K.N. King

Helion

Język C

Nowoczesne programowanie

Język C żyje i ma się dobrze
Sprawdź, co nowego w wersji C99!

- Jak wygląda proces standaryzacji języka?
- Jak komentować kod?
- Jak przygotować projekt programu?

Wydanie II

Język C. Nowoczesne programowanie. Wydanie II

Autor: K. N. King
Tłumaczenie: Przemysław Szeremiota
ISBN: 978-83-246-2805-6
Tytuł oryginału: C Programming: A Modern Approach, 2nd Edition
Format: B5, stron: 928

Język C żyje i ma się dobrze. Sprawdź, co nowego w wersji C99!

- Jak wygląda proces standaryzacji języka?
- Jak komentować kod?
- Jak przygotować projekt programu?

Język C należy do nielicznej grupy języków, które sprawdzają się w środowiskach produkcyjnych, a jednocześnie nadają się do nauki programowania na uczelniach wyższych. Dzięki logicznej i przejrzystej składni, jasno określonym zasadom wykorzystania oraz ogromnym możliwościom język ten pomimo swojego wieku cieszy się popularnością i uznaniem. Nawet dziś, kiedy na rynku panują niepodzielnie Java oraz .NET, język C znalazł swoją niszę i świetnie ją wypełnia. Na tym polu żaden współczesny język nie ma z nim żadnych szans!

Kolejne wydanie książki rozszerzono między innymi o elementy zawarte w specyfikacji oznaczonej numerem C99 (ISO 9899:1999). Co jeszcze wyróżnia tę książkę? Jej pierwsze wydanie było wykorzystywane na kursach programowania prowadzonych przez 225 uczelni. Dzięki temu zaliczana jest ona do najbardziej znaczących wydawnictw dotyczących języka C. Wydanie drugie powiela zalety pierwszego, a dodatkowo zostało rozbudowane o jeszcze większą liczbę przykładów, pytań, ćwiczeń i zadań programistycznych.

W trakcie pasjonującej lektury – zgadza się, K.N. King potrafi w ten sposób pisać o swoim ulubionym języku – poznasz wszystkie aspekty programowania w języku C, począwszy od jego historii, poprzez fundamentalne pojęcia funkcji, zmiennych, a skończywszy na zarządzaniu pamięcią oraz wykorzystaniu wskaźników. „Język C. Nowoczesne programowanie. Wydanie II" to obowiązkowa pozycja dla każdego studenta poznającego tajniki tego języka. Programiści znający język C niewątpliwie docenią kunszt autora, a książka znajdzie zastosowanie jako przekrojowy przewodnik – taka pozycja powinna być na półce każdego programisty!

Poznaj język C, korzystając z uznanego podręcznika!

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

SPIS TREŚCI

3 Formatowanie wejścia-wyjścia

W poszukiwaniu nieosiągalnego na przeszkodzie staje tylko prostota.

Do najczęściej wykorzystywanych funkcji bibliotecznych języka C należą `printf` i `scanf`, służące do obsługi formatowanego wejścia i wyjścia programu. W tym rozdziale przekonasz się o możliwościach tych funkcji, ale też o koniecznej ostrożności w ich stosowaniu. W podrozdziale 3.1 zajmiemy się funkcją `printf`. Głównym zagadnieniem podrozdziału 3.2 będzie funkcja `scanf`. W żadnym z podrozdziałów nie zgłębimy jednak wszystkich detali — niektóre będą musiały poczekać do rozdziału 22.

3.1. Funkcja printf

Funkcja `printf` służy do wypisywania na wyjściu programu zawartości ciągu znaków, określanego mianem **ciągu formatującego**, który może zawierać symbole zastępcze dla wartości zmiennych wstawianych do ciągu wypisywanego. W wywołaniu funkcji `printf` musi się znajdować ciąg formatujący, uzupełniony wartościami, które mają być podstawione w odpowiednie miejsca ciągu na wyjściu programu:

```
printf(ciąg-formatujący, wyrażenie1, wyrażenie1, …);
```

Wartości przekazywane do podstawienia do ciągu formatującego mogą być stałymi, zmiennymi albo całymi wyrażeniami. Nie istnieje ograniczenie liczby wartości wypisywanych w ramach pojedynczego wywołania funkcji `printf`.

Ciąg formatujący może zawierać zarówno zwyczajne znaki drukowalne, jak i tak zwane **specyfikatory konwersji**, rozpoczynające się od znaku `%`. Specyfikator konwersji to symbol zastępczy reprezentujący wartość, która ma zostać wstawiona w dane miejsce ciągu formatującego, wraz z opisem sposobu wypisania wartości. Informacje znajdujące się za znakiem `%` określają sposób konwersji

przekazanej wartości z jej reprezentacji wewnętrznej (binarnej) na reprezentację drukowaną (znakową) — stąd pojęcie „specyfikatora konwersji". Na przykład specyfikator konwersji `%d` mówi, że `printf` ma zamienić przekazaną wartość typu `int` z jej reprezentacji binarnej na ciąg znaków kolejnych cyfr. Podobnie specyfikator `%f` nakazuje zamianę wartości zmiennoprzecinkowej (typu `float`) na znakową.

Zwyczajne znaki zawarte w ciągu formatującym są wypisywane bez modyfikacji. Specyfikatory konwersji są natomiast zastępowane przez znakowe reprezentacje przekazanych wartości. Weźmy następujący przykład:

```
int i, j;
float x, y;

i = 10;
j = 20;
x = 43.2892f;
y = 5527.0f;

printf("i = %d, j = %d, x = %f, y = %f\n", i, j, x, y);
```

Takie wywołanie funkcji `printf` spowoduje wypisanie na wyjściu:

```
i = 10, j = 20, x = 43.289200, y = 5527.000000
```

Zwyczajne znaki w ciągu formatującym zostały po prostu skopiowane na wyjście. Cztery specyfikatory konwersji zostały zaś zastąpione przez odpowiednio reprezentowane wartości zmiennych `i`, `j`, `x` i `y` (w tej kolejności).

Specyfikatory konwersji

Specyfikatory konwersji pozwalają programistom zachować dużą dozę kontroli nad wyglądem (formatem) wypisywanych ciągów i wartości. Z drugiej strony bywają skomplikowane i trudne do ogarnięcia. Istotnie, szczegółowe opisywanie specyfikatorów konwersji byłoby na tym wstępnym etapie omówienia przedwczesne. Zapoznamy się więc tylko z najważniejszymi cechami i możliwościami dawanymi przez specyfikatory.

W rozdziale 2. zauważyliśmy, że specyfikator konwersji może zawierać informacje sterujące formatowaniem wartości. W szczególności zastosowaliśmy specyfikator `%.1f`, aby ograniczyć liczbę wypisywanych cyfr po przecinku w wartości typu `float`. Ogólniej rzecz biorąc, specyfikator konwersji może przyjąć postać *%m.pX* albo *%-m.pX*, gdzie *m* i *p* to stałe całkowite, a *X* to litera. Wartości *m* i *p* są opcjonalne. W przypadku nieobecności *p* nie stosuje się również kropki oddzielającej *m* od *p*. W specyfikatorze konwersji `%10.2f` *m* wynosi `10`, *p* wynosi `2`, a *X* to `f`. W specyfikatorze `%10f` *m* wynosi `10`, *p* (wraz z kropką) zostało pominięte, a *X* to `f`. Za to w `%.2f` *m* zostało pominięte, a *p* wynosi `2`.

Minimalna szerokość pola *m* określa minimalną liczbę znaków, jaka zostanie wypisana na wyjściu przy wypisywaniu wartości. Jeśli wypisywana wartość jest w reprezentacji znakowej krótsza niż *m* znaków, zostanie wyrównana do minimalnej szerokości pola i do prawej strony pola (innymi słowy, przed właściwą wartość wstawiona będzie odpowiednia liczba spacji). Na przykład specyfikator `%4d`

Standard nie wymaga od kompilatorów języka C sprawdzania, czy liczba specyfikatorów konwersji określona w ciągu formatującym odpowiada liczbie przekazanych wartości. Poniższe wywołanie funkcji `printf` posiada więcej specyfikatorów konwersji niż wartości do wypisania:

```
printf("%d %d\n", i);      /*** ŹLE ***/
```

Funkcja `printf` wypisze poprawnie wartość zmiennej `i`, a następnie wypisze drugą — oczekiwaną, ale nieokreśloną — wartość liczbową. Podobnie kłopotliwy jest nadmiar wartości w stosunku do specyfikatorów konwersji:

```
printf("%d\n", i, j);      /*** ŹLE ***/
```

W tym przypadku funkcja `printf` wypisze jedynie wartość `i`, a wartość `j` zostanie pominięta.

Kompilatory nie są też zobligowane do sprawdzania, czy specyfikatory konwersji odpowiadają typom przekazywanych wartości. Jeśli programista zastosuje nieodpowiednie specyfikatory, program może wypisać na wyjściu kompletne bzdury. Weźmy dla przykładu wywołanie funkcji `printf`, w którym zmienna `i` typu `int` i zmienna `x` typu `float` zostaną przekazane w złej kolejności:

```
printf("%f %d\n", i, x);      /*** ŹLE ***/
```

Ponieważ funkcja `printf` realizuje wytyczne z ciągu formatującego, wypisze na wyjściu wartość `float`, a za nią wartość `int`. Niestety, obie wartości jako źle zinterpretowane będą niepoprawne.

spowoduje wypisanie wartości `123` jako `·123` (w całym bieżącym rozdziale w miejsce niewidocznych spacji ilustrujących formatowanie wartości będą wstawiane znaki `·`). Z kolei kiedy wypisywana wartość będzie dłuższa niż *m* znaków, pole zostanie automatycznie poszerzone do szerokości potrzebnej do zmieszczenia wartości: specyfikator `%4d` dla wartości `12345` spowoduje wypisanie na wyjściu `12345` — bez ucinania cyfr. Znak minusa przed *m* wymusza wyrównanie wartości w polu do lewej strony: specyfikator `%4d` dla wartości `123` da na wyjściu `123·`.

Znaczenie **specyfikatora precyzji** *p* jest trudniejsze do opisania, ponieważ jego działanie jest zależne od *X*, czyli właściwego **specyfikatora konwersji**. *X* określa rodzaj konwersji do przeprowadzenia przed wypisaniem wartości. Do najpopularniejszych konwersji wartości liczbowych należą:

PiO

- `d` — konwersja wartości całkowitej do postaci dziesiętnej. W takim układzie *p* oznacza minimalną liczbę cyfr do wypisania (w razie potrzeby przed wypisywaną liczbą dopisywane są zera). Przy braku *p* uznaje się, że ma ono wartość 1 (innymi słowy, `%d` jest tym samym co `%.1d`).
- `e` — konwersja wartości zmiennoprzecinkowej do postaci wykładnikowej (w tzw. notacji naukowej). W takim układzie *p* określa liczbę cyfr do wypisania po przecinku dziesiętnym (wartość domyślna to 6). Dla *p* równego `0` część po przecinku nie jest wyświetlana w ogóle.
- `f` — konwersja wartości zmiennoprzecinkowej do postaci dziesiętnej, bez wykładnika. W tym układzie *p* ma takie samo znaczenie jak przy konwersji `e`.

- g — konwersja wartości zmiennoprzecinkowej do postaci wykładnikowej albo dziesiętnej, zależnie od rozmiaru liczby. W tym układzie *p* oznacza maksymalną liczbę cyfr znaczących (*nie* cyfr po przecinku) do wypisania. Inaczej niż przy konwersji f, konwersja g nie będzie wypisywała zer po prawej stronie wartości. Co więcej, jeśli konwertowana wartość nie ma cyfr po przecinku, konwersja g nie wypisze także symbolu przecinka.

Specyfikator konwersji g jest przydatny zwłaszcza do wypisywania wartości, dla których rozmiar reprezentacji nie da się ustalić na etapie pisania programu, oraz wartości z bardzo szerokich dziedzin. Konwersja g dla niezbyt wielkich i niezbyt małych wartości będzie owocowała zapisem dziesiętnym. Dla wartości bardzo małych i bardzo dużych zastosowany będzie zapis wykładnikowy, wymagający mniejszej liczby znaków.

specyfikatory dla wartości całkowitych ➤ 7.1
specyfikatory dla wartości zmiennoprzecinkowych ➤ 7.2
specyfikatory dla wartości znakowych ➤ 7.3
specyfikatory dla ciągów znaków ➤ 13.3

%d, %e, %f i %g to bynajmniej nie wszystkie specyfikatory konwersji. Pozostałe będą stopniowo wprowadzane przy okazji omawiania kolejnych zagadnień. Pełna lista specyfikatorów wraz z objaśnieniem ich znaczenia i działania znajduje się w podrozdziale 22.3.

PROGRAM

Wykorzystanie printf do formatowania liczb

Poniższy program ilustruje sposób wykorzystania funkcji printf do wypisywania wartości całkowitych i zmiennoprzecinkowych w rozmaitych formatach:

tprintf.c

```
/* Wypisuje wartości int i float w różnych formatach */

#include <stdio.h>

int main(void)
{
  int i;
  float x;

  i = 40;
  x = 839.21f;

  printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
  printf("|%10.3f|%10.3e|%-10g|\n", x, x, x);

  return 0;
}
```

Znaki | w ciągach formatujących dla funkcji printf służą jedynie do rozdzielenia wypisywanych wartości i zilustrowania szerokości pól, w których są wypisywane. W przeciwieństwie do znaków % i \ znak | nie ma specjalnego znaczenia dla funkcji printf. Program wypisuje na wyjściu coś takiego:

```
|40|   40|40   |  040|
|  839.210| 8.392e+02|839.21    |
```

Spróbujmy przeanalizować znaczenie i działanie specyfikatorów konwersji wykorzystanych w tym programie:

- `%d` — wypisanie `i` w zapisie dziesiętnym przy jak najmniejszej liczbie znaków.
- `%5d` — wypisanie wartości `i` w zapisie dziesiętnym w polu o szerokości co najmniej 5 znaków. Ponieważ właściwa wartość `i` ma zaledwie dwa znaki, pole jest wypełniane trzema spacjami od lewej strony.
- `%-5d` — wypisanie wartości `i` w zapisie dziesiętnym w polu o szerokości co najmniej 5 znaków. Ponieważ właściwa wartość `i` ma zaledwie dwa znaki, pole jest wypełniane trzema spacjami po prawej stronie wartości (wartość `i` jest więc wyrównana w pięcioznakowym polu do lewej strony).
- `%5.3d` — wypisanie wartości `i` w zapisie dziesiętnym w polu o szerokości co najmniej 5 znaków i przy reprezentowaniu wartości co najmniej trzema cyframi. Ponieważ `i` ma tylko dwie cyfry, przed właściwą wartością wstawiana jest pojedyncza cyfra zero. Wynikowa trzyznakowa wartość jest wypisywana w polu o szerokości 5 znaków, a więc jest poprzedzona dwoma spacjami (wartość `i` jest wyrównana do prawej strony pola).
- `%10.3f` — wypisanie wartości `x` w zapisie dziesiętnym z przecinkiem w polu o szerokości co najmniej 10 znaków, z trzema cyframi po przecinku. Ponieważ wartość `x` zajmuje jedynie 7 znaków (trzy przed przecinkiem i trzy po przecinku oraz sam przecinek), jest uzupełniana trzema spacjami z lewej strony.
- `%10.3e` — wypisanie wartości `x` w zapisie wykładnikowym w polu o szerokości co najmniej 10 znaków, z trzema cyframi po przecinku. Tak zapisana wartość `x` zajmuje 9 znaków, więc jest uzupełniona spacją z lewej strony.
- `%-10g` — wypisanie wartości `x` w zapisie wykładnikowym albo dziesiętnym z przecinkiem w polu o szerokości co najmniej 10 znaków, z trzema cyframi po przecinku. W naszym przypadku wartość `x` została zamieniona na dziesiętną z przecinkiem. Obecność znaku – w specyfikatorze konwersji wymusza wyrównanie wartości do lewej strony pola czterema spacjami za właściwą wartością.

Znaki sterujące

Symbol `\n`, wykorzystywany już w poprzednich ciągach formatujących, to przykład tak zwanego *znaku sterującego* (ang. *escape sequence*). Znaki sterujące pozwalają na osadzanie w ciągach znaków, które zapisane inaczej byłyby kłopotliwe dla kompilatora. Dotyczy to przede wszystkim znaków niedrukowalnych terminali znakowych oraz znaków posiadających specjalne znaczenie dla samego kompilatora (jak `"`). Kompletną listę znaków sterujących zamieścimy później, na razie wystarczy wykaz najważniejszych:

znaki sterujące ➤ 7.3

`\a` sygnał dzwonka,
`\b` kasowanie poprzedniego znaku (ang. *backspace*),
`\n` nowy wiersz,
`\t` tabulator poziomy.

Takie symbole w ciągu formatującym reprezentują czynności do wykonania w czasie wypisywania ciągu na wyjściu programu. Otóż wypisanie `\a` spowoduje na większości maszyn wygenerowanie dźwięku brzęczyka terminala; wypisanie `\b`

PiO

cofnie kursor o jedną pozycję; wypisanie `\n` przesunie kursor do nowego wiersza; wypisanie `\t` przesunie kursor do następnej pozycji tabulatora.

Ciąg formatujący może zawierać dowolną liczbę znaków sterujących. Spójrzmy na kolejny przykład wywołania `printf` z ciągiem formatującym zawierającym łącznie sześć znaków sterujących:

```
printf("Towar\tCena\tData\n\tjed.\tzakupu");
```

Wykonanie tej instrukcji spowoduje wypisanie na wyjściu dwóch wierszy:

```
Towar  Cena   Data
       jed.   zakupu
```

Innym popularnym znakiem sterującym jest `\"`, który reprezentuje w ciągu znak podwójnego cudzysłowu `"`. Ponieważ sam znak `"` oznacza początek albo koniec literału napisowego, nie może w takiej postaci pojawić się wewnątrz literału — musi zostać oznaczony jako znak sterujący znakiem ukośnika. Oto przykład:

```
printf("\"Ahoj!\"");
```

Taka instrukcja spowoduje wypisanie na wyjściu komunikatu:

```
"Ahoj!"
```

Podobna sytuacja dotyczy znaku lewego ukośnika. Jeśli zechcemy umieścić taki znak w wypisywanym ciągu, nie możemy go wstawić wprost do literału napisowego, bo kompilator zakłada, że znak ukośnika jest zapowiedzią znaku sterującego. Aby wypisać znak `\`, trzeba go również poprzedzić znakiem `\`, a więc wstawić do ciągu parę `\\`:

```
printf("\\");       /* wypisuje na wyjściu pojedynczy znak \ */
```

3.2. Funkcja scanf

Tak jak funkcja `printf` służy do formatowania wyjścia programu, tak funkcja `scanf` obsługuje formatowane wejście. Ciąg formatujący dla funkcji `scanf` również może zawierać zwyczajne znaki i specyfikatory konwersji. Konwersje obsługiwane przez funkcję `scanf` pokrywają się z grubsza z konwersjami funkcji `printf`.

W wielu przypadkach ciąg formatujący funkcji `scanf` zawiera wyłącznie specyfikatory konwersji, jak w poniższym przykładzie:

```
int i, j;
float x, y;

scanf("%d%d%f%f", &i, &j, &x, &y);
```

Załóżmy, że użytkownik wprowadza na wejście programu następujący wiersz:

```
1 -20 .3 -4.0e3
```

Funkcja `scanf` wczyta taki wiersz i rozpocznie stosowanie specyfikatorów konwersji i podstawianie uzyskanych wartości pod zmienne przekazane w wywołaniu: `1` pod `i`, `-20` pod `j`, `0.3` pod `x` i `-4000.0` pod `y`. Takie „upakowane" ciągi sterujące są dla funkcji `scanf` typowe. W przypadku funkcji `printf` częściej stosuje się rozmaite „dekoracje" i napisy objaśniające, otaczające wyprowadzane wartości.

Funkcja `scanf` (tak jak `printf` zresztą) zastawia na nieświadomych i nieostrożnych użytkowników kilka wnyków. Programista stosujący funkcję `scanf` musi starannie sprawdzać, czy liczba specyfikatorów konwersji odpowiada liczbie zmiennych przekazanych w wywołaniu i czy poszczególne konwersje odpowiadają typom przekazanych zmiennych — podobnie jak w przypadku `printf`, kompilator nie ma obowiązku przeprowadzania takiej kontroli i wykrywania ewentualnych niezgodności. Kolejna pułapka czai się w znaku `&`, który zazwyczaj poprzedza każdą zmienną przekazywaną do `scanf`. Znak `&` jest zazwyczaj (choć nie zawsze) konieczny i to programista ma obowiązek pamiętać o jego stosowaniu.

Pominięcie symbolu `&` przy zmiennej przekazywanej do funkcji `scanf` prowadzi do nieprzewidywalnych wyników działania programu. Potencjalnie są to efekty katastrofalne. Najczęściej taki błąd prowadzi do wyłożenia się programu. W najlepszym przypadku podstawienie wartości pod zmienną będzie nieskuteczne — zmienna zachowa swoją poprzednią wartość (co nie oznacza, że będzie miała jakąkolwiek określoną wartość, jeśli np. nie została zainicjalizowana!). Pominięcie znaku `&` jest niestety częstym błędem. Niektóre kompilatory wykrywają taki błąd i generują komunikat z ostrzeżeniem w rodzaju „argument nie jest wskaźnikiem" (o wskaźnikach powiemy sobie w rozdziale 11.; to właśnie znak `&` tworzy wskaźnik do zmiennej). Ostrzeżenia związane z wywołaniami funkcji `scanf` trzeba traktować bardzo poważnie.

Wywołania funkcji `scanf` są efektywnym, ale potencjalnie niebezpiecznym sposobem wczytywania danych do programów. Wielu zawodowych programistów C unika funkcji `scanf`. Wolą oni wczytać do programu całość danych wejściowych w postaci znakowej i dopiero później zamienić ją na oczekiwane wartości liczbowe. My natomiast będziemy korzystać ze `scanf` całkiem sporo, zwłaszcza w początkowych rozdziałach książki — jest to zwyczajnie najprostszy sposób wczytywania wartości liczbowych do programu. Trzeba tylko mieć świadomość, że wiele programów zachowa się niepoprawnie, kiedy użytkownik wprowadzi do nich nieodpowiednie dane wejściowe. Wkrótce się przekonamy, że można zresztą skutecznie sprawdzać, czy funkcji `scanf` udało się wczytać z wejścia oczekiwane dane (a jeśli nie, odpowiednio zareagować, zamiast brnąć dalej w program z niepoprawnymi danymi). Takie sprawdziany są jednak mało zasadne w zakresie początkowych przykładów prezentowanych w książce — nadmiernie rozbudowałyby program, który ma przecież przede wszystkim ilustrować bieżące zagadnienie.

wykrywanie błędów w scanf ➤ 22.3

Działanie funkcji scanf

Funkcja `scanf` robi w istocie znacznie więcej, niż dotychczas powiedziano. Jest to w zasadzie mechanizm dopasowywania wzorców w ciągach znaków, który próbuje pogrupować znaki wejściowe i dopasować je do specyfikatorów konwersji.

Działaniem funkcji `scanf` sterujemy za pomocą ciągu formatującego. Funkcja `scanf` analizuje zawartość tego ciągu, od lewej do prawej strony. Dla każdego napotkanego w ciągu specyfikatora konwersji próbuje w ciągu wejściowym programu zlokalizować wartość odpowiedniego typu. W czasie tego wyszukiwania automatycznie pomija znaki odstępów. Znaleziony podciąg pasujący do specyfikatora konwersji jest wczytywany aż do miejsca, w którym wystąpi znak niepasujący do wymagań konwersji. Jeśli udało się wczytać taki podciąg, funkcja `scanf` dokonuje konwersji i przechodzi do analizy reszty ciągu formatującego. Pierwsze nieudane dopasowanie specyfikatora konwersji do danych wczytywanych z wejścia kończy działanie całej funkcji, bez rozpatrywania reszty ciągu formatującego (i bez uwzględniania reszty danych wejściowych).

W toku poszukiwania pierwszego znaku liczby funkcja `scanf` ignoruje wszystkie **znaki odstępów** (spacje, znaki tabulacji poziomej i pionowej, znaki wysuwu formularza i znaki nowego wiersza). Dzięki temu na wejściu dane można podawać w jednym wierszu albo rozproszyć je pomiędzy różnymi wierszami. Przy danym wywołaniu funkcji `scanf`:

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

użytkownik może wprowadzić na wejście np. trzy wiersze danych:

```
1
-20   .3
   -4.0e3
```

dla funkcji `scanf` wejście jest widoczne jako ciągły strumień znaków:

```
··1¤-20···.3¤···-4.0e3¤
```

(w celu uwidocznienia znaków odstępów zastosowaliśmy znak · dla spacji i znak ¤ dla nowego wiersza). Ponieważ funkcja `scanf` pomija znaki odstępów i szuka przede wszystkim początku liczby i podciągu znaków nadających się do konwersji, takie dane wejściowe mogą być za jednym zamachem wczytane przez funkcję `scanf`. Poniższy schemat ilustruje działanie funkcji `scanf`. Znak `p` oznacza tu pomijanie bieżącego znaku wejścia, znak `w` oznacza wczytywanie znaków do bieżącej konwersji:

```
··1¤-20···.3¤···-4.0e3¤
ppwpwwwpppwwpppppwwwwww
```

Ostatni znak strumienia wejściowego, czyli pierwszy znak za ostatnią skonwertowaną grupą znaków, jest przez funkcję `scanf` „podglądany", ale nie jest wczytywany. Zostanie on wczytany i pominięty bądź skonwertowany przy następnym wywołaniu `scanf`.

Według jakich reguł `scanf` rozpoznaje liczbę całkowitą albo zmiennoprzecinkową w ciągu danych wejściowych? Otóż kiedy `scanf` ma wczytać liczbę całkowitą, szuka w ciągu wejściowym znaku cyfry, ewentualnie znaku + albo –.

Po znalezieniu takiego znaku wczytuje wszystkie kolejne znaki aż do napotkania znaku niebędącego cyfrą. Natomiast w przypadku liczby zmiennoprzecinkowej `scanf` szuka:

znaku + albo – (opcjonalnie), a za nim

ciągu cyfr (zawierającego ewentualnie znak przecinka dziesiętnego), a za nim

ciągu wykładnika (opcjonalnie); ciąg wykładnika składa się z litery `e` (albo `E`), opcjonalnego znaku (+ albo –) i co najmniej jednej cyfry.

W przypadku funkcji `scanf` konwersje `%e`, `%f` i `%g` można stosować zamiennie — wszystkie trzy reprezentują te same reguły dopasowania wartości zmiennoprzecinkowej.

PiO

Kiedy `scanf` napotyka znak niezgodny z regułą dopasowania dla bieżącej konwersji, znak ten jest „odkładany" z powrotem do strumienia wejściowego, aby był dostępny przy obsłudze następnych specyfikatorów konwersji albo dla kolejnych wywołań `scanf` wczytujących kolejne wartości. Weźmy na przykład następujące (niewątpliwie patologiczne) rozmieszczenie naszych czterech liczb wejściowych:

```
1-20.3-4.0e3¤
```

Zastosujemy takie same wywołanie `scanf` jak poprzednio:

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

Funkcja `scanf` będzie przetwarzać taki ciąg wejściowy następująco:

- Bieżąca konwersja: `%d`. Pierwszy niepusty znak wejścia to `1`. Ponieważ liczba całkowita może zaczynać się od cyfry, znak jest akceptowany jako pierwszy znak podciągu do dopasowania. Funkcja wczytuje następny znak: `-`. Taki znak nie może się pojawić wewnątrz zapisu wartości całkowitej, więc `scanf` odkłada znak z powrotem do strumienia i podstawia pod pierwszą zmienną (`i`) wartość 1.
- Bieżąca konwersja: `%d`. Pierwszy niepusty znak wejścia to `-` (dozwolony); kolejne znaki to `2`, `0` i `.` (kropka). Liczba całkowita nie może zawierać kropki, więc znak jest odkładany z powrotem do strumienia wejściowego, a funkcja podstawia pod kolejną zmienną (`j`) wartość –20.
- Bieżąca konwersja: `%f`. Pierwszy niepusty znak wejścia to `.` (dozwolony); kolejne znaki to `3` i `-` (minus). Liczba zmiennoprzecinkowa nie zawiera znaku `-` po znaku cyfry, więc znak `-` jest odkładany z powrotem do strumienia wejściowego, a funkcja podstawia pod kolejną zmienną (`x`) wartość 0.3.
- Bieżąca konwersja: `%f`. Pierwszy niepusty znak wejścia to `-` (dozwolony); kolejne znaki to `4`, `.`, `0`, `e`, `3` i ¤ (nowy wiersz). Liczba zmiennoprzecinkowa nie może zawierać znaku nowego wiersza, więc znak jest odkładany z powrotem do strumienia wejściowego, a funkcja podstawia pod kolejną zmienną (`y`) wartość -4.0×10^3.

W tym przykładzie funkcja `scanf` mogła skutecznie dopasować wszystkie kolejne specyfikatory konwersji do ciągu wejściowego. Ponieważ znak nowego wiersza nie został „zużyty", będzie pierwszym znakiem wczytywanym przy następnym wywołaniu `scanf`.

Zwykłe znaki w ciągu formatującym funkcji scanf

Zasadę dopasowywania wzorców do podciągów strumienia wejściowego można rozszerzyć, zapisując ciąg formatujący zawierający poza specyfikatorami konwersji również zwykłe napisy. W takim przypadku pomiędzy specyfikatorami konwersji funkcja `scanf` będzie porównywała kolejne znaki wejścia ze znakami ciągu formatującego. Porównanie takie odbywa się różnie, zależnie od tego, czy ciąg formatujący zawiera znaki odstępów:

- **Znaki odstępów w ciągu formatującym.** Kiedy w ciągu formatującym znajdują się znaki odstępu, funkcja `scanf` będzie w strumieniu wejściowym wczytywała kolejne znaki tak długo, aż napotka pierwszy znak niebędący znakiem odstępu (ten znak zostanie „odłożