 3
     = 732 239

N₂   = 1461 · f(2005, 2) / 4 + 153 · g(2) / 5 + 21
     = (1461 · 2004) / 4 + (153 · 15) / 5 + 21
     = 2 927 844 / 4 + 2 295 / 5 + 21
     = 731 961 + 459 + 21
     = 732 441

Odstęp w dniach = N₂ − N₁
                = 732 441 − 732 239
                = 202
```

Wobec tego odstęp między podanymi datami wynosi 202 dni. Powyższy wzór sprawdza się dla dat późniejszych niż 1 marca 1900 roku (do *N* trzeba dodać 1, jeśli potrzebna jest nam data między 1 marca 1800 a 28 lutego 1900; trzeba dodać 2, jeśli data mieści się między 1 marca 1700 a 28 lutego 1800).

Napisz program pozwalający użytkownikowi podać dwie daty i następnie wyliczający odstęp między tymi datami w dniach. Postaraj się podzielić program na logicznie spójne funkcje. Na przykład powinna wystąpić funkcja biorąca jako parametr strukturę `date` i zwracająca wartość *N* wyliczoną dla tej struktury. Funkcja ta będzie wywołana dwukrotnie, raz dla każdej daty. Liczbę dni policzymy jako różnicę tych dwóch wywołań.

3. Napisz funkcję `elapsed_time` mającą jako parametry dwie struktury `time`, zwracającą strukturę `time` opisującą czas, jaki upłynął między przekazanymi strukturami (w godzinach, minutach i miesiącach). Wywołanie:

   ```
   elapsed_time (time1, time2)
   ```

   gdzie `time1` ma wartość 3:45:15, a `time2` wartość 9:44:03, powinno zwrócić strukturę `time` zawierającą 5 godzin, 58 minut i 48 sekund. Należy uważać w przypadku przekroczenia północy.

4. Jeśli weźmiemy wartość *N* obliczoną jak w ćwiczeniu 2., odejmiemy 621 049 i wynik potraktujemy modulo 7, otrzymamy liczbę od 0 do 6 odpowiadającą dniowi tygodnia (od niedzieli do soboty). Na przykład dla 8 sierpnia 2004 roku wartość *N* wynosi 732 239. Różnica 732 239–621 049 daje 111 190, natomiast 111 190 % 7 daje 2, czyli tego dnia był wtorek.

   Używając funkcji z poprzedniego ćwiczenia, napisz program pokazujący dzień tygodnia odpowiadający przekazanej dacie. Niech program wyświetla polskie nazwy dni tygodnia, na przykład „poniedziałek”.

5. Napisz funkcję `clockKeeper`, której parametrem będzie struktura `dateAndTime` zdefiniowana w tym rozdziale. Funkcja powinna wywołać funkcję `timeUpdate` i jeśli czas dojdzie do północy, powinna wywołać funkcję `dateUpdate` przestawiająca datę na następny dzień. Niech funkcja zwraca zaktualizowaną strukturę `dateAndTime`.

6. Zamień funkcję `dateUpdate` z programu 8.4 na jej wersję zmodyfikowaną tak, aby korzystała z literałów złożonych opisanych w tym rozdziale. Uruchom program, aby sprawdzić poprawność działania jego nowej wersji.

9

Łańcuchy znakowe

Teraz możemy dokładniej zająć się łańcuchami znakowymi. Do najważniejszych zadań programów należy przetwarzanie informacji, a liczby w najprzeróżniejszych formach to tylko połowa z nich. Drugą połowę stanowią słowa, znaki oraz ich kombinacje z liczbami. Choć w języku C brak łańcuchowego typu danych w takiej postaci jak w innych językach, wiesz już, że brak ten można uzupełnić przez wykorzystanie połączenia typu `char` z tablicami. Ponadto z danymi łańcuchowymi można pracować przy użyciu niektórych funkcji dostępnych w bibliotece oraz napisanych własnoręcznie. W tym rozdziale opisane są następujące zagadnienia:

- tablice znaków,
- używanie tablic znaków o zmiennej wielkości,
- stosowanie znaków cytowanych,
- dodawanie tablic znaków do struktur,
- przetwarzanie łańcuchów jak danych.

Rozszerzenie wiadomości o łańcuchach

Trochę powiedzieliśmy o nich w rozdziale 2., w którym pisaliśmy swój pierwszy program. W instrukcji:

```
printf ("Programowanie w C to niezła zabawa.\n");
```

argument przekazywany funkcji `printf`:

```
"Programowanie w C to niezła zabawa.\n"
```

jest właśnie łańcuchem znakowym. Łańcuch jest ograniczony cudzysłowami; może zawierać dowolną kombinację liter, cyfr i znaków specjalnych poza cudzysłowami. Jak jednak zaraz zobaczymy, cudzysłowy też można stosować, tylko trzeba to robić w specjalny sposób.

Omawiając typ danych `char`, zaznaczyliśmy, że zmienne tego typu mogą zawierać wyłącznie *pojedyncze* znaki. Aby takiej zmiennej przypisać znak, podajemy go w parze apostrofów. Wobec tego przypisanie:

```
znakPlusa = '+';
```

powoduje przypisanie zmiennej `znakPlusa` znaku `'+'`. Wiemy już, że ważne jest, czy używamy cudzysłowu czy pary apostrofów (zwanych też pojedynczym cudzysłowem); jeśli `znakPlusa` zadeklarowaliśmy jako zmienną typu `char`, instrukcja:

```
znakPlusa = "+";
```

będzie niepoprawna.

Trzeba pamiętać, że w języku C pojedyncze i podwójne cudzysłowy oznaczają stałe różnego typu.

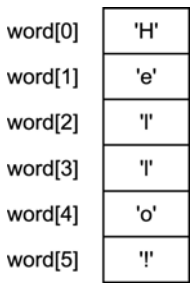
Tablice znaków

Jeśli chcemy posługiwać się zmiennymi mającymi więcej niż jeden znak[1], sięgamy po tablice znaków.

W programie 7.6 tablicę znakową o nazwie `word` zdefiniowaliśmy następująco:

```
char word [] = { 'H', 'e', 'l', 'l', 'o', '!' };
```

Pamiętajmy, że jeśli nie podamy jawnie wielkości tablicy, kompilator C automatycznie obliczy tę liczbę na podstawie wyrażenia inicjalizującego; wobec tego powyższa instrukcja zarezerwuje miejsce dokładnie na sześć znaków, tak jak na rysunku 9.1.



Rysunek 9.1. Położenie tablicy word w pamięci komputera

Aby wyświetlić zawartość tablicy `word`, przechodzimy po jej wszystkich elementach i pokazujemy je, korzystając z formantu `%c`.

---

[1] Przypomnijmy, że do zapisu tak zwanych „szerokich znaków" można użyć typu `wchar_t`; typ ten służy do zapisu pojedynczych znaków z międzynarodowego zestawu. W tej chwili mówimy o zapisywaniu ciągów składających się z wielu znaków.

W ten sposób możemy utworzyć sobie zestaw funkcji przydatnych do operowania na łańcuchach znakowych. Typowe działania wykonywane na łańcuchach znakowych to łączenie dwóch łańcuchów w całość (konkatenacja), kopiowanie jednego łańcucha do innego, pobieranie części łańcucha znakowego (podłańcucha) i porównywanie dwóch łańcuchów (czyli sprawdzanie, czy łańcuchy te zawierają takie same znaki). Weźmy pod uwagę pierwszą z opisanych operacji — konkatenację — i napiszmy funkcję, która ją zrealizuje. Naszą funkcję `concat` możemy zdefiniować następująco:

```
concat (result, str1, n1, str2, n2);
```

gdzie `str1` i `str2` to dwie łączone tablice znakowe, `n1` i `n2` to odpowiednio liczba ich znaków. W ten sposób funkcja nasza jest dostatecznie elastyczna, aby łączyć dwie tablice znakowe dowolnej długości. Parametr `result` to tablica znakowa, w której ma być umieszczony wynik połączenia `str1` i `str2` — wszystko zobaczymy w programie 9.1.

Program 9.1. **Funkcja łącząca dwie tablice znakowe**

```
// Funkcja łącząca dwie tablice znakowe

#include <stdio.h>

void concat (char result[], const char str1[], int n1,
                  const char str2[], int n2)
{
    int i, j;

    // kopiowanie str1 do wyniku, result

    for ( i = 0;  i < n1;  ++i )
        result[i] = str1[i];

    // kopiowanie str2 do wyniku, result

    for ( j = 0;  j < n2;  ++j )
        result[n1 + j] = str2[j];
}

int main (void)
{
    void concat (char result[], const char str1[], int n1,
                      const char str2[], int n2);
    const char  s1[3] = { 'T', 'o', ' ' };
    const char  s2[7] = { 'd', 'z', 'i', 'a', 'ł', 'a', '.' };
    char  s3[10];
    int   i;

    concat (s3, s1, 3, s2, 7);

    for ( i = 0;  i < 10;  ++i )
        printf ("%c", s3[i]);

    printf ("\n");

    return 0;
}
```

Program 9.1. **Wyniki**

```
To działa.
```

Pierwsza pętla `for` w funkcji `concat` kopiuje znaki z tablicy `str1` do tablicy `result`. Pętla ta jest wykonywana `n1` razy, czyli tyle razy, ile znaków jest w tablicy `str1`.

Druga pętla `for` kopiuje łańcuch `str2` do tablicy `result`. Łańcuch `str1` miał `n1` znaków, więc kopiowanie zaczyna się od znaku `result[n1]`, czyli miejsca tuż za ostatnim znakiem z `str1`. Kiedy ta pętla skończy swoje działanie, tablica `result` zawiera `n1+n2` znaków tworzących łańcuch `str1` z doczepionym łańcuchem `str2`.

W procedurze `main` zdefiniowano dwie tablice typu `const — character` `s1` i `s2`. Pierwsza jest inicjalizowana znakami `'T'`, `'o'` i `' '`. Ostatni znak to spacja — jak najbardziej poprawna stała znakowa. Druga tablica jest inicjalizowana znakami `'d'`, `'z'`, `'i'`, `'a'`, `'ł'`, `'a'` i `'.'`. Trzecia tablica znakowa to `s3`; jest dostatecznie duża, aby pomieścić wszystkie znaki z `s1` i `s2`, czyli razem 10 znaków. Nie jest ona deklarowana ze słowem kluczowym `const`, gdyż jej zawartość będzie się zmieniała.

Instrukcja:

```
concat (s3, s1, 3, s2, 7)
```

wywołuje funkcję `concat` w celu połączenia tablic `s1` i `s2`, wynik ma być w tablicy `s3`. Parametry 3 i 7 — przekazywane do funkcji — określają, ile znaków mają odpowiednio `s1` i `s2`.

Kiedy funkcja `concat` skończy swoje działanie i następuje powrót do `main`, przygotowywana jest pętla `for` wyświetlająca wyniki. Widzimy 10 elementów tablicy `s3` — tak więc funkcja `concat` działa poprawnie. Jednak w pokazanym programie zakładamy, że pierwszy parametr tej funkcji — tablica wynikowa — jest dostatecznie duży, aby pomieścić wynik łączenia dwóch pozostałych przekazanych tablic. Niespełnienie tego warunku może spowodować nieprzewidywalne skutki.

Łańcuchy znakowe zmiennej długości

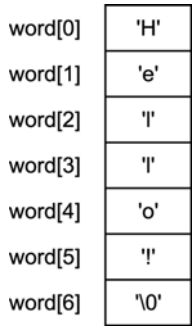
Możemy też, korzystając z rozwiązań podobnych jak w funkcji `concat`, tworzyć inne funkcje obsługujące tablice znakowe. Możemy zatem przygotować zestaw procedur, których parametrami będą tablice znakowe i liczby znaków w tych tablicach. Niestety, już po krótkim czasie pracy z takimi funkcjami zliczanie znaków w poszczególnych tablicach staje się coraz bardziej irytujące — szczególnie jeśli długości tych tablic co chwilę się zmieniają. Przydałaby się metoda pozwalająca obsługiwać tablice znakowe bez konieczności troszczenia się o to, ile mają znaków.

Metoda taka istnieje i polega na umieszczeniu specjalnego znaku na końcu każdego łańcucha. W ten sposób sama funkcja może sprawdzić, czy doszła do końca takiego łańcucha. Tworząc funkcje działające na takich łańcuchach, możemy już nie podawać liczby znaków w poszczególnych łańcuchach.

W języku C znakiem specjalnym oznaczającym koniec łańcucha jest tak zwany znak *null*, zapisywany jako `'\0'`. Wobec tego instrukcja:

```
const char word [] = { 'H', 'e', 'l', 'l', 'o', '!', '\0' };
```

tworzy tablicę znakową `word` mającą *siedem* znaków; ostatni z nich to znak null (przypomnijmy jeszcze, że w języku C odwrotny ukośnik — \ — pełni rolę szczególną i nie jest liczony jako odrębny znak; wobec tego w C `'\0'` to pojedynczy znak). Tablicę `word` pokazano na rysunku 9.2.



Rysunek 9.2. Tablica word z końcowym znakiem null

Aby pokazać, jak używa się łańcuchów znakowych *o zmiennej długości*, napiszemy funkcję zliczającą znaki w łańcuchu — będzie to program 9.2. Funkcję tę nazwiemy `stringLength`, jej parametrem będzie łańcuch zakończony znakiem null. Funkcja określa liczbę znaków tablicy i zwraca tę liczbę. Niech liczba znaków nie zawiera końcowego znaku null. Wobec tego wywołanie:

```
stringLength (characterString)
```

w sytuacji, kiedy wcześniej `characterString` zdefiniowano następująco:

```
char characterString[] = { 'k', 'o', 't', '\0' };
```

powinno zwrócić liczbę 3.

Program 9.2. **Zliczanie znaków łańcucha**

```
// Funkcja zliczająca znaki w łańcuchu

#include <stdio.h>

int stringLength (const char string[])
{
    int count = 0;

    while ( string[count] != '\0' )
        ++count;
```

```
    return count;
}

int main (void)
{
    int   stringLength (const char string[]);
    const char word1[] = { 'a', 's', 't', 'e', 'r', '\0' };
    const char word2[] = { 'a', 't', '\0' };
    const char word3[] = { 'a', 'w', 'e', '\0' };

    printf ("%i   %i   %i\n", stringLength (word1),
            stringLength (word2), stringLength (word3));

    return 0;
}
```

Program 9.2. **Wyniki**

```
5   2   3
```

Parametr funkcji `stringLength` jest tablicą stałą (słowo kluczowe `const`), gdyż w tej tablicy nie dokonujemy żadnych zmian, jedynie zliczamy jej znaki.

W funkcji `stringLength` mamy zmienną `count`, której początkową wartością jest 0. Program wchodzi w pętlę `while` i przechodzi po całej tablicy `string` aż do osiągnięcia znaku null. Kiedy funkcja dochodzi do tego znaku, co oznacza dojście do końca łańcucha, następuje wyjście z pętli `while`, zwracana jest wartość zmiennej `count`. Wartość ta oznacza liczbę znaków w łańcuchu, przy czym kończący znak null nie jest już liczony. Warto by przeanalizować działanie programu na małej tablicy znakowej, aby sprawdzić, jak `count` uzyskuje w pętli prawidłową wartość.

W procedurze `main` mamy trzy tablice znakowe — `word1`, `word2` i `word3`. Wywołanie funkcji `printf` pokazuje wyniki wywołania `stringLength` dla każdej z tych tablic.

Inicjalizowanie i pokazywanie tablic znakowych

Czas teraz wrócić do funkcji `concat` z programu 9.1 i napisać ją tak, aby działała na łańcuchach o zmiennej długości. Oczywiście funkcja ta musi zostać nieco zmieniona, gdyż nie potrzebujemy już liczby znaków w łańcuchach. Teraz funkcja będzie miała trzy parametry — dwie łączone tablice znakowe i tablicę na wynik.

Zanim zajmiemy się programem, powiemy jeszcze o dwóch miłych cechach języka C związanych z przetwarzaniem łańcuchów znakowych.

Na początek chodzi o inicjalizowanie takich tablic. Tablice znakowe możemy inicjalizować, podając po prostu stały łańcuch znakowy, a nie kolejne znaki. Na przykład instrukcja:

```
char word[] = { "Hello!" };
```

może zostać użyta do wstawienia do tablicy znakowej `word` znaków `'H'`, `'e'`, `'l'`, `'l'`, `'o'`, `'!'` i `'\0'`. Inicjalizując w ten sposób tablicę znakową, możemy też pominąć nawiasy klamrowe, więc instrukcja:

```
char word[] = "Hello!";
```

jest jak najzupełniej poprawna. Obie powyższe instrukcje są równoważne zapisowi:

```
char word[] = { 'H', 'e', 'l', 'l', 'o', '!', '\0' };
```

Jeśli podajemy wielkość tablicy, musimy zagwarantować, że będzie dostatecznie duża, aby zmieścił się także końcowy znak null. Zatem w przypadku deklaracji:

```
char word[7] = { "Hello!" };
```

kompilator ma w zmiennej `word` dość miejsca także na końcowy znak null. Jednak instrukcja:

```
char word[6] = { "Hello!" };
```

już tego warunku nie spełnia — końcowy znak null nie mieści się w tablicy. Co gorsza, kompilator nie informuje o tym problemie.

Wszystkie łańcuchy znakowe występujące w programach C są automatycznie kończone znakiem null. Dzięki temu funkcje typu `printf` mogą znaleźć koniec łańcucha. Zatem w wywołaniu:

```
printf ("Programowanie w C to niezła zabawa.\n");
```

po znaku nowego wiersza automatycznie dokładany jest znak null. W ten sposób do funkcji `printf` przekazywana jest informacja, gdzie kończy się łańcuch formatujący.

Inna ciekawa cecha języka C dotyczy wyświetlania łańcuchów znakowych. Do wyświetlenia tablicy znakowej zakończonej znakiem null można w łańcuchu formatującym funkcji `printf` użyć formantu `%s`. Jeśli zatem `word` jest tablicą znaków zakończoną znakiem null, wywołanie funkcji `printf`:

```
printf ("%s\n", word);
```

powoduje wyświetlenie całej zawartości tablicy `word`. Funkcja `printf` zakłada, że jeśli natknie się na formant `%s`, odpowiedni parametr będzie łańcuchem znakowym zakończonym znakiem null.

Opisane tu dwie cechy języka C możemy teraz włączyć do treści funkcji `main` programu 9.3, pokazującego przerobioną wersję funkcji `concat`. Już nie przekazujemy tej funkcji długości łańcuchów, więc musi ona odnaleźć końce łańcuchów za pomocą znaków null. Poza tym, kiedy łańcuch `str1` jest kopiowany do tablicy `result`, musimy dopilnować, aby wraz z nim *nie* był kopiowany końcowy znak null, gdyż to spowodowałoby zakończenie w tym miejscu łańcucha `result`. Musimy natomiast wstawić do `result` znak null *po* skopiowaniu do tej tablicy łańcucha `str2`.

Program 9.3. **Łączenie łańcuchów znakowych**

```
#include <stdio.h>

int main (void)
```

```
{
    void concat (char result[], const char str1[], const char str2[]);
    const char  s1[] = { "To " };
    const char  s2[] = { "działa." };
    char s3[20];

    concat (s3, s1, s2);

    printf ("%s\n", s3);

    return 0;
}

// Funkcja łącząca dwa łańcuchy znakowe

void concat (char result[], const char str1[], const char str2[])
{
    int i, j;

    // kopiowanie str1 do result

    for ( i = 0;  str1[i] != '\0';  ++i )
        result[i] = str1[i];

    // kopiowanie str2 do result

    for ( j = 0;  str2[j] != '\0';  ++j )
        result[i + j] = str2[j];

    // Zakańczanie połączonych łańcuchów znakiem null

    result [i + j] = '\0';
}
```

Program 9.3. **Wyniki**

```
To działa.
```

W pierwszej pętli `for` funkcji `concat` znaki z łańcucha `str1` są kopiowanie do tablicy `result` aż do napotkania znaku null. Pętla `for` kończy się zaraz po znalezieniu znaku null, znak ten nie zostanie skopiowany do wyniku.

W drugiej pętli znaki z łańcucha `str2` są kopiowane do tablicy `result` zaraz za ostatnim znakiem ze `str1`. Pętla ta wykorzystuje fakt, że w zmiennej `i` została wartość równa liczbie znaków ze `str1` (poza znakiem null), kiedy poprzednia pętla skończyła swoje działanie. Wobec tego przypisanie:

```
result[i + j] = str2[j];
```

kopiuje znaki ze `str2` od razu we właściwe miejsca tablicy `result`.

Kiedy druga pętla kończy swoje działanie, funkcja `concat` dostawia na koniec całego łańcucha wynikowego znak null. Warto przeanalizować tę funkcję dokładniej, aby dobrze zrozumieć przeznaczenie zmiennych `i` i `j`. Wiele błędów programistycznych dotyczących łańcuchów znakowych wynika z użycia indeksu przesuniętego o 1 w którąś stronę.

Pamiętajmy, że używamy indeksu 0, aby odwołać się do pierwszego znaku z tablicy. Dodatkowo, jeśli tablica znakowa ma `n` znaków (bez znaku null), to `string[n – 1]` dotyczy ostatniego znaku z łańcucha innego niż null, z kolei `string[n]` oznacza ostatni znak, znak null. Poza tym łańcuch znakowy trzeba definiować tak, aby zmieściło się w nim `n + 1` znaków — znak null też musi się zmieścić.

W procedurze `main` zdefiniowaliśmy dwie tablice znakowe — `s1` i `s2` — ich wartości ustawiliśmy, korzystając z nowej, opisanej wcześniej techniki inicjalizacji. Tablica `s3` ma zawierać 20 znaków, dzięki czemu mamy dość miejsca na połączenie naszych łańcuchów i nie musimy sobie zaprzątać głowy dokładnym zliczaniem znaków.

Dalej wywoływana jest funkcja `concat`, jej parametry to łańcuchy `s1`, `s2` i `s3`. Wynik jest umieszczany w `s3`, po czym pokazywany przy użyciu formantu `%s`. Choć według definicji `s3` zawiera 20 znaków, funkcja `printf` wyświetli tylko znaki sprzed znaku null — niezależnie od tego, ile ich będzie.

Porównywanie dwóch łańcuchów znakowych

Jeśli chcemy sprawdzić, czy dwa łańcuchy są sobie równe, nie możemy użyć zwykłego porównania typu:

```
if ( łańcuch1 == łańcuch2 )
    ...
```

gdyż operator równości wolno stosować tylko do zmiennych typów prostych, takich jak `float`, `int` czy `char`, a nie do typów bardziej złożonych, takich jak struktury czy tablice.

Aby sprawdzić, czy dwa łańcuchy są sobie równe, trzeba jawnie porównywać znaki parami. Jeśli dojdziemy do końca obu łańcuchów jednocześnie i wszystkie porównywane znaki okażą się identyczne, łańcuchy są sobie równe. W przeciwnym wypadku łańcuchy są różne.

Przydałaby się funkcja mogąca porównywać dwa łańcuchy znakowe, tak jak w programie 9.4. Możemy wywołać funkcję `equalStrings`, przekazując jej jako parametry oba porównywane łańcuchy. Interesuje nas tylko, czy te łańcuchy są sobie równe, więc funkcja może zwracać wartość logiczną `true`, jeśli łańcuchy są równe, i `false`, kiedy jest inaczej. W ten sposób łatwo będzie używać tej funkcji w takich warunkach:

```
if ( equalStrings (string1, string2) )
    ...
```

Program 9.4. **Porównywanie dwóch łańcuchów znakowych**

```
// Funkcja sprawdzająca, czy dwa łańcuchy są sobie równe

#include <stdio.h>
#include <stdbool.h>

bool equalStrings (const char s1[], const char s2[])
{
    int  i = 0;
    bool areEqual;
```

```
    while ( s1[i] == s2[i] &&
                s1[i] != '\0'  &&  s2[i] != '\0' )
        ++i;

    if ( s1[i] == '\0'  &&  s2[i] == '\0' )
        areEqual = true;
    else
        areEqual = false;

    return areEqual;
}

int main (void)
{
    bool  equalStrings (const char s1[], const char s2[]);
    const char  stra[] = "łańcuchy — porównywanie";
    const char  strb[] = "łańcuchy";

    printf ("%i\n", equalStrings (stra, strb));
    printf ("%i\n", equalStrings (stra, stra));
    printf ("%i\n", equalStrings (strb, "łańcuchy"));

    return 0;
}
```

Program 9.4. **Wyniki**

```
0
1
1
```

Funkcja `equalStrings` korzysta z pętli `while` do przejścia po łańcuchach znakowych `s1` i `s2`. Pętla jest wykonywana tak długo, aż oba łańcuchy znakowe okażą się równe (`s1[i] == s2[i]`) i któryś z łańcuchów się skończy (`s1[i] != '\0' && s2[i] != '\0'`). Zmienna `i` używana jako indeks w obu tablicach jest zwiększana w każdym kroku pętli `while`.

Instrukcja `if` wykonywana po pętli `while` sprawdza, czy jednocześnie osiągnięto koniec łańcuchów `s1` i `s2`. Można by użyć też instrukcji:

```
if ( s1[i] == s2[i] )
    ...
```

i osiągnąć dokładnie taki sam wynik. Jeśli jesteśmy na końcu obu łańcuchów, muszą one być identyczne; wtedy `areEqual` jest ustawiane na `true`, a następnie zwracane do miejsca wywołania. W przeciwnym razie łańcuchy się różnią, `areEqual` jest ustawiane na `false` i wtedy zwracane.

W funkcji `main` przygotowujemy dwie tablice znakowe — `stra` i `strb`; obie mają wartości początkowe. Pierwsze wywołanie `equalStrings` przekazuje jako parametry obie te tablice. Łańcuchy nie są sobie równe, więc nasza funkcja prawidłowo zwraca wartość `false`, czyli 0.

W drugim wywołaniu funkcji `equalStrings` dwukrotnie przekazujemy łańcuch `stra`. Funkcja prawidłowo zwraca `true`, gdyż łańcuchy są sobie równe.

Ciekawsze jest trzecie wywołanie `equalStrings`. Jak widać, można przekazać stałą znakową do funkcji oczekującej na tablicę znaków. W rozdziale 10., poświęconym wskaźnikom, zobaczymy, jak to właściwie działa. Funkcja `equalStrings` porównuje łańcuch znakowy z `strb` z literałem `"łańcuchy"`, zwraca `true`, co sugeruje, że ciągi są sobie równe.

Wprowadzanie łańcuchów znakowych

Nieobca jest nam myśl, że kiedy chcemy wyświetlić łańcuch znakowy, używamy formantu `%s`. A co zrobić, kiedy zechcemy wczytać łańcuch znakowy? Istnieje kilka możliwości. Pierwsza z nich to funkcja `scanf`, której możemy użyć z formantem `%s` — wczytamy łańcuch aż do spacji, tabulatora lub znaku końca wiersza, którykolwiek z nich pojawi się pierwszy. Wobec tego instrukcje:

```
char string[81];

scanf ("%s", string);
```

powodują wczytanie łańcucha znaków i zapisanie go do tablicy znakowej `string`. W przeciwieństwie do wcześniejszych wywołań funkcji `scanf`, tym razem *nie* umieszczamy znaku `&` przed nazwą tablicy (dlaczego, powiemy w rozdziale 10.).

Jeśli wywołamy `scanf` jak powyżej i następnie wpiszemy:

```
Gravity
```

funkcja `scanf` odczyta łańcuch `"Gravity"` i zapisze go w tablicy `string`. Jeśli jednak wpisalibyśmy tekst:

```
iTunes playlist
```

w tablicy `string` zapisany zostałby jedynie łańcuch `"iTunes"`, gdyż za nim znajduje się spacja, która dla `scanf` oznacza koniec łańcucha znakowego. Przy następnym wywołaniu `scanf` z takimi samymi parametrami w tablicy `string` zapisany zostałby łańcuch `"playlist"`, gdyż funkcja `scanf` zawsze wczytuje dane, zaczynając od znaku wczytanego ostatnio.

Funkcja `scanf` automatycznie kończy wczytany łańcuch znakiem null. Wobec tego wykonanie powyższych instrukcji i podanie następującego wiersza:

```
abcdefghijklmnopqrstuvwxyz
```

spowoduje wczytanie wszystkich małych liter alfabetu łacińskiego w pierwszych 26 komórkach tablicy `string`, natomiast wartością `string[26]` będzie znak null.

Jeśli `s1`, `s2` i `s3` zostaną zdefiniowane jako tablice znakowe odpowiedniej wielkości, wykonanie instrukcji:

```
scanf ("%s%s%s", s1, s2, s3);
```

i podanie tekstu:

```
taki mały komputerek
```

spowoduje przypisanie zmiennej s1 łańcucha "taki", zmiennej s2 — łańcucha "mały", a zmiennej s3 — łańcucha "komputerek". Gdybyśmy natomiast podali tekst:

```
rozszerzenie systemu
```

funkcja scanf przypisze zmiennej s1 łańcuch "rozszerzenie", zmiennej s2 — łańcuch "systemu", a następnie, jako że nie ma już żadnych danych do wczytania, będzie czekała na podanie dalszych danych.

W programie 9.5 używamy funkcji scanf do odczytu trzech łańcuchów znakowych.

Program 9.5. **Wczytanie łańcuchów znakowych za pomocą funkcji scanf**

```
// Program pokazujący użycie formantu %s funkcji scanf

#include <stdio.h>

int main (void)
{
    char s1[81], s2[81], s3[81];

    printf ("Podaj tekst:\n");

    scanf ("%s%s%s", s1, s2, s3);

    printf ("\ns1 = %s\ns2 = %s\ns3 = %s\n", s1, s2, s3);
    return 0;
}
```

Program 9.5. **Wyniki**

```
Podaj tekst:
systemowa magistrala
rozszerzeń

s1 = systemowa
s2 = magistrala
s3 = rozszerzeń
```

W powyższym przykładzie funkcja scanf jest wywoływana po to, aby wczytać trzy łańcuchy znakowe — s1, s2 i s3. Pierwszy wiersz wpisanego tekstu zawiera tylko dwa łańcuchy (czyli ciągi znaków aż do spacji), program czeka na podanie dalszego tekstu. Kiedy tekst ten zostanie podany, wywołanie funkcji printf pozwala sprawdzić, że w zmiennych s1, s2 i s3 zostały prawidłowo wpisane łańcuchy "systemowa", "magistrala" i "rozszerzeń".

Jeśli w powyższym programie podamy więcej niż 80 znaków niezawierających żadnej spacji, tabulatora ani znaku nowego wiersza, scanf przepełni jedną z tablic znakowych. W przypadku formantu %s funkcja ta po prostu będzie dalej czytała znaki i zapisywała je w zmiennej, póki nie napotka jednego z wymienionych tu ograniczników.

Jeśli w formancie po znaku % podamy liczbę, liczba ta określi maksymalną liczbę znaków, które chcemy wczytać. Jeśli zatem w programie 9.5 użyjemy wywołania:

```
scanf ("%80s%80s%80s", s1, s2, s3);
```

funkcja scanf nie wczyta więcej niż 80 znaków do żadnej ze zmiennych s1, s2 ani s3 (nadal potrzebujemy jednak dodatkowego miejsce na znak null na końcu wczytywanego łańcucha; dlatego właśnie używamy formantów %80s, a nie %81s).

Wczytanie pojedynczego znaku

Biblioteka standardowa zawiera kilka funkcji służących do czytania oraz pisania pojedynczych znaków i całych łańcuchów znakowych. Funkcja getchar pozwala odczytać z terminala pojedynczy znak. Wielokrotne wywołania tej funkcji odczytują kolejne znaki. Po natknięciu się na znak końca wiersza funkcja getchar zwraca znak '\n'. Jeśli zatem podamy znaki abc, kolejne wywołania funkcji getchar zwracać będą — najpierw 'a', potem 'b', potem 'c', a czwarte wywołanie getchar zwróci '\n'. Piąte wywołanie spowoduje, że program będzie czekał na podanie dalszych danych.

Można by się zastanawiać, po co tu funkcja getchar, skoro pojedyncze znaki możemy wczytywać za pomocą wywołania funkcji scanf z formantem %c. Użycie scanf w takiej sytuacji jest jak najbardziej poprawne, ale funkcja getchar jest wygodniejsza, ponieważ jej zadaniem jest wyłącznie odczyt pojedynczych znaków, wobec czego może nie mieć parametrów. Funkcja ta zwraca pojedynczy znak, który może być przypisany zmiennej lub wykorzystany inaczej, tak jak tego akurat wymaga program.

W wielu aplikacjach przetwarzających tekst trzeba wczytać cały wiersz tekstu. Zwykle taki wiersz zapisywany w określonym miejscu nazywamy „buforem", tam następuje dalsze przetwarzanie. Do wczytania bufora nie można użyć funkcji scanf z formantem %s, gdyż wczytywanie się skończy po natknięciu się na pierwszą spację w danych.

W bibliotece standardowej istnieje jednak gotowa funkcja; jest to funkcja gets; jej jedynym zastosowaniem jest właśnie wczytywanie pojedynczego wiersza tekstu. Program 9.6 jest ciekawym ćwiczeniem — pokazuje, jak można za pomocą funkcji getchar zaimplementować funkcję podobną do gets, tutaj nazwaną readLine. Funkcja ta pobiera pojedynczy parametr — tablicę znakową, w której ma być umieszczony odczytany wiersz tekstu. W tablicy tej sytuowane są wszystkie wczytane znaki poza końcowym znakiem nowego wiersza.

Program 9.6. **Wczytywanie wiersza danych**

```
#include <stdio.h>

int main (void)
{
    int   i;
    char  line[81];
    void readLine (char buffer[]);

    for ( i = 0; i < 3; ++i )
    {
        readLine (line);
        printf ("%s\n\n", line);
    }

    return 0;
```

```
}

// Funkcja wczytująca wiersz tekstu

void readLine (char buffer[])
{
    char  character;
    int   i = 0;

    do
    {
        character = getchar ();
        buffer[i] = character;
        ++i;
    }
    while ( character != '\n' );

    buffer[i – 1] = '\0';
}
```

Program 9.6. **Wyniki**

```
To jest przykładowy wiersz tekstu.
To jest przykładowy wiersz tekstu.

abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz

procedury z biblioteki roboczej
procedury z biblioteki roboczej
```

Pętla `do` w funkcji `readLine` służy do „zbierania" wiersza danych w tablicy znakowej `buffer`. Każdy znak zwracany przez funkcję `getchar` jest zapisywany w następnej komórce tablicy. Po natknięciu się na znak nowego wiersza, co oznacza koniec wpisywania danych, następuje wyjście z pętli. W tablicy umieszczany jest znak null zastępujący znak nowego wiersza podany jako ostatni. Prawidłowy indeks miejsca w tablicy to `i`–1, gdyż w czasie wychodzenia z pętli nastąpiło dodatkowe przesunięcie indeksu o jedno miejsce.

Funkcja `main` zawiera tablicę znakową `line`, która ma dość miejsca na maksymalnie 81 znaków. Dzięki temu możliwe jest zapisanie tam całej linii danych uzupełnionej znakiem null (ze względów historycznych przyjmuje się, że „standardowy terminal" pozwala zapisać w jednej linii 80 znaków). Jednak nawet w oknach, które mają tylko 80 znaków w wierszu nadal występuje ryzyko przepełnienia — wystarczy, że użytkownik będzie wpisywał dane w kolejnym wierszu, nie wciskając klawisza *Enter*. Warto by wzbogacić funkcję `readLine` o dodatkowy parametr określający długość bufora po to, by nigdy jej nie przekroczyć.

Innym dobrym pomysłem na usprawnienie tego programu jest poprawienie interaktywności przez wyświetlenie w wierszu poleceń informacji o tym, czego program potrzebuje. Wystarczy dodać poniższy wiersz kodu do pętli `do...while` w funkcji `readline()`, aby użytkownik wiedział, co ma robić:

```
printf("Wpisz linijkę tekstu o długości do 80 znaków i naciśnij klawisz Enter: \n");
```

W napisie takim można też określić oczekiwany format danych wejściowych, na przykład symbol dolara przed kwotą pieniężną albo dwukropek między godziną i minutą. Tego typu wskazówki to kolejny sposób na zminimalizowanie ilości błędów przy wprowadzaniu danych.

Dalej program wchodzi w pętlę `for`, która po prostu trzykrotnie wywołuje funkcję `readLine`. W każdym wywołaniu odczytywany jest jeden wiersz tekstu. Wiersze te są po prostu pokazywane na ekranie, aby sprawdzić poprawność działania naszej funkcji. Po pokazaniu trzeciej porcji danych program 9.6 kończy swoje działanie.

Nasz następny program, czyli 9.7, to praktyczna aplikacja do przetwarzania tekstu; program ten zlicza słowa w podanym fragmencie tekstu. Utworzymy funkcję `countWords` biorącą jako parametr łańcuch znakowy i zwracającą liczbę słów w tym łańcuchu. Dla uproszczenia założymy, że słowo to jedna lub więcej liter. Funkcja przeszukuje łańcuch znakowy aż do pierwszej litery, następnie wszystkie znaki aż do znaku niebędącego literą traktuje jako wchodzące w skład tego samego słowa. Później funkcja znowu odszukuje następną literę, która wyznacza początek następnego słowa.

Program 9.7. **Zliczanie słów**

```
// Funkcja sprawdzająca, czy dany znak jest literą

#include <stdio.h>
#include <stdbool.h>

bool alphabetic (const char c)
{
    if ( (c >= 'a'  &&  c <= 'z') || (c >= 'A'  &&  c <= 'Z') )
        return true;
    else
        return false;
}

/* Funkcja zliczająca słowa w łańcuchu. */

int countWords (const char string[])
{
    int   i, wordCount = 0;
    bool  lookingForWord = true, alphabetic (const char c);

    for ( i = 0;  string[i] != '\0';  ++i )
        if ( alphabetic(string[i]) )
        {
            if ( lookingForWord )
            {
                ++wordCount;
                lookingForWord = false;
            }
        }
        else
            lookingForWord = true;

    return wordCount;
}
```

```
int main (void)
{
    const char  text1[] = "No, to tak.";
    const char  text2[] = "No i jeszcze raz... znowu.";
    int   countWords (const char string[]);

    printf ("%s — słowa = %i\n", text1, countWords (text1));
    printf ("%s — słowa = %i\n", text2, countWords (text2));

    return 0;
}
```

Program 9.7. **Wyniki**

```
No, to tak. — słowa = 3
No i jeszcze raz... znowu. — słowa = 5
```

Funkcja `alphabetic` jest bardzo prosta — sprawdza tylko, czy przekazany jej znak jest małą lub wielką literą. Jeśli tak, funkcja zwraca `true`, co oznacza, że przekazany znak jest literą. W przeciwnym wypadku funkcja zwraca `false`.

Funkcja `countWords` już nie jest taka prosta. Liczba całkowita `i` służy jako indeks przechodzący po wszystkich znakach łańcucha. Zmienna `lookingForWord` służy jako flaga wskazująca, czy aktualnie szukamy początku nowego słowa. Kiedy działanie tej funkcji się zaczyna, oczywiście szukamy początku nowego słowa, więc flagę tę ustawiamy na `true`. Zmienna lokalna `wordCount` jest po prostu licznikiem słów w łańcuchu znakowym.

Dla każdego kolejnego znaku z łańcucha wywołujemy funkcję `alphabetic`, aby sprawdzić, czy dany znak jest literą. Jeśli jest to litera, kontrolujemy wartość flagi `lookingForWord`, aby sprawdzić, czy akurat szukamy nowego słowa. Jeśli tak, wartość zmiennej `wordCount` jest zwiększana o 1, a flaga `lookingForWord` jest ustawiana na `false`, co oznacza, że już nie szukamy początku nowego słowa.

Jeśli znak jest literą i flaga `lookingForWord` ma wartość `false`, akurat jesteśmy *w środku* słowa. Wtedy pętla `for` kontynuuje swoje działanie na następnym znaku łańcucha.

Jeśli znak nie jest literą, to albo osiągnęliśmy koniec słowa, albo nadal szukamy nowego słowa — wtedy flaga `lookingForWord` jest ustawiana na `true` (nawet jeśli wcześniej miała już wartość `true`).

Kiedy sprawdzone zostaną wszystkie znaki łańcucha, funkcja zwraca wartość `wordCount`, podając tym samym liczbę wyrazów w łańcuchu znakowym.

Warto by pokazać tabelę z kolejnymi wartościami różnych zmiennych z funkcji `countWords`, aby sprawdzić, jak działa algorytm. Jest to właśnie tabela 9.1, gdzie pokazano pierwsze wywołanie funkcji `countWords` z powyższego przykładu. Pierwszy wiersz tabeli 9.1 pokazuje wartość początkową zmiennych `wordCount` oraz `lookingForWord` przed wejściem do pętli `for`. W kolejnych wierszach mamy wartości tych samych zmiennych w trakcie działania pętli `for`. Zatem drugi wiersz tabeli pokazuje, jak po pierwszym przejściu pętli wartość `wordCount` została ustawiona na 1, a flaga `lookingForWord` na `false` (czyli 0). W ostatnim wierszu tabeli pokazano ostateczne wartości zmiennych po przeanalizowaniu całego łańcucha znakowego. Warto poświęcić nieco czasu na analizę

tej tabeli i przyjrzeć się, jak mają się pokazane zmienne do logiki funkcji `countWords`. Wtedy algorytm użyty do zliczania słów nie będzie miał już przed nami żadnych tajemnic.

Tabela 9.1. Działanie funkcji countWords

i	string[i]	wordCount	lookingForWord
		0	true
0	'N'	1	false
1	'o'	1	false
2	','	1	true
3	' '	1	true
4	't'	2	false
5	'o'	2	false
6	' '	2	true
7	't'	3	false
8	'a'	3	false
9	'k'	3	false
10	'.'	3	true
11	'\0'	3	true

Łańcuch pusty

Teraz zajmijmy się bardziej praktycznym użyciem funkcji `countWords`. Tym razem skorzystamy ze zdefiniowanej wcześniej funkcji `readLine`, aby użytkownik mógł wprowadzić wiele wierszy tekstu. Następnie program zliczy słowa w tekście i pokaże uzyskany wynik.

Aby program był bardziej elastyczny, nie ograniczamy ani nie ustalamy ilości wierszy tekstu, które będą podane. Wobec tego musimy jakoś umożliwić użytkownikowi poinformowanie programu, kiedy wprowadzanie tekstu się skończyło. Jednym ze sposobów jest powtórne wciśnięcie klawisza *Enter* po wpisaniu ostatniego wiersza. Kiedy funkcja `readLine` wczyta taki wiersz, od razu natknie się na znak nowego wiersza, więc zamiast tego znaku wpisze null — będzie to pierwszy i jedyny znak w całym buforze. Program może ten szczególny przypadek zidentyfikować i stwierdzić, że podano ostatni wiersz tekstu.

Łańcuch znakowy, który poza znakiem null nie zawiera żadnych znaków, ma w języku C specjalną nazwę — jest to *łańcuch pusty*. Nazwa ta doskonale oddaje jego charakter. Funkcja `stringLength` jako długość łańcucha pustego zwróci 0. Funkcja `concat` też, zgodnie z oczekiwaniami, będzie łączyła „nic" z dowolnym innym łańcuchem. Nawet funkcja `equalStrings` zadziała prawidłowo, jeśli którykolwiek lub oba podane jej łańcuchy będą puste (przy czym dwa łańcuchy puste są uważane za równe sobie).

Zawsze trzeba pamiętać, że łańcuch pusty zawiera jeden znak — znak null.

Czasami potrzebne jest zapisanie pustego łańcucha znakowego w programie. Służy do tego literał składający się z pary cudzysłowów. Wobec tego instrukcja:

```
char  buffer[100] = "";
```

definiuje tablicę znakową `buffer` i jako jej wartość ustawia łańcuch pusty. Zauważmy, że łańcuch `""` to *co innego* niż łańcuch `" "`, gdyż ten drugi zawiera jedną spację (osoby mające jeszcze jakieś wątpliwości mogą na podanych łańcuchach wywołać funkcję `equalStrings`).

Program 9.8 wykorzystuje funkcje `readLine`, `alphabetic` i `countWords` z poprzednich programów.

Program 9.8. **Zliczanie słów we fragmencie tekstu**

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

bool alphabetic (const char c)
{
    if ( (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') )
        return true;
    else
        return false;
}

void readLine (char buffer[])
{
    char character;
    int i = 0;

    do
    {
        character = getchar ();
        buffer[i] = character;
        ++i;
    }
    while ( character != '\n' );

    buffer[i - 1] = '\0';
}

int countWords (const char string[])
{
    int i, wordCount = 0;
    bool lookingForWord = true, alphabetic (const char c);

    for ( i = 0; string[i] != '\0'; ++i )
        if ( alphabetic(string[i]) )
        {
            if ( lookingForWord )
            {
                ++wordCount;
                lookingForWord = false;
            }
        }
        else
```

```
            lookingForWord = true;

    return wordCount;
}


int main (void)
{
    char  text[81];
    int   totalWords = 0;
    void  readLine (char  buffer[]);
    bool  endOfText = false;

    printf ("Podaj tekst.\n");
    printf ("Kiedy skończysz, wciśnij 'ENTER'.\n\n");

    while ( ! endOfText )
    {
        readLine (text);

        if ( text[0] == '\0' )
            endOfText = true;
        else
            totalWords += countWords (text);
    }

    printf ("\nW podanym tekście wystąpiło %i słów.\n", totalWords);

    return 0;
}
```

Program 9.8. **Wyniki**

```
Podaj tekst.
Kiedy skończysz, wciśnij 'ENTER'.

Marta spojrzała ku sufitowi, gdzie góra lazanii majaczyła niczym cętkowany grzbiet górski.
Zaraz została ukoronowana tiarą z krążków sera ricotta z sosem pomidorowym. Kawałki
wołowiny wykwitły na jej czole niczym piegi. Po drugim uderzeniu jej koronacja na królową
kuchni dobiegła końca.
Enter
W podanym tekście wystąpiło 41 słów.
```

Wiersz, w którym jest napisane tylko *Enter*, oznacza dodatkowe wciśnięcie klawisza *Enter*.

Zmienna `endOfText` służy jako flaga wskazująca koniec tekstu. Pętla `while` działa tak długo, jak długo ta flaga ma wartość `false`. W tej pętli program wywołuje funkcję `readLine` wczytującą kolejne wiersze tekstu. Instrukcja `if` sprawdza, czy tekst w tablicy `text` nie jest pustym ciągiem (czyli czy nie wciśnięto samego *Enter*). Jeśli jest pustym ciągiem, bufor zawiera łańcuch pusty, więc flaga `endOfText` jest ustawiana na `true`.

Jeśli bufor zawiera jakiś tekst, wywoływana jest funkcja `countWords` zliczająca słowa w tablicy `text`. Wartość zwrócona przez tę funkcję jest dodawana do zmiennej `totalWords`, która zawiera liczbę słów stopniowo zwiększaną od początku analizy tekstu.

Kiedy pętla `while` kończy swoje działanie, program pokazuje wartość `totalWords` wraz ze stosownym komunikatem.

Wydawać się może, że powyższy program nie ogranicza zanadto koniecznego nakładu pracy, gdyż tekst trzeba i tak wpisać z klawiatury. Jednak — jak zobaczymy w rozdziale 15. — tego samego programu można użyć do zliczania słów występujących na przykład w pliku na dysku. Wobec tego autor przygotowujący rękopis na komputerze może nasz program docenić, gdyż pozwoli zliczyć słowa w tym rękopisie (jeśli sam rękopis jest w zwykłym pliku tekstowym, a nie w formacie procesora tekstu takiego jak Microsoft Word).

Cytowanie znaków

Jak wspominaliśmy już wcześniej, w języku C odwrotny ukośnik ma specjalne znaczenie — służy nie tylko do zapisu znaku nowego wiersza i znaku null. Tak jak odwrotny ukośnik i litera `n` tworzą znak nowego wiersza, tak samo inne kombinacje odwrotnego ukośnika i innych liter realizują pewne funkcje specjalne. Znaki te, często określane jako *znaki cytowane*, zestawiono w tabeli 9.2.

Tabela 9.2. Znaki cytowane

Znak	Nazwa
`\a`	Sygnał dźwiękowy
`\b`	Backspace
`\f`	Następna strona (także Form Feed lub FF)
`\n`	Znak nowego wiersza
`\r`	Powrót karetki
`\t`	Tabulator
`\v`	Tabulator w pionie
`\\`	Odwrotny ukośnik
`\"`	Cudzysłów
`\'`	Apostrof
`\?`	Pytajnik
`\nnn`	Znak o kodzie ósemkowym *nnn*
`\unnnn`	Znak Unicode o podanym kodzie
`\Unnnnnnnn`	Znak Unicode o podanym kodzie
`\xnn`	Znak o kodzie szesnastkowym *nn*

Siedem znaków z początku tabeli 9.2 w większości urządzeń wyświetlających realizuje jakieś funkcje. Znak *sygnału dźwiękowego* — `\a` — powoduje zwykle uruchomienie wbudowanego głośniczka (nazywanego „bzyczkiem”). Wobec tego instrukcja `printf`:

```
printf ("\aSYSTEM ZOSTANIE ZAMKNIĘTY W CIĄGU 5 MINUT!!\n");
```

spowoduje wydanie dźwięku, a następnie wyświetlenie podanego komunikatu.

Włączenie znaku *backspace* — `'\b'` — do łańcucha znakowego powoduje, że terminal w danym miejscu cofa się o jeden znak, jeśli jest to możliwe. Analogicznie wywołanie:

```
printf ("%i\t%i\t%i\n", a, b, c);
```

spowoduje wyświetlenie wartości zmiennej `a`, przejście do następnego położenia tabulatora (zwykle tabulatory są wstawiane w co ósmej kolumnie), pokazanie wartości zmiennej `b`, przejście do następnego tabulatora i pokazanie tam wartości zmiennej `c`. Tabulator taki jest szczególnie przydatny do wyrównania danych ułożonych kolumnami.

Aby do łańcucha znakowego wstawić odwrotny ukośnik, trzeba podać dwa znaki takiego ukośnika; wywołanie funkcji `printf`:

```
printf ("\\t to tabulator poziomy.\n");
```

spowoduje wyświetlenie tekstu:

```
\t to tabulator poziomy.
```

Zauważmy, że na początku łańcucha mamy dwa odwrotne ukośniki — `\\` — dlatego nie jest wyświetlany tabulator.

Aby do łańcucha znakowego wstawić cudzysłów, trzeba go poprzedzić odwrotnym ukośnikiem. Wobec tego wywołanie instrukcji `printf`:

```
printf ("Powiedział \"Witam\"."\n);
```

spowoduje wyświetlenie napisu:

```
Powiedział "Witam".
```

Aby w zmiennej znakowej umieścić apostrof, trzeba go poprzedzić odwrotnym ukośnikiem. Jeśli `c` jest zmienną typu `char`, instrukcja:

```
c = '\'';
```

przypisze zmiennej `c` znak apostrofu.

Znak odwrotnego ukośnika, za którym jest pytajnik — `?` — oznacza właśnie pytajnik. Jest to czasami niezbędne, kiedy mamy do czynienia z *trójznakami* z zestawów znaków spoza ASCII. Więcej szczegółów na ten temat podajemy w dodatku A.

Ostatnie cztery pozycje tabeli 9.2 pozwalają na wstawienie *dowolnego* znaku do łańcucha znakowego. W zapisie `'\`*nnn*`'` napis *nnn* oznacza od jednej do trzech cyfr *ósemkowych*. W zapisie `'\x`*nn*`'` napis *nn* to z kolei liczba szesnastkowa. Liczby takie oznaczają wewnętrzny *kod* znaku. Dzięki temu do łańcuchów można dołączać znaki, które nie są dostępne bezpośrednio z klawiatury. Jeśli na przykład do łańcucha chcemy zapisać znak *Escape*, mający ósemkowo kod 33, możemy napisać `\033` lub `\x1b`.

Znak null — `'\0'` — to specyficzny znak specjalny, który omawialiśmy nieco wcześniej. Jest to znak o wartości 0; ponieważ po prostu *jest* zerem, często programiści używają go w instrukcjach warunkowych i warunkach pętli, kiedy w grą wchodzi obsługa łańcuchów o zmiennej długości. Na przykład pętla zliczająca znaki w łańcuchu znakowym występująca w funkcji `stringLength` z programu 9.2 może zostać zapisana w formie:

```
while ( string[count] )
    ++count;
```

Wartość `string[count]` nie jest zerem tak długo, jak długo nie dojdziemy do znaku null, kiedy to następuje wyjście z pętli `while`.

Jeszcze raz trzeba tu odnotować, że każdy z omawianych tutaj znaków jest pojedynczym znakiem — łańcuch `"\033\"Hello\"\n"` ma tylko dziewięć znaków (nie licząc końcowego znaku null): znak `'\033'`, cudzysłów zapisywany jako `'\"'`, pięć znaków słowa `Hello`, znowu cudzysłów i znak nowego wiersza. Jeśli przekażemy powyższy łańcuch funkcji `stringLength`, faktycznie otrzymamy w odpowiedzi długość równą 9 (znowu pomijany jest końcowy znak null).

Znaki Unicode tworzymy, podając po `\u` cztery cyfry szesnastkowe kody lub po `\U` osiem cyfr szesnastkowych. W ten sposób zapisuje się znaki z rozszerzonych zestawów znaków, czyli znaków wymagających więcej niż standardowych ośmiu bitów. W ten sposób można podawać 16-bitowe i 32-bitowe znaki oraz wstawiać takie znaki w łańcuchach znakowych oraz w stałych. Więcej informacji na ten temat zamieszczono w dodatku A.

Jeszcze o stałych łańcuchach

Jeśli na samym końcu łańcucha znakowego umieścimy odwrotny ukośnik i zaraz za nim będzie znak nowego wiersza, kompilator C będzie mógł znak nowego wiersza zignorować. Taka technika kontynuowania łańcuchów w kolejnych wierszach używana jest głównie do zapisywania długich łańcuchów znakowych. W rozdziale 12. poświęconym preprocesorowi zobaczymy, jak tej samej techniki można użyć do kontynuowania definicji *makr*.

Jeśli zabraknie znaku kontynuacji wiersza, to podczas próby inicjalizacji łańcucha znakowego wartością wielowierszową kompilator C wygeneruje komunikat o błędzie. Na przykład błędna jest instrukcja:

```
char letters[] =
    { "abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ" };
```

Jeśli natomiast na końcu każdego wiersza będącego kontynuacją poprzedniego umieścimy odwrotny ukośnik, stała łańcuchowa będzie zapisana w wielu wierszach:

```
char letters[] =
    { "abcdefghijklmnopqrstuvwxyz\
ABCDEFGHIJKLMNOPQRSTUVWXYZ" };
```

Ciąg dalszy kontynuowanego wiersza bezwzględnie musi zaczynać się od *początku* następnego wiersza — w przeciwnym razie spacje zostaną też wstawione do łańcucha. Powyższa instrukcja spowoduje zatem zdefiniowanie tablicy `letters` i zainicjalizowanie jej łańcuchem:

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Innym sposobem dzielenia długich łańcuchów na części jest dzielenie ich na sąsiadujące łańcuchy. Tak przystające do siebie stałe łańcuchy, oddzielone co najwyżej białymi znakami (spacjami, tabulatorami czy znakami nowego wiersza), kompilator automatycznie połączy w całość. Jeśli wobec tego napiszemy:

```
"raz"  "dwa"  "trzy"
```

będzie to równoważne zapisowi:

```
"razdwatrzy"
```

Zatem tablicę `letters` zawierającą początkowo wszystkie litery alfabetu możemy zapisać jako:

```
char letters[] =
    { "abcdefghijklmnopqrstuvwxyz"
      "ABCDEFGHIJKLMNOPQRSTUVWXYZ" };
```

W końcu w przypadku poniższych trzech wywołań funkcji `printf`:

```
printf ("Programowanie w języku C to niezła zabawa\n");
printf ("Programowanie"  " w języku C to niezła zabawa\n");
printf ("Programowanie"  " w języku C"  " to niezła zabawa\n");
```

każda funkcja `printf` otrzymuje *pojedynczy* parametr, gdyż w drugim i trzecim wywołaniu kompilator połączy stałe łańcuchowe w całość.

Łańcuchy znakowe, struktury i tablice

Wszystkie podstawowe elementy języka C można składać na różne sposoby w potężne konstrukcje programistyczne. W rozdziale 8. poświęconym strukturom widzieliśmy na przykład, jak łatwo można zdefiniować tablicę struktur. Program 9.9 przedstawia dalsze przykłady tablic struktur oraz łańcuchów znakowych o zmiennej długości.

Załóżmy, że chcemy napisać program działający jak słownik. Wtedy, kiedy tylko natkniemy się na słowo, którego nie rozumiemy, możemy to słowo wpisać w programie; program odszuka je w słowniku i poda definicję.

Podczas rozmyślania nad takim programem z początku przychodzi do głowy pytanie, jak zapisać słowa i ich definicje. Słowo i jego definicja są logicznie ze sobą powiązane, więc naturalnym rozwiązaniem jest użycie struktur. Możemy na przykład utworzyć strukturę `entry`, która będzie zawierała słowo i jego definicję:

```
struct entry
{
```

```
    char  word[15];
    char  definition[50];
};
```

W powyższej definicji mamy dość miejsca na 14-literowe słowa (pamiętajmy, że cały czas mamy do czynienia z łańcuchami znakowymi o zmiennej długości, więc musimy mieć miejsce na znak null) oraz na 49-znakowe definicje tych słów. Oto przykład zmiennej typu `struct entry`, zainicjalizowanej słowem „pulpa" i jego definicją:

```
struct entry word1 = { "pulpa", "bezkształtna masa" };
```

Chcemy, aby nasz słownik miał wiele haseł, więc logiczne będzie zdefiniowanie tablicy zawierającej struktury `entry`:

```
struct entry dictionary[100];
```

Teraz możemy zapisać 100 haseł. Oczywiście, jeśli chcemy opracować słownik języka polskiego, jest to o wiele, wiele za mało — sensowny słownik polszczyzny powinien zawierać przynajmniej 80 000 haseł. Wtedy jednak program musiałby być całkiem inaczej napisany; przede wszystkim konieczne byłoby przechowywanie słownika na dysku, a nie w pamięci.

Kiedy mamy już podstawową strukturę słownika, zastanówmy się, jak powinien wyglądać. Większość słowników jest uporządkowana alfabetycznie, więc zrobimy podobnie. Załóżmy, że ułatwi to czytanie słownika. Później znajdziemy głębsze uzasadnienie dla takiej organizacji haseł.

Czas już pomyśleć o programie. Wygodnie byłoby utworzyć funkcję odszukującą słowa. Jeśli słowo zostanie znalezione, funkcja zwróci jego numer w ramach słownika; w przeciwnym razie funkcja zwróci –1, co będzie oznaczało brak słowa w słowniku. Zatem typowe wywołanie takiej funkcji — nazwijmy ją `lookup` — będzie wyglądało tak:

```
entry = lookup (dictionary, word, entries);
```

W tym wypadku funkcja `lookup` przegląda słownik `dictionary` w poszukiwaniu słowa z łańcucha znakowego `word`. Trzeci parametr — `entries` — to liczba haseł w słowniku. Funkcja po odnalezieniu żądanego słowa zwraca numer odpowiedniego hasła ze słownika. Jeśli słowa nie znajdzie, zwraca –1.

W programie 9.9 funkcja `lookup` w celu porównania szukanego słowa z hasłem ze słownika korzysta z funkcji `equalStrings` z programu 9.4.

Program 9.9. **Użycie programu przeszukującego słownik**

```
// Program realizujący przeszukiwanie słownika

#include <stdio.h>
#include <stdbool.h>

struct entry
{
    char  word[15];
    char  definition[50];
};
```

```
bool equalStrings (const char s1[], const char s2[])
{
    int i = 0;
    bool areEqual;

    while ( s1[i] == s2 [i] &&
                s1[i] != '\0' && s2[i] != '\0' )
        ++i;

    if ( s1[i] == '\0' && s2[i] == '\0' )
        areEqual = true;
    else
        areEqual = false;

    return areEqual;
}
// Funkcja odszukująca słowo w słowniku

int lookup (const struct entry dictionary[], const char search[],
            const int entries)
{
    int  i;
    bool equalStrings (const char s1[], const char s2[]);

    for ( i = 0;  i < entries;  ++i )
        if ( equalStrings (search, dictionary[i].word) )
            return i;

    return -1;
}

int main (void)
{
    const struct entry  dictionary[100] =
      {  { "abakan",    "przestrzenny, monumentalny gobelin" },
         { "abakus",    "starożytne liczydło" },
         { "abazja",    "utrata zdolności chodzenia" },
         { "abduktor",  "mięsień odwodzący kończynę" },
         { "abietyna",  "substancja żywiczna z drzew iglastych" },
         { "abisal",    "najgłębsza strefa mórz, oceanów i jezior" },
         { "abrazja",   "niszczenie wybrzeży przez fale" },
         { "absces",    "inaczej: ropień" },
         { "absynt",    "nalewka spirytusowa na ziołach" },
         { "achterdek", "rufowa część pokładu" } };

    char word[10];
    int  entries = 10;
    int  entry;
    int  lookup (const struct entry dictionary[], const char search[],
                 const int entries);

    printf ("Podaj słowo: ");
    scanf ("%14s", word);
    entry = lookup (dictionary, word, entries);

    if ( entry != -1 )
        printf ("%s\n", dictionary[entry].definition);
```

```
    else
        printf ("Niestety, słowo %s nie występuje w moim słowniku.\n", word);

    return 0;
}
```

Program 9.9. **Wyniki**

```
Podaj słowo: abazja
utrata zdolności chodzenia
```

Program 9.9. **Wyniki (ponowne uruchomienie)**

```
Podaj słowo: abchazja
Niestety, słowo abchazja nie występuje w moim słowniku.
```

Funkcja `lookup` po kolei przegląda wszystkie hasła ze słownika i dla każdego z nich wywołuje funkcję `equalStrings`, aby sprawdzić, czy łańcuch `search` pasuje do pola `word` z danego hasła słownika. Jeśli tak, funkcja zwraca wartość zmiennej `i`, czyli numer hasła ze słownika. W takim wypadku funkcja kończy swoje działanie instrukcją `return`, choć jest w trakcie realizacji pętli `for`.

Jeśli funkcja `lookup` przejrzy wszystkie hasła ze słownika i nie znajdzie dopasowania, instrukcja `return` kończy działanie funkcji i zwraca informację o nieznalezieniu hasła (–1).

Lepsza metoda szukania

Użyta przez nas metoda szukania haseł w słowniku jest nader prosta — przeglądamy kolejne hasła, aż znajdziemy odpowiednie albo wyczerpiemy wszystkie możliwości, dochodząc do końca słownika. W przypadku słowników tak małych, jak w naszym programie, jest to rozwiązanie jak najbardziej na miejscu. Jeśli jednak mamy duży słownik, z setkami, czy nawet tysiącami haseł, rozwiązanie to staje się nieefektywne z uwagi na czas przeglądania kolejno wszystkich pozycji. Czas ten może być istotny, nawet jeśli będzie trwał ułamek sekundy. Przy wybieraniu danych jednym z najważniejszych zagadnień jest kwestia czasu wymaganego na wybieranie. Przeszukiwanie danych jest w informatyce tak powszechne, że badacze wiele pracy poświęcili na tworzenie sprawnych algorytmów wyszukujących (włożono w to bez mała tyle pracy, ile w algorytmy sortujące).

Do utworzenia szybciej działającej funkcji `lookup` możemy wykorzystać fakt, że nasz słownik jest uporządkowany alfabetycznie. Pierwszym, co od razu przychodzi na myśl, jest wyposażenie funkcji `lookup` w „umiejętność” stwierdzenia, że zaszła już za daleko. Jeśli na przykład w słowniku z programu 9.9 szukamy słowa „ablacja”, to po dojściu do słowa „abrazja” wiemy już, że słowa ablacja w słowniku nie ma. Gdyby było, powinno znaleźć się *przed* „abrazją”.

Opisana powyżej strategia optymalizacji pozwoli ograniczyć nieco czas wyszukiwania, ale tylko wtedy, gdy danego słowa *nie* będzie w słowniku. Tak naprawdę interesuje nas ograniczenie czasu poszukiwania w większości sytuacji. Takim algorytmem jest *wyszukiwanie binarne*.

Zasady działania wyszukiwania binarnego są dość proste. Aby zobaczyć, jak działa ten algorytm, wyobraźmy sobie grę w zgadywanie. Jedna osoba wybiera liczbę od 1 do 99, druga próbuje tę liczbą odgadnąć w jak najmniejszej liczbie prób. Po każdej próbie odgadnięcia określamy, czy podano liczbę za małą, za dużą czy trafiono. Po kilku próbach większość zgadujących dochodzi do wniosku, że najlepiej ograniczać zakres możliwości przez dzielenie całego zakresu na równe części. Pierwszą próbą odgadnięcia może być podanie liczby 50. Podpowiedź: „Za mała" oznacza, że powinniśmy podawać liczby z zakresu od 51 do 99. Podpowiedź: „Za duża" oznacza, że powinniśmy wybierać między 1 a 49; tak czy inaczej, liczba możliwych odpowiedzi została ograniczona ze 100 do 49.

Proces dzielenia na części można kontynuować z pozostałymi 49 liczbami. Jeśli pierwsza podpowiedź brzmiała: „Za mała", próbujemy zgadywać liczbę z zakresu od 51 do 99, czyli 75. Cały proces kontynuujemy tak długo, aż zakres zawęzimy do pojedynczej możliwej odpowiedzi. Średnio procedura ta będzie wymagała znacznie mniej prób niż inne metody szukania.

Wiemy już, jak działa wyszukiwanie binarne. Teraz podamy formalny opis tego algorytmu. Szukamy elementu *x* w tablicy *M* zawierającej *n* elementów. W algorytmie zakłada się, że elementy tablicy *M* są uporządkowane rosnąco.

Algorytm wyszukiwania binarnego

Krok 1: Ustawiamy *low* na 0, *high* na *n*–1.

Krok 2: Jeśli *low* > *high*, *x* w *M* nie występuje i działanie algorytmu kończy się.

Krok 3: Ustawiamy *mid* na (*low*+*high*)/2.

Krok 4: Jeśli *M*[*mid*] < *x*, ustawiamy *low* na *mid*+1, przechodzimy do kroku 2.

Krok 5: Jeśli *M*[*mid*] > *x*, ustawiamy *high* na *mid*–1 i przechodzimy do kroku 2.

Krok 6: *M*[*mid*] równe jest *x* i algorytm kończy swoje działanie.

Dzielenie w kroku 3. jest dzieleniem całkowitoliczbowym, więc jeśli *low* jest równe 0, a *high* 49, wartością *mid* jest 24.

Teraz, kiedy mamy już algorytm, możemy napisać nową funkcję `lookup`, która skorzysta z tego algorytmu. Wyszukiwanie binarne musi mieć możliwość sprawdzenia, czy jedna wartość jest mniejsza lub większa od innej, czy też równa; funkcję `equalStrings` zastąpimy funkcją tak porównującą dwa łańcuchy znakowe. Wywołamy funkcję `compareStrings` zwracającą –1, jeśli pierwszy łańcuch jest mniejszy od drugiego, 0 — jeśli łańcuchy są równe i 1 — jeśli pierwszy łańcuch jest większy od drugiego[2]. Wobec tego wywołanie funkcji:

```
compareStrings ("alfa", "alternatywa");
```

zwróci wartość –1, gdyż pierwszy łańcuch jest mniejszy od drugiego (mniejszy, czyli występuje wcześniej w słowniku). Wywołanie:

---

[2] Funkcja `compareStrings` może zachowywać się niezgodnie z oczekiwaniami w przypadku polskich liter. To, czy zachowa się poprawnie, czy nie, zależy od systemu operacyjnego — *przyp. tłum.*

```
compareStrings ("zombi", "yeti");
```

zwróci 1, gdyż wyraz „zombi” jest mniejszy od „yeti”.

W programie 9.10 pokazano naszą nową funkcję — `compareStrings`. Funkcja `lookup` do przeszukiwania słownika wykorzystuje teraz wyszukiwanie binarne. Funkcja `main` jest taka sama jak w programie poprzednim.

Program 9.10. **Zmodyfikowane przeszukiwanie słownika z użyciem wyszukiwania binarnego**

```
// Program realizujący przeszukiwanie słownika

#include <stdio.h>

struct entry
{
    char  word[15];
    char  definition[50];
};

// Funkcja porównująca dwa łańcuchy znakowe

int compareStrings (const char s1[], const char s2[])
{
    int  i = 0, answer;

    while ( s1[i] == s2[i] && s1[i] != '\0'  &&  s2[i] != '\0' )
        ++i;

    if ( s1[i] < s2[i] )
        answer = -1;            /* s1 < s2 */
    else if ( s1[i] == s2[i] )
        answer = 0 ;            /* s1 == s2 */
    else
        answer = 1;             /* s1 > s2 */

    return answer;
}

// Funkcja odszukująca słowo w słowniku

int lookup (const struct entry dictionary[], const char search[],
            const int entries)
{
    int  low = 0;
    int  high = entries - 1;
    int  mid, result;
    int  compareStrings (const char  s1[], const char s2[]);

    while ( low <= high )
    {
        mid = (low + high) / 2;
        result = compareStrings (dictionary[mid].word, search);

        if ( result == -1 )
            low = mid + 1;
        else if ( result == 1 )
```

```
            high = mid - 1;
        else
            return mid;      /* znaleziono */
    }

    return -1;               /* nie znaleziono */
}

int main (void)
{
    const struct entry  dictionary[100] =
      {  { "abakan",    "przestrzenny, monumentalny gobelin" },
         { "abakus",    "starożytne liczydło" },
         { "abazja",    "utrata zdolności chodzenia" },
         { "abduktor",  "mięsień odwodzący kończynę" },
         { "abietyna",  "substancja żywiczna z drzew iglastych" },
         { "abisal",    "najgłębsza strefa mórz, oceanów i jezior" },
         { "abrazja",   "niszczenie wybrzeży przez fale" },
         { "absces",    "inaczej: ropień" },
         { "absynt",    "nalewka spirytusowa na ziołach" },
         { "achterdek", "rufowa część pokładu" } };

    char word[10];
    int  entries = 10;
    int  entry;
    int  lookup (const struct entry dictionary[], const char search[],
                 const int entries);

    printf ("Podaj słowo: ");
    scanf ("%14s", word);
    entry = lookup (dictionary, word, entries);

    if ( entry != -1 )
        printf ("%s\n", dictionary[entry].definition);
    else
        printf ("Niestety, słowo %s nie występuje w moim słowniku.\n", word);

    return 0;
}
```

Program 9.10. **Wyniki**

```
Podaj słowo: absces
inaczej: ropień
```

Program 9.10. **Wyniki (ponowne uruchomienie)**

```
Podaj słowo: abstynent
Niestety, słowo abstynent nie występuje w moim słowniku.
```

Funkcja `compareStrings` jest identyczna z funkcją `equalStrings`. Dopiero wtedy, kiedy następuje wyjście z pętli `while`, funkcja porównuje znaki, które spowodowały zakończenie działania tej pętli, i jeśli `s1[i]` jest mniejszy od `s2[i]`, łańcuch `s1` jest leksykograficznie mniejszy od `s2`; funkcja zwraca –1. Jeśli `s1[i]` jest równe `s2[i]`,

oba łańcuchy są sobie równe i zwracane jest 0. Jeśli nie zachodzi żaden z powyższych warunków, `s1` musi być większe od `s2`, a wtedy zwracana jest wartość 1.

Funkcja `lookup` ma zmienne `low` i `high` typu `int`; początkowo przypisuje im wartości wynikające z zasad działania algorytmu wyszukiwania binarnego. Pętla `while` działa tak długo, aż `low` nie przekroczy wartości `high`. W tej pętli `mid` wyliczane jest jako suma `low` i `high` podzielona przez 2. Następnie wywoływana jest funkcja `compareString`; jako parametry otrzymuje słowo `dictionary[mid]` oraz słowo przez nas szukane. Zwrócona wartość jest przypisywana do zmiennej `result`.

Jeśli funkcja `compareStrings` zwróci wartość –1, czyli kiedy słowo `dictionary[mid].word` będzie mniejsze od `search`, funkcja `lookup` ustawia wartość `low` na `mid+1`. Jeśli `compareStrings` zwróci 1, co znaczy, że `dictionary[mid].search` jest większe od `search`, funkcja `lookup` ustawia `high` na `mid–1`. Jeśli nie zostanie zwrócone ani –1, ani 1, oba łańcuchy muszą być sobie równe, więc `lookup` zwróci wartość zmiennej `mid` — numer hasła szukanego w słowniku.

Jeśli zdarzy się, że `low` przekroczy wartość `high`, słowa nie ma w słowniku. Wtedy funkcja `lookup` zwróci wartość –1.

Operacje na znakach

Zmienne i stałe znakowe są często używane w porównaniach i wyrażeniach arytmetycznych. Aby prawidłowo ich wtedy używać, trzeba zrozumieć, jak obsługuje je kompilator języka C.

Kiedy w wyrażeniu języka C używana jest stała lub zmienna znakowa, automatycznie jest konwertowana (i dalej traktowana) jak wartość całkowitoliczbowa.

W rozdziale 5. widzieliśmy, jak wyrażenie:

```
c >= 'a'  &&  c <= 'z'
```

pozwala sprawdzić, czy zmienna znakowa `c` jest małą literą. Jak wspomniano wcześniej, wyrażenia takiego można użyć w systemach wykorzystujących zestaw znaków ASCII, gdzie małe litery mają kolejne kody i nie występują między nimi żadne inne znaki. Pierwsza część powyższego wyrażenia, porównująca wartość zmiennej ze stałą znakową `'a'`, polega na porównaniu zmiennej `c` z kodem odpowiadającym znakowi `'a'`. W kodowaniu ASCII litera `'a'` ma kod 97, litera `'b'` 98 i tak dalej. Wobec tego wyrażenie `c > 'a'` będzie prawdziwe (będzie miało niezerową wartość), jeśli kod znaku ze zmiennej `c` będzie większy lub równy 97. Jednak w zestawie ASCII warunek ten spełniają nie tylko małe litery, lecz także na przykład nawiasy klamrowe, a zatem zakres musimy dodatkowo ograniczyć, aby wychwycić tylko małe litery. Z tego powodu wartość zmiennej `c` porównujemy jeszcze ze stałą znakową `'z'`, której kod ASCII to 122.

Omówione powyżej porównywanie zmiennej `c` ze znakami `'a'` i `'z'` polega na porównywaniu wartości tej zmiennej z kodami odpowiadającymi literom `'a'` i `'z'`, równie dobrze można by zapisać:

```
c >= 97  &&  c <= 122
```

Jednak preferowany jest zapis pokazany wcześniej, gdyż wtedy nic nie trzeba wiedzieć o tym, jakie kody odpowiadają poszczególnym literom, więc intencje programisty są bardziej czytelne.

Wywołanie funkcji `printf`:

```
printf ("%i\n", c);
```

pozwala pokazać kod odpowiadający znakowi `c`. Jeśli używamy kodowania ASCII, instrukcja:

```
printf ("%i\n", 'a');
```

pokaże wartość 97.

Spróbujmy przewidzieć, jaki wynik dadzą dwie następujące instrukcje:

```
c = 'a' + 1;
printf ("%c\n", c);
```

W kodzie ASCII wartością znaku `'a'` jest 97, więc pierwsza instrukcja powoduje przypisanie zmiennej znakowej `c` liczby 98. Kod ten odpowiada znakowi `'b'`, więc funkcja `printf` właśnie taki znak wyświetli.

Choć dodawanie jedności do stałej znakowej wydaje się mało sensowne, poprzedni przykład pokazuje ważną technikę stosowaną do konwersji znaków od `'0'` do `'9'` na odpowiadające im liczby, od 0 do 9. Przypomnijmy, że znak `'0'` nie jest tym samym co liczba 0, znak `'1'` to co innego niż liczba 1 i tak dalej. Znak `'0'` ma w ASCII kod 48, co możemy zobaczyć, stosując instrukcję:

```
printf ("%i\n", '0');
```

Załóżmy, że zmienna `c` zawiera jeden ze znaków z zakresu od `'0'` do `'9'` i chcemy przekształcić tę wartość na odpowiadającą jej liczbę z zakresu od 0 do 9. Właściwie we wszystkich systemach kodowania znaków cyfrom odpowiadają kolejne kody, więc analogiczny wynik możemy uzyskać także w innych systemach, odejmując od znaków cyfr znak `'0'`. Jeśli zatem `i` jest z definicji zmienną typu `int`, instrukcja:

```
i = c - '0';
```

spowoduje zamianę cyfry ze zmiennej `c` w odpowiadającą jej liczbę. Załóżmy na przykład, że zmienna `c` zawiera znak `'5'`, w ASCII o kodzie 53. Wartością znaku `'0'` w ASCII jest 48, więc powyższe wyrażenie to odejmowanie 48 od 53, zatem zmienna `i` otrzyma wartość 5. Na innych maszynach, które korzystają z innych zestawów znaków, a nie z ASCII, wynik zwykle jest taki sam, choć znakom `'5'` i `'0'` odpowiadają inne liczby.

Opisana technika może zostać rozszerzona na całe łańcuchy zawierające cyfry. Pokazano to w programie 9.11, w którym funkcja `strToInt` przekształca łańcuch cyfr przekazany jej jako parametr na odpowiadającą mu liczbę. Funkcja ta kończy analizowanie ciągu po natknięciu się w nim na pierwszy znak niebędący cyfrą i zwraca uzyskany dotąd wynik. Zakłada się, że wielkość zmiennej typu `int` wystarczy do zapisania dowolnej wartości z łańcucha znakowego.

Program 9.11. **Zamiana łańcucha znakowego na odpowiadającą mu liczbę całkowitą**

```
// Funkcja zamieniająca łańcuch znakowy na liczbę całkowitą

#include <stdio.h>

int strToInt (const char  string[])
{
    int  i, intValue, result = 0;

    for ( i = 0;  string[i] >= '0'  &&  string[i] <= '9';  ++i )
    {
        intValue = string[i] - '0';
        result = result * 10 + intValue;
    }

    return result;
}

int main (void)
{
    int  strToInt (const char  string[]);

    printf ("%i\n", strToInt("245"));
    printf ("%i\n", strToInt("100") + 25);
    printf ("%i\n", strToInt("13x5"));

    return 0;
}
```

Program 9.11. **Wyniki**

```
245
125
13
```

Pętla `for` jest wykonywana tak długo, jak długo znak `string[i]` jest cyfrą. W każdym przejściu pętli znak ten jest konwertowany na odpowiadającą mu liczbę, a następnie dodawany do dotychczasowego wyniku `result` pomnożonego przez 10. Aby zobaczyć funkcjonowanie tego kodu, przyjrzyjmy się dokładniej działaniu pętli, gdy funkcja jako parametr otrzymała łańcuch `"245"`. W pierwszym przejściu pętli `intValue` otrzymuje wartość `string[0]-'0'`. Znak `string[0]` to `'2'`, więc `intValue` otrzymuje wartość 2. W pierwszym przejściu pętli zmienna `result` ma wartość 0, więc mnożenie tego 0 przez 10 da 0; kiedy dodamy do wyniku wartość zmiennej `intValue`, a całość przypiszemy znowu zmiennej `result`, zmienna ta po pierwszym przejściu pętli będzie miała wartość 2.

W drugim przejściu pętli, w wyniku odjęcia `'0'` od `'4'`, zmienna `intValue` otrzymuje wartość 4. Mnożąc `result` przez 10, otrzymujemy 20, gdy dodamy do tego `intValue`, mamy 24; taką liczbę zapisujemy w zmiennej `result`.

W trzecim przejściu pętli, w wyniku odjęcia `'0'` od `'5'`, zmienna `intValue` otrzymuje wartość 5. Mnożymy `result` przez 10 (240), dodajemy do tego `intValue` i mamy 245. To jest wartość zmiennej `result` po trzecim przejściu pętli.

Kiedy dochodzimy do znaku null kończącego łańcuch, pętla `for` jest przerywana, a nasza funkcja zwraca wartość zmiennej `result`, czyli `245`.

Funkcję `strToInt` można poprawić na dwa sposoby. Po pierwsze, obecnie nie uwzględnia ona liczb ujemnych. Po drugie, nie wiemy, czy analizowany ciąg w ogóle zawierał *jakąkolwiek* cyfrę, na przykład wywołanie `strToInt("xxx")` zwróci 0. Wykonanie takich poprawek zostawiamy czytelnikowi jako ćwiczenie.

Na tym kończymy omawianie łańcuchów znakowych. Jak widać, język C umożliwia proste i skuteczne przetwarzanie takich łańcuchów. Biblioteka standardowa zawiera szereg funkcji operujących na łańcuchach. Istnieją w niej na przykład funkcję: `strlen` określająca długość łańcucha znakowego, `strcmp` porównująca dwa łańcuchy, `strcat` łącząca dwa łańcuchy, `strcpy` kopiująca jeden łańcuch do innego, `atoi` zamieniająca łańcuch na liczbę całkowitą, a także funkcje `isupper`, `islower`, `isalpha` i `isdigit` sprawdzające, czy przekazany im znak jest wielką literą, małą literą, literą w ogóle czy cyfrą. Dobrym ćwiczeniem jest przepisanie przykładowych programów z tego rozdziału przy użyciu funkcji bibliotecznych. W dodatku B, poświęconym bibliotece standardowej języka C, znajdziemy informacje o wielu funkcjach bibliotecznych.

Ćwiczenia

1. Przepisz i uruchom jedenaście programów pokazanych w tym rozdziale. Uzyskane wyniki porównaj z wynikami pokazanymi w tekście.
2. Odpowiedz, dzięki czemu mogliśmy zastąpić instrukcję `while` z funkcji `equalStrings` w programie 9.4 instrukcją:

   ```
   while ( s1[i] == s2[i]  &&  s1[i] != '\0' )
   ```

   uzyskując taki sam wynik?
3. Funkcja `countWords` z programów 9.7 i 9.8 nieprawidłowo zlicza słowa zawierające apostrof, gdyż traktuje je jak dwa osobne słowa. Zmodyfikuj tę funkcję tak, aby rozwiązać opisany problem. Dodatkowo rozszerz funkcję tak, aby liczby dodatnie i ujemne, łącznie z ewentualnym znakiem, kropką dziesiętną i częścią ułamkową, traktowała jako pojedyncze słowa.
4. Napisz funkcję `substring` pobierającą część łańcucha. Funkcja ta ma być wywoływana następująco:

   ```
   substring (źródło, początek, liczba, wynik);
   ```

   gdzie `źródło` to łańcuch, z którego wycinamy część, `początek` to indeks pierwszego wycinanego znaku, `liczba` to liczba wycinanych znaków, a `wynik` to uzyskany podciąg. Na przykład wywołanie:

   ```
   substring ("character", 4, 3, result);
   ```

   powoduje wycięcie łańcucha `"act"` (trzy znaki poczynając od znaku o indeksie 4).

Pamiętaj o wstawieniu na koniec podłańcucha `result` znaku null. Niech funkcja sprawdza, czy łańcuch, z którego następuje wycinanie, ma tyle znaków, ilu żądamy. Jeśli nie, niech wycięty podłańcuch kończy się wraz z końcem łańcucha wejściowego. Na przykład wywołanie:

```
substring ("dwa słowa", 4, 20, result);
```

powinno wyciąć napis „słowa”, mimo że w wywołaniu zażądano 20 znaków.

5. Napisz funkcję `findString` sprawdzającą, czy jeden ciąg występuje w drugim. Pierwszym parametrem funkcji niech będzie przeszukiwany łańcuch, a drugim — łańcuch szukany. Jeśli funkcja znajdzie żądany łańcuch, niech zwróci jego położenie w łańcuchu przeszukiwanym. Jeśli poszukiwanie zakończy się niepowodzeniem, funkcja ma zwrócić –1. Na przykład wywołanie:

   ```
   index = findString ("maskotka", "kot");
   ```

   szuka w łańcuchu `"maskotka"` łańcucha `"kot"`. Jako że `"kot"` występuje w przeszukiwanym łańcuchu, funkcja zwraca 3, gdyż pierwszy znak podłańcucha ma indeks 3.

6. Napisz funkcję `removeString` usuwającą podaną liczbę znaków z łańcucha znakowego. Funkcja będzie miała trzy parametry — łańcuch wejściowy, indeks pierwszego usuwanego znaku oraz liczbę usuwanych znaków. Jeśli zatem tablica znakowa `tekst` zawiera napis `"to nie ten!"`, to wywołanie:

   ```
   removeString (tekst, 3, 4);
   ```

   spowoduje usunięcie słowa „nie” (wraz ze znajdującą się przed nim spacją). Wobec tego uzyskany zostanie łańcuch `"to ten!"`.

7. Napisz funkcję `insertString` wstawiającą jeden łańcuch znakowy do innego. Parametrami tej funkcji będą łańcuch źródłowy, łańcuch wstawiany oraz miejsce wstawienia tego łańcucha. Wobec tego wywołanie:

   ```
   insertString (tekst, "jest ", 7);
   ```

   jeśli `tekst` zdefiniowano tak samo jak w poprzednim ćwiczeniu, spowoduje wstawienie do zmiennej `tekst` łańcucha `"jest "` i umieszczenie pierwszego znaku w `tekst[10]`. Po takim wywołaniu tablica `tekst` będzie zawierała napis `"to nie jest ten!"`.

8. Korzystając z funkcji `findString`, `removeString` i `insertString` z poprzednich ćwiczeń, napisz funkcję `replaceString` pobierającą jako parametry trzy następujące łańcuchy znakowe:

   ```
   replaceString (zrodlo, s1, s2);
   ```

   i zastępującą wystąpienie `s1` w łańcuchu `zrodlo` łańcuchem `s2`. Funkcja ma za pomocą `findString` znaleźć `s1`, wywołać `removeString` w celu usunięcia `s1`, w końcu wywołać `insertString`, aby w to miejsce wstawić `s2`.
   Zatem wywołanie:

   ```
   replaceString (tekst, "1", "jeden");
   ```

zastąpi pierwsze wystąpienie łańcucha "1" w zmiennej tekst (jeśli ono istnieje) łańcuchem "jeden". Analogicznie wywołanie:

```
replaceString (tekst, "*", "");
```

usunie z tablicy tekst pierwszą gwiazdkę, gdyż zostanie ona zastąpiona ciągiem pustym.

9. Można jeszcze udoskonalić funkcję replaceString, jeśli jej wynik będzie mówił, czy podstawienie się udało, czyli czy znaleziono zastępowany łańcuch w łańcuchu wejściowym. Jeśli zatem powodzenie powoduje zwrócenie true, niepowodzenie zaś — false, to pętla:

```
do
    jeszczeJest = replaceString (tekst, " ", "");
while ( jeszczeJest == true );
```

usunie z łańcucha tekst *wszystkie* spacje.

Zaimplementuj opisaną modyfikację w funkcji replaceString i wypróbuj ją na różnych łańcuchach, aby uzyskać pewność, że działa poprawnie.

10. Napisz funkcję dictionarySort, która alfabetycznie posortuje słownik zdefiniowany w programach 9.9 i 9.10.

11. Rozszerz funkcję strToInt z programu 9.11 tak, aby w przypadku, kiedy pierwszym znakiem łańcucha jest symbol minusa, liczba była traktowana jako ujemna.

12. Napisz funkcję strToFloat zamieniającą łańcuch znakowy na wartość zmiennoprzecinkową. Niech funkcja ta uwzględnia także początkowy minus oznaczający liczbę ujemną. Wobec tego wywołanie:

```
strToFloat ("–867.6921");
```

ma zwrócić liczbę –867.6921.

13. Jeśli c jest małą literą i używamy kodowania ASCII, to wyrażenie:

```
c – 'a' + 'A'
```

da w wyniku wielką literę odpowiadającą dotychczasowej wartości c.

Napisz funkcję uppercase, która zamieni wszystkie małe litery w łańcuchu na ich wielkie odpowiedniki.

14. Napisz funkcję intToStr zamieniającą liczbę całkowitą na łańcuch znakowy. Pamiętaj o prawidłowej obsłudze liczb ujemnych.

10

Wskaźniki

W tym rozdziale zajmiemy się jednym z najważniejszych mechanizmów języka C, *wskaźnikami*. Tym, co w największym stopniu odróżnia język C od innych języków programowania, jest właśnie elastyczność i ogromne możliwości użycia wskaźników. Wskaźniki pozwalają wydajnie obsługiwać złożone struktury danych, zmieniać wartości przekazywane do funkcji jako parametry, używać pamięci alokowanej dynamicznie (więcej na ten temat w rozdziale 16.), a w końcu w bardziej zwarty i skuteczny sposób obsługiwać tablice.

Doświadczeni programiści języka C używają wskaźników praktycznie do wszystkiego, dlatego w tym rozdziale opisuję różne sposoby ich implementacji i użycia, między innymi:

- definiowanie prostych wskaźników;
- używanie wskaźników w typowych wyrażeniach w języku C;
- implementowanie wskaźników do struktur, tablic i funkcji;
- tworzenie list powiązanych przy użyciu wskaźników;
- dodawanie słowa kluczowego `const` do wskaźników;
- przekazywanie wskaźników jako argumentów do funkcji.

Choć wskaźniki to jeden z najtrudniejszych do opanowania aspektów języka C, gdy już zrozumiesz podstawowe zasady ich działania, zaczniesz pisać znacznie elegantsze i wydajniejsze programy.

Wskaźniki i przekierowania

Aby zrozumieć sposób działania wskaźników, najpierw trzeba zrozumieć pojęcie *przekierowania*. Spotykamy się z nim na co dzień. Załóżmy na przykład, że potrzebujemy nowego cartridge'a do drukarki. W firmie, w której pracujemy, wszystkie zakupy obsługuje wydział zaopatrzenia. Wobec tego dzwonimy do Jacka z zaopatrzenia i prosimy go o zamówienie nowego cartridge'a. Z kolei Jacek dzwoni do lokalnego sklepu i zamawia cartridge. W ten sposób nasze zamówienie jest przekierowywane, gdyż nie zamówiliśmy potrzebnej rzeczy sami bezpośrednio w sklepie.

To samo pojęcie przekierowania dotyczy działania wskaźników w języku C. Wskaźnik także zapewnia pośredni dostęp do danych. I tak jak istnieją pewne powody, dla których wszystkie zakupy są realizowane przez dział zaopatrzenia (na przykład użytkownicy nie wiedzą, w którym sklepie zamawia się cartridge'e), tak samo są ważne powody przemawiające za używaniem wskaźników w języku C.

Definiowanie zmiennej wskaźnikowej

Ale dość już gadania; zobaczmy, jak działają wskaźniki. Załóżmy, że zdefiniowaliśmy zmienną `count`:

```
int count = 10;
```

Możemy teraz zdefiniować następną zmienną — `int_pointer` — która umożliwi pośredni dostęp do wartości zmiennej `count`:

```
int *int_pointer;
```

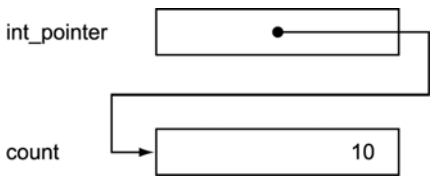
Gwiazdka to informacja dla kompilatora C, która oznacza, że zmienna `int_pointer` jest wskaźnikiem — tutaj do zmiennej typu `int`. Znaczy to, że zmienna `int_pointer` służy do pośredniego sięgania do wartości jednej lub różnych zmiennych typu `int`.

Gdy wcześniej używaliśmy funkcji `scanf`, korzystaliśmy z dodatkowego operatora `&`. Jest to operator jednoargumentowy, nazywany operatorem *adresu*, i służy do tworzenia wskaźnika do zmiennej. Wyrażenie `&x` można przypisać dowolnej zmiennej wskaźnikowej, jeśli jej typ wskaźnikowy odpowiada typowi zmiennej `x`.

Wobec tego, mając podane wcześniej definicje zmiennych `count` i `int_pointer`, możemy napisać:

```
int_pointer = &count;
```

tworzymy w ten sposób pośrednie powiązanie zmiennych `int_pointer` i `count`. Operator adresu ma wpływ na to, co przypisujemy zmiennej `int_pointer`, nie wpływa natomiast na wartość `count`. Operator ten zwraca *wskaźnik* do zmiennej `count`. Powiązanie zmiennych `int_pointer` i `count` pokazano na rysunku 10.1. Strzałka oznacza, że `int_pointer` nie zawiera bezpośrednio wartości `count`, ale wskazuje tę zmienną.



Rysunek 10.1. Wskaźnik na liczbę typu int

Aby odwołać się do wartości zmiennej count za pośrednictwem wskaźnika int_pointer, używamy operatora *wyłuskania* zapisywanego jako gwiazdka — *. Jeśli zatem zmienna x jest typu int, instrukcja:

```
x = *int_pointer;
```

powoduje przypisanie wartości zmiennej x pośrednio wskazywanej przez int_pointer. Poprzednio zmienna int_pointer wskazywała zmienną count, pokazana instrukcja spowoduje przypisanie zmiennej x wartości zmiennej count, czyli 10.

Powyższe instrukcje wstawiono do programu 10.1, pokazującego dwa najważniejsze operatory wskaźnikowe, czyli operator adresu — & — oraz operator wyłuskania — *.

Program 10.1. **Pokaz użycia wskaźników**

```
// Program pokazujący użycie wskaźników

#include <stdio.h>

int main (void)
{
    int  count = 10, x;
    int  *int_pointer;

    int_pointer = &count;
    x = *int_pointer;

    printf ("count = %i, x = %i\n", count, x);

    return 0;
}
```

Program 10.1. **Wyniki**

```
count = 10, x = 10
```

Zmienne count i x deklarujemy normalnie jako zmienne typu int. W następnym wierszu deklarujemy zmienną int_pointer jako „wskaźnik do zmiennej typu int". Zauważmy, że oba te wiersze można by połączyć w jedną deklarację:

```
int  count = 10, x, *int_pointer;
```

Następnie do zmiennej count stosujemy operator adresu. Powoduje to utworzenie wskaźnika do tej zmiennej, który z kolei jest przypisywany przez program zmiennej int_pointer.

Wykonanie następnej instrukcji programu:

```
x = *int_pointer;
```

odbywa się następująco: operator przekierowania przekazuje do kompilatora C informację, że zmienną int_pointer ma traktować jako wskaźnik do pewnej informacji. Wskaźnik ten będzie potem używany do sięgnięcia do pewnej wartości, której typ wynika z typu wskaźnika.

W momencie deklaracji poinformowaliśmy kompilator, że `int_pointer` wskazuje liczby całkowite, tak więc dla kompilatora wyrażenie `*int_pointer` daje wartość typu `int`. Skoro `int_pointer` wskazuje zmienną `count`, to wyrażenie to tak naprawdę wskazuje właśnie wartość tej zmiennej.

Trzeba zdać sobie sprawę z tego, że program 10.1 stanowi sfabrykowany przykład użycia wskaźników i nie pokazuje praktycznego sposobu ich używania. Praktyczne zasady pokażemy już niedługo, kiedy tylko poznamy najważniejsze sposoby definiowania i przetwarzania wskaźników.

Program 10.2 demonstruje niektóre ciekawe cechy zmiennych wskaźnikowych. Używamy tutaj wskaźnika do znaku.

Program 10.2. **Podstawy wskaźników, ciąg dalszy**

```
// Jeszcze nieco przykładów wskaźników

#include <stdio.h>

int main (void)
{
    char  c = 'Q';
    char  *char_pointer = &c;

    printf ("%c %c\n", c, *char_pointer);

    c = '/';
    printf ("%c %c\n", c, *char_pointer);

    *char_pointer = '(';
    printf ("%c %c\n", c, *char_pointer);

    return 0;
}
```

Program 10.2. **Wyniki**

```
Q Q
/ /
( (
```

Zmienna znakowa `c` jest definiowana i następnie inicjalizowana za pomocą znaku `'Q'`. W następnym wierszu definiujemy zmienną `char_pointer` jako „wskaźnik do typu `char`”, czyli wartość tej zmiennej będziemy traktowali jako pośrednie odwołanie (właśnie wskaźnik) do zmiennej typu `char`. Zauważmy, że tej zmiennej normalnie możemy przypisać wartość początkową. Wartość przypisywana zmiennej `char_pointer` to wskaźnik na zmienną `c` otrzymywany przez zastosowanie do zmiennej `c` operatora adresu. Zauważmy, że taka inicjalizacja spowodowałaby błąd kompilacji, bowiem zmienna `c` została zdefiniowana *po* tej instrukcji; zmienna musi być zadeklarowana *wcześniej*, niż pojawi się do niej jakiekolwiek odwołanie.

Deklarację zmiennej `char_pointer` i przypisanie jej wartości początkowej można szybko zapisać w dwóch instrukcjach:

```
char *char_pointer;
char_pointer = &c;
```

natomiast nie zadziałają już instrukcje:

```
char  *char_pointer;
*char_pointer = &c;
```

co można by wnioskować po deklaracji jednowierszowej.

Zawsze trzeba pamiętać, że w języku C wartość wskaźnikowa jest bezużyteczna, dopóki nie zostanie ustawiona.

Pierwsze wywołanie `printf` po prostu pokazuje zawartość zmiennej `c` oraz zawartość zmiennej wskazywanej przez `char_pointer`. Ustawiliśmy zmienną `char_pointer` tak, aby wskazywała zmienną `c`, więc tutaj też widzimy wartość tej samej zmiennej `c`. Można to zauważyć w pierwszym wierszu wyników programu.

W następnym wierszu programu zmiennej znakowej `c` przypisywany jest znak `'/'`. Zmienna `char_pointer` stale wskazuje zmienną `c`; wywołanie `printf` z parametrem `*char_pointer` prawidłowo wyświetla nową wartość zmiennej `c`. Jest to ważne. Jeżeli nie zmieni się wartość wskaźnika `char_pointer`, wyrażenie `*char_pointer` *zawsze* pokaże aktualną wartość zmiennej `c`. Wobec tego, kiedy wartość `c` się zmieni, zmieni się też wartość `*char_pointer`.

To, co na razie powiedzieliśmy, pomoże zrozumieć działanie następnego wyrażenia w programie. Gdy nie zmieni się wartość zmiennej `char_pointer`, wyrażenie `*char_pointer` zawsze będzie się odwoływało do zmiennej `c`. Wobec tego w wyrażeniu:

```
*char_pointer = '(';
```

zmiennej `c` przypisujemy znak lewego nawiasu. Formalnie, znak `'('` przypisujemy zmiennej wskazywanej przez `char_pointer`. Wiemy, że tą zmienną jest `c`, gdyż tak ustawiliśmy wskaźnik `char_pointer` na początku programu.

Dotychczasowe rozważania stanowią klucz do zrozumienia działania wskaźników. Jeśli nadal cokolwiek jest niejasne, należy jeszcze raz przeczytać początek tego rozdziału.

Wskaźniki w wyrażeniach

W programie 10.3 definiujemy dwa wskaźniki na liczby całkowite, `p1` i `p2`. Zauważmy, że wartość, którą pokazuje wskaźnik, może być użyta w wyrażeniach arytmetycznych. Jeśli `p1` jest wskaźnikiem na liczbę całkowitą, co możemy powiedzieć o zasadach użycia `*p1` w wyrażeniach?

Program 10.3. **Wskaźniki w wyrażeniach**

```
// Jeszcze o wskaźnikach

#include <stdio.h>
```

```
int main (void)
{
    int  i1, i2;
    int  *p1, *p2;

    i1 = 5;
    p1 = &i1;
    i2 = *p1 / 2 + 10;
    p2 = p1;

    printf ("i1 = %i, i2 = %i, *p1 = %i, *p2 = %i\n", i1, i2, *p1, p2);

    return 0;
}
```

Program 10.3. **Wyniki**

```
i1 = 5, i2 = 12, *p1 = 5, *p2 = 5
```

Po zdefiniowaniu zmiennych `i1` i `i2` typu `int` oraz wskaźników na liczby `int` — `p1` i `p2` — przypisujemy zmiennej `i1` wartość 5, następnie w `p1` zapisujemy wskaźnik na `i1`. Wartość `i2` wyliczamy z wyrażenia:

```
i2 = *p1 / 2 + 10;
```

Z omówienia programu 10.2 wynika, że jeśli wskaźnik `px` wskazuje zmienną `x` i `px` zdefiniowano jako wskaźnik na taki typ, jakiego typu jest `x`, użycie `*px` w wyrażeniu pod każdym względem jest równoważne zastosowaniu w tym wyrażeniu zmiennej `x`.

W programie 10.3 zmienna `p1` jest wskaźnikiem na typ `int`, więc całe wyrażenie jest obliczane zgodnie z arytmetyką całkowitoliczbową. Skoro wartością `*p1` jest 5 (`p1` wskazuje `i1`), ostatecznym wynikiem jest 12 i taka właśnie wartość jest przypisywana zmiennej `i2` (operator dereferencji wskaźnika * ma wyższy priorytet niż operator arytmetyczny dzielenia; operator ten wraz z operatorem adresu — & — mają priorytet wyższy od wszystkich operatorów binarnych języka C).

W następnej instrukcji wartość zmiennej `p1` jest przypisywana zmiennej `p2`. Przypisanie to jest jak najbardziej poprawne i powoduje, że `p2` wskazuje tę samą informację co `p1`. Skoro `p1` wskazuje zmienną `i1`, po przypisaniu `p2` *także* wskazuje zmienną `i1` (zresztą do jednej informacji można mieć dowolnie wiele wskaźników).

Instrukcja `printf` potwierdza, że wartości `i1`, `*p1` i `*p2` są takie same (5), a wartość zmiennej `i2` wynosi 12.

Wskaźniki i struktury

Widzieliśmy, jak wskaźnik może wskazywać dowolny typ podstawowy, taki jak `int` czy `char`. Jednak wskaźniki mogą także wskazywać struktury. W rozdziale 8. strukturę `date` zdefiniowaliśmy następująco:

```
struct date
{
    int day;
    int month;
    int year;
};
```

Tak jak definiujemy zmienne typu `struct date`:

```
struct date  todaysDate;
```

tak samo możemy zdefiniować wskaźnik na zmienną `struct date`:

```
struct date *datePtr;
```

Tak zdefiniowana zmienna `datePtr` może być już używana zgodnie ze zwykłymi zasadami. Możemy na przykład ustawić ją, aby wskazywała zmienną `todaysDate` — wystarczy przypisanie:

```
datePtr = &todaysDate;
```

Teraz możemy pośrednio sięgać do pól struktury `date` wskazywanej przez `datePtr`:

```
(*datePtr).day = 21;
```

Instrukcja ta powoduje ustawienie pola dnia struktury `date` wskazywanej przez `datePtr` na 21. Nawiasy są konieczne, gdyż operator pola struktury — `.` — ma wyższy priorytet niż operator wyłuskania — *.

Aby sprawdzić, jaka wartość jest zapisana w polu `month` struktury `date` wskazywanej przez `datePtr`, wystarczy użyć instrukcji:

```
if ( (*datePtr).month == 12 )
    ...
```

Wskaźniki na struktury są w języku C tak często używane, że do ich obsługi istnieje specjalny operator. Operator ten — `->` — (składający się z kreski i znaku większości) zastępuje zapis:

```
(*x).y
```

Powyższe wyrażenie jest zatem równoważne:

```
x->y
```

Tak więc powyższą instrukcję `if` można wygodnie zapisać jako:

```
if ( datePtr->month == 12 )
    ...
```

Pierwszy program pokazujący użycie struktur — 8.1 — możemy przepisać, korzystając ze wskaźników na struktury, jak to zrobiono w programie 10.4.

Program 10.4. **Użycie wskaźników na struktury**

```
// Program pokazujący użycie wskaźników na struktury

#include <stdio.h>
```

```
int main (void)
{
    struct date
    {
        int  day;
        int  month;
        int  year;
    };

    struct date  today, *datePtr;

    datePtr = &today;

    datePtr->day = 25;
    datePtr->month = 9;
    datePtr->year = 2015;

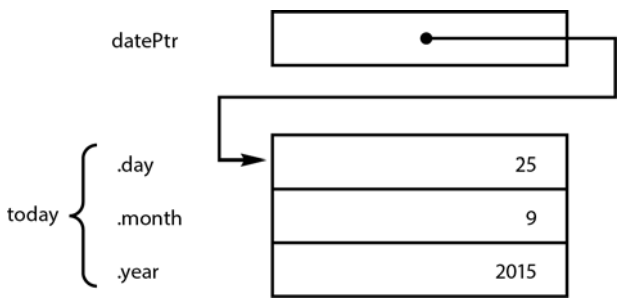
    printf ("Dzisiaj jest %i.%i.%.2i.\n",
             datePtr->day, datePtr->month, datePtr->year % 100);

    return 0;
}
```

Program 10.4. **Wyniki**

```
Dzisiaj jest 25.9.15.
```

Na rysunku 10.2 pokazano położenie w pamięci zmiennych `today` i `datePtr` po wykonaniu przypisań z powyższego programu.



Rysunek 10.2. Wskaźnik na strukturę

Trzeba tu ponownie przypomnieć, że powyższy program, ze wskaźnikami na struktury, wcale nie jest lepszy od programu 8.1. Już wkrótce powiemy, dlaczego warto stosować wskaźniki na struktury.

Struktury zawierające wskaźniki

Oczywiście wskaźnik może być elementem struktury. W definicji:

```
struct  intPtrs
{
    int  *p1;
    int  *p2;
};
```

struktura `intPtrs` zawiera dwa wskaźniki na dane typu `int` — `p1` i `p2`. Można normalnie definiować zmienne typu `struct intPtrs`:

```
struct intPtrs pointers;
```

Zmienna `pointers` może być teraz normalnie używana, tylko trzeba pamiętać, że sama `pointers` *nie jest* wskaźnikiem, ale zwykłą strukturą mającą dwa pola wskaźnikowe.

Program 10.5 pokazuje sposób obsługi struktury `intPtrs` w programie.

Program 10.5. **Użycie struktur zawierających wskaźniki**

```
// Funkcja pokazująca użycie struktur zawierających wskaźniki

#include <stdio.h>

int main (void)
{
    struct  intPtrs
    {
        int  *p1;
        int  *p2;
    };

    struct intPtrs pointers;
    int  i1 = 100, i2;

    pointers.p1 = &i1;
    pointers.p2 = &i2;
    *pointers.p2 = -97;

    printf ("i1 = %i, *pointers.p1 = %i\n", i1, *pointers.p1);
    printf ("i2 = %i, *pointers.p2 = %i\n", i2, *pointers.p2);
    return 0;
}
```

Program 10.5. **Wyniki**

```
i1 = 100, *pointers.p1 = 100
i2 = -97, *pointers.p2 = -97
```

Znajdujące się za definicjami zmiennych przypisanie:

```
pointers.p1 = &i1;
```

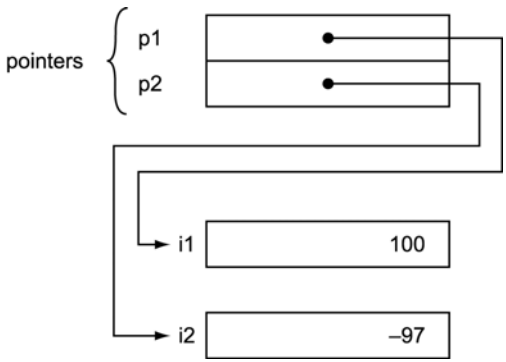
powoduje ustawienie pola `p1` zmiennej `pointers` na zmienną `i1`, drugie zaś przypisanie:

```
pointers.p2 = &i2;
```

ustawia pole p2 tak, aby wskazywało zmienną i2. Następnie zmiennej wskazywanej przez pointers.p2 przypisujemy wartość –97. Wskazywaną zmienną jest i2, więc ona otrzymuje wartość –97. W tym przypisaniu zbędne są nawiasy, gdyż — jak wspominaliśmy wcześniej — kropka będąca operatorem pól struktury ma wyższy priorytet niż operator wyłuskania. Oczywiście nawiasy można zastosować choćby dla bezpieczeństwa, bo czasem trudno zapamiętać, który operator ma wyższy priorytet.

Dwa znajdujące się dalej wywołania funkcji printf pokazują poprawność naszych przypisań.

Na rysunku 10.3 zilustrowano związek między zmiennymi i1, i2 i pointers po przypisaniach z programu 10.5. Jak widać, pole p1 wskazuje zmienną i1, która zawiera wartość 100. Pole p2 wskazuje zmienną i2 zawierającą wartość –97.



Rysunek 10.3. Struktura zawierająca wskaźniki

Listy powiązane

Pojęcie wskaźników na struktury i struktur zawierających wskaźniki daje programiście ogromne możliwości, gdyż pozwala tworzyć zaawansowane struktury danych, takie jak *listy powiązane*, *listy podwójnie powiązane* oraz *drzewa*.

Załóżmy, że mamy następującą strukturę:

```
struct entry
{
    int           value;
    struct entry  *next;
};
```

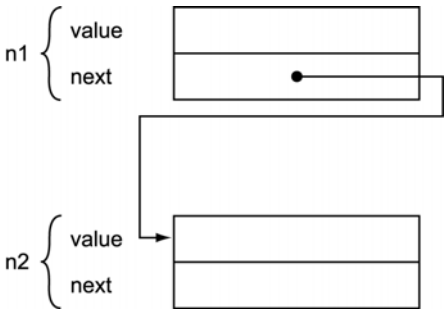
Mamy zatem strukturę entry zawierającą dwa pola. Pierwsze z nich to po prostu liczba całkowita — value. Drugie pole struktury nazywa się next (*następny*) i jest *wskaźnikiem struktury* entry. Zastanówmy się nad tym przez chwilę — w strukturze entry jest wskaźnik innej struktury entry. Jest to jak najzupełniej poprawna konstrukcja języka C. Załóżmy teraz, że zdefiniujemy następujące dwie zmienne typu struct entry:

```
struct entry n1, n2;
```

Ustawiamy wskaźnik w strukturze `n1` tak, aby wskazywał strukturę `n2`:

```
n1.next = &n2;
```

Instrukcja ta wiąże ze sobą `n1` i `n2`, jak to pokazano na rysunku 10.4.

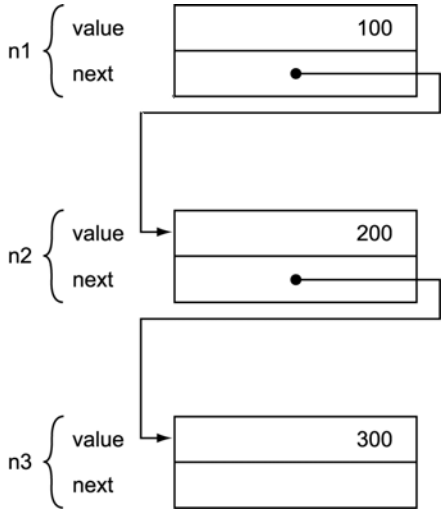


Rysunek 10.4. Powiązane struktury

Przy założeniu, że zmienna `n3` także jest typu `struct entry`, moglibyśmy dodać następny powiązany element:

```
n2.next = &n3;
```

Spowoduje to utworzenie całego łańcucha powiązanych elementów, tak jak na rysunku 10.5. Jest to lista powiązana, pokazana w programie 10.6.



Rysunek 10.5. Lista powiązana

Program 10.6. **Użycie list powiązanych**

```
// Funkcja korzystająca z list powiązanych

#include <stdio.h>

int main (void)
{
    struct  entry
    {
        int           value;
        struct entry  *next;
    };

    struct entry n1, n2, n3;
    int          i;

    n1.value = 100;
    n2.value = 200;
    n3.value = 300;

    n1.next = &n2;
    n2.next = &n3;

    i = n1.next->value;
    printf ("%i  ", i);

    printf ("%i\n", n2.next->value);

    return 0;
}
```

Program 10.6. **Wyniki**

```
           200   300
```

Mamy trzy struktury typu `struct entry`: `n1`, `n2` i `n3`. Każda z nich ma pole całkowitoliczbowe `value` oraz pole wskaźnikowe na strukturę `entry`, czyli `next`. Program przypisuje polom `value` kolejnych struktur wartości 100, 200 i 300.

Następne dwie instrukcje:

```
n1.next = &n2;
n2.next = &n3;
```

tworzą listę powiązaną, w której pole `next` struktury `n1` wskazuje strukturę `n2`, a pole `next` struktury `n2` wskazuje `n3`.

Wykonanie instrukcji:

```
i = n1.next->value;
```

odbywa się następująco — wartość pola `value` struktury `entry` wskazywanej przez `n1.next` jest przypisywana do zmiennej `i` typu `int`. Ustawiliśmy `n1.next` tak, aby wskazywało strukturę `n2`, więc sięgamy do pola `value` struktury `n2`. Wobec tego w instrukcji powyższej zmiennej `i` przypisywana jest wartość 200, co potwierdza następna instrukcja `printf`.

Jeśli poprawne jest wyrażenie `n1.next->value`, to `n1.next.value` już nie, gdyż pole `n1.next` zawiera wskaźnik na strukturę, nie zawiera natomiast samej struktury. To rozróżnienie jest bardzo ważne i jego niezrozumienie szybko prowadzi do błędów w programach.

Operator pola struktury (`.`) oraz operator wskaźnika struktury (`->`) mają w języku taki sam priorytet. Wyrażenia takie jak powyższe, w których występują oba te operatory, są interpretowane od lewej strony do prawej. Wobec tego nasze wyrażenie zostanie zinterpretowane jako:

```
i = (n1.next)->value;
```

— i o to chodziło.

Drugie wywołanie `printf`, występujące w programie 10.6, pokazuje pole `value` wskazywane przez `n2.next`. Ustawiliśmy `n2.next` tak, aby wskazywało strukturę `n3`, więc program pokaże wartość `n3.value`.

Jak wspominaliśmy wcześniej, listy powiązane są użytecznymi narzędziami programistycznymi. Lista powiązana znakomicie upraszcza operacje, takie jak wstawianie i usuwanie elementów z dużych zbiorów uporządkowanych.

Jeśli na przykład mamy `n1`, `n2` i `n3` zdefiniowane tak jak poprzednio, możemy bez trudu usunąć z listy `n2`; wystarczy po prostu tak ustawić pole `next` zmiennej `n1`, aby wskazywało to, na co dotąd wskazywało pole `next` struktury `n2`:

```
n1.next = n2.next;
```

Taka instrukcja spowoduje skopiowanie wskaźnika z `n2.next` do `n1.next`, a skoro `n2.next` wskazywało `n3`, teraz `n1.next` będzie wskazywało `n3`. Dalej, jako że `n1` nie wskazuje już `n2`, właśnie usunęliśmy `n2` z naszej listy. Rysunek 10.6 pokazuje sytuację po wykonaniu powyższej instrukcji. Oczywiście moglibyśmy bezpośrednio spowodować, aby `n1` wskazywało `n3`; wystarczyłoby zapisać:

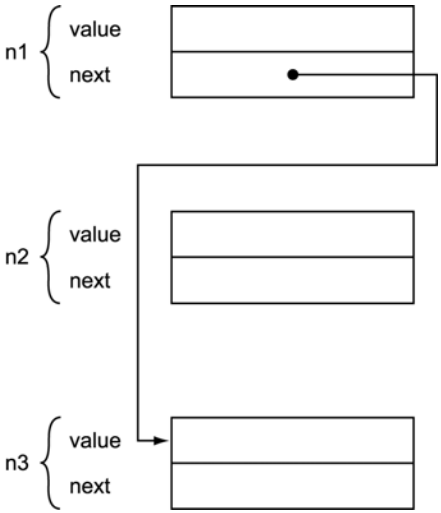
```
n1.next = &n3;
```

Jednak w takim wypadku instrukcja przestałaby być ogólna, gdyż z góry musielibyśmy wiedzieć, że `n2` wskazuje `n3`.

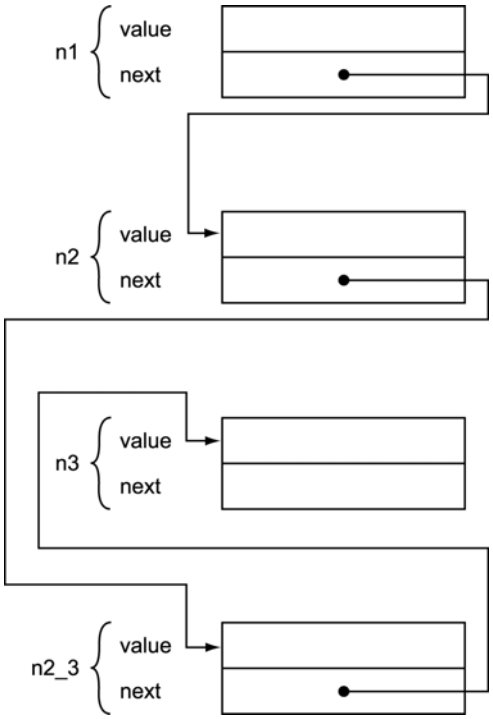
Wstawianie elementów na listę jest równie proste. Jeśli na naszą chcemy wstawić listę zmienną `n2_3` typu `struct entry`, po prostu ustawiamy pole `n2_3.next` tak, aby wskazywało to, co dotąd wskazywało pole `n2.next`; poza tym ustawiamy `n2.next`, aby wskazywało nową strukturę — `n2_3`. Wobec tego instrukcje:

```
n2_3.next = n2.next;
n2.next = &n2_3;
```

spowodują wstawienie `n2_3` na listę zaraz za elementem `n2`. Zauważmy, że kolejność powyższych dwóch instrukcji jest ważna, gdyż druga z nich powoduje nadpisanie wskaźnika z `n2.next`; zamiana kolejności uniemożliwiłaby przypisanie pierwotnej wartości polu `n2_3.next`. Naszą listę z wstawioną strukturą `n2_3` pokazano na rysunku 10.7. Zauważmy, że `n2_3` nie umieszczono między `n1` a `n3`; chodzi o podkreślenie, że zmienna ta może znajdować się w dowolnym miejscu w pamięci, nie musi fizycznie być między `n1` a `n3`.



Rysunek 10.6. Usunięcie elementu z listy powiązanej



Rysunek 10.7. Wstawienie elementu na listę powiązaną

Jest to podstawowa zaleta użycia list powiązanych do przechowywania informacji — elementy listy nie muszą być umieszczane w pamięci kolejno, tak jak elementy tablicy.

Zanim zajmiemy się pisaniem funkcji obsługujących listy powiązane, musimy jeszcze powiedzieć o dwóch rzeczach. Zwykle przechowuje się przynajmniej jeden dodatkowy wskaźnik związany z listą, wskazujący początek listy. Jeśli zatem mamy naszą pierwotną trzyelementową listę, złożoną ze zmiennych `n1`, `n2` i `n3`, zdefiniujemy jeszcze zmienną wskaźnikową `list_pointer` i ustawimy ją tak, aby wskazywała początek listy:

```
struct entry *list_pointer = &n1;
```

Zakładamy, że `n1` zdefiniowaliśmy wcześniej. Wskaźnik początku listy przydaje się, kiedy trzeba przejść po kolejnych elementach listy, co wkrótce zobaczymy.

Druga kwestia to konieczność wskazania końca listy. Jest to obowiązkowe, aby na przykład procedura przeszukująca listę „wiedziała", że już cała lista została przeanalizowana. Stosowana jest konwencja, według której koniec listy oznacza się pustym wskaźnikiem — *null*. Polu wskaźnikowemu ostatniego elementu listy przypisujemy właśnie taką wartość i to wystarcza[1].

W naszej trzyelementowej liście koniec możemy zaznaczyć, zapisując pusty wskaźnik w polu `n3.next`:

```
n3.next = (struct entry *) 0;
```

W rozdziale 12., kiedy będziemy omawiali użycie preprocesora, zobaczymy, jak można poprawić czytelność powyższego przypisania.

Używamy tutaj operatora rzutowania, aby stałej 0 nadać właściwy typ (czyli „wskaźnik na strukturę `entry`"). Nie jest to konieczne, ale poprawia czytelność kodu.

Na rysunku 10.8 pokazano listę powiązaną z programu 10.6, ze wskaźnikiem na typ `struct entry` — `list_pointer` — wskazującym początek listy, oraz z polem `n3.next`, zawierającym pusty wskaźnik.

W programie 10.7 użyto wszystkich omówionych dotąd pojąć. Program wykorzystuje pętlę `while`, aby przejrzeć elementy listy i wyświetlić wartość pola `value` dla wszystkich jej elementów.

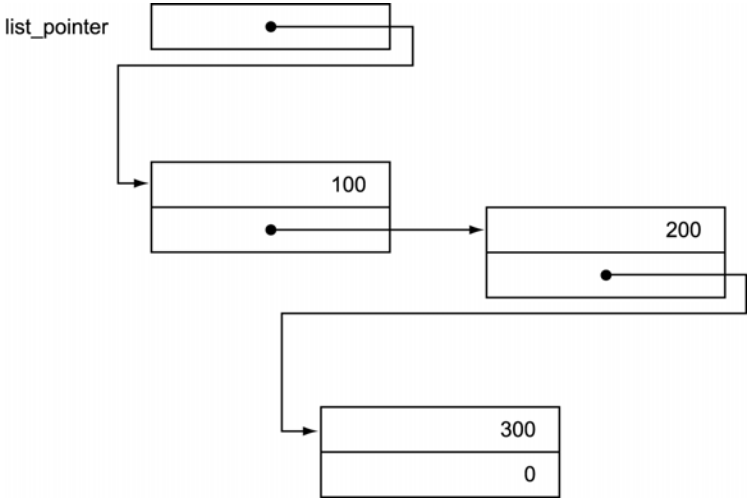
Program 10.7. **Przechodzenie po liście powiązanej**

```
// Program przechodzący po liście powiązanej

#include <stdio.h>

int main (void)
{
    struct  entry
    {
```

[1] Wskaźnik pusty nie musi być wewnętrznie zapisywany jako 0. Jednak kompilator musi interpretować przypisanie wskaźnikowi stałej 0 jako przypisanie wskaźnika pustego. Dotyczy to też porównywania wskaźnika ze stałą 0 — takie porównanie musi być interpretowane jako sprawdzanie, czy wskaźnik jest pusty.



Rysunek 10.8. Lista powiązana ze wskaźnikiem listy i kończącym wskaźnikiem pustym

```
    int           value;
    struct entry  *next;
};

struct entry  n1, n2, n3;
struct entry  *list_pointer = &n1;

n1.value = 100;
n1.next  = &n2;

n2.value = 200;
n2.next  = &n3;

n3.value = 300;
n3.next  = (struct entry *) 0;   // Oznaczenie końca listy pustym wskaźnikiem

while ( list_pointer != (struct entry *) 0 ) {
    printf ("%i\n", list_pointer->value);
    list_pointer = list_pointer->next;
}

return 0;
}
```

Program 10.7. **Wyniki**

```
100
200
300
```

W programie zdefiniowano zmienne n1, n2 i n3 oraz zmienną wskaźnikową list_pointer. Ta ostatnia początkowo wskazuje n1 — pierwszy element listy. Następnie program wiąże trzy elementy w listę, przy czym pole next zmiennej n3 jest ustawiane na pusty wskaźnik, aby tak oznaczyć koniec listy.

Dalej pętla while przechodzi po kolejnych elementach listy. Pętla ta jest wykonywana tak długo, aż list_pointer przybierze wartość pustego wskaźnika. Instrukcja printf w pętli while pokazuje wartość pola value elementu wskazywanego właśnie przez list_pointer.

Instrukcja znajdująca się za wywołaniem funkcji printf:

```
list_pointer = list_pointer->next;
```

powoduje, że wskaźnik przechodzi według pola next wskazywanej struktury do struktury następnej. Wobec tego w pierwszym przebiegu pętli instrukcja ta pobiera wskaźnik z n1.next (pamiętajmy, że wskaźnik list_pointer początkowo ustawiliśmy na n1) i przypisuje go zmiennej list_pointer. Jako że wartość ta nie jest pustym wskaźnikiem, lecz wskazuje strukturę n2, następuje kolejny przebieg pętli.

W drugim przebiegu pętli wyświetlana jest wartość n2.value, czyli 200. Pole next struktury n2 jest przypisywane zmiennej list_pointer, a ponieważ wskazywało ono strukturę n3, to list_pointer teraz też wskazuje n3.

Kiedy pętla while jest wykonywana po raz trzeci, wywołanie funkcji printf wyświetla wartość 300 z pola n3.value. Teraz wskaźnik list_pointer->next, czyli n3.next, jest kopiowany do zmiennej list_pointer; a że jest to wskaźnik pusty, działanie pętli while po trzech przejściach kończy się.

Omówione tutaj działanie pętli while można prześledzić z ołówkiem w ręku, aby dokładnie przeanalizować wartości poszczególnych zmiennych. Zrozumienie działania tej pętli stanowi klucz do zrozumienia działania wskaźników w języku C. Trzeba tu jeszcze odnotować, że dokładnie takiej samej pętli while można użyć do przeglądania listy *dowolnej* wielkości, gdy tylko jej koniec jest oznaczony pustym wskaźnikiem.

Pracując z praktycznie używanymi listami powiązanymi, zwykle nie wiążemy ze sobą jawnie definiowanych zmiennych, jak dotąd to robiliśmy. Tutaj po prostu chcemy opisać działanie pewnych mechanizmów. W praktyce zwykle przydziela się pamięć na każdy nowy element listy i elementy te wiąże się w listy dynamicznie. Służy do tego mechanizm *dynamicznej alokacji pamięci*, omawiany w rozdziale 16.

Słowo kluczowe const a wskaźniki

Widzieliśmy, jak można deklarować zmienne i tablice ze słowem kluczowym const, aby poinformować kompilator (i czytelnika naszego kodu), że zawartość danej zmiennej lub tablicy nie powinna się zmieniać w trakcie wykonywania programu. W przypadku wskaźników trzeba wziąć pod uwagę dwie rzeczy: czy zmieniać się będzie wskaźnik i czy będzie się zmieniać wskazywana przezeń wartość. Zastanówmy się nad tym. Załóżmy, że mamy następujące deklaracje:

```
char c = 'X';
char *charPtr = &c;
```

Zmienna wskaźnikowa `charPtr` wskazuje zmienną znakową `c`. Jeśli zawsze ma ona tę zmienną wskazywać, można ją zadeklarować ze słowem kluczowym `const`:

```
char * const charPtr = &c;
```

Interpretujemy to jako „`charPtr` to stały wskaźnik znaku". Wobec tego instrukcja typu:

```
charPtr = &d;      // niepoprawnie
```

powoduje, że kompilator GNU C pokaże komunikat typu[2]:

```
foo.c:10: warning: assignment of read-only variable 'charPtr'
```[3]

Jeśli teraz chcemy, aby wartość wskazywana przez `charPtr` nie zmieniała się *za pośrednictwem zmiennej wskaźnikowej `charPtr`*, możemy to zapisać następująco:

```
const char *charPtr = &c;
```

Tym razem definicję czytamy jako „`charPtr` wskazuje stałą wartość znakową". Nie oznacza to oczywiście, że nie można by zmieniać wartości zmiennej `c` wskazywanej przez `charPtr`; po prostu nie można użyć instrukcji typu:

```
*charPtr = 'Y';      // niepoprawnie
```

Powyższa instrukcja spowoduje, że kompilator GNU C pokaże komunikat typu:

```
foo.c:11: warning: assignment of read-only location
```[4]

Kiedy zmienna wskaźnikowa i wskazywane przez nią miejsce w pamięci nie mogą być zmieniane, używamy następującej deklaracji:

```
const char * const *charPtr = &c;
```

Pierwsze słowo `const` oznacza, że wskazywana pamięć nie może zmieniać swojej wartości. Drugie słowo znaczy, że nie może zmienić się sam wskaźnik. Tak, to faktycznie trochę skomplikowane, ale warto o tym powiedzieć w tym miejscu[5].

---

[2] Inne kompilatory mogą pokazywać inne komunikaty; niektóre mogą nawet nie pokazać żadnego komunikatu.

[3] `foo.c: 10: ostrzeżenie: przypisanie do zmiennej tylko do odczytu 'charPtr'` — *przyp. tłum.*

[4] `foo.c: 11: ostrzeżenie: przypisanie wartości do pamięci oznaczonej jako tylko do odczytu` — *przyp. tłum.*

[5] Słowo kluczowe `const` nie jest używane w każdym programie, w którym można by go użyć; pojawia się tylko w wybranych przykładach. Póki nie nabierzemy wprawy w czytaniu wyrażeń, takich jak powyższe, utrudniałoby zrozumienie kodu.

Wskaźniki i funkcje

Wskaźniki i funkcje świetnie ze sobą współpracują. Można użyć wskaźnika jako parametru funkcji, tak jak każdego innego typu; tak samo funkcja może zwrócić wskaźnik.

Pierwsza sytuacja — przekazywanie wskaźników jako parametrów — jest prosta: wskaźnik podaje się normalnie, jak każdy inny parametr. Aby zatem do funkcji `print_list` przekazać wskaźnik `list_pointer` z naszego poprzedniego programu, po prostu piszemy:

```
print_list (list_pointer);
```

W funkcji `print_list` odpowiedni parametr formalny musi być zdefiniowany jako wskaźnik do stosownego typu:

```
void print_list (struct entry *pointer)
{
    ...
}
```

Dalej parametru formalnego `pointer` używamy jak każdego innego wskaźnika. Warto powiedzieć o jednej rzeczy związanej z przekazywaniem wskaźników jako parametrów — wartość wskaźnika jest kopiowana na parametr formalny w chwili wywołania funkcji, więc żadne zmiany wartości tego parametru w funkcji nie wpłyną na przekazany wskaźnik. Jest tu jednak haczyk — jeśli sam wskaźnik nie zmieni się w funkcji, to może zmienić się wartość wskazywana przez ten wskaźnik. Powinno to się wyjaśnić po przeanalizowaniu programu 10.8.

Program 10.8. **Użycie wskaźników i funkcji**

```
// Program pokazujący użycie wskaźników i funkcji

#include <stdio.h>

void test (int *int_pointer)
{
    *int_pointer = 100;
}

int main (void)
{
    void test (int *int_pointer);
    int  i = 50, *p = &i;

    printf ("Przed wywołaniem test i = %i\n", i);
    test (p);
    printf ("Po wywołaniu test i = %i\n", i);

    return 0;
}
```

Program 10.8. **Wyniki**

```
Przed wywołaniem test i = 50
Po wywołaniu test i = 100
```

Funkcja test jako parametr pobiera wskaźnik na liczbę całkowitą. W tej funkcji jest tylko jedna instrukcja przypisująca zmiennej wskazywanej przez int_pointer wartość 100.

Funkcja main zawiera definicję stałej i typu int o wartości początkowej 50 oraz definicję wskaźnika na liczbę int — p — który początkowo wskazuje zmienną i. Program wyświetla wartość i, wywołuje funkcję test, przekazując jej zmienną p jako parametr. Jak widać, funkcja test faktycznie zmienia wartość i na 100.

Teraz przyjrzyjmy się programowi 10.9.

Program 10.9. **Użycie wskaźników do zamiany wartości**

```
// Jeszcze o wskaźnikach i funkcjach

#include <stdio.h>

void exchange (int * const pint1, int * const pint2)
{
    int temp;

    temp = *pint1;
    *pint1 = *pint2;
    *pint2 = temp;
}

int main (void)
{
    void exchange (int * const pint1, int * const pint2);
    int  i1 = -5, i2 = 66, *p1 = &i1, *p2 = &i2;

    printf ("i1 = %i, i2 = %i\n", i1, i2);
    exchange (p1, p2);
    printf ("i1 = %i, i2 = %i\n", i1, i2);

    exchange (&i1, &i2);
    printf ("i1 = %i, i2 = %i\n", i1, i2);

    return 0;
}
```

Program 10.9. **Wyniki**

```
i1 = -5, i2 = 66
i1 = 66, i2 = -5
i1 = -5, i2 = 66
```

Zadaniem funkcji exchange jest zamiana wartości dwóch liczb całkowitych wskazywanych przez parametry tej funkcji. Z nagłówka tej funkcji:

```
void exchange (int * const pint1, int * const pint2)
```

wynika, że funkcja exchange ma dwa parametry, a wskaźniki nie będą się zmieniały (użyto słowa kluczowego const).

Lokalna zmienna całkowitoliczbowa `temp` służy do przechowania wartości jednej z liczb podczas zamiany. Wartością tej zmiennej staje się wartość wskazywana przez `pint1`. Wtedy wartość wskazywana przez `pint2` jest kopiowana do zmiennej wskazywanej przez `pint1`, w końcu wartość `temp` jest zapisywana do zmiennej wskazywanej przez `pint1`, co kończy zamianę.

Funkcja `main` zawiera zmienne `i1` i `i2` o wartościach odpowiednio –5 i 66. Dalej mamy dwa wskaźniki, `p1` i `p2`, które wskazują `i1` i `i2`. Program wyświetla wartości `i1` i `i2`, wywołuje funkcję `exchange` i przekazuje jej jako parametry wskaźniki `p1` i `p2`. Funkcja `exchange` zamienia wartość wskazywaną przez `p1` na wartość wskazywaną przez `p2`. Skoro `p1` wskazuje `i1`, a `p2` wskazuje `i2`, funkcja zamienia wartości `i1` i `i2`. Widać to w wyniku drugiego wywołania funkcji `printf`.

Drugie wywołanie funkcji `exchange` jest ciekawsze. Tym razem parametry przekazywane tej funkcji, wskaźniki na `i1` i `i2`, są przygotowywane na miejscu, przez użycie operatora adresu. Wyrażenie `&i1` daje wskaźnik na zmienną `i1`, jest to wyrażenie zgodne z typem oczekiwanym przez funkcję. To samo dotyczy drugiego parametru. Jak widać w wynikach programu, funkcja `exchange` wykonuje swoje zadanie i zamienia z powrotem wartości `i1` i `i2`.

Trzeba zdać sobie sprawę, że bez wskaźników niemożliwe byłoby napisanie funkcji `exchange` zamieniającej wartości dwóch zmiennych, gdyż moglibyśmy z funkcji zwrócić tylko jedną wartość, a funkcja nie może trwale zmienić wartości swoich parametrów. Przeanalizujmy dokładnie program 10.9. Pokazuje on jedną z najważniejszych zasad użycia wskaźników w języku C.

Program 10.10 demonstruje, jak funkcja może zwracać wskaźnik. Program zawiera definicję funkcji `findEntry`, która przechodzi przez listę powiązaną w poszukiwaniu ustalonej wartości. Kiedy wartość ta zostanie znaleziona, program zwraca wskaźnik do odpowiedniego elementu listy. Jeśli wartość nie zostanie znaleziona, program zwróci wskaźnik pusty.

Program 10.10. **Zwracanie przez funkcję wskaźnika**

```
#include <stdio.h>

struct entry
{
    int  value;
    struct entry  *next;
};

struct entry *findEntry (struct entry *listPtr, int match)
{
    while ( listPtr != (struct entry *) 0 )
        if ( listPtr->value == match )
            return (listPtr);
        else
            listPtr = listPtr->next;

    return (struct entry *) 0;
}
```

```
int main (void)
{
    struct entry  *findEntry (struct entry *listPtr, int match);
    struct entry  n1, n2, n3;
    struct entry  *listPtr, *listStart = &n1;

    int search;

    n1.value = 100;
    n1.next = &n2;

    n2.value = 200;
    n2.next = &n3;

    n3.value = 300;
    n3.next = 0;

    printf ("Podaj szukaną wartość: ");
    scanf ("%i", &search);

    listPtr = findEntry (listStart, search);

    if ( listPtr != (struct entry *) 0 )
        printf ("Znaleziono %i.\n", listPtr->value);
    else
        printf ("Nie znaleziono.\n");

    return 0;
}
```

Program 10.10. **Wyniki**

```
Podaj szukaną wartość: 200
Znaleziono 200.
```

Program 10.10. **Wyniki (ponowne uruchomienie)**

```
Podaj szukaną wartość: 400
Nie znaleziono.
```

Program 10.10. **Wyniki (ponowne uruchomienie)**

```
Podaj szukaną wartość: 300
Znaleziono 300.
```

Z nagłówka funkcji:

```
struct entry  *findEntry (struct entry *listPtr, int match);
```

wynika, że `findEntry` zwraca wskaźnik na strukturę `entry` i pobiera taki wskaźnik jako pierwszy parametr. Drugim parametrem jest liczba całkowita. Funkcja zaczyna działanie od utworzenia pętli `while`, w której przeglądane są kolejne elementy listy. Pętla ta działa tak długo, aż wartość `match` zostanie znaleziona w polu `value` któregoś elementu listy

(wtedy zwracane jest `listPtr`) albo osiągnięty zostanie pusty wskaźnik (wtedy wychodzimy z pętli, zwracany jest pusty wskaźnik).

Po przygotowaniu listy, tak jak w poprzednim programie, funkcja `main` pyta użytkownika o szukaną wartość, następnie wywołuje funkcję `findEntry` ze wskaźnikiem na początek listy (parametr `listStart`) oraz wartością podaną przez użytkownika (parametr `search`). Funkcja `findEntry` zwraca wskaźnik na typ `struct entry`, który jest przypisywany do zmiennej `listPtr`. Jeśli `listPtr` nie jest pusty, pokazywane jest pole `value` struktury wskazywanej przez `listPtr`. Powinna to być taka sama wartość, jakiej szukał użytkownik. Jeśli wskaźnik `listPtr` jest pusty, pokazywany jest komunikat: „Nie znaleziono".

Wyniki działania programu pokazują, że na liście znaleziono prawidłowo wartości 200 i 300, a 400 już nie.

Wskaźnik zwracany przez funkcję `findEntry` nie wydaje się służyć niczemu sensownemu. Jednak w przypadkach bliższych praktycznym zastosowaniom wskaźnik taki pozwala dostać się do znalezionych elementów listy. Moglibyśmy na przykład użyć listy powiązanej w naszym słowniku z rozdziału 9. Wtedy funkcja `findEntry` (albo `lookup`, jak ją nazywaliśmy w poprzednim rozdziale) szukałaby na liście żądanego słowa. Zwracany wskaźnik pozwalałby sięgnąć do pola `definition` odpowiadającego znalezionemu hasłu.

Zorganizowanie słownika w formie listy powiązanej ma szereg zalet. Wstawianie do słownika nowego słowa staje się proste — wystarczy ustalić miejsce na liście, gdzie ma pojawić się nowe słowo i wystarczy ustawić kilka wskaźników, jak to pokazywaliśmy wcześniej w tym rozdziale. Równie proste jest usuwanie pozycji słownika. W końcu, jak dowiemy się w rozdziale 16., takie rozwiązanie pozwala skorzystać z dynamicznego powiększania słownika.

Jednak lista powiązana jako struktura danych słownika ma też poważną wadę — nie można w niej zrealizować szybkiego binarnego algorytmu wyszukiwania. Algorytm ten działa tylko na bezpośrednio indeksowanych tablicach. Niestety, nie istnieje szybsza metoda przeszukiwania list powiązanych niż tylko normalne przeglądanie element po elemencie. Jednym ze sposobów skorzystania z zalet prostego wstawiania i usuwania elementów, a jednocześnie szybkiego wyszukiwania, jest skorzystanie z innej struktury danych — *drzewa*. Istnieją też inne rozwiązania, takie jak tablice asocjacyjne. Czytelnicy zainteresowani tym tematem powinni skorzystać z istniejącego dorobku informatyki, na przykład z książki Donalda E. Knutha *The Art. of Computer Programming. Volume 3: Sorting and Searching* wydanej przez Addison-Wesley. Omówiono tam struktury danych, które pozwalają na wykorzystanie wspomnianych technik.

Wskaźniki i tablice

W języku C jednym z najpowszechniejszych zastosowań wskaźników jest wskazywanie tablic. Podstawowym powodem używania wskaźników na tablice jest wygoda zapisu i szybkość działania programów. Wskaźniki na tablice zwykle powodują zmniejszenie ilości kodu i ilości zajmowanej pamięci, dzięki czemu programy działają szybciej. Dlaczego tak jest, wyjaśnimy w tej części rozdziału.

Jeśli mamy tablicę 100 liczb całkowitych `values`, możemy zdefiniować wskaźnik `valuePtr`, którego używa się do korzystania z tych liczb:

```
int  *valuePtr;
```

Kiedy definiujemy wskaźnik sięgający do elementu tablicy, nie definiujemy go jako wskaźnika do tablicy, lecz jako wskaźnik do typu elementów przechowywanych w tej tablicy.

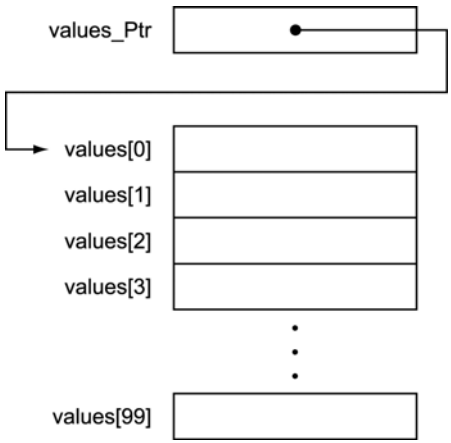
Jeśli mamy tablicę znakową `text`, analogicznie możemy zdefiniować wskaźnik dostępowy do elementów tej tablicy:

```
char *textPtr;
```

Aby zmienna `valuesPtr` wskazywała pierwszy element tablicy `values`, po prostu piszemy:

```
valuePtr = values;
```

W tej sytuacji nie używa się operatora adresu, gdyż kompilator C samą nazwę tablicy, bez indeksu, traktuje jako wskaźnik na tę tablicę. Wobec tego samo podanie `values` bez indeksu da wskaźnik do pierwszego elementu tej tablicy — co widać na rysunku 10.9.



Rysunek 10.9. Wskaźnik elementu tablicy

Równoważnym sposobem uzyskania wskaźnika na początek tablicy `values` jest zastosowanie operatora adresu do pierwszego elementu tej tablicy. Wobec tego instrukcja:

```
valuesPtr = &values[0];
```

może zastąpić przypisanie poprzednie.

Aby zmienna `textPtr` wskazywała pierwszy znak w tablicy `text`, używamy instrukcji:

```
textPtr = text;
```

lub

```
textPtr = &text[0];c
```

Wybór jednej z nich jest wyłącznie kwestią gustu programisty.

Prawdziwa siła wskaźników na tablice objawia się, kiedy chcemy przechodzić po elementach tablicy. Jeśli zdefiniowaliśmy już zmienną `valuesPtr` oraz ustawiliśmy ją, aby wskazywała pierwszy element tablicy, to wyrażenie:

```
*valuesPtr
```

pozwoli pozyskać pierwszy element tablicy `values`, czyli `values[0]`. Aby za pośrednictwem `valuesPtr` odwołać się do elementu `values[3]`, dodajemy do tego wskaźnika 3 i używamy operatora wyłuskania:

```
*(valuesPtr + 3)
```

W ogóle wyrażenie:

```
*(valuesPtr + i)
```

pozwoli sięgnąć do wartości `values[i]`.

Aby zatem ustawić `values[10]` na 27, możemy napisać:

```
values[10] = 27;
```

lub (korzystając z `valuesPtr`):

```
*(valuesPtr + 10) = 27;
```

Aby ustawić `valuesPtr` tak, by wskazywał drugi element tablicy `values`, możemy użyć operatora adresu do elementu `values[1]`, a wynik przypisać zmiennej `valuesPtr`:

```
valuesPtr = &values[1];
```

Jeśli `valuesPtr` wskazuje `values[0]`, możemy ustawić tę zmienną tak, aby wskazywała `values[1]` — wystarczy dodać do niej 1:

```
valuesPtr += 1;
```

Jest to zupełnie poprawne wyrażenie języka C, można go używać do wskaźników *dowolnego* typu.

Ogólnie rzecz biorąc, jeśli `a` jest tablicą elementów typu `x`, `px` jest wskaźnikiem na typ `x`, a `i` i `n` to stałe lub zmienne całkowite, wtedy instrukcja:

```
px = a;
```

powoduje ustawienie `px` na pierwszy element tablicy `a`, z kolei wyrażenie:

```
*(px + i)
```

powoduje odwołanie się do wartości z `a[i]`. Dalej, instrukcja:

```
px += n;
```

ustawia `px` o `n` elementów dalej w tablicy, *niezależnie od typu elementów tablicy*.

Przy obsłudze wskaźników wyjątkowo przydatne są operatory inkrementacji i dekrementacji — `++` i `—`. Zastosowanie operatora inkrementacji do wskaźnika powoduje, że do wskaźnika dodawane jest 1; analogicznie działa operator dekrementacji,

ale z odejmowaniem. Jeśli zatem zmienną `textPtr` zdefiniowano jako wskaźnik typu `char`, który wskazuje początek tablicy `text` zawierającej dane `char`, instrukcja:

```
++textPtr;
```

ustawia ten wskaźnik na następny znak z `text`, czyli `text[1]`. Analogicznie instrukcja:

```
—textPtr;
```

powoduje cofnięcie wskaźnika `textPtr` na poprzedni element tablicy `text` — oczywiście, jeśli wcześniej `textPtr` nie wskazywał początku tablicy `text`.

W języku C jak najbardziej dopuszczalne jest porównywanie wskaźników. Jest to szczególnie przydatne do porównywania dwóch wskaźników z tej samej tablicy. Jeśli na przykład chcemy sprawdzić, czy wskaźnik `valuesPtr` wskazuje poza koniec 100-elementowej tablicy, porównujemy wskaźnik z ostatnim elementem tablicy. Zatem wyrażenie:

```
valuesPtr > &values[99]
```

jest prawdziwe (niezerowe), jeśli `valuesPtr` wskazuje za ostatnim elementem tablicy `values`. Przypomnijmy, że takie wyrażenie można zastąpić wyrażeniem:

```
valuesPtr > values + 99
```

Wynika to stąd, że `values` bez żadnego indeksu wskazuje początek tablicy `values` (zapisywany też jako `&values[0]`).

Program 10.11 pokazuje wskaźniki na tablice. Funkcja `arraySum` wylicza sumę elementów znajdujących się w tablicy liczb całkowitych.

Program 10.11. **Użycie wskaźników na tablice**

```
// Funkcja sumuje elementy tablicy liczb całkowitych

#include <stdio.h>

int arraySum (int array[], const int n)
{
    int  sum = 0, *ptr;
    int  * const arrayEnd = array + n;

    for ( ptr = array;  ptr < arrayEnd;  ++ptr)
        sum += *ptr;

    return sum;
}

int main (void)
{
    int  arraySum (int array[], const int n);
    int  values[10] = {3, 7, —9, 3, 6, —1, 7, 9, 1, —5};

    printf ("Suma to %i\n", araySum (values, 10));

    return 0;
}
```

Program 10.11. **Wyniki**

```
Suma to 21
```

W funkcji `arraySum` definiujemy stały wskaźnik na liczbę całkowitą i nadajemy mu taką wartość, aby wskazywał tuż za ostatni element tablicy `array`. Następnie przygotowujemy pętlę `for`, która przechodzi po kolejnych elementach tej tablicy. Wartość `ptr` jest ustawiana tak, aby przy wejściu do pętli wskazywać początek tablicy `array`. W każdym przejściu tablicy element tablicy `array` wskazywany przez `ptr` jest dodawany do wartości `sum`. Wartość `ptr` jest zwiększana w pętli, aby wskazywała następny element tablicy. Kiedy `ptr` wskaże poza koniec tablicy, działanie pętli `for` kończy się, a wartość `sum` jest zwracana do miejsca wywołania naszej funkcji.

Parę słów o optymalizacji programu

Powiedzieliśmy, że zmienna lokalna `arrayEnd` tak naprawdę nie jest potrzebna, gdyż w pętli `for` moglibyśmy jawnie porównywać wartość `ptr` z końcem tablicy `array`:

```
for ( ...; pointer <= array + n; ... )
```

Jedynym powodem użycia `arrayEnd` jest optymalizacja. Przy każdym przejściu pętli `for` badany jest warunek pętli. Wyrażenie `array+n` nie zmienia się w całej pętli, więc jego wartość jest stała. Skoro wyliczamy tę wartość *przed* wejściem do pętli, oszczędzamy czas, który w innym wypadku byłby zużyty na wyliczenie wartości podanego wyrażenia w każdym przejściu pętli. Choć w przypadku tablicy 10-elementowej oszczędności są praktycznie zerowe, szczególnie jeśli funkcja `arraySum` w całym programie jest tylko raz wywoływana, to oszczędności mogą być znacznie większe, kiedy funkcja będzie intensywnie używana na dużych tablicach.

Inne zagadnienie z dziedziny optymalizacji dotyczy samych wskaźników. W omawianej wcześniej funkcji `arraySum` wyrażenie `*ptr` jest używane w pętli `for` do sięgania do elementów tablicy. Wcześniej w funkcji `arraySum` użylibyśmy po prostu zmiennej indeksowej, na przykład `i`, a do sumy `sum` dodawalibyśmy kolejne wartości `array[i]`. Normalnie indeksowanie tablic działa wolniej od sięgania do elementów przy użyciu wskaźników. To właśnie jest jeden z najważniejszych powodów, dla których używa się wskaźników do obsługi elementów tablicy — uzyskany kod działa szybciej. Oczywiście, jeśli do elementów tablicy raczej nie sięga się kolejno, wskaźniki niewiele dadzą, gdyż obliczenie wyrażenia `*(pointer+j)` zajmuje tyle samo czasu co obliczenie wyrażenia `array[j]`.

To tablica czy wskaźnik?

Przypomnijmy — aby do funkcji przekazać tablicę, po prostu podajemy nazwę tej tablicy, jak to zrobiliśmy w funkcji `arraySum`. Trzeba też pamiętać, że wystarczy podać nazwę tablicy, aby uzyskać wskaźnik na nią. Wynika stąd, że w wywołaniu funkcji `arraySum` tak naprawdę przekazujemy do niej *wskaźnik* na tablicę `values`. Wyjaśnia to, dlaczego mogliśmy zmieniać w funkcji wartości poszczególnych elementów tablicy.

Jeśli jednak do funkcji przekazujemy wskaźnik na tablicę, to dlaczego parametry formalne nie są deklarowane jako wskaźniki? Czyli czemu deklaracja tablicy `array` w funkcji `arraySum` nie wygląda następująco:

```
int *array;
```

Czy wszystkie odwołania do tablic w funkcjach nie powinny być wykonywane przez zmienne wskaźnikowe?

Przypomnijmy sobie poprzednią dyskusję o wskaźnikach i tablicach. Jak wspomnieliśmy, jeśli `valuesPtr` wskazuje element tego samego typu, jakiego są elementy tablicy `values`, wyrażenie `*(valuesPtr+i)` jest pod każdym względem równoważne wyrażeniu `values[i]`, jeśli tylko `valuePtr` wskazuje początek tablicy `values`. Wobec tego za pomocą wyrażenia `*(values+i)` możemy odwołać się do `i`-tego elementu tablicy `values`. Ogólnie rzecz biorąc, jeśli `x` jest tablicą dowolnego typu, wyrażenie `x[i]` jest równoważne wyrażeniu `*(x+i)`.

Jak widać, w języku C wskaźniki i tablice są ze sobą ściśle powiązane, dlatego właśnie w funkcji `arraySum` deklarujemy zmienną `array` jako „tablicę wartości typu `int`" *lub* jako „wskaźnik na zmienną typu `int`". Obie deklaracje zadziałają w poprzednim programie równie dobrze — można to sprawdzić.

Jeśli ktoś chce odwoływać się do elementów tablicy przekazanej do funkcji, odpowiedni parametr formalny należy zadeklarować jako tablicę. Jeśli ktoś woli używać wskaźników, ten sam parametr należy zadeklarować jako wskaźnik.

Teraz, kiedy zadeklarowaliśmy tablicę `array` jako wskaźnik danych typu `int`, możemy używać wskaźników, eliminując z funkcji zmienną `ptr` i wykorzystując zamiast niej `array`, tak jak w programie 10.12.

Program 10.12. **Sumowanie elementów z tablicy**

```
// Funkcja sumuje elementy tablicy liczb całkowitych, 2. wersja

#include <stdio.h>

int arraySum (int *array, const int n)
{
    int  sum = 0;
    int  * const arrayEnd = array + n;

    for ( ; array < arrayEnd;  ++array )
        sum += *array;

    return sum;
}

int main (void)
{
    int  arraySum (int *array, const int n);
    int  values[10] = {3, 7, -9, 3, 6, -1, 7, 9, 1, -5};

    printf ("Suma to %i\n", araySum (values, 10));

    return 0;
}
```

Program 10.12. **Wyniki**

```
Suma to 21
```

Powyższy program właściwie nie wymaga dodatkowych wyjaśnień. Pierwsze wyrażenie w pętli `for` zostało pominięte, gdyż zaczynając pętlę, nie inicjalizujemy żadnych zmiennych. Warto tylko powtórzyć, że przy wywoływaniu funkcji `arraySum` przekazujemy wskaźnik do tablicy `values`, a w samej funkcji nazywamy go `array`. Zmiana wartości `array` (w przeciwieństwie do zmian wartości elementów `array`) nie wypływa na tablicę `values`. Wobec tego bezpośrednio zwiększamy wartość wskaźnika `array`, gdyż i tak nie ma to wpływu na samą tablicę. (Pamiętamy przy tym, oczywiście, że wystarczy zmienić wartość elementu wskazywanego przez wskaźnik, aby zmienić wartość elementów w tablicy).

Wskaźniki na łańcuchy znakowe

Jednym z najpowszechniejszych zastosowań tablic są wskaźniki na łańcuchy znakowe. Powodem jest wygoda zapisu i szybkość działania. Aby pokazać, jak łatwo użyć wskaźników na łańcuchy znakowe, napiszemy funkcję `copyString` kopiującą jeden łańcuch na drugi. Jeśli napiszemy tę funkcję normalnie, z indeksowaniem tablic, może ona mieć na przykład taką postać:

```
void copyString (char to[], char from[])
{
    int i;

    for ( i = 0; from[i] != '\0'; ++i )
        to[i] = from[i];

    to[i] = '\0';
}
```

Z pętli `for` wychodzimy jeszcze przed skopiowaniem znaku *null* do tablicy `to`, stąd obecność ostatniej instrukcji naszej funkcji.

Jeśli zapiszemy `copyString`, korzystając ze wskaźników, zmienna indeksowa `i` przestaje być potrzebna. Wskaźnikową wersję tej samej funkcji pokazano w programie 10.13.

Program 10.13. **Wskaźnikowa wersja funkcji copyString**

```
#include <stdio.h>

void copyString (char *to, char *from)
{
    for ( ; *from != '\0'; ++from, ++to )
        *to = *from;

    *to = '\0';
}

int main (void)
{
```

```
    void  copyString (char *to, char *from);
    char  string1[] = "Łańcuch do kopiowania.";
    char  string2[50];

    copySstring (string2, string1);
    printf ("%s\n", string2);

    copyString (string2, "Ten także.");
    printf ("%s\n", string2);

    return 0;
}
```

Program 10.13. **Wyniki**

```
Łańcuch do kopiowania.
Ten także.
```

Funkcja `copyString` ma dwa parametry formalne, `to` i `from`, oba zadeklarowane jako wskaźniki na znaki, a nie jako tablice znakowe, jak to robiliśmy wcześniej. Odzwierciedla to sposób użycia w kodzie tych dwóch zmiennych.

Po wejściu do pętli `for` (bez warunków początkowych) zaczynamy kopiowanie łańcucha wskazywanego przez `from` na łańcuch wskazywany przez `to`. W każdym przejściu pętli wskaźniki `from` i `to` są zwiększane o 1. Powoduje to ustawienie wskaźnika `from` na następny znak, który będzie kopiowany, a wskaźnika `to` na miejsce, w które zostanie wstawiony następny znak.

Kiedy wskaźnik `from` wskaże znak *null*, wychodzimy z pętli `for`, a funkcja w łańcuchu docelowym dodaje jeszcze kończący znak *null*.

W funkcji `main` funkcja `copyString` jest wywoływana dwukrotnie: raz w celu skopiowania łańcucha `string1` do `string2`, następnie w celu skopiowania łańcucha stałego `"Ten także."` do łańcucha `string2`.

Stałe łańcuchy znakowe a wskaźniki

Z faktu, że wywołanie:

```
copyString (string2, "Ten także.");
```

w ogóle zadziałało, wynika, że kiedy jako parametr funkcji przekazujemy łańcuch stały, to tak naprawdę przekazujemy wskaźnik do tego łańcucha. Jest to prawdą w każdej sytuacji — jeśli tylko w języku C używamy stałego łańcucha znakowego, tworzony jest wskaźnik na ten łańcuch. Jeżeli zatem zadeklarujemy `textPtr` jako łańcuch znakowy:

```
char  *textPtr;
```

to instrukcja:

```
textPtr = "To jest łańcuch znakowy.";
```

powoduje przypisanie zmiennej `textPtr` *wskaźnika* na stały łańcuch znakowy `"To jest łańcuch znakowy."`. Trzeba odróżnić wskaźniki znakowe i tablice znakowe, gdyż pokazane przypisanie w przypadku tablic znakowych jest *nieprawidłowe*. Jeśli zatem na przykład zdefiniowalibyśmy zmienną `text` jako tablicę danych typu `char`:

```
char  text[80];
```

to już *nie moglibyśmy* użyć instrukcji:

```
text = "Tak nie można.";
```

Jedyna sytuacja, kiedy język C pozwala wykonać tego typu przypisanie, to inicjalizacja tablicy znakowej:

```
char  text[80] = "Teraz jest dobrze.";
```

Takie inicjalizowanie tablicy `text` nie powoduje zapisania w zmiennej `text` wskaźnika do łańcucha `"Teraz jest dobrze."`, ale powoduje wstawienie do tablicy `text` poszczególnych znaków tego łańcucha.

Jeśli `text` jest tablicą wskaźników na znaki, inicjalizacja `text` za pomocą instrukcji:

```
char  *text = "Teraz jest dobrze.";
```

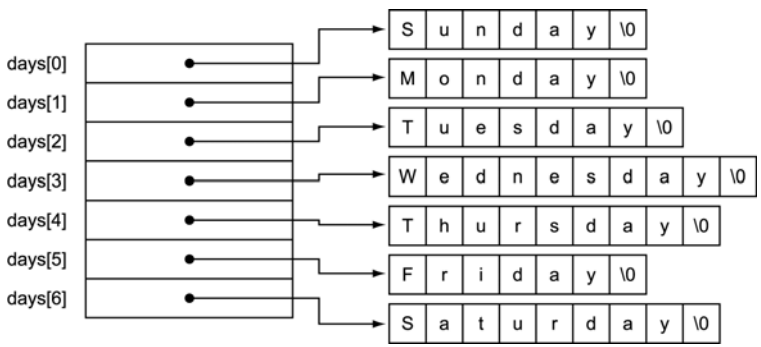
powoduje przypisanie wskaźnika na łańcuch znakowy `"Teraz jest dobrze."`.

Następny przykład pozwalający dostrzec różnice między łańcuchami znakowymi a wskaźnikami na łańcuchy znakowe to tablica `days`, zawierająca *wskaźniki* do angielskich nazw dni tygodnia.

```
char *days[] =
    { "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
      "Saturday" };
```

Tablica ta ma siedem elementów, z których każdy jest wskaźnikiem na łańcuch znakowy. Element `days[0]` zawiera wskaźnik na łańcuch `"Sunday"`, `days[1]` zawiera wskaźnik na łańcuch `"Monday"` i tak dalej (patrz rysunek 10.10). Poniższa instrukcja pozwala wyświetlić przykładowo nazwę trzeciego dnia tygodnia:

```
printf ("%s\n", days[3]);
```



Rysunek 10.10. Tablica wskaźników

Jeszcze raz o inkrementacji i dekrementacji

Aż do teraz, kiedy mówiliśmy o operatorach inkrementacji i dekrementacji, były to jedyne operatory występujące w wyrażeniu. Kiedy piszemy ++x, wiemy, że do zmiennej x dodana zostanie liczba 1. Jak już widzieliśmy, jeśli x jest wskaźnikiem na tablicę, powoduje to z kolei ustawienie x na następny element tej tablicy.

Operatory inkrementacji i dekrementacji mogą być używane w wyrażeniach zawierających także inne operatory. Wtedy ważna staje się dokładniejsza znajomość sposobu ich działania.

Jak dotąd operatory inkrementacji i dekrementacji zawsze umieszczaliśmy *przed* modyfikowanymi zmiennymi. Wobec tego, aby zwiększyć wartość zmiennej i, pisaliśmy:

```
++i;
```

Tak naprawdę operator inkrementacji można równie dobrze umieścić *po* zmiennej:

```
i++;
```

Oba wyrażenia są poprawne i dają taki sam efekt — zwiększają wartość zmiennej i. W pierwszym wypadku operator został umieszczony przed swoim argumentem; jest to operator *przedrostkowy*, w drugim — za nim; jest to operator *przyrostkowy*.

To samo dotyczy operatora dekrementacji. Instrukcja:

```
—i;
```

działa tak samo jak:

```
i—;
```

i odejmuje 1 od zmiennej i.

Różnica między operatorami przedrostkowymi i przyrostkowymi ujawnia się, kiedy mamy do czynienia z bardziej złożonymi wyrażeniami.

Załóżmy, że mamy dwie zmienne całkowite i i j. Jeśli ustawimy i na 0, a potem napiszemy:

```
j = ++i;
```

wartością przypisaną do j będzie 1. Zmienna jest inkrementowana jeszcze *przed* użyciem jej wartości w wyrażeniu. Wobec tego w powyższym wyrażeniu wartość zmiennej i jest zwiększana z 0 na 1, następnie jest przypisywana zmiennej j tak, jakbyśmy użyli dwóch następujących instrukcji:

```
++i;
j = i;
```

Jeśli użyjemy operatora przyrostkowego:

```
j = i++;
```

inkrementacja i będzie miała miejsce już *po* przypisaniu wartości tej zmiennej do j. Jeśli zatem przed wywołaniem powyższej instrukcji zmienna i będzie miała wartość 0,

do j zostanie przypisane 0, a dopiero wtedy i zostanie zwiększone o 1. Odpowiadają temu instrukcje:

```
j = i;
++i;
```

Oto następny przykład. Jeśli i jest równe 1, instrukcja:

```
x = a[—i];
```

spowoduje przypisanie zmiennej x wartości a[0], gdyż zmienna i jest dekrementowana przed użyciem jej jako indeksu tablicy a. Instrukcja:

```
x = a[i—];
```

z kolei spowodowałaby przypisanie zmiennej x wartości a[1], gdyż i byłoby zmniejszone po użyciu tej zmiennej jako indeksu tablicy a.

Trzeci przykład pokazujący różnicę między operatorami przedrostkowymi i przyrostkowymi wiąże się z wywołaniem funkcji printf:

```
printf ("%i\n", ++i);
```

Wartość zmiennej i zostanie powiększona, a wtedy dopiero wartość zostanie przekazana do printf. Gdyby użyć zapisu:

```
printf ("%i\n", i++);
```

wartość zmiennej i zostałaby zwiększona po wysłaniu jej do funkcji printf. Wobec tego, jeśli i jest równe 100, pierwsza funkcja printf wyświetli 101, a druga 100. W obu przypadkach wartość i po tej instrukcji będzie równa już 101.

Ostatni przykład — jeśli textPtr jest wskaźnikiem znakowym, wyrażenie:

```
*(++textPtr)
```

najpierw zwiększy wartość textPtr, następnie pobierze wskazywany znak, z kolei wyrażenie:

```
*(textPtr++)
```

pobierze znak wskazywany przez textPtr, a potem przesunie wskaźnik. W obu wypadkach nawiasy są zbędne, gdyż operatory * i ++ mają takie same priorytety, ale są łączne od prawej do lewej.

Teraz wróćmy do funkcji copyString z programu 10.13 i przepiszmy ją tak, aby inkrementację włączyć bezpośrednio do instrukcji przypisania.

Wskaźniki to i from są inkrementowane po każdym przypisaniu w pętli for, możemy te inkrementacje włączyć do instrukcji przypisania, stosując formę przyrostkową operatorów. Poprawiona wersja pętli for z programu 10.13 przybiera postać:

```
for ( ; *from != '\0'; )
    *to++ = *from++;
```

Wykonanie przypisania w pętli odbywa się następująco. Znak wskazywany przez from jest pobierany, następnie wskaźnik from jest przesuwany na następny znak łańcucha

źródłowego. Odczytany znak jest umieszczany w miejscu wskazanym przez to, po czym wskaźnik to jest przesuwany na następny element łańcucha docelowego.

Powyższą instrukcję przypisania trzeba tak dokładnie przemyśleć, aż nie będzie żadnych wątpliwości dotyczących jej działania. Instrukcje tego typu są powszechnie stosowane w programach w języku C, dlatego trzeba je doskonale rozumieć.

Powyższa instrukcja for staje się coraz bardziej ułomna, gdyż nie ma wyrażenia początkowego ani wyrażenia pętli. Tak naprawdę w tym wypadku lepiej sprawdziłaby się już pętla while; zmianę taką naniesiono w programie 10.14. Program ten zawiera kolejną wersję funkcji copyString. Pętla while wykorzystuje fakt, że znak *null* ma wartość 0; jest to powszechna praktyka doświadczonych programistów języka C.

Program 10.14. **Nowa wersja funkcji copyString**

```
// Funkcja kopiująca łańcuch na inny, 2. wersja wskaźnikowa

#include <stdio.h>

void copyString (char *to, char *from)
{
    while ( *from )
        *to++ = *from++;

    *to = '\0';
}

int main (void)
{
    void  copyString (char *to, char *from);
    char  string1[] = "Łańcuch do kopiowania.";
    char  string2[50];

    copyString (string2, string1);
    printf ("%s\n", string2);

    copyString (string2, "Ten także.");
    printf ("%s\n", string2);

    return 0;
}
```

Program 10.14. **Wyniki**

```
Łańcuch do kopiowania.
Ten także.
```

Operacje na wskaźnikach

Jak widzieliśmy, liczby całkowite można dodawać do wskaźników lub je od nich odejmować. Możemy porównywać dwa wskaźniki, sprawdzając, czy są sobie równe lub czy jeden jest większy od drugiego. Jedyne działanie, jakie można wykonywać

na dwóch wskaźnikach, to odjęcie dwóch wskaźników tego samego typu. W języku C wynikiem odejmowania dwóch wskaźników jest liczba elementów zawartych między tymi wskaźnikami. Jeśli zatem mamy dwa wskaźniki na różne elementy jednej tablicy, wyrażenie `b–a` reprezentuje liczbę elementów między tymi dwoma wskaźnikami. Jeśli na przykład `p` wskazuje jakiś element w tablicy `x`, instrukcja:

```
n = p – x;
```

spowoduje przypisanie zmiennej `n` (jest to zmienna całkowitoliczbowa) indeksu elementu tablicy `x` wskazywanego przez `p`[6]. Jeśli zatem `p` wskazuje setny element tablicy `x`:

```
p = &x[99];
```

to po wykonaniu powyższego odejmowania otrzymamy wartość `n` równą `99`.

Praktycznym zastosowaniem podanych tu informacji o odejmowaniu wskaźników jest modyfikacja funkcji `stringLength` z rozdziału 9.

W programie 10.15 wskaźnik znakowy `cptr` służy do przeglądania znaków ze `string` tak długo, aż osiągnięty zostanie znak *null*. Wtedy od `cptr` odejmowany jest wskaźnik `string`, co daje liczbę elementów (tu: znaków) w łańcuchu. Wyniki działania programu pokazują, że funkcja działa poprawnie.

Program 10.15. **Użycie wskaźników do określania długości łańcucha**

```
// Funkcja zliczająca znaki w łańcuchu – wersja wskaźnikowa

#include <stdio.h>

int  stringLength (const char  *string)
{
    const char *cptr = string;

    while ( *cptr )
        ++cptr;
    return cptr – string;
}

int main (void)
{
    int  stringLength (const char *string);

    printf ("%i  ", stringLength ("test stringLength"));
    printf ("%i  ", stringLength (""));
    printf ("%i\n", stringLength ("gotowe"));

    return 0;
}
```

[6] Faktyczny typ liczby całkowitej ze znakiem zwracanej jako wynik odejmowania dwóch wskaźników (na przykład `int`, `long int` lub `long long int`) to `ptrdiff_t` zdefiniowany w standardowym pliku nagłówkowym `<stddef.h>`.

Program 10.15. **Wyniki**

```
17  0  8
```

Wskaźniki na funkcje

Aby nasz wykład był kompletny, powiemy jeszcze o pewnym, nieco trudniejszym zagadnieniu — wskaźnikach na funkcje. Kiedy używa się wskaźników na funkcje, kompilator C musi znać nie tylko zmienną wskaźnikową wskazującą funkcję, lecz także typ wartości zwracanej przez funkcję oraz liczbę i typy parametrów tej funkcji. Do zadeklarowania zmiennej `fnPtr` jako „wskaźnika na bezparametrową funkcję zwracającą wartość `int`" używamy zapisu:

```
int  (*fnPtr) (void);
```

Nawiasy otaczające wyrażenie `*fnPtr` są niezbędne, gdyż w przeciwnym razie kompilator C potraktowałby taką deklarację jako deklarację funkcji `fnPtr` zwracającej wskaźnik na daną `int` (operator wywołania funkcji — `()` — ma wyższy priorytet niż operator wyłuskania — `*`).

Aby nasza zmienna wskazywała jakąś konkretną funkcję, po prostu przypisujemy jej nazwę funkcji. Jeśli zatem chcemy wskazać funkcję `lookup` zwracającą liczbę `int` i niemającą parametrów, używamy instrukcji:

```
fnPtr = lookup;
```

Zapisanie nazwy funkcji bez towarzyszących jej nawiasów jest traktowane analogicznie jak podanie nazwy tablicy bez indeksów. Kompilator języka C automatycznie tworzy wskaźnik do funkcji o podanej nazwie. Przed nazwą funkcji może wystąpić ampersand — `&` —, ale nie jest to obowiązkowe.

Jeśli w programie nie zdefiniowano wcześniej funkcji `lookup`, trzeba ją zadeklarować przed powyższym przypisaniem. Jeśli zatem wcześniej pojawi się instrukcja:

```
int  lookup (void);
```

możemy już dalej przypisać funkcję `lookup` zmiennej `fnPtr`.

Można wywołać funkcję pośrednio wskazywaną przez zmienną wskaźnikową; wystarczy zastosować operator wywołania funkcji do wskaźnika i podać w nawiasach liczbę parametrów, na przykład:

```
entry = fnPtr ();
```

wywołuje funkcję wskazywaną przez `fnPtr`, zwróconą przez nią wartość wstawia do zmiennej `entry`.

Typowym zastosowaniem wskaźników na funkcje jest przekazywanie ich jako parametrów do innych funkcji. Wykorzystywane jest to w standardowej bibliotece języka C, na przykład w funkcji `qsort` realizującej algorytm *quicksort* na tablicy. Funkcja ta jako jeden z parametrów pobiera wskaźnik funkcji wywoływanej przez `qsort`, kiedy trzeba porównać

dwa elementy sortowanej tablicy. W ten sposób `qsort` może służyć do sortowania tablic dowolnego typu, gdyż samo porównywanie elementów wykonywane jest w funkcji użytkownika, a nie w samej funkcji `qsort`. W dodatku B poświęconym standardowej bibliotece języka C omawiamy funkcję `qsort` nieco dokładniej i pokazujemy przykład jej użycia.

Innym typowym zastosowaniem wskaźników na funkcje są tablice rozkładu. Samych funkcji nie można zapisywać w elementach tablicy, ale można tam zapisywać wskaźniki na funkcje. Wiedząc o tym, możemy tworzyć tablice zawierające wskaźniki na funkcje, które będziemy potem wywoływać. Możemy na przykład utworzyć tablicę z funkcjami przetwarzającymi różne polecenia podawane przez użytkownika. Każda pozycja takiej tablicy będzie zawierała nazwę polecenia oraz wskaźnik funkcji wywoływanej w celu obsługi tego polecenia. Teraz, kiedy użytkownik poda polecenie, możemy odszukać je w tablicy i wywołać odpowiadającą mu funkcję.

Wskaźniki a adresy w pamięci

Zanim skończymy omawianie wskaźników w języku C, powinniśmy powiedzieć nieco o sposobie ich realizacji. Pamięć komputera można wyobrażać sobie jako liniowy zbiór komórek. Każda komórka pamięci komputera ma numer nazywany *adresem*. Zwykle pierwszy adres w pamięci komputera to 0. W większości systemów komputerowych „komórki" nazywa się *bajtami*.

Komputer korzysta z pamięci do zapisywania instrukcji programu oraz wartości zmiennych związanych z tym programem. Jeśli zatem zadeklarujemy zmienną `count` typu `int`, system przypisze działającemu programowi miejsce w pamięci komputera na wartość zmiennej `count`.

Na szczęście jedną z zalet języków wysokiego poziomu, takich jak język C, jest to, że nie trzeba samemu walczyć z adresami pamięci przypisanymi zmiennym — tym zajmuje się system. Jednak świadomość, że z każdą zmienną związany jest niepowtarzalny adres w pamięci, znacznie ułatwia zrozumienie sposobu działania wskaźników.

Kiedy do zmiennej zastosujemy operator adresu, uzyskiwana wartość to faktyczny adres zmiennej w pamięci komputera (i stąd nazwa operatora adresu). Zatem instrukcja:

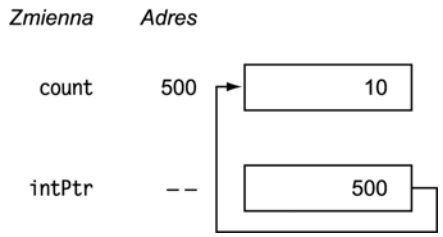
```
intPtr = &count;
```

przypisuje zmiennej `intPtr` adres pamięci przydzielony zmiennej `count`. Jeśli zmienna `count` znajduje się pod adresem 500 i ma wartość 10, instrukcja ta przypisze zmiennej `intPtr` wartość 500, a wskazywaną wartością będzie 10. Pokazano to na rysunku 10.11.

Adres zmiennej `intPtr` na rysunku 10.11 oznaczono jako —, gdyż nie jest on tu istotny.

Zastosowanie do zmiennej wskaźnikowej operatora wyłuskania, jak w wyrażeniu:

```
*intPtr
```



Rysunek 10.11. Wskaźniki i adresy w pamięci

powoduje potraktowanie wartości zmiennej jako adresu. Wartość zapisana pod podanym adresem jest pobierana i interpretowana zgodnie z zadeklarowanym typem zmiennej wskaźnikowej. Jeśli zatem `intPtr` jest wskaźnikiem na `int`, wartość zapisana w pamięci pod adresem `*intPtr` jest interpretowana jako liczba całkowita. Wynikiem całego wyrażenia jest wartość 10 typu `int`.

Analogicznie odbywa się zapisywanie wartości w miejscu wskazywanym wskaźnikiem, na przykład:

```
*intPtr = 20;
```

Pobierana jest zawartość zmiennej `intPtr`, po czym zmienna jest traktowana jako adres w pamięci. Następnie podana wartość jest zapisywana w uzyskanym adresie. W powyższej instrukcji zatem wartość 20 jest zapisywana w pamięci o adresie 500.

Czasami programiści systemowi muszą skorzystać z pamięci o ściśle określonym adresie. Wtedy przydaje się opisany wyżej sposób działania zmiennych wskaźnikowych.

Jak widać, wskaźniki są naprawdę użytecznymi konstrukcjami języka C. Elastyczność definiowania zmiennych wskaźnikowych jest znacznie większa, niż mogliśmy to opisać w tym rozdziale; można na przykład definiować wskaźnik do wskaźnika, a nawet wskaźnik do wskaźnika od wskaźnika. Tego typu konstrukcje przekraczają zakres naszej książki, choć są one jedynie logiczną konsekwencją tego, co o wskaźnikach już wiemy.

Wskaźniki to zwykle najtrudniejsze zagadnienia dla nowicjuszy. Jeśli cokolwiek jest niejasne, należy wrócić do odpowiednich części tego rozdziału i ponownie je przeczytać. Rozwiązanie załączonych ćwiczeń dodatkowo pomoże w zrozumieniu materiału.

Ćwiczenia

1. Przepisz i uruchom piętnaście programów pokazanych w tym rozdziale. Uzyskane wyniki porównaj z wynikami pokazanymi w tekście.
2. Napisz funkcję `insertEntry` wstawiającą na listę powiązaną nowy element. Niech ta funkcja ma jako parametry wstawiane hasło (typu `struct entry`, zgodnie z definicją z tego rozdziału) oraz wskaźnik elementu listy, po którym ma być wstawiony nowy element.

3. Funkcja z ćwiczenia 2. po prostu wstawia element po elemencie istniejącym, wobec czego nie można wstawić nowego elementu na początek listy, przed wszystkimi innymi elementami. Jak można zmodyfikować tę samą funkcję, aby uniknąć opisanego problemu?

   (Podpowiedź! Warto rozważyć przygotowanie specjalnej struktury, wskazującej na początek listy).

4. Napisz funkcję `removeEntry` usuwającą element z listy powiązanej. Jedynym parametrem tej funkcji powinien być wskaźnik listy. Niech funkcja usuwa element znajdujący się za elementem wskazywanym przez parametr (czemu nie można usuwać elementu wskazywanego przez parametr?). Potrzebna będzie specjalna struktura z ćwiczenia 3., która pozwoli usuwać pierwszy element.

5. *Lista podwójnie powiązana* to lista, której elementy zawierają wskaźniki do poprzedniego i następnego elementu listy. Zdefiniuj strukturę opisującą listę podwójnie powiązaną i napisz niewielki program realizujący taką listę, a także pokazujący jej elementy.

6. Utwórz funkcje `insertEntry` i `removeEntry` dla list podwójnie powiązanych. Funkcje te mają być podobne do funkcji z poprzednich ćwiczeń, obsługujących zwykłe listy. Czemu teraz funkcja `removeEntry` może jako parametr mieć element usuwany bezpośrednio?

7. Napisz wskaźnikową wersję funkcji `sort` z rozdziału 7. Zapewnij, aby funkcja używała tylko wskaźników — także do indeksowania chodzenia po pętli.

8. Napisz funkcję `sort3` sortującą rosnąco trzy liczby całkowite (funkcję zaimplementuj bez użycia tablic).

9. Napisz ponownie funkcję `readLine` z rozdziału 9. tak, aby korzystała ze wskaźnika znakowego, a nie z tablicy.

10. Napisz ponownie funkcję `compareStrings` z rozdziału 9., skorzystaj ze wskaźników zamiast tablic.

11. Mając definicję struktury `date`, taką jak w niniejszym rozdziale, napisz funkcję `dateUpdate` pobierającą jako parametr wskaźnik do struktury `date` i aktualizującą tę strukturę na następny dzień (zobacz program 8.4).

12. Jeśli dane są następujące deklaracje:

```
char  *message = "Programowanie w C to niezła zabawa\n";
char  message2[] = "I kto to mówi\n";
char  *format = "x = %i\n";
int x = 100;
```

    określ, czy każde wywołanie funkcji `printf` z poniższego zbioru jest poprawne i da takie same wyniki jak inne wywołania z tego zbioru:

```
/*** zbiór 1 ***/
printf ("Programowanie w C to niezła zabawa\n");
printf ("%s", "Programowanie w C to niezła zabawa\n");
printf ("%s", message);
printf (message);

/*** zbiór 2 ***/
printf ("I kto to mówi\n");
printf ("%s", message2);
printf (message2);
printf ("%s", &message2[0]);

/*** zbiór 3 ***/
printf ("kto to mówi\n");
printf (message2 + 2);
printf ("%s", message2 + 2);
printf ("%s", &message2[2]);

/*** zbiór 4 ***/
printf ("x = %i\n", x);
printf (format, x);
```

11

Operacje bitowe

Jak wspominaliśmy wcześniej, język C tworzono z myślą o programowaniu systemowym. Jest to doskonale widoczne w przypadku wskaźników, które pozwalają programiście na bardzo ścisłą kontrolę nad pamięcią komputera. Tak samo programiści systemowi często muszą działać na poziomie pojedynczych bitów poszczególnych słów maszynowych. W tym rozdziale nauczysz się wykonywania działań na poszczególnych bitach przy użyciu takich narzędzi języka C jak:

- bitowy operator `AND`,
- bitowy operator `OR`,
- bitowy operator `OR` wykluczające,
- operator negacji bitowej,
- operator przesunięcia w lewo,
- operator przesunięcia w prawo,
- pola bitowe.

Podstawowe wiadomości o bitach

Przypomnijmy sobie pojęcie *bajta*, om`awiane w poprzednim rozdziale. W większości systemów bajt składa się z ośmiu mniejszych jednostek zwanych *bitami*. Bit może przybierać jedną z dwóch wartości — 1 lub 0. Wobec tego na przykład bajt zapisany w pamięci pod adresem 1000 można wyrazić jako łańcuch ośmiu cyfr binarnych (dwójkowych):

```
01100100
```

Skrajny prawy bit w bajcie to bit *najmniej znaczący*, zaś bit skrajny z lewej strony to bit *najbardziej znaczący*. Jeśli łańcuch bitów potraktujemy jako liczbę całkowitą, skrajny prawy bit oznacza 2^0, czyli 1, bit sąsiadujący ze skrajnym odpowiada 2^1, czyli 2, następny bit to 2^2, czyli 4, i tak dalej. Wobec tego, pokazana powyżej liczba odpowiada wartości $2^2+2^5+2^6 = 4+32+64 = 100$.

Inaczej obsługiwane są liczby ujemne. W większości systemów są one zapisywane w formie tak zwanego „uzupełnienia do dwóch". W zapisie takim skrajny lewy bit jest *bitem znaku*. Jeśli jest ustawiony na 1, liczba jest ujemna; w przeciwnym razie, jeśli ma wartość 0, liczba jest dodatnia. Pozostałe bity oznaczają wartość liczby. W zapisie uzupełnienia do dwóch wartość –1 zapisujemy, stosując same bity o wartości 1:

```
11111111
```

Wygodnym sposobem konwersji liczb ujemnych z układu dziesiętnego na dwójkowy jest najpierw dodanie do nich 1, zapisanie wartości bezwzględnej dwójkowo i następnie dopełnienie wszystkich bitów, czyli wszystkie bity 1 zmienia się na 0, a bity 0 zmienia się na 1. Wobec tego, aby zapisać dwójkowo –5, dodajemy 1 (otrzymujemy –4); zapisujemy dwójkowo 4 — mamy 00000100; w końcu dopełniamy bity, otrzymując 11111011.

Aby zamienić liczbę ujemną z formy dwójkowej z powrotem na dziesiętną, najpierw dopełniamy wszystkie bity, wynik zamieniamy na liczbę dziesiętną, zmieniamy znak wyniku i odejmujemy od uzyskanej liczby 1.

Teraz, kiedy wiemy o dwóch uzupełniających się wzajemnie formach zapisu, stwierdzamy, że największą liczbą dodatnią, jaką można zapisać na n bitach, jest $2^{n-1}-1$. Zatem na 8 bitach można zapisać liczby do 2^7-1, czyli do 127. Analogicznie najmniejsza liczba ujemna, jaką można zapisać na n bitach, to -2^{n-1}, czyli dla ośmiu bitów –128 (warto zastanowić się, dlaczego największa liczba dodatnia i najmniejsza ujemna nie mają takiej samej wartości bezwzględnej).

W większości współczesnych procesorów liczby całkowite zajmują ciągły obszar czterech bajtów, czyli 32 bity. Zatem największą możliwą do zapisania liczbą dodatnią jest $2^{31}-1$ czyli 2 147 483 647, a najmniejszą liczbą ujemną jest –2 147 483 648.

W rozdziale 3. omówiliśmy modyfikator `unsigned` i pokazaliśmy, jak można go użyć do „zwiększenia objętości" zmiennej. Jest to możliwe dzięki wykorzystaniu skrajnego lewego bitu, który normalnie używany był na znak. Tym razem znak jest niepotrzebny, bo modyfikator `unsigned` powoduje, że znak przestaje być potrzebny. Ten dodatkowy bit dwukrotnie zwiększa zakres dostępnych liczb. Ściślej rzecz biorąc, n bitów pozwala zapisać wartości do 2^{n-1}. W komputerach zapisujących liczby całkowite na 32 bitach oznacza to, że zakres dopuszczalnych wartości rozciąga się od 0 do 4 294 967 296.

Operatory bitowe

Mamy już nieco wiedzy wstępnej, więc czas zająć się dostępnymi operatorami bitowymi. Operatory tego typu dostępne w języku C zestawiono w tabeli 11.1.

Wszystkie operatory z tabeli 11.1, z wyjątkiem operatora negacji bitowej (~), są operatorami binarnymi i jako takie mają dwa argumenty. Działania bitowe robi się na liczbach całkowitych, takich jak `short`, `int`, `long`, `long long`, czy to `signed`, czy `unsigned`, a także na znakach. Nie można wykonywać działań bitowych na liczbach zmiennoprzecinkowych.

Tabela 11.1. Operatory bitowe

Symbol	Działanie
&	bitowe AND
|	bitowe OR
^	bitowe OR wyłączające (EX-OR)
~	bitowa negacja
<<	przesunięcie w lewo
>>	przesunięcie w prawo

Bitowy operator AND

Kiedy w języku C łączymy bitowym operatorem AND dwie wartości, porównywane są poszczególne bity tych wartości. Bitowy operator AND, nazywany też bitową koniunkcją, zapisujemy za pomocą symbolu &. Bit wyniku jest jedynką tylko wtedy, gdy oba odpowiednie bity w argumentach działania są jedynkami; wszelkie inne kombinacje dają 0. Jeśli *b1* i *b2* to bity obu argumentów, to wynik działania AND na tych bitach zapisuje się w formie *tabelki prawdy* opisującej wynik dla wszystkich możliwych wartości argumentów.

```
b1      b2   b1 & b2
-------------------------
0       0       0
0       1       0
1       0       0
1       1       1
```

Jeśli na przykład w1 i w2 są typu short int, w1 jest równe 25, a w2 77, to instrukcja:

```
w3 = w1 & w2;
```

spowoduje przypisanie zmiennej w3 liczby 9. Łatwo zauważyć, dlaczego tak jest, jeśli tylko zapiszemy w1, w2 i w3 jako liczby binarne. Załóżmy, że mamy do czynienia z 16-bitowymi wartościami short int.

```
w1    0000000000011001     25
w2    0000000001001101   & 77
      -------------------------
w3    0000000000001001      9
```

Jeśli tylko pamiętamy, jak działa operator logicznego AND (&&), zwracający prawdę, gdy prawdziwe są oba jego operandy, łatwo będzie zapamiętać też działanie bitowego AND. Trzeba tylko uważać, aby obu tych operatorów nie pomylić! Logiczny operator AND — && — używany jest w wyrażeniach logicznych i zwraca prawdę lub fałsz; nie wykonuje natomiast bitowego działania AND.

Bitowa operacja AND jest często używana do maskowania wartości, czyli operatora tego można używać do ustawiania poszczególnych bitów na 0. Na przykład instrukcja:

```
w3 = w1 & 3;
```

przypisuje zmiennej w3 wartość w1 bitowo wymnożoną (AND) przez stałą 3. Efektem jest ustawienie wszystkich bitów w3, poza dwoma skrajnymi prawymi bitami, na 0; ostatnie dwa bity w1 są zachowywane w w3.

Bitowego operatora AND, tak jak wszystkich innych operatorów dwuargumentowych, można używać w przypisaniach; instrukcja:

```
word &= 15;
```

realizuje to samo działanie co:

```
word = word & 15;
```

i powoduje ustawienie skrajnych czterech prawych bitów zmiennej word na zero.

Kiedy w operacjach bitowych używa się stałych, zwykle wygodnie jest stałe te zapisywać w systemie ósemkowym lub szesnastkowym. Na wybór jednego z tych systemów wpływa wielkość danych. Na przykład: kiedy używamy komputera 32-bitowego, zwykle stosujemy zapis szesnastkowy, gdyż 32 dzieli się bez reszty przez 4 (4 to liczba bitów jednej cyfry szesnastkowej).

W programie 11.1 pokazano użycie bitowego operatora AND. W programie tym używamy tylko liczb dodatnich, więc wszystkie zmienne są deklarowane jako unsigned int.

Program 11.1. **Bitowy operator AND**

```
// Program pokazujący bitowy operator AND

#include <stdio.h>

int main (void)
{
    unsigned int  word1 = 077u, word2 = 0150u, word3 = 0210u;

    printf ("%o  ", word1 & word2);
    printf ("%o  ", word1 & word1);
    printf ("%o  ", word1 & word2 & word3);
    printf ("%o\n", word1 & 1);

    return 0;
}
```

Program 11.1. **Wyniki**

```
           50   77  10  1
```

Przypomnijmy, że jeśli stała całkowitoliczbowa ma wiodące 0, jest zapisana ósemkowo. Wobec tego trzy zmienne typu unsigned int — word1, word2 i word3 — otrzymują początkowe wartości *ósemkowe* 077, 0150 i 0210. Przypomnijmy jeszcze z rozdziału 3., że jeśli za stałą całkowitoliczbową znajduje się litera u lub U, stała ta jest traktowana jako liczba bez znaku.

Pierwsze wywołanie printf pokazuje wartość 50, będącą wynikiem bitowego mnożenia AND zmiennych word1 i word2. Wartość ta jest liczona następująco:

```
word1    ... 000 111 111     077
word2    ... 001 101 000  & 0150
       --------------------------
         ... 000 101 000     050
```

Pokazano jedynie dziewięć skrajnych prawych bitów wartości, gdyż wszystkie bity po lewej stronie są zerami. Liczby dwójkowe ułożono w 3-bitowe grupy, aby ułatwić czytanie liczb dwójkowych jako ósemkowe i odwrotnie.

Drugie wywołanie funkcji `printf` powoduje wyświetlenie wartości 77 będącej wynikiem mnożenia `AND` zmiennej `word1` przez samą siebie. Z definicji, każda wielkość *x* mnożona binarnie `AND` przez samą siebie daje *x*.

Trzecie wywołanie `printf` pokazuje wynik mnożenia `AND` przez siebie trzech wartości: `word1`, `word2` i `word3`. Operacja bitowego mnożenia `AND` działa tak, że nie ma znaczenia, czy wyrażenie `a & b & c` jest wyliczane jako `(a & b) & c`, czy jako `a & (b & c)`; dla jasności wykładu przetwarzanie będziemy wykonywali od strony lewej do prawej. Jako ćwiczenie dla czytelnika zostawiamy sprawdzenie, że uzyskana ósemkowa wartość 10 jest prawidłowym wynikiem, oraz wymnożenie `AND` wartości `word1`, `word2` i `word3`.

Ostatnie wywołanie `printf` powoduje pobranie skrajnego prawego bitu zmiennej `word1`. Jest to inna metoda sprawdzenia, czy liczba całkowita jest parzysta czy nieparzysta: skrajny prawy bit liczb nieparzystych to 1, parzystych — 0. Jeśli zatem wykonamy instrukcję:

```
if ( word1 & 1 )
   ...
```

wyrażenie jest prawdziwe, jeśli wartość `word1` jest nieparzysta (mnożenie bitowe `AND` daje 1) i fałszywe, jeśli wartość `word1` jest parzysta. Uwaga! W przypadku systemów zapisujących liczby w systemie uzupełnienia do dwóch opisana metoda nie działa dla liczb ujemnych.

Bitowy operator OR

Kiedy w języku C dwa argumenty łączy się bitowym operatorem `OR`, argumenty te także są analizowane bit po bicie. Tym razem jednak wynikowy bit jest jedynką, kiedy jedynką jest odpowiedni bit pierwszego *lub* drugiego argumentu. Oto tabelka prawdy operatora bitowego `OR`.

```
b1      b2   b1 | b2
-------------------------
0       0       0
0       1       1
1       0       1
1       1       1
```

Jeśli zatem zmienna `w1` jest typu `unsigned int` i jest równa ósemkowo `0431`, `w2` też jest typu `unsigned int` i jest równa ósemkowo `0152`, bitowy operator `OR` wykonany na `w1` i `w2` da ósemkowy wynik `0573`:

```
w1    ... 100 011 001    0431
w2    ... 001 101 010  | 0152
```

```
    ---------------------------
      ... 101 111 011    0573
```

Tak jak w przypadku bitowego operatora AND, trzeba pamiętać, aby nie mylić bitowego operatora OR (|) z logicznym operatorem alternatywy (||); ten drugi pozwala sprawdzić, czy prawdziwa jest jedna z dwóch wartości logicznych.

Bitowy operator OR często służy do ustawiania pewnych bitów słowa na 1. Na przykład instrukcja:

```
w1 = w1 | 07;
```

ustawia trzy skrajne prawe bity w1 na 1, niezależnie od tego, jakie wcześniej były wartości tych bitów. Można oczywiście użyć specjalizowanego operatora przypisania:

```
w1 |= 07;
```

Przykładowy program pokazujący użycie operatora OR zostanie zaprezentowany później w tym rozdziale.

Bitowy operator OR wyłączającego

Bitowy operator OR wyłączającego (określany też jako EX-OR lub XOR) działa następująco: jeśli jeden z dwóch odpowiadających sobie bitów jest równy 1 — ale nie oba — wynikiem jest 1. W przeciwnym razie wynikiem jest 0. Tabelka prawdy operatora XOR pokazana została poniżej:

```
b1      b2   b1 ^ b2
-------------------------
0       0       0
0       1       1
1       0       1
1       1       0
```

Jeśli zmienne w1 i w2 są równe ósemkowo 0536 i 0266, to wynik działania w1 XOR w2 to ósemkowe 0750:

```
w1    ... 101 011 110    0536
w2    ... 010 110 110  ^ 0266
    ---------------------------
      ... 111 101 000    0750
```

Ciekawą cechą operatora XOR jest to, że dowolna wartość wymnożona XOR przez samą siebie daje 0. Ten trik był intensywnie wykorzystywany przez programistów asemblera do szybkiego zerowania wartości i do sprawdzania, czy są one sobie równe. Nie jest to metoda polecana w języku C, gdyż nie oszczędza czasu oraz sprawia, że programy są mniej czytelne.

Innym ciekawym zastosowaniem operatora XOR jest wymiana dwóch wartości bez konieczności wykorzystywania dodatkowej pamięci. Wiadomo, że aby normalnie zamienić miejscami dwie liczby całkowite i1 i i2, potrzebne są instrukcje:

```
temp = i1;
i1 = i2;
i2 = temp;
```

Korzystając z operatora XOR, możemy wartości zamienić miejscami bez używania dodatkowej pamięci na zmienną tymczasową:

```
i1 ^= i2;
i2 ^= i1;
i1 ^= i2;
```

Jako ćwiczenie pozostawiamy czytelnikowi sprawdzenie, że takie instrukcje faktycznie zamienią miejscami wartości i1 i i2.

Operator negacji bitowej

Operator negacji bitowej to operator jednoargumentowy, który po prostu „przełącza" bity swojego argumentu. Każdy bit będący dotąd jedynką staje się zerem i odwrotnie. Tabela prawdy tego operatora jest bardzo prosta:

```
b1      ~b1
------------
 0       1
 1       0
```

Jeśli w1 jest 16-bitową liczbą typu short int, a jej wartość ósemkowa to 0122457, jej bitowa negacja da ósemkowo 0055320:

```
 w1   1 010 010 100 101 111    0122457
~w1   0 101 101 011 010 000    0055320
```

Operatora negacji bitowej (~) nie wolno mylić z operatorem minusa arytmetycznego (–) czy logicznym operatorem negacji (!). Jeśli zatem zmienna w1 jest typu int i ma wartość 0, to –w1 też da 0. Jeśli do w1 zastosujemy operator negacji bitowej, wszystkie bity zostaną ustawione na 1, co odpowiada wartości –1 w przypadku liczb ze znakiem uzupełnianych do dwóch. W końcu, jeśli zastosujemy do w1 operator logicznej negacji, uzyskamy wynik true (1), gdyż w1 początkowo jest równe false (0).

Operator negacji bitowej jest przydatny, kiedy nie wiemy, jaka jest długość wartości mierzona w bitach. Uzyskujemy w ten sposób większą przenośność programu, tak więc program jest mniej zależny od typu komputera, na jakim jest uruchamiany. Aby na przykład ustawić najmniej znaczący bit liczby w1 typu int na 0, możemy ją wymnożyć AND z liczbą int składającą się z samych jedynek z wyjątkiem ostatniego, zerowego bitu. Odpowiednia instrukcja języka C będzie miała postać:

```
w1 &= 0xFFFFFFFE;
```

ale zadziała tylko na komputerach, w których liczby całkowite są zapisywane na 32 bitach.

Jeśli użyjemy zapisu:

```
w1 &= ~1;
```

wielkość w1 zostanie pomnożona AND przez odpowiednią wartość na wszystkich maszynach, gdyż mamy tu negację bitową wszystkich bitów poza ostatnim, niezależnie od tego, jak wiele ich jest — w systemie 32-bitowym będzie to 31 bitów.

Program 11.2 stanowi podsumowanie wszystkich omówionych dotąd operatorów bitowych. Zanim jednak przejdziemy dalej, koniecznie trzeba wspomnieć o priorytetach tych operatorów. AND, OR oraz XOR mają niższy priorytet niż wszystkie inne operatory arytmetyczne i porównania, ale wyższy niż logiczne AND i OR. Bitowe AND ma wyższy priorytet niż bitowe XOR, a to ma wyższy priorytet niż bitowe OR. Jednoargumentowy operator negacji ma priorytet wyższy niż jakikolwiek operator binarny. Zestawienie priorytetów wszystkich operatorów zamieściliśmy w dodatku A.

Program 11.2. **Ilustracja operatorów bitowych**

```
/* Program pokazujący użycie operatorów bitowych. */

#include <stdio.h>

int main (void)
{
    unsigned int  w1 = 0525u, w2 = 0707u, w3 = 0122u;

    printf ("%o   %o   %o\n", w1 & w2, w1 | w2, w1 ^ w2);
    printf ("%o   %o   %o\n", ~w1, ~w2, ~w3);
    printf ("%o   %o   %o\n", w1 ^ w1, w1 & ~w2, w1 | w2 | w3);
    printf ("%o   %o\n", w1 | w2 & w3, w1 | w2 & ~w3);
    printf ("%o   %o\n", ~(~w1 & ~w2), ~(~w1 | ~w2));

    w1 ^= w2;
    w2 ^= w1;
    w1 ^= w2;
    printf ("w1 = %o, w2 = %o\n", w1, w2);

    return 0;
}
```

Program 11.2. **Wyniki**

```
505   727   222
37777777252   37777777070   37777777655
0   20   727
527   725
727   505
w1 = 707, w2 = 525
```

Aby sprawdzić zrozumienie działania operatorów bitowych, należy z papierem i ołówkiem przerobić każde z działań z programu 11.2. Program był uruchomiony w systemie zapisującym liczby typu int na 32 bitach.

W czwartym wywołaniu funkcji printf trzeba pamiętać, że bitowy operator AND ma wyższy priorytet niż bitowy operator OR; wpływa to na ostateczny wynik wyrażenia.

Piąte wywołanie printf pokazuje prawo de Morgana mówiące, że ~(~a & ~b) jest równe a | b, z kolei ~(~a | ~b) jest równe a & b. Znajdujący się dalej ciąg instrukcji służy do sprawdzenia wymiany wartości za pomocą operatora XOR.

Operator przesunięcia w lewo

Kiedy wartość jest przesuwana w lewo, przesunięciu w lewo podlegają poszczególne bity tej wartości. Parametrem tego działania jest liczba miejsc, o które robione jest przesunięcie. Bity przesuwane poza najstarszy bit są tracone, natomiast od strony bitu najmłodszego wartość jest uzupełniana zerami. Wobec tego, jeśli `w1` jest równe 3, wyrażenie:

```
w1 = w1 << 1;
```

zapisywane też jako:

```
w1 <<= 1;
```

powoduje przesunięcie 3 o jedno miejsce w lewo — w zmiennej `w1` umieszczane jest 6:

```
w1       ... 000 011    03
w1 << 1  ... 000 110    06
```

Argument po lewej stronie operatora `<<` to przesuwana wartość, argument po stronie prawej to liczba miejsc, o które wykonywane jest przesunięcie. Jeśli przesuniemy `w1` o jeszcze jedno miejsce w lewo, uzyskamy ósemkowe `014`:

```
w1       ... 000 110    06
w1 << 1  ... 001 100   014
```

Przesuwanie w lewo jest w praktyce równoważne mnożeniu przesuwanej wartości przez dwa. Zresztą niektóre kompilatory języka C mnożenie przez potęgi dwóch realizują za pomocą przesuwania w lewo, gdyż zwykle przesuwanie działa znacznie szybciej niż mnożenie.

Program pokazujący użycie operatora przesunięcia w lewo pokażemy po omówieniu operatora przesunięcia w prawo.

Operator przesunięcia w prawo

Jak wynika z jego nazwy, operator przesunięcia w prawo (`>>`) przesuwa bity wartości w prawo. Bity najmłodsze są przy tym tracone. Podczas przesuwania w prawo wartości bez znaku z lewej strony wartość tę uzupełnia się zerami. Dla wartości ze znakiem sposób uzupełniania zależy od znaku przesuwanej wartości i od konkretnej implementacji operatora w danym systemie. Jeśli bitem znaku jest `0` (czyli wartość jest dodatnia), wartość jest uzupełniana zerami z lewej strony. Jeśli wartość jest ujemna, czyli najstarszy bit jest jedynką, sposób uzupełniania wartości zależy od konkretnej implementacji. Jeśli uzupełnianie wykonywane jest jedynkami, mamy do czynienia z *arytmetycznym* przesunięciem w prawo; jeśli zerami, jest to przesunięcie *logiczne*.

Nigdy nie należy polegać na założeniu, że system będzie stosować arytmetyczne czy logiczne przesunięcie w prawo, gdyż program może potem na jednych komputerach działać, a na innych już nie.

Jeśli `w1` jest zmienną typu `unsigned int` zapisywaną na 32 bitach i ma wartość szesnastkową `F777EE22`, to przesunięcie jej o jedno miejsce w prawo, zapisywane przy użyciu instrukcji:

```
w1 >>= 1;
```

powoduje przypisanie w1 wartości szesnastkowej 7BBBF711:

```
w1        1111 0111 0111 0111 1110 1110 0010 0010    F777EE22
w1 >> 1   0111 1011 1011 1011 1111 0111 0001 0001    7BBBF711
```

Gdyby w1 było zadeklarowane jako (signed) int, na jednych komputerach uzyskalibyśmy taki sam wynik, a na innych, stosujących arytmetyczne przesunięcie w prawo, wynikiem byłoby FBBBF711.

Trzeba jeszcze odnotować, że jeśli w języku C próbujemy przesunąć wartość w lewo lub w prawo o tyle samo miejsc co dana wartość ma bitów lub o więcej, wynik jest nieokreślony. Wobec tego, jeśli na naszym komputerze wartości int są zapisywane na 32 bitach, przesunięcie takiej wartości o 32 lub więcej bitów, czy to w lewo, czy w prawo, może dać różne wyniki. Także przesuwanie o ujemną liczbę miejsc daje wynik nieokreślony.

Funkcja przesuwająca

Teraz użyjemy operatorów przesunięcia w lewo i w prawo w konkretnym programie — będzie to program 11.3. Niektóre systemy mają jedną instrukcję maszynową przesunięcia; przesuwanie odbywa się w lewo, jeśli liczba miejsc jest dodatnia, i w prawo, jeśli liczba miejsc do przesunięcia jest ujemna. W języku C napiszemy funkcję, która będzie zachowywała się analogicznie. Funkcja ta będzie miała dwa parametry — przesuwaną wartość i liczbę miejsc do przesunięcia. Jeśli liczba miejsc przesunięcia jest dodatnia, przesuwamy wartość w lewo o wskazaną liczbę miejsc. Jeśli liczba miejsc jest ujemna, wartość przesuwamy w prawo o moduł liczby miejsc.

Program 11.3. **Funkcja przesuwająca wartości**

```
// Funkcja przesuwająca wartość typu unsigned int: w lewo, jeśli
// liczba miejsc jest dodatnia, i w prawo, jeśli jest ujemna

#include <stdio.h>

unsigned int shift (unsigned int value, int n)
{
    if ( n > 0 )      // przesunięcie w lewo
        value <<= n;
    else              // przesunięcie w prawo
        value >>= -n;

    return value;
}

int main (void)
{
    unsigned int  w1 = 0177777u, w2 = 0444u;
    unsigned int  shift (unsigned int value, int n);

    printf ("%o\t%o\n", shift (w1, 5), w1 << 5);
    printf ("%o\t%o\n", shift (w1, -6), w1 >> 6);
    printf ("%o\t%o\n", shift (w2, 0), w2 >> 0);
```

```
    printf ("%o\n", shift (shift (w1, -3), 3));

    return 0;
}
```

Program 11.3. **Wyniki**

```
7777740 7777740
1777    1777
444     444
177770
```

W funkcji `shift` z programu 11.3 deklarujemy parametr `value` jako `unsigned int`; w ten sposób gwarantujemy sobie, że przesuwanie w prawo będzie powodowało wypełnianie zerami, czyli mamy do czynienia z logicznym przesuwaniem w prawo.

Jeśli wartość `n` licznika przesunięcia jest większa od zera, funkcja przesuwa wartość w lewo o `n` bitów. Jeśli `n` jest liczbą ujemną (lub zerem), funkcja przesuwa wartość `value` w prawo o minus `n` miejsc.

Pierwsze wywołanie funkcji `shift` w `main` to przesunięcie `w1` w lewo o pięć bitów. Wywołanie `printf` pokazuje wynik funkcji `shift` i równocześnie bezpośredni wynik przesunięcia `w1` w lewo o pięć miejsc tak, że można te wartości porównać ze sobą.

Drugie wywołanie funkcji `shift` powoduje przesunięcie `w1` o sześć miejsc w prawo. Wynik jest identyczny jak przy bezpośrednim przesunięciu `w1` o sześć miejsc w prawo.

W trzecim wywołaniu `shift` jako liczbę miejsc do przesunięcia podajemy zero. W tym wypadku funkcja przesuwa wartość o zero miejsc w prawo, co — jak widać — w ogóle nie wpływa na naszą wartość.

Ostatnie wywołanie `printf` pokazuje wynik zagnieżdżonego wywołania funkcji `shift`. Najpierw wykonywane jest wywołanie wewnętrzne, które nakazuje przesunięcie `w1` w prawo o trzy miejsca. Wynik tego wywołania, czyli `0017777`, jest przekazywany do funkcji `shift` przesuwającej tę wartość w lewo o trzy miejsca. Jak widać, powoduje to ustawienie trzech najmniej znaczących bitów `w1` na zera (ten sam efekt moglibyśmy uzyskać, wykonując po prostu bitową funkcję `AND` na naszej wartości i na stałej `~7`).

Rotowanie bitów

W ramach następnego przykładu powiążemy omawiane dotąd operacje bitowe, aby utworzyć funkcję rotującą wartość w lewo lub w prawo. Rotowanie jest podobne do przesuwania, ale jeśli wartość jest przesuwana w lewo, najstarsze bity, przesuwane poza wartość, są ponownie umieszczane na bitach najmłodszych. Jeśli rotowanie odbywa się w prawo, usuwane bity najmłodsze uzupełniają bity najstarsze. Jeśli zatem mamy do czynienia z 32-bitowymi liczbami `unsigned int`, przesunięcie szesnastkowej wartości 80000000 w lewo o jeden bit daje szesnastkowe 00000001, gdyż bit 1., który normalnie przy przesunięciu w lewo byłby utracony, teraz trafia z powrotem do liczby jako bit najmłodszy.

Nasza funkcja ma dwa parametry — pierwszy to rotowana wartość, drugi to liczba miejsc, o które chcemy naszą wartość przesunąć. Jeśli drugi parametr jest dodatni, przesuwamy w lewo; jeśli ujemny, przesuwamy w prawo.

Pożądaną funkcjonalność można dość łatwo uzyskać. Aby wyliczyć przesunięcie `x` w lewo o `n` bitów, gdzie `x` jest typu `int`, `n` zaś jest nie mniejsze niż 0 i nie większe niż liczba bitów `int` pomniejszona o 1, można pobrać skrajnych `n` lewych bitów `x`, przesunąć `x` w lewo o `n` bitów i wstawić pobrane wcześniej bity do `x` z prawej strony. Analogicznie realizuje się rotowanie w prawo.

W programie 11.4 pokazano funkcję `rotate` wykorzystującą opisany przed chwilą algorytm. W funkcji tej zakładamy, że liczba `int` ma 32 bity. W ćwiczeniach na koniec tego rozdziału napiszemy analogiczną funkcję wolną od tego założenia.

Program 11.4. **Implementacja funkcji rotującej**

```
// Program pokazujący rotowanie liczb całkowitych

#include <stdio.h>

int main (void)
{
    unsigned int  w1 = 0xabcdef00u, w2 = 0xffff1122u;
    unsigned int  rotate (unsigned int value, int n);

    printf ("%x\n", rotate (w1, 8));
    printf ("%x\n", rotate (w1, -16));
    printf ("%x\n", rotate (w2, 4));
    printf ("%x\n", rotate (w2, -2));
    printf ("%x\n", rotate (w1, 0));
    printf ("%x\n", rotate (w1, 44));

    return 0;
}

// Funkcja rotująca liczbę unsigned int w lewo lub w prawo

unsigned int rotate (unsigned int value, int n)
{
    unsigned int result, bits;

    // redukcja liczby bitów do dopuszczalnego zakresu

    if ( n > 0 )
        n = n % 32;
    else
        n = -(-n % 32);

    if ( n == 0 )
        result = value;
    else if ( n > 0 ) {     // rotacja w lewo
        bits = value >> (32 - n);
        result = value << n  |  bits;
    }
    else {                  // rotacja w prawo
        n = -n;
```

```
        bits = value << (32 – n);
        result = value >> n  |  bits;
    }

    return result;
}
```

Program 11.4. **Wyniki**

```
cdef00ab
ef00abcd
fff1122f
bfffc448
abcdef00
def00abc
```

Funkcja najpierw koryguje liczbę miejsc przesunięcia `n` do pożądanego zakresu. W kodzie:

```
if ( n > 0 )
    n = n % 32;
else
    n = –(–n % 32);
```

najpierw sprawdzamy, czy `n` jest liczbą dodatnią. Jeśli tak, liczymy `n` modulo rozmiar liczby `int` (założyliśmy, że są to 32 bity). W ten sposób `n` mieści się już w zakresie od 0 do 31. Jeśli wartość `n` jest ujemna, najpierw bierzemy liczbę do niej przeciwną, potem liczymy modulo. Robimy dlatego tak, że w C nie zdefiniowano znaku wyniku modulo dla liczb ujemnych. Jeden system może dać wynik dodatni, inny ujemny. Jeśli bierzemy liczbę przeciwną, mamy gwarancję, że wynik będzie dodatni; wtedy ponownie bierzemy wartość przeciwną, stosując jednoargumentowy operator minus — ostatecznie wynik mieści się w zakresie od –31 do 0.

Jeśli skorygowana liczba miejsc rotacji wynosi 0, po prostu przypisujemy zmiennej `result` wartość `value`. W przeciwnym razie faktycznie dokonujemy rotacji.

Rotacja `n`-bitowa w lewo odbywa się w trzech krokach. Najpierw pobieranych jest `n` skrajnych lewych bitów `value`. Robimy to, przesuwając wartość `value` w prawo o liczbę miejsc równą różnicy rozmiaru liczby `int` (u nas 32) i `n`. Następnie wartość `value` jest przesuwana o `n` bitów w lewo; w końcu pobrane wcześniej bity są z powrotem wstawiane operacją bitowego `OR`. Podobnie odbywa się rotacja w prawo.

W funkcji `main` tym razem, dla odmiany, używamy zapisu szesnastkowego. Pierwsze wywołanie funkcji `rotate` rotuje wartość `w1` o osiem bitów w lewo. Jak widać, uzyskujemy wartość `cdef00ab`, czyli wynik rotacji `abcdef00` o osiem bitów w lewo.

Drugie wywołanie `rotate` to rotacja `w1` o 16 bitów w prawo.

Następne dwa wywołania `rotate` podobnie traktują wartość `w2` i nie wymagają dodatkowych wyjaśnień. Przedostatnie wywołanie `rotate` ma liczbę miejsc równą 0 — faktycznie wynik pozostaje niezmieniony.

Ostatnie wywołanie `rotate` pokazuje rotację w lewo o 44 miejsca; jest to równoważne rotacji w lewo o 12 miejsc (bo 44 % 12 to 12).

Pola bitowe

Znając wszystkie omówione dotąd operatory bitowe, możemy wykonywać wszelkie zaawansowane operacje na pojedynczych bitach. Działania bitowe często są wykonywane na spakowanych danych. Tak jak można użyć typu `short int`, aby w niektórych systemach zaoszczędzić nieco pamięci, tak samo można pakować dane do pojedynczych bitów całego bajta lub słowa, zamiast od razu używać całego bajta (słowa), na przykład flagi, mające wartość logiczną prawdy lub fałszu, mogą być zapisywane na pojedynczych bitach. Zadeklarowanie zmiennej `char` na taką flagę w większości systemów powoduje zużycie ośmiu bitów (jednego bajta); zmienne typu `_Bool` też zwykle zajmują osiem bitów. Co więcej, jeśli w dużej tablicy chcemy przechowywać wiele takich flag, ilość zmarnowanej pamięci znacząco wzrośnie.

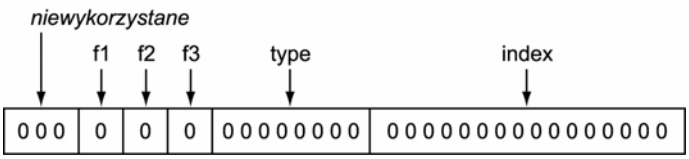
Język C zawiera dwie metody pakowania informacji w celu bardziej optymalnego wykorzystania pamięci. Pierwsza metoda polega na zapisywaniu danych w zwykłych zmiennych, na przykład typu `int`, a potem sięganiu do poszczególnych bitów przy użyciu operatorów bitowych. Druga metoda polega na zdefiniowaniu specjalnej struktury danych, nazywanej *polem bitowym*.

Aby pokazać użycie pierwszej metody, załóżmy, że chcemy spakować pięć wartości w słowie, gdyż musimy przechować bardzo dużą tablicę takich właśnie danych. Załóżmy, że trzy wartości to flagi, `f1`, `f2` i `f3`; czwarta to liczba całkowita `type` z zakresu od `1` do `255`, a ostatnia wartość to `index` z zakresu od `0` do `100 000`.

Flagi `f1`, `f2` i `f3` wymagają tylko trzech bitów — po jednym na flagę. Zapisywanie wartości `type` wymaga ośmiu bitów. W końcu na daną `index` potrzebujemy 18 bitów. Wobec tego na wszystkie sześć wartości potrzebujemy 29 bitów. Można by zatem użyć jednej zmiennej typu `int` na wszystkie te wartości:

unsigned int packed_data;

wtedy moglibyśmy przypisywać konkretne wartości poszczególnym bitom (inaczej *polom*) tej zmiennej. Przykład takiego spakowania pokazano na rysunku 11.1; zakładamy, że wielkość zmiennej `packed_data` to 32 bity.



Rysunek 11.1. Rozmieszczenie pól bitowych w zmiennej packed_data

Zauważmy, że zmienna `packed_data` ma jeszcze trzy niewykorzystane bity. Stosując do tej zmiennej odpowiednie operacje bitowe, możemy pobrać poszczególne wartości składowe. Aby na przykład ustawić wartość `7` w polu `type`, przesuwamy wartość `7` o odpowiednią liczbę miejsc w lewo, a następnie stosujemy bitowe `OR` do włączenia tej wartości do `packed_data`:

```
packed_data |= 7 << 18;
```

Wartość pola type możemy ustawić na dowolną liczbę n z zakresu 0 do 255, stosując instrukcję:

```
packed_data |= n << 18;
```

Aby zagwarantować, że n faktycznie będzie z zakresu od 0 do 255, możemy przed przesunięciem wymnożyć tę wartość bitowym AND przez stałą 0xff.

Oczywiście powyższa instrukcja zadziała tylko wtedy, gdy pole type będzie miało wartość 0. W przeciwnym razie musimy wymnożyć to pole bitowym AND przez wartość (nazywaną maską) mającą ustawione na 0 wszystkie bity należące do pola type; pozostałe bity są jedynkami:

```
packed_data &= 0xfc03ffff;
```

Aby zaoszczędzić sobie ręcznego wyliczania maski i uniezależnić się od wielkości zmiennej packed_data, możemy do zerowania pola type użyć następującej instrukcji:

```
packed_data &= ~(0xff << 18);
```

Łącząc opisane wcześniej instrukcje, możemy ustawić pole type zmiennej packed_data na wartość zapisaną w najmniej znaczących ośmiu bitach liczby n niezależnie od wcześniejszej wartości tego pola:

```
packed_data = (packed_data & ~(0xff << 18)) | ((n & 0xff) << 18);
```

Część nawiasów jest zbędna, ale dodano je w celu poprawienia czytelności wyrażenia.

Widzimy, jak skomplikowane wyrażenie jest potrzebne do realizacji dość prostego zadania ustawienia bitów pola type. Pobieranie wartości tego pola jest już łatwiejsze — można przesunąć wartość tego pola na najmniej znaczące bity i wykonać bitowe AND z maską odpowiedniej długości. Aby zatem odczytać pole type ze zmiennej packed_data i przypisać jego wartość zmiennej n, stosujemy instrukcję:

```
n = (packed_data >> 18) & 0xff;
```

Język C zawiera jednak wygodniejszą metodę obsługi pól bitowych. Używana jest do tego specjalna składnia definicji struktury, która pozwala utworzyć pole bitowe i przypisać mu nazwę. Kiedy w języku C mówi się o „polach bitowych", to chodzi o takie właśnie rozwiązanie.

Aby zdefiniować opisane wcześniej pola bitowe, możemy przygotować następującą strukturę packed_struct:

```
struct packed_struct
{
    unsigned int  :3;
    unsigned int  f1:1;
    unsigned int  f2:1;
    unsigned int  f3:1;
    unsigned int  type:8;
    unsigned int  index:18;
};
```

Struktura packed_struct ma sześć pól, pierwsze z nich nie ma nazwy. Zapis :3 oznacza trzy bity bez nazwy. Drugie pole — f1 — także jest typu unsigned int. Znajdujący się

zaraz za jego nazwą zapis `:1` oznacza, że to pole ma jeden bit. Analogicznie zdefiniowane są flagi `f2` i `f3`. Pole `type` ma osiem bitów, natomiast pole `index` — 18.

Kompilator języka C automatycznie spakuje wszystkie powyższe pola bitowe. Co ważne, w takim rozwiązaniu do poszczególnych pól możemy odwoływać się równie wygodnie jak do zwykłych pól struktur. Jeśli zatem zdefiniujemy zmienną `packed_data`:

```
struct packed_struct  packed_data;
```

łatwo będzie ustawić pole `type` tej zmiennej na `7`:

```
packed_data.type = 7;
```

Równie łatwo ustawić wartość tego pola na dowolną inną wartość `n`:

```
packed_data.type = n;
```

W tym wypadku nie musimy martwić się już, czy wartość `n` nie jest zbyt duża, gdyż zostaną użyte jedynie młodsze bity wartości `n`.

Pobieranie wartości z pola bitowego też odbywa się automatycznie, aby więc wartość pola `type` pobrać ze zmiennej `packed_data`, piszemy po prostu:

```
n = packed_data.type;
```

Pól bitowych można używać w zwykłych wyrażeniach; automatycznie będą one konwertowane na liczby całkowite. Zatem instrukcja:

```
i = packed_data.index / 5 + 1;
```

jest jak najbardziej prawidłowa, podobnie jak:

```
if ( packed_data.f2 )
    ...
```

sprawdzająca ustawienie flagi `f2`. Warto jeszcze dodać, że nie ma żadnej gwarancji, iż pola będą wewnętrznie zapisywane od prawej do lewej czy odwrotnie. Nie powinno to stanowić żadnego problemu, jeśli nie będziemy musieli obsługiwać danych zapisanych przez inny program na innym komputerze. Wtedy musimy wiedzieć, jak ułożone są pola bitowe, i przygotować odpowiednie deklaracje. Aby na przykład na maszynie rozmieszczającej pola od prawej do lewej uzyskać taką samą postać danych jak na rysunku 11.1, możemy zdefiniować strukturę `packed_struct` następująco:

```
struct packed_struct
{
    unsigned int  index:18;
    unsigned int  type:8;
    unsigned int  f3:1;
    unsigned int  f2:1;
    unsigned int  f1:1;
    unsigned int  :3;
};
```

Nigdy nie należy robić założeń dotyczących sposobu umieszczania struktur w pamięci, niezależnie od tego, czy zawierają one pola bitowe.

W strukturze zawierającej pola bitowe można też umieszczać zwykłe pola. Jeśli zatem chcemy zdefiniować strukturę mającą pola typu `int`, `char` i dwie jednobitowe flagi, możemy napisać:

```
struct  table_entry
{
    int           count;
    char          c;
    unsigned int  f1:1;
    unsigned int  f2:1;
};
```

Jeszcze kilka uwag o polach bitowych. Można je deklarować tylko jako liczby całkowite lub wartości `_Bool`. Jeśli w deklaracji użyjemy typu `int`, od konkretnej implementacji zależy, czy będzie to liczba ze znakiem czy bez znaku. Bezpieczniej pisać `signed int` lub `unsigned int`. Pole bitowe nie może mieć wymiarów, to znaczy nie można tworzyć tablicy pól, takich jak `flag:1[5]`. Nie można też pobierać adresu pola bitowego — nie istnieje przecież coś takiego jak „wskaźnik pola bitowego”.

Pola bitowe są pakowane w *jednostki* w miarę ich pojawiania się w definicji struktury. Wielkość takiej jednostki zależy od implementacji, ale zwykle jest to słowo.

Kompilator C *nie zmienia* układu pól bitowych w celu optymalizowania zajętości pamięci.

Istnieje szczególne pole bitowe — niemające nazwy, o długości 0. Używa się go do wyrównania następnego pola do granicy używanej jednostki.

Na tym kończymy omawianie działań bitowych w języku C. Jak widać, jego możliwości w tym zakresie są ogromne. Język zawiera bitowe `AND`, `OR`, `XOR`, negację oraz przesunięcia w lewo i w prawo. Pola bitowe umożliwiają przeznaczanie wybranej liczby bitów na dane; zapisywanie i odczytywanie takich pól jest bardzo wygodne, nie musimy stosować masek czy przesunięć.

W rozdziale 13. powiemy więcej o typach danych. Powiemy między innymi, co się stanie, jeśli będziemy wykonywali operacje bitowe na wartościach różnych typów — na przykład `unsigned long` i `short int`.

Zanim przejdziemy do następnego rozdziału, powtórzmy omówiony materiał, wykonując następujące ćwiczenia.

Ćwiczenia

1. Przepisz i uruchom cztery programy pokazane w tym rozdziale. Uzyskane wyniki porównaj z wynikami pokazanymi w tekście.
2. Napisz program sprawdzający, czy używany system robi arytmetyczne czy logiczne przesunięcie w prawo.
3. Jeśli wiadomo, że wyrażenie `~0` daje liczbę całkowitą składającą się z samych jedynek, napisz funkcję `int_size` zwracającą liczbę bitów zmiennej typu `int`.
4. Korzystając z wyników ćwiczenia 3., zmodyfikuj funkcję `rotate` z programu 11.4 tak, aby nie korzystała z założenia o wielkości danych typu `int`.

5. Napisz funkcję `bit_test` mającą dwa parametry — typu `unsigned int` i liczbę bitów `n`. Niech funkcja zwraca 1, jeśli `n`-ty bit jest ustawiony, i 0, jeśli `n`-ty bit jest wygaszony. Załóż, że bit 0 to skrajny lewy bit. Napisz też funkcję `bit_set`, która będzie miała takie same parametry i będzie zwracała wynik ustawienia `n`-tego bitu w przekazanej wartości.
6. Napisz funkcję `bitpat_search`, która będzie szukała danego wzorca bitów w wartości typu `unsigned int`. Funkcja będzie miała trzy parametry:

   ```
   bitpat_search (źródło, wzorzec, n)
   ```

   Funkcja będzie szukała w `źródle` (zaczynając od skrajnego lewego bitu) `n` bitów podanego `wzorca`. Jeśli zostaną znalezione, funkcja zwróci numer bitu, od którego zaczyna się wystąpienie wzorca. W przeciwnym wypadku funkcja zwróci –1. Na przykład wywołanie:

   ```
   index = bitpat_search (0xe1f4, 0x5, 3);
   ```

   spowoduje, że funkcja `bitpat_search` będzie szukała w liczbie `0xe1f4` (dwójkowo 1110 0001 1111 0100) wystąpień trzybitowego wzorca `0x5` (dwójkowo 101). Funkcja zwróci 11, gdyż wzorzec ten występuje w podanej liczbie i zaczyna się od 11-go bitu.

   Funkcja nie może czynić żadnych założeń dotyczących wielkości liczb typu `int` (zobacz też ćwiczenie 3.).
7. Napisz funkcję `bitpat_get` pobierającą podany zestaw bitów. Funkcja ta będzie miała trzy parametry: liczbę `unsigned int`, numer bitu początkowego oraz liczbę bitów. Numerowanie bitów zaczyna się od 0, od lewej strony. Funkcja ma pobierać z pierwszego parametru żądaną liczbę bitów i zwrócić wynik. Zatem wywołanie:

   ```
   bitpat_get (x, 0, 3)
   ```

   pobierze z `x` trzy pierwsze z lewej bity. Wywołanie:

   ```
   bitpat_get (x, 3, 5)
   ```

   pobierze pięć bitów, przy czym zacznie od czwartego bitu z lewej strony.
8. Napisz funkcję `bitpat_set`, która ustawi podane bity zgodnie z przekazaną wartością. Funkcja będzie miała cztery parametry: wskaźnik liczby `unsigned int`, w której bity będą ustawiane; wartość `unsigned int` zawierającą ustawianą wartość (docelowe bity są z prawej strony); wartość `int` określającą numer pierwszego bitu, który ma być ustawiony (skrajny lewy bit ma numer 0); w końcu wartość `int` określającą długość pola. Wywołanie:

   ```
   bitpat_set (&x, 0, 2, 5);
   ```

   spowoduje ustawienie pięciu bitów `x`, przy czym zacznie od trzeciego bitu z lewej strony (bit numer 2) na zero. Analogicznie wywołanie:

   ```
   bitpat_set (&x, 0x55u, 0, 8);
   ```

   ustawi osiem lewych bitów `x` na szesnastkową wartość `55`.

   Nie wolno dokonywać żadnych założeń dotyczących wielkości danych typu `int` (zobacz ćwiczenie 3.).

12

Preprocesor

W tym rozdziale omówimy następną specjalną cechę języka C, którą może pochwalić się niewiele innych języków programowania wysokiego poziomu. Preprocesor C upraszcza tworzenie programów czytelnych, łatwych do modyfikowania i przenoszenia na inne platformy. Preprocesora można użyć do dostosowania języka C do konkretnych potrzeb lub lepszego dopasowania go do swojego stylu programowania. W tym rozdziale nauczysz się:

- tworzyć własne stałe i makra przy użyciu instrukcji `#define`;
- dołączać do programu własne pliki biblioteczne za pomocą instrukcji `#include`;
- posługiwać się instrukcjami `#ifdef`, `#endif`, `#else` oraz `#ifndef`.

Preprocesor jest jednym z etapów kompilacji programu w C. Wyłapuje on specjalne instrukcje wplecione w zwykły program. Preprocesor, zgodnie ze swoją nazwą, przetwarza program, *zanim* zajmie się tym sam kompilator. Instrukcje preprocesora charakteryzują się znakiem #, który musi być pierwszym czarnym znakiem w wierszu. Jak zobaczymy, instrukcje preprocesora, nazywane dyrektywami, różnią się nieco od zwykłych instrukcji języka C. W prawie każdym opisanym do tej pory programie została użyta dyrektywa preprocesora `#include`. Jej zastosowanie jest jednak szersze, o czym wkrótce się przekonasz, a na razie zaczniemy od dyrektywy `#define`.

Dyrektywa #define

Jednym z podstawowych zastosowań dyrektywy `#define` jest przypisanie nazw symbolicznych stałym programowym. Instrukcja preprocesora:

```
#define  YES    1
```

definiuje nazwę `YES` i utożsamia ją z wartością 1. Nazwa `YES` może być dalej używana wszędzie tam, gdzie można zastosować stałą 1. Jeśli nazwa ta pojawi się gdziekolwiek w programie, zostanie przez preprocesor zastąpiona automatycznie wartością 1. Możemy na przykład nazwy `YES` użyć w instrukcji:

```
gameOver = YES;
```

Instrukcja ta przypisuje zmiennej gameOver wartość YES. Nie musimy zastanawiać się, co właściwie oznacza YES; tak naprawdę zmiennej gameOver zostanie przypisana wartość 1. Instrukcja preprocesora:

```
#define  NO    0
```

definiuje nazwę NO i powoduje, że jej wystąpienia w programie zostaną zastąpione wartością 0. Wobec tego instrukcja:

```
gameOver = NO;
```

przypisze zmiennej gameOver wartość NO, a instrukcja:

```
if ( gameOver == NO )
    ...
```

porówna wartość zmiennej gameOver ze zdefiniowaną wcześniej wartością NO.
Jedno z nielicznych miejsc, gdzie *nie można* użyć tak zdefiniowanej zmiennej, to literał znakowy; wobec tego instrukcja:

```
char  *charPtr = "YES";
```

spowoduje ustawienie zmiennej charPtr tak, że będzie wskazywała łańcuch "YES", a nie "1".

Nazwa zdefiniowana w opisany sposób *nie jest* zmienną. Wobec tego nie można jej przypisać wartości, chyba że wynikiem podstawienia takiej wartości będzie naprawdę zmienna. Jeśli definiujemy nazwę używaną w programie, to preprocesor będzie automatycznie podstawiał wszystko, co znajdzie się po prawej stronie dyrektywy #define. Jest to działanie analogiczne do wyszukiwania i podstawiania tekstu w edytorze tekstowym; w tym wypadku preprocesor zastępuje zdefiniowaną nazwę związanym z nią tekstem.

Zwróćmy uwagę na specyficzną składnię dyrektywy #define — nie ma tu znaku równości, który przypisywałby nazwie YES wartość 1. Co więcej, całe wyrażenie nie kończy się średnikiem. Wkrótce stanie się jasne, dlaczego tak jest. Najpierw jednak spójrzmy na niewielki program wykorzystujący definicje YES i NO. Funkcja isEven z programu 12.1 zwraca YES, jeśli jej parametr jest parzysty, i NO, jeśli jest nieparzysty.

Program 12.1. **Dyrektywa #define**

```
#include <stdio.h>

#define  YES    1
#define  NO     0

// Funkcja sprawdzająca, czy przekazana liczba jest parzysta

int isEven (int number)
{
    int  answer;

    if ( number % 2 == 0 )
        answer = YES;
    else
        answer = NO;
```

```
    return answer;
}

int main (void)
{
    int isEven (int number);

    if ( isEven (17) == YES )
        printf ("tak ");
    else
        printf ("nie ");

    if ( isEven (20) == YES )
        printf ("tak\n");
    else
        printf ("nie\n");

    return 0;
}
```

Program 12.1. **Wyniki**

```
nie tak
```

Program zaczyna się od dyrektywy `#define`. Nie jest to konieczne; dyrektywa ta może wystąpić *w dowolnym miejscu* w programie; musi być tylko zdefiniowana przed odwołaniem się do definiowanej nazwy. Tak definiowane nazwy zachowują się inaczej niż zmienne — nie istnieje coś takiego jak definicja lokalna. Kiedy nazwa zostanie zdefiniowana, czy to w jakiejś funkcji, czy na zewnątrz, zostanie zastąpiona odpowiadającym jej tekstem *wszędzie* w dalszej części programu. Większość programistów grupuje dyrektywy `#define` razem i umieszcza je na początku programu (lub w pliku *włączanym*[1], z którego mogą być dostępne w wielu plikach źródłowych).

Programiści często używają tak zdefiniowanej nazwy `NULL` do oznaczania wskaźnika pustego[2].

Dołączając do programu definicję:

```
#define  NULL   0
```

możemy potem pisać czytelniejszy kod, na przykład:

```
while ( listPtr != NULL )
    ...
```

Pętla `while` będzie wykonywana, dopóki wartość `listPtr` nie stanie się wskaźnikiem pustym.

---

[1] Dalej powiemy, jak dyrektywy `#define` można umieścić w specjalnym pliku, włączanym później do programu.

[2] Nazwa `NULL` jest już zdefiniowana w pliku `<stddef.h>`. Pliki włączane, takie jak tu nadmieniony, omówimy już wkrótce.

Oto kolejny przykład użycia zdefiniowanej nazwy. Załóżmy, że chcemy napisać trzy funkcje obliczające powierzchnię koła, obwód koła i objętość kuli o danym promieniu. We wszystkich tych funkcjach potrzebna będzie stała π, której wartość dość trudno zapamiętać, więc warto zdefiniować taką stałą na początku programu i potem odwoływać się do niej w poszczególnych funkcjach[3].

Program 12.2 pokazuje przygotowanie definicji stałej i użycie jej w programie.

Program 12.2. **Dalszy ciąg definicji**

```
/* Funkcja wyliczająca pole powierzchni i obwód koła oraz objętość
   kuli o danym promieniu. */

#include <stdio.h>

#define  PI      3.141592654

double area (double r)             /* pole powierzchni */
{
    return PI * r * r;
}

double circumference (double r)    /* obwód */
{
    return 2.0 * PI * r;
}

double volume (double r)           /* objętość */
{
    return 4.0 / 3.0 * PI * r * r * r;
}

int main (void)
{
    double area (double r), circumference (double r),
           volume (double r);

    printf ("promień = 1: %.4f   %.4f   %.4f\n",
            area(1.0), circumference(1.0), volume(1.0));

    printf ("promień = 4.98: %.4f   %.4f   %.4f\n",
            area(4.98), circumference(4.98), volume(4.98));

    return 0;
}
```

Program 12.2. **Wyniki**

```
promień = 1: 3.1416   6.2832   4.1888
promień = 4.98: 77.9128   31.2903   517.3403
```

[3] W pliku nagłówkowym <math.h> zdefiniowano już stałą M_PI. Aby jej używać, wystarczy włączyć ten plik do programu.

Na początku programu definiujemy nazwę PI, której odpowiada wartość 3,141592654. Użycie tej nazwy w funkcjach area, circumference i volume powoduje automatyczne podstawienie odpowiedniej wartości. Przypisanie stałej do nazwy symbolicznej zwalnia z konieczności pamiętania o wartości tej stałej przy każdym jej użyciu. Co więcej, jeśli kiedyś postanowimy wartość tej stałej zmienić (bo na przykład stwierdzimy, że się pomyliliśmy), wystarczy dokonać zmiany tylko w jednym miejscu — w dyrektywie #define. Bez tego trzeba by przejrzeć cały program, znaleźć wszystkie wystąpienia stałej i wszędzie ją pozmieniać.

Uważni czytelnicy zauważyli, że wszystkie definiowane nazwy zapisywaliśmy wielkimi literami (YES, NO, NULL i PI). Chodzi o to, aby takie wartości łatwo odróżnić od zmiennych. Część programistów konsekwentnie zapisuje wszystkie definiowane nazwy wielkimi literami, dzięki czemu łatwo stwierdzić, czy mamy do czynienia ze zmienną czy z podstawianą stałą. Inną, powszechnie stosowaną praktyką jest poprzedzanie takich nazw literą *k*. Wtedy już nie musimy się ograniczać do wielkich liter. Przykładami nazw zgodnych z tą konwencją mogą być kMaximumValues czy kSignificantDigits.

Rozszerzalność programu

Rozbudowę programu ułatwia korzystanie z definiowanych nazw do zapisywania stałych. Jeśli na przykład definiujemy tablicę, musimy podać liczbę jej elementów, jawnie lub nie (za pośrednictwem listy inicjalizacyjnej). W dalszych instrukcjach programu zwykle korzystamy ze znajomości wielkości tablicy. Jeśli na przykład tablicę dataValues zdefiniowano następująco:

```
float  dataValues[1000];
```

istnieje duże prawdopodobieństwo, że dalej w programie wykorzystamy fakt, że dataValues zawiera 1000 elementów. Na przykład w pętli for:

```
for ( i = 0;  i < 1000;  ++i )
    ...
```

wykorzystujemy 1000 jako górne ograniczenie pętli. Instrukcja:

```
if ( index > 999 )
    ...
```

pozwala sprawdzić, czy wartość indeksu przekracza dopuszczalną wielkość tablicy.

Załóżmy teraz, że musimy zwiększyć tablicę dataValues z 1000 do 2000 elementów. Trzeba będzie zmienić wszystkie instrukcje wykorzystujące fakt, że dataValues zawierała 1000 komórek.

Lepszym sposobem definiowania wielkości tablicy, ułatwiającym późniejsze modyfikowanie programu, jest zdefiniowanie nazwy opisującej wielkość tablicy. Jeśli zatem mamy zdefiniowaną nazwę MAXIMUM_DATAVALUES:

```
#define  MAXIMUM_DATAVALUES    1000
```

naszą tablicę możemy zdefiniować następująco:

```
float  dataValues[MAXIMUM_DATAVALUES];
```

Z tej samej stałej będziemy korzystali tam, gdzie potrzebne będzie ograniczenie wielkości tablicy. Aby przejrzeć elementy tablicy `dataValues`, możemy użyć instrukcji `for` w postaci:

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

Aby sprawdzić, czy indeks przekracza dopuszczalny zakres, możemy napisać:

```
if ( index > MAXIMUM_DATAVALUES – 1 )
    ...
```

i tak dalej. Co najważniejsze, łatwo będzie teraz zwiększyć tablicę `dataValues` do 2000 elementów; wystarczy zmienić definicję stałej:

```
#define  MAXIMUM_DATAVALUES    2000
```

Jeśli w programie, odwołując się do wielkości tej tablicy, wszędzie używaliśmy stałej `MAXIMUM_DATAVALUES`, nie trzeba już robić żadnych zmian.

Przenośność programu

Inną, przyjemną cechą definiowania stałych jest ułatwienie przenoszenia programów między różnymi systemami. Czasami konieczne bywa użycie stałej powiązanej z konkretnym komputerem. Może na przykład chodzić o konkretny adres w pamięci, o nazwę pliku czy liczbę bitów w słowie. Przypomnijmy sobie funkcję `rotate` z programu 11.4, która wykorzystywała fakt, że zmienna typu `int` ma 32 bity na używanym przez nas komputerze.

Jeśli uruchomilibyśmy taki program na innym komputerze, na którym dana `int` jest zapisywana na 64 bitach, funkcja `rotate` nie zadziałałaby prawidłowo[4]. Przeanalizujmy poniższy kod. Kiedy program musi używać wartości zależnych od architektury komputera, warto tego typu zależności jak najbardziej izolować. Znaczącą pomocą może okazać się dyrektywa `#define`. Nowa wersja funkcji `rotate` jest łatwiejsza do uruchomienia w innym systemie, choć w tym wypadku problem jeszcze nie jest zbyt poważny. Oto nowa wersja funkcji:

```
#include <stdio.h>

#define kIntSize  32     // *** zależne od systemu!!! ***

// Funkcja rotująca wartość typu unsigned int w lewo lub w prawo

unsigned int rotate (unsigned int value, int n)
{
    unsigned int result, bits;
```

[4] Można oczywiście tak napisać funkcję `rotate`, aby sama sprawdzała, ile bitów ma zmienna typu `int`, i wtedy program będzie całkowicie niezależny od systemu. Stosowne przykłady zamieściliśmy w ćwiczeniach 3. i 4. umieszczonych na końcu 11. rozdziału.

```
    /* redukujemy licznik przesunięcia do ustalonego zakresu */

    if ( n > 0 )
        n = n % kIntSize;
    else
    n = -(-n % kIntSize);

    if ( n == 0 )
        result = value;
    else if ( n > 0 )     /* rotacja w lewo */
    {
        bits = value >> (kIntSize - n);
        result = value << n  |  bits;
    }
    else                  /* rotacja w prawo */
    {
        n = -n;
        bits = value << (kIntSize - n);
        result = value >> n  |  bits;
    }

    return result;
}
```

Bardziej złożone definicje

Definicje nazwy mogą obejmować więcej niż zwykłą stałą. Mogą zawierać wyrażenie, a jak wkrótce zobaczymy, także cokolwiek innego!

Poniższa definicja stałej `DWA_PI` opisuje iloczyn `2.0` i `3.141592654`:

```
#define  DWA_PI    2.0 * 3.141592654
```

Definicji tej można używać wszędzie tam, gdzie można by użyć wyrażenia `2.0·3.141592654`. Zatem w funkcji `circumference` moglibyśmy w instrukcji `return` użyć zapisu:

```
return DWA_PI * r;
```

Kiedy w programie pojawia się zdefiniowana wcześniej nazwa, zastępowana jest *wszystkim*, co wystąpiło po prawej stronie dyrektywy `#define`. Kiedy zatem w powyższej instrukcji `return` preprocesor natknie się na stałą `DWA_PI`, zastąpi ją wyrażeniem `2.0·3.141592654`.

To, że preprocesor podstawia dosłownie tekst, wyjaśnia, dlaczego dyrektywy `#define` nie kończymy zwykle średnikiem. Gdybyśmy to zrobili, do podstawienia użyty zostałby także średnik. Jeśli mielibyśmy do czynienia z definicją:

```
#define  PI    3.141592654;
```

i dalej napisalibyśmy:

```
return 2.0 * PI * r;
```

po zadziałaniu preprocesora program wyglądałby następująco:

```
return 2.0 * 3.141592654; * r;
```

czyli wystąpiłby błąd składniowy.

Sama definicja preprocesora nie musi zawierać poprawnego wyrażenia języka C, jeśli tylko będą poprawne wyrażenia uzyskiwane w wyniku podstawień. Na przykład definicja:

```
#define  LEFT_SHIFT_8    << 8
```

jest poprawna, choć wyrażenie zawarte w stałej `LEFT_SHIFT_8` samo w sobie nie jest poprawne składniowo. Jeśli użyjemy jednak tej stałej następująco:

```
x = y LEFT_SHIFT_8;
```

wartość zmiennej y zostanie przesunięta w lewo o 8 miejsc, a wynik będzie przypisany zmiennej x. Znacznie bardziej praktyczne będą definicje:

```
#define  AND      &&
#define  OR       ||
```

dzięki którym dalej będziemy mogli pisać:

```
if ( x > 0  AND  x < 10 )
    ...
```

czy:

```
if ( y == 0  OR  y == value )
    ....
```

Jeśli dodamy do tego jeszcze definicję równości:

```
#define  ROWNE    ==
```

to możemy pisać:

```
if ( y ROWNE 0  OR  y ROWNE value )
    ...
```

unikając ryzyka pomylenia porównania z przypisaniem; w ten sposób można też poprawić czytelność programu.

Wprawdzie powyższe przykłady pokazują możliwości dyrektywy `#define`, ale pamiętać trzeba, że takie przedefiniowywanie używanego języka programowania uchodzi za złą praktykę programistyczną. Co więcej, dla osoby z zewnątrz taki kod może być trudny do zrozumienia.

Możemy pójść dalej i definiować jedne wartości na podstawie innych:

```
#define PI         3.141592654
#define DWA_PI     2.0 * PI
```

Definicja nazwy `DWA_PI` wykorzystuje zdefiniowaną wcześniej nazwę `PI`, dzięki czemu zbędne jest ponowne podawanie wartości `3.141592654`.

Jeśli nawet zmienimy kolejność definicji:

```
#define DWA_PI     2.0 * PI
#define PI         3.141592654
```

wszystko nadal będzie działało poprawnie. Zawsze można odwoływać się do innych wartości, jeśli tylko są zdefiniowane wszystkie używane nazwy.

Starannie dobrane definicje pozwalają ograniczyć ilość dokumentacji w programie. Weźmy pod uwagę następującą instrukcję:

```
if ( year % 4 == 0  &&  year % 100 != 0  ||  year % 400 == 0 )
    ...
```

Wiemy, że pokazane wyrażenie pozwala sprawdzić, czy rok `year` jest przestępny. Teraz rozważmy następującą definicję i zmodyfikowaną instrukcję `if`:

```
#define ROK_PRZESTEPNY   year % 4 == 0  &&  year % 100 != 0  \
                      || year % 400 == 0
    ...
if ( ROK_PRZESTEPNY )
    ...
```

Normalnie preprocesor zakłada, że cała definicja mieści się w jednym wierszu. Jeśli potrzebny jest następny wiersz, ostatnim znakiem w pierwszym wierszu musi być odwrotny ukośnik sygnalizujący preprocesorowi kontynuację definicji w następnym wierszu; ukośnik ten nie jest włączany do podstawianego wyrażenia. Takich kontynuacji wiersza może być więcej; każdy nieostatni wiersz musi kończyć się odwrotnym ukośnikiem.

Powyższa instrukcja `if` jest znacznie łatwiejsza do rozumienia niż jej poprzednia wersja. Zbędne są nawet jakiekolwiek komentarze. Definicja `ROK_PRZESTEPNY` działa zupełnie jak funkcja. Taką samą czytelność programu możemy osiągnąć, wywołując funkcję `rok_przestepny`. Wybór między funkcją a definicją jest kwestią preferencji programisty. Oczywiście funkcja może być bardziej uniwersalna, gdyż może mieć parametr, dzięki czemu łatwo sprawdzać, czy dowolny rok jest przestępny, a nie tylko rok zapisany w zmiennej `year`, jak to ma miejsce w naszej definicji. Jednak definicję można tak zapisać, aby miała parametry.

Parametry i makra

Definicja `ROK_PRZESTEPNY` może mieć parametr `y`:

```
#define ROK_PRZESTEPNY(y)   y % 4 == 0  &&  y % 100 != 0  \
                         || y % 400 == 0
```

W przeciwieństwie do funkcji, nie podajemy typu parametru `y`, gdyż robimy podstawienie tekstowe, a nie wywołujemy funkcji.

Zauważmy, że między definiowaną nazwą a lewym nawiasem nie może być żadnych spacji.

Mamy już powyższą definicję, możemy zatem używać instrukcji:

```
if ( ROK_PRZESTEPNY (year) )
    ...
```

albo:

```
if ( ROK_PRZESTEPNY (next_year) )
    ...
```

W ten sposób raz sprawdzamy, czy przestępny jest rok `year`, a potem — czy przestępny jest rok `next_year`. W powyższej instrukcji podstawiona zostanie definicja, ale zamiast

parametru y użyta będzie zmienna next_year. Wobec tego uzyskana instrukcja będzie miała postać:

```
if ( next_year % 4 == 0  && next_year % 100 != 0
        || next_year % 400 == 0 )
    ...
```

W języku C takie definicje nazywane są *makrami*. Przez makra zwykle rozumie się definicje mające parametry. Zaletą definiowania czegoś w formie makra zamiast funkcji jest to, że nieistotny jest typ parametru. Weźmy na przykład pod uwagę makro KWADRAT wyliczające po prostu kwadrat swojego parametru. Definicja:

```
#define  KWADRAT(x)    x * x
```

pozwala pisać dalej:

```
y = KWADRAT (v);
```

co spowoduje przypisanie zmiennej y wartości v^2. Co ważne, v może być typu int, może być typu long, czy choćby float; zawsze użyjemy *tego samego* makra. Gdyby KWADRAT był funkcją mającą, dajmy na to, parametr typu int, nie można by wyliczyć kwadratu wartości typu double. O jednym jeszcze trzeba powiedzieć — wobec tego, że makra są podstawiane przez preprocesor bezpośrednio do programu, nieuchronnie zajmują więcej pamięci niż analogiczne funkcje. Z drugiej jednak strony, wywołanie i odebranie wyniku funkcji trwa dłużej.

Choć definicja makra KWADRAT jest prosta, wystarcza do pokazania pewnej pułapki, w którą nader łatwo wpaść. Instrukcja:

```
y = KWADRAT (v);
```

powoduje przypisanie zmiennej y wartości v^2. A co się stanie, jeśli wywołamy:

```
y = KWADRAT (v + 1);
```

Wbrew naszym oczekiwaniom, zmiennej y nie zostanie przypisana wartość $(v + 1)^2$. Preprocesor podstawia tekst, więc przykład powyższy zostanie zinterpretowany jako:

```
y = v + 1 * v + 1;
```

co oczywiście nie jest tym, o co chodziło. Aby uniknąć tego typu problemów, definicję makra KWADRAT należy uzupełnić o nawiasy:

```
#define  KWADRAT(x)   ( (x) * (x) )
```

Powyższe wyrażenie może wydawać się nieco dziwne, ale pamiętajmy, że w makrze tym pod x zostanie podstawione wszystko, co podamy jako parametr. Teraz już instrukcja:

```
y = KWADRAT (v + 1);
```

zostanie zgodnie z oczekiwaniami zinterpretowana jako:

```
y = ( (v + 1) * (v + 1) );
```

Podczas definiowania makr szczególnie przydatny może się okazać operator wyboru. Poniższa dyrektywa definiuje makro MAX podające większą z dwóch wartości:

```
#define  MAX(a,b)   ( ((a) > (b)) ? (a) : (b) )
```

Makro to pozwala użyć instrukcji typu:

```
limit = MAX (x + y, minValue);
```

Powyższa instrukcja przypisze zmiennej limit większą z wartości — x+y lub minValue. Wokół całego wyrażenia MAX dodano nawiasy, aby prawidłowo były interpretowane wyrażenia takie jak:

```
MAX (x, y) * 100
```

Z kolei nawiasy otaczające poszczególne parametry zapewniają prawidłową interpretację wyrażeń:

```
MAX (x & y, z)
```

Bitowy operator AND ma niższy priorytet niż operator > używany w definicji makra, więc zostałby później wykonany, co powodowałoby nieprawidłowe działanie definicji.

Oto makro sprawdzające, czy podany znak jest małą literą:

```
#define IS_LOWER_CASE(x)   ( ((x) >= 'a') && ((x) <= 'z') )
```

Teraz możemy używać instrukcji typu:

```
if ( IS_LOWER_CASE (c) )
    ...
```

Można nawet napisać makro konwertujące małe litery z zestawu ASCII na wielkie; inne znaki pozostają niezmienione.

```
#define TO_UPPER(x)  ( IS_LOWER_CASE (x) ? (x) – 'a' + 'A' : (x) )
```

Teraz pętla:

```
while ( *string != '\0' )
{
    *string = TO_UPPER (*string);
    ++string;
}
```

zastąpi w łańcuchu string wszystkie małe litery wielkimi[5].

Zmienna liczba parametrów makra

Makro może mieć zmienną liczbę parametrów. Informujemy o tym preprocesor, umieszczając trzy kropki na końcu listy parametrów. Wszystkie pozostałe parametry

---

[5] Istnieje zestaw funkcji bibliotecznych służących do konwersji i porównywania znaków, na przykład odpowiednikami makr IS_LOWER_CASE i TO_UPPER są funkcje islower i toupper. Więcej szczegółów na ten temat podajemy w dodatku B poświęconym bibliotece standardowej języka C.

z listy są dostępne razem w definicji makra przez specjalny identyfikator: __VA_ARGS__. Oto przykładowe makro — debugPrintf — korzystające ze zmiennej liczby parametrów:

```
#define debugPrintf(...)   printf ("DEBUG: " __VA_ARGS__);
```

Przykładowe użycie tego makra może wyglądać następująco:

```
debugPrintf ("Witaj, świecie!\n");
```

lub:

```
debugPrintf ("i = %i, j = %i\n", i, j);
```

W pierwszym przypadku wynikiem będzie:

```
DEBUG: Witaj, świecie!
```

a w drugim, jeśli i miałoby wartość 100, a j — 200, wynikiem byłoby:

```
DEBUG: i = 100, j = 200
```

W pierwszym wypadku wywołanie funkcji printf zostanie rozwinięte jako:

```
printf ("DEBUG: " " Witaj, świecie\n");
```

Sąsiadujące ze sobą literały łańcuchowe są łączone, więc ostatecznie wywołanie to przybierze postać:

```
printf ("DEBUG: Witaj, świecie\n");
```

Operator

Jeśli przed parametrem makra umieścimy znak #, preprocesor na podstawie tego parametru utworzy stałą znakową. Na przykład definicja:

```
#define str(x)    # x
```

powoduje, że wywołanie:

```
str (testing)
```

zostanie zinterpretowane jako:

```
"testing"
```

Wywołanie funkcji printf:

```
printf (str (Programowanie w C to niezła zabawa.\n));
```

jest zatem odpowiednikiem:

```
printf ("Programowanie w C to niezła zabawa.\n");
```

Preprocesor otacza parametr makra cudzysłowami; zachowywane są wszelkie cudzysłowy i odwrotne ukośniki występujące w parametrze:

```
str ("witaj")
```

daje:

```
"\"witaj\""
```

Bardziej praktycznym przykładem użycia operatora # może być następujące makro:

```
#define  printint(var)   printf (# var " = %i\n", var)
```

Makro to służy do pokazania wartości zmiennej całkowitej. Jeśli `count` będzie taką zmienną o wartości 100, instrukcja:

```
printint (count);
```

zostanie rozwinięta jako:

```
printf ("count" " = %i\n", count);
```

Zatem operator # umożliwia tworzenie łańcucha znakowego na podstawie parametru makra. Spacja między tym operatorem a nazwą parametru jest opcjonalna.

Operator

Operator ## występujący w definicjach makr służy do łączenia ze sobą dwóch *elementów*. Przed nim (lub za nim) występuje nazwa parametru makra. Preprocesor pobiera parametr makra i tworzy z niego i danych znajdujących się przed (za) operatorem ## jeden element.

Załóżmy na przykład, że mamy listę zmiennych od `x1` do `x100`. Można utworzyć makro `printx`, które jako parametr pobierze wartość od 1 do 100 i pokaże odpowiednią zmienną `x`:

```
#define printx(n)   printf ("%i\n", x ## n)
```

Fragment:

```
x ## n
```

mówi, że program pobierze elementy występujące przed i za operatorem ## (czyli litera `x` i `n`) i utworzy z nich jeden nowy element. Zatem wywołanie:

```
printx (20);
```

zostanie rozwinięte jako:

```
printf ("%i\n", x20);
```

Makro `printx` może nawet korzystać z pokazanego wcześniej makra `printint`, aby pokazać nazwę zmiennej i wyświetlić wartość tej zmiennej:

```
#define printx(n)   printint(x ## n)
```

Wywołanie:

```
printx (10);
```

najpierw zostanie rozwinięte do postaci:

```
printint (x10);
```

a następnie do:

```
printf ("x10" " = %i\n", x10);
```

i w końcu przybierze postać:

```
printf ("x10 = %i\n", x10);
```

Dyrektywa #include

Kiedy nabędziemy nieco doświadczenia w programowaniu w C, szybko dorobimy się zestawu makr, z których będziemy chcieli korzystać we wszystkich programach. Jednak zamiast wpisywać wszystkie te makra w każdym programie, możemy skorzystać z preprocesora, zebrać te definicje w odrębny plik i *włączyć* je do programu za pomocą pojedynczej dyrektywy `#include`. Zwykle tak włączane pliki mają rozszerzenie `.h` i są nazywane plikami *nagłówkowymi* lub *włączanymi*.

Załóżmy, że piszemy zestaw programów wykonujących przeliczenia miar. Przydatny będzie zestaw stałych potrzebnych podczas tych konwersji:

```
#define INCHES_PER_CENTIMETER   0.394
#define CENTIMETERS_PER_INCH    1 / INCHES_PER_CENTIMETER

#define QUARTS_PER_LITER        1.057
#define LITERS_PER_QUART        1 / QUARTS_PER_LITER

#define OUNCES_PER_GRAM         0.035
#define GRAMS_PER_OUNCE         1 / OUNCES_PER_GRAM
```

Załóżmy, że wszystkie powyższe definicje mamy w pliku *metric.h*. Jeśli jakiś program potrzebuje definicji z tego pliku, wystarczy, że użyje dyrektywy:

```
#include "metric.h"
```

Dyrektywa ta musi wystąpić przed odwołaniem się do jakiejkolwiek definicji z pliku *metric.h*; zwykle dyrektywy `#include` umieszcza się na początku pliku źródłowego. Preprocesor odszukuje podany plik i całą jego zawartość wstawia w miejsce wystąpienia dyrektywy `#include`. Wobec tego instrukcje z włączanego tak pliku są traktowane tak, jakby były wpisane bezpośrednio w danym programie.

Podwójne cudzysłowy otaczające nazwę pliku informują preprocesor, że wskazanego pliku ma szukać w jednym lub wielu katalogach plików (zwykle najpierw przeszukiwany jest katalog, w którym jest sam program, ale zależy to od używanego sytemu). Jeśli plik nie zostanie znaleziony, preprocesor automatycznie przeszukuje pozostałe katalogi *systemowe*, jak to opisano dalej.

Jeśli zamiast cudzysłowów do otoczenia nazwy pliku zostaną użyte nawiasy kątowe:

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

preprocesor szuka włączanego pliku w specjalnym katalogu lub katalogach systemowych. Katalogi te także zależą od używanego sytemu. W systemach Unix (w tym Mac OS X) katalogiem tym jest */usr/include*; dzięki temu można znaleźć standardowy plik nagłówkowy *stdio.h*: */usr/include/stdio.h*.

Aby zobaczyć praktyczne użycie plików włączanych, wpiszemy sześć podanych wyżej definicji do pliku *metric.h*, a następnie przepiszemy i uruchomimy program 12.3.

Program 12.3. **Użycie dyrektywy #include**

```
/* Program pokazujący użycie dyrektywy #include.
   Uwaga: program zakłada, że w pliku metric.h
   znajdują się potrzebne definicje.        */

#include <stdio.h>
#include "metric.h"

int main (void)
{
    float  liters, gallons;

    printf ("*** Litry na galony ***\n\n");
    printf ("Podaj liczbę litrów: ");
    scanf ("%f", &liters);

    gallons = liters * QUARTS_PER_LITER / 4.0;
    printf ("%g litrów = %g galonów\n", liters, gallons);

    return 0;
}
```

Program 12.3. **Wyniki**

```
*** Litry na galony ***

Podaj liczbę litrów: 55.75
55.75 litrów = 14.73 galonów
```

Powyższy program jest dość prosty, gdyż odwołujemy się tylko do jednej definicji z pliku *metric.h*. Jednak widać już, jak działa — po włączeniu definicji z *metric.h* dyrektywą `#include` możemy z tych definicji normalnie korzystać.

Jedną z przyjemniejszych cech plików włączanych jest to, że możemy wszystkie swoje definicje umieszczać w jednym miejscu, a zatem mamy gwarancję, że wszystkie programy odwołują się do tych samych wartości. Jeśli później w tych definicjach wykryjemy błąd, wystarczy go poprawić w jednym tylko miejscu; potem ponownie skompilować programy używające tych definicji, i to już wszystko.

Do pliku włączanego można wstawiać dowolne dyrektywy i instrukcje, nie tylko dyrektywy `#define`. Plików włączanych używa się do umieszczania w jednym miejscu definicji preprocesora, definicji struktur, deklaracji prototypów i deklaracji zmiennych globalnych; jest to przejaw dobrej praktyki programowania.

Ostatnia informacja dotycząca plików włączanych jest taka: włączanie plików można zagnieżdżać, czyli jeden plik włączany sam może zawierać inny taki plik, ten — następny i tak dalej.

Systemowe pliki włączane

Jak wspominaliśmy wcześniej, plik `<stddef.h>` zawiera definicję `NULL` używaną często do sprawdzania, czy wskaźnik jest pusty. Wcześniej wspomnieliśmy jeszcze, że plik nagłówkowy `<math.h>` zawiera definicję stałej `M_PI` podającej przybliżenie liczby π.

Plik nagłówkowy `<stdio.h>` zawiera deklaracje procedur wejścia i wyjścia ze standardowej biblioteki i jest opisany szczegółowo w rozdziale 15. Plik ten trzeba włączyć zawsze, kiedy w programie chcemy użyć którejś z procedur wejścia-wyjścia.

Dwa kolejne przydatne systemowe pliki włączane to `<limits.h>` i `<float.h>`. Pierwszy z nich zawiera wartości zależne od używanego systemu, określające wielkości różnych znakowych i całkowitych typów danych. Na przykład maksymalna wielkość liczby typu `int` kryje się pod nazwą `INT_MAX`, a maksymalna wartość typu `unsigned long int` — pod nazwą `ULONG_MAX`.

Plik nagłówkowy `<float.h>` zawiera informacje o typach zmiennoprzecinkowych. Przykładowo stała `FLT_MAX` określa maksymalną wielkość zmiennoprzecinkową, a `FLT_DIG` — liczbę znaczących cyfr dziesiętnych dla zmiennych typu `float`.

Inne systemowe pliki włączane zawierają deklaracje prototypów różnych funkcji bibliotecznych z biblioteki systemowej. Przykładowo plik `<string.h>` zawiera prototypy funkcji bibliotecznych realizujących działania na łańcuchach znakowych, takie jak kopiowanie, porównywanie i złączanie.

Więcej szczegółów na temat tych plików nagłówkowych podajemy w dodatku B.

Kompilacja warunkowa

Preprocesor języka C umożliwia tak zwaną *kompilację warunkową*. Często służy ona do tworzenia jednego programu źródłowego, który może być później kompilowany na różne systemy komputerowe. Często też używa się go do włączania i wyłączania poszczególnych instrukcji w programie, szczególnie instrukcji związanych z uruchamianiem programu, a pokazujących różnego rodzaju wartości pośrednie, czy śledzeniem jego działania.

Dyrektywy #ifdef, #endif, #else i #ifndef

Wcześniej w tym rozdziale pokazaliśmy, jak można zwiększyć przenośność funkcji `rotate` z rozdziału 11. Widzieliśmy, jak przydatne może być w tym użycie dyrektywy `#define`. Definicja:

```
#define kIntSize   32
```

posłużyła do wydzielenia zależności wartości typu `unsigned int` od konkretnej liczby bitów. Zauważyliśmy też, że takie oparcie się na liczbie bitów jest zbędne, gdyż program sam może określić liczbę bitów wartości `unsigned int`.

Niestety, czasem program musi korzystać z parametrów zależnych od konkretnego systemu — na przykład od nazw plików czy od konkretnych cech systemu operacyjnego.

Jeśli mamy duży program z wieloma tego typu zależnościami od poszczególnych elementów sprzętowych i programowych systemu, może okazać się, że przy przenoszeniu go na inne systemy będziemy musieli dokonać wielu zmian (dlatego takie zależności należy ograniczać do niezbędnego minimum).

Opisany problem, związany z koniecznością zmian wielu definicji, możemy ograniczyć, korzystając z możliwości kompilacji warunkowej preprocesora. Takie oto proste dyrektywy:

```
#ifdef  UNIX
#   define  DATADIR    "/uxn1/data"
#else
#   define  DATADIR    "\usr\data"
#endif
```

spowodują, że stała `DATADIR` będzie miała wartość `"/uxn1/data"`, jeśli wcześniej zdefiniowany zostanie symbol `UNIX`, i `"\usr\data"` w przeciwnym wypadku. Jak widać, możemy po znaku `#` zaczynającym dyrektywy preprocesora wstawiać spacje.

Dyrektywy `#ifdef`, `#else` i `#endif` zachowują się zgodnie ze zdrowym rozsądkiem. Jeśli podczas kompilacji programu za pomocą dyrektywy `#define` zdefiniowany będzie symbol podany w dyrektywie `#ifdef`, kompilowany będzie kod znajdujący się zaraz za `#ifdef`; w przeciwnym razie kod ten zostanie pominięty, a kompilowany będzie kod znajdujący się za `#else`, `#elif` lub `#endif`. Aby na poziomie preprocesora zdefiniować symbol `UNIX`, wystarcza dyrektywa:

```
#define  UNIX    1
```

lub nawet sama:

```
#define  UNIX
```

Większość kompilatorów pozwala definiować nazwy na poziomie preprocesora w chwili kompilowania programu za pomocą specjalnej opcji kompilator. Na przykład podczas wywołania opcji kompilator `gcc` następująco:

```
gcc –D UNIX program.c
```

zdefiniowana zostanie stała `UNIX`, wobec czego wyrażenie `#ifdef UNIX` w pliku *program.c* zostanie spełnione (zauważmy, że `–D UNIX` musi wystąpić *przed* nazwą programu). Technika ta umożliwia definiowanie nazw bez zmieniania czegokolwiek w kodzie programu.

Nazwie definiowanej w wierszu poleceń można też przypisać od razu wartość. Przykładowo:

```
gcc –D GNUDIR=/c/gnustep program.c
```

wywołuje kompilator `gcc`, definiując nazwę `GNUDIR` o wartości `/c/gnustep`.

Unikanie wielokrotnego włączania plików nagłówkowych

Dyrektywę `#ifndef` stosuje się podobnie jak dyrektywę `#ifdef`, tyle że objęte nią instrukcje są kompilowane wtedy, gdy podany symbol *nie jest* zdefiniowany. Dyrektywa ta często służy do unikania wielokrotnego włączania tego samego pliku. Chcemy na przykład w plikach nagłówkowych zagwarantować sobie, że będą one tylko raz włączone do programu — definiujemy w nich niepowtarzalny identyfikator, który potem będziemy sprawdzać. Weźmy pod uwagę następujące dyrektywy:

```
#ifndef _MYSTDIO_H
#define _MYSTDIO_H
  ...
#endif /* _MYSTDIO_H */
```

Załóżmy, że powyższy kod umieściliśmy w pliku *mystdio.h*. Jeśli w programie dodamy dyrektywę:

```
#include "mystdio.h"
```

dyrektywa `#ifndef` sprawdzi, czy zdefiniowano nazwę `_MYSTDIO_H`. Po stwierdzeniu, że nie, preprocesor włączy do kompilacji kod aż do dyrektywy `#endif`. Spowoduje to włączenie wszystkich instrukcji z naszego pliku nagłówkowego. Zauważmy, że już w drugim wierszu włączanego pliku definiujemy symbol `_MYSTDIO_H` i jeśli ponownie włączymy nasz plik nagłówkowy, symbol ten będzie zdefiniowany, wobec czego cała treść pliku tym razem nie zostanie włączona. Dzięki temu nasz plik będzie dołączony do programu tylko raz.

Opisana metoda jest stosowana w systemowych plikach nagłówkowych — warto zajrzeć tam i sprawdzić!

Dyrektywy preprocesora #if i #elif

Dyrektywa preprocesora `#if` pozwala w ogólniejszy sposób sterować kompilacją warunkową. Dyrektywy `#if` można używać do sprawdzenia, czy stałe wyrażenie daje wartość różną od zera. Jeśli tak, do kompilacji włączane są dalsze wiersze, aż do dyrektywy `#else`, `#elif` lub `#endif`. W przeciwnym razie są one pomijane. W ramach przykładu załóżmy, że definiujemy nazwę `OS` o wartości `1` — jeśli systemem operacyjnym jest Macintosh OS, `2` — jeśli systemem jest Windows, `3` — w przypadku Linuksa, i tak dalej. Moglibyśmy teraz napisać ciąg dyrektyw warunkowo kompilujący nasz program w zależności od wartości nazwy `OS`:

```
#if    OS == 1  /* Mac OS */
   ...
#elif  OS == 2  /* Windows */
   ...
#elif  OS == 3  /* Linux */
   ...
#else
   ...
#endif
```

W przypadku większości kompilatorów stałą preprocesora OS możemy zdefiniować w wierszu poleceń, korzystając z omawianej wcześniej opcji –D. Polecenie:

```
gcc –D OS=2 program.c
```

skompiluje plik *program.c* i zdefiniuje jednocześnie nazwę OS o wartości 2. Spowoduje to skompilowanie programu na potrzeby systemu Windows.

W dyrektywach #if można też używać specjalnego operatora:

```
defined (nazwa)
```

Zestawy dyrektyw:

```
#if  defined (DEBUG)
   ...
#endif
```

oraz:

```
#ifdef  DEBUG
   ...
#endif
```

działają tak samo. Dyrektywy:

```
#if defined (WINDOWS) || defined (WINDOWSNT)
#   define BOOT_DRIVE "C:/"
#else
#   define BOOT_DRIVE "D:/"
#endif
```

definiują nazwę BOOT_DRIVE o wartości "C:/", jeśli zdefiniowano symbol WINDOWS lub WINDOWSNT i "D:/" w przeciwnym wypadku.

Dyrektywa #undef

Czasami przydatne byłoby usunięcie definicji jakiejś nazwy. Służy do tego dyrektywa #undef. Aby usunąć definicję symbolu nazwa, piszemy:

```
#undef  nazwa
```

Zatem dyrektywa:

```
#undef  WINDOWS_NT
```

usuwa definicję nazwy WINDOWS_NT. Jeśli dalej pojawi się dyrektywa #ifdef WINDOWS_NT lub #if defined (WINDOWS_NT), jej warunek nie będzie spełniony.

Na tym kończymy omawianie preprocesora. Pokazaliśmy, jak go wykorzystać, aby programy były łatwiejsze do czytania, pisania i modyfikowania. Pokazaliśmy, jak można użyć plików włączanych do grupowania definicji i deklaracji w pliku wspólnym dla wielu programów. Niektóre dyrektywy preprocesora, nieopisane w tym rozdziale, omawiamy w dodatku A.

W następnym rozdziale więcej powiemy o typach danych i konwersjach tych typów. Najpierw jednak pora na zestaw ćwiczeń.

Ćwiczenia

1. Przepisz i uruchom trzy programy pokazane w tym rozdziale, pamiętając o wpisaniu także pliku włączonego związanego z programem 12.3. Uzyskane wyniki porównaj z wynikami pokazanymi w tekście.
2. W używanym systemie znajdź pliki nagłówkowe `<stdio.h>`, `<limits.h>` i `<float.h>` (w systemach Unix należy ich szukać w katalogu */usr/include*). Sprawdź, co zawierają.
3. Zdefiniuj makro `MIN` podające mniejszą z dwóch wartości. Napisz program sprawdzający tę definicję.
4. Zdefiniuj makro `MAX3` podające największą z trzech wartości. Napisz program sprawdzający tę definicję.
5. Napisz makro `SHIFT` działające tak samo jak funkcja `shift` z programu 11.3.
6. Napisz makro `IS_UPPER_CASE` zwracające wartość różną od zera, jeśli podany mu znak jest wielką literą.
7. Napisz makro `IS_ALPHABETIC` zwracające wartość różną od zera, jeśli przekazany mu znak jest literą. Niech makro to korzysta z makra `IS_LOWER_CASE` z tego rozdziału i z makra `IS_UPPER_CASE` z ćwiczenia 6.
8. Napisz makro `IS_DIGIT` zwracające wartość różną od zera, jeśli przekazany mu znak jest cyfrą od `'0'` do `'9'`. Na podstawie tego makra napisz kolejne — `IS_SPECIAL` — zwracające niezerową wartość, jeśli podany znak jest znakiem specjalnym, czyli nie jest cyfrą ani literą. Skorzystaj z makra `IS_ALPHABETIC` z ćwiczenia 7.
9. Napisz makro `ABSOLUTE_VALUE` wyliczające wartość bezwzględną swojego parametru. Zapewnij prawidłowe wyliczanie wyrażeń typu:

   ```
   ABSOLUTE_VALUE (x + delta)
   ```

10. Rozważ definicję makra `printint` z tego rozdziału:

    ```
    #define printint(n)  printf ("%i\n", x ## n)
    ```

    Czy za jego pomocą można wyświetlić wartości 100 zmiennych od `x1` do `x100`? Odpowiedź uzasadnij.

    ```
    for (i = 1; i < 100; ++i)
        printx (i);
    ```

    Przetestuj funkcje z biblioteki standardowej, będące odpowiednikami naszych makr z ćwiczeń 6., 7. i 8. Odpowiednie funkcje to `isupper`, `isalpha` i `isdigit`. Aby użyć tych funkcji, trzeba włączyć do programu *systemowy* plik nagłówkowy `<ctype.h>`.

13

Jeszcze o typach danych — wyliczenia, definicje typów oraz konwersje typów

W tym rozdziale omawiamy typy danych, o których jeszcze nie mówiliśmy. Powiemy o *wyliczeniowym* typie danych, a także o instrukcji `typedef`, która umożliwia przypisanie podstawowym lub pochodnym typom danych ich własnej nazwy. W końcu poznamy dokładne zasady stosowane przez kompilator przy konwersji typów danych w wyrażeniach. Choć trzy poruszane w tym rozdziale tematy są zróżnicowane, ich znajomość jest niezbędna do tego, by maksymalnie wykorzystać dane w programie. Oto lista omawianych zagadnień:

- sposoby posługiwania się wyliczeniami;
- tworzenie własnych etykiet dla istniejących typów danych języka C za pomocą instrukcji `typedef`;
- konwertowanie istniejących typów danych na inne.

Wyliczeniowe typy danych

Czy nie byłoby dobrze, gdybyśmy mogli zdefiniować zmienną i podać zestaw wartości, jakie mogą być w tej zmiennej przechowywane? Załóżmy na przykład, że mielibyśmy zmienną `myColor`, w której przechowywalibyśmy jeden z kolorów podstawowych: `red`, `yellow` lub `blue`. Tego typu możliwości dają wyliczeniowe typy danych.

Definicja wyliczeniowego typu danych zaczyna się od słowa kluczowego `enum`. Zaraz za tym słowem podana jest nazwa typu, potem lista identyfikatorów (zamknięta w nawiasy klamrowe) opisująca dopuszczalne wartości danego typu danych. Na przykład instrukcja:

```
enum primaryColor { red, yellow, blue };
```

definiuje typ danych `primaryColor`. Zmienne tego typu mogą mieć tylko wartości `red`, `yellow`, `blue` i *żadnych innych*. Przynajmniej tak jest w teorii. Próba przypisania takiej zmiennej innej wartości powoduje w niektórych kompilatorach błąd, ale inne kompilatory w ogóle tego nie sprawdzają.

Aby zadeklarować zmienną typu `enum primaryColor`, używamy słowa kluczowego `enum`, potem podajemy nazwę typu wyliczeniowego i listę zmiennych. Zatem instrukcja:

```
enum primaryColor  myColor, gregsColor;
```

definiuje dwie zmienne — `myColor` i `gregsColor` — typu `primaryColor`. Jedyne dopuszczalne wartości tych zmiennych to `red`, `yellow` i `blue`. Wobec tego dalej poprawne będą instrukcje:

```
myColor = red;
```

oraz:

```
if ( gregsColor == yellow )
    ...
```

Innym przykładem wyliczeniowego typu danych jest lista miesięcy — `enum month` — której wartościami mogą być nazwy miesięcy:

```
enum month  { styczeń, luty, marzec, kwiecień, maj, czerwiec,
              lipiec, sierpień, wrzesień, październik, listopad, grudzień };
```

Kompilator C traktuje identyfikatory wyliczeniowe jako stałe całkowite. Kolejnym nazwom z naszej listy przypisywane są następne wartości, poczynając od `0`. Jeśli nasz program zawiera następujące dwa wiersze:

```
enum month  thisMonth;
    ...
thisMonth = luty;
```

zmiennej `thisMonth` przypisywana jest wartość `1` (a nie nazwa `luty`) — `luty` to drugi identyfikator na liście wyliczeniowej.

Jeśli z którymś z identyfikatorów typu wyliczeniowego chcemy związać jakąś konkretną liczbę całkowitą, możemy to zrobić w definicji typu danych. Kolejne identyfikatory typu wyliczeniowego otrzymają następne liczby. Jeśli na przykład mamy definicję:

```
enum direction { góra, dół, lewo = 10, prawo };
```

typ wyliczeniowy `direction` będzie miał wartości `góra`, `dół`, `lewo` i `prawo`. Kompilator przypisze pierwszemu identyfikatorowi wartość `0`, drugiemu — `1`; trzeciemu, `lewo` — pokazaną wartość `10`, a ostatniemu, `prawo` — wartość `11`.

Program 13.1 to prosty przykład wykorzystujący wyliczeniowe typy danych. Pierwszemu identyfikatorowi przypisano wartość `1`, dzięki czemu wszystkim miesiącom odpowiadają wartości od `1` do `12`. Program odczytuje numer miesiąca, po czym w instrukcji `switch` sprawdza, jaki miesiąc został wybrany. Przypomnijmy, że kompilator traktuje wartości typu wyliczeniowego jako stałe całkowite, więc można ich użyć w instrukcji `switch`. Zmienna `days` otrzymuje wartość określającą liczbę dni w danym miesiącu; liczba ta jest wyświetlana i na tym kończy się działanie instrukcji `switch`. Dla lutego dodano specjalny test.

Program 13.1. **Użycie wyliczeniowych typów danych**

```
// Program pokazujący liczbę dni w poszczególnych miesiącach

#include <stdio.h>

int main (void)
{
    enum month  { styczen = 1, luty, marzec, kwiecien, maj, czerwiec,
                  lipiec, sierpien, wrzesien, pazdziernik, listopad, grudzien
 };
    enum month  aMonth;
    int         days;

    printf ("Podaj numer miesiąca: ");
    scanf ("%i", &aMonth);

    switch (aMonth) {
        case styczen:
        case marzec:
        case maj:
        case lipiec:
        case sierpien:
        case pazdziernik:
        case grudzien:
                 days = 31;
                 break;
        case kwiecien:
        case czerwiec:
        case wrzesien:
        case listopad:
                 days = 30;
                 break;
        case luty:
                 days = 28;
                 break;
        default:
                 printf ("niewłaściwy numer miesiąca\n");
                 days = 0;
                 break;
    }

    if ( days != 0 )
        printf ("Podany miesiąc ma %i dni.\n", days);

    if ( aMonth == luty )
        printf ("...albo i 29, jeśli rok jest przestępny\n");

    return 0;
}
```

Program 13.1. **Wyniki**

```
Podaj numer miesiąca: 5
Podany miesiąc ma 31 dni.
```

Program 13.1. **Wyniki (ponowne uruchomienie)**

```
Podaj numer miesiąca: 2
Podany miesiąc ma 28 dni.
...albo i 29, jeśli rok jest przestępny
```

Identyfikatory typu wyliczeniowego mogą mieć tę samą wartość. Na przykład w definicji:

```
enum switch  { no=0, off=0, yes=1, on=1 };
```

przypisanie zmiennej wartości `yes` lub `on` da tę samą wartość — 1.

Zmiennym typu wyliczeniowego można też bezpośrednio przypisywać liczby całkowite, ale wtedy trzeba używać operatora rzutowania. Jeśli zatem zmienna typu `int` ma wartość 6, wyrażenie:

```
thisMonth = (enum month) (monthValue – 1);
```

jest poprawne i przypisuje zmiennej `thisMonth` wartość 5.

Pisząc programy wykorzystujące typy wyliczeniowe, raczej nie powinniśmy korzystać z tego, że wartości typów wyliczeniowych są traktowane jako liczby całkowite. Raczej należy traktować je jako całkiem niezależne typy danych. Jeśli musimy zmienić wartość przypisaną typowi wyliczeniowemu, robimy to tylko w miejscu zdefiniowania tego typu. Gdy dokonujemy jakichkolwiek założeń co do wartości przypisywanych poszczególnym wartościom typu wyliczeniowego, tracimy wszystkie zalety stosowania typów wyliczeniowych.

Kiedy definiujemy wyliczeniowe typy danych, dopuszczalne są takie modyfikacje, jak podczas definiowania struktur — może zostać pominięta nazwa typu danych, zmienne danego typu można deklarować w miejscu definiowania samego typu. Oto przykład pokazujący, jak to się robi:

```
enum { wschód, zachód, południe, północ } kierunek;
```

Mamy tu definicję nienazwanego typu wyliczeniowego z wartościami `wschód`, `zachód`, `południe` i `północ` oraz zmienną `kierunek` tego typu.

Definicje typów wyliczeniowych i zmiennych tych typów zachowują się analogicznie jak definicje i zmienne strukturalne — zdefiniowanie wyliczeniowego typu danych w bloku programu ogranicza zakres definicji do tegoż bloku. Z drugiej strony, zdefiniowanie typu wyliczeniowego na początku programu, poza jakąkolwiek funkcją, czyni tę definicję globalną dla całego pliku źródłowego.

Definiując wyliczeniowe typy danych, musimy zapewnić, że identyfikatory wartości będą niepowtarzalne, to znaczy nie będą takie same jak nazwy innych identyfikatorów i nazwy zmiennych definiowane w danym zakresie.

Instrukcja typedef

Język C umożliwia przypisywanie typowi danych alternatywnej nazwy. Służy do tego instrukcja `typedef`. Instrukcja:

```
typedef int Counter;
```

definiuje typ danych `Counter` równoważny typowi `int`. Dalej można normalnie definiować zmienne typu `Counter`:

```
Counter  j, n;
```

Kompilator języka C traktuje definicje zmiennych `j` i `n` z powyższej instrukcji jak zwykłe zmienne całkowitoliczbowe. Podstawową zaletą typów definiowanych za pomocą słowa kluczowego `typdef` jest poprawa czytelności kodu. Z definicji zmiennych jasno wynika, czemu w programie służą te zmienne. Zadeklarowanie ich normalnie, jako zmiennych typu `int`, nie wskazywałoby, do czego zamierzamy ich używać. Oczywiście ważny jest tu staranny dobór samoopisującej nazwy typu.

W wielu wypadkach instrukcję `typedef` można zastąpić dyrektywą `#define`; na przykład powyższy kod można by zamienić na:

```
#define  Counter  int
```

i uzyskalibyśmy bardzo podobny wynik. Jednak instrukcja `typedef` jest obsługiwana przez sam kompilator C, a nie przez preprocesor, więc instrukcja `typedef` jest bardziej elastyczna niż dyrektywa `#define` — swobodniej można dobierać nazwy definiowanych typów danych. Na przykład poniższa instrukcja `typedef`:

```
typedef  char  Linebuf [81];
```

definiuje typ `Linebuf` będący tablicą 81 znaków. Jeśli zdefiniujemy teraz zmienne tego typu:

```
Linebuf  text, inputLine;
```

zmienne `text` i `inputLine` będą tablicami zawierającymi po 81 znaków. Jest to równoważne deklaracji:

```
char  text[81], inputLine[81];
```

Zauważmy, że w tym wypadku nie można było zastąpić definicji typu `Linebuf` dyrektywą preprocesora `#define`.

Poniższa instrukcja `typedef` definiuje typ danych `StringPtr` jako wskaźnik na znak, `char`:

```
typedef  char *StringPtr;
```

Zmienne deklarowane dalej jako zmienne typu `StringPtr`:

```
StringPtr  buffer;
```

traktowane są przez kompilator C jako wskaźniki znakowe.

Jeśli za pomocą instrukcji `typedef` chcemy zdefiniować nowy typ danych, dokonujemy tego w trzech krokach:

1. Zapisujemy instrukcję tak, jak w przypadku deklarowania zmiennej pożądanego typu.
2. Tam, gdzie normalnie pojawia się nazwa zmiennej, wstawiamy nazwę nowego typu.
3. Przed całą definicją umieszczamy słowo kluczowe `typedef`.

Oto przykład tej procedury — definiujemy typ danych `Date` jako strukturę zawierającą pola całkowite `day`, `month` i `year`. W definicji typu, przed średnikiem, zamiast nazwy zmiennej podajemy nazwę typu, czyli `Date`. Wszystko poprzedzamy słowem kluczowym `typedef`:

```
typedef struct
    {
        int  day;
        int  month;
        int  year;
    } Date;
```

Teraz możemy już definiować zmienne typu `Date`:

```
Date  birthdays[100];
```

Tak więc zdefiniowaliśmy tablicę `birthdays` zawierającą 100 struktur `Date`.

Jeśli tworzymy programy składające się z więcej niż jednego pliku źródłowego (czym zajmiemy się w rozdziale 14.), warto wspólne definicje `typedef` umieszczać w odrębnych plikach włączanych potem do programu dyrektywą `#include`.

Załóżmy teraz, że korzystamy z pakietu graficznego, który musi obsługiwać odcinki, okręgi i tak dalej. Na pewno intensywnie będziemy korzystali z systemu współrzędnych. Oto instrukcja `typedef` definiująca typ danych `Point` będący strukturą zawierającą dwa pola typu `float`, `x` i `y`:

```
typedef struct
{
    float  x;
    float  y;
} Point;
```

Teraz możemy przejść do tworzenia własnej biblioteki graficznej, wykorzystującej taki właśnie typ danych `Point`. Na przykład deklaracja:

```
Point  origin = { 0.0, 0.0 }, currentPoint;
```

zdefiniuje zmienne `origin` i `currentPoint` typu `Point` oraz ustawi pola `x` i `y` zmiennej `origin` na 0.0.

Oto funkcja `distance` wyliczająca odległość dwóch punktów:

```
#include <math.h>

double  distance (Point p1, Point p2)
{
    double  diffx, diffy;

    diffx = p1.x – p2.x;
    diffy = p1.y – p2.y;

    return sqrt (diffx * diffx + diffy * diffy);
}
```

Jak wspominaliśmy już wcześniej, `sqrt` jest funkcją wyliczającą pierwiastek kwadratowy; znajdziemy ją w bibliotece standardowej. Deklarację tej funkcji umieszczono w pliku nagłówkowym *math.h*, stąd powyższa dyrektywa `#include`.

Pamiętajmy, że instrukcja `typedef` nie tworzy nowego typu danych, tylko nową nazwę typu danych. Wobec tego zmienne `j` i `n` typu `Counter` zdefiniowane na początku tego podrozdziału będą przez kompilator traktowane jak najzwyklejsze zmienne typu `int`.

Konwersje typów danych

W rozdziale 3., kiedy po raz pierwszy omawialiśmy typy danych, krótko powiedzieliśmy, że podczas wyliczania wyrażeń system niejawnie wykonuje pewne konwersje. Interesowały nas typy danych `float` i `int`. Widzieliśmy, jak podczas wykonywania działań na wartościach `int` i `float` te pierwsze były automatycznie konwertowane do typu `float`.

Widzieliśmy też, jak można użyć operatora rzutowania typów do wymuszania konwersji. Zatem w instrukcji:

```
srednia = (float) suma / n;
```

wartość zmiennej `suma` jest konwertowana na typ danych `float` i dopiero wtedy wykonywane jest dzielenie. Dzięki temu dzielenie zawsze będzie traktowane jako działanie zmiennoprzecinkowe.

Podczas wyliczania wyrażeń zawierających dane różnych typów kompilator języka C postępuje zgodnie ze ściśle określonymi zasadami.

Poniżej zestawiono konwersje wykonywane podczas wyliczania wyrażeń w takiej kolejności, w jakiej są one faktycznie robione:

1. Jeśli którykolwiek argument jest typu `long double`, drugi też jest konwertowany na ten typ; wynik także jest typu `long double`.
2. Jeśli którykolwiek argument jest typu `double`, drugi też jest konwertowany na ten typ; wynik także jest typu `double`.
3. Jeśli którykolwiek argument jest typu `float`, drugi też jest konwertowany na ten typ; wynik także jest typu `float`.
4. Jeśli którykolwiek argument jest typu `_Bool`, `char`, `short int`, `bit field` lub typu wyliczeniowego, jest konwertowany na typ danych `int`.
5. Jeśli którykolwiek argument jest typu `long long int`, drugi też jest konwertowany na ten typ; wynik także jest typu `long long int`.
6. Jeśli którykolwiek argument jest typu `long int`, drugi też jest konwertowany na ten typ; wynik także jest typu `long int`.
7. Jeśli doszliśmy aż do tego punktu, oba argumenty są typu `int`, wynik także będzie typu `int`.

Jest to nieco uproszczony opis konwersji argumentów w wyrażeniu. Zasady komplikują się, kiedy część argumentów nie ma znaku (jest zadeklarowana jako `unsigned`). Pełny opis zasad konwersji podajemy w dodatku A.

Jak widać, kiedy dochodzimy do zapisu „wynik także jest tego typu", proces konwersji dobiegł końca.

W ramach przykładu pokażemy, jakie konwersje zostaną wykonane przy wyliczaniu wartości wyrażenia, w którym zmienna `f` jest typu `float`, `i` typu `int`, `l` typu `long int`, a `s` typu `short int`:

```
f * i + l / s
```

Najpierw zajmijmy się mnożeniem `f` przez `i`, czyli mnożeniem wartości typu `float` przez wartość typu `int`. Z punktu 3. powyższego zestawienia wynika, że skoro `f` jest typu `float`, drugi argument — `i` — także zostanie przekształcony na typ `float` i takiego typu będzie wynik mnożenia.

Następnie mamy dzielenie zmiennej `l` przez zmienną `s`, czyli dzielenie wartości typu `long int` przez wartość typu `short int`. Zgodnie z punktem 4. wartość `short int` zostanie przekształcona do typu `int`. Dalej, z punktu 6. wynika, że skoro jeden z argumentów (`l`) jest typu `long int`, drugi też zostanie do tego typu przekształcony; wynik dzielenia też będzie typu `long int`. Wobec tego dzielenie da wartość typu `long int`, a ułamkowa część ilorazu zostanie odrzucona.

W końcu z punktu 3. wynika, że jeśli jeden z argumentów działania jest typu `float` (wynik mnożenia `f*i`), drugi też zostanie przekształcony na typ `float`. Wynik też będzie typu `float`. Zatem po podzieleniu `l` przez `s` iloraz przekształcimy na typ `float` i dodamy do iloczynu `f` i `i`. Ostateczny wynik całego wyrażenia będzie zatem miał typ `float`.

Pamiętajmy, że zawsze można użyć operatora rzutowania typu do jawnego wymuszenia konwersji typów. W ten sposób można decydować o sposobie wyliczania wartości wyrażeń.

Gdybyśmy na przykład w powyższym wyrażeniu nie chcieli, aby odrzucona została część ułamkowa ilorazu `l` i `s`, moglibyśmy za pomocą operatora rzutowania typu promować jeden z argumentów do typu `float`:

```
f * i + (float) l / s
```

W takim wyrażeniu liczba `l`, przed dzieleniem, zostałaby przekonwertowana do typu `float`, gdyż operator rzutowania ma wyższy priorytet niż operator dzielenia. Skoro jeden z argumentów dzielenia byłby typu `float`, drugi z nich — `s` — także zostałby promowany do tego typu. Iloraz też byłby typu `float`.

Znak wartości

Kiedy wartość typu `signed int` lub `signed short int` jest konwertowana na większą liczbę całkowitą, pozostaje liczbą ze znakiem. Jeśli zatem wartość typu `short int` jest równa –5, po konwersji do typu `long int` też będzie miała wartość –5. Jeśli promujemy typ wartości bez znaku, zgodnie ze zdrowym rozsądkiem, nie jest dodawany znak liczby.

W niektórych systemach dane typu `char` są traktowane jako wartości ze znakiem. Wobec tego, kiedy taka informacja jest konwertowana na typ `int`, jest to typ ze znakiem. Jeśli używamy tylko podstawowego zestawu znaków ASCII, nie stanowi to żadnego problemu. Kiedy jednak mamy do czynienia ze znakiem spoza tego zestawu, problem może się pojawić. W komputerach Mac na przykład stała znakowa `'\377'` jest konwertowana na wartość –1, gdyż wartość ta potraktowana jako wartość ze znakiem jest interpretowana jako –1.

Przypomnijmy, że język C pozwala definiować zmienne typu `char` z modyfikatorem `unsigned`. Dzięki temu można uniknąć ewentualnych problemów. Zmienna `unsigned char` w trakcie promowania jej typu nigdy nie stanie się wartością ze znakiem, zatem także w przypadku wartości 8-bitowych będzie ona większa od zera. Normalnie wartości 8-bitowe mieszczą się w zakresie od –128 do +127. Zmienna typu `unsigned char` ma wartości z zakresu od 0 do 255.

Jeśli chcemy wymusić konwersję zmiennej znakowej na wartość ze znakiem, musimy ją zadeklarować jako `signed char`. Wtedy podczas konwersji uzyskamy liczbę całkowitą ze znakiem — także na komputerach, dla których nie jest to zachowanie domyślne.

Konwersja parametrów

Używaliśmy dotąd prototypów wszystkich pisanych przez nas funkcji. W rozdziale 7. powiedzieliśmy, że jest to bardzo rozsądne rozwiązanie, gdyż fizycznie funkcja może zostać zdefiniowana przed swoim wywołaniem lub po nim, a nawet w innym pliku źródłowym. Wspomnieliśmy też, że kompilator automatycznie konwertuje typy parametrów funkcji na typy oczekiwane przez funkcję. Jedynym sposobem poinformowania kompilatora o tym, jakich typów funkcja oczekuje, jest właśnie użycie albo samej definicji funkcji, albo deklaracji prototypu.

Przypomnijmy, że jeśli kompilator nie widzi ani definicji funkcji, ani jej prototypu, zakłada, że funkcja zwraca wartość typu `int`. Kompilator zakłada też, że wszelkie parametry typów `_Bool`, `char` i `short` są typu `int`, a parametry typu `float` są typu `double`.

Załóżmy na przykład, że kompilator znajdzie w programie następujący kod:

```
float  x;
    ...
y = absoluteValue (x);
```

Jeśli wcześniej nie wystąpiła definicja ani deklaracja funkcji `absoluteValue`, kompilator wygeneruje kod konwertujący wartość zmiennej `x` typu `float` na typ `double` i dopiero tę wartość przekaże do funkcji. Kompilator założy też, że funkcja zwraca wartość typu `int`.

Jeśli funkcja `absoluteValue` w innym pliku źródłowym zostanie zdefiniowana następująco:

```
float absoluteValue (float x)
{
    if ( x < 0.0 )
        x = –x;

    return x;
}
```

to mamy kłopot. Po pierwsze, funkcja zwraca wartość `float`, a kompilator zakładał, że jest to wartość `int`. Po drugie, funkcja oczekuje parametru typu `float`, a kompilator przekazuje wartość typu `double`.

Pamiętajmy, żeby zawsze podawać prototypy używanych funkcji. Wtedy kompilator nie będzie robił błędnych założeń dotyczących typów zwracanych przez funkcje i typów ich parametrów.

Teraz, kiedy wiemy już nieco więcej o typach danych, czas zająć się podziałem programów na wiele plików źródłowym. Zajmiemy się tym w rozdziale 14., ale teraz utrwalmy sobie nowe wiadomości, wykonując stosowne ćwiczenia.

Ćwiczenia

1. Zdefiniuj za pomocą instrukcji `typedef` typ danych `FunctionPtr` będący wskaźnikiem do bezparametrowych funkcji zwracających wartość typu `int`. Informacje, jak deklarować tego typu zmienne, znajdziesz w rozdziale 10.
2. Napisz funkcję `monthName`, która jako parametr będzie pobierać wartość typu `enum month` (zgodnie z definicją z tego rozdziału) i będzie zwracać wskaźnik na łańcuch znakowy z nazwą miesiąca. Dzięki temu można będzie wyświetlać zmienne typu `enum month` za pomocą instrukcji:

   ```
   printf ("%s\n", monthName (aMonth));
   ```

3. Niech dane będą następujące deklaracje zmiennych:

   ```
   float     f = 1.00;
   short int i = 100;
   long int  l = 500L;
   double    d = 15.00;
   ```

   Mając siedem reguł konwersji typów, które podano wcześniej w tym rozdziale, wyznacz typ i wartość następujących wyrażeń:

   ```
   f + i
   l / d
   i / l + f
   l * i
   f / 2
   i / (d + f)
   l / (i * 2.0)
   l + i / (double) l
   ```

14

Praca z większymi programami

Programy, jakimi dotąd się zajmowaliśmy, były bardzo małe i dość proste. Niestety, programy, które służą rozwiązywaniu praktycznych problemów, zwykle nie są ani małe, ani proste. W tym rozdziale nauczymy się, jak sobie radzić z takimi programami. Jak zobaczymy, język C zawiera cechy niezbędne do skutecznej pracy nad dużymi aplikacjami. Co więcej, można użyć różnych programów pomocniczych, o których wspomnimy w tym rozdziale, aby sobie ułatwić pracę nad dużymi projektami.

Niektóre z tematów opisanych w tym rozdziale dotyczą konkretnego systemu operacyjnego lub środowiska programistycznego, ale warto zapoznać się z ogólnymi koncepcjami na wypadek, gdyby w przyszłości trzeba było skorzystać z innego środowiska. Oto lista najważniejszych poruszanych zagadnień:

- dzielenie dużych programów na pliki;
- kompilowanie wielu plików do postaci jednego pliku wykonywalnego;
- praca ze zmiennymi zewnętrznymi;
- posługiwanie się plikami nagłówkowymi;
- udoskonalanie programów za pomocą narzędzi pomocniczych.

Dzielenie programu na wiele plików

Do tego momentu zawsze zakładaliśmy, że cały nasz program mieści się w pojedynczym pliku i taki plik edytujemy za pomocą edytora tekstowego — takiego jak *emacs*, *vim* czy jakiś edytor systemu Windows — potem kompilujemy i wykonujemy. W tym jednym pliku znajdowały się wszystkie funkcje programu poza, oczywiście, funkcjami systemowymi, jak `printf` czy `scanf`. Dołączaliśmy też standardowe pliki nagłówkowe, takie jak `<stdio.h>` czy `<stdbool.h>`, aby skorzystać ze zdefiniowanych tam funkcji. Tego typu rozwiązanie dobrze się sprawdza, kiedy mamy do czynienia z niewielkimi programami, czyli zawierającymi do około 100 instrukcji. Jeśli jednak program jest większy, przestaje

to wystarczać. W miarę jak wzrasta liczba instrukcji, wzrasta też czas edycji programu i w konsekwencji czas jego kompilacji. Mało tego, duże zadania programistyczne zwykle wymagają więcej niż jednego programisty. Gdybyśmy chcieli nadal wszystko trzymać w jednym pliku, choćby skopiowanym, to może się okazać, że nie sposób zapanować nad takim projektem.

Język C obsługuje programowanie modułowe, co oznacza, że nie wszystkie instrukcje danego programu muszą znaleźć się w jednym pliku. Wobec tego możemy wpisać kod jednego modułu do jednego pliku, drugiego modułu do innego pliku i tak dalej. W tym wypadku *moduł* oznacza pojedynczą funkcję lub szereg powiązanych ze sobą funkcji, które chcemy połączyć w pewną całość logiczną.

Osoby korzystające z okienkowych narzędzi do zarządzania projektami, takich jak CodeWarrior firmy Metroworks, Code::Blocks, Microsoft Visual Studio czy Xcode firmy Apple, mają ułatwione tworzenie programów wielomodułowych. Po prostu wskazują pliki wchodzące w skład bieżącego projektu, a resztą zajmuje się narzędzie. W następnym podrozdziale powiemy, jak sobie poradzić z wieloma plikami, gdy nie mamy do dyspozycji żadnego narzędzia ani zintegrowanego środowiska programistycznego (IDE). Zatem dalej będziemy zakładać, że program kompilujemy z wiersza poleceń, korzystając na przykład z programu *gcc* lub *cc*.

Kompilowanie wielu plików z wiersza poleceń

Załóżmy, że podzieliliśmy program na trzy moduły logiczne i wpisaliśmy do trzech plików: część do modułu pierwszego w pliku *mod1.c*, cześć do drugiego w pliku *mod2.c*, a funkcję `main` do pliku *main.c*. Jeśli chcemy poinformować system, że te trzy pliki składają się na jeden i ten sam program, po prostu podajemy kompilatorowi nazwy wszystkich trzech plików. Jeśli na przykład używamy programu *gcc*, to wywołanie:

```
$ gcc mod1.c mod2.c main.c –o dbtest
```

spowoduje skompilowanie kolejno plików *mod1.c*, *mod2.c* i *main.c*. Jeśli w plikach tych zostaną wykryte jakieś błędy, zobaczymy je osobno dla każdego pliku, na przykład:

```
mod2.c:10: mod2.c: In function 'foo':
mod2.c:10: error: 'i' undeclared (first use in this function)
mod2.c:10: error: (Each undeclared identifier is reported only once
mod2.c:10: error: for each function it appears in.)
```

Kompilator informuje, że w pliku *mod2.c* w 10. wierszu jest błąd w funkcji `foo`. Nie widzimy żadnych komunikatów dotyczących plików *mod1.c* i *main.c*, co oznacza, że w plikach tych nie wystąpiły błędy.

Zwykle, jeśli w module zostaną wykryte błędy, trzeba taki moduł poprawić[1]. Ponieważ w naszym wypadku błąd został wykryty w module *mod2.c*, edytujemy tylko ten plik. Następnie możemy ponownie przekompilować nasze moduły:

[1] Błąd może też wynikać na przykład z problemów z plikiem nagłówkowym — wtedy modułu nie należy edytować, ale trzeba edytować właśnie plik nagłówkowy.

```
$ gcc mod1.c mod2.c main.c –o dbtest
$
```

Nie zostały już zgłoszone żadne błędy, tworzony jest plik wykonywalny *dbtest*.

Normalnie kompilator generuje pliki pośrednie dla każdego kompilowanego pliku źródłowego. Domyślnie plikowi źródłowemu *mod.c* odpowiada plik pośredni *mod.o* (większość kompilatorów systemu Windows zachowuje się podobnie, ale czasem zamiast plików pośrednich z rozszerzeniem *.o* tworzone są pliki z rozszerzeniem *.obj*). Zwykle po zakończeniu kompilacji pliki pośrednie są automatycznie usuwane. Niektóre kompilatory (w tym standardowy kompilator C systemu Unix) zostawiają te pliki, chyba że kompilujemy po jednym pliku naraz. Jest to potem wykorzystywane w ten sposób, że ponownie kompilowane są już tylko te pliki, które zmieniły się od ostatniej kompilacji. Wobec tego, że w powyższym przykładzie pliki *mod1.c* i *main.c* nie zgłosiły błędów kompilacji, po zakończeniu działania programu *gcc* ich pliki pośrednie — *mod1.o* i *main.o* — pozostaną na dysku. W ten sposób kompilator może sprawdzić, czy plik źródłowy — *mod.c* — został zmieniony od ostatniej kompilacji, czyli od chwili utworzenia pliku *mod.o*. Jeśli kompilator nie usuwa plików pośrednich, możemy użyć następującego polecenia (tym razem korzystamy z kompilatora *cc*):

```
$ cc mod1.o mod2.c main.o –o dbtest
```

Zatem nie tylko rezygnujemy z edycji *mod1.c* i *main.c*, lecz także nie kompilujemy ich ponownie.

Jeśli używany przez nas kompilator automatycznie usuwa pliki pośrednie *.o*, nadal możemy skorzystać z zalet kompilacji przyrostowej; musimy skompilować każdy moduł osobno i użyć opcji –c. Opcja ta sprawia, że kompilator nie konsoliduje naszego programu (czyli nie tworzy pliku wykonywalnego), a za to zachowuje pliki pośrednie. Jeśli zatem napiszemy:

```
$ gcc –c mod2.c
```

skompilowany zostanie plik *mod2.c*, a uzyskany kod pośredni będzie wstawiony do pliku *mod2.o*.

Jeśli zatem chcemy skompilować trzymodułowy program *dbtest*, wykorzystując kompilację przyrostową, wywołujemy kolejno:

```
$ gcc –c mod1.c                    Kompilacja mod1.c => mod1.o
$ gcc –c mod2.c                    Kompilacja mod2.c => mod2.o
$ gcc –c main.c                    Kompilacja main.c => main.o
$ gcc mod1.o mod2.o main.o –o dbtest   Tworzenie pliku wykonywalnego
```

Każdy z trzech modułów jest kompilowany osobno. Jak widać, nie wystąpiły żadne błędy kompilacji; gdyby się jakieś pojawiły, trzeba by odpowiedni moduł poprawić i ponownie skompilować. Ostatni wiersz:

```
$ gcc mod1.o mod2.o main.o –o dbtest
```

zawiera jedynie pliki pośrednie, nie ma w nim natomiast plików źródłowych. W takim wypadku pliki pośrednie są po prostu łączone w wykonywalny plik wynikowy — *dbtest*.

Jeśli powyższy przykład rozszerzymy na programy składające się z wielu modułów, zobaczymy, że możemy wygodnie kompilować naprawdę duże programy, bazując na opisanym mechanizmie kompilacji przyrostowej. Na przykład polecenia:

```
$ gcc –c legal.c        Kompilacja pliku legal.c, umieszczenie wyniku w legal.o
$ gcc legal.o makemove.o exec.o enumerator.o evaluator.o display.o –o superchess
```

pozwalają skompilować sześciomodułowy program, w którym tylko moduł *legal.c* wymaga powtórnej kompilacji.

Jak zobaczymy pod koniec tego rozdziału, kompilacja przyrostowa może być zautomatyzowana za pomocą narzędzia *make*. Narzędzia IDE wspomniane na początku tego rozdziału mają wbudowaną wiedzę o tym, co wymaga ponownej kompilacji.

Komunikacja między modułami

Do zapewnienia sprawnej komunikacji między modułami rozmieszczonymi w osobnych plikach można wykorzystać kilka sposobów. Jeśli funkcja z danego pliku musi wywołać funkcję z innego pliku, wywołanie to wykonuje się normalnie, normalnie też przekazuje się parametry. Oczywiście w pliku, w którym funkcja jest wywoływana, zawsze trzeba pamiętać o *włączeniu prototypu funkcji, aby kompilator znał typy jej parametrów i typ wartości zwracanej*. Jak już mówiliśmy w rozdziale 13., kompilator wobec braku informacji o funkcji zakłada, że zwraca ona wartość typu `int`, wszystkie parametry typów `short` i `char` konwertuje do typu `int`, a typu `float` — do typu `double`.

Trzeba pamiętać, że jeśli nawet podamy kompilatorowi wiele modułów naraz, to i tak każdy z tych modułów kompilowany jest pojedynczo. Oznacza to, że jeden moduł „nie wie" nic o definicjach struktur, typach funkcji czy parametrach z innych modułów. Tylko na programiście spoczywa zapewnienie kompilatorowi potrzebnych informacji.

Zmienne zewnętrzne

Funkcje umieszczone w osobnych plikach mogą komunikować się za pośrednictwem *zmiennych zewnętrznych*, które stanowią rozwinięcie pomysłu zmiennych globalnych omawianych w rozdziale 7.

Zmienna zewnętrzna to taka zmienna, której wartość jest dostępna i może być zmieniana w innym module. W module, który chce do takiej zmiennej sięgnąć, zmienną normalnie deklarujemy, umieszczając jednak przed tą deklaracją słowo kluczowe `extern`. W ten sposób system jest informowany, że korzystamy ze zmiennej globalnej z innego pliku.

Załóżmy, że chcemy zdefiniować zmienną typu `int` o nazwie `moveNumber`, której wartość będzie potrzebna w funkcji znajdującej się w innym pliku. W rozdziale 7. dowiedzieliśmy się, że jeśli napiszemy:

```
int  moveNumber = 0;
```

na początku programu, *poza* jakąkolwiek funkcją, to taka zmienna będzie dostępna w dowolnej funkcji w całym module. Zmienna `moveNumber` jest w tej sytuacji zmienną globalną.

Taka sama definicja zmiennej `moveNumber` spowoduje, że zmienna ta będzie dostępna także w innych plikach. Powyższa instrukcja definiuje zmienną nie tyle globalną, ile *zewnętrzną* zmienną globalną. Aby odwołać się do jej wartości w innym module, musimy w tym module zadeklarować tę właśnie zmienną, poprzedzając deklarację słowem kluczowym `extern`:

```
extern int  moveNumber;
```

Teraz można sięgać do wartości `moveNumber` i modyfikować ją w każdym module, w którym pojawia się deklaracja, taka jak powyżej.

Gdy używamy zmiennych zewnętrznych, zawsze trzeba pamiętać o ważnej zasadzie. Zmienne muszą być gdzieś *zdefiniowane*. Robi się to na dwa sposoby. Po pierwsze, deklaruje się zmienną poza jakąkolwiek funkcją, ale deklaracji tej *nie* poprzedza się słowem kluczowym `extern`:

```
int  moveNumber;
```

W tym wypadku zmiennej można też przypisać wartość początkową, jak to robiliśmy poprzednio.

Drugi sposób to zadeklarowanie zmiennej zewnętrznej poza jakimikolwiek funkcjami, umieszczenie przed tą deklaracją słowa kluczowego `extern`, ale jednocześnie *jawne przypisanie wartości początkowej*:

```
extern  int moveNumber = 0;
```

Oba te sposoby wykluczają się wzajemnie.

Kiedy mamy do czynienia ze zmiennymi zewnętrznymi, słowo kluczowe `extern` może zostać pominięte tylko w jednym z plików źródłowych. Jeśli go nigdzie nie pominiemy, musimy w jednym miejscu przypisać zmiennej wartość początkową.

Spójrzmy na prosty programik pokazujący użycie zmiennych zewnętrznych. Załóżmy, że w pliku *main.c* mamy następujący kod:

```
#include <stdio.h>

int  i = 5;

int main (void)
{
    printf ("%i  ", i);

    foo ();

    printf ("%i\n", i);

    return 0;
}
```

Dzięki temu, że definicja zmiennej `i` jest definicją zmiennej globalnej, zmienna ta będzie dostępna we wszystkich modułach; wystarczy użyć odpowiedniej deklaracji `extern`. Załóżmy teraz, że w pliku *foo.c* wpiszemy następujące instrukcje:

```
extern int i;

void foo (void)
{
    i = 100;
}
```

Kompilując razem moduły *main.c* i *foo.c* poleceniem typu:

```
$ gcc main.c foo.c
```

a następnie wykonując program, uzyskujemy następujący wynik:

```
5  100
```

Wynika stąd, że funkcja `foo` może sięgnąć do zmiennej zewnętrznej `i` oraz może zmienić jej wartość.

Wartość zmiennej zewnętrznej `i` jest używana *wewnątrz* funkcji `foo`, więc moglibyśmy deklarację `i` też umieścić wewnątrz tej funkcji:

```
void foo (void)
{
    extern int  i;

    i = 100;
}
```

Jeśli natomiast dostępu do zmiennej `i` wymaga wiele funkcji z pliku *foo.c*, łatwiej zrobić deklarację `extern` raz, na początku pliku. Jeśli jednak tylko jedna funkcja sięga do takiej zmiennej albo funkcji takich jest niewiele, to trzeba pamiętać o jednym — napisanie osobnych deklaracji `extern` tylko wewnątrz tych funkcji, w których jest to niezbędne, poprawia organizację programu i pozwala ograniczyć dostęp do zmiennej tam, gdzie jest on niezbędny.

Deklarując tablicę zewnętrzną, nie musimy podawać jej wielkości. Zatem deklaracja:

```
extern char  text[];
```

pozwala odwołać się do tablicy znakowej `text` zdefiniowanej w innym miejscu. Tak jak w przypadku tablic przekazywanych jako parametry formalne, jeśli tablica zewnętrzna jest wielowymiarowa, trzeba podać wszystkiej jej wymiary z wyjątkiem pierwszego. Zatem deklaracja:

```
extern int  matrix[][50];
```

jest wystarczająca, jeśli mamy do czynienia z 50-kolumnową tablicą dwuwymiarową `matrix`.

Static a extern: porównanie zmiennych i funkcji

Wiemy już, że dowolna zmienna zdefiniowana poza funkcją jest nie tylko zmienną globalną, lecz także zmienną zewnętrzną. Często dochodzi do sytuacji, w których chcemy zdefiniować zmienną globalną, ale *nie* zewnętrzną. Innymi słowy, chcemy zdefiniować zmienną globalną lokalną dla danego modułu (pliku). Definiowanie zmiennej w ten sposób ma sens, jeśli nie będą jej potrzebowały żadne funkcje z innych modułów. W takim wypadku używamy zmiennej z kwantyfikatorem `static`.

Jeśli instrukcja:

```
static int  moveNumber = 0;
```

pojawi się poza jakąkolwiek funkcją, zmienna `moveNumber` będzie dostępna z dowolnego miejsca w pliku, w którym ta definicja się znajduje, ale *dla funkcji z innych plików będzie niedostępna*.

Jeśli mamy zdefiniować zmienną globalną, która nie musi być dostępna z innych plików, w deklaracji używamy słowa kluczowego `static`. Jest to eleganckie rozwiązanie programistyczne — deklaracja `static` lepiej opisuje sposób stosowania zmiennej i nie powoduje konfliktów między modułami, które „nieświadomie" używałyby różnych zmiennych globalnych o takich samych nazwach.

Jak wspominaliśmy wcześniej, można bezpośrednio wywoływać funkcje zdefiniowane w innych plikach. W przeciwieństwie do zmiennych, zbędne są jakiekolwiek dodatkowe zabiegi, używanie słowa `extern` i tak dalej.

Kiedy *definiujemy* funkcję, możemy ją zadeklarować jako `extern` lub `static`; domyślne jest ustawienie `extern`. Funkcje statyczne, ze słowem kluczowym `static`, mogą być wywoływane tylko z tego pliku, w którym występują. Jeśli zatem mamy funkcję `squareRoot`, to umieszczenie przed jej deklaracją słowa `static` uniemożliwia wywołanie tej funkcji spoza modułu, w którym jest deklaracja:

```
static double  squareRoot (double x)
{
    ...
}
```

Definicja funkcji `squareRoot` staje się lokalna dla pliku, w którym jest umieszczona.

To samo rozumowanie, które odnosiło się do sensowności stosowania zmiennych statycznych, odnosi się także do stosowania funkcji statycznych.

Na rysunku 14.1 podsumowano komunikację między różnymi modułami. Mamy tu dwa moduły, *mod1.c* i *mod2.c*.

W module *mod1.c* zdefiniowano dwie funkcje — `doSquare` i `main`. Funkcja `main` wywołuje `doSquare`, a ta z kolei wywołuje `square`. Ta ostatnia jest zdefiniowana w module `mod2.c`.

Jako że funkcja `doSquare` została zadeklarowana ze słowem kluczowym `static`, może być wywoływana tylko w module `mod1.c` i żadnym innym. W module *mod1.c* zdefiniowano dwie zmienne globalne — `x` i `result` — obie typu `double`. Zmienna `x` jest dostępna z dowolnego modułu konsolidowanego z *mod1.c*. Z kolei słowo kluczowe `static` przed definicją zmiennej `result` powoduje, że ta zmienna jest dostępna tylko dla funkcji z modułu *mod1.c* (czyli tylko dla `main` i `doSquare`).

```
double x;
static double result;

static void doSquare (void)
{
    double square (void);

    x = 2.0;
    result = square ();

}

int main (void)
{
    doSquare ();
    printf ("%g\n", result);

    return 0;
}
```

mod1.c

```
extern double x;

double square(void)
{

    return x * x;
}
```

mod2.c

Rysunek 14.1. Komunikacja między modułami programu

Podczas wykonywania programu procedura `main` wywołuje funkcję `doSquare`. Ta przypisuje zmiennej globalnej `x` wartość 2.0, następnie wywołuje funkcję `square`. Funkcja `square` jest zdefiniowana w innym pliku źródłowym — *mod2.c* — i nie zwraca wartości typu `int`, więc `doSquare` ma odpowiednią deklarację na początku.

Funkcja `square` jako swój wynik zwraca podniesioną do kwadratu wartość zmiennej globalnej `x`. Ponieważ funkcja `square` chce sięgnąć do wartości tej zmiennej, a zmienna ta jest zdefiniowana w innym pliku (*mod1.c*), w *mod2.c* pojawia się stosowana deklaracja `extern`. W tym wypadku nie ma znaczenia, czy deklaracja jest w funkcji `square`, czy na zewnątrz.

Wartość zwracana przez funkcję `square` w funkcji `doSquare` jest przypisywana zmiennej globalnej `result`, a ta jest zwracana do `main`. W funkcji `main` jest wyświetlana wartość zmiennej globalnej `result`. W naszym przykładzie na terminalu wyświetlona zostanie wartość 4.0 (jest to przecież wynik podniesienia do kwadratu wartości 2.0).

Jeśli coś w tym przykładzie jest jeszcze niejasne, należy go jeszcze raz gruntownie przeanalizować. Ten króciutki i nieprzydatny praktycznie program bardzo dobrze pokazuje, na czym polega komunikacja między modułami. Jego zrozumienie jest niezbędnym warunkiem nauczenia się skutecznego pisania dużych programów.

Wykorzystanie plików nagłówkowych

W rozdziale 12. omówiliśmy pojęcie plików nagłówkowych. Jak powiedzieliśmy, można wszystkie często wykorzystywane definicje pogrupować w taki plik i potem plik ten włączać tam, gdzie chcemy skorzystać z zawartych w nim definicji. Dyrektywa `#include` nie przydaje się nigdzie tak, jak w dużych programach, które trzeba dzielić na osobne moduły.

Jeśli nad danym programem pracuje więcej niż jeden programista, pliki włączane stanowią metodę standaryzacji kodu — każdy programista korzysta z tych samych definicji i tych samych wartości. Poza tym programiści są zwolnieni z czasochłonnego i podatnego na błędy wpisywania potrzebnych definicji w każdym pliku. Ostatnie dwa punkty stają się jeszcze ważniejsze, jeśli zaczynamy do plików włączanych wstawiać definicje struktur, deklaracje zmiennych zewnętrznych, definicje `typedef` i prototypy funkcji. Różne moduły dużych programów zawsze odwołują się do wspólnych struktur danych. Centralizując definicje tych struktur danych w formie jednego lub wielu plików włączanych, eliminujemy błędy związane z występowaniem w dwóch modułach różnych definicji tej samej struktury danych. Definicja jest tylko jedna — w pliku włączanym.

Przypomnijmy sobie strukturę danych z rozdziału 8. Oto przykład pliku włączanego podobnego do pliku, którego użylibyśmy, tworząc program obsługujący daty w różnych modułach. Jest to też dobry przykład połączenia różnych technik, o których dotąd się uczyliśmy.

```
// Plik nagłówkowy do obsługi dat

#include <stdbool.h>

// Typy wyliczeniowe

enum kMonth { styczeń=1, luty, marzec, kwiecień, maj, czerwiec,
              lipiec, sierpień, wrzesień, październik, listopad, grudzień
 };

enum kDay { niedziela, poniedziałek, wtorek, środa, czwartek, piątek, sobota };


struct date
{
    enum kDay   day;
    enum kMonth month;
    int         year;
};

// typ obsługujący datę
typedef struct date Date;

// Funkcja do obsługi dat
Date  dateUpdate (Date today);
int   numberOfDays (Date d);
bool  isLeapYear (Date d);

// Makro wstawiające datę do struktury
#define setDate(s,dd,mm,yy)  s = (Date) {dd, mm, yy}

// Odwołanie do zmiennej zewnętrznej
extern Date todaysDate;
```

Plik nagłówkowy zawiera definicje dwóch wyliczeniowych typów danych — `kMonth` i `kDay` — i strukturę danych (spójrzmy na sposób użycia wyliczeniowych typów danych). Instrukcja `typedef` służy do utworzenia typu danych `Date`, zadeklarowana jest funkcja korzystająca z tego typu. W pliku mamy jeszcze makro ustawiające wartości daty

(korzystające z literałów złożonych), w końcu mamy zmienną zewnętrzną `todaysDate`, przechowującą dzisiejszą datę (która jest zdefiniowana w jednym z plików źródłowych).

Jako przykład użycia powyższego pliku nagłówkowego obejrzyjmy zmodyfikowaną wersję funkcji `dateUpdate` z rozdziału 8.

```
#include "date.h"

// Funkcja wyliczająca jutrzejszą datę
Date dateUpdate (Date today)
{
    Date tomorrow;

    if ( today.day != numberOfDays (today) )
        setDate (tomorrow, today.day + 1, today.month, today.year);
    else if (today.month == grudzień )    // koniec roku
        setDate (tomorrow, 1, styczeń, today.year + 1);
    else                                  // koniec miesiąca
        setDate (tomorrow, 1, today.month + 1, today.year);

    return tomorrow;
} .
```

Inne narzędzia służące do pracy z dużymi programami

Jak wspominaliśmy wcześniej, doskonałym narzędziem do budowy większych programów jest środowisko IDE. Jeśli jednak ktoś woli pracować z wierszem poleceń, istnieją narzędzia, które mogą być przydatne. Narzędzia te nie są częścią języka C, ale mogą przyspieszyć programowanie — a o to przecież chodzi.

Oto lista narzędzi, którymi warto się zainteresować podczas tworzenia większych programów. W systemie Unix mamy do dyspozycji mnóstwo poleceń, które są pomocne przy programowaniu. To, co pokazujemy teraz, jest zaledwie drobną próbką. Gdy obsługujemy wiele plików, warto rozważyć możliwość nauczenia się programowania w języku skryptowym, takim jak interpreter poleceń systemu Unix (tzw. *powłoka*).

Narzędzie make

To użyteczne narzędzie (lub jego odpowiednik GNU — *gnumake*) pozwala opisywać listy plików i zależności między nimi w formie pliku *Makefile*. Program *make* automatycznie kompiluje tylko te pliki, dla których jest to niezbędne. Decyzje podejmowane są na podstawie czasu modyfikacji plików. Jeśli zatem program *make* stwierdzi, że plik źródłowy (*.c*) jest nowszy niż odpowiadający mu plik z kodem pośrednim (*.o*), skompiluje ponownie dany plik źródłowy. Można też podawać pliki źródłowe zależne od plików nagłówkowych. Można na przykład stwierdzić, że moduł *datefunc.o* zależy od pliku źródłowego *datefunc.c* oraz od pliku nagłówkowego *date.h*. Jeśli zmieni się tylko plik nagłówkowy, program ponownie automatycznie skompiluje *datafunc.c*.

Oto prosty plik *Makefile*, który służy do kompilacji trzech modułów używanych w tym rozdziale. Zakładamy, że plik ten jest w tym samym katalogu co pliki źródłowe.

```
$cat Makefile
SRC = mod1.c mod2.c main.c
OBJ = mod1.o mod2.o main.o
PROG = dbtest

$(PROG): $(OBJ)
        gcc $(OBJ) –o $(PROG)

$(OBJ): $(SRC)
```

Nie będziemy tutaj szczegółowo omawiać budowy pokazanego pliku *Makefile*. Najkrócej mówiąc, definiujemy pliki źródłowe (SRC), odpowiadające im pliki z kodem pośrednim (OBJ), nazwę pliku wykonywalnego (PROG) oraz pewne zależności. Pierwsza z tych zależności:

```
$(PROG): $(OBJ)
```

mówi, że plik wykonywalny zależy od plików z kodem pośrednim. Jeśli zatem zmienimy jeden czy więcej plików z kodem pośrednim, trzeba ponownie utworzyć plik wykonywalny. Wykonujemy to za pomocą następującego polecenia gcc (odstępy na początku to tabulatory):

```
        gcc $(OBJ) –o $(PROG)
```

Ostatni wiersz pliku *Makefile*:

```
$(OBJ): $(SRC)
```

informuje, że każdy plik z kodem pośrednim zależy od odpowiadającego mu pliku źródłowego. Jeśli zatem zmieni się plik źródłowy, konieczne jest odtworzenie odpowiedniego pliku z kodem pośrednim. Narzędzie *make* ma wbudowane reguły mówiące, jak to zrobić.

Oto, co się stanie po pierwszym uruchomieniu *make*:

```
$ make
gcc    –c –o mod1.o mod1.c
gcc    –c –o mod2.o mod2.c
gcc    –c –o main.o main.c
gcc mod1.o mod2.o main.o o dbtest
$
```

Świetna sprawa! Program *make* skompilował wszystkie pojedyncze pliki źródłowe, następnie pliki pośrednie skonsolidował w plik wykonywalny.

Jeśli w pliku *mod2.c* wystąpiłby błąd, z programu *make* uzyskalibyśmy następujące wyniki:

```
gcc    –c –o mod1.o mod1.c
gcc    –c –o mod2.o mod2.c
mod2.c: In function 'foo2':
mod2.c:3: error: 'i' undeclared (first use in this function)
mod2.c:3: error:(Each undeclared identifier is reported only once
mod2.c:3: error: for each function it appears in.)
make: *** [mod2.o] Error 1
$
```

Program *make* stwierdził, że podczas kompilowania modułu *mod2.c* pojawił się błąd, więc przerwał swoje działanie; jest to zachowanie domyślne.

Jeśli poprawimy plik *mod2.c* i ponownie uruchomimy program *make*, uzyskamy następujące wyniki:

```
$ make
gcc    –c –o mod2.o mod2.c
gcc    –c –o main.o main.c
gcc mod1.o mod2.o main.o o dbtest
$
```

Zauważmy, że plik *mod1.c* nie był już ponownie kompilowany. Po prostu nie było to konieczne. Właśnie w takim zachowaniu leży cała siła i elegancja programu *make*.

Nawet dla tak prostego pliku *Makefile* można zacząć używać programu *make*. W dodatku E, poświęconym zasobom, powiemy, gdzie można znaleźć więcej informacji o omawianym narzędziu.

Narzędzie cvs

Jest to jedno z kilku narzędzi do obsługi kodu źródłowego. Zapewnia ono automatyczne śledzenie wersji kodu źródłowego, śledzi historię zmian w modułach. Dzięki temu można odtworzyć określoną wersję programu (czy to w celu odtworzenia potrzebnej wersji systemu, czy to w ramach serwisu klienta). Program CVS pozwala rejestrować pobranie pliku źródłowego (program *cvs* z parametrem `checkout`), modyfikować plik i następnie ponownie przyjmować plik (*cvs* z parametrem `commit`). Mechanizm ten pozwala uniknąć konfliktów w przypadku, kiedy więcej niż jeden programista chce korzystać z tego samego pliku źródłowego. Użycie programu *cvs* pozwala współpracować programistom różnie zlokalizowanym; możliwa jest współpraca za pośrednictwem sieci.

Narzędzia systemu Unix

W systemie Unix istnieje szereg poleceń, które ułatwiają i przyspieszają tworzenie dużych programów, na przykład program *ar* pozwala tworzyć własne biblioteki. Zatem można zbierać w całość często używane funkcje lub funkcje, którymi chcemy podzielić się z innymi. Tak jak w przypadku kompilowania programu wykorzystującego funkcje z biblioteki matematycznej korzystamy z opcji `–lm`, tak samo możemy za pomocą opcji *–llib* dołączyć własną bibliotekę. W trakcie fazy konsolidacji w bibliotece odszukiwane są funkcje, następnie są stamtąd pobierane i dołączane do programu.

Inne polecenia, takie jak *grep* czy *sed*, przydają się do szukania łańcuchów w pliku lub do dokonywania zmian globalnych na zbiorze plików. Jeśli na przykład umiemy choć trochę programować interpreter poleceń, możemy skorzystać z programu *sed* do zmiany jednej zmiennej na inną we wszystkich wystąpieniach. Może to dziać się jednocześnie w wielu plikach. Polecenie *grep* po prostu odszukuje łańcuch w podanym pliku lub plikach. Jest to przydatne do znalezienia zmiennej czy funkcji w dużej grupie plików źródłowych, czy makr w plikach nagłówkowych. Polecenie:

```
$ grep todaysDate main.c
```

spowoduje przeszukanie pliku *main.c* i podanie wszystkich wierszy zawierających napis `todaysDate`. Polecenie:

```
$ grep –n todaysDate *.c *.h
```

przeszukuje wszystkie pliki źródłowe i nagłówkowe z bieżącego katalogu, następnie pokazuje każde wystąpienie szukanego łańcucha poprzedzone numerem wiersza w pliku (wynika to z użycia opcji `–n`). Wiemy już, jak język C ułatwia dzielenie dużych programów na mniejsze moduły, jak wykonywana jest przyrostowa i niezależna kompilacja poszczególnych modułów. Pliki nagłówkowe stanowią swoisty „klej" łączący moduły; umieszcza się w nich wspólne deklaracje prototypów, makra, definicje struktur, wyliczeń i tak dalej.

Osoby korzystające ze środowiska IDE mają ułatwione zadanie w zakresie zarządzania programami wielomodułowymi. Aplikacja IDE śledzi pliki wymagające rekompilacji. Podczas korzystania z kompilatora działającego w wierszu poleceń, takiego jak *gcc*, trzeba pamiętać samemu, które pliki wymagają skompilowania, albo można skorzystać z narzędzi typu *make*. W przypadku kompilowania programu z wiersza poleceń potrzebne są też zwykle inne narzędzia — do przeszukiwania plików źródłowych, dokonywania w nich globalnych zmian oraz tworzenia i obsługi bibliotek programów.

15

Operacje wejścia i wyjścia w języku C

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

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

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

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

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

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

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

Wejście i wyjście znakowe: funkcje getchar i putchar

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

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

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

```
putchar (c);
```

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

Wywołanie:

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

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

Formatowanie wejścia i wyjścia: funkcje printf i scanf

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

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

Funkcja printf

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

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

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

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

Tabela 15.1. Flagi funkcji printf

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

Tabela 15.2. Modyfikatory funkcji printf określające szerokość i precyzję

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

Tabela 15.3. Modyfikatory typu stosowane w funkcji printf

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

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

W tabeli 15.4 zestawiono znaki konwersji umieszczane w łańcuchu formatującym.

Tabela 15.4. Znaki konwersji funkcji printf

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

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

Program 15.1 pokazuje część możliwości formatowania za pomocą funkcji `printf`.

Program 15.1. **Użycie formatów funkcji printf**

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

#include <stdio.h>

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

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

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

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

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

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

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

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

    return 0;
}
```

Program 15.1. **Wyniki**

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

```
61817  170571  f179  61817
75000  222370  124f8  75000
1311768465173141112 11064254740221505317O 1234567812345678 1311768465173141112

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

Znaki:
X
  X  X
58

Łańcuchy znakowe:
abcdefghijklmnopqrstuvwxyz
abcde
    abcdefghijklmnopqrstuvwxyz
              abcde
abcde

Wskaźniki:
0xbffffc20  0xbffffbf0

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

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

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

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

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

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

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

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

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

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

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

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

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

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

co pozwala dynamicznie zmieniać format wyświetlania wartości.

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

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

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

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

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

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

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

Postać wyniku w przypadku użycia formatu `%p` jest zależna od konkretnej implementacji; w naszym przykładzie pokazujemy adresy zapisane szesnastkowo. Zgodnie z pokazanym wynikiem zmienna wskaźnikowa `ip` zawiera adres `bffffc20`, a wskaźnik `cp` — `bffffbf0`.

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

Funkcja scanf

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

Tabela 15.5. Modyfikatory konwersji w funkcji scanf

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

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

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

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

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

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

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

Tabela 15.6. Znaki konwersji funkcji scanf

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

Tabela 15.6. Znaki konwersji funkcji scanf (ciąg dalszy)

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

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

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

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

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

kiedy podano następujące dane:

```
29   w
```

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

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

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

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

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

i podamy jej dane:

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

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

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

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

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

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

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

Wywołanie funkcji `scanf`:

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

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

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

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

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

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

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

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

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

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

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

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

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

Operacje wejścia i wyjścia na plikach

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

Przekierowanie wejścia-wyjścia do pliku

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

Program 15.2. **Prosty przykład**

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

#include <stdio.h>

main()
{

    float d = 6.5;
    float half, square, cube;

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

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

    return 0;
}
```

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

```
program1502 > results.txt
```

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

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

Program 15.3. **Prosty, ale bardziej interaktywny przykład**

```
// Program odbierający od użytkownika jedną liczbę i zwracający wyniki kliku działań arytmetycznych.
#include <stdio.h>

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

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

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

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

```
program1503 > results2.txt
```

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

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

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

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

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

```
program1503 < simp4.txtn
```

W terminalu pojawią się następujące informacje:

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

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

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

Oczywiście można jednocześnie przekierować wejście i wyjście programu:

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

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

Przekierowywanie wejścia i wyjścia do pliku bardzo często jest wystarczającym rozwiązaniem. Załóżmy na przykład, że piszemy artykuł do gazety i wpisywaliśmy tekst do pliku *article*. Program 9.8 zliczał słowa w tekście. Tego samego programu możemy teraz użyć do zliczenia słów w naszym artykule; wystarczy wydać polecenie[1]:

```
wordcount < article
```

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

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

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

Koniec pliku

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

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

Program 15.4. **Kopiowanie znaków ze standardowego wejścia na standardowe wyjście**

```
// Program pokazujący podawane znaki aż do napotkania końca pliku

#include <stdio.h>

int main (void)
{
    int  c;

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

    return 0;
}
```

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

```
copyprog < infile
```

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

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

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

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

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

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

Funkcje specjalne do obsługi plików

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

Funkcja fopen

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

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

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

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

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

```
#include <stdio.h>

FILE  *inputFile;

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

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

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

Teraz wiemy już, czy udało się plik otworzyć.

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

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

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

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

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

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

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

otwiera plik binarny do odczytu.

Funkcje getc i putc

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

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

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

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

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

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

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

Funkcja fclose

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

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

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

```
fclose (inputFile);
```

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

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

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

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

Program 15.5. **Kopiowanie plików**

```
// Program kopiujący pliki

#include <stdio.h>

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

    // pobranie od użytkownika nazw plików

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

    // otwieramy plik wejściowy i wynikowy

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

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

    // kopiowanie pliku in na plik out

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

    // zamykanie otwartych plików

    fclose (in);
    fclose (out);

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


    return 0;
}
```

Program 15.5. **Wyniki**

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

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

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

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

Funkcja feof

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

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

spowodują wyświetlenie na ekranie komunikatu `"Koniec danych"`.

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

Funkcje fprintf i fscanf

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

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

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

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

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

Funkcje fgets i fputs

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

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

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

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

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

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

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

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

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

Wskaźniki stdin, stdout i stderr

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

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

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

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

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

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

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

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

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

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

Funkcja exit

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

```
exit (n);
```

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

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

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

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

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

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

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

    return inFile;
}
```

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

Zmiana nazw i usuwanie plików

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

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

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

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

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

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

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

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

Ćwiczenia

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

16

Rozmaitości, techniki zaawansowane

W tym rozdziale przedstawimy pewne cechy języka, o których dotąd nie mówiliśmy, na przykład parametry wiersza poleceń czy dynamiczną alokację pamięci. Tematyka tego rozdziału jest zróżnicowana, ale wszystkie opisane zagadnienia warto znać, ponieważ mają one praktyczne zastosowanie w programach. Oto lista tematów opisanych w tym rozdziale:

- zasada działania instrukcji `goto` i powody, dla których należy tej instrukcji unikać;
- optymalne wykorzystanie przestrzeni dzięki wykorzystaniu unii;
- dodawanie instrukcji pustej do programów;
- implementowanie instrukcji zawierających przecinek jako operator;
- wykorzystanie argumentów z wiersza poleceń w programie;
- dynamiczna alokacja pamięci przy użyciu funkcji `malloc()` i `calloc()` oraz kasowanie zawartości pamięci za pomocą funkcji `free()`.

Pozostałe instrukcje języka

W tym podrozdziale przedstawimy dwie instrukcje, których jeszcze nie poznaliśmy — będą to `goto` i instrukcja pusta.

Instrukcja goto

Kto kiedykolwiek programował strukturalnie, wie, jak złą reputację ma instrukcja `goto`. Prawie każdy język programowania ma taką instrukcję.

Wykonanie instrukcji `goto` powoduje natychmiastowe przekazanie sterowania do wskazanego punktu sterowania. Skok następuje natychmiast i bezwarunkowo, w chwili natknięcia się na `goto`. Aby wskazać, gdzie ma nastąpić skok, trzeba zastosować *etykietę*.

Etykieta to nazwa podlegająca takim samym regułom jak nazwy zmiennych, za nazwą tą znajduje się dwukropek. Etykieta jest umieszczana przed instrukcją, do której ma nastąpić skok; musi ona znaleźć się w tej samej funkcji, w której jest instrukcja `goto`.

Zatem na przykład instrukcja:

```
goto brak_danych;
```

powoduje, że program wykonuje skok do instrukcji poprzedzonej etykietą `brak_danych:`. Etykieta ta może być umieszczona w dowolnym miejscu funkcji, przed instrukcją `goto` lub za nią:

```
brak_danych:  printf ("Niespodziewany koniec danych.\n");
    ...
```

Leniwi programiści zwykle nadużywają instrukcji `goto` do realizowania skoków w inne części kodu. Instrukcja `goto` przerywa normalny przebieg sterowania w programie, wobec czego trudniej takie programy analizować. Zastosowanie w programie bardzo wielu instrukcji `goto` może całkiem uniemożliwić taką analizę. Z tego powodu instrukcje `goto` uważa się za niezgodne z zasadami dobrego programowania.

Instrukcja pusta

Język C pozwala umieścić pojedynczy średnik tam, gdzie normalnie występuje instrukcja. Jest to instrukcja pusta, która nic nie robi. Choć instrukcja taka w pierwszej chwili może wydać się zbędna, programiści często wykorzystują ją w pętlach `while`, `for` i `do`. Poniższa instrukcja na przykład służy do zapisania wszystkich znaków wczytywanych ze standardowego wejścia w tablicy znakowej wskazywanej przez zmienną `text` aż do napotkania znaku nowego wiersza:

```
while ( (*text++ = getchar ()) != '\n' )
    ;
```

Wszystkie potrzebne działania są zaszyte w warunkach pętli `while`. Pusta instrukcja jest niezbędna, bo gdyby jej zabrakło, kompilator potraktowałby następną instrukcję jako treść pętli.

Poniższa instrukcja `for` kopiuje znaki ze standardowego wejścia na standardowe wyjście aż do napotkania końca pliku:

```
for ( ; (c = getchar ()) != EOF;  putchar (c) )
    ;
```

Następna instrukcja `for` zlicza znaki ze standardowego wejścia:

```
for ( count = 0;  getchar () != EOF;  ++count )
    ;
```

Ostatnim przykładem użycia instrukcji pustej będzie pętla kopiująca łańcuch znakowy wskazywany przez `from` w miejsce wskazywane przez `to`:

```
while ( (*to++ = *from++) != '\0' )
    ;
```

Czytelnikom należy się uczciwe ostrzeżenie, że wielu programistów ma tendencję do umieszczania możliwie dużo logiki programu w warunkach pętli `while` czy `for`. Nie warto wstępować do tego klubu programistycznego — w zasadzie w warunkach pętli powinny być tylko warunki. Wszystko, poza warunkami, powinno znaleźć się w treści pętli. Jedyne uzasadnienie dla takiego zagęszczania kodu to szybkość działania programu. Jeśli jednak nie jest ona krytyczna, należy unikać tego typu wyrażeń.

Poprzednią instrukcję `while` można zapisać w bardziej czytelnej formie:

```
while ( *from != '\0' )
    *to++ = *from++;

*to = '\0';
```

Użycie unii

Jedne z najbardziej niezwykłych konstrukcji języka C to *unie*. Są one używane głównie w bardziej złożonych aplikacjach, gdzie konieczne jest przechowywanie różnych typów danych w tym samym miejscu. Jeśli na przykład chcemy zdefiniować zmienną `x`, która będzie mogła zawierać pojedynczy znak, liczbę zmiennoprzecinkową lub liczbę całkowitą, najpierw definiujemy unię — niech się nazywa `mixed`:

```
union mixed
{
    char   c;
    float  f;
    int    i;
};
```

Deklaracja unii jest identyczna jak deklaracja struktury, tyle że inne jest słowo kluczowe — `union` zamiast `struct`. Prawdziwa różnica tkwi jednak w sposobie alokowania pamięci. Deklarując zmienną typu `union mixed`, jak poniżej:

```
union mixed  x;
```

nie mówimy, że zmienna `x` będzie zawierała trzy osobne pola — `c`, `f` oraz `i`; `x` będzie zawierała *jedną* wartość — albo `c`, albo `f`, albo `i`. Dzięki temu zmiennej `x` można użyć do przechowywania znaku, liczby zmiennoprzecinkowej lub liczby całkowitej, ale nie wszystkich trzech jednocześnie (ani nawet dwóch spośród trzech). Aby w zmiennej `x` zapisać pojedynczy znak, piszemy:

```
x.c = 'K';
```

Analogicznie odczytujemy znak z unii. Aby zatem pokazać wartość zmiennej `x`, możemy użyć kodu:

```
printf ("Znak = %c\n", x.c);
```

Aby w zmiennej `x` zapisać liczbę zmiennoprzecinkową, używamy zapisu `x.f`:

```
x.f = 786.3869;
```

W końcu, aby w x zapisać wynik dzielenia całkowitej zmiennej `count` przez 2, piszemy:

```
x.i = count / 2;
```

Pola typów `float`, `char` i `int` zmiennej x wszystkie zajmują to samo miejsce w pamięci, tylko jedno z nich w danej chwili może być użyte. Co więcej, na użytkowniku spoczywa odpowiedzialność za pobranie takiego typu danej z unii, jaki został ostatnio zapisany.

Pole unii podlega takim samym regułom arytmetyki jak każda inna zmienna danego typu. Zatem w wyrażeniu:

```
x.i / 2
```

zastosowana zostanie arytmetyka całkowitoliczbowa, gdyż `x.i` oraz 2 są liczbami całkowitymi.

Unia może zawierać dowolnie wiele pól. Kompilator C zadba o to, aby unia otrzymała dość pamięci na największe z pól. Struktury, podobnie jak tablice, mogą zawierać unie. Kiedy definiujemy unię, nie trzeba podawać jej nazwy, a zmienne można deklarować w definicji samej unii. Można też deklarować wskaźniki na unie, które niczym nie różnią się od wskaźników struktur.

W deklaracji zmiennej można zainicjalizować jedno pole unii. Jeśli nie zostanie podana nazwa tego pola, wartość zostanie przypisana *pierwszemu* polu, jak poniżej:

```
union mixed x = { '#' };
```

W ten sposób ustawiamy pierwsze pole x, czyli `c`, nadając mu wartość `#`.

Gdy podajemy nazwę pola, możemy inicjalizować dowolne pole unii:

```
union mixed x = { .f = 123.456; };
```

W ten sposób nadaliśmy wartość `123.456` zmiennej x będącej unią typu `mixed`.

Automatyczne zmienne będące uniami możemy deklarować, korzystając z innej zmiennej tego samego typu:

```
void foo (union mixed x)
{
    union mixed  y = x;
    ...
}
```

W powyższej funkcji `foo` przypisujemy parametr x zmiennej automatycznej y będącej unią.

Użycie unii pozwala definiować tablice mające elementy różnych typów, na przykład instrukcja:

```
struct
{
    char             *name;
    enum symbolType  type;
    union
    {
        int    i;
        float  f;
        char   c;
    } data;
} table [kTableEntries];
```

powoduje utworzenie tablicy `table` mającej `kTableEntries` elementów. Każdy element zawiera strukturę składającą się ze wskaźnika znakowego `name`, pola wyliczeniowego `type` oraz unii `data`. W tablicy każde pole `data` może zawierać liczbę typu `int`, `float` lub znak `char`. Pola `type` można użyć do zapamiętywania, jakiego typu wartość jest w tej chwili przechowywana w polu `data`. Na przykład: przypisując temu polu wartość `INTEGER`, wiedzielibyśmy, że w polu `data` jest liczba `int`. Analogicznie typowi `float` mogłaby odpowiadać wartość `FLOATING`, a typowi `char` — wartość `CHARACTER`. Informacja ta pozwoliłaby nam decydować, jak odwoływać się do pola `data` w danym elemencie tablicy.

Aby w `table[5]` zapisać znak `'#'`, a także ustawić pole `type` tak, aby zapamiętać, że jest to właśnie znak, potrzebne są dwie instrukcje:

```
table[5].data.c = '#';
table[5].type = CHARACTER;
```

Odwołując się do elementów `table`, możemy określić typ danej w każdym z nich — wystarczy kilka porównań, na przykład poniższa pętla wyświetli wszystkie nazwy i odpowiadające im wartości:

```
enum symbolType { INTEGER, FLOATING, CHARACTER };

    ...

for ( j = 0;  j < kTableEntries;  ++j ) {
    printf ("%s  ", table[j].name);

    switch ( table[j].type ) {
        case INTEGER:
             printf ("%i\n", table[j].data.i);
             break;
        case FLOATING:
             printf ("%f\n", table[j].data.f);
             break;
        case CHARACTER:
             printf ("%c\n", table[j].data.c);
             break;
        default:
             print ("Nieznany typ (%i), element %i\n", table[j].type, j );
             break;
    }
}
```

Opisany sposób przechowywania danych może przydać się na przykład do zapisu tablic symboli, gdzie każdemu symbolowi przypisujemy nazwę, typ oraz wartość (i ewentualnie jeszcze inne informacje o symbolu).

Przecinek jako operator

Na pierwszy rzut oka można sobie nie zdawać sprawy z tego, że w wyrażeniach można używać przecinka jako operatora. Operator przecinek ma wyjątkowo niski priorytet. W rozdziale 4., w którym mówiliśmy o pętlach, dowiedzieliśmy się, że w instrukcji `for`

możemy włączyć więcej niż jedno wyrażenie pod warunkiem rozdzielenia przecinkami poszczególnych wyrażeń. I tak instrukcja `for` zaczynająca się od:

```
for ( i = 0, j = 100;  i != 10;  ++i, j -= 10 )
    ...
```

przed rozpoczęciem wykonywania pętli inicjalizuje zmienną `i` na 0, a zmienną `j` na 100. W każdym przejściu pętli wartość `i` jest zwiększana o 1, a wartość zmiennej `j` jest zmniejszana o 10.

Operator przecinek służy do rozdzielania wielu wyrażeń wszędzie tam, gdzie normalnie może być jedno wyrażenie. Wyrażenia wyliczane są od strony lewej do prawej. Wobec tego instrukcja:

```
while ( i < 100 )
    sum += data[i], ++i;
```

powoduje dodanie wartości `data[i]` do zmiennej `sum`, a następnie zwiększenie wartości zmiennej `i`. Zauważmy, że zbędne są tutaj nawiasy klamrowe, gdyż za wyrażeniem `while` znajduje się tylko jedna instrukcja (składająca się z dwóch wyrażeń rozdzielonych przecinkiem).

Ponieważ wszystkie operatory języka C zwracają wartość, to samo dotyczy operatora przecinek — zwraca on wartość ostatniego wyrażenia z prawej strony.

Zauważmy, że czym innym jest operator przecinek, a czym innym przecinki rozdzielające parametry wywołania funkcji czy nazwy funkcji w liście deklaracji.

Kwalifikatory typu

Poniższe kwalifikatory mogą być używane przed zmiennymi, aby przekazać kompilatorowi dodatkowe informacje o przeznaczeniu tych zmiennych oraz czasami w celu ułatwienia generowania sprawniej działającego kodu.

Kwalifikator register

Jeśli funkcja wyjątkowo intensywnie wykorzystuje jakąś zmienną, można zażądać od kompilatora, aby w miarę możności usprawnił dostęp do tej zmiennej. Zwykle oznacza to żądanie, aby w chwili wykonywania funkcji zmienna ta była umieszczana w rejestrze. Takie żądanie przekazujemy, poprzedzając deklarację zmiennej słowem kluczowym `register`:

```
register int   index;
register char  *textPtr;
```

Słowa kluczowego `register` można używać w odniesieniu zarówno do zmiennych lokalnych, jak i do parametrów formalnych. Na różnych maszynach odmienne typy zmiennych można umieszczać w rejestrach. Zwykle jest to możliwe w przypadku podstawowych typów danych i wskaźników na dowolne typy.

Jeśli nawet używany przez nas system umożliwia zadeklarowanie zmiennej z kwalifikatorem `register`, nie ma żadnej gwarancji, że taka deklaracja będzie honorowana. Zależy to od kompilatora.

Warto wiedzieć jeszcze, że nie można odwołać się do adresu zmiennej zadeklarowanej jako rejestrowa; poza tym zmienne te zachowują się analogicznie jak wszystkie inne zmienne automatyczne.

Kwalifikator volatile

Jest to swojego rodzaju odwrotność deklaracji `const`. Kwalifikator `volatile` informuje kompilator, że dana zmienna *będzie* zmieniała swoją wartość. Deklarację taką podaje się, aby kompilator nie wykonał optymalizacji związanej z powtórnym przypisaniem zmiennej wartości lub z powtórnym sprawdzaniem jej wartości bez widocznej zmiany tej wartości. Dobrym przykładem jest port systemowy. Załóżmy, że mamy port wyjściowy wskazywany przez jedną ze zmiennych naszego programu. Jeśli do portu chcemy zapisać dwa znaki, na przykład O i potem N, możemy użyć następującego kodu:

```
*outPort = 'O';
*outPort = 'N';
```

Dobrze napisany kompilator może zauważyć dwa kolejne przypisania do tej samej zmiennej i jako że zmienna `outPort` nie była w międzyczasie modyfikowana, po prostu usunąć pierwsze przypisanie. Aby tego uniknąć, deklarujemy zmienną `outPort` jako wskaźnik z kwalifikatorem `volatile`:

```
volatile char  *outPort;
```

Kwalifikator restrict

Podobnie jak kwalifikator `register`, tak i `restrict` jest podpowiedzią optymalizacji dla kompilatora. Kompilator może ten kwalifikator całkiem zignorować. Służy on do informowania, że dany wskaźnik jest jedynym odwołaniem (bezpośrednim lub pośrednim) wskazującym daną wartość. Oznacza to, że interesująca nas wartość nie jest wskazywana przez żaden inny wskaźnik ani zmienną.

Poniższe dwa wiersze:

```
int * restrict intPtrA;
int * restrict intPtrB;
```

przekazują do kompilatora informację, że w całym zakresie obowiązywania identyfikatorów `intPtrA` i `intPtrB` nigdy nie sięgną do tej samej wartości. Ich użycie do wskazywania na przykład liczb w tablicy jest naprzemienne.

Parametry wiersza poleceń

Program wielokrotnie przez nas tworzony wymaga podania przez użytkownika niewielkiej ilości danych. Danymi tymi może być liczba wskazująca potrzebną liczbę trójkątną lub słowo, które chcielibyśmy znaleźć w słowniku.

Zamiast wymuszać na użytkowniku podawanie tych informacji do programu, możemy podać je w chwili wywoływania tego programu. Służą do tego *parametry wiersza poleceń*.

Jak powiedzieliśmy już wcześniej, jedynym wyróżnikiem funkcji `main` wśród innych funkcji jest to, że jej nazwa jest specyficzna — od tak nazwanej funkcji zaczyna się wykonywanie programu. Tak naprawdę funkcja ta jest normalnie *wywoływana* przez program uruchomieniowy języka C, tak jak wywoływana jest każda funkcja. Kiedy funkcja `main` kończy swoje działanie, sterowanie wraca do programu uruchomieniowego, który wie, że nasz program zakończył swoje działanie.

Kiedy funkcja `main` jest wywoływana, przekazywane są do niej dwa parametry. Pierwszy z nich, zwyczajowo nazywany `argc` (od *arg*ument *c*ount, czyli *licznik parametrów*), to liczba całkowita mówiąca, ile parametrów przekazano w wierszu poleceń. Drugi parametr to tablica znakowa, zwyczajowo nazywana `argv` (od *arg*ument *v*ector, czyli *tablica parametrów*). W tablicy tej jest `argc+1` wskaźników znakowych; wartość `argc` jest nie mniejsza od 0. Pierwsza pozycja tablicy `argv` to nazwa uruchomionego programu lub wskaźnik do pustego ciągu, w przypadku kiedy nazwa programu w używanym systemie jest niedostępna. Kolejne elementy tej tablicy wskazują wartości przekazane w wierszu poleceń podczas uruchamiania programu. Ostatni wskaźnik tablicy `argv`, czyli `argv[argc]`, z definicji jest pusty.

Aby sięgnąć do parametrów wiersza poleceń, funkcja `main` musi zostać prawidłowo zadeklarowana jako funkcja dwuparametrowa. Zwykle deklaracja wygląda następująco:

```
int main (int argc, char *argv[])
{
    ...
}
```

Pamiętajmy, że deklaracja `argv` to deklaracja tablicy, której elementy są wskaźnikami na łańcuchy znakowe. Aby praktycznie wykorzystać parametry przekazywane z wiersza poleceń, przypomnijmy sobie program 9.10, wyszukujący słowo w słowniku i pokazujący jego znaczenie. Możemy tak przygotować parametry wiersza poleceń, aby interesujące nas słowo było podawane wraz z wywołaniem programu:

```
lookup abisal
```

W ten sposób program nie musi już pytać użytkownika o słowo — jest ono podane w wierszu poleceń.

Kiedy wykonane zostanie powyższe polecenie, system automatycznie przekaże funkcji `main` wskaźnik do łańcucha znakowego `"abisal"` w elemencie `argv[1]`. Przypomnijmy, że `argv[0]` zawiera wskaźnik do nazwy programu — w tym wypadku `"lookup"`.

Funkcja `main` ma następującą postać:

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

int main (int argc, char *argv[])
{
    const struct entry  dictionary[100] =
      {  { "abakan",    "przestrzenny, monumentalny gobelin" },
         { "abakus",    "starożytne liczydło" },
         { "abazja",    "utrata zdolności chodzenia" },
         { "abduktor",  "mięsień odwodzący kończynę" },
         { "abietyna",  "substancja żywiczna z drzew iglastych" },
         { "abisal",    "najgłębsza strefa mórz, oceanów i jezior" },
         { "abrazja",   "niszczenie wybrzeży przez fale" },
         { "absces",    "inaczej: ropień" },
         { "absynt",    "nalewka spirytusowa na ziołach" },
         { "achterdek", "rufowa część pokładu" } };

    int  entries = 10;
    int  entryNumber;
    int  lookup (const struct entry dictionary[], const char search[],
                 const int entries);

    if ( argc != 2 )
    {
        fprintf (stderr, "Nie podano w wierszu poleceń szukanego słowa.\n");
        return EXIT_FAILURE;
    }

    entryNumber = lookup (dictionary, argv[1], entries);

    if ( entryNumber != -1 )
        printf ("%s\n", dictionary[entryNumber].definition);
    else
        printf ("Niestety, słowo %s nie występuje w moim słowniku.\n", argv[1]);

    return EXIT_SUCCESS;
}
```

Funkcja `main` sprawdza, czy w wierszu poleceń przekazano szukane słowo. Jeśli nie podano słowa lub podano więcej niż jedno słowo, wartość `argc` jest różna od 2. Wtedy program pisze komunikat błędu do standardowego pliku błędów i kończy swoje działanie, przekazując do systemu kod wyjścia `EXIT_FAILURE`.

Jeśli `argc` ma wartość 2, wywoływana jest funkcja `lookup` odszukująca w słowniku słowo wskazywane przez `argv[1]`. Jeśli poszukiwanie się powiedzie, pokazywana jest definicja znalezionego słowa.

Teraz zajmijmy się innym przykładem użycia parametrów wiersza poleceń. Program 15.3 służył do kopiowania pliku. Pokazany poniżej program 16.1 wykonuje to samo, ale nazwy obu plików przekazujemy mu w wierszu poleceń, zamiast dopiero później prosić użytkownika o te nazwy.

Program 16.1. **Program kopiujący pliki, wykorzystujący parametry wiersza poleceń**

```
// Program kopiujący jeden plik na drugi, 2. wersja

#include <stdio.h>

int main (int argc, char *argv[])
{
    FILE  *in, *out;
    int   c;

    if ( argc != 3 ) {
        fprintf (stderr, "Wymagane są nazwy dwóch plików\n");
        return 1;
    }

    if ( (in = fopen (argv[1], "r")) == NULL ) {
        fprintf (stderr, "Nie mogę odczytać %s.\n", argv[1]);
        return 2;
    }

    if ( (out = fopen (argv[2], "w")) == NULL ) {
        fprintf (stderr, "Nie mogę pisać do %s.\n", argv[2]);
        return 3;
    }

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

    printf ("Przekopiowano wskazany plik.\n");

    fclose (in);
    fclose (out);

    return 0;
}
```

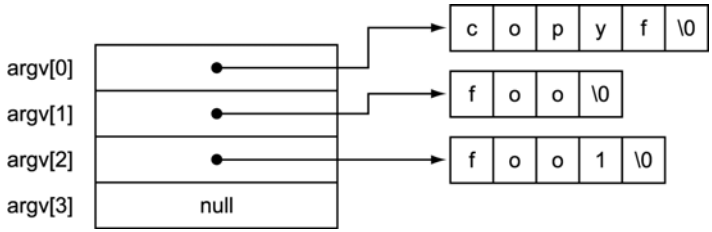
Program najpierw sprawdza, czy po nazwie programu podano nazwy obu plików. Jeśli tak, nazwa pliku wejściowego jest w łańcuchu wskazywanym przez `argv[1]`, a nazwa pliku wynikowego w łańcuchu wskazywanym przez `argv[2]`. Po otwarciu pierwszego pliku do odczytu i drugiego do zapisu, a także po sprawdzeniu, czy oba otwarcia się powiodły, program kopiuje znaki z pierwszego pliku tak samo jak poprzednio.

Zauważmy, że w tym wypadku program może zakończyć swoje działanie na cztery sposoby. Są to: nieprawidłowa liczba parametrów w wierszu poleceń, niemożność otwarcia pliku do odczytu, niemożność otwarcia pliku do zapisu, wszystko się udało. Pamiętajmy, że jeśli zamierzamy używać kodu wyjścia, to *zawsze* musimy jakiś kod podać, kończąc działanie programu. Jeśli program zakończy się wskutek przejścia poza funkcję `main`, zwróci *nieokreślony* kod wyjścia.

Jeśli program 16.1 nazwiemy *copyf* i wywołamy go następująco:

```
copyf foo foo1
```

to w chwili przekazania sterowania do funkcji `main` tablica `argv` będzie wyglądała tak jak na rysunku 16.1.



Rysunek 16.1. Tablica argv w chwili uruchamiania programu copyf

Pamiętajmy, że parametry wiersza poleceń *zawsze* są łańcuchami znakowymi. Wykonanie programu *power* (potęgowanie) z parametrami 2 i 16, tak jak poniżej:

```
power 2 16
```

spowoduje przekazanie w `argv[1]` wskaźnika do łańcucha `"2"` i w `argv[2]` wskaźnika do łańcucha `"16"`. Jeśli program ma zinterpretować te parametry jako liczby, musi sam je skonwertować. Można użyć różnych funkcji z biblioteki standardowej, na przykład `sscanf`, `atof`, `atoi`, `strtod` czy `strtol`. Wszystkie one zostały opisane w dodatku B.

Dynamiczna alokacja pamięci

Kiedy w języku C definiujemy zmienną typu prostego, tablicę czy strukturę, zawsze rezerwujemy na nią miejsce w pamięci. Kompilator C automatycznie zaalokuje tyle miejsca, ile potrzeba.

Jednak przydatne, a często niezbędne, jest *dynamiczne* alokowanie pamięci podczas działania programu. Załóżmy, że mamy program odczytujący zbiór danych z pliku do tablicy. Przyjmijmy, że nie wiemy, ile tych danych będzie. Możemy postąpić na trzy sposoby:

- Już przed skompilowaniem programu definiujemy tablicę tak, aby pomieściła największą możliwą liczbę elementów.
- Używamy tablicy o zmiennej wielkości, aby podczas działania programu dostosowywać jej wielkość.
- Alokujemy tablicę dynamicznie, korzystając z funkcji alokacji pamięci dostępnych w języku C.

Pierwsze rozwiązanie wymaga zdefiniowania tablicy na największą możliwą liczbę elementów:

```
#define kMaxElements    1000

struct dataEntry  dataArray [kMaxElements];
```

Póki plik zawiera nie więcej niż tysiąc elementów, wszystko jest w porządku. Jeśli jednak liczba tych elementów przekroczy tysiąc, musimy zmodyfikować program, zmieniając wartość `kMaxElements`, i ponownie go skompilować. Niezależnie od tego,

ile elementów ustalimy jako maksimum, zawsze istnieje pewne prawdopodobieństwo, że w przyszłości okaże się to ilością niewystarczającą.

W drugim rozwiązaniu przed wczytaniem danych określamy liczbę potrzebnych elementów (na przykład na podstawie wielkości pliku), po czym możemy zdefiniować tablicę o zmiennej wielkości:

```
struct dateEntry dataArray [dataItems];
```

W tym wypadku zakładamy, że zmienna `dataItems` zawiera wspomnianą wcześniej liczbę wczytywanych elementów.

Korzystając z funkcji dynamicznie alokujących pamięć, możemy pomieścić tyle danych, ile potrzebujemy. Rozwiązanie takie umożliwia alokowanie pamięci podczas działania programu. Musimy jednak poznać jeden operator i trzy nowe funkcje.

Funkcje calloc i malloc

W standardowej bibliotece języka C funkcje `calloc` i `malloc` służą do dynamicznego alokowania pamięci. Funkcja `calloc` ma dwa parametry określające liczbę alokowanych elementów i wielkość każdego z nich w *bajtach*. Funkcja ta zwraca wskaźnik do początku zaalokowanego bloku pamięci. Zaalokowany obszar jest automatycznie zerowany.

Funkcja `calloc` zwraca wskaźnik typu `void`, czyli ogólny typ wskaźnikowy języka C. Zanim będziemy mogli zapisać dane w tym obszarze, musimy przekształcić wskaźnik na właściwy typ, stosując operator rzutowania.

Funkcja `malloc` działa podobnie, ale ma tylko jeden parametr — łączną ilość alokowanych bajtów; funkcja ta nie zeruje automatycznie zaalokowanego obszaru.

Funkcje dynamicznie alokujące pamięć są zadeklarowane w standardowym pliku nagłówkowym `<stdlib.h>`, który powinniśmy włączyć do wszystkich programów, w których chcemy korzystać z dynamicznej alokacji pamięci.

Operator sizeof

Aby w sposób niezależny od maszyny określić wielkość danych, na które chcemy zarezerwować pamięć za pomocą funkcji `calloc` lub `malloc`, powinniśmy używać operatora `sizeof`. Operator ten zwraca wielkość wskazanego elementu w bajtach. Jego parametrem może być zmienna, nazwa tablicy, nazwa typu podstawowego, nazwa typu złożonego lub wyrażenie. Jeśli na przykład napiszemy:

```
sizeof (int)
```

otrzymamy liczbę bajtów niezbędnych do zapisania liczby typu `int`. W przypadku komputera z Pentium 4 jest to 4, gdyż liczba `int` zajmuje 32 bity. Jeśli `x` jest tablicą 100 liczb typu `int`, wyrażenie:

```
sizeof (x)
```

poda ilość pamięci potrzebnej na zapisanie 100 liczb całkowitych zawartych w x (w komputerze z Pentium 4 będzie to zatem 400). Wyrażenie:

```
sizeof (struct dataEntry)
```

zwróci ilość pamięci potrzebnej na strukturę `dataEntry`. W końcu, jeśli `data` jest tablicą elementów `struct dataEntry`, wyrażenie:

```
sizeof (data) / sizeof (struct dataEntry)
```

poda liczbę elementów z tablicy `data` (tablica ta musi być wcześniej zadeklarowana, nie może być parametrem formalnym funkcji ani tablicą wskazywaną z zewnątrz). Wyrażenie:

```
sizeof (data) / sizeof (data[0])
```

także zwraca ten sam wynik. Makro:

```
#define  ELEMENTS(x)    (sizeof(x) / sizeof (x[0]))
```

po prostu stanowi uogólnienie opisanej tu techniki. Dzięki temu możemy pisać kod następująco:

```
if ( i >= ELEMENTS (data) )
    ...
```

oraz tak:

```
for ( i = 0;  i < ELEMENTS (data);  ++i )
    ...
```

Trzeba pamiętać, że `sizeof` jest operatorem, a nie funkcją, choć bardziej przypomina funkcję. Operator ten jest interpretowany w chwili kompilowania programu, a nie podczas działania programu, chyba że parametrem jest tablica zmiennej długości. Jeśli nie, kompilator wyznacza wartość wyrażenia z `sizeof` i zastępuje całość konkretną wartością traktowaną jako stała.

Operatora `sizeof` należy używać zawsze, kiedy tylko jest to możliwe, aby uniknąć konieczności obliczania i zaszywania w programie konkretnych wielkości.

Wróćmy teraz do dynamicznego alokowania pamięci — jeśli chcemy zaalokować pamięć na 1000 liczb całkowitych, funkcję `calloc` możemy wywołać następująco:

```
#include <stdlib.h>
    ...
int  *intPtr;
    ...
intPtr = (int *) calloc (sizeof (int), 1000);
```

Gdy skorzystamy z funkcji `malloc`, analogiczne wywołanie wygląda tak:

```
intPtr = (int *) malloc (1000 * sizeof (int));
```

Pamiętajmy, że zarówno `calloc`, jak i `malloc` zwracają wskaźnik typu `void`, który trzeba jeszcze rzutować na odpowiedni typ. W powyższym przykładzie rzutowaliśmy uzyskany wskaźnik na wskaźnik na liczbę całkowitą i dopiero wtedy przypisaliśmy zmiennej `intPtr`.

Jeśli zażądamy więcej pamięci, niż mamy do dyspozycji, funkcja `calloc` (jak i `malloc`) zwróci wskaźnik pusty. Gdy używamy obu tych funkcji, musimy sprawdzać wskaźnik, czy faktycznie otrzymaliśmy potrzebną pamięć.

Poniższy fragment kodu alokuje pamięć na 1000 wskaźników liczb całkowitych i sprawdza zwrócony wskaźnik. Jeśli alokacja się nie powiodła, program wyświetla komunikat o błędzie i kończy swoje działanie.

```
#include <stdlib.h>
#include <stdio.h>
    ...
int  *intPtr;
    ...
intptr = (int *) calloc (sizeof (int), 1000);

if ( intPtr == NULL )
{
    fprintf (stderr, "Nie powiodło się wywołanie calloc()\n");
    exit (EXIT_FAILURE);
}
```

Jeśli alokacja się uda, można używać wskaźnika `intPtr` tak, jakby wskazywał tablicę 1000 liczb całkowitych. Aby zatem ustawić wszystkie tysiąc elementów na –1, moglibyśmy napisać:

```
for ( p = intPtr; p < intPtr + 1000; ++p )
    *p = -1;
```

pod warunkiem, że `p` zadeklarowano jako wskaźnik na liczbę całkowitą.

Aby zarezerwować pamięć na `n` elementów typu `struct dataEntry`, najpierw trzeba zadeklarować wskaźnik odpowiedniego typu:

```
struct dataEntry  *dataPtr;
```

Teraz możemy wywołać funkcję `calloc`, rezerwując odpowiednią liczbę elementów:

```
dataPtr = (struct dataEntry *) calloc (n, sizeof (struct dataEntry));
```

Wykonywanie powyższej instrukcji odbywa się następująco:

1. Wywoływana jest funkcja `calloc` z dwoma parametrami. Pierwszy z nich określa ilość `n` elementów, drugi — wielkość każdego elementu.
2. Funkcja `calloc` zwraca wskaźnik do zaalokowanej pamięci. Jeśli pamięci nie uda się zaalokować, funkcja zwraca wskaźnik pusty.
3. Wskaźnik jest rzutowany na typ „wskaźnik struktury `dataEntry`”, po czym jest przypisywany do zmiennej `dataPtr`.

Powtórzmy, że konieczne jest stałe kontrolowanie wartości `dataPtr`, aby sprawdzać poprawność zaalokowania pamięci. Wskaźnika tego używa się normalnie, jakby wskazywał tablicę `n` elementów `dataEntry`. Jeśli na przykład struktura `dataEntry` ma pole `index`, możemy temu polu przypisać wartość 100:

```
dataPtr->index = 100;
```

Funkcja free

Kiedy kończymy korzystanie z pamięci zaalokowanej za pomocą funkcji `malloc` lub `calloc`, powinniśmy użyć funkcji `free`, aby pamięć tę zwrócić do systemu. Jedynym parametrem tej funkcji jest wskaźnik na początek zaalokowanego bloku, zwrócony przy wywołaniu `calloc` lub `malloc`. Zatem wywołanie:

```
free (dataPtr);
```

zwróci pamięć zaalokowaną przez pokazane wcześniej wywołanie `calloc` — jednak pod warunkiem, że `dataPtr` nadal wskazuje *początek* zaalokowanej pamięci.

Funkcja `free` nie zwraca żadnej wartości.

Pamięć zwolniona przy użyciu funkcji `free` może być ponownie wykorzystana przez kolejne wywołania `calloc` i `malloc`. W przypadku programów, które muszą alokować więcej pamięci niż jednorazowo dostępna, użycie funkcji `free` jest bezwzględnie konieczne. Trzeba tylko pilnować, aby funkcja `free` otrzymywała prawidłowy wskaźnik początku zaalokowanej wcześniej pamięci.

Dynamiczne alokowanie pamięci jest nieocenione, kiedy mowa o powiązanych strukturach danych, takich jak listy powiązane. Kiedy do listy dodajemy nowy element, możemy dynamicznie zaalokować pamięć na ten element, po czym powiązać listę z pamięcią zaalokowaną funkcją `calloc` lub `malloc`. Załóżmy na przykład, że `listEnd` wskazuje koniec jednokrotnie powiązanej funkcji listy elementów typu `struct entry`:

```
struct entry
{
    int           value,
    struct entry  *next;
};
```

Oto funkcja `addEntry`, mająca jeden parametr — wskaźnik początku listy powiązanej. Funkcja ta dodaje na koniec listy nową pozycję.

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

// dodajemy na koniec powiązanej listy nowy element

struct entry *addEntry (struct entry *listPtr)
{
    // znajdujemy koniec listy

    while ( listPtr->next != NULL )
        listPtr = listPtr->next;

    // alokujemy pamięć na nowy element

    listPtr->next = (struct entry *) malloc (sizeof (struct entry));

    // ustawiamy nowy koniec listy

    if ( listPtr->next != NULL )
        (listPtr->next)->next = (struct entry*) NULL;

    return listPtr->next;
}
```

Jeśli alokowanie pamięci się uda, w polu `next` nowo alokowanego hasła z listy (wskazywanego przez `listPtr->next`) wstawiany jest wskaźnik pusty.

Funkcja zwraca wskaźnik do nowego elementu listy lub pusty wskaźnik, gdy alokowanie się nie powiedzie. Jeśli przywołamy w pamięci obraz listy powiązanej i prześledzimy na nim sposób wykonania funkcji `addEntry`, łatwiej będzie zrozumieć działanie wszystkich omawianych funkcji.

Jest jeszcze jedna funkcja związana z dynamicznym alokowaniem pamięci — to funkcja `realloc`. Można jej używać do powiększania lub zmniejszania wcześniej zaalokowanego obszaru pamięci. Więcej szczegółów na ten temat podajemy w dodatku B.

W tym rozdziale kończymy omawianie języka C. W rozdziale 17. nauczymy się usuwać błędy z programów napisanych w C i poznamy pewne przydatne techniki. Jedna z nich zakłada wykorzystanie preprocesora, inna użycie specjalnego narzędzia — interaktywnego programu uruchomieniowego.

Ćwiczenia

1. Przepisz i wykonaj przykładowy program z tego rozdziału. Sprawdź wyniki, porównując oryginalny plik wybrany do skopiowania z plikiem podanym do skopiowania.
2. Dokończ program pobierający słowo jako argument wiersza poleceń i szukający tego słowa w swojej tablicy pojęć i definicji. Jeśli znajdzie to słowo, powinien wyświetlić jego definicję. W przeciwnym przypadku niech program informuje użytkownika, że szukane słowo nie zostało znalezione.

17

Usuwanie błędów z programów

W tym rozdziale poznamy dwie techniki usuwania błędów z programów. Jedna z nich to zastosowanie preprocesora do włączania do programu instrukcji usuwania błędów. Druga technika polega na użyciu interaktywnego programu uruchomieniowego. Posłużymy się popularnym programem *gdb*. Jeśli nawet ktoś korzysta z innego programu uruchomieniowego (takiego jak *dbx* czy program wbudowany w któreś z narzędzi IDE), to zasady działania są zwykle podobne.

Część tematów opisanych w tym rozdziale, podobnie jak to było w przypadku rozdziału 14., może nie dotyczyć niektórych systemów operacyjnych i środowisk programistycznych, ale same przedstawiane koncepcje są równie istotne dla każdego.

Usuwanie błędów za pomocą preprocesora

Jak wspominaliśmy w rozdziale 12., kompilacja warunkowa jest przydatna do uruchamiania programów. Preprocesor języka C może wstawiać do programu kod obsługujący błędy. Jeśli dobrze dobierzemy dyrektywy `#ifdef`, kod obsługi błędów może być w miarę potrzeb włączany lub wyłączany. W programie 17.1 wczytywane są trzy liczby całkowite i pokazywana jest ich suma. Jeśli zdefiniowana jest nazwa preprocesora `DEBUG`, kod obsługi błędów (wstawiający dane do strumienia `stderr`) jest wkompilowywany do całego programu. Jeśli nazwa `DEBUG` nie jest zdefiniowana, kod obsługi błędów nie jest włączany.

Program 17.1. **Dodawanie instrukcji związanych z usuwaniem błędów za pomocą preprocesora**

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

int process (int i, int j, int k)
{
    return i + j + k;
```

```
}
int main (void)
{
    int  i, j, k, nread;

    nread = scanf ("%d %d %d", &i, &j, &k);

#ifdef DEBUG
    fprintf (stderr, "Liczba wczytanych liczb = %i\n", nread);
    fprintf (stderr, "i = %i, j = %i, k = %i\n", i, j, k);
#endif

    printf ("%i\n", process (i, j, k));
    return 0;
}
```

Program 17.1. **Wyniki**

```
1 2 3
Liczba wczytanych liczb = 3
i = 1, j = 2, k = 3
6
```

Program 17.1. **Wyniki (ponowne uruchomienie)**

```
1 2 e
Liczba wczytanych liczb = 2
i = 1, j = 2, k = 0
3
```

Wartość zmiennej k w drugim wypadku może być dowolna, gdyż zmienna ta nie została ustawiona w funkcji scanf, a wcześniej nie była zainicjalizowana.

Instrukcje:

```
#ifdef DEBUG
    fprintf (stderr, "Liczba wczytanych liczb = %i\n", nread);
    fprintf (stderr, "i = %i, j = %i, k = %i\n", i, j, k);
#endif
```

są analizowane przez preprocesor. Jeśli zdefiniowany jest identyfikator DEBUG (#ifdef DEBUG), preprocesor włącza do programu obie instrukcje fprintf do kompilacji. Jeśli identyfikator DEBUG nie jest zdefiniowany, obie instrukcje są pomijane w dalszej kompilacji. Jak widać, po wczytaniu liczb program pokazuje dodatkowe komunikaty. W przypadku drugiego uruchomienia programu użytkownik podał niewłaściwy znak — e. Teraz dodatkowy komunikat informuje o błędzie. Zauważmy, że aby wyłączyć cały kod do obsługi błędów, wystarczy usunąć z programu wiersz:

```
#define DEBUG
```

i już wywołania fprintf nie będą do programu włączane. Choć nasz program jest na tyle krótki, że można dyskutować sensowność stosowania dodatkowych dyrektyw, to zwróćmy

uwagę, jak łatwo będzie w podobny sposób włączać i wyłączać pomocniczy kod w programie liczącym kilkaset wierszy — także wystarczy zmiana jednego wiersza.

Można nawet kontrolować włączanie dodatkowego kodu w chwili wywoływania kompilatora z wiersza poleceń. Jeśli korzystamy z kompilatora *gcc*, polecenie:

```
gcc –D DEBUG debug.c
```

skompiluje plik *debug.c* i jednocześnie zdefiniuje identyfikator `DEBUG`. Jest to równoważne wstawieniu do programu wiersza:

```
#define DEBUG
```

Przyjrzyjmy się nieco dłuższemu programowi 17.2. Ma on dwa parametry w wierszu poleceń. Oba są konwertowane na liczby całkowite, które przypisywane są odpowiednio zmiennym `arg1` i `arg2`. Aby przekształcić parametry wiersza poleceń na liczby, używamy funkcji `atoi`. Funkcja ta jako parametr przyjmuje łańcuch znakowy i zwraca odpowiadającą takiemu łańcuchowi liczbę całkowitą. Funkcja ta jest zadeklarowana w standardowym pliku nagłówkowym — `<stdlib.h>` — który włączamy do naszego programu.

Program 17.2. **Kompilacja kodu pomagającego w usuwaniu błędów**

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

int process (int i1, int i2)
{
    int  val;

#ifdef DEBUG
    fprintf (stderr, "process (%i, %i)\n", i1, i2);
#endif
    val = i1 * i2;
#ifdef DEBUG
    fprintf (stderr, "return %i\n", val);
#endif
    return val;
}

int main (int argc, char *argv[])
{
    int arg1 = 0, arg2 = 0;

    if (argc > 1)
        arg1 = atoi (argv[1]);
    if (argc == 3)
        arg2 = atoi (argv[2]);
#ifdef DEBUG
    fprintf (stderr, "przetworzono %i parametrów\n", argc – 1);
    fprintf (stderr, "arg1 = %i, arg2 = %i\n", arg1, arg2);
#endif
    printf ("%i\n", process (arg1, arg2));

    return 0;
}
```

Program 17.2. **Wyniki**

```
$ gcc –D DEBUG p18-2.c       Kompilacja ze zdefiniowaną nazwą DEBUG
$ a.out 5 10
przetworzono 2 parametrów
arg1 = 5, arg2 = 10
process (5, 10)
return 50
50
```

Program 17.2. **Wyniki (ponowne uruchomienie)**

```
$ gcc p18-2.c                Kompilacja bez zdefiniowanej nazwy DEBUG
$ a.out 2 5
10
```

Po przetworzeniu swoich parametrów program wywołuje funkcję `process`, przekazując jej jako parametry wartości z wiersza poleceń. Jak widać, jeśli zdefiniowany jest identyfikator `DEBUG`, pokazywane są różne komunikaty pomagające usuwać błędy; kiedy identyfikator ten nie jest zdefiniowany, pokazywane są tylko wyniki działania programu.

Kiedy program jest gotów do rozpowszechniania, instrukcje usuwania błędów można wyłączyć z kodu, tak aby nie wpływały na plik wykonywalny; wystarczy nie definiować nazwy `DEBUG`. Jeśli później zostanie znaleziony jakiś błąd, ponownie można wkompilować instrukcje pomocnicze, aby sprawdzić, co się dzieje.

Opisana metoda jest o tyle niewygodna, iż powoduje, że czytanie programów staje się utrudnione. Można jednak zmienić sposób użycia preprocesora i zdefiniować makro mające zmienną liczbę parametrów, które będzie wstawiało pomocniczy kod:

```
#define DEBUG(fmt, ...) fprintf (stderr, fmt, __VA_ARGS__)
```

Teraz możemy używać tego makra zamiast funkcji `fprintf`:

```
DEBUG ("process (%i, %i)\n", i1, i2);
```

Powyższe makro zostanie rozwinięte do postaci:

```
fprintf (stderr, "process (%i, %i)\n", i1, i2);
```

Makro `DEBUG` może być używane w dowolnym miejscu programu, a zasada jego działania jest oczywista, co widać w programie 17.3.

Program 17.3. **Definiowanie makra DEBUG**

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

#define DEBUG(fmt, ...) fprintf (stderr, fmt, __VA_ARGS__)

int process (int i1, int i2)
{
    int  val;

    DEBUG( "process (%i, %i)\n", i1, i2);
```

```
    val = i1 * i2;
    DEBUG( "return %i\n", val);

    return val;
}

int main (int argc, char *argv[])
{
    int arg1 = 0, arg2 = 0;

    if (argc > 1)
        arg1 = atoi (argv[1]);
    if (argc == 3)
        arg2 = atoi (argv[2]);

    DEBUG( "przetworzono %i parametrów\n", argc - 1);
    DEBUG( "arg1 = %i, arg2 = %i\n", arg1, arg2);
    printf ("%i\n", process (arg1, arg2));

    return 0;
}
```

Program 17.3. **Wyniki**

```
$ gcc pre3.c
$ a.out 8 12
przetworzono 2 parametrów
arg1 = 8, arg2 = 12
process (8, 12)
return 96
96
```

Jak widać, tak zapisany program jest znacznie bardziej czytelny. Kiedy nie potrzebujemy pomocy w usuwaniu błędów, po prostu wstawiamy pustą definicję makra `DEBUG`:

```
#define DEBUG(fmt, ...)
```

W ten sposób informujemy preprocesor, że ma zastąpić wywołania makra `DEBUG` „niczym", czyli po prostu usunąć wszystkie wywołania tego makra.

Można jeszcze bardziej rozwinąć makro `DEBUG` tak, aby można było nim sterować zarówno na etapie kompilacji, jak i na etapie wykonywania programu. W tym celu deklarujemy zmienną globalną `Debug` opisującą poziom informowania o stanie programu. Wszystkie instrukcje `DEBUG` mające poziom niższy od bieżącego lub mu równy, pokazują swoje wyniki; pozostałe „milczą". Teraz `DEBUG` ma zatem przynajmniej dwa parametry; pierwszym z nich jest poziom zgłaszania stanu:

```
DEBUG (1, "przetworzono dane\n");
DEBUG (3, "liczba elementów = %i\n", nelems);
```

Jeśli poziom raportowania to 1 lub 2, tylko pierwsze wywołanie `DEBUG` pokaże wynik. Gdy poziom raportowania wynosi co najmniej 3, zadziałają oba wywołania `DEBUG`. Poziom raportowania można ustawić w chwili wywołania programu, w wierszu poleceń:

```
a.out –d1              Ustawienie poziomu raportowania na 1
a.out –d3              Ustawienie poziomu raportowania na 3
```

Definicja nowej wersji makra DEBUG jest prosta:

```
#define DEBUG(level, fmt, ...) \
    if (Debug >= level) \
        fprintf (stderr, fmt, __VA_ARGS__)
```

Zatem wywołanie:

```
DEBUG (3, "liczba elementów = %i\n", nelems);
```

zostanie rozwinięte jako:

```
    if (Debug >= 3)
        fprintf (stderr, "liczba elementów = %i\n", nelems);
```

Jeśli zdefiniujemy puste makro DEBUG, wszystkie jego wywołania będą usuwane przez preprocesor podczas kompilacji.

Poniższa definicja spełnia wszystkie podane dotąd wymagania, a przy tym umożliwia kontrolowanie zachowania DEBUG na etapie kompilacji.

```
#ifdef DEBON
#  define DEBUG(level, fmt, ...) \
      if (Debug >= level) \
          fprintf (stderr, fmt, __VA_ARGS__)
#else
#  define DEBUG(level, fmt, ...)
#endif
```

Podczas kompilacji programu zawierającego powyższą definicję (definicję tę najwygodniej umieścić w pliku nagłówkowym i do programu włączać) można zdefiniować nazwę DEBON lub nie. Jeśli skompilujemy program *prog.c* następująco:

```
$ gcc prog.c
```

wkompilowana zostanie „pusta definicja" makra DEBUG, pochodząca z gałęzi `#else` z powyższego kodu. Jeśli z kolei program skompilujemy następująco:

```
$ gcc –D DEBON prog.c
```

w program zostanie wbudowane makro DEBUG wywołujące funkcję `fprintf` w zależności od bieżącego poziomu raportowania.

Jeśli w czasie wykonywania programu podczas kompilacji włączyliśmy raportowanie, możemy ustalić jego poziom. Jak już mówiliśmy, służy do tego opcja wiersza poleceń:

```
$ a.out –d3
```

W tym wypadku poziom raportowania ustalono na 3. Wartość tej opcji pobieramy w programie z wiersza poleceń i ustawiamy odpowiednio (globalną) zmienną Debug określającą poziom raportowania. Funkcja `fprintf` zostanie wywołana tylko wtedy, gdy dane wywołanie makra DEBUG będzie miało poziom równy co najmniej 3.

Zauważmy, że wywołanie `a.out –d0` ustawia poziom raportowania na zero; wtedy nie są pokazywane żadne informacje pomocne w usuwaniu błędów, choć odpowiedni kod cały czas istnieje w programie.

Podsumowując, stwierdzamy, że utworzyliśmy dwuwarstwowy system raportowania — kod raportujący może być włączany do programu lub nie. Kiedy na etapie kompilacji zostanie do kodu włączony, to stosowanie różnych poziomów raportowania może generować odmienne komunikaty.

Usuwanie błędów przy użyciu programu gdb

Program *gdb* to interaktywny program uruchomieniowy (tzw. *debugger*) o ogromnych możliwościach. Często jest używany w parze z kompilatorem GNU — *gcc*. Pozwala uruchamiać program, zatrzymywać go w wybranym miejscu, pokazywać lub ustawiać zmienne oraz podejmować wykonanie przerwanego programu. Pozwala śledzić wykonywanie programu, a nawet wykonywać program krokowo, wiersz po wierszu. Program *gdb* umożliwia też określenie miejsca generowania pliku stanu (tzw. *core dump*). Plik taki jest generowany, kiedy wystąpi jakieś nieprzewidziane zdarzenie, jak dzielenie przez zero czy próba sięgnięcia poza zakres tablicy. Tworzony jest plik o nazwie *core*, który zawiera obraz pamięci procesu w chwili kończenia wykonywania programu[1].

Aby można było skorzystać ze wszystkich możliwości *gdb*, program trzeba skompilować za pomocą kompilatora *gcc* z opcją `–g`. Opcja ta powoduje dodanie przez kompilator dodatkowych informacji, takich jak typy zmiennych i struktur, nazwy plików źródłowych oraz instrukcje języka C, do pliku wynikowego.

Program 17.4 próbuje sięgnąć do elementów spoza dopuszczalnego zakresu tablicy.

Program 17.4. **Prosty program do pokazania, jak używamy gdb**

```
#include <stdio.h>

int main (void)
{
    const int data[5] = {1, 2, 3, 4, 5};
    int  i, sum;

    for (i = 0; i >= 0; ++i)
        sum += data[i];

    printf ("suma = %i\n", sum);

    return 0;
}
```

[1] System może być skonfigurowany tak, aby wyłączyć automatyczne tworzenie pliku *core*. Wiąże się to z dużym rozmiarem takich plików. Czasami brak takiego pliku wynika z ustawienia maksymalnej wielkości pliku, jaki użytkownik może utworzyć (polecenie *ulimit*).

Oto wynik działania powyższego programu w systemie Mac OS X w oknie terminala (w innych systemach mogą pojawiać się inne komunikaty):

```
$ a.out
Segmentation fault
```

Skorzystajmy z programu uruchomieniowego *gdb*, aby odnaleźć błąd. Przykład niewątpliwie jest nieco naciągany, ale dobrze pokazuje, o co naprawdę chodzi.

Najpierw sprawdźmy, czy program skompilowaliśmy z opcją –g. Następnie możemy uruchomić *gdb* z plikiem wykonywalnym, domyślnie *a.out*. Najpierw pojawiają się informacje wstępne:

```
$ gcc –g p18.8.c       Ponownie kompilujemy program z informacjami dla gdb
$ gdb a.out            Uruchamiamy gdb z plikiem wykonywalnym
GNU gdb 5.3-20030128 (Apple version gdb-309) Thu Dec  4 15:41:30 GMT 2003)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Typ "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "powerpc-apple-darwin".
Reading symbols for shared libraries .. done
```

Kiedy program *gdb* jest gotów do działania, pokazuje znak zachęty — (gdb). W naszym prostym przykładzie po prostu nakazujemy uruchomienie programu, podając polecenie run. Powoduje to, że *gdb* zaczyna wykonywać program, aż pojawi się sytuacja krytyczna:

```
(gdb) run
Starting program: /Users/stevekochan/MySrc/c/a.out
Reading symbols for shared libraries . done

Program received signal EXC_BAD_ACCESS, Could not access memory.
0x00001d7c in main () at p18-4.c:9
9                sum += data[i];
(gdb)
```

Zatem w naszym programie, tak samo jak poprzednio, wystąpił błąd, ale tym razem sytuacja nadal jest pod kontrolą *gdb*. Co ważne, teraz w takiej sytuacji możemy sprawdzić wartości poszczególnych zmiennych.

Jak widać powyżej, w dziewiątym wierszu program usiłował nieprawidłowo sięgnąć do pamięci. Problematyczny wiersz programu jest automatycznie pokazywany. Aby uzyskać nieco informacji o okolicznościach błędu, korzystamy z polecenia list. Wyświetla ono 9 wierszy otaczających wiersz wskazany (5 wierszy wcześniejszych i 4 dalsze):

```
(gdb) list 9
4       {
5           const int data[5] = {1, 2, 3, 4, 5};
6           int  i, sum;
7
8           for (i = 0; i >= 0; ++i)
9               sum += data[i];
10
```

```
11          printf ("suma = %i\n", sum);
12
13          return 0;
(gdb)
```

Wartości zmiennych możemy sprawdzać przy użyciu polecenia `print`. Sprawdźmy, jaka była wartość zmiennej `sum` w chwili przerwania programu:

```
(gdb) print sum
$1 = -1089203864
```

Oczywiście taka wartość nic nie znaczy; zresztą za każdym razem może ona być całkiem inna. Zapis `$n` jest używany przez program *gdb* do numerowania wyświetlanych dotąd zmiennych, aby później łatwo było odwoływać się do nich.

Spójrzmy, jaką wartość ma zmienna indeksująca `i`:

```
(gdb) print i
$2 = 232
```

Oho! Coś jest nie tak, jak być powinno. W tablicy mamy pięć elementów, a próbujemy sięgnąć do elementu 233. W różnych systemach błąd ten może pojawić się wcześniej lub później, ale pojawi się.

Zanim zakończymy działanie *gdb*, spójrzmy na jeszcze jedną zmienną. Tak elegancko *gdb* obsługuje zmienne tablicowe i struktury:

```
(gdb) print data          Pokaż zawartość tablicy data
$3 = {1, 2, 3, 4, 5}
(gdb) print data[0]       Pokaż wartość pierwszego elementu
$4 = 1
```

Przykład pracy ze strukturą zobaczymy później. Aby zakończyć pierwszy przykład użycia *gdb*, musimy nauczyć się kończyć działanie tego programu. Służy do tego polecenie `quit`:

```
(gdb) quit
The program is running. Exit anyway? (y or n) y
$
```

Mimo że w programie wystąpił błąd, formalnie nadal był w *gdb* aktywny; błąd spowodował jedynie zawieszenie jego wykonywania, ale nie zakończenie. Dlatego właśnie program *gdb* zażądał potwierdzenia, że chcemy go zamknąć.

Użycie zmiennych

Program uruchomieniowy *gdb* zawiera dwa podstawowe polecenia służące do obsługi zmiennych. Jedno to pokazywane już `print`. Drugie pozwala ustawiać wartość zmiennej — `set var`. Polecenie `set` ma szereg różnych opcji; jedna z nich — `var` — służy właśnie do ustawiania wartości zmiennych:

```
(gdb) set var i=5
(gdb) print i
$i = 5
(gdb) set var i=i*2          Możemy użyć dowolnych poprawnych wyrażeń
```

```
(gdb) print i
$2 = 10
(gdb) set var i=$1+20        Możemy też używać tak zwanych „zmiennych podręcznych"
(gdb) print i
$3 = 25
```

Zmienna musi być w danej funkcji dostępna, a proces *aktywny*, czyli uruchomiony. Program *gdb* cały czas przechowuje aktywny wiersz, aktywny plik (chodzi o plik źródłowy programu) oraz aktualną funkcję. Kiedy *gdb* zaczyna swoje działanie bez pliku *core*, aktualną funkcją jest `main`, aktualnym plikiem jest plik zawierający funkcję `main`, a aktualnym wierszem jest pierwszy wiersz wykonywalny w funkcji `main`. Jeśli zostanie użyty plik *core*, aktualny wiersz, plik i funkcja są ustawiane w miejscu przerwania działania programu.

Jeśli nie istnieje zmienna lokalna o podanej nazwie, *gdb* szuka zmiennej zewnętrznej o takiej samej nazwie. W powyższym przykładzie w chwili wydawania poleceń funkcją aktywną była `main`, a `i` była jej zmienną lokalną.

Gdy wpisujemy nazwę zmiennej, możemy jednocześnie podać nazwę funkcji, korzystając z zapisu *`funkcja`*`::`*`zmienna`*. W ten sposób można sięgnąć do lokalnej zmiennej wskazanej funkcji, na przykład:

```
(gdb) print main::i              Pokazujemy wartość zmiennej i z funkcji main
$4 = 25
(gdb) set var main::i=0          Ustawiamy wartość zmiennej i z funkcji main
```

Zwróćmy uwagę, że próba ustawienia zmiennej w nieaktywnej funkcji (czyli funkcji, która nie jest ani aktualnie wykonywana, ani nie czeka na zakończenie działania innej funkcji) jest błędem i powoduje pojawienie się komunikatu:

```
No symbol "var" in current context.
```

czyli: „W bieżącym kontekście brak symbolu `"var"`".

Zmienne globalne są dostępne bezpośrednio jako *`'plik'`*`::`*`zmienna`*. W ten sposób wymuszamy na *gdb* skorzystanie ze zmiennej zewnętrznej z pliku *`plik`*, z pominięciem zmiennej lokalnej o takiej samej nazwie.

Do pól struktur i unii możemy sięgać, korzystając ze zwykłej składni języka C. Jeśli `datePtr` to wskaźnik na strukturę typu `date`, instrukcja `print datePtr->year` wyświetli pole `year` tej struktury.

Odwoływanie się do struktury lub unii bez podania konkretnego pola powoduje, że pokazywana jest zawartość całej struktury czy unii.

Można na *gdb* wymusić pokazanie wartości zmiennej w innym formacie, na przykład szesnastkowo. W tym celu należy za instrukcją `print` dopisać / oraz literę określającą wybrany format. Wiele poleceń programu *gdb* można skracać do jednej litery. W poniższym przykładzie skracamy polecenie `print` do samego `p`:

```
(gdb) set var i=35      Ustawienie zmiennej i na 35
(gdb) p /x i            Pokazanie wartości i szesnastkowo
$i = 0x23
```

Pokazywanie plików źródłowych

Program *gdb* zawiera polecenia dające dostęp do plików źródłowych. Umożliwia to usuwanie błędów z programów bez konieczności sięgania do tych programów w edytorze.

Jak wspomniano wcześniej, *gdb* korzysta z aktualnego wiersza i aktualnego pliku. Widzieliśmy, jak za pomocą polecenia `list` można wyświetlić otoczenie aktualnego wiersza (polecenie `list` można skracać do `l`). Przy każdym następnym wpisaniu polecenia `list` (lub przy wciśnięciu klawisza *Return*) program pokazuje 10 następnych wierszy pliku źródłowego. Wartość 10 jest ustawieniem domyślnym, które może być zmieniane poleceniem `listsize`.

Jeśli chcemy wyświetlić grupę wierszy, możemy podać numer początkowy i końcowy, rozdzielone przecinkiem:

```
(gdb) list 10,15          Pokaż wiersze od 10. do 15.
```

Wiersze składające się na definicję funkcji możemy wyświetlić, podając nazwę tej funkcji jako parametr polecenia `list`:

```
(gdb) list foo        Pokazanie wierszy tworzących funkcję foo
```

Jeśli funkcja znajduje się w innym pliku źródłowym, program *gdb* automatycznie przełącza się na ten plik. Nazwę aktualnego pliku można sprawdzić poleceniem `info source`.

Podanie znaku plus po poleceniu `list` powoduje pokazanie następnych 10 wierszy z bieżącego pliku, czyli jest to równoważne wydaniu samego polecenia `list`. Podanie znaku minus powoduje pokazanie poprzednich 10 wierszy. Za opcjami + i – można podawać liczbę wierszy.

Kontrola nad wykonywaniem programu

Pokazywanie wierszy z pliku nie zmienia sposobu działania programu. Do tego celu trzeba używać innych poleceń. Widzieliśmy już dwa polecenia sterujące wykonywaniem programu — `run` uruchamiające program od początku oraz `quit` kończące jego działanie.

Za poleceniem `run` można podać parametry wiersza poleceń, a także ewentualne przekierowania (< i >). Kolejne wywołanie `run` bez żadnych parametrów spowoduje skorzystanie z tych samych parametrów i przekierowań co poprzednio. Aktualne parametry można wyświetlić za pomocą polecenia `show args`.

Wstawianie punktów przerwania

Do wstawienia w programie *punktu przerwania* można użyć polecenia `break`. Punkt przerwania, jak sama nazwa wskazuje, to miejsce, w którym wykonywanie programu jest wstrzymywane. Można wtedy sprawdzić wartości zmiennych i dokładnie przeanalizować stan programu w danym miejscu.

Punkt przerwania można wstawić w dowolnym wierszu programu; wystarczy podać numer wiersza. Jeśli podamy numer wiersza, ale nie podamy funkcji ani pliku, punkt zostanie wstawiony w pliku bieżącym. Jeśli podamy funkcję, punkt przerwania jest wstawiany na pierwszej wykonywalnej instrukcji tej funkcji.

```
(gdb) break 12           Ustawienie punktu przerwania w 12. wierszu
Breakpoint 1 at 0x1da4: file mod1.c, line 12.
(gdb) break main         Ustawienie punktu przerwana na początku funkcji main
Breakpoint 2 at 0x1d6c: file mod1.c, line 3.
(gdb) break mod2.c:foo   Punkt przerwania w funkcji foo w pliku mod2.c
Breakpoint 3 at 0x1dd8: file mod2.c, line 4.
```

Kiedy podczas wykonywania program dojdzie do punktu przerwania, *gdb* zawiesza jego działanie i umożliwia użytkownikowi interakcję, pokazując aktualny punkt przerwania i jego wiersz. W tej chwili możemy zrobić wszystko: wyświetlać i ustawiać zmienne, ustawiać i usuwać punkty przerwania i tak dalej. Aby podjąć dalsze wykonanie programu, piszemy po prostu `continue` skracane do `c`.

Praca krokowa

Innym przydatnym poleceniem do sterowania wykonywaniem programu jest polecenie `step`, skracane do `s`. Powoduje ono wykonanie w programie jednego wiersza programu. Jeśli za poleceniem `step` podamy liczbę, wykonana zostanie taka liczba kroków. Zauważmy, że jeden wiersz może zawierać wiele instrukcji języka C; jednak *gdb* działa w oparciu o wiersze i w jednym kroku wykona wszystkie instrukcje z takiego wiersza. Jeśli instrukcja jest zapisana w wielu wierszach, jeden krok w pierwszym wierszu tej instrukcji powoduje wykonanie wszystkich jej wierszy. Krokowego wykonywania programu można używać zawsze tam, gdzie można użyć polecenia `continue` — czyli po punkcie przerwania lub po odebraniu sygnału.

Jeśli aktualny wiersz zawiera wywołanie funkcji, polecenie `step` powoduje wejście do tej funkcji (pod warunkiem że nie jest to funkcja z biblioteki systemowej; do tych się nie wchodzi). Jeśli zamiast `step` użyjemy polecenia `next`, *gdb* wywoła funkcję, ale nie wejdzie do niej w trybie krokowym.

Pewne cechy *gdb* wypróbujemy w programie 17.5, który sam nie jest przydatny do niczego konkretnego.

Program 17.5. **Użycie programu uruchomieniowego gdb**

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

struct date {
    int day;
    int month;
    int year;
};


struct date foo (struct date x)
{
    ++x.day;

    return x;
}
int main (void)
{
```

```
    struct date today = {11, 10, 2014};
    int         array[5] = {1, 2, 3, 4, 5};
    struct date *newdate, foo ();
    char        *string = "test string";
    int         i = 3;

    newdate = (struct date *) malloc (sizeof (struct date));
    newdate->day = 15;
    newdate->month = 11;
    newdate->year = 2014;

    today = foo (today);

    free (newdate);

    return 0;
}
```

Oto przykładowa sesja z programem 17.5. Czytelnik może uzyskać nieco inne wyniki, gdyż zależą one od wersji i systemu, w którym program *gdb* jest uruchamiany.

Program 17.5. **Sesja z gdb**

```
$ gcc –g p18-5.c
$ gdb a.out
GNU gdb 5.3-20030128 (Apple version gdb-309) Thu Dec  4 15:41:30 GMT 2003)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Typ "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "powerpc-apple-darwin".
Reading symbols for shared libraries .. done
(gdb) list main
14
15          return x;
16      }
17
18      int main (void)
19      {
20          struct date today = {11, 10, 2014};
21          int         array[5] = {1, 2, 3, 4, 5};
22          struct date *newdate, foo ();
23          char        *string = "test string";
(gdb) break main              Ustawienie punktu przerwania w funkcji main
Breakpoint 1 at 0x1ce8: file p18-5.c, line 20.
(gdb) run                     Uruchomienie programu
Starting program: /Users/stevekochan/MySrc/c/a.out
Reading symbols for shared libraries . done

Breakpoint 1, main () at p18-5.c:20
20          struct date today = {11, 10, 2014};
(gdb) step                     Wykonanie wiersza 20
21          int         array[5] = {1, 2, 3, 4, 5};
(gdb) print today
$1 = {
 day = 11,
```

```
 month = 10,
 year = 2014}
(gdb) print array              Tablica ta nie została jeszcze zainicjalizowana
$2 = {-1881069176, -1880816132, -1880815740, -1880816132, -1880846287}
(gdb) step                     Wykonanie następnego wiersza
23          char        *string = "test string";
(gdb) print array              Sprawdźmy teraz
$3  = {1, 2, 3, 4, 5}          Już lepiej
(gdb) list 23,28
23      char        *string = "test string";
24          int         i = 3;
25
26          newdate = (struct date *) malloc (sizeof (struct date));
27          newdate->day = 15;
28          newdate->month = 11;
 (gdb) step 5                   Wykonanie 5 wierszy
29          newdate->year = 2014;
(gdb) print string
$4 = 0x1fd4 "test string"
(gdb) print string[1]
$5 = 101 'e'
(gdb) print array[i]           Program ustawił i na 3
$6 = 3
(gdb) print newdate            Jest to zmienna wskaźnikowa
$7 = (struct date *) 0x100140
(gdb) print newdate->month
$8 = 11
(gdb) print newdate->day + i   Dowolne wyrażenie języka C
$9 = 18
(gdb) print $7                 Sięgamy do poprzedniej wartości
$10 = (struct date *) 0x100140
(gdb) info locals              Pokazujemy wartości wszystkich zmiennych lokalnych
today = {
  day = 11,
  month = 10,
  year = 2014}
array = {1, 2, 3, 4, 5}
newdate = (struct date *) 0x100140
string = 0x1fd4 "test string"
i = 3
(gdb) break foo                Wstawiamy punkt przerwania na początku foo
Breakpoint 2 at 0x1c98: file p18-5.c, line 13.
(gdb) continue                 Wykonujemy program dalej
Continuing

Breakpoint 2, foo (x={ day = 11, month = 10, year = 2014}) at p18-5.c:13
13        ++x.day; 0x8e in foo:25: {
(gdb) print today              Pokaż wartość zmiennej today
 No symbol "today" in current context
(gdb) print main::today        Pokaż wartość zmiennej today z funkcji main
$11 = {
  day = 11,
  month = 10,
  year = 2014}
(gdb) step
15        return x;
(gdb) print x.day
$12 = 12
```

```
(gdb) continue
Continuing.
Program exited normally.
(gdb)
```

Zwróćmy uwagę na jeszcze jedną cechę *gdb* — kiedy dojdziemy do punktu przerwania albo wykonujemy program krokowo, pokazywany jest wiersz programu, który zostanie wykonany jako następny, a nie wiersz wykonany ostatnio. Dlatego przy pierwszym wyświetleniu tablicy `array` nie była ona jeszcze zainicjalizowana. Wykonanie krokowo jednego wiersza spowodowało tę inicjalizację. Zauważmy, że deklaracje inicjalizujące zmienne automatyczne są traktowane jako wiersze wykonywalne (zresztą, naprawdę kompilator generuje dla nich kod wykonywalny).

Wyliczanie i usuwanie punktów przerwania

Kiedy punkt przerwania zostanie ustawiony, pozostaje tak długo, aż program *gdb* zakończy swoje działanie albo punkt ten zostanie usunięty. Wszystkie punkty przerwania możemy podejrzeć za pomocą polecenia `break`:

```
(gdb) info break
Num Type          Disp Enb Address    What
1   breakpoint    keep y   0x00001c9c in main at p18-5.c:20
2   breakpoint    keep y   0x00001c4c in foo at p18-5.c:13
```

Punkt przerwania możemy usunąć z danego wiersza, stosując polecenie `clear`, za którym pojawić się musi numer wiersza. Punkt można usunąć z początku funkcji przez podanie w poleceniu `clear` nazwy funkcji.

```
(gdb) clear 20      Usunięcie punktu przerwania z wiersza 20
Deleted breakpoint 1
(gdb) info break
Num Type          Disp Enb Address    What
2   breakpoint    keep y   0x00001c4c in foo at p18-5.c:13
(gdb) clear foo     Usuwamy punkt przerwania z początku funkcji foo
Deleted breakpoint 2
(gdb) info break
No breakpoints or watchpoints.
(gdb)
```

Uzyskiwanie śladu stosu

Czasami chcielibyśmy wiedzieć dokładnie, jak były wywoływane funkcje. Informacje te są przydatne przy badaniu pliku *core*. Przy użyciu polecenia `backtrace` można zajrzeć na *stos wywołań*. Polecenie `backtrace` można skrócić do `bt`. Oto przykład jego użycia w programie 17.5:

```
(gdb) break foo
Breakpoint 1 at 0x1c4c: file p18-5.c, line 13.
(gdb) run
Starting program: /Users/stevekochan/MySrc/c/a.out
Reading symbols for shared libraries . done
```

```
Breakpoint 1, foo (x={day = 11, month = 10, year = 2014}) at p18-5.c:13
13          ++x.day;
(gdb) bt                Pokaż ślad stosu
#0  foo (x={day = 11, month = 10, year = 2014}) at p18-5.c:13
#1  0x00001d48 in main () at p18-5.c:31
(gdb)
```

Po przerwaniu wykonania programu w funkcji `foo` użyliśmy polecenia `backtrace`. Na stosie widać dwa wywołania — funkcji `foo` i `main`. Jak widać, podane są też parametry tych funkcji. Różne polecenia (takie jak `up`, `down`, `frame`, `info args`), tutaj nieomawiane, pozwalają badać parametry przekazywane do poszczególnych funkcji oraz analizować zmienne lokalne.

Wywoływanie funkcji, ustawianie tablic i zmiennych

W programie *gdb* można używać wywołań funkcji:

```
(gdb) print foo(*newdate)    Wywołanie funkcji foo ze strukturą newdate
$13 = {
  day = 16,
  month = 11,
  year = 2014
}
(gdb)
```

Funkcja `foo` jest zdefiniowana w programie 17.5.

Możemy przypisać wartości tablicy lub strukturze, podając je w nawiasach klamrowych:

```
(gdb) print array
$14 = {1, 2, 3, 4, 5}
(gdb) set var array = {100, 200}
(gdb) print array
$15 = {100, 200, 0, 0}      Niepodane wartości ustawiono na 0
(gdb) print today
$16 = {
  day = 11,
  month = 10,
  year = 2014
}
(gdb) set var today={8, 8, 2014}
(gdb) print today
$17 = {
  day = 8,
  month = 8,
  year = 2014
}
(gdb)
```

Uzyskiwanie informacji o poleceniach gdb

Aby uzyskać informacje o różnych poleceniach programu *gdb* lub o typach poleceń (nazywanych w *gdb klasami*), używamy polecenia `help`.

Polecenie `help` bez parametrów poda wszystkie dostępne klasy[2].

```
(gdb) help
Lista klas poleceń:

aliases -- aliasy innych poleceń
breakpoints -- wymuszanie zatrzymania programu w wybranych punktach
data -- analizowanie danych
files -- obsługa i badanie plików
internals -- obsługa systemu
obscure -- zaciemnianie kodu
running -- wykonywanie programu
stack -- analiza stosu
status -- zapytania o stan
support -- wsparcie
tracepoint -- śledzenie wykonania programu bez przerywania go
user-defined -- polecenia użytkownika

Napisz "help" i nazwę klasy, aby zobaczyć polecenia tej klasy.
Napisz "help" i nazwę polecenia, aby zobaczyć pełny opis.
Można stosować skróty poleceń, o ile nie prowadzi to do niejednoznaczności.
```

Teraz możemy uzyskać pomoc dotyczącą jednej z wyżej wyliczonych klas:

```
(gdb) help breakpoints
Zatrzymywanie programu w wybranych punktach.

Lista poleceń:

awatch -- stała kontrola wartości wyrażenia
break -- ustawienie punktu przerwania w podanym wierszu lub funkcji
catch -- ustawienie punktu przechwytywania do wyłapywania zdarzeń
clear -- usunięcie punktu przerwania z wybranego wiersza lub funkcji
commands -- ustawienie poleceń wykonywanych po dotarciu do punktu przerwania
condition -- punkt przerwania numer N ma zadziałać tylko w razie spełnienia warunku
delete -- usunięcie wybranych punktów przerwania lub automatycznie pokazywanych wyrażeń
disable -- wyłączenie wybranych punktów
enable -- powtórne włączenie wybranych punktów
future-break -- ustawienie punktu przerwania dla wyrażenia
hbreak -- ustawienie sprzętowego punktu przerwania
ignore -- ustawienie licznika przejść bez przerwania dla punktu numer N
rbreak -- ustawienie punktu przerwania dla wszystkich funkcji pasujących do wzorca
rwatch -- ustawienie odczytu wartości dla wyrażenia
save-breakpoints -- zapisanie definicji aktywnych punktów przerwania w formie skryptu
set exception-catch-type-regexp -
          ustawienie regexp, aby pasowało do typu wyjątku przechwyconego obiektu
set exception-throw-type-regexp -
          ustawienie regexp, aby pasowało do typu wyjątku wyrzuconego obiektu
show exception-catch-type-regexp -
          pokazanie regexp, aby pasowało do typu wyjątku przechwyconego obiektu
show exception-throw-type-regexp -
          pokazanie regexp, aby pasowało do typu wyjątku wyrzuconego obiektu
tbreak -- ustawienie tymczasowego punktu przerwania
tcatch -- ustawienie tymczasowych punktów wyłapujących zdarzenia
thbreak -- ustawienie tymczasowych sprzętowych punktów przerwania
watch -- ustawienie punktu kontroli wartości wyrażenia
```

[2] Dla wygody polskiego czytelnika przetłumaczono wyświetlaną treść przykładów — *przyp. tłum.*

```
Napisz "help" i nazwę polecenia, aby zobaczyć jego pełny opis.
Można stosować skróty poleceń, o ile nie prowadzi to do niejednoznaczności.
(gdb)
```

Można w końcu podać polecenie, na przykład jedno z poleceń z poprzedniej listy.

```
(gdb) help break
Ustawia punkt przerwania we wskazanym wierszu lub funkcji.
Parametrem może być numer wiersza, nazwa funkcji lub "*" i adres.
Jeśli podano numer wiersza, punkt przerwania będzie w tym wierszu.
Jeśli podano funkcję, punkt przerwania będzie na jej początku.
Jeśli podano adres, punkt przerwania będzie pod tym adresem.
Jeśli brak parametru, używany jest aktualny adres wykonania programu z wybranej
ramki stosu. Jest to przydatne do przerywania wykonania programu przy powrocie.

W jednym miejscu może być wiele punktów przerwania. Ma to sens, jeśli
wykorzystuje się punkty przerwania warunkowe.

break ... if <cond> ustawia warunek <cond> na tworzony punkt.

Zestawienie poleceń do obsługi punktów przerwania jest dostępne po wpisaniu
"help breakpoints".
(gdb)
```

Zatem jak widać, w sam program *gdb* wbudowano mnóstwo przydatnych informacji pomocniczych. Warto z nich korzystać!

Na koniec

Z uwagi na brak miejsca nie możemy tu opisać wielu możliwości programu *gdb*. Możliwości te to:

- ustawianie tymczasowych punktów przerwania, usuwanych po dojściu do nich;
- włączanie i wyłączanie punktów przerwania, bez usuwania ich;
- zapisywanie wybranych obszarów pamięci w żądanym formacie;
- ustawianie punktów kontroli wartości, które pozwalają przerwać działanie programu, kiedy dana wartość się zmienia (na przykład kiedy zmienia się wartość zmiennej);
- podawanie listy wartości wyświetlanych w chwili zatrzymania programu;
- ustawianie według nazw własnych zmiennych pomocniczych.

Dodatkowo, jeśli korzystasz ze zintegrowanego środowiska programistycznego (*integrated development environment* — IDE), to prawdopodobnie masz do dyspozycji narzędzia diagnostyczne w dużym stopniu podobne do opisanych w tym rozdziale poleceń *gdb*. Ponieważ opisanie wszystkich dostępnych IDE w jednym rozdziale jest niewykonalne, najlepiej zrobisz, jeśli samodzielnie przejrzysz narzędzia diagnostyczne dostępne w używanym przez Ciebie programie. Możesz nawet celowo wprowadzić kilka błędów, aby zobaczyć, czy uda się je wykryć za pomocą debugera.

W tabeli 17.1 zestawiono polecenia *gdb* omówione w tym rozdziale. Pogrubieniem zaznaczono litery, które stanowią skrót polecenia.

Tabela 17.1. Najczęściej używane polecenia gdb

Polecenie	Znaczenie
Plik źródłowy	
`list [n]`[3]	Pokazuje wiersze wokół wiersza *n* lub następnych 10 wierszy, jeśli *n* nie podano.
`list m,n`	Pokazuje wiersze od *m* do *n*.
`list +[n]`	Pokazuje wiersze wokół *n* wierszy w przód w pliku lub 10 następnych wierszy, jeśli nie podano *n*.
`list –[n]`	Pokazuje wiersze wokół *n* wierszy wstecz w pliku lub 10 poprzednich wierszy, jeśli nie podano *n*.
`list func`	Pokazuje wiersze funkcji *func*.
`listsize n`	Ustala liczbę wierszy pokazywanych poleceniem `list`.
`info source`	Pokazuje nazwę aktualnego pliku źródłowego.
Zmienne i wyrażenia	
`print /fmt expr`	Pokazuje wyrażenie *expr* zgodnie z formatem *fmt*, który może być dziesiętny (`d`), bez znaku (`u`), ósemkowy (`o`), szesnastkowy (`x`), znakowy (`c`), zmiennoprzecinkowy (`f`), binarny (`t`) lub może być adresem (`a`).
`info locals`	Pokazuje wartości zmiennych lokalnych z aktualnej funkcji.
`set var var=expr`	Ustawia wartość zmiennej *var* na wyrażenie *expr*.
Punkty przerwania	
`break n`	Ustawia punkt przerwania w wierszu *n*.
`break func`	Ustawia punkt przerwania na początku funkcji *func*.
`info brak`	Pokazuje wszystkie punkty przerwania.
`clear [n]`	Usuwa punkt przerwania z wiersza *n* lub z wiersza następnego.
`clear func`	Usuwa punkt przerwania z początku funkcji *func*.

[3] Zauważmy, że w każdym poleceniu otrzymującym numer wiersza lub nazwę funkcji można dodać nazwę pliku z dwukropkiem, na przykład `list main.c:1` lub `break main.c:12`.

Tabela 17.1. Najczęściej używane polecenia gdb (ciąg dalszy)

Polecenie	Znaczenie
Uruchamianie programu	
`run [args] [<plik] [>plik]`	Uruchamia program.
`continue`	Kontynuuje wykonywanie programu.
`step [n]`	Wykonuje program krokowo, robiąc 1 lub *n* kroków.
`next [n]`	Wykonuje program krokowo, robiąc 1 lub *n* kroków, ale nie wchodzi do wnętrza funkcji.
`quit`	Kończy działanie programu *gdb*.
Pomoc	
`help [cmd]`	Pokazuje klasy poleceń lub informacje o poleceniu *cmd*.
`help [class]`	Pokazuje klasy poleceń lub informacje o klasie *class*.

18

Programowanie obiektowe

Wobec faktu, że programowanie obiektowe (określane jako OOP — *Object-Oriented Programming*) jest tak popularne, oraz wobec tego, że wiele powszechnie stosowanych języków obiektowych powstało na bazie języka C (wystarczy wymienić choćby C++, C#, Javę czy Objective-C), krótko omówimy ten rodzaj programowania.

Zaczynamy od omówienia podstawowych pojęć obiektowości, potem pokażemy prosty program zapisany w trzech ze wspomnianych wyżej języków (tylko trzech, bo trzy zawierają w swojej nazwie język C). Nie chodzi o nauczenie programowania w żadnym z tych języków, ani nawet o uchwycenie podstawowych ich cech, ale o pokazanie, jak w ogóle wygląda programowanie obiektowe. W niniejszym rozdziale można znaleźć:

- podstawowe pojęcia obiektowości — obiekty, klasy i metody;
- najważniejsze różnice między programowaniem proceduralnym a programowaniem obiektowym;
- porównanie rozwiązania problemu w trzech obiektowych językach programowania: Objective-C, C++ i C#.

Czym zatem jest obiekt?

Obiekt to *rzecz*. O programowaniu obiektowym należy myśleć jako o pewnej rzeczy i o tym, co z tą rzeczą możemy zrobić. Jest to inna filozofia niż w przypadku programowania w językach takich jak C; język C formalnie nazywany jest językiem programowania proceduralnego. W C najpierw zastanawiamy się, co chcemy zrobić (ewentualnie piszemy stosowne funkcje), a dopiero potem zastanawiamy się nad używanymi obiektami. W programowaniu obiektowym jest dokładnie odwrotnie.

Weźmy przykład z życia codziennego — samochód. Oczywiście samochód jest obiektem; obiektem posiadanym przez nas. Nie mamy samochodu jako pojęcia ogólnego; mamy konkretny samochód, zrobiony być może na warszawskim Żeraniu, może w Japonii, a może jeszcze gdzie indziej. Nasz samochód ma numer identyfikacyjny nadwozia (VIN), który jednoznacznie go określa.

W obiektowym żargonie *nasz* samochód jest *instancją* (egzemplarzem) samochodu. Sam *samochód* to nazwa *klasy*, której instancją jest nasze auto. Zatem zawsze, kiedy montowany jest nowy samochód, tworzona jest nowa instancja klasy samochód. Wszystkie instancje nazywane są *obiektami*.

Idźmy dalej — nasz samochód może być srebrny, może mieć czarne wnętrze, może mieć nadwozie otwarte lub typu sedan. Samochód może też pewne rzeczy robić. Na przykład samochodem się jedzie, tankuje się go, myje (przynajmniej powinno się myć), serwisuje i tak dalej. Wszystko to pokazano w tabeli 18.1.

Tabela 18.1. Czynności dotyczące obiektu

Obiekt	Co z nim robimy?
Nasz samochód	jedziemy
	tankujemy
	myjemy
	serwisujemy

Czynności z tabeli 18.1 mogą być robione w naszym samochodzie lub w każdym innym, na przykład nasza siostra może jechać swoim samochodem, tankować go i tak dalej.

Instancje i metody

Jednoznacznie dający się wskazać egzemplarz klasy to instancja. Wykonywane na niej czynności to *metody*. Metody mogą dotyczyć instancji, a czasem samej klasy, na przykład umycie naszego samochodu dotyczy jednej instancji (tak naprawdę wszystkie metody z tabeli 18.1 są metodami instancji). Podanie liczby modeli produkowanych przez producenta naszego samochodu dotyczy całej klasy, więc jest to metoda klasy.

W języku C++ metody instancji wywołuje się następująco:

```
Instancja.metoda();
```

W języku C# metodę wywołuje się tak samo:

```
Instancja.metoda();
```

W Objective-C składnia jest już inna:

```
[Instancja metoda]
```

Wróćmy do poprzedniego przykładu i zapiszmy wyrażenia zgodnie z powyższą składnią. Załóżmy, że `naszeAuto` jest obiektem klasy `Auto`. W tabeli 18.2 pokazano wyrażenia w omawianych językach obiektowych.

Tabela 18.2. Komunikaty w językach obiektowych

C++	C#	Objective-C	Czynność
`naszeAuto.jazda()`	`naszeAuto.jazda()`	`[naszeAuto jazda]`	Jedziemy naszym samochodem.
`naszeAuto.tankuj()`	`naszeAuto.tankuj()`	`[naszeAuto tankuj]`	Tankujemy samochód.
`naszeAuto.umyj()`	`naszeAuto.umyj()`	`[naszeAuto umyj]`	Myjemy swój samochód.
`naszeAuto.serwis()`	`naszeAuto.serwis()`	`[naszeAuto serwis]`	Serwisujemy swój samochód.

Te same metody możemy wywoływać dla auta naszej siostry:

`autoZuzy.jazda()`	`autoZuzy.jazda()`	`[autoZuzy jazda]`

Właśnie stosowanie tych samych metod do różnych obiektów jest jedną z najważniejszych cech programowania obiektowego.

Inne ważne pojęcie — polimorfizm — wiąże się z możliwością wysyłania tego samego komunikatu do instancji różnych klas. Jeśli na przykład mamy klasę `Lodz` oraz jej instancję `naszaLodz`, to dzięki polimorfizmowi możemy napisać:

```
naszaLodz.serwis()
naszaLodz.umyj()
```

Najważniejsze jest to, że możemy napisać takie metody klasy `Lodz`, które będą „wiedziały" o serwisowaniu łodzi. Serwisowanie to prawdopodobnie będzie czymś całkiem innym niż serwisowanie samochodu. To jest najważniejsze w polimorfizmie.

Ważnym rozróżnieniem między językami obiektowymi a językiem C jest fakt, że w językach obiektowych pracujemy z obiektami — samochodami czy łodziami. W języku C zwykle mamy do czynienia z funkcjami (czyli procedurami). W tak zwanych językach proceduralnych, takich jak C, możemy napisać funkcję `serwis`, a w niej zaszyć kod obsługujący różne pojazdy — samochody, łodzie i rowery. Jeśli będziemy chcieli kiedyś obsłużyć inny rodzaj pojazdu, na przykład rower, będziemy musieli zmodyfikować wszystkie funkcje, których ta zmiana dotyczy. W językach obiektowych w takiej sytuacji definiujemy nową klasę pojazdu i jej nowe metody. Nie musimy martwić się o inne klasy pojazdów — są one niezależne od naszej klasy, więc nie musimy ich modyfikować. Co więcej, możemy nawet nie mieć do nich dostępu.

W pisanych przez nas programach obiektowych zwykle nie mamy do czynienia z klasami samochodów czy łodzi, ale raczej z oknami, prostokątami, schowkami i tak dalej. Przykładowe komunikaty będą wyglądały następująco (w języku C#):

`myWindows.erase()`	Wyczyszczenie zawartości okna.
`myRect.getArea()`	Obliczenie powierzchni prostokąta.

`userText.spellCheck()`	Sprawdzenie pisowni fragmentu tekstu.
`deskCalculator.setAccumulator(0.0)`	Wyzerowanie akumulatora.
`favoritePlaylist.showSongs()`	Pokazanie piosenek z ulubionej „playlisty”.

Program w C do obsługi ułamków

Załóżmy, że potrzebny jest program obsługujący ułamki — ich dodawanie, odejmowanie, mnożenie i tak dalej. Moglibyśmy zdefiniować stosowną strukturę, a następnie przygotować zestaw funkcji obsługujących taką strukturę.

Elementarne działania na ułamkach w języku C pokazano w programie 18.1. Ustawiamy licznik i mianownik ułamka, następnie pokazujemy jego wartość.

Program 18.1. **Ułamki w języku C**

```
// Prosty program obsługujący ułamki
#include <stdio.h>

typedef struct {
    int numerator;
    int denominator;
} Fraction;

int main (void)
{
    Fraction myFract;

    myFract.numerator = 1;
    myFract.denominator = 3;

    printf ("Nasz ułamek to %i/%i\n", myFract.numerator, myFract.denominator);

    return 0;
}
```

Program 18.1. **Wyniki**

```
Nasz ułamek to 1/3
```

W trzech następnych podrozdziałach pokażemy obsługę ułamków kolejno w językach Objective-C, C++ i C#. Dalej omówimy programowanie obiektowe ogólnie, więc ten rozdział należy czytać po kolei.

Klasa Objective-C obsługująca ułamki

Język Objective-C został stworzony przez Brada Coxa na początku lat 80. Język ten oparto na języku SmallTalk-80, w 1988 roku został objęty licencją przez NeXT Software.

Kiedy w 1988 roku firma Apple przejęła NeXT Software, NEXTSTEP stał się podstawą systemu operacyjnego Mac OS X. Większość aplikacji istniejących obecnie w systemie Mac OS X została napisana właśnie w Objective-C.

Program 18.2 pokazuje definicję i użycie klasy `Fraction`.

Program 18.2. **Ułamki w języku Objective-C**

```
// Program obsługujący ułamki — język Objective-C

#import <stdio.h>
#import <objc/Object.h>

//-------- sekcja @interface --------

@interface Fraction: Object
{
    int numerator;
    int denominator;
}
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
-(void) print;

@end

//-------- sekcja @implementation --------

@implementation Fraction;

// pobieranie wartości pól

-(int) numerator
{
    return numerator;
}

-(int) denominator
{
    return denominator;
}

// ustawianie wartości pól

-(void) setNumerator: (int) num
{
    numerator = num;
}

-(void) -setDenominator: (int) denom
{
    denominator = denom;
}

// inne

-(void) print
{
```

```
    printf ("Wartość ułamka to %i/%i\n", numerator, denominator);
}

@end

//-------- sekcja programu --------

int main (void)
{
    Fraction   *myFract;


    myFract = [Fraction new];

    [myFract setNumerator: 1];
    [myFract setDenominator: 3];

    printf ("Licznik to %i, mianownik to %i\n",
        [myFract numerator], [myFract denominator]);
    [myFract print];     // korzystamy z metody wyświetlającej ułamek

    [myFract free];

    return 0;
}
```

Program 18.2. **Wyniki**

```
Licznik to 1, mianownik to 3
Wartość ułamka to 1/3
```

Jak widać, program 18.2 jest podzielony logicznie na trzy części (sekcje): `@interface`, `@implementation` oraz sekcję programu. Zwykle każdą sekcję umieszcza się w osobnym pliku. Sekcja `@interface` zwykle jest umieszczana w pliku nagłówkowym, który jest dołączany do każdego programu wykorzystującego daną klasę. W ten sposób kompilator wie, jakie są zmienne i metody w klasie.

Sekcja `@implementation` zawiera kod stanowiący implementację wymienionych wcześniej metod. W końcu sekcja programu to kod realizujący zadania, jakie przed naszym programem stawiamy.

Nazwa nowej klasy to `Fraction`, jej klasa bazowa to `Object`. Klasy dziedziczą metody i zmienne ze swoich klas bazowych.

W sekcji `@interface` widzimy deklaracje:

```
int numerator;
int denominator;
```

mówiące, że obiekt `Fraction` ma dwa pola całkowitoliczbowe, `numerator` i `denominator` (odpowiednio licznik i mianownik).

Deklarowane w tej sekcji pola to zmienne instancji. Zawsze, kiedy tworzymy nowy obiekt, powstaje nowy, niepowtarzalny zestaw tych zmiennych. Wobec tego, jeśli mamy dwa ułamki — `fracA` i `fracB` — każdy z nich ma własne zmienne instancji, czyli każdy ma własne zmienne `nominator` i `denominator`.

Musimy zdefiniować metody do obsługi ułamków. Potrzebna jest możliwość nadania ułamkowi określonej wartości. Nie mamy bezpośredniego dostępu do wewnętrznego zapisu ułamka (czyli nie mamy bezpośredniego dostępu do zmiennych instancji), musimy napisać metody ustawiające licznik i mianownik ułamka. Mogą być też potrzebne metody odczytujące zmienne instancji[1].

To, że zmienne instancji dla danego obiektu pozostają ukryte przed użytkownikiem tego obiektu, jest kolejną kluczową zasadą programowania obiektowego. Zasada ta nazywana jest *enkapsulacją danych*. Gwarantuje ona, że w przypadku rozszerzania lub modyfikowania klasy cały kod obsługujący dane klasy (czyli zmienne instancji) pozostanie w metodach klasy. Enkapsulacja danych zapewnia eleganckie oddzielenie programisty i twórcy klasy.

Oto deklaracja jednej z metod ustawiających wartość zmiennej instancji:

```
-(int) numerator;
```

Znajdujący się na początku minus (–) mówi, że metoda jest metodą instancji. Jedyna inna możliwość to użycie znaku plus (+) oznaczającego metodę klasy. Metoda klasy wykonuje pewne działania na samej klasie, na przykład tworzy instancję tej klasy. Jest to zasada podobna jak przy montażu nowego samochodu, samochód to klasa, której nową instancję chcemy utworzyć — zatem potrzebna jest nam metoda klasy.

Metoda instancji wykonuje pewne działania na konkretnej instancji klasy — ustawia wartości, pobiera wartości, wyświetla wartość i tak dalej. Kontynuując naszą analogię motoryzacyjną, możemy pójść dalej — kiedy już zmontujemy samochód, konieczne może być zatankowanie go. Tankowanie dotyczy konkretnego samochodu, więc jest to odpowiednik metody instancji.

Kiedy deklarujemy nową metodę (podobnie jak w przypadku deklarowania funkcji), informujemy kompilator Objective-C, czy metoda zwraca wartość, a jeśli tak, to jakiego typu. W tym celu podajemy odpowiedni typ w nawiasach za znakiem minus lub plus. Zatem deklaracja:

```
-(int) numerator;
```

mówi, że metoda instancji nazywająca się `numerator` zwraca wartość typu `int`. Analogicznie wiersz kodu:

```
-(void) setNumerator: (int) num;
```

definiuje metodę niezwracającą żadnej wartości, służącą do ustawienia licznika naszego ułamka.

Jeśli metoda ma parametry, po jej nazwie dopisujemy dwukropek, dopiero za nim są parametry. Zatem metody ustawiające wartości zmiennych instancji to `setNumerator:` i `setDenominator:`; każda z nich będzie miała jeden parametr. Gdyby nie dwukropek, metody te nie miałyby żadnych parametrów.

---

[1] Tak naprawdę *możemy* sięgnąć bezpośrednio do zmiennych instancji, ale jest to uważane za bardzo zły styl programowania.

Metoda `setNumerator:` pobiera parametr będący liczbą całkowitą `num` i po prostu zapisuje go w zmiennej instancji `numerator`. Analogicznie `setDenominator:` zapisuje wartość swojego parametru `denom` w zmiennej instancji `denominator`. Zwróćmy uwagę na to, że metody te mają bezpośredni dostęp do zmiennych swojej instancji.

Ostatnia metoda, jaką zdefiniowaliśmy, to `print`. Jej zadaniem jest wyświetlenie wartości ułamka. Jak widać, ta metoda nie ma parametrów i nie zwraca żadnej wartości. Po prostu wyświetla licznik i mianownik ułamka rozdzielone ukośnikiem.

W funkcji `main` definiujemy zmienną `myFract`:

```
Fraction *myFract;
```

Zatem zmienna `myFract` jest obiektem typu `Fraction`, czyli w zmiennej tej przechowujemy obiekty klasy `Fraction`. Gwiazdka znajdująca się przed nazwą zmiennej oznacza wskaźnik. Tak naprawdę zmienna `myFract` wskazuje strukturę zawierającą dane odpowiedniej instancji klasy `Fraction`.

Teraz, kiedy mamy miejsce na obiekt klasy `Fraction`, musimy taki obiekt utworzyć — tak jak w fabryce produkuje się samochód. Do tworzenia obiektu służy wiersz:

```
myFract = [Fraction new];
```

Chcemy zaalokować pamięć na nowy ułamek. Wyrażenie:

```
[Fraction new]
```

wysyła komunikat do nowo utworzonej klasy `Fraction`. Prosimy tę klasę o zastosowanie metody `new`… No tak, ale przecież nie definiowaliśmy takiej metody, więc skąd się wzięła? Otóż została odziedziczona z klasy bazowej.

Teraz możemy ustawić wartość obiektu `Fraction`. W wierszach:

```
[myFract setNumerator: 1];
[myFract setDenominator: 3];
```

właśnie to robimy. Pierwsza instrukcja wysyła komunikat `setNumerator:` do obiektu `myFract`. Przekazywany jest przy tym parametr o wartości 1. Następnie sterowanie jest przekazywane do metody `setNumerator:`, którą zdefiniowaliśmy w klasie `Fraction`. System wykonujący programy języka Objective-C „wie”, której metody należy użyć, gdyż „wie”, że `myFract` jest obiektem klasy `Fraction`.

W metodzie `setNumerator:` jest jeden tylko wiersz, który wartość przekazaną jako parametr zapisuje w zmiennej instancji `numerator`. Zatem ustawiliśmy licznik obiektu `myFract` na 1.

Dalej znajduje się wywołanie podobnie działającej metody `setDenominator:`.

Kiedy mamy już gotowy ułamek, wywołujemy jego dwie metody akcesorowe, pobierające zmienne instancji `numerator` i `denominator` obiektu `myFract`. Wyniki są wyświetlane za pomocą funkcji `printf`.

W końcu program wywołuje metodę `print`. Metoda ta pokazuje wartość ułamka będącego adresatem komunikatu. Choć w tym programie widzieliśmy, jak można pobierać wartości zmiennych instancji, to aby wykład uczynić kompletnym, zdefiniowaliśmy w klasie `Fraction` osobną metodę `print`.

Ostatni komunikat w naszym programie to:

```
[myFract free]
```

Powoduje on zwolnienie pamięci zajmowanej przez obiekt `myFract`.

Klasa C++ obsługująca ułamki

Program 18.3 to przykładowa klasa `Fraction` napisana w języku C++. Język C++ stał się bardzo popularny. Został stworzony przez Bjarne'a Stroustroupa w Bell Laboratories i był pierwszym językiem obiektowym opartym na języku C — przynajmniej my nie wiemy o żadnym wcześniejszym języku. Uwaga na temat kompilacji tego programu: jeśli korzystasz ze zintegrowanego środowiska programistycznego (IDE), w którym można kompilować zarówno programy w C, jak i C++, i do tej pory pisałeś programy w języku C, to program prawdopodobnie automatycznie zapisze nowy plik z rozszerzeniem *.c*. Może to spowodować powstanie szeregu błędów. Rozwiązaniem jest zmiana rozszerzenia na *.cpp*, aby kompilator potraktował źródło jako kod w języku C++.

Program 18.3. **Obsługa ułamków w języku C++**

```
#include <iostream>

class Fraction
{
private:
    int numerator;
    int denominator;

public:
    void setNumerator (int num);
    void setDenominator (int denom);
    int Numerator (void);
    int Denominator (void);
    void print (Fraction f);
};

void Fraction::setNumerator (int num)
{
    numerator = num;
}

void Fraction::setDenominator (int denom)
{
    denominator = denom;
}

int Fraction::Numerator (void)
{
    return numerator;
}

int Fraction::Denominator (void)
```

```
{
    return denominator;
}

void Fraction::print (Fraction f)
{
    std::cout << "Wartość ułamka to " << numerator << '/'
              << denominator << '\n';
}
int main (void)
{
    Fraction myFract;

    myFract.setNumerator (1);
    myFract.setDenominator (3);

    myFract.print (myFact);

    return 0;
}
```

Program 18.3. **Wyniki**

```
Wartość ułamka to 1/3
```

Pola C++ (zmienne instancji) `numerator` i `denominator` są poprzedzone deklaracją `private`, która zwiększa stopień enkapsulacji danych, czyli uniemożliwia dostęp do tych pól spoza klasy.

Metoda `setNumerator` zadeklarowana jest następująco:

```
void Fraction::setNumerator (int num)
```

Nazwa metody poprzedzona jest zapisem `Fraction::`, co oznacza, że metoda ta należy do klasy `Fraction`.

Nowa instancja klasy `Fraction` jest tworzona tak samo jak zwykłe zmienne w języku C:

```
Fraction  myFract;
```

Licznik i mianownik ułamka są ustawiane odpowiednio na 1 i 3:

```
myFract.setNumerator (1);
myFract.setDenominator (3);
```

Następnie wartość ułamka jest wyświetlana za pomocą metody `print`.

Zapewne większości czytelników w programie 18.3 najdziwniejsza wydała się instrukcja stanowiąca treść metody `print`:

```
std::cout << "Wartość ułamka to " << numerator << '/'
          << denominator << '\n';
```

`cout` to nazwa standardowego strumienia wyjściowego, analogicznego do `stdout` w języku C. Znak `<<` to *operator wstawiania do strumienia*, który zapewnia łatwe wypisywanie danych. Przypomnijmy, że w C `<<` jest operatorem bitowego przesunięcia w lewo. Mamy tu do czynienia z ważnym aspektem programowania w języku C++ — operatory mogą być *przeciążane*, co wiąże się z przypisaniem tych operatorów do poszczególnych klas.

W tym wypadku przeciążony został operator przesunięcia w lewo tak, że kiedy jego lewym argumentem jest strumień, zamiast przesunięcia wywoływana jest metoda zapisująca sformatowane dane do strumienia wyjściowego.

Innym przykładem przeciążania operatorów jest użycie operatora + do dodawania do siebie ułamków:

```
myFract + myFract2
```

tak, że w przypadku powyższego wyrażenia wywołana zostanie stosowna metoda klasy `Fraction`.

Każde wyrażenie znajdujące się za operatorem << jest wyliczane i przekazywane do standardowego strumienia wyjściowego. W tym wypadku najpierw wypisywany jest literał `"Wartość ułamka to "`, potem licznik ułamka, znak `'/'`, mianownik ułamka i znak nowego wiersza.

Język C++ ma szereg interesujących cech. W dodatku E zainteresowani czytelnicy znajdą wskazówki, jak znaleźć dobry podręcznik tego języka.

Zauważmy jeszcze, że w powyższym programie zdefiniowaliśmy metody dostępowe klasy `Fraction` `Numerator()` i `Denominator()`, ale ich nie użyliśmy.

Klasa C# obsługująca ułamki

Ostatni przykład w tym rozdziale to program 18.4 pokazujący klasę obsługującą ułamki w języku C#. Kompilator języka C# wchodzi w skład pakietu Visual Studio i język ten jest jedną z kluczowych technologii programistycznych platformy .NET. Jeśli chcesz go wypróbować, na stronie *www.visualstudio.com/en-US/products/visual-studio-express-vs* znajdziesz darmową wersję środowiska Visual Studio.

Program 18.4. **Ułamki w języku C#**

```
using System;

class Fraction
{
    private int numerator;
    private int denominator;

    public int Numerator
    {
        get
        {
            return numerator;
        }

        set
        {
            numerator = value;
        }
    }

    public int Denominator
```

```
    {
        get
        {
            return denominator;
        }

        set
        {
            denominator = value;
        }
    }

    public void print ()
    {
        Console.WriteLine("Wartość ułamka to {0}/{1}",
            numerator, denominator);
    }
}

class example
{
    public static void Main()
    {
        Fraction myFract = new Fraction();

        myFract.Numerator = 1;
        myFract.Denominator = 3;

        myFract.print ();

    }
}
```

Program 18.4. **Wyniki**

```
Wartość ułamka to 1/3
```

Program w języku C# wygląda nieco inaczej niż poprzednie dwa programy, ale mimo to nietrudne powinno być domyślenie się, jak działa. Definicja klasy `Fraction` zaczyna się od deklaracji dwóch zmiennych instancji — `numerator` i `denominator` — z kwalifikatorem `private`. Metody `Numerator` i `Denominator` mają dwie metody — pobierającą i ustawiającą wartość w formie *właściwości*. Przyjrzyjmy się bliżej metodzie `Numerator`:

```
public int Numerator
{
    get
    {
        return numerator;
    }

    set
   {
       numerator = value;
    }
}
```

Kod get jest wykonywany, kiedy wartość pola numerator potrzebna jest w wyrażeniu, na przykład:

```
num = myFract.Numerator;
```

Kod set jest wykonywany w przypadku przypisywania metodzie wartości, na przykład:

```
myFract.Numerator = 1;
```

Przy wywołaniu metody przypisywana wartość jest wstawiana do zmiennej value. Zwróćmy uwagę na to, że za nazwami metod nie są używane nawiasy.

Można definiować metody mające parametry opcjonalne oraz metody ustawiające wartość mające różne zestawy parametrów. Na przykład w C# można zdefiniować metodę ustawiającą wartość ułamka od razu na 2/5:

```
myFract.setNumAndDen (2, 5)
```

Wróćmy jednak do programu 18.4. Instrukcja:

```
Fraction myFract = new Fraction();
```

służy do tworzenia nowej instancji klasy Fraction i przypisania wyniku zmiennej myFract typu Fraction. Następnie, korzystając z metod dostępowych, ustawiamy Fraction na 1/3.

Następnie wywołujemy metodę print, aby pokazać wartość naszego ułamka. W metodzie print korzystamy z metody WriteLine klasy Console. Jest ona podobna do funkcji printf, ale do umieszczania w łańcuchu wartości zamiast zapisu z symbolem procentu wykorzystujemy zapis {0}, {1} i tak dalej. W przeciwieństwie do funkcji printf, nie musimy troszczyć się o typy wyświetlanych wartości.

Tak jak w przypadku języka C++, metody pobierające wartości pól ułamka zostały zdefiniowane, ale ich nie użyliśmy.

Na tym kończymy to krótkie wprowadzenie do programowania obiektowego. Mamy nadzieję, że rozdział ten pozwolił zorientować się, na czym programowanie obiektowe polega i czym się różni od programowania w językach takich jak C. Pokazaliśmy trzy programy obsługujące ułamki napisane w trzech różnych językach obiektowych. Osoby naprawdę potrzebujące ułamków muszą oczywiście pokazane klasy znacznie rozszerzyć, aby obsłużyć przynajmniej dodawanie, odejmowanie, mnożenie, dzielenie, odwrotności i redukowanie ułamków. Jest to dość proste.

Następnym krokiem jest skorzystanie z dobrego podręcznika wybranego języka obiektowego; pewne sugestie można znaleźć w dodatku E.

A

Język C w skrócie

W tym dodatku podsumowano język C w takim układzie, aby łatwo było znaleźć potrzebne informacje. Nie jest to pełna definicja języka, ale nieformalny opis jego elementów. Po zapoznaniu się z całą książką należy uważnie przeczytać ten dodatek. W ten sposób nie tylko utrwalimy sobie zdobytą wiedzę, lecz także lepiej zrozumiemy filozofię języka C.

Dodatek ten oparto na standardzie ANSI C11 (ISO/IEC 9899:2011).

1.0. Dwuznaki i identyfikatory

1.1. Dwuznaki

W tabeli A.1 zestawiono specjalne sekwencje par znaków (dwuznaków), które są równoważne podanym znakom pojedynczym.

Tabela A.1. Dwuznaki i ich odpowiedniki

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

1.2. Identyfikatory

W języku C *identyfikator* to ciąg liter (wielkich i małych), *nazw znaków uniwersalnych* (zobacz punkt 1.2.1), cyfr i podkreśleń. Pierwszy znak identyfikatora musi być literą, podkreśleniem lub nazwą znaku uniwersalnego. Pierwszych 31 znaków identyfikatora na pewno będzie znaczące w przypadku identyfikatora zewnętrznego, a pierwsze 63 w przypadku identyfikatora wewnętrznego lub nazwy makra.

1.2.1. Nazwy znaków uniwersalnych

Nazwa znaku uniwersalnego składa się ze znaków `\u`, za którymi następują cztery cyfry szesnastkowe, albo znaków `\U`, za którymi występuje osiem cyfr szesnastkowych. Jeśli pierwszy znak identyfikatora jest znakiem uniwersalnym, nie może to być cyfra. Znaki uniwersalne używane w identyfikatorach nie mogą także zawierać znaków mniejszych od $A0_{16}$ (poza 24_{16}, 40_{16} i 60_{16}) ani znaków z zakresu od $D800_{16}$ do $DFFF_{16}$ włącznie.

Nazwy znaków uniwersalnych mogą występować w nazwach identyfikatorów, stałych znakowych oraz w łańcuchach znakowych.

1.2.2. Słowa kluczowe

Identyfikatory zestawione w tabeli A.2 to słowa kluczowe, mające specjalne znaczenie dla kompilatora C.

Tabela A.2. Słowa kluczowe

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

2.0. Komentarze

Komentarze do programu można wstawiać na dwa sposoby. Komentarz może zaczynać się od dwóch ukośników — //. Wszystkie znaki znajdujące się dalej, aż do końca wiersza, są ignorowane przez kompilator.

Komentarz może też zaczynać się od /* i kończyć */. W komentarzu takim mogą wystąpić dowolne znaki, tego typu komentarz może zajmować wiele wierszy.

Komentarze mogą występować w programie wszędzie tam, gdzie mogą występować białe znaki. Jednak komentarzy nie można zagnieżdżać, wobec tego pierwsze znaki */ kończą komentarz, niezależnie od tego, ilu /* użyto wcześniej.

3.0. Stałe

3.1. Stałe całkowitoliczbowe

Stała całkowitoliczbowa to ciąg cyfr, ewentualnie poprzedzony znakiem plus lub minus. Jeśli pierwszą cyfrą jest `0`, jest to liczba ósemkowa; wtedy wszystkie cyfry muszą mieścić się w zakresie od `0` do `7`. Jeśli pierwsza cyfra to `0`, a zaraz za nią jest znak `x` lub `X`, liczba jest stałą szesnastkową, a jej cyframi są zwykłe cyfry od `0` do `9` oraz litery od `a` do `f` (lub od `A` do `F`).

Na koniec stałej liczby całkowitej można dodać literę `l` lub `L`, aby była to stała typu `long int`. Jeśli wartość nie mieści się w typie `long int`, traktowana jest jako `long long int`. W końcu, jeśli i to jest za mało, liczba jest traktowana jako `unsigned long long int`. Tak samo, przez dodanie na końcu `l` lub `L`, zwiększa się zakres liczb zapisanych ósemkowo lub szesnastkowo.

Aby liczba od razu była traktowana jako wartość typu `long long int`, na koniec dodaje się litery `ll` lub `LL`.

Aby stała była traktowana jako liczba bez znaku — `unsigned` — na koniec dodaje się literę `u` lub `U`. Jeśli stała jest zbyt duża, aby zmieścić się w typie `unsigned int`, jest traktowana jako `unsigned long int`. Jeśli i na to jest za duża, traktowana jest jako `unsigned long long int`. Do stałej można dodać przyrostek oznaczający `unsigned` i jednocześnie przyrostek `long long`, aby uzyskać wartość `unsigned long long int`.

Jeśli stała liczba całkowita zapisana dziesiętnie jest zbyt duża, aby zmieścić się w typie `signed int`, jest traktowana jako `long int`. Jeśli jest i na to zbyt duża, traktowana jest jako `long long int`.

Jeśli stała całkowita zapisana ósemkowo lub szesnastkowo jest zbyt duża, aby zmieścić się w typie `signed int`, traktowana jest jako `unsigned int`. Jeśli także w tym typie nie mieści się, traktowana jest jako `unsigned long int`. Jeśli jest i na to zbyt duża, traktowana jest jako wartość `long long int`. W końcu, jeśli wszystkie dotychczasowe konwersje zawiodły, jest traktowana jako `unsigned long long int`.

3.2. Stałe zmiennoprzecinkowe

Stała zmiennoprzecinkowa składa się z ciągu cyfr, kropki dziesiętnej i drugiego ciągu cyfr. Wartości ujemne mogą być poprzedzone minusem. Każdy z tych ciągów cyfr może zostać pominięty, ale nie oba jednocześnie.

Jeśli za stałą zmiennoprzecinkową znajduje się litera `e` lub `E`, a za nią liczba całkowita (ewentualnie ze znakiem), to mamy do czynienia z notacją naukową. Owa liczba całkowita to *wykładnik* oznaczający potęgę 10, przez jaką należy pomnożyć liczbę poprzedzającą wykładnik (ta liczba to *mantysa*), na przykład `1.5e–2` oznacza $1.5 \cdot 10^{-2}$, czyli 0.015.

Zmiennoprzecinkowa stała *szesnastkowa* składa się z wiodącego `0x` lub `0X`, co najmniej jednaj cyfry szesnastkowej, litery `p` lub `P` i opcjonalnie wykładnika binarnego ze znakiem, na przykład `0x3p10` to $3 \cdot 2^{10}$.

Stałe zmiennoprzecinkowe są traktowane przez kompilator jako wartości typu `double`. Aby stała była typu `float`, należy za nią dodać `f` lub `F`. Aby była typu `long double`, należy dodać `l` lub `L`.

3.3. Stałe znakowe

Znaki ujęte w apostrofy to stałe znakowe. To, jak będzie traktowane umieszczenie w apostrofach większej liczby znaków, zależy od implementacji. Jeśli potrzebny znak jest spoza zestawu standardowego, można go zapisać jako znak uniwersalny (punkt 1.2.1).

3.3.1. Cytowanie znaków

W języku C rozpoznawane są znaki specjalne, cytowane, zapisywane przez poprzedzenie odpowiedniego kodu odwrotnym ukośnikiem. Znaki takie zestawiono w tabeli A.3.

Tabela A.3. Znaki specjalne

Znak	Znaczenie
`\a`	Uruchomienie głośniczka
`\b`	Backspace
`\f`	Nowa strona
`\n`	Nowy wiersz
`\r`	Powrót karetki
`\t`	Tabulator poziomy
`\v`	Tabulator pionowy
`\\`	Odwrotny ukośnik
`\"`	Cudzysłów
`\'`	Apostrof
`\?`	Pytajnik
`\`*nnn*	Znak zapisany ósemkowo
`\u`*nnnn*	Nazwa znaku uniwersalnego
`\U`*nnnnnnnn*	Nazwa znaku uniwersalnego
`\x`*nn*	Znak zapisany szesnastkowo

W przypadku znaków zapisanych ósemkowo można podać od jednej do trzech cyfr ósemkowych. W ostatnich trzech przypadkach używa się cyfr szesnastkowych.

3.3.2. Stałe szerokie znaki

Stały szeroki znak zapisujemy, korzystając z notacji `L'x'`. Stała taka jest typu `wchar_t`, który to typ został zdefiniowany w standardowym pliku nagłówkowym `<stddef.h>`. Stałe szerokie znaki umożliwiają zapisywanie znaków z zestawów, które nie mieszczą się w normalnym typie `char`.

3.4. Stałe łańcuchy znakowe

Ciąg zawierający zero lub więcej znaków ujętych w cudzysłów to stały łańcuch znakowy. W łańcuchu takim można umieszczać dowolne poprawne znaki, w tym podane wcześniej znaki specjalne. Kompilator na koniec takiego łańcucha automatycznie wstawi znak null (`'\0'`).

Normalnie kompilator tworzy wskaźnik typu „wskaźnik `char`" do pierwszego znaku łańcucha. Jeśli jednak stała łańcuchowa zostanie użyta w operatorze `sizeof` do inicjalizacji tablicy znakowej lub przy operatorze `&`, typem łańcucha jest „tablica elementów `char`".

Stałe łańcuchy znakowe nie mogą być modyfikowane w programie.

3.4.1. Łączenie łańcuchów znakowych

Preprocesor automatycznie łączy sąsiadujące ze sobą łańcuchy znakowe. Łańcuchy te mogą być rozdzielane białymi znakami. Wobec tego poniższe trzy łańcuchy:

```
"to" " jest "
     "łańcuch"
```

są równoważne jednemu łańcuchowi:

```
"to jest łańcuch"
```

3.4.2. Znaki wielobajtowe

Łańcuchy znakowe mogą być *przesuwane* zależnymi od implementacji ciągami znaków tak, aby możliwe było wstawianie do nich znaków wielobajtowych.

3.4.3. Stałe łańcuchy szerokich znaków

Stałe łańcuchy znakowe z rozszerzonego zestawu znaków zapisujemy, korzystając z notacji `L"..."`. Typem takiej stałej jest „wskaźnik `wchar_t`", gdzie typ `wchar_t` zdefiniowany jest w `<stddef.h>`.

3.5. Stałe wyliczeniowe

Identyfikator zadeklarowany jako wartość typu wyliczeniowego jest traktowany jak stała tego typu; w innych sytuacjach jest traktowany przez kompilator jako dana typu `int`.

4.0. Typy danych i deklaracje

W tej części omówimy podstawowe typy danych, pochodne typy danych, wyliczeniowe typy danych oraz instrukcję `typedef`. Zbierzemy też informacje o sposobie deklarowania zmiennych.

4.1. Deklaracje

Podczas definiowania struktury, unii, typu wyliczeniowego lub typu w instrukcji `typedef`, kompilator automatycznie nie alokuje żadnej pamięci. Definicja po prostu przekazuje do kompilatora, jak wygląda typ danych i (opcjonalnie) nadaje takiemu typowi nazwę. Definicje takie mogą występować w funkcjach lub poza nimi. W pierwszym wypadku typ jest znany tylko w funkcji; w drugim jest znany w całej reszcie pliku.

Po zdefiniowaniu typu można deklarować zmienne tego typu. Zmienna zadeklarowana w dowolnym typie ma przydzieloną pamięć — z wyjątkiem deklaracji `extern`; wtedy alokacja pamięci może nastąpić, ale nie zawsze ma to miejsce (zobacz podrozdział 6.0).

Język C umożliwia alokowanie pamięci na wartości danego typu w chwili definiowania tego typu. Wykonujemy to, po prostu podając zmienne przed średnikiem kończącym definicję.

4.2. Podstawowe typy danych

Podstawowe typy danych języka C zestawiono w tabeli A.4. Zmienną w którymś z tych typów deklaruje się następująco:

```
typ  nazwa = wartość_początkowa;
```

Przypisanie zmiennej wartości początkowej jest opcjonalne, omawiamy je w podrozdziale 6.2. Naraz możemy zdefiniować więcej niż jedną zmienną — korzystamy wtedy ze składni:

```
typ  nazwa = wartość_początkowa, nazwa = wartość_początkowa, ...;
```

Przed deklaracją typu można też podać sposób umieszczania zmiennej w pamięci (klasę). Jeśli podana zostanie klasa zmiennej, a zmienna jest typu `int`, słowo `int` można pominąć, na przykład instrukcja:

```
static  counter;
```

deklaruje zmienną `counter` jako `static int`.

Modyfikator `signed` może być używany także przed typami `short int`, `int`, `long int` i `long long int`. Typy te domyślnie mają znak, więc użycie tego modyfikatora niczego nie wnosi.

Tabela A.4. Zestawienie podstawowych typów danych

Typ	Znaczenie
`int`	Liczba całkowita, czyli bez części ułamkowej. Musi mieć co najmniej 16 bitów.
`short int`	Liczba całkowita o ograniczonej długości. Pochłania połowę pamięci zajmowanej na niektórych maszynach przez typ `int`, ma przynajmniej 16 bitów.
`long int`	Liczba całkowita o zwiększonej długości. Musi mieć przynajmniej 32 bity.
`long long int`	Liczba całkowita o znacznie zwiększonej długości. Musi mieć przynajmniej 64 bity.
`unsigned int`	Dodatnia liczba całkowita, dwukrotnie większa od liczb `int`. Musi mieć przynajmniej 16 bitów.
`float`	Liczba zmiennoprzecinkowa, czyli z częścią ułamkową. Musi mieć dokładność przynajmniej sześciu cyfr.
`double`	Liczba zmiennoprzecinkowa o zwiększonej dokładności. Musi mieć przynajmniej 10 cyfr znaczących.
`long double`	Liczba zmiennoprzecinkowa o znacząco zwiększonej dokładności. Musi mieć przynajmniej 10 cyfr znaczących.
`char`	Pojedynczy znak. W niektórych systemach w wyrażeniach może być rozszerzany i mieć znak.
`unsigned char`	Typ analogiczny z `char`, ale zapewnia, że w wyniku promocji do typu `int` nie pojawi się znak.
`signed char`	Typ analogiczny z `char`, ale gwarantuje, że w przypadku promocji do typu `int` zostanie dołączony znak.
`_Bool`	Typ logiczny, wystarcza do zapisania wartości 0 i 1.
`float _Complex`	Liczba zespolona.
`double _Complex`	Liczba zespolona o zwiększonej dokładności.
`long double _Complex`	Liczba zespolona o znacząco zwiększonej dokładności.
`void`	Brak typu danych. Używany, aby zapewnić, że funkcja nie zwróci wartości, albo do jawnego „odrzucania" wyniku wyrażenia. Używany też przy deklarowaniu ogólnego typu wskaźnikowego — `void *`.

Typy danych `_Complex` i `_Imaginary` umożliwiają deklarowanie liczb zespolonych i urojonych oraz wykonywanie na nich obliczeń. Do obliczeń służą funkcje z biblioteki obsługującej arytmetykę zespoloną. Normalnie do programu trzeba włączać plik `<complex.h>`, który zawiera definicje makr i deklaracje funkcji obsługujących liczby zespolone i urojone.

Na przykład zmienną `c1` typu `double _Complex` można zadeklarować i zainicjalizować wartością `5+10.5i` za pomocą instrukcji

```
double _Complex  c1 = 5 + 10.5 * I;
```

Funkcje biblioteczne `creal` i `cimag` pozwalają pobrać odpowiednio część rzeczywistą i urojoną zmiennej `c1`.

Nie wymaga się implementacji obsługi typów `_Complex` i `_Imaginary`; dopuszczalna jest także implementacja tylko jednego z tych typów.

Do programu można dołączyć plik nagłówkowy `<stdbool.h>`, aby ułatwić pracę ze zmiennymi logicznymi. W pliku tym zdefiniowano makra `bool`, `true` i `false`, dzięki czemu można stosować instrukcje typu:

```
bool  koniecDanych = false;
```

4.3. Pochodne typy danych

Pochodny typ danych to taki, który wykorzystuje jeden lub więcej podstawowych typów danych. Typami pochodnymi są tablice, struktury, unie oraz wskaźniki. Funkcja zwracająca wartość ustalonego typu też jest uważana za pochodny typ danych. Wszystkie typy pochodne, z wyjątkiem funkcji, omówimy w kolejnych podrozdziałach. Funkcje zostaną omówione osobno, w podrozdziale 7.0.

4.3.1. Tablice

4.3.1.1. Tablice jednowymiarowe

Tablice mogą zawierać elementy dowolnego typu podstawowego lub typu pochodnego. Nie można deklarować tablic funkcji (choć istnieją tablice wskaźników funkcji).

Deklaracja tablicy ma następującą postać:

```
typ  nazwa[n]   = { wyrażenieInicjujące, wyrażenieInicjujące, ... };
```

Wyrażenie *n* określa liczbę elementów tablicy *nazwa*; może ono zostać pominięte — wtedy wielkość tablicy określa się na podstawie liczby wartości inicjalizujących lub na podstawie największego indeksu użytego w *inicjalizatorach oznaczonych*.

Każda wartość inicjalizująca dla tablic musi być globalnym wyrażeniem stałym. Wartości inicjalizujących może być mniej niż elementów tablicy, ale nie może ich być więcej. Jeśli podanych zostanie mniej wartości, zainicjalizowane są tylko wskazane elementy tablicy. Pozostałe elementy uzyskują wartość `0`.

W zakresie inicjalizacji szczególnymi tablicami są tablice znakowe, które zawsze można inicjalizować stałym łańcuchem znakowym; na przykład instrukcja:

```
char today[] = "poniedziałek";
```

deklaruje tablicę znakową inicjalizowaną znakami `'p'`, `'o'`, `'n'`, `'i'`, `'e'`, `'d'`, `'z'`, `'i'`, `'a'`, `'ł'`, `'e'`, `'k'` i `'\0'`.

Jeśli jawnie zostanie podana wielkość tablicy znakowej oraz tablica ta zostanie zainicjalizowana, ale będzie za krótka i nie zmieści się w niej znak null, kompilator go nie wstawi:

```
char today[6] = "poniedziałek";
```

Zadeklarowana zostanie tablica znakowa `today` mająca miejsce na sześć znaków, która następnie będzie zainicjalizowana wartościami `'p'`, `'o'`, `'n'`, `'i'`, `'e'`, `'d'`, `'z'`, `'i'`, `'a'`, `'ł'`, `'e'` i `'k'`.

Zamykając numer elementu w parze nawiasów klamrowych, możemy wykonywać inicjalizacje w dowolnej kolejności, na przykład instrukcje:

```
int    x = 1233;
int    a[] = { [9] = x + 1, [3] = 3, [2] = 2, [1] = 1 };
```

definiują 10-elementową tablicę `a` (dziesięcioelementową, gdyż największy użyty indeks to 9). Tablica ta jest inicjalizowana tak, że jej ostatni element otrzymuje wartość `x + 1` (1234), a pierwsze trzy odpowiednio wartości 1, 2 i 3.

4.3.1.2. Tablice o zmiennej długości

W funkcji lub bloku instrukcji możemy wymiarować tablicę, korzystając ze zmiennych. Wielkość tablicy jest wtedy wyliczana w trakcie wykonywania programu; na przykład w funkcji:

```
int makeVals (int n)
{
    int valArray[n];
    ...
}
```

definiowana jest tablica automatyczna `valArray` mająca `n` elementów, przy czym `n` jest określane w trakcie wykonywania programu i może być różne w różnych wywołaniach funkcji. Tablic o zmiennej długości nie można inicjalizować w chwili ich deklarowania.

4.3.1.3. Tablice wielowymiarowe

Tablice wielowymiarowe deklaruje się przy użyciu składni

typ nazwa`[`*d1*`][`*d2*`]...[`*dn*`] = `*listaInicjalizująca*`;`

Tablica *nazwa* zawiera *d1*·*d2*·...·*dn* elementów typu *typ*, na przykład instrukcja:

```
int  three_d [5][2][20];
```

definiuje trójwymiarową tablicę `three_d` zawierającą łącznie 200 liczb całkowitych.

Do poszczególnych elementów tablic wielowymiarowych odwołujemy się, podając kolejne indeksy w osobnej parze nawiasów kwadratowych; na przykład instrukcja:

```
three_d [4][0][15] = 100;
```

powoduje zapisanie w podanym elemencie tablicy `three_d` wartości 100.

Tablice wielowymiarowe można inicjalizować tak samo jak tablice jednowymiarowe. Do zachowania właściwej kolejności przypisań można użyć zagnieżdżonych par nawiasów klamrowych.

Poniżej deklarujemy dwuwymiarową tablicę `matrix` mającą cztery wiersze i trzy kolumny:

```
int matrix[4][3] =
        { { 1, 2, 3 },
          { 4, 5, 6 },
          { 7, 8, 9 } };
```

Elementy z pierwszego wiersza tablicy `matrix` otrzymują wartości 1, 2 i 3. Elementy drugiego wiersza otrzymują wartości 4, 5 i 6, a trzeciego — 7, 8 i 9. Elementy czwartego wiersza są inicjalizowane zerami, gdyż brak dla nich wartości inicjalizujących. Deklaracja:

```
static int matrix[4][3] =
          { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

inicjalizuje tablicę `matrix` takimi samymi wartościami, gdyż elementy tablicy wielowymiarowej są inicjalizowane kolejnymi wymiarami, od wymiaru lewego do prawego.

W deklaracji:

```
int matrix[4][3] =
         { { 1 },
           { 4 },
           { 7 } };
```

ustawiamy pierwszy element pierwszego wiersza tablicy `matrix` na 1, pierwszy element drugiego wiersza na 4, a pierwszy element trzeciego wiersza na 7. Wszystkie pozostałe elementy są zerowane.

W końcu deklaracja:

```
int matrix[4][3] = {  [0][0] = 1, [1][1] = 5, [2][2] = 9 };
```

inicjalizuje podane elementy wskazanymi wartościami.

4.3.2. Struktury

Ogólny format deklaracji struktury jest następujący:

```
struct nazwa
{
    deklaracjaPola
    deklaracjaPola
      ...
} listaZmiennych;
```

Struktura *nazwa* zawiera pola zgodne z poszczególnymi *deklaracjamiPola*. Każda taka deklaracja składa się z określenia typu i nazwy jednego pola lub kilku pól. W definicji struktury można deklarować zmienne; wystarczy je wyliczyć przed średnikiem kończącym definicję; można też użyć osobnej deklaracji:

```
struct nazwa  listaZmiennych;
```

Z takiego zapisu nie można skorzystać, jeśli w definicji struktury pominięta zostanie *nazwa*. Wtedy deklaracje wszystkich zmiennych muszą wystąpić w definicji struktury.

Inicjalizowanie zmiennych strukturalnych wygląda podobnie jak inicjalizowanie tablic. Poszczególne pola inicjalizujemy, podając ich wartości w parach nawiasów klamrowych. W przypadku inicjalizowania struktur globalnych każda wartość z listy musi być wyrażeniem stałym.

Deklaracja:

```
struct  punkt
{
    float x;
    float y;
}  start = {100.0, 200.0};
```

definiuje strukturę `punkt` oraz zmienną `start` typu `struct point`, która jest inicjalizowana. Można inicjalizować tylko wybrane pola:

```
.pole = wartość;
```

na przykład:

```
struct punkt  end = { .y = 500, .x = 200 };
```

Deklaracja:

```
struct  entry
{
    char  *word;
    char  *def;
} dictionary[1000] = {
    { "a",         "pierwsza litera alfabetu" },
    { "abdykacja", "zrzeczenie się tronu"     }
    { "abisal",    "najgłębsza strefa morza"  }
};
```

deklaruje słownik `dictionary` zawierający 1000 struktur `entry`, przy czym pierwsze trzy z nich są inicjalizowane łańcuchami znakowymi. Korzystając z inicjalizatorów oznaczonych, moglibyśmy to samo zapisać jako:

```
struct  entry
{
    char  *word;
    char  *def;
} dictionary[1000] = {
    [0].word = "a",         [0].def = "pierwsza litera alfabetu",
    [1].word = "abdykacja", [1].def = "zrzeczenie się tronu",
    [2].word =  "abisal",   [2].def = "najgłębsza strefa morza"
};
```

lub jako:

```
struct  entry
{
    char  *word;
    char  *def;
} dictionary[1000] =
```

```
   { {.word = "a",         .def = "pierwsza litera alfabetu"},
     {.word = "abdykacja", .def = "zrzeczenie się tronu"},
     {.word =  "abisal",   .def = "najgłębsza strefa morza"}
};
```

Automatyczne zmienne strukturalne można inicjalizować innymi strukturami tego samego typu:

```
struct date  tomorrow = today;
```

W ten sposób deklarujemy zmienną strukturalną `tomorrow` typu `date`, której przypisujemy treść zadeklarowanej wcześniej struktury `today` typu `date`.

`deklaracjaPola` ma postać:

```
typ nazwaPola : n
```

i deklaruje pole *`nazwaPola`* mające *n* bitów w strukturze, gdzie *n* jest liczbą całkowitą. Poszczególne pola mogą być pakowane na jednych maszynach od lewej strony do prawej, a na innych — od prawej do lewej. Jeśli pominięta zostanie *`nazwaPola`*, a *n* jest równe 0, pole takie służy do wyrównania pozostałych pól do brzegu *jednostki alokacji pamięci*, przy czym ta *jednostka* jest zależna od implementacji. Typem pola może być `_Bool`, `int`, `signed int` lub `unsigned int`. Od implementacji zależy, czy pole typu `int` będzie traktowane jako `signed int` czy `unsigned int`. Do pól nie można stosować operatora adresu (`&`), z takimi polami nie można też tworzyć tablic.

4.3.3. Unie

Ogólny format deklaracji unii jest następujący:

```
union nazwa
{
    deklaracjaPola
    deklaracjaPola
      ...
} listaZmiennych;
```

Definiujemy tutaj unię *`nazwa`* z polami opisanymi przez *`deklaracjePola`*. Każde pole unii znajduje się w tym samym miejscu w pamięci, kompilator zaś dba o zarezerwowanie takiej ilości miejsca, aby starczyło na największe z tych pól.

Wraz z definicją unii można od razu deklarować zmienne, można też deklarować je później za pomocą zapisu:

```
union nazwa  listaZmiennych;
```

pod warunkiem że unii nadano nazwę.

Na programiście spoczywa odpowiedzialność za pobieranie z unii wartości takiego typu, jakiego typu wartość ostatnio zapisano. Możemy zainicjalizować *pierwsze* pole unii, podając jego wartość w nawiasach klamrowych; w przypadku zmiennych globalnych wartość musi być wyrażeniem stałym:

```
union  shared
{
    long long int  l;
```

```
    long int       w[2];
} swap = { 0xffffffff };
```

Deklarowana jest zmienna `swap` będąca unią, jej pole `l` ustawiane jest na wartość szesnastkową `ffffffff`. Aby zainicjalizować inne pole, trzeba podać jego nazwę:

```
union shared swap = {.w[0] = 0x0, .w[1] = 0xffffffff; }
```

Automatyczne zmienne będące uniami można inicjalizować uniami takiego samego typu:

```
union shared swap2 = swap;
```

4.3.4. Wskaźniki

Podstawowy sposób deklarowania wskaźników to:

```
typ  *nazwa;
```

Identyfikator *name* to „wskaźnik *typu*”, przy czym *typ* może być typem podstawowym lub pochodnym, na przykład:

```
int  *ip;
```

to deklaracja zmiennej `ip` będącej wskaźnikiem na wartość `int`, z kolei deklaracja:

```
struct entry  *ep;
```

to deklaracja zmiennej `ep` będącej wskaźnikiem na strukturę `entry`.

Wskaźniki wskazujące elementy tablicy deklaruje się jako wskaźniki takiego typu, jakiego typu są elementy tablicy; na przykład poprzednia deklaracja `ip` może być też użyta do zadeklarowania wskaźnika elementów tablicy zawierającej elementy `int`.

Można też deklarować bardziej złożone wskaźniki, na przykład w deklaracji:

```
char *tp[100];
```

zmienna `tp` jest tablicą 100 wskaźników znaków, natomiast w deklaracji:

```
struct entry (*fnPtr) (int);
```

zmienna `fnPtr` to wskaźnik funkcji zwracającej strukturę `entry` i mającej jeden parametr typu `int`.

Możemy sprawdzać, czy wskaźnik nie jest pusty, porównując go ze stałym wyrażeniem o wartości 0. Twórcy poszczególnych implementacji decydują o sposobie wewnętrznej reprezentacji wskaźników pustych o wartościach innych niż 0. Jednak nawet wtedy porównanie takiego niezerowego wskaźnika pustego i zera musi dać wynik pozytywny.

Sposób konwersji wskaźników na liczby całkowite i odwrotnie zależy od użytej maszyny, tak samo jak wielkość liczb potrzebnych do zapisania wskaźnika.

Typ „wskaźnik typu `void`” to ogólny typ wskaźnikowy. Język gwarantuje, że wskaźnik dowolnego typu może być przypisany do wskaźnika `void` i z powrotem bez zmiany wartości.

Poza tym przypadkiem szczególnym nie można przypisywać sobie nawzajem wskaźników różnych typów; zwykle powoduje to wygenerowanie ostrzeżenia przez kompilator.

4.4. Wyliczeniowe typy danych

Ogólny format deklarowania wyliczeniowych typów danych jest następujący:

```
enum nazwa { enum1, enum2, ... } listaZmiennych;
```

Typ wyliczeniowy *nazwa* definiuje się jako typ wyliczeniowy zawierający wartości *enum1*, *enum2* i tak dalej; każdy element jest identyfikatorem lub identyfikatorem ze znakiem plus i wyrażeniem stałym. Z kolei *listaZmiennych* jest opcjonalną listą zmiennych (z opcjonalnymi wartościami inicjalizującymi) deklarowanych jak zmienne typu `enum` *nazwa*.

Kompilator poszczególnym identyfikatorom przypisuje kolejne liczby całkowite, zaczynając od zera. Jeśli za identyfikatorem znajduje się znak równości oraz wyrażenie stałe, identyfikatorowi przypisywana jest podana wartość. Kolejnym identyfikatorom przypisujemy wartości, poczynając od wyrażenia stałego plus 1. Identyfikatory wyliczeniowe są przez kompilator traktowane jako stałe liczby całkowite.

Jeśli przydatne byłoby zadeklarowanie zmiennej wcześniej zdefiniowanego i nazwanego typu wyliczeniowego, może zostać użyta konstrukcja:

```
enum  nazwa  listaZmiennych;
```

Zmienna zadeklarowana jako zmienna typu wyliczeniowego może mieć tylko wartości tego typu wyliczeniowego, choć innej sytuacji kompilator może nie traktować jako błąd.

4.5. Instrukcja typedef

Instrukcja `typedef` służy do przypisania nowej nazwy podstawowemu lub pochodnemu typowi danych. Instrukcja ta nie definiuje nowego typu, ale po prostu nadaje nową nazwę istniejącemu typowi. Wobec tego później zmiennym zadeklarowanym z nową nazwą typu można przypisywać wartości tak, jak wszystkim innym zmiennym typu bazowego.

Tworząc definicję `typedef`, postępujemy tak jak przy zwykłej deklaracji zmiennej. Następnie wstawiamy nazwę nowego typu tam, gdzie normalnie umieszczana jest nazwa zmiennej. W końcu, przed całą definicją umieszczamy słowo kluczowe `typedef`.

W poniższym przykładzie:

```
typedef struct
    {
        float  x;
        float  y;
    } Punkt;
```

strukturze mającej dwa pola zmiennoprzecinkowe — `x` i `y` — nadajemy nazwę `Punkt`. Później możemy już normalnie deklarować zmienne typu `Punkt`:

```
Punkt  origin = { 0.0, 0.0 };
```

4.6. Modyfikatory typu const, volatile i restrict

Słowo kluczowe `const` umieszczone przed deklaracją typu oznacza dla kompilatora, że dana wartość nie może być modyfikowana. Zatem deklaracja:

```
const  int x5 = 100;
```

to deklaracja zmiennej x5 typu int mającej stałą wartość 100. Zmiennej takiej w czasie działania programu nie będzie można przypisywać żadnej innej wartości. Kompilator *nie* musi jednak ostrzegać o próbie zmiany wartości zmiennej zadeklarowanej jako const.

Modyfikator volatile informuje kompilator, że wartość zmienia się, zwykle dynamicznie. Jeśli zmienna volatile zostanie użyta w wyrażeniu, jej wartość jest odczytywana w każdym wystąpieniu z osobna.

Aby zadeklarować zmienną port17 jako wskaźnik char typu volatile, piszemy:

```
volatile char *port17;
```

Słowo kluczowe restrict dotyczy tylko wskaźników. Stanowi ono podpowiedź dotyczącą optymalizacji (podobnie jak słowo kluczowe register w przypadku zmiennych). Dla kompilatora słowo to oznacza, że dany wskaźnik jest jedynym odwołującym się do pewnego obiektu, czyli że w bieżącym zakresie nie istnieje żaden inny wskaźnik wskazujący ten sam obiekt. W poniższych wierszach:

```
int  * restrict intPtrA;
int  * restrict intPtrB;
```

informujemy kompilator, że w zakresie obowiązywania intPtrA i intPtrB nigdy nie wskażą tej samej wartości. Jeśli będą wskazywały liczby całkowite (na przykład w tablicy), zawsze będą wskazywały inne liczby.

5.0. Wyrażenia

Nazwy zmiennych, nazwy funkcji, nazwy tablic, stałe, wywołania funkcji, wartości z tablic oraz wartości unii to wyrażenia. Jeśli do którejś z tych wartości zastosujemy operator jednoargumentowy, nadal będziemy mieli do czynienia z wyrażeniem. Tak samo będzie, kiedy połączymy dwa lub więcej tych wyrażeń operatorem binarnym lub operatorem trójargumentowym. W końcu wyrażenie ujęte w nawiasy także jest wyrażeniem.

Wszystkim wyrażeniom dowolnego typu — poza void — odpowiada obiekt nazywany *l-wartością*, czyli lvalue. Jeśli takiej l-wartości można przypisać wartość, nazywamy ją *l-wartością modyfikowalną*.

W niektórych miejscach w programie mogą wystąpić tylko l-wartości modyfikowalne. Wyrażenie po lewej stronie operatora przypisania zawsze musi być l-wartością modyfikowalną. Co więcej, operatory inkrementacji i dekrementacji mogą działać tylko na l-wartościach modyfikowalnych, tak samo jak jednoargumentowy operator adresu (&) — w tym wypadku wyjątkiem jest funkcja.

5.1. Zestawienie operatorów języka C

W tabeli A.5 zestawiono operatory języka C. Uporządkowano je według malejących priorytetów. Operatory pogrupowano w zestawy o tym samym priorytecie.

Tabela A.5. Zestawienie operatorów języka C

Operator	Opis	Łączność
`()`	Wywołanie funkcji	
`[]`	Odwołanie do elementu tablicy	
`->`	Odwołanie do pola wskaźnika struktury	z lewej do prawej
`.`	Odwołanie do pola struktury	
–	Minus jednoargumentowy	
+	Plus jednoargumentowy	
++	Inkrementacja	
—	Dekrementacja	
!	Negacja logiczna	
~	Uzupełnienie do jedności	z prawej do lewej
*	Odwołanie do wskaźnika (wyłuskanie)	
&	Adres	
`sizeof`	Wielkość obiektu	
(*typ*)	Rzutowanie typu (konwersja)	
*	Mnożenie	
/	Dzielenie	z lewej do prawej
%	Modulo	
+	Dodawanie	z lewej do prawej
–	Odejmowanie	
<<	Przesunięcie w lewo	z lewej do prawej
>>	Przesunięcie w prawo	
<	Mniejsze	
<=	Mniejsze lub równe	z lewej do prawej
>	Większe	
>=	Większe lub równe	
==	Równe	z lewej do prawej
!=	Nierówne	
&	Bitowe AND	z lewej do prawej
^	Bitowe XOR	z lewej do prawej
|	Bitowe OR	z lewej do prawej
&&	Logiczne AND	z lewej do prawej

Tabela A.5. Zestawienie operatorów języka C (ciąg dalszy)

Operator	Opis	Łączność
`||`	Logiczne OR	z lewej do prawej
`?:`	Operator wyboru	od prawej do lewej

`=`
`*= /= %=`
`+= -= &=`
`^= |=`
`<<= >>=`
	Operatory przypisania	od prawej do lewej

`,`	Operator przecinek	od prawej do lewej

Jako przykład rozważmy następujące wyrażenie:

```
b | c & d * e
```

Operator mnożenia ma wyższy priorytet niż bitowe OR i bitowe AND, gdyż w tabeli A.5 znajduje się wyżej. Analogicznie bitowy operator AND ma wyższy priorytet niż bitowy operator OR. Wobec tego powyższe wyrażenie zostanie zinterpretowane jako:

```
b | (c & ( d * e ) )
```

Teraz zajmijmy się wyrażeniem:

```
b % c * d
```

Modulo i mnożenie występują w tabeli A.5 w tej samej grupie, więc mają taki sam priorytet. Łączność ich jest od lewej do prawej, więc wyrażenie będzie interpretowane jako:

```
( b % c ) * d
```

Z kolei wyrażenie:

```
++a->b
```

zostanie zinterpretowane jako:

```
++(a->b)
```

gdyż operator -> ma wyższy priorytet niż ++.

W końcu, wobec faktu, że operatory przypisania są grupowane od prawej do lewej, instrukcja:

```
a = b = 0;
```

oznacza tak naprawdę:

```
a = (b = 0);
```

zatem ostatecznie zmienne a i b otrzymają wartość 0. Z kolei w przypadku wyrażenia:

```
x[i] + ++i
```

nie wiadomo, czy kompilator wyliczy najpierw wyrażenie po lewej, czy po prawej stronie operatora plus. W tym wypadku ta kolejność może zmienić uzyskany wynik, gdyż wartość `i` może zostać zwiększona przed wyliczeniem `x[i]`.

Innym przykładem wyrażenia, w którym nie została określona kolejność wykonywania obliczeń, jest:

```
x[i] = ++i
```

Tutaj nie wiadomo, czy zmienna `i` zostanie zwiększona przed, czy po odczytaniu wartości `x[i]`.

Nieokreślona jest także kolejność wyliczania wartości parametrów funkcji. Wobec tego w wywołaniu:

```
f (i, ++i);
```

wartość `i` może zostać najpierw zwiększona albo może zostać zwiększona w drugiej kolejności. Może zatem zdarzyć się, że do funkcji zostaną przekazane dwie takie same wartości albo dwie różne.

Język C gwarantuje, że operatory `&&` i `||` będą interpretowane od lewej do prawej strony. W przypadku `&&` język gwarantuje też, że jeśli pierwszy argument będzie zerem, drugi nie będzie interpretowany. Dla operatora `||` istnieje gwarancja, że jeśli pierwszy nie jest zerem, drugi nie będzie wyliczany. Warto o tym pamiętać przy tworzeniu wyrażeń:

```
if ( dataFlag  ||  checkData (myData) )
    ...
```

gdyż funkcja `checkData` zostanie wywołana tylko wtedy, gdy zmienna `dataFlag` będzie miała wartość `0`. Teraz następny przykład. Jeśli tablica `a` zawiera `n` elementów, instrukcja:

```
if (index >= 0  &&  index < n  &&  a[index] == 0))
    ...
```

odwoła się do elementu z tablicy tylko wtedy, gdy `index` jest poprawnym indeksem tablicy.

5.2. Wyrażenia stałe

Wyrażenie stałe to wyrażenie, w którym każdy wyraz jest wartością stałą. Wyrażeń stałych trzeba używać w następujących sytuacjach:

1. Jako wartości fraz `case` w instrukcji `switch`.
2. Do określania wielkości tablicy inicjalizowanej lub deklarowanej globalnie.
3. Do przypisywania wartości identyfikatorom typów wyliczeniowych.
4. Do określania wielkości pól bitowych w definicji struktury.
5. Do przypisywania wartości inicjalizujących zmiennym statycznym.
6. Do przypisywania wartości początkowych zmiennym globalnym.
7. W wyrażeniach w dyrektywach `#if` preprocesora.

W pierwszych czterech sytuacjach wyrażenie musi składać się ze stałych całkowitoliczbowych, stałych znakowych, stałych wyliczeniowych oraz wyrażeń `sizeof`. Jedyne dopuszczalne operatory to operatory arytmetyczne, bitowe, porównania, operator warunkowy oraz operator rzutowania. Operator `sizeof` nie może zostać użyty w wyrażeniach z tablicami o zmiennej długości, gdyż wtedy wynik zależy od sposobu zadziałania programu, zatem nie jest wyrażeniem stałym.

W przypadkach 5. i 6. można użyć dodatkowo jawnie lub niejawnie operatora adresu. Jednak można go użyć tylko do zmiennych i funkcji globalnych i statycznych. Jeśli na przykład `x` jest zmienną globalną lub statyczną, wyrażenie:

```
&x + 10
```

jest poprawnym wyrażeniem stałym. Wyrażenie:

```
&a[10] – 5
```

także jest wyrażeniem stałym, jeśli tylko `a` jest tablicą globalną lub statyczną. W końcu, stałe jest wyrażenie:

```
a + sizeof (char) * 100
```

gdyż `&a[0]` jest równoważne wyrażeniu `a`.

W ostatnim przypadku — po `#if` — obowiązują takie same zasady jak w pierwszych czterech, ale nie można użyć operatora `sizeof`, stałych wyliczeniowych ani rzutowania typów. Jednak można wykorzystać specjalny operator `defined` (zobacz podrozdział 9.2.3).

5.3. Operatory arytmetyczne

Jeśli:

`a, b`	są wyrażeniami dowolnego typu bazowego poza `void`;
`i, j`	są wyrażeniami całkowitoliczbowymi,

to wyrażenie:

`~a`	neguje wartość `a`;
`+a`	podaje wartość `a`;
`a + b`	dodaje `a` do `b`;
`a – b`	odejmuje `b` od `a`;
`a * b`	mnoży `a` przez `b`;
`a / b`	dzieli `a` przez `b`;
`i % j`	podaje resztę z dzielenia `i` przez `j`.

W każdym wyrażeniu wykonywane są zwykłe konwersje argumentów (zobacz podrozdział 5.17). Jeśli `a` jest `unsigned`, `~a` jest liczone najpierw przez promowanie tej wielkości do odpowiedniego typu całkowitego, potem odjęcie wyniku od największej możliwej wartości danego typu i w końcu dodanie jedności do wyniku.

Jeśli dzielimy dwie liczby całkowite, z wyniku odrzucana jest część ułamkowa. Jeśli któryś argument jest ujemny, kierunek obcięcia nie jest określony, zatem –3/2 może dać w wyniku na jednych maszynach –1, a na innych –2. W przypadku liczb dodatnich obcięcie wykonywane jest zawsze do zera, czyli 3/2 zawsze da 1. W podrozdziale 5.15 zestawiono działania na liczbach całkowitych i wskaźnikach.

5.4. Operatory logiczne

Jeśli:

`a, b` są wyrażeniami dowolnego typu bazowego poza `void` lub oba są wskaźnikami,

to wyrażenie:

`a && b` ma wartość 1, jeśli `a` i `b` oba są niezerowe; w przeciwnym razie wyrażenie jest zerem;

`a || b` ma wartość 1, jeśli choć jedno z wyrażeń `a` i `b` jest niezerowe; w przeciwnym razie wyrażenie jest zerem;

`! a` ma wartość 1, jeśli `a` jest zerem, i ma wartość 0 w przeciwnym wypadku;

Do `a` i `b` stosowane są normalne zasady konwersji liczb (zobacz podrozdział 5.17). Typem wyniku zawsze jest `int`.

5.5. Operatory porównania

Jeśli:

`a, b` są wyrażeniami dowolnego typu bazowego poza `void` lub oba są wskaźnikami,

to wyrażenie:

`a < b` ma wartość 1, jeśli `a` jest mniejsze od `b`, i ma wartość 0 w przeciwnym wypadku;

`a <= b` ma wartość 1, jeśli `a` jest mniejsze lub równe `b`, i ma wartość 0 w przeciwnym wypadku;

`a > b` ma wartość 1, jeśli `a` jest większe od `b`, i ma wartość 0 w przeciwnym wypadku;

`a >= b` ma wartość 1, jeśli `a` jest mniejsze lub równe `b`, i ma wartość 0 w przeciwnym wypadku;

`a == b` ma wartość 1, jeśli `a` jest równe `b`, i ma wartość 0 w przeciwnym wypadku;

`a != b` ma wartość 1, jeśli `a` jest różne od `b`, i ma wartość 0 w przeciwnym wypadku.

Do `a` i `b` stosowane są normalne zasady konwersji liczb (zobacz podrozdział 5.17). Pierwsze cztery testy porównania mają sens dla wskaźników, jeśli wskaźniki te wskazują tę samą tablicę lub pola tej samej struktury czy unii. Wynik zawsze jest typu `int`.

5.6. Operatory bitowe

Jeśli:

`i, j, n` są dowolnymi wyrażeniami będącymi liczbami całkowitymi,

to wyrażenie:

`i & j` wyznacza bitową koniunkcję, `i AND j`;

`i | j` wyznacza bitową alternatywę, `i OR j`;

`i ^ j` mnoży `XOR i` przez `j`;

`~i` dopełnia `i`;

`i << n` przesuwa `i` w lewo o `n` bitów;

`i >> n` przesuwa `i` w prawo o n bitów.

Do `i` i `j` stosowane są normalne zasady konwersji liczb, choć nie dotyczy to operatorów << oraz >> (zobacz podrozdział 5.17). Jeśli liczba bitów do przesunięcia jest ujemna albo większa lub równa liczbie bitów w obiekcie przesuwanym, wynik przesunięcia jest nieokreślony. W niektórych komputerach przesunięcie w prawo jest arytmetyczne (dopełnianie bitem znaku), w innych jest logiczne (dopełnienie zerami).Typ wyniku przesunięcia to lewy parametr po ewentualnych zmianach typu.

5.7. Operatory inkrementacji i dekrementacji

Jeśli:

`lv` jest modyfikowalną l-wartością, która nie została zadeklarowana ze słowem kluczowym `const`,

to wyrażenie:

`++lv` zwiększa `lv`, używa uzyskanej wartości w ewentualnym wyrażeniu;

`lv++` zwiększa `lv`, używa pierwotnej wartości w ewentualnym wyrażeniu.

—`lv` zmniejsza `lv`, używa uzyskanej wartości w ewentualnym wyrażeniu;

`lv`— zmniejsza `lv`, używa pierwotnej wartości w ewentualnym wyrażeniu.

5.8. Operatory przypisania

Jeśli:

`lv` to modyfikowalne l-wyrażenie niezadeklarowane jako `const`;

`op` to dowolny operator, którego można użyć w przypisaniach (zobacz tabelę A.5);

`a` to wyrażenie,

wtedy wyrażenie:

`lv = a` przypisuje wartość `a` do `lv`;

`lv` *`op`*`= a` przypisuje `lv` wartość `lv` *`op`* `a`.

W pierwszym wyrażeniu, jeśli `a` to zmienna jednego z typów bazowych (poza `void`), następuje przekształcenie na typ `lv`. Jeśli `lv` jest wskaźnikiem, musi być takiego samego typu jak `lv`, wskaźnikiem `void` lub wskaźnikiem pustym.

Jeśli `lv` jest wskaźnikiem `void`, to `a` może być dowolnego typu wskaźnikowego. Drugie wyrażenie jest równoważne wyrażeniu `lv = lv` *`op`* `(a)`, z tym że wartość `lv` jest obliczana tylko raz (rozważ `x[i++] += 10`).

5.9. Operator wyboru

Jeśli:

`a, b, c` są wyrażeniami,

to wyrażenie:

`a ? b : c` ma wartość `b`, jeśli `a` nie jest zerem; w przeciwnym razie ma wartość `c`; zawsze wyliczane jest tylko jedno z wyrażeń — albo `b`, albo `c`.

Wyrażenia `b` i `c` muszą być tego samego typu. Jeśli nie są tego samego typu, ale są liczbami, wykonywane są stosowne konwersje. Jeśli jedno z nich jest wskaźnikiem, a drugie zerem, to drugie jest traktowane jako wskaźnik pusty typu zgodnego z drugim wyrażeniem. Jeśli jedno jest wskaźnikiem `void`, a drugie wskaźnikiem innego typu, drugie jest konwertowane do typu „wskaźnik `void`" i takiż wynik jest zwracany.

5.10. Operator rzutowania

Jeśli:

`typ` to nazwa podstawowego typu danych, typu wyliczeniowego (poprzedzonego słowem kluczowym `enum`), typu zdefiniowanego przez `typedef` lub typu pochodnego;

`a` to wyrażenie,

wtedy wyrażenie:

`(` *`typ`* `)` konwertuje wyrażenie `a` na pożądany typ.

5.11. Operator sizeof

Jeśli:

`typ` to typ jak powyżej;

`a` to wyrażenie,

wtedy wyrażenie:

`sizeof (typ)`	ma wartość równą liczbie bajtów potrzebnych do zapisania wartości podanego typu;
`sizeof a`	ma wartość równą liczbie bajtów potrzebnych na przechowanie wyniku uzyskanego po wyliczeniu wyrażenia `a`.

Jeśli *typ* to `char`, wynikiem z definicji jest `1`. Jeśli `a` to tablica o ustalonej wielkości (ustalonej jawnie lub niejawnie, przez inicjalizację) `i` nie jest parametrem formalnym ani tablicą `extern` bez ustalonej wielkości, `sizeof` podaje liczbę bajtów potrzebnych na zapisanie elementów w `a`.

Jeśli `a` jest nazwą klasy, wynikiem operacji `sizeof(a)` jest rozmiar struktury danych potrzebnej do przechowywania jednego egzemplarza klasy `a`.

Typem zwracanym przez operator `sizeof` jest typ całkowitoliczbowy `size_t` zdefiniowany w standardowym pliku nagłówkowym `<stddef.h>`.

Jeśli `a` jest tablicą o zmiennej długości, operator `sizeof` jest interpretowany w trakcie działania programu. W pozostałych przypadkach wartość tego operatora jest wyznaczana na etapie kompilacji, a uzyskany wynik może być używany w wyrażeniach stałych (zobacz podrozdział 5.2).

5.12. Operator przecinek

Jeśli:

`a, b`	to wyrażenia,

wtedy wyrażenie:

`a, b`	powoduje interpretację najpierw `a`, potem `b`; typ i wartość całości zależy od wyrażenia `b`.

5.13. Podstawowe działania na tablicach

Jeśli:

`a`	jest tablicą mającą `n` elementów;
`i`	jest wyrażeniem całkowitoliczbowym;
`v`	jest wyrażeniem,

to wyrażenie:

`a[0]`	to odwołanie do pierwszego elementu tablicy `a`;
`a[n – 1]`	to odwołanie do ostatniego elementu tablicy `a`;
`a[i]`	to odwołanie do `i`-tego elementu tablicy `a`;
`a[i] = v`	powoduje zapisanie w `a[i]` wartości `v`.

We wszystkich tych przypadkach typem wyrażenia jest typ elementów umieszczanych w tablicy. Podsumowanie działań na wskaźnikach i tablicach podano w podrozdziale 5.15.

5.14. Podstawowe działania na strukturach[1]

Jeśli:

`x`	to modyfikowalne l-wyrażenie typu `struct` *s*;
`y`	to wyrażenie typu `struct` *s*;
`m`	to nazwa jednego z elementów struktury *s*;
`v`	to wyrażenie,

wtedy wyrażenie:

`x`	odwołuje się do całej struktury i jest typu `struct` *s*;
`y.m`	odwołuje się do pola `m` struktury `y` i jest takiego typu, jakiego typu jest pole `m`;
`x.m = v`	przypisuje wartość `v` polu `m` zmiennej `x`; całe wyrażenie jest takiego typu, jakiego typu jest pole `m`;
`x = y`	przypisuje `y` do `x` i jest typu `struct` *s*;
`f (y)`	wywołuje funkcję `f` i przekazuje jej zawartość struktury `y` jako parametr; w `f` odpowiedni parametr formalny musi być zadeklarowany jako parametr typu `struct` *s*;
`return y;`	zwraca strukturę `y`; wartość zwracana przez funkcję musi być typu `struct` *s*.

5.15. Podstawowe działania na wskaźnikach

Jeśli:

`x`	jest l-wartością typu *t*;
`pt`	jest modyfikowalną l-wartością typu „wskaźnik na typ *t*”;
`v`	jest wyrażeniem,

to wyrażenie:

`&x`	podaje wskaźnik `x` i jest typu „wskaźnik na typ *t*”;
`pt = &x`	ustawia wskaźnik `pt` tak, aby wskazywał `x`; jest typu „wskaźnik na typ `t`”;
`pt = 0`	powoduje, że `pt` staje się wskaźnikiem pustym;
`pt == 0`	sprawdza, czy `pt` jest wskaźnikiem pustym;
`*pt`	odwołuje się do wartości wskazywanej przez `pt` i jest typu `t`;
`*pt = v`	powoduje przypisanie wartości `v` obiektowi wskazywanemu przez `pt` i jest typu `t`.

[1] Te same informacje odnoszą się także do unii.

5.15.1. Wskaźniki tablic

Jeśli:

`a`	to tablica elementów typu `t`;
`pa1`	jest modyfikowalną l-wartością typu „wskaźnik na `t`”, wskazującą jeden z elementów tablicy `a`;
`pa2`	jest modyfikowalną l-wartością typu „wskaźnik na `t`”, wskazującą jeden z elementów tablicy `a` lub wskazującą tuż za ostatnim elementem `a`;
`v`	to wyrażenie;
`n`	to wyrażenie całkowitoliczbowe,

wtedy wyrażenie:

`a`, `&a`, `&a[0]`	podaje wskaźnik pierwszego elementu tablicy `a`;
`&a[n]`	podaje wskaźnik `n`-tego elementu tablicy `a` i jest typu „wskaźnik na `t`”;
`*pa1`	odwołuje się do elementu wskazywanego przez `pa1` i jest typu `t`;
`*pa1 = v`	zapisuje wartość `v` w elemencie tablicy wskazywanym przez `pa1`, jest typu `t`;
`++pa1`	powoduje, że `pa1` wskazuje następny element tablicy `a` niezależnie od typu tych elementów; jest ono typu „wskaźnik do `t`”;
`—pa1`	powoduje, że `pa1` wskazuje poprzedni element tablicy `a` niezależnie od typu tych elementów; jest ono typu „wskaźnik do `t`”;
`*++pa1`	zwiększa wartość wskaźnika `pa1`, a następnie odwołuje się do elementu wskazywanego teraz przez `pa1`; wyrażenie to jest typu `t`;
`*pa1++`	zwraca wartość wskazywaną przez `pa1`, po czym wskaźnik ten inkrementuje o jeden; wyrażenie jest typu `t`;
`pa1 + n`	podaje wskaźnik wskazujący `n` elementów tablicy `a` dalej niż sam wskaźnik `pa1`, jest typu „wskaźnik do `t`”;
`pa1 – n`	podaje wskaźnik wskazujący `n` elementów tablicy `a` bliżej (wcześniej) niż sam wskaźnik `pa1`, jest typu „wskaźnik do `t`”;
`*(pa1 + n) = v`	zapisuje wartość `v` w elemencie wskazywanym przez `pa1 + n`, jest typu `t`;
`pa1 < pa2`	sprawdza, czy `pa1` wskazuje element wcześniejszy niż `pa2`, jest typu `int` (do porównywania dwóch wskaźników można używać dowolnych operatorów relacyjnych);
`pa2 – pa1`	podaje liczbę elementów mieszczących się między wskaźnikami `p2` i `p1` (o ile `pa2` wskazuje element dalszy niż `pa1`), jest typu całkowitoliczbowego;

`a + n`	podaje wskaźnik `n`-tego elementu tablicy `a`, jest typu „wskaźnik na `t`” i pod każdym względem jest równoważne wyrażeniu `&a[n]`;
`*(a + n)`	odwołuje się do wartości `n`-tego elementu tablicy `a`, jest typu `t` i pod każdym względem jest równoważne wyrażeniu `a[n]`.

Typ liczby całkowitej uzyskiwanej przy odejmowaniu od siebie dwóch wskaźników to `ptrdiff_t` zdefiniowany w standardowym pliku nagłówkowym `<stddef.h>`.

5.15.2. Wskaźniki na struktury[2]

Jeśli:

`x`	to l-wyrażenie typu `struct s`;
`ps`	to modyfikowalna l-wartość typu „wskaźnik na `struct s`”;
`m`	to nazwa pola struktury `s`; pole jest typu `t`;
`v`	to wyrażenie,

wtedy wyrażenie:

`&x`	podaje wskaźnik `x` i jest typu „wskaźnik na `struct s`”;
`ps = &x`	ustawia wskaźnik `ps`, aby wskazywał strukturę `x`; jest typu „wskaźnik na `struct s`”;
`ps->m`	odwołuje się do pola `m` struktury wskazywanej przez `ps`, jest typu `t`;
`(*ps).m`	także odwołuje się do pola `m` struktury wskazywanej przez `ps` i jest pod każdym względem równoważne `ps->m`;
`ps->m = v`	przypisuje polu `m` struktury wskazywanej przez `ps` wartość `v`, jest typu `t`.

5.16. Literały złożone

Literał złożony to ujęta w nawiasy nazwa typu, za którą znajduje się lista inicjalizacyjna. Tworzona jest nienazwana lista podanego typu, której zasięg jest ograniczony do bloku, w jakim taką wartość utworzono; jeśli wartość zostanie wykreowana poza jakimkolwiek blokiem, jej zasięg jest globalny. W przypadku wartości globalnych wszystkie wyrażenia inicjalizujące muszą być stałymi.

Na przykład:

```
(struct point) {.x = 0, .y = 0}
```

to wyrażenie dające strukturę typu `struct point` z ustalonymi wartościami początkowymi. Taka wartość może być przypisana innej strukturze `struct point`, na przykład:

```
origin = (struct point) {.x = 0, .y = 0};
```

[2] Te same informacje odnoszą się także do unii.

lub może zostać przekazana funkcji wymagającej parametru typu `struct point`:

```
moveToPoint ((struct point) {.x = 0, .y = 0});
```

Można też definiować wartości innych, niestrukturalnych typów, jeśli na przykład `intPtr` jest typu `int *`, instrukcja:

```
intPtr = (int [100]) {[0] = 1, [50] = 50, [99] = 99 };
```

(która może wystąpić w dowolnym miejscu w programie) ustawia zmienną `intPtr` tak, aby wskazywała tablicę 100 liczb całkowitych; trzy elementy tej tablicy są inicjalizowane, tak jak to pokazano.

Jeśli wielkość tablicy nie jest podana, jest określana na podstawie listy inicjalizacyjnej.

5.17. Konwersje podstawowych typów danych

W wyrażeniach arytmetycznych język C przekształca argumenty w ściśle ustalonej kolejności:

Etap 1: Jeśli którykolwiek operand jest typu `long double`, drugi operand też jest przekształcany do tego typu, i to jest typ wyniku wyrażenia.

Etap 2: Jeśli którykolwiek operand jest typu `double`, drugi operand też jest przekształcany do tego typu, i to jest typ wyniku wyrażenia.

Etap 3: Jeśli którykolwiek operand jest typu `float`, drugi operand też jest przekształcany do tego typu, i to jest typ wyniku wyrażenia.

Etap 4: Jeśli któryś z operandów jest typu `_Bool`, `char`, `short int`, jest polem bitowym `int` lub informacją typu wyliczeniowego, jest konwertowany do typu `int` pod warunkiem, że się w tym typie mieści. W przeciwnym razie jest konwertowany na typ `unsigned int`. Jeśli oba operandy są tego samego typu, to taki jest też typ wyniku.

Etap 5: Jeśli oba operandy są ze znakiem lub oba są bez znaku, mniejszy typ całkowity jest promowany do typu większego, i to jest typ wyniku.

Etap 6: Jeśli operand bez znaku jest co do rozmiaru nie mniejszy od operandu ze znakiem, ten drugi zostaje pozbawiony znaku i taki jest typ wyniku.

Etap 7: Jeśli operator ze znakiem pozwala zapisać wszystkie możliwe wartości operandu bez znaku, do tego drugiego dodawany jest znak i taki jest typ uzyskanego ostatecznie wyniku.

Etap 8: Jeśli doszło aż do tego kroku, oba operandy są konwertowane do typu `unsigned` odpowiadającego operandowi ze znakiem.

Etap 4. formalnie nazywany jest *promocją do typu `int`*.

Konwersje operandów w typowych sytuacjach są dobrze zdefiniowane, choć trzeba powiedzieć o kilku rzeczach:

1. Konwersja typu `char` na typ `int` na niektórych maszynach może obejmować dodanie znaku, chyba że typ `char` zadeklarowano z modyfikatorem `unsigned`.
2. Konwersja liczby całkowitej ze znakiem na dłuższy typ całkowity powoduje rozszerzenie znaku w lewo; konwersja liczby całkowitej bez znaku na typ dłuższy powoduje wypełnienie z lewej strony zerami.
3. Konwersja dowolnej wartości na typ `_Bool` daje wartość `0`, jeśli pierwotna wartość była zerem, i `1` w każdym innym wypadku.
4. Konwersja dłuższej liczby całkowitej na krótszą powoduje odrzucenie lewej części liczby.
5. Konwersja dowolnej liczby zmiennoprzecinkowej na całkowitą powoduje odrzucenie ułamkowej części. Jeśli liczba całkowita jest zbyt mała, aby pomieścić część całkowitą liczby zmiennoprzecinkowej, wynik działania jest nieokreślony, podobnie jak w przypadku konwersji ujemnej liczby zmiennoprzecinkowej na liczbę całkowitą bez znaku.
6. Konwersja dłuższej liczby zmiennoprzecinkowej na krótszą może powodować zaokrąglenie przed odrzuceniem części, choć nie zawsze musi to mieć miejsce.

6.0. Klasy zmiennych i zakres

Pojęcie *klasa zmiennej* dotyczy sposobu alokowania pamięci na tę zmienną przez kompilator oraz zasięgu definicji funkcji. Klasy to `auto`, `static`, `extern` i `register`. W deklaracji można pomijać klasy — będą wtedy stosowane klasy domyślne, co omówimy dalej w tym podrozdziale.

Pojęcie *zasięgu* lub *zakresu* opisuje, jak daleko dany identyfikator jest rozumiany w programie. Identyfikator zdefiniowany poza funkcjami i blokami kodu (dalej określanymi mianem *BLOKÓW*) może być używany w dowolnym miejscu całego pliku. Identyfikatory definiowane w BLOKU są lokalne dla tego BLOKU i mogą lokalnie zmieniać definicję identyfikatora zewnętrznego. Nazwy etykiet są rozpoznawane w BLOKU, podobnie jak nazwy parametrów formalnych. Nazwy etykiet, struktur i pól struktur, unii i pól unii oraz nazwy typów wyliczeniowych nie różnią się niczym od nazw zmiennych i nazw funkcji. Jednak identyfikatory typów wyliczeniowych *różnią* się od nazw zmiennych i innych identyfikatorów wyliczeniowych zdefiniowanych w tym samym zakresie.

6.1. Funkcje

Jeśli w definicji funkcji podano klasę, musi to być albo `static`, albo `extern`. Funkcje deklarowane z klasą `static` są dostępne tylko w tym pliku, w którym funkcję taką zdefiniowano. Funkcje zadeklarowane jako `extern` (lub bez klasy) mogą być wywoływane przez funkcje znajdujące się w innych plikach.

6.2. Zmienne

W tabeli A.6 zestawiono różne klasy zmiennych dostępne podczas deklarowania zmiennych. Opisano ich zasięg oraz sposoby inicjalizacji.

Tabela A.6. Zmienne — ich klasy, zasięg i inicjalizacja

Jeśli klasą jest	i zmienną zadeklarowano,	to można się do niej odwoływać	i można ją inicjalizować	Uwagi
`static`	poza jakimkolwiek BLOKIEM	w całym pliku	tylko wyrażeniami stałymi	Zmienne inicjalizowane są tylko raz w chwili uruchamiania programu. Zmienne zachowują swoje wartości w BLOKACH. Wartością domyślną jest 0.
	w BLOKU	w BLOKU		
`extern`	poza jakimkolwiek BLOKIEM	w całym pliku	tylko wyrażeniami stałymi	Zmienna musi być przynajmniej w jednym miejscu zadeklarowana bez słowa kluczowego `extern` lub musi mieć tam słowo kluczowe `extern` oraz wartość początkową.
	w BLOKU	w BLOKU		
`auto`	w BLOKU	w BLOKU	dowolne prawidłowe wyrażenie	Zmienna jest inicjalizowana przy każdym wejściu do BLOKU, nie ma wartości domyślnej.
`register`	w BLOKU	w BLOKU	dowolne prawidłowe wyrażenie	Nie jest gwarantowane umieszczenie zmiennej w rejestrze. Na takie zmienne można nakładać różne ograniczenia, nie można pobrać ich adresu. Zmienna jest inicjalizowana przy każdym wejściu do BLOKU, nie ma wartości domyślnej.

Tabela A.6. Zmienne — ich klasy, zasięg i inicjalizacja (ciąg dalszy)

Jeśli klasą jest	i zmienną zadeklarowano,	to można się do niej odwoływać	i można ją inicjalizować	Uwagi
pominięta	poza BLOKIEM	gdziekolwiek w pliku lub w innych plikach zawierających odpowiednią deklarację	tylko wyrażenia stałe	Deklaracja taka może wystąpić tylko w jednym miejscu. Zmienna jest inicjalizowana podczas uruchamiania programu, a wartością domyślną jest 0.
	w BLOKU	(zobacz `auto`)	(zobacz `auto`)	Podobnie jak `auto`.

7.0. Funkcje

W tej części podsumowujemy informacje o składni funkcji i ich działaniu.

7.1. Definicja funkcji

Ogólna postać definicji funkcji jest następująca:

```
zwracanyTyp nazwa ( typ1 param1, typ2 param2, ... )
{
    deklaracjeZmiennych

    instrukcjaProgramu
    instrukcjaProgramu
    ...
    return wyrażenie;
}
```

Powyżej zdefiniowano funkcję *nazwa* zwracająca wartość typu *zwracanyTyp*, mającą parametry formalne *param1*, *param2*… odpowiednio typów *typ1*, *typ2*…

Zmienne lokalne zwykle deklaruje się na początku funkcji, ale nie jest to wymóg. Deklaracje zmiennych mogą wystąpić w dowolnym miejscu, ale widoczne będą dopiero od miejsca deklaracji.

Jeśli funkcja nie zwraca żadnej wartości, jako *zwracanyTyp* podaje się `void`.

Jeśli w nawiasach podane zostanie tylko słowo kluczowe `void`, funkcja nie ma parametrów. Jeśli ostatnim (lub jedynym) parametrem są trzy kropki (`...`), funkcja ma zmienną liczbę parametrów, na przykład:

```
int  printf (char *format, ...)
{
    ...
}
```

W przypadku deklarowania tablic jednowymiarowych jako parametrów funkcji, nie trzeba podawać liczby ich elementów. Dla tablic wielowymiarowych trzeba podać wszystkie wymiary poza pierwszym.

Instrukcję `return` omówiono w podrozdziale 8.9.

Jeśli przed definicją funkcji zostanie umieszczone słowo kluczowe `inline`, jest to podpowiedź dla kompilatora, że w miejscu użycia takiej funkcji należy umieścić jej kod zamiast jej wywołania. Dzięki temu funkcja szybciej działa. Przykładowo:

```
inline int min (int a, int b)
{
    return ( a < b ? a : b);
}
```

7.2. Wywołanie funkcji

Ogólna postać deklaracji funkcji jest następująca:

```
nazwa ( arg1, arg2, ... )
```

Funkcja *nazwa* jest wywoływana z parametrami *arg1*, *arg2*… Jeśli funkcja nie ma parametrów, podaje się tylko nawiasy, otwierający i zamykający (na przykład `initialize()`). W przypadku wywoływania funkcji zdefiniowanej już po tym wywołaniu lub w innym pliku należy podać *deklarację prototypu* tej funkcji, która wygląda następująco:

```
zwracanyTyp nazwa ( typ1 param1, typ2 param2, ... );
```

Wtedy kompilator zna typ zwracany przez funkcję, liczbę parametrów i ich typy, na przykład:

```
long double  power (double x, int n);
```

to deklaracja funkcji zwracającej wartość typu `long double`, mającej dwa parametry — pierwszy typu `double`, drugi typu `int`. Nazwy parametrów podawane w nawiasach nie mają znaczenia, można je w ogóle pominąć:

```
long double  power (double, int);
```

Jeśli kompilator wcześniej natknął się na definicję funkcji lub deklarację jej prototypu, w chwili wywołania funkcji typy wszystkich parametrów są automatycznie konwertowane stosownie do tej definicji lub deklaracji.

Jeśli nie zostanie podana definicja ani prototyp deklaracji, kompilator zakłada, że funkcja zwraca wartość typu `int`, wszystkie parametry typu `float` konwertuje do typu `double` oraz promuje parametry całkowitoliczbowe zgodnie z opisem w podrozdziale 5.17. Pozostałe parametry funkcji są przekazywane bez żadnych konwersji.

Funkcje mające zmienną liczbę parametrów trzeba odpowiednio zadeklarować. Inaczej kompilator ma pełne prawo założyć, że ma do czynienia z funkcją o stałej liczbie parametrów, natomiast ich liczba jest określana na podstawie wywołania.

Funkcja, dla której zwracaną wartość zadeklarowano jako `void`, powoduje, że kompilator oznaczy wszystkie wywołania tej funkcji, które będą próbowały skorzystać ze zwróconej przez nią wartości.

Wszystkie parametry funkcji są przekazywane przez wartość, czyli ich wartości nie mogą być wewnątrz funkcji zmieniane. Jeśli do funkcji przekazany zostanie wskaźnik, funkcja *może* zmienić wartość wskazywanej danej, ale nie może zmienić samego wskaźnika.

7.3. Wskaźniki funkcji

Nazwa funkcji, podana bez nawiasów, daje wskaźnik na tę funkcję. Do uzyskania wskaźnika funkcji można też użyć operatora adresu.

Jeśli `fp` to wskaźnik funkcji, wskazywaną funkcję możemy wywołać, pisząc albo:

```
fp ()
```

albo:

```
(*fp) ()
```

Jeśli taka funkcja ma jakieś parametry, podaje się je normalnie w nawiasach.

8.0. Instrukcje

Instrukcja programu to dowolne poprawne wyrażenie (zwykle przypisanie lub wywołanie funkcji), za którym znajduje się średnik lub które jest częścią opisanych dalej instrukcji specjalnych. Instrukcję może poprzedzać *etykieta* składająca się z identyfikatora i dwukropka zaraz za tym identyfikatorem (zobacz podrozdział 8.6).

8.1. Instrukcje złożone

Instrukcje programu możemy ująć w parę nawiasów klamrowych, tworząc w ten sposób instrukcję *złożoną*, czyli *blok*, który może wystąpić wszędzie tam, gdzie normalnie jest pojedyncza instrukcja. Blok może mieć własne deklaracje zmiennych nadpisujące tak samo nazwane zmienne spoza bloku. Zasięgiem takich zmiennych lokalnych jest blok, w którym je zdefiniowano.

8.2. Instrukcja break

Ogólna postać instrukcji `break` jest następująca:

```
break;
```

Wykonanie instrukcji `break` wewnątrz instrukcji `for`, `while`, `do` lub `switch` powoduje natychmiastowe wykonanie instrukcji zewnętrznej. Program dalej jest wykonywany zaraz za przerwaną instrukcją pętli lub instrukcją `switch`.

8.3. Instrukcja continue

Ogólna postać instrukcji `continue` jest następująca:

```
continue;
```

Wykonanie instrukcji `continue` wewnątrz pętli powoduje pominięcie końcówki aktualnej iteracji. Poza tym pętla jest wykonywana normalnie.

8.4. Instrukcja do

Ogólna postać instrukcji `do` jest następująca:

```
do
    instrukcjaProgramu
while ( wyrażenie );
```

`instrukcjaProgramu` jest wykonywana tak długo, jak długo *`wyrażenie`* daje niezerową wartość. Zauważmy, że *`wyrażenie`* jest wykonywane za każdym razem *po wykonaniu `instrukcjiProgramu`*. Instrukcja `do` gwarantuje, że *`instrukcjaProgramu`* zostanie wykonana przynajmniej raz.

8.5. Instrukcja for

Ogólna postać instrukcji `for` jest następująca:

```
for (wyrażenie1;  wyrażenie2;  wyrażenie3)
    instrukcjaProgramu
```

Wartość wyrażenia *`wyrażenie1`* jest obliczana w chwili wejścia do pętli. Następnie wyliczane jest *`wyrażenie2`*. Jeśli da ono niezerową wartość, wyliczane jest *`wyrażenie3`*. Naprzemienne wykonywanie *`instrukcjiProgramu`* i *`wyrażenia3`* odbywa się tak długo, jak długo *`wyrażenie2`* jest różne od zera. Zauważmy, że wobec tego, iż *`wyrażenie2`* jest interpretowane przed każdym wykonaniem *`instrukcjiProgramu`*, ta ostatnia może nie być wykonana ani raz, jeśli *`wyrażenie2`* jest zerem w chwili wejścia do pętli.

Zmienne lokalne pętli można deklarować w *`wyrażeniu1`*. Zasięgiem takich zmiennych jest pętla `for`. Na przykład w instrukcji:

```
for ( int i = 0; i < 100; ++i )
    ...
```

deklarujemy zmienną `i` typu `int` i ustawiamy ją na zero w chwili uruchomienia pętli. Zmienna ta jest dostępna dla wszystkich instrukcji w pętli, ale po zakończeniu działania pętli przestaje być dostępna.

8.6. Instrukcja goto

Ogólna postać instrukcji `goto` jest następująca:

```
goto identyfikator;
```

Wykonanie instrukcji `goto` powoduje przekazanie sterowania bezpośrednio do funkcji oznaczonej etykietą *`identyfikator`*. Oznaczona etykietą instrukcja musi być w tej samej funkcji co instrukcja `goto`.

8.7. Instrukcja if

Ogólna postać instrukcji `if` jest następująca:

```
if ( wyrażenie )
    instrukcjaProgramu
```

Jeśli wynik interpretacji *`wyrażenia`* nie jest zerem, wykonywana jest *`instrukcjaProgramu`*. W przeciwnym razie instrukcja ta jest pomijana.

Kolejna postać instrukcji `if` to:

```
if ( wyrażenie )
    instrukcjaProgramu1
else
    instrukcjaProgramu2
```

Jeśli wartość *`wyrażenia`* nie jest zerem, wykonywana jest *`instrukcjaProgramu1`*; w przeciwnym razie wykonywana jest *`instrukcjaProgramu2`*. Jeśli *`instrukcjaProgramu2`* to kolejna instrukcja `if`, można tworzyć cały ciąg instrukcji `if-else if`:

```
if ( wyrażenie1 )
    instrukcjaProgramu1
else if ( wyrażenie2 )
    instrukcjaProgramu2
    ...
else
    instrukcjaProgramuN
```

Fraza `else` zawsze dotyczy ostatniej instrukcji `if`, która nie ma wcześniej `else`. Korzystając z nawiasów klamrowych, możemy w razie potrzeby zmienić to powiązanie.

8.8. Instrukcja pusta

Ogólna postać instrukcji pustej to:

```
;
```

Wykonanie instrukcji pustej nie powoduje niczego i służy głównie do spełnienia formalnych wymogów składniowych instrukcji, takich jak `for`, `do` czy `while`. Na przykład w poniższej instrukcji, kopiującej znaki z `from` do `to`:

```
while ( *to++ = *from++ )
    ;
```

instrukcja pusta jest niezbędna, gdyż w pętli `while` wymagane jest wystąpienie instrukcji.

8.9. Instrukcja return

Ogólna postać instrukcji `return` to:

```
return;
```

Wykonanie instrukcji `return` powoduje, że sterowanie jest przekazywane do funkcji wywołującej. Pokazana powyżej postać instrukcji `return` jest używana w przypadku funkcji niezwracających wartości.

Jeśli sterowanie przejdzie do końca funkcji i nie zostanie tam znaleziona instrukcja `return`, program zadziała tak, jakby wystąpiła tam pokazana powyżej bezparametrowa instrukcja `return` — nie jest zwracana żadna wartość.

Druga postać instrukcji `return` to:

```
return wyrażenie;
```

Do funkcji wywołującej zwracana jest wartość *wyrażenia*. Jeśli typ *wyrażenia* jest niezgodny z typem zwracanym wskazanym w deklaracji funkcji, wartość jest automatycznie konwertowana na odpowiedni typ i dopiero wtedy zwracana.

8.10. Instrukcja switch

Ogólna postać instrukcji `switch` jest następująca:

```
switch ( wyrażenie )
{
    case stała1:
        instrukcjaProgramu
        instrukcjaProgramu
        ...
        break;
    case stała2:
        instrukcjaProgramu
        instrukcjaProgramu
        ...
        break;
    ...
    case stałaN:
        instrukcjaProgramu
        instrukcjaProgramu
        ...
        break;
    default:
        instrukcjaProgramu
        instrukcjaProgramu
        ...
        break;
}
```

Najpierw wyliczana jest wartość *wyrażenia*, która później zostaje porównywana z kolejnymi wyrażeniami stałymi: *stała1*, *stała2*, ..., *stałaN*. Jeśli wartość *wyrażenia* pasuje do jednej ze stałych, wykonywane są instrukcje znajdujące się zaraz za tą stałą. Jeśli nie zostanie dopasowana żadna wartość, wykonywana jest fraza `default`, pod warunkiem

że została zdefiniowana. Jeśli jej zabraknie, może nie być wykonana żadna instrukcja znajdująca się wewnątrz `switch`.

Wynik *`wyrażenia`* musi być liczbą całkowitą, poza tym żadne dwie stałe nie mogą być takie same. Pominięcie instrukcji `break` powoduje, że po wykonaniu instrukcji jednej stałej wykonywane będą instrukcje następnej stałej.

8.11. Instrukcja while

Ogólna postać instrukcji `while` jest następująca:

```
while ( wyrażenie )
    instrukcjaProgramu
```

Instrukcja *`instrukcjaProgramu`* jest wykonywana tak długo, jak długo *`wyrażenie`* jest różne od zera. Zauważmy, że skoro *`wyrażenie`* jest interpretowane każdorazowo przed wykonaniem *`instrukcjiProgramu`*, może się zdarzyć, że nie zostanie ona wykonana ani razu.

9.0. Preprocesor

Preprocesor analizuje kod źródłowy przed przekazaniem go do kompilatora. Preprocesor wykonuje następujące zadania:

1. Zastępuje trójznaki (zobacz podrozdział 9.1) ich odpowiednikami.
2. Łączy wszystkie wiersze kończące się odwrotnym ukośnikiem w pojedyncze długie wiersze.
3. Dzieli program na strumień elementów.
4. Usuwa komentarze i zastępuje je pojedynczymi spacjami.
5. Interpretuje dyrektywy preprocesora (zobacz podrozdział 9.2) oraz rozwija makra.

9.1. Trójznaki

Aby obsłużyć zestawy znaków inne niż ASCII, preprocesor obsługuje trzyznakowe ciągi (nazywane *trójznakami*) z tabeli A.7, traktując je szczególnie zarówno w samym programie, jak i w łańcuchach znakowych.

9.2. Dyrektywy preprocesora

Wszystkie dyrektywy preprocesora zaczynają się od znaku #, który musi być pierwszym czarnym znakiem w wierszu. Za znakiem # mogą ewentualnie pojawić się spacje lub tabulatory.

Tabela A.7. Trójznaki

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

9.2.1. Dyrektywa #define

Ogólna postać dyrektywy `#define` jest następująca:

```
#define nazwa  tekst
```

W ten sposób dla preprocesora definiuje się identyfikator *nazwa* i wiąże z nim dowolny *tekst* zaczynający się od pierwszego niebiałego znaku po *nazwie*, kończący się wraz z końcem wiersza. Każde następne użycie *nazwy* w programie powoduje, że identyfikator jest zastępowany *tekstem*.

Inna ogólna postać dyrektywy `#define` jest następująca:

```
#define nazwa (param1, param2, ..., paramN)  tekst
```

Makro *nazwa* ma parametry *param1*, *param2*, ..., *paramN*, z których każdy jest identyfikatorem. Kiedy w programie użyjemy *nazwy* z listą parametrów, w to miejsce wstawiony zostanie *tekst*, przy czym parametry zostaną zastąpione parametrami przekazanymi do makra.

Jeśli makro ma zmienną liczbę parametrów, wtedy listę parametrów kończymy trzykropkiem. Pozostałe parametry w makrze są dostępne jako całość za pośrednictwem specjalnego identyfikatora `__VA_ARGS__`. Poniższe makro — `myPrintf` — ma najpierw łańcuch formatujący, a za nim zmienną liczbę parametrów:

```
#define myPrintf(...)    printf("DEBUG: " __VA_ARGS__);
```

Oto przykład prawidłowego użycia powyższego makra:

```
myPrintf ("Witaj, świecie!\n");
```

albo:

```
myPrintf ("i = %i, j = %i\n", i, j);
```

Jeśli definicja powinna być dłuższa niż jeden wiersz, każdy wiersz poza ostatnim musi kończyć się odwrotnym ukośnikiem. Kiedy nazwa zostanie raz zdefiniowana, można używać jej w dowolnym miejscu pliku.

W definicjach makr z parametrami można używać dwuargumentowego operatora #. Za nim znajduje się nazwa parametru makra. W przypadku jego użycia preprocesor otacza wartość przekazaną w wywołaniu makra cudzysłowem, robiąc z niego łańcuch znakowy. Jeśli na przykład mamy następującą definicję:

```
#define  printfint(x)  printf (# x " = %d\n", x)
```

i wywołujemy ją:

```
printint (count);
```

preprocesor wywołanie to rozwinie na:

```
printf ("count" " = %i\n", count);
```

czyli:

```
printf ("count = %i\n", count);
```

Preprocesor w przypadku przekształcania danych na łańcuchy znakowe poprzedza wszystkie cudzysłowy (") i odwrotne ukośniki (\) znakiem \. Zatem jeśli mamy definicję:

```
#define  str(x)  # x
```

to wywołanie:

```
str (Łańcuch "\t" zawiera tabulator)
```

zostanie rozwinięte na:

```
"Łańcuch \"\\t\" zawiera tabulator"
```

W dyrektywach `#define` mających parametry można też użyć operatora ##. Znajduje się on przed nazwą parametru makra lub za nią. Preprocesor pobiera wartość przekazaną do makra, po czym tworzy z niej i wartości znajdującej się za operatorem ## (lub przed nim) całość. Jeśli na przykład mamy makro:

```
#define printx(n)  print ("%i\n", x ## n );
```

i wywołanie:

```
printx (5)
```

to w wyniku uzyskamy kod:

```
printf ("%i\n", x5);
```

Przy definicji:

```
#define printx(n) printf ("x" # n " = %i\n", x ## n );
```

wywołanie:

```
printx(10)
```

ostatecznie zostanie zinterpretowane jako:

```
printf ("x10 = %i\n", x10);
```

Operatory # i ## nie wymagają otaczania spacjami.

9.2.2. Dyrektywa #error

Dyrektywa `#error` ma postać:

```
#error tekst
```

Powoduje wypisanie przez preprocesor podanego *tekstu* jako komunikatu błędu.

9.2.3. Dyrektywa #if

Dyrektywa `#if` ma ogólną postać:

```
#if wyrażenie_stałe
    ...
#endif
```

Analizowana jest wartość *wyrażenia_stałego*. Jeśli wynik nie jest zerem, przetwarzane są wszystkie instrukcje programu od `#if` aż do `#endif`. Inaczej wiersze te są pomijane przez preprocesor i przez kompilator.

Inna ogólna postać dyrektywy `#if` to:

```
#if wyrażenie_stałe_1
    ...
#elif wyrażenie_stałe_2
    ...
#elif wyrażenie_stałe_n
    ...
#else
    ...
#endif
```

Jeśli *wyrażenie_stałe_1* nie jest zerem, przetwarzane są wiersze do najbliższego `#elif`, a reszta — aż do `#endif` — jest pomijana. Jeśli *wyrażenie_stałe_2* nie jest zerem, wykonywane są wiersze do następnego `#elif`, a cała reszta — aż do `#endif` — jest pomijana. Jeżeli wszystkie wyrażenia stałe są zerami, wykonywane są wiersze między `#else` a `#endif` (o ile w ogóle występuje `#else`). Do sprawdzania, czy zdefiniowana została jakaś stała, można użyć specjalnego operatora `defined`. Wobec tego, jeśli mamy dyrektywy:

```
#if defined (DEBUG)
    ...
#endif
```

kod między `#if` a `#endif` zostanie wykonany tylko wtedy, gdy wcześniej zostanie zdefiniowany identyfikator `DEBUG` (zobacz też podrozdział 9.2.4). Nawiasy wokół identyfikatora nie są obowiązkowe, więc równie dobrze zadziała:

```
#if defined DEBUG
```

9.2.4. Dyrektywa #ifdef

Ogólna postać dyrektywy `#ifdef` to:

```
#ifdef identyfikator
    ...
#endif
```

Jeśli *identyfikator* został wcześniej zdefiniowany (dyrektywą `#define` lub opcją `–D` wiersza poleceń używaną w chwili kompilacji programu), przetwarzane będą wszystkie wiersze od `#ifdef` do `#endif`. W przeciwnym razie wiersze te zostaną pominięte. Tak jak w przypadku dyrektywy `#if`, z dyrektywą `#ifdef` można łączyć dyrektywy `#elif` i `#else`.

9.2.5. Dyrektywa #ifndef

Ogólna postać dyrektywy `#ifndef` to:

```
#ifndef identyfikator
    ...
#endif
```

Jeśli *identyfikator* nie został wcześniej zdefiniowany, przetwarzane będą wszystkie wiersze od `#ifndef` do `#endif`. W przeciwnym razie wiersze te zostaną pominięte. Tak jak w przypadku dyrektywy `#if`, z dyrektywą `#ifndef` można łączyć dyrektywy `#elif` i `#else`.

9.2.6. Dyrektywa #include

Ogólna postać dyrektywy `#include` to:

```
#include "nazwaPliku"
```

Preprocesor przeszukuje katalogi ustalone dla danej implementacji, szukając pliku *nazwaPliku*. Zwykle najpierw przeszukiwany jest katalog, w którym mamy plik z programem. Jeśli tu plik nie zostanie znaleziony, przeszukiwane są pozostałe standardowe katalogi. Po znalezieniu pliku preprocesor włącza zawartość tego pliku do programu tam, gdzie wystąpiła dyrektywa `#include`. Następnie analizowane i interpretowane są dyrektywy preprocesora z włączonego pliku; mogą być wśród nich dalsze dyrektywy `#include`.

Inna postać dyrektywy `#include` to:

```
#include <nazwaPliku>
```

W tym wypadku plik jest szukany tylko w katalogach standardowych. Poza tym ta postać dyrektywy `#include` zachowuje się identycznie jak poprzednia.

W końcu można użyć zdefiniowanej wcześniej stałej preprocesora. Zadziała zatem także następujący kod:

```
#define DATABASE_DEFS    </usr/data/database.h>
    ...
#include DATABASE_DEFS
```

9.2.7. Dyrektywa #line

Ogólna postać dyrektywy `#line` to:

```
#line  stała  "nazwaPliku"
```

Dyrektywa ta powoduje, że dalsze wiersze programu kompilator traktuje tak, jakby nazwą pliku była *nazwaPliku*, a wiersze miały kolejne numery, od *stałej* poczynając. Jeśli nie zostanie podana *nazwaPliku*, używana jest nazwa z poprzedniej dyrektywy `#line`, a jeśli i tej nie ma, stosowana jest faktyczna nazwa pliku.

Dyrektywa `#line` służy głównie do kontrolowania nazw plików i numerów wierszy pokazywanych w przypadku znalezienia błędu przez kompilator.

9.2.8. Dyrektywa #pragma

Ogólna postać dyrektywy `#pragma` to:

```
#pragma  tekst
```

Dyrektywa ta powoduje, że preprocesor wykonuje pewne działania zależne od konkretnej implementacji, na przykład dyrektywa:

```
#pragma loop_opt(on)
```

może powodować dodatkową optymalizację pętli przez jakiś konkretny kompilator języka C. Jeśli kompilator napotka nieznaną mu dyrektywę `#pragma`, ignoruje ją.

Po dyrektywie `#pragma` można użyć specjalnego słowa kluczowego `STDC`. Obecnie obsługiwane „przełączniki", jakie mogą występować za `#pragma STDC`, to `FP_CONTRACT`, `FENV_ACCESS` oraz `CX_LIMITED_RANGE`.

9.2.9. Dyrektywa #undef

Ogólna postać dyrektywy `#undef` to:

```
#undef identyfikator
```

Podany *identyfikator* przestaje być znany preprocesorowi. Występujące dalej dyrektywy `#ifdef` i `#ifndef` zachowują się tak, jakby podany *identyfikator* nigdy nie został zdefiniowany.

9.2.10. Dyrektywa

Jest to pusta dyrektywa, ignorowana przez preprocesor.

9.3. Identyfikatory predefiniowane

W tabeli A.8 zestawiono identyfikatory definiowane przez sam preprocesor.

Tabela A.8. Predefiniowane identyfikatory preprocesora

Identyfikator	Znaczenie
`__LINE__`	Numer wiersza aktualnie kompilowanego.
`__FILE__`	Nazwa pliku źródłowego aktualnie kompilowanego.
`__DATE__`	Data pliku kompilowanego w formacie "*mm dd rrrr*".
`__TIME__`	Czas pliku kompilowanego w formacie "*hh*:*mm*:*ss*".
`__STDC__`	Równe 1, jeśli używany kompilator jest zgodny ze standardem ANSI. W przeciwnym razie identyfikator ten ma wartość 0.
`__STDC_HOSTED__`	Równe 1, jeśli implementacja jest obsługiwana, i 0 w przeciwnym wypadku.
`__STDC_VERSION__`	Z definicji równe 199901L.

B

Standardowa biblioteka C

Standardowa biblioteka języka C zawiera mnóstwo funkcji, które są dostępne we wszystkich programach pisanych w języku C. W tym dodatku nie wyliczymy wszystkich funkcji, tylko te najczęściej używane. Pełną listę wszystkich funkcji łatwo znaleźć w dokumentacji dołączanej do kompilatora lub w zewnętrznych źródłach, na przykład tych wymienionych w dodatku E.

W tym dodatku między innymi nie omawiamy funkcji obsługujących daty i czas (jakich jak `time`, `ctime`, `localtime`), wykonujących dalekie skoki (`setjmp` i `longjmp`), generujących informacje diagnostyczne (`assert`), obsługujących zmienną liczbę parametrów (`va_list`, `va_start`, `va_arg` i `va_end`), obsługujących sygnały (`signal` i `raise`), obsługujących ustawienia lokalne (zdefiniowane w pliku `<locale.h>`) oraz służących do obsługi łańcuchów znakowych.

Standardowe pliki nagłówkowe

W tej części dodatku opiszemy zawartość niektórych plików nagłówkowych, takich jak `<stddef.h>`, `<stdbool.h>`, `<limits.h>`, `<float.h>` i `<stdinit.h>`.

<stddef.h>

Plik ten zawiera pewne definicje standardowe, oto one:

Definicja	Znaczenie
`NULL`	Stała: pusty wskaźnik.
`offsetof(`*`struktura`*`, `*`pole`*`)`	Przesunięcie w bajtach pola *`pole`* od początku struktury *`struktura`*. Wynik jest typu `size_t`.
`ptrdiff_t`	Typ całkowitoliczbowy będący wynikiem odejmowania od siebie dwóch wskaźników.

`size_t`	Typ całkowitoliczbowy uzyskiwany w wyniku użycia operatora `sizeof`.
`wchar_t`	Typ całkowitoliczbowy używany do zapisu „szerokich znaków” (zobacz dodatek A).

<limits.h>

Ten plik nagłówkowy zawiera różne, zależne od konkretnej implementacji, ograniczenia typów całkowitych i znakowych. Niektóre wartości minimalne gwarantuje standard ANSI; zostały one podane na końcu opisów w nawiasach.

Definicja	Znaczenie
`CHAR_BIT`	Liczba bitów w znaku `char` (8).
`CHAR_MAX`	Maksymalna wartość typu `char` (127, jeśli jest to wielkość ze znakiem, i 255 w przeciwnym razie).
`CHAR_MIN`	Minimalna wartość typu `char` (–127, jeśli jest to wielkość ze znakiem, i 0 w przeciwnym razie).
`SCHAR_MAX`	Maksymalna wartość typu `signed char` (127).
`SCHAR_MIN`	Minimalna wartość typu `signed char` (–127).
`UCHAR_MAX`	Maksymalna wartość typu `unsigned char` (255).
`SHRT_MAX`	Maksymalna wartość typu `short int` (32767).
`SHRT_MIN`	Minimalna wartość typu `short int` (–32767).
`USHRT_MAX`	Maksymalna wartość typu `unsigned short int` (65535).
`INT_MAX`	Maksymalna wartość typu `int` (32767).
`INT_MIN`	Minimalna wartość typu `int` (–32767).
`UINT_MAX`	Maksymalna wartość typu `unsigned int` (65535).
`LONG_MAX`	Maksymalna wartość typu `long int` (2147483647).
`LONG_MIN`	Minimalna wartość typu `long int` (–2147483647).
`ULONG_MAX`	Maksymalna wartość typu `unsigned long int` (4294967295).
`LLONG_MAX`	Maksymalna wartość typu `long long int` (9223372036854775807).
`LLONG_MIN`	Minimalna wartość typu `long long int` (–9223372036854775807).
`ULLONG_MAX`	Maksymalna wartość typu `unsigned long long int` (18446744073709551615).

<stdbool.h>

Ten plik nagłówkowy zawiera definicje ułatwiające pracę ze zmiennymi logicznymi (typu `_Bool`).

Definicja	Znaczenie
`bool`	Inna nazwa typu danych `_Bool`.
`true`	Z definicji równe 1.
`false`	Z definicji równe 0.

<float.h>

Ten plik nagłówkowy zawiera różne ograniczenia związane z arytmetyką zmiennoprzecinkową. Wielkości minimalne są podawane w nawiasach na końcu każdego opisu. Poniżej nie podano wszystkich definicji.

Definicja	Znaczenie
`FLT_DIG`	Liczba cyfr znaczących dla danych typu `float` (6).
`FLT_EPSILON`	Najmniejsza wartość, która dodana do 1.0 nie będzie równa 1.0 (1e–5).
`FLT_MAX`	Maksymalna wartość typu `float` (1e+37).
`FLT_MAX_EXP`	Maksymalna wartość typu `float` (1e+37).
`FLT_MIN`	Minimalna znormalizowana wartość typu `float` (1e–37).

Istnieją podobne definicje dla typów `double` i `long double`. Wystarczy zamienić początkowe napisy `FLT` na `DBL`, aby uzyskać parametry typu `double`, i na `LDBL`, aby uzyskać parametry typu `long double`. Na przykład `DBL_DIG` określa liczbę cyfr znaczących typu `double`, a `LDBL_DIG` liczbę cyfr znaczących typu `long double`.

Pamiętać trzeba, że do pobierania informacji o środowisku i zachowania pełniejszej kontroli nad wielkościami zmiennoprzecinkowymi wykorzystywany jest plik nagłówkowy `<fenv.h>`. Na przykład istnieje tam funkcja `fesetround` pozwalająca podać kierunek zaokrąglania wartości do wartości zdefiniowanej w `<fenv.h>`: `FE_TONEAREST`, `FE_UPWARD`, `FE_DOWNWARD` i `FE_TOWARDZERO`. Możemy też zerować, podnosić i sprawdzać wyjątki liczb zmiennoprzecinkowych. Używa się do tego odpowiednio funkcji `feclearexcept`, `feraiseexcept` i `fetextexcept`.

<stdint.h>

Ten plik nagłówkowy zawiera definicje różnych stałych, których używa się do obliczeń całkowitoliczbowych mniej zależnych od konkretnej maszyny. Na przykład do zadeklarowania liczby całkowitej ze znakiem o długości dokładnie 32 bitów można użyć `typedef int32_t`; nie trzeba wtedy zakładać, że program jest uruchamiany

na komputerze 32-bitowym. Analogicznie `int_least32_t` to typ danych całkowitych mający przynajmniej 32 bity. Inne typy z tego pliku umożliwiają na przykład wybranie najszybciej działających liczb całkowitych. Więcej informacji na ten temat można znaleźć bezpośrednio w pliku lub w dokumentacji.

Oto jeszcze kilka przydatnych definicji z omawianego pliku.

Definicja	Znaczenie
`intptr_t`	Liczba całkowita, która na pewno wystarcza do zapisania wskaźnika.
`uintptr_t`	Liczba całkowita bez znaku, która na pewno wystarcza do zapisania wskaźnika.
`intmax_t`	Największy typ całkowity ze znakiem.
`uintmax_t`	Największy typ całkowity bez znaku.

Funkcje obsługujące łańcuchy znakowe

Poniższe funkcje wykonują działania na tablicach znakowych. W opisach *s*, *s1* i *s2* to zakończone znakami null tablice znakowe, *c* to wartość typu `int`, a `n` to liczba typu `size_t` (zgodnie z definicją z pliku *stddef.h*). W przypadku funkcji `strnxxx` *s1* i *s2* mogą wskazywać tablice znakowe bez znaku null na końcu.

Aby użyć poniższych funkcji, w programie trzeba włączyć plik nagłówkowy `<string.h>`:

```
#include <string.h>
```

`char *strcat (s1, s2)`

Łączy łańcuch znakowy *s2* z końcem łańcucha *s1*, na końcu uzyskanego łańcucha wstawia null. Funkcja zwraca *s1*.

`char *strchr (s, c)`

Odszukuje w łańcuchu *s* pierwsze wystąpienie znaku *c*. Jeśli znak zostanie znaleziony, zwracany jest wskaźnik tego znaku; w przeciwnym razie zwracany jest wskaźnik pusty.

`int strcmp (s1, s2)`

Porównuje łańcuchy *s1* i *s2*. Jeśli *s1* jest mniejszy od *s2*, zwraca wartość mniejszą od zera. Jeśli oba łańcuchy są równe, zwraca zero. W przeciwnym wypadku zwraca wartość większą od zera.

`char *strcoll (s1, s2)`

Działa jak `strcmp`, ale *s1* i *s2* to wskaźniki łańcuchów zapisanych zgodnie z bieżącymi ustawieniami lokalnymi.

`char *strcpy (s1, s2)`

Kopiuje *s2* do *s1*, zwraca *s1*.

`char *strerror(n)`

Zwraca komunikat odpowiadający błędowi o podanym numerze.

`size_t strcspn(s1, s2)`

Zlicza maksymalną liczbę znaków początkowych niewystępujących w *s2*, zwraca wynik.

`size_t strlen (s)`

Zwraca liczbę znaków w *s*, nie wlicza końcowego znaku null.

`char *strncat (s1, s2, n)`

Kopiuje *s2* na *koniec s1* do czasu, aż zostanie osiągnięty koniec łańcucha *s2* albo skopiowane zostanie *n* znaków. Zwraca *s1*.

`int strncmp (s1, s2, n)`

Działa tak samo jak `strcmp`, ale porównuje co najwyżej *n* znaków.

`char *strncpy (s1, s2, n)`

Kopiuje *s2* do *s1* do czasu, aż zostanie osiągnięty koniec łańcucha *s2* albo skopiowane zostanie *n* znaków. Zwraca *s1*.

`char *strrchr (s, c)`

Wyszukuje w łańcuchu s ostatnie wystąpienie znaku *c*. Jeśli znak ten zostanie znaleziony, zwraca wskaźnik do niego. W przeciwnym razie zwraca wskaźnik pusty.

`char *strpbrk (s1, s2)`

Znajduje pierwsze wystąpienie dowolnego znaku z *s2* w łańcuchu *s1*, zwraca wskaźnik do znalezionego znaku lub pusty wskaźnik w razie niepowodzenia.

`size_t strspn (s1, s2)`

Zlicza możliwe dużo początkowych znaków *s1* występujących w *s2*, zwraca wynik.

`char *strstr (s1, s2)`

Wyszukuje w łańcuchu *s1* pierwsze wystąpienie łańcucha *s2*. Jeśli poszukiwanie się uda, zwraca wskaźnik do początku znalezionego *s2*. W przeciwnym razie, jeśli w *s1* nie zostanie znalezione *s2*, zwraca wskaźnik pusty.

`char *strtok (s1, s2)`

Dzieli łańcuch *s1* na części na podstawie ograniczników wskazanych w łańcuchu *s2*. W pierwszym wywołaniu analizowany jest łańcuch *s1*, natomiast *s2* zawiera listę znaków ograniczających części. Funkcja wstawia do *s1* znak wskazujący koniec części, zwraca wskaźnik jej początku. W kolejnych wywołaniach *s1* powinno być wskaźnikiem pustym. Kiedy nie ma już więcej części, zwracany jest wskaźnik pusty.

`size_t strxfrm (s1, s2, n)`

Przekształca co najwyżej *n* znaków z łańcucha *s2*, wynik umieszcza w *s1*. Dwa tak przekształcone łańcuchy z ustawieniami lokalnymi mogą być porównywane za pomocą funkcji `strcmp`.

Obsługa pamięci

Poniższe funkcje obsługują tablice znakowe. Z założenia mają szybko przeszukiwać pamięć i kopiować dane z jednego miejsca w inne. Wymagają użycia pliku nagłówkowego `<string.h>`:

```
#include <string.h>
```

Poniżej, w ich opisach, *m1* i *m2* są typu `void *`, *c* jest daną typu `int` przekształcaną na typ `unsigned char`, a *n* to liczba typu `size_t`.

`void *memchar (m1, c, n)`

Wyszukuje w *m1* pierwsze wystąpienie *c*, zwraca znaleziony wskaźnik lub pusty wskaźnik w razie niepowodzenia. Sprawdzaniu podlega *n* znaków.

`void *memcmp (m1, m2, n)`

Porównuje *n* znaków z *m1* i *m2*. Jeśli są równe, zwraca 0. Jeśli nie, zwracana jest różnica wynikająca z porównania pierwszych niepasujących znaków. Jeśli zatem odpowiedni znak z *m1* był mniejszy od odpowiedniego znaku z *m2*, zwracana jest wartość mniejsza od zera; w przeciwnym razie zwracana jest wartość większa od zera.

`void *memcpy(m1, m2, n)`

Kopiuje *n* znaków z *m2* do *m1*, zwraca *m1*.

`void *memmove (m1, m2, n)`

Działa jak `memcpy`, ale zadziała zawsze, nawet wtedy, gdy *m1* i *m2* zachodzą na siebie.

`void *memset(m1, c, n)`

Ustawia pierwszych *n* znaków z *m1* na wartość *c*, zwraca *m1*.

Zauważmy, że opisane funkcje nie traktują znaku null jako specjalnej wartości. Mogą korzystać ze wskaźników dowolnego typu, wystarczy zastosować rzutowanie. Jeśli zatem `data1` i `data2` to tablice 100 liczb typu `int`, wywołanie:

```
memcpy ((void *) data2, (void *) data1, sizeof (data1));
```

skopiuje 100 liczb całkowitych z `data1` do `data2`.

Funkcje obsługi znaków

Poniższe funkcje służą do obsługi pojedynczych znaków. Aby ich użyć, trzeba do programu włączyć plik `<ctype.h>`:

```
#include <ctype.h>
```

Każda z poniższych funkcji jako parametr ma wartość typu `int` (tutaj *c*), zwraca wartość `TRUE` (niezerową) w razie spełnienia warunku oraz `FALSE` (zero) w przeciwnym razie.

Nazwa	Sprawdza
`isalnum`	Czy *c* jest znakiem alfanumerycznym?
`isalpha`	Czy *c* jest literą?
`isblank`	Czy *c* jest znakiem białym (spacja lub tabulator)?
`iscntrl`	Czy *c* jest znakiem kontrolnym?
`isdigit`	Czy *c* jest cyfrą?
`isgraph`	Czy *c* jest znakiem graficznym (czyli czarnym znakiem)?
`islower`	Czy *c* jest małą literą?
`isprint`	Czy *c* jest znakiem drukowalnym, czyli czarnym znakiem lub spacją?
`ispunct`	Czy *c* jest znakiem interpunkcyjnym (dowolny znak poza spacją i znakami alfanumerycznymi)?
`isspace`	Czy *c* jest białym znakiem (spacją, znakiem nowego wiersza, znakiem powrotu karetki, tabulatorem poziomym lub pionowym, znakiem nowej strony)?
`isupper`	Czy *c* jest wielką literą?
`isxdigit`	Czy *c* jest cyfrą szesnastkową?

Następujące dwie funkcje służą do podmiany znaków:

`int tolower (c)`

Zwraca odpowiednik *c* będący małą literą. Jeśli *c* nie jest wielką literą, nie ulega zmianie.

`int toupper (c)`

Zwraca odpowiednik *c* będący wielką literą. Jeśli *c* nie jest małą literą, nie ulega zmianie.

Funkcje wejścia i wyjścia

Oto najczęściej używane funkcje wejścia i wyjścia z biblioteki standardowej C. Aby używać ich w swoich programach, trzeba do tych programów dołączyć plik nagłówkowy `<stdio.h>`:

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

W pliku tym znajdują się deklaracje funkcji I/O oraz definicje nazw `OEF`, `NULL`, `stdin`, `stdout`, `stderr` (wszystkie są stałymi) oraz `FILE`.

W poniższych opisach *nazwaPliku*, *nazwaPliku1*, *nazwaPliku2*, *tryb* i *format* to wskaźniki łańcuchów zakończonych znakiem null. Parametr *bufor* to wskaźnik tablicy znakowej, *filePtr* jest wskaźnikiem do struktury `FILE`, *n* i *rozmiar* to dodatnie liczby całkowite typu `size_t`, a *i* i *c* są typu `int`.

`void clearerr (filePtr)`

Czyści znacznik końca pliku i wskaźniki błędu związane z plikiem opisywanym przez *filePtr*.

`int fclose (filePtr)`

Zamyka plik opisywany przez *filePtr*; zwraca 0, jeśli zamykanie się powiedzie, i `EOF` w razie błędu.

`int feof (filePtr)`

Jeśli dany plik doszedł do końca, zwraca wartość niezerową. W przeciwnym razie zwraca zero.

`int ferror (filePtr)`

Sprawdza błąd związany ze wskazanym plikiem; zwraca 0, jeśli błąd istnieje, i wartość niezerową w przeciwnym razie.

`int fflush (filePtr)`

Zapisuje wszystkie dane z wewnętrznych buforów do wskazanego pliku; zwraca 0 jako oznakę sukcesu i `EOF` jako oznakę błędu.

`int fgetc (filePtr)`

Zwraca następny znak z pliku *filePtr* lub `EOF`, jeśli natknie się na koniec pliku (pamiętać trzeba, że funkcja ta zwraca wartość typu `int`).

`int fgetpos (filePtr, fpos)`

Pobiera aktualne ustawienie położenia w pliku *filePtr*, zapisuje je w zmiennej *fpos* typu `fpos_t` (zdefiniowanego w pliku `<stdio.h>`). Funkcja `fgetpos` zwraca zero, jeśli wszystko przebiegnie bezbłędnie, i wartość niezerową w razie błędu. Zobacz też `fsetpos`.

`char *fgets (bufor, i, filePtr)`

Odczytuje ze wskazanego pliku znaki, aż odczyta ich *i*–1 albo odczyta znak nowego wiersza. Odczytane znaki są zapisywane w tablicy znakowej *bufor*. Jeśli odczytany zostanie znak nowego wiersza, zostanie także zapisany w tablicy. Jeśli zostanie odnaleziony koniec pliku lub wystąpi błąd, zwracana jest wartość `NULL`; w przeciwnym razie zwracany jest `bufor`.

`FILE *fopen (nazwaPliku, tryb)`

Otwiera wskazany plik w podanym trybie. Dopuszczalne są tryby `"r"` (odczytu), `"w"` (zapisu), `"a"` (dopisywania na końcu pliku), `"r+"` (odczytu i zapisu; zaczyna się od istniejącego pliku), `"w+"` (odczytu i zapisu; jeśli plik istniał, jest zamazywany) oraz `"a+"` (dopisywania). Jeśli plik jest otwarty w trybie dopisywania (`"a"` lub `"a+"`), niemożliwe jest nadpisywanie danych wcześniej istniejących w pliku.

W systemach rozróżniających pliki binarne i pliki tekstowe podczas korzystania z pliku binarnego trzeba dodać do trybu literę `"b"` (na przykład `"rb"`).

Jeśli wywołanie `fopen` się uda, zwracany jest wskaźnik struktury `FILE` obsługującej dany plik. W przeciwnym razie zwracany jest wskaźnik pusty.

`int fprintf (filePtr, format, arg1, arg2, ..., argn)`

Zapisuje podane parametry w pliku *filePtr*, korzysta przy tym z łańcucha formatującego *format*. Znaki formatujące są takie same jak w funkcji `printf` (zobacz rozdział 15.). Zwracana jest liczba zapisanych znaków. Ujemna wartość znaczy, że wystąpił błąd.

`int fputc (c, filePtr)`

Zapisuje wartość *c* (jako daną typu `unsigned char`) do pliku *filePtr*, zwraca *c*, jeśli się to uda, i `EOF` w razie błędu.

`int fputs (bufor, filePtr)`

Zapisuje w pliku znaki z tablicy *bufor*, aż znaleziony zostanie kończący znak null. Funkcja ta *nie dopisuje* automatycznie znaku nowego wiersza. W razie błędu funkcja zwraca `EOF`.

`size_t fread (bufor, rozmiar, n, filePtr)`

Odczytuje ze wskazanego pliku *n* elementów do *bufora*. Każdy element ma *rozmiar* bajtów danych. Na przykład wywołanie:

```
numread = fread (text, sizeof (char), 80, in_file);
```

odczytuje z pliku `in_file` 80 znaków, zapisuje je w tablicy `text`. Zwracana jest liczba odczytanych znaków.

`FILE *freopen (nazwaPliku, tryb, filePtr)`

Zamyka plik związany z *filePtr* i otwiera plik *nazwaPliku* w trybie *tryb* (zobacz funkcja `fopen`). Następnie otwierany jest plik związany z *filePtr*. Jeśli działanie `freopen` uda się, zwracane jest *filePtr*; w przeciwnym razie zwracany jest wskaźnik pusty. Funkcja `freopen` często służy do zmiany przypisania plików `stdin`, `stdout` i `stderr`, na przykład wywołanie:

```
if ( ( freopen ("inputData", "r", stdin) == NULL ) {
    ...
}
```

powoduje zmianę przypisania strumienia wejściowego na plik `"inputData"`, otwarty wcześniej w trybie do odczytu. Kolejne wywołania operacji I/O na standardowym wejściu dotyczą już pliku `inputData`.

`int fscanf (filePtr, format, arg1, arg2, ..., argn)`

Odczytuje dane z pliku *filePtr* zgodnie ze specyfikacją formatu *format*. Odczytywane wartości są zapisywane w kolejnych parametrach, z których wszystkie muszą być wskaźnikami. W parametrze *format* można używać takich samych znaków specjalnych jak w funkcji `scanf` (zobacz rozdział 15.). Funkcja `fscanf` zwraca liczbę odczytanych i przypisanych danych (przypisanie nie dotyczy parametrów `%n`), a w przypadku dotarcia do końca pliku przed przekształceniem pierwszej wartości zwracana jest wartość `EOF`.

`int fseek (filePtr, offset, mode)`

Ustawia podany plik tak, aby jego wskaźnik położenia znajdował się *offset* bajtów (wartość typu `long int`) od początku pliku, od dotychczasowego położenia lub od końca pliku; punkt odniesienia zależy od wartości parametru *mode*. Jeśli parametr ten ma wartość `SEEK_SET`, położenie jest określane względem początku pliku. Jeśli parametr *mode* ma wartość `SEEK_CUR`, położenie jest określane względem położenia dotychczasowego. W końcu, wartość `SEEK_END` powoduje pozycjonowanie względem końca pliku. Wartości `SEEK_SET`, `SEEK_CUR` i `SEEK_END` (liczby całkowite) zdefiniowano w pliku `<stdio.h>`.

W systemach rozróżniających pliki tekstowe i binarne działanie opcji `SEEK_END` może być nieobsługiwane w przypadku plików binarnych. Dla plików tekstowych wartość parametru *offset* musi być zerem lub musi być wartością zwróconą

przez wcześniejsze wywołanie funkcji `ftell`. W tym ostatnim wypadku parametr *mode* musi mieć wartość `SEEK_SET`.

Jeśli wywołanie `fseek` nie powiedzie się, zwracana jest wartość niezerowa.

`int fsetpos (filePtr, fpos)`

Ustawia bieżące położenie pliku *filePtr* na wartość *fpos* typu `fpos_t` (typ ten zdefiniowano w pliku nagłówkowym `<stdio.h>`). Jeśli wywołanie powiedzie się, zwraca zero; w przeciwnym razie zwraca wartość niezerową. Zobacz też `fgetpos`.

`long ftell (filePtr)`

Zwraca względną odległość (w bajtach) bieżącego położenia w pliku *filePtr*; w razie błędu zwraca `–1L`.

`size_t fwrite (bufor, rozmiar, n, filePtr)`

Zapisuje *n* elementów z *bufora* do wskazanego pliku. Każdy element ma *rozmiar* bajtów. Zwraca liczbę zapisanych elementów.

`int getc (filePtr)`

Odczytuje i zwraca następny znak ze wskazanego pliku. W razie błędu lub dojścia do końca pliku zwraca wartość `EOF`.

`int getchar (void)`

Odczytuje i zwraca następny znak z pliku `stdin`. W razie błędu lub dojścia do końca pliku zwraca `EOF`.

`char *gets (bufor)`

Odczytuje dane ze *stdin* do bufora, aż odczytany zostanie znak nowego wiersza. W *buforze nie jest zapisywany* znak nowego wiersza, za to jest dopisywany znak null. Jeśli pojawi się błąd lub nie zostaną odczytane żadne znaki, zwracany jest wskaźnik pusty. W przeciwnym razie zwracany jest *bufor*. Funkcję tę usunięto ze specyfikacji ANSI C11, ale można ją znaleźć w starszych programach, więc warto wiedzieć, jak działa.

`void perror (komunikat)`

Zapisuje do pliku `stderr` opis ostatniego błędu, podanego jako *komunikat*. Na przykład poniższy fragment kodu:

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

if ( (in = fopen ("data", "r")) == NULL ) {
    perror ("data file read");
    exit (EXIT_FAILURE);
}
```

w razie błędu wywołania funkcji `fopen` generuje komunikat błędu, ewentualnie podaje użytkownikowi dodatkowe informacje o zaistniałym błędzie.

`int printf (format, arg1, arg2, ..., argn)`

Zapisuje podane parametry do pliku `stdout` zgodnie z przekazanym łańcuchem znakowym *`format`* (zobacz rozdział 15.). Zwraca liczbę zapisanych znaków.

`int putc (c, filePtr)`

Zapisuje do wskazanego pliku parametr *`c`* jako `unsigned char`. Jeśli zapis się uda, zwraca *`c`*. W przeciwnym razie zwraca `EOF`.

`int putchar (c)`

Zapisuje do pliku `stdout` parametr *`c`* jako `unsigned char`. Jeśli zapis się uda, zwraca *`c`*. W przeciwnym razie zwraca `EOF`.

`int puts (bufor)`

Zapisuje w pliku znaki z tablicy *`bufor`* do pliku `stdout`, aż znaleziony zostanie kończący znak null. Automatycznie na koniec dopisywany jest znak nowego wiersza (czyli inaczej niż w funkcji `fputs`). W razie błędu funkcja zwraca `EOF`.

`int remove (nazwaPliku)`

Usuwa podany plik. W razie błędu zwraca wartość różną od zera.

`int rename (nazwaPliku1, nazwaPliku2)`

Zmienia nazwę pliku *`nazwaPliku1`* na *`nazwaPliku2`*. W przypadku niepowodzenia operacji zwraca wartość różną od zera.

`void rewind (filePtr)`

Zeruje bieżące położenie w podanym pliku.

`int scanf (format, arg1, arg2, ..., argn)`

Odczytuje dane z pliku `stdin` zgodnie ze specyfikacją formatu *`format`*. Odczytywane wartości są zapisywane w kolejnych parametrach, które muszą być wskaźnikami. W parametrze *`format`* można używać takich samych znaków specjalnych jak w funkcji `scanf` (zobacz rozdział 15.). Funkcja `scanf` zwraca liczbę odczytanych i przypisanych danych (przypisanie nie dotyczy parametrów `%n`), a w przypadku dotarcia do końca pliku przed przekształceniem pierwszej danej zwracana jest wartość `EOF`.

`FILE *tmpfile (void)`

Tworzy i otwiera tymczasowy plik binarny w trybie aktualizacji (`"r+b"`), w razie błędu zwraca `NULL`. Plik tymczasowy jest automatycznie usuwany

po zakończeniu działania programu (istnieje też funkcja `tmpnam`, która tworzy niepowtarzalne nazwy plików tymczasowych).

`int ungetc (c, filePtr)`

Wstawia znak z powrotem do pliku. Zwracane tak znaki nie są zapisywane w tym pliku, ale umieszczane w jego buforze. Następne wywołanie funkcji `fgetc` zwróci znak *c*. Funkcja `ungetc` tak może zwrócić tylko jeden znak, czyli przed ponownym wywołaniem tej funkcji musi być z pliku coś odczytane. Funkcja zwraca *c*, jeśli znak udało się „zwrócić", i `EOF` w przeciwnym wypadku.

Funkcje formatujące dane w pamięci

Funkcje `sprintf` i `sscanf` pozwalają przekształcać dane w pamięci. Ich działanie jest analogiczne do działania funkcji `fprintf` i `fscanf`, ale jako pierwszy parametr zamiast wskaźnika `FILE` używany jest wskaźnik łańcucha znakowego. Użycie tych funkcji wymaga włączenia do programu pliku nagłówkowego `<stdio.h>`.

`int sprintf (bufor, format, arg1, arg2, ..., argn)`

Zapisuje podane parametry do łańcucha znakowego *bufor* zgodnie z przekazanym łańcuchem *format* (zobacz rozdział 15.). Na koniec *bufora* automatycznie wstawiany jest znak null. Zwraca liczbę zapisanych znaków z pominięciem końcowego znaku null. Poniższy kod:

```
int  version = 2;
char fname[125];
    ...
sprintf (fname, "/usr/data%i/2015", version);
```

powoduje zapisanie w zmiennej `fname` łańcucha `"/usr/data2/2015"`.

`int sscanf (bufor, format, arg1, arg2, ..., argn)`

Odczytuje dane z łańcuch znakowego *bufor* zgodnie ze specyfikacją formatu *format*. Odczytywane wartości są zapisywane w kolejnych parametrach, które muszą być wskaźnikami (zobacz rozdział 15.). Funkcja `sscanf` zwraca liczbę odczytanych danych. Przykładowo kod:

```
char buffer[] = "July 16, 2014", month[10];
int  day, year;
    ...
sscanf (buffer, "%s %d, %d", month, &day, &year);
```

zapisuje w zmiennej `month` wartość `"July"`, w `day` liczbę 16, a w zmiennej `year` liczbę 2014. Kod:

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

```
if ( sscanf (argv[1], "%f", &fval) != 1 ) {
    fprintf (stderr, "Nieprawidłowa liczba: %s\n", argv[1]);
    exit (EXIT_FAILURE);
}
```

zamienia pierwszy parametr z wiersza poleceń (wskazywany przez `argv[1]`) na liczbę zmiennoprzecinkową i sprawdza wartość zwracaną przez `sscanf`, aby się upewnić, czy konwersja się udała (inne metody konwersji łańcuchów na liczby opiszemy za chwilę).

Konwersja łańcucha na liczbę

Poniższe funkcje konwertują łańcuchy znakowe na liczby. Aby ich użyć, trzeba do programu włączyć plik nagłówkowy `<stdlib.h>`:

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

W poniższych opisach *s* to wskaźnik łańcucha zakończonego znakiem null, *end* to wskaźnik znaku, a *base* to wartość typu `int`.

We wszystkich poniższych funkcjach pomijane są początkowe białe znaki, interpretacja łańcucha kończy się zaś po napotkaniu znaku niedozwolonego w typie danych, na który jest wykonywana konwersja.

`double atof (s)`

Konwertuje łańcuch wskazywany przez *s* na liczbę zmiennoprzecinkową, zwraca wynik konwersji.

`int atoi (s)`

Konwertuje łańcuch wskazywany przez *s* na liczbę typu `int`, zwraca wynik konwersji.

`int atol (s)`

Konwertuje łańcuch wskazywany przez *s* na liczbę typu `long int`, zwraca wynik konwersji.

`int atoll(s)`

Konwertuje łańcuch wskazywany przez *s* na liczbę typu `long long int`, zwraca wynik konwersji.

`double strtod (s, end)`

Konwertuje łańcuch *s* na wartość typu `double`, zwraca wynik. Jeśli *end* nie jest wskaźnikiem pustym, *end* wskazuje znak, na którym konwersja się skończyła.

Na przykład poniższy kod:

```
#include <stdlib.h>
    ...
char    buffer[] = "  123.456xyz", *end;
```

```
double  value;
    ...
value = strtod (buffer, &end);
```

powoduje przypisanie zmiennej `value` wartości `123.456`. Wskaźnik `end` jest tak ustawiany, że wskazuje znak z `buffer`, na którym zakończyła się konwersja, czyli w tym wypadku znak `'x'`.

`flaot strtof (`*`s`*`, `*`end`*`)`

Działa tak samo jak `strtod`, ale swój parametr konwertuje na typ `float`.

`long int strtol (`*`s`*`, `*`end`*`, `*`base`*`)`

Konwertuje łańcuch *`s`* na liczbę typu `long int` i zwraca wynik tej konwersji. Parametr *`base`* to podstawa systemu liczbowego, w jakim zapisana jest liczba. Jeśli parametr ten ma wartość 0, liczba może być zapisana dziesiętnie, ósemkowo (z wiodącym zerem) lub szesnastkowo (z wiodącym `0x` lub `0X`). Jeśli *`base`* ma wartość 16, wartość opcjonalnie może być poprzedzona wiodącym `0x` lub `0X`.

We wskaźniku *`end`* umieszczany jest adres znaku, który zakończył konwersję (jeśli wskaźnik ten nie jest pusty).

`long double strtold (`*`s`*`, `*`end`*`)`

Działa tak samo jak `strtod`, ale swój pierwszy parametr konwertuje na typ `long double`.

`long long int strtoll (`*`s`*`, `*`end`*`, `*`base`*`)`

Działa jak `strtol`, ale zwraca wartość typu `long long int`.

`unsigned long int strtoul (`*`s`*`, `*`end`*`, `*`base`*`)`

Konwertuje łańcuch *`s`* na wartość typu `unsigned long int`, zwraca wynik konwersji. Pozostałe parametry są interpretowane tak samo jak w funkcji `strtol`.

`unsigned long long int strtoull (`*`s`*`, `*`end`*`, `*`base`*`)`

Konwertuje łańcuch *`s`* na liczbę typu `unsigned long long int`, zwraca wynik. Pozostałe parametry są interpretowane tak samo jak w funkcji `strtol`.

Dynamiczna alokacja pamięci

Poniższe funkcje służą do dynamicznego alokowania i zwalniania pamięci. W każdej z nich *`n`* i *`rozmiar`* to liczby całkowite typu `size_t`, a *`pointer`* to wskaźnik `void`. Aby użyć tych funkcji, w programie trzeba dodać następujący wiersz:

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

`void *calloc (n, rozmiar)`

Alokuje ciągły obszar pamięci na `n` elementów, z których każdy ma *`rozmiar`* bajtów. Alokowana pamięć jest zerowana. Jeśli alokacja się powiedzie, funkcja zwraca wskaźnik do tej pamięci; w razie błędu zwracany jest pusty wskaźnik.

`void free (pointer)`

Zwalnia blok pamięci wskazywany przez *`pointer`*, wcześniej zaalokowany przez `calloc`, `malloc` lub `realloc`.

`void *malloc (rozmiar)`

Alokuje ciągły obszar pamięci wielkości *`rozmiar`* bajtów. Jeśli alokacja się powiedzie, funkcja zwraca wskaźnik do tej pamięci; w razie błędu zwracany jest pusty wskaźnik.

`void *realloc (pointer, rozmiar)`

Zmienia wielkość zaalokowanej uprzednio pamięci na *`rozmiar`* bajtów, zwraca wskaźnik nowego bloku (blok ten może zostać przeniesiony) lub wskaźnik pusty w razie błędu.

Funkcje matematyczne

Poniżej wyliczono funkcje matematyczne. Aby ich użyć, trzeba w programie umieścić następujący wiersz:

```
#include <math.h>
```

Standardowy plik nagłówkowy `<tgmath.h>` zawiera niezależne od typów makra, którymi można wywoływać funkcje matematyczne bez troszczenia się o typy parametrów. Na przykład w zależności od typu parametru i typu wartości zwracanej można użyć sześciu różnych obliczeń pierwiastka kwadratowego:

- `double sqrt (double x)`,
- `float sqrtf (float x)`,
- `long double sqrtl (long double x)`,
- `double complex csqrt (double complex x)`,
- `float complex csqrtf (float complex f)`,
- `long double complex csqrtl (long double complex x)`.

Można nie zawracać sobie głowy sześcioma funkcjami, wystarczy zamiast `<math.h>` i `<complex.h>` zastosować `<tgmath.h>` i można używać niezależnych od typu wywołań funkcji `sqrt`. Odpowiednie makro, zdefiniowane w `<tgmath.h>`, zapewnia wywołanie odpowiedniej funkcji.

Wracając do <math.h>, do sprawdzania poszczególnych właściwości liczb zmiennoprzecinkowych, przekazywanych jako parametry, możemy użyć następujących makr.

`int fpclassify (x)`

Klasyfikuje *x* jako nieliczbę (NaN, FP_NAN), wartość nieskończoną (FP_INFINITE), zwykłą liczbę (FP_NORMAL), małą wartość (FP_SUBNORMAL), zero (FP_ZERO) lub inną; dodatkowe kategorie zależą od używanej implementacji, ale wszystkie są zdefiniowane w <math.h> i zaczynają się od FP_.

`int isfin (x)`

Czy *x* to wartość skończona?

`int isinf (x)`

Czy *x* jest wartością nieskończoną?

`int isgreater (x, y)`

Czy $x > y$?

`int isgreaterequal (x, y)`

Czy $x \geq y$?

`int islessequal (x, y)`

Czy $x \leq y$?

`int islessgreater (x, y)`

Czy $x < y$ lub $x > y$?

`int isnan (x)`

Czy *x* jest wartością NaN? (NaN oznacza Not-a-Number, nieliczbę).

`int isnormal (x)`

Czy *x* jest normalną wartością?

`int isunordered (x, y)`

Czy dla *x* i *y* kolejność jest nieustalona? (Na przykład jedna z nich lub obie są wartościami NaN).

`int signbit (x)`

Czy znak *x* jest ujemny?

W dalszym ciągu *x*, *y* i *z* będą typu double, *r* to kąt wyrażony w radianach jako wartość double, a *n* to liczba typu int.

Więcej informacji o sposobie zgłaszania błędów przez opisywane funkcje można znaleźć w dokumentacji.

`double acos (x)`[1].

Zwraca wartość arcus cosinusa *x* jako kąt podawany w radianach, mieszczący się w zakresie [0, π]; *x* należy do zakresu [–1, 1].

`double acosh(x)`

Zwraca hiperboliczny arcus cosinus *x* dla $x \geq 1$.

`double asin (x)`

Zwraca arcus sinus *x* wyrażony jako kąt w radianach z zakresu [–π/2, π/2]; *x* należy do przedziału [–1, 1].

`double asinh (x)`

Zwraca hiperboliczny arcus sinus *x*.

`double atan (x)`

Zwraca arcus tangens *x* jako kąt w radianach z przedziału [–π/2, π/2].

`double atanh(x)`

Zwraca hiperboliczny arcus tangens *x*, $|x| \leq 1$.

`double atan2 (y, x)`

Zwraca arcus tangens *y*/*x* jako kąt w radianach z przedziału [–π, π].

`double ceil (x)`

Zwraca najmniejszą liczbę całkowitą większą lub równą *x*. Zwracana wartość jest jednak typu `double`!

`double copysign (x, y)`

Zwraca wartość co do modułu równą *x*, o znaku *y*.

`double cos (r)`

Zwraca cosinus *r*.

`double cosh (x)`

Zwraca cosinus hiperboliczny *x*.

---

[1] Biblioteka matematyczna zawiera wersje funkcji matematycznych pobierające i zwracające wartości typów `float`, `double` i `long double`. Tutaj omawiamy wersje `double`. Wersje `float` nazywają się tak samo, ale na końcu mają literę `f` (na przykład `acosf`). Wersje `long double` mają na końcu literę `l` (na przykład `acosl`).

`double erf (x)`
Wylicza i zwraca funkcję błędu *x*.

`double erfc (x)`
Wylicza i zwraca komplementarną funkcję błędu *x*.

`double exp (x)`
Zwraca e^x.

`double expm1 (x)`
Zwraca e^x-1.

`double fabs (x)`
Zwraca wartość bezwzględną *x*.

`double fdim (x, y)`
Zwraca $x-y$, jeśli $x > y$; w przeciwnym razie zwraca 0.

`double floor (x)`
Zwraca największą liczbę całkowitą mniejszą lub równą *x*. Zwracana wartość jest typu `double`!

`double fma (x, y, z)`
Zwraca $(x \times y) + z$.

`double fmax (x, y)`
Zwraca większą z liczb *x* i *y*.

`double fmin (x, y)`
Zwraca mniejszą z liczb *x* i *y*.

`double fmod (x, y)`
Zwraca zmiennoprzecinkową resztę z dzielenia *x* przez *y*. Znak wyniku jest taki jak znak *x*.

`double frexp (x, exp)`
Dzieli *x* na znormalizowany ułamek i potęgę dwójki. Zwraca ułamek z zakresu [1/2, 1] i zapisuje w zmiennej wskazywanej przez *exp* — wykładnik potęgowania. Jeśli *x* jest równe 0, zwracana wartość i wykładnik są zerami.

`int hypot (x, y)`
Zwraca pierwiastek kwadratowy sumy x^2+y^2.

`int ilogb (x)`
Wybiera wykładnik *x* jako liczbę całkowitą ze znakiem.

`double ldexp (x, n)`
Zwraca $x \cdot 2^n$.

`double lgamma (x)`
Zwraca logarytm naturalny wartości bezwzględnej gamma z *x*.

`double log (x)`
Zwraca logarytm naturalny *x*, $x \geq 0$.

`double logb (x)`
Zwraca wykładnik ze znakiem liczby *x*.

`double log1p (x)`
Zwraca logarytm naturalny $(x+1)$, $x \geq -1$.

`double log2 (x)`
Zwraca $\log_2 x$, $x \geq 0$.

`double log10 (x)`
Zwraca $\log_{10} x$, $x \geq 0$.

`long int lrint (x)`
Zwraca *x* zaokrąglone do najbliższej liczby całkowitej `long`.

`long long int llrint (x)`
Zwraca *x* zaokrąglone do najbliższej liczby całkowitej `long long`.

`long long int llround (x)`
Zwraca wartość *x* zaokrągloną do najbliższej liczby `long long int`.
Połówki zawsze są zaokrąglane w górę (zatem 0.5 zawsze jest zaokrąglane do 1).

`long int lround (x)`
Zwraca wartość *x* zaokrągloną do najbliższej liczby `long int`. Połówki zawsze są zaokrąglane w górę (zatem 0.5 zawsze jest zaokrąglane do 1).

`double modf (x, ipart)`
Pobiera z *x* część całkowitą i ułamkową. Część ułamkowa jest zwracana, część całkowita jest zaś zapisywana jako wartość typu `double` wskazywana przez *`ipart`*.

`double nan (s)`

Jeśli to możliwe, zwraca NaN zgodnie z zawartością określoną przez łańcuch wskazywany przez *s*.

`double nearbyint (x)`

Zwraca najbliższą *x* liczbę całkowitą w formie zmiennoprzecinkowej.

`double nextafter (x, y)`

Zwraca następną dającą się zapisać wartość *x* w stronę *y*.

`double nexttoward (x, ly)`

Zwraca następną dającą się zapisać wartość *x* w stronę *y*. Działa podobnie jak `nextafter`, ale drugi parametr jest typu `long double`.

`double pow (x, y)`

Zwraca wartość x^y. Jeśli *x* jest mniejsze od zera, *y* musi być liczbą całkowitą. Jeśli *x* jest równe zero, *y* musi być większe od zera.

`double remainder (x, y)`

Zwraca resztę z dzielenia *x* przez *y*.

`double remquo (x, y, quo)`

Zwraca resztę z dzielenia *x* przez *y*, iloraz zapisuje w zmiennej typu `int` wskazywanej przez *quo*.

`double rint (x)`

Zwraca najbliższą *x* liczbę całkowitą jako wartość zmiennoprzecinkową. Może powodować wyjątek liczb zmiennoprzecinkowych, jeśli wartość wyniku nie jest równa parametrowi *x*.

`double round (x)`

Zwraca wartość *x* zaokrągloną do najbliższej liczby całkowitej jako liczbę zmiennoprzecinkową. Połówki zawsze są zaokrąglane od zera (czyli 0.5 zawsze jest zaokrąglane do 1.0).

`double scalbln (x, n)`

Zwraca $x \cdot \texttt{FLT\_RADIX}^n$, gdzie *n* jest liczbą typu `long int`.

`double scalbn (x, n)`

Zwraca $x \cdot \texttt{FLT\_RADIX}^n$.

`double sin (r)`

Zwraca sinus *r*.

`double sinh (x)`

Zwraca sinus hiperboliczny *x*.

`double sqrt (x)`

Zwraca pierwiastek kwadratowy *x*, $x > 0$.

`double tan (r)`

Zwraca tangens *r*.

`double tanh (x)`

Zwraca tangens hiperboliczny *x*.

`double tgamma (x)`

Zwraca wartość gamma *x*.

`double trunc (x)`

Odrzuca z parametru *x* część ułamkową, ale wynik zwraca jako wartość typu `double`.

Arytmetyka zespolona

Plik nagłówkowy `<complex.h>` zawiera definicje różnych typów i funkcji do obsługi liczb zespolonych. Poniżej zestawiono kilka makr z tego pliku, a dalej funkcje wykonujące obliczenia zespolone.

Definicja	Znaczenie
`complex`	Zastępcza nazwa typu `_Complex`.
`_Complex_I`	Makro służące do podawania jednostki urojonej, na przykład `4 + 6.2 * _Complex_I` oznacza $4+6.2i$.
`imaginary`	Zastępcza nazwa typu `_Imaginary`; zdefiniowane tylko wtedy, gdy dana implementacja obsługuje urojone typy danych.
`_Imaginary_I`	Makro pozwalające określić urojoną część liczby zespolonej.

W wymienionych poniżej funkcjach *y* i *z* są typu `double complex`, *x* jest typu `double`, a *n* typu `int`.

`double complex cabs (z)`[2]

Zwraca wartość bezwzględną liczby zespolonej *z*.

---

[2] Biblioteka do obliczeń zespolonych zawiera wersje funkcji matematycznych pobierające i zwracające wartości typów `float complex`, `double complex` i `long double complex`. Tutaj omawiamy wersje `double`. Wersje `float` nazywają się tak samo, ale na końcu mają literę `f` (na przykład `cacosf`). Wersje `long double` mają na końcu literę `l` (na przykład `cacosl`).

`double complex cacos (z)`
Zwraca wartość zespolonego arcus cosinusa *z*.

`double complex cacosh(z)`
Zwraca zespolony hiperboliczny arcus cosinus *z*.

`double carg (z)`
Zwraca kąt liczby *z*.

`double complex casin (z)`
Zwraca zespolony arcus sinus *z*.

`double complex casinh (z)`
Zwraca zespolony hiperboliczny arcus sinus *z*.

`double complex catan (z)`
Zwraca zespolony arcus tangens *z*.

`double complex catanh(z)`
Zwraca zespolony hiperboliczny arcus tangens *z*.

`double complex ccos (z)`
Zwraca zespolony cosinus *z*.

`double complex ccosh (z)`
Zwraca zespolony cosinus hiperboliczny *z*.

`double complex cexp (z)`
Zwraca e^z.

`double cimag (z)`
Zwraca urojoną część *z*.

`double complex clog (z)`
Zwraca zespolony logarytm naturalny *z*.

`double complex conj (z)`
Zwraca liczbę zespoloną sprzężoną z *z* (odwrócony jest znak części urojonej).

`double complex cpow (y, z)`
Zwraca wartość y^z.

`double complex cproj (z)`

Zwraca rzutowanie *z* na sferę Riemanna.

`double complex creal (z)`

Zwraca rzeczywistą część *z*.

`double complex csin (z)`

Zwraca zespolony sinus *z*.

`double complex csinh (z)`

Zwraca zespolony sinus hiperboliczny *z*.

`double complex csqrt (z)`

Zwraca pierwiastek kwadratowy *z*.

`double complex ctan (z)`

Zwraca zespolony tangens *z*.

`double complex ctanh (z)`

Zwraca zespolony tangens hiperboliczny *z*.

Funkcje ogólnego przeznaczenia

Niektóre funkcje z biblioteki nie pasują do żadnej z wymienionych dotąd kategorii. Aby użyć tych funkcji, należy do programu włączyć plik nagłówkowy `<stdlib.h>`.

`int abs (n)`

Zwraca wartość bezwzględną przekazanego parametru *n* typu `int`.

`void exit (n)`

Kończy wykonywanie programu, zamyka wszystkie otwarte pliki i zwraca kod wyjścia podany w parametrze *n* typu `int`. Jako kodu można użyć zdefiniowanych w `<stdlib.h>` wartości `EXIT_SUCCESS` (wyjście bez błędu) i `EXIT_FAILURE` (wyjście z błędem).

Inne podobne procedury istniejące w bibliotece to `abort` i `atexit`.

`char *getenv (s)`

Zwraca wskaźnik do wartości zmiennej środowiskowej wskazywanej przez *s*. Jeśli taka zmienna nie istnieje, zwraca pusty wskaźnik. Funkcja ta jest zależna od konkretnego systemu, na przykład w systemie Unix do pobrania wartości zmiennej środowiskowej `HOME` do zmiennej programowej `homedir` można użyć kodu:

```
char  *homedir;
    ...
homedir = getenv ("HOME");
```

`long int labs (l)`

Zwraca wartość bezwzględną parametru *l* typu `long int`.

`long long int llabs (ll)`

Zwraca wartość bezwzględną parametru *ll* typu `long long int`.

`void qsort (arr, n, rozmiar, comp_fn)`

Sortuje dane w tablicy wskazywanej przez wskaźnik typu `void`, *arr*. W tablicy jest *n* elementów, każdy o wielkości *rozmiar*, przy czym *n* i *rozmiar* są typu `size_t`. Czwarty parametr to wskaźnik funkcji zwracającej wartość `int`, mającej dwa wskaźniki `void` jako parametry. Funkcja `qsort` wywołuje wskazywaną funkcję zawsze, kiedy musi porównać dwa elementy tablicy. Wywoływana tak funkcja otrzymuje jako parametry wskaźniki porównywanych elementów. Funkcja porównująca musi zwrócić wartość mniejszą od zera, zero lub wartość większą od zera, kiedy odpowiednio pierwszy element jest mniejszy, równy lub większy od elementu drugiego.

Oto przykład użycia funkcji `qsort` do posortowania tablicy zawierającej 1000 liczb typu `int`:

```
#include <stdlib.h>
    ...
int main (void)
{
    int data[1000], comp_ints (void *, void *);
        ...
    qsort (data, 1000, sizeof(int), comp_ints);
        ...
}

int comp_ints (void *p1, void *p2)
{
    int  i1 = * (int *) p1;
    int  i2 = * (int *) p2;
    return i1 – i2;
}
```

Inna, nieopisywana tutaj, funkcja `besearch` ma parametry podobne jak `qsort`, ale służy do wyszukiwania binarnego w uporządkowanej tablicy danych.

`int rand (void)`

Zwraca liczbę losową z zakresu [0, `RAND_MAX`], przy czym wartość `RAND_MAX` jest zdefiniowana w pliku nagłówkowym `<stdlib.h>` i ma wartość co najmniej `32767`. Zobacz też `srand`.

`void srand (`*`ziarno`*`)`

Inicjalizuje generator liczb losowych przekazanym *`ziarnem`* typu `unsigned int`.

`int system (`*`s`*`)`

Przekazuje polecenie z tablicy znakowej wskazywanej przez *s* do wykonania przez system operacyjny. Zwraca wartość pozyskaną z systemu. Jeśli *s* jest pustym wskaźnikiem, `system` zwraca niezerową wartość, jeśli można uruchomić interpreter poleceń systemowych.

Przykładowo, w systemie Unix wywołanie:

```
system ("mkdir /usr/tmp/data");
```

powoduje utworzenie katalogu */usr/tmp/data* (przy założeniu, że mamy do tego uprawnienia).

C

Kompilator gcc

W tym dodatku omówimy najczęściej stosowane opcje kompilatora *gcc*. W systemie Unix, aby poznać wszystkie dostępne opcje, należy wydać polecenie `man gcc`. Można też zajrzeć na witrynę *gcc*, *http://gcc.gnu.org/onlinedocs*, gdzie jest dostępna pełna dokumentacja *online*.

Omówione w tym rozdziale opcje wiersza poleceń dotyczą wersji 4.9 i nie obejmują rozszerzeń innych dostawców.

Ogólna postać polecenia

Ogólna postać polecenia *gcc* to:

```
gcc [opcje] plik [plik ...]
```

Elementy ujęte w nawiasy kwadratowe są opcjonalne.

Program *gcc* kompiluje wszystkie pliki z podanej listy. Normalnie kompilacja obejmuje działanie preprocesora, właściwą kompilację, asemblację i konsolidację. Korzystając z opcji wiersza poleceń, możemy zmienić przebieg tego procesu.

Rozszerzenia poszczególnych plików wejściowych decydują o sposobie ich interpretacji. Można tę funkcjonalność wyłączyć za pomocą opcji wiersza poleceń `–x` (szczegóły w dokumentacji *gcc*). Tabela C.1 zawiera listę typowych rozszerzeń.

Opcje wiersza poleceń

W tabeli C.2 zestawiono typowe opcje używane przy kompilacji programów napisanych w języku C.

Tabela C.1. Typowe rozszerzenia plików źródłowych

Rozszerzenie	Znaczenie
`.c`	plik źródłowy języka C
`.cc, .cpp`	plik źródłowy języka C++
`.h`	plik nagłówkowy
`.m`	plik źródłowy języka Objective-C
`.pl`	plik źródłowy języka Perl
`.o`	kod pośredni (plik po fazie kompilacji właściwej)

Tabela C.2. Najczęściej używane opcje programu gcc

Rozszerzenie	Znaczenie	Przykład
`—help`	Pokazuje zestawienie typowych opcji wiersza poleceń.	`gcc —help`
`–c`	Nie wykonuje konsolidacji programu, zachowuje pliki z kodem pośrednim (z rozszerzeniem *.o*).	`gcc –c enumerator.c`
`–dumpversion`	Pokazuje używaną wersję *gcc*.	`gcc –dumpversion`
`–g`	Włącza informacje dla programu uruchomieniowego, zwykle *gdb* (jeśli obsługiwane są różne takie programy, można użyć opcji `–ggdb`).	`gcc –g testprog.c –o testprog`
`–D id` `–D id=wartość`	W pierwszym wypadku definiuje identyfikator preprocesora `id` o wartości `1`. W drugim — definiuje identyfikator `id` o podanej `wartości`.	`gcc –D DEBUG=3 test.c`
`–E`	Uruchamiany jest tylko preprocesor, wyniki jego działania kierowane są na standardowe wyjście. Opcja ta jest przydatna do sprawdzania wyników działania preprocesora.	`gcc –E enumerator.c`
`–I dir`	Dodaje katalog *dir* do listy katalogów, w których szukane są pliki włączane. Podany katalog jest przeszukiwany przed katalogami standardowymi.	`gcc –I /users/steve/ include x.c`
`–lbiblioteka`	Szuka funkcji bibliotecznych w podanej `bibliotece`. Opcja ta powinna być podana po plikach wymagających funkcji z tej biblioteki. Program konsolidujący szuka w standardowych lokalizacjach biblioteki *libbiblioteka.a*.	`gcc mathfuncs.c –lm`

Tabela C.2. Najczęściej używane opcje programu gcc — ciąg dalszy

Rozszerzenie	Znaczenie	Przykład
`–L dir`	Dodaje katalog *dir* do listy katalogów, w których szukane są biblioteki. Katalog ten jest sprawdzany przed katalogami standardowymi.	`gcc –L /users/steve/lib x.c`
`–o plikexe`	Tworzy plik wykonywalny o nazwie *plikexe*.	`gcc dbtest.c –o dbtest`
`–Olevel`	Optymalizuje kod pod kątem szybkości działania zgodnie z poziomem `level`, gdzie `level` może mieć wartość 1, 2 lub 3. Jeśli nie zostanie podana wartość tego parametru, czyli użyta zostanie opcja –0, wartością domyślną jest 1. Większe liczby oznaczają wyższy stopień optymalizacji i mogą wydłużać czas kompilacji oraz ograniczać możliwości działania programów uruchomieniowych takich jak *gdb*.	`gcc –O3 m1.c m2.c –o mathfuncs`
`–std=standard`	Wskazuje użyty standard plików źródłowych C[1]. Aby skorzystać ze standardu ANSI C99 bez rozszerzeń GNU, należy podać wartość `c99`.	`gcc –std=c99 mod1.c mod2.c`
`–Wostrzeżenie`	Włącza ostrzeżenia podane w parametrze `ostrzeżenie`. Przydatne są opcje `all` (włącza opcjonalne ostrzeżenia, zwykle przydatne) oraz `error` (zamienia wszystkie ostrzeżenia w błędy, więc wymusza zrobienie poprawek).	`gcc –Werror mod1.c mod2.c`

[1] Obecnie domyślnym ustawieniem jest `gnu89`, co oznacza ANSI C90 z rozszerzeniami GNU. Po zaimplementowaniu wszystkich możliwości C99 zostanie to zmienione na `gnu99` (czyli ANSI C99 plus rozszerzenia GNU).

D

Typowe błędy

Poniżej zestawiono błędy najczęściej popełniane przez programistów pracujących w języku C. Kolejność jest dość przypadkowa. Mamy nadzieję, że znajomość tej listy pomoże uniknąć popełniania takich błędów w pisanych programach.

1. *Błędne umieszczenie średnika.*

   Przykład:

   ```
   if (j == 100 );
       j = 0;
   ```

   W powyższej instrukcji wartość zmiennej `j` zawsze będzie ustawiana na 0, a to z powodu średnika nieprawidłowo umieszczonego po nawiasie zamykającym. Średnik ten jest składniowo prawidłowy (oznacza instrukcję pustą), więc kompilator nie zgłasza żadnego błędu. Tego typu błąd często występuje też w przypadku instrukcji pętli `while` i `for`.

2. *Mylenie operatora = z operatorem ==.*

   Przykład:

   ```
   if ( a = 2 )
       printf ("Twój ruch.\n");
   ```

   Powyższa instrukcja jest prawidłowa; jej wynikiem jest przypisanie zmiennej `a` wartości 2, a następnie wywołanie funkcji `printf`. Funkcja ta będzie *zawsze* wywoływana, gdyż wartość wyrażenia w instrukcji `if` zawsze będzie niezerowa (będzie równa 2).

3. *Pominięcie deklaracji prototypu.*

   Przykład:

   ```
   result = squareRoot (2);
   ```

   Jeśli funkcja `squareRoot` zostanie zdefiniowana dalej w pliku, ale nie będzie jawnie zadeklarowana, kompilator założy, że funkcja ta zwraca wartość `int`. Co więcej, kompilator przekonwertuje parametry `float` na typ `double`, a `_Bool`, `char` i `short` na `int`. Nie będzie wykonywana żadna inna konwersja parametrów. Pamiętajmy, że zawsze najlepiej dołączyć deklaracje prototypów *wszystkich*

używanych funkcji (wywoływanych jawnie lub pośrednio) z pliku nagłówkowego, nawet jeśli funkcje są zdefiniowane wcześniej w programie.

4. *Mylenie priorytetów operatorów*.

   Przykład:

   ```
   while ( c = getchar () != EOF )
       ...
   if ( x & 0xF == y )
       ...
   ```

   W pierwszym przykładzie wartość zwracana przez `getchar` jest porównywana najpierw z wartością `EOF`. Wynika to stąd, że test na nierówność ma wyższy priorytet niż operator przypisania. Wobec tego zmiennej `c` przypisana jest wartość prawdy lub fałszu, będąca wynikiem poprzedniego testu. Jeśli odczytanym znakiem było `EOF`, zmienna `c` otrzymuje wartość 1. W przeciwnym razie zmienna ta otrzymuje wartość 0.

   W drugim przykładzie stała liczba całkowita `0xF` jest najpierw porównywana z `y`, gdyż operator porównania ma wyższy priorytet niż operatory bitowe. Wynikiem będzie wynik porównania (0 lub 1), który następnie zostanie pomnożony AND z wartością `x`.

5. *Mylenie stałej znakowej z łańcuchem znakowym*.

   W instrukcji:

   ```
   text = 'a';
   ```

   zmiennej `text` przypisywany jest pojedynczy znak. W instrukcji:

   ```
   text = "a";
   ```

   zmiennej `text` przypisywany jest wskaźnik łańcucha znakowego `"a"`. W pierwszym wypadku zmienna `text` jest deklarowana jako zmienna typu `char`, ale w drugim powinna być typu „wskaźnik na `char`”.

6. *Użycie nieprawidłowego indeksu tablicy*.

   Przykład:

   ```
   int a[100], i, sum = 0;
       ...
   for ( i = 1;  i <= 100;  ++i )
       sum += a[i];
   ```

   Poprawne indeksy tablicy mieszczą się w zakresie od 0 do liczby elementów pomniejszonej o 1. Wobec tego w powyższej pętli indeks jest nieprawidłowy, gdyż ostatnia dopuszczalna wartość to 99, a nie 100. Autor takiej instrukcji prawdopodobnie chciał zacząć działanie pętli od pierwszego elementu tablicy, ale ten ma numer 0, nie 1.

7. *Nieprzewidzenie dodatkowego miejsca na znak null w deklaracji łańcucha znakowego*.

   Deklarując tablice znakowe, musimy pamiętać, że ich wielkość powinna być taka, aby oprócz samego łańcucha zmieścił się także końcowy znak null, na przykład łańcuch `"witaj"` wymaga sześciu znaków, gdyż oprócz liter trzeba zapisać jeszcze null.

8. *Mylenie operatorów* `->` *oraz* `.` *przy odwoływaniu się do pól struktury*.

   Operator `.` używany jest do zmiennych strukturalnych, natomiast operator `->` do zmiennych *wskaźnikowych*. Jeśli zatem zmienna `x` jest strukturą, do odwołania się do jej pola `m` używamy zapisu `x.m`. Jeśli `x` jest wskaźnikiem struktury, trzeba już użyć zapisu `x->m`.

9. *Pomijanie w wywołaniu funkcji* `scanf` *symbolu ampersanda (*`&`*) przed zmiennymi niebędącymi wskaźnikami*.

   Przykład:

   ```
   int  number;
       ...
   scanf ("%i", number);
   ```

   Wszystkie parametry funkcji `scanf` znajdujące się po łańcuchu formatującym muszą być wskaźnikami.

10. *Użycie zmiennej wskaźnikowej przed jej zainicjalizowaniem*.

    Przykład:

    ```
    char  *char_pointer;
    *char_pointer = 'X';
    ```

    Operatora wyłuskania wobec zmiennej wskaźnikowej można używać dopiero *po ustawieniu* tej zmiennej tak, aby wskazywała jakieś miejsce w pamięci. W tym przykładzie zmienna `char_pointer` nie została ustawiona, więc pokazane przypisanie jest nieuprawnione.

11. *Pominięcie instrukcji* `break` *na końcu poszczególnych przypadków w instrukcji* `switch`.

    Jeśli na końcu któregoś przypadku (fraza `case`) nie pojawi się instrukcja `break`, wykonywany będzie kod należący do następnego przypadku.

12. *Zakończenie średnikiem dyrektywy preprocesora*.

    Tego typu błąd zwykle bierze się ze zwyczaju kończenia wszystkich instrukcji średnikami. Pamiętajmy jednak, że wszystko, co znajduje się w prawej części dyrektywy `#define` zostanie wstawione do programu. Jeśli zatem będziemy mieli definicję:

    ```
    #define    END_OF_DATA   999;
    ```

    to rozwinięcie wyrażenia:

```
if ( value == END_OF_DATA )
    ...
```

spowoduje błąd składniowy, gdyż kompilator otrzyma od preprocesora następujący kod:

```
if ( value == 999; )
    ...
```

13. *Pomijanie nawiasów wokół parametrów definicji makr.*

    Przykład:

```
#define  reciprocal(x)    1 / x
    ...
w = reciprocal (a + b);
```

    Powyższe przypisanie zostanie zinterpretowane nieprawidłowo jako:

```
w = 1 / a + b;
```

14. *Pominięcie zamknięcia nawiasu lub cudzysłowu w instrukcji.*

    Przykład:

```
total_earning = (cash + (investments * inv_interest) + (savings * sav_interest);
printf("Suma pieniędzy na Twoim koncie: %.2f", total_earning);
```

    W pierwszym wierszu zastosowano nawiasy, aby rozdzielić wizualnie części równania i sprawić, by kod był bardziej czytelny, ale takie działanie zawsze stwarza ryzyko popełnienia błędu (można zapomnieć zamknąć nawias albo wpisać za dużo zamknięć). Natomiast w drugim wierszu brakuje zamknięcia cudzysłowu w łańcuchu przekazywanym do funkcji `printf()`. W obu przypadkach kompilator zgłosi błąd, chociaż czasami może wskazać inne miejsce jego wystąpienia niż rzeczywiste. Jest to spowodowane tym, że kompilator może wykorzystać nawias lub cudzysłów z dalszego wiersza kodu.

15. *Niedołączenie nagłówka zawierającego definicję funkcji z biblioteki C wykorzystywanej w programie.*

    Przykład:

```
double answer = sqrt(value1);
```

    Jeśli na początku programu zawierającego powyższy wiersz nie ma dyrektywy `#include` dołączającej plik `<math.h>`, to wywołanie funkcji `sqrt()` spowoduje błąd braku definicji tej funkcji.

16. *Umieszczenie w dyrektywie* `#define` *spacji między nazwą makra a jego listą parametrów.*

    Przykład:

```
#define MIN (a,b)   ( ( (a) < (b) ) ? (a) : (b) )
```

    Powyższa definicja jest niepoprawna, gdyż pierwszą spację po definiowanej nazwie preprocesor uzna za początek podstawianego tekstu. Wobec tego, jeśli w programie znajdzie się wywołanie:

```
minVal = MIN (val1, val2)l
```

zostanie ono rozwinięte jako:

```
minVal = (a,b)  ( ( (a) < (b) ) ? (a) : (b) )(val1,val2);
```

a przecież nie o to chodziło.

17. *Użycie w wywołaniu makra wyrażenia dającego efekty uboczne.*

    Przykład:

    ```
    #define SQARE(x)   (x) * (x)
        ...
    w = SQUARE (++v);
    ```

    Wywołanie makra `SQUARE` spowoduje *dwukrotne* zwiększenie wartości zmiennej `v`, gdyż powyższe wywołanie tego makra zostanie zinterpretowane jako:

    ```
    w = (++v) * (++v);
    ```

E

Zasoby

Dodatek ten zawiera listę wybranych zasobów, w których można znaleźć dalsze informacje. Część informacji jest dostępna w sieci, część w formie książkowej.

Język programowania C

Język C ma już około 40 lat, więc nie brakuje o nim informacji. Oto kilka wybranych źródeł.

Książki

Brian W. Kernighan, Dennis M. Ritchie, *Język ANSI C*, Wydawnictwa Naukowe i Techniczne, Warszawa 1998.

Książka ta od zawsze pełniła rolę biblii języka C i stanowiła swoisty punkt odniesienia. Jest to pierwsza książka, jaką napisano o języku C. Jej współautorem jest Dennis Ritchie, twórca C. Książka ta ma już wprawdzie ponad 25 lat, ale jej drugie, czyli najnowsze, wydanie wciąż jest uznawane za podstawę.

Samuel P. III Harbison, Guy L. Steele Jr., *C: A Reference Manual, 5th Ed.*, Prentice Hall, Inc., Englewood-Cliffs 2002.

Następna doskonała książka dla programistów języka C.

P.J. Plauger, *The Standard C Library*, Prentice Hall, Inc., Englewood-Cliffs 1992.

W tej książce omówiono bibliotekę standardową C, ale jak wynika z daty jej publikacji, nie obejmuje ona dodatków związanych ze standardem ANSI C99 (takich jak biblioteka do obsługi liczb zespolonych).

Witryny WWW

http://www.ansi.org/

Witryna ANSI. Tutaj można nabyć oficjalną specyfikację ANSI C. Aby znaleźć specyfikacje ANSI C11, w oknie wyszukiwarki należy wpisać 9899:2011.

www.opengroup.org/onlinepubs/007904975/idx/index.html
Doskonała sieciowa dokumentacja funkcji bibliotecznych (obejmuje także funkcje C spoza standardu ANSI).

Grupa dyskusyjna

```
comp.lang.c
```

Jest to grupa poświęcona językowi C. Można tam zadawać pytania oraz pomagać innym po nabyciu pewnego doświadczenia. Warto też obserwować toczące się tam dyskusje. Dobrą metodą uzyskania dostępu do tej grupy jest użycie witryny *http://groups.google.com*.

Kompilatory C i zintegrowane środowiska programistyczne

Oto lista witryn, z których można pobrać lub na których można kupić kompilatory C i środowiska programistyczne, a także pozyskać dokumentację dostępną *online*.

gcc

http://gcc.gnu.org/
Kompilator C tworzony przez Free Software Foundation (FSF) — *gcc*. Można go za darmo pobrać ze wskazanej witryny.

MinGW

www.mingw.org
Osoby, które chcą pisać programy w C do środowiska Windows, mogą z tej witryny pobrać kompilator GNU *gcc*. Warto też rozważyć pobranie prostego w użyciu środowiska MSYS.

CygWin

http://www.cygwin.com/
CygWin oferuje podobne do systemu Linux środowisko programistyczne działające w Windows. Środowisko to jest dostępne za darmo.

Visual Studio

http://msdn.microsoft.com/vstudio
Visual Studio to środowisko IDE firmy Microsoft, które pozwala tworzyć aplikacje w wielu różnych językach programowania.

CodeWarrior

www.freescale.com/webapp/sps/site/homepage.jsp?code=CW_HOME
Kiedyś środowisko CodeWarrior było własnością Metrowerks, ale przeszło w ręce firmy Freescale. Zawiera ono profesjonalne narzędzia programistyczne i jest dostępne dla wielu systemów operacyjnych, w tym Linux, Mac OS X, Solaris i Windows.

Code::Blocks

www.codeblocks.org
Code::Blocks to bezpłatne środowisko programistyczne do tworzenia programów w jdęzykach C, C++ i Fortran na różnych platformach, w tym Windows, Linux i Mac.

Różne

Poniżej zestawiono zasoby, które pozwalają dowiedzieć się więcej o programowaniu obiektowym i o narzędziach programistycznych.

Programowanie obiektowe

Timothy Budd, *An Introduction to Object-Oriented Programming, 3rd Ed.*, Addison-Wesley Publishing Company, Boston 2001.

Książka uważana za klasyeczny podręcznik programowania obiektowego.

Język C++

Stephen Prata, *C++ Primer Plus, 6th Ed.*, Addison-Wesley, Indianopolis 2011.
Podręczniki tego autora otrzymują bardzo dobre noty; ten jest poświęcony językowi C++.
Bjarne Stroustrup, *Język C++. Kompendium wiedzy*, Helion, Gliwice 2014.
Najnowsze wydanie klasycznego tekstu samego twórcy C++.

Język C#

Charles Petzold, *Programming in the Key of C#*, Microsoft Press, Redmond 2003.
Książka uznawana za najlepszy podręcznik dla początkujących w C#.
Jesse Liberty, *Programming C# 3.0,* O'Reilly & Associates, Cambridge 2008.

Dobre wprowadzenie do języka C#, przeznaczone dla bardziej doświadczonych programistów.

Joseph Albahari, Ben Albahari, *C# 5.0 in a Nutshell: The Definitive Reference*, O'Reilly Media, Sebastopol 2012.

Doskonała publikacja, zwłaszcza dla znających podstawy języka.

Język Objective-C

Stephen Kochan, *Objective-C. Vademecum profesjonalisty. Wydanie III*, Helion, Gliwice 2012.

Wprowadzenie do języka Objective-C mojego autorstwa, także dla osób nieznających C ani niemających doświadczenia w programowaniu obiektowym.

https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html

Nie jest to książka w typowym tego słowa znaczeniu, tylko dokument internetowy zawierający oficjalne wprowadzenie firmy Apple do języka programowania Objective-C. Opisano w nim wiele elementów tego języka oraz podano przykłady ich użycia.

Narzędzia programistyczne

www.gnu.org/manual/manual.html

Tutaj znaleźć można mnóstwo przydatnych podręczników, w tym podręczniki dotyczące *cvs*, *gdb*, *make* i innych narzędzi systemu Unix działających w trybie wiersza poleceń.

Skorowidz

E

F

G

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z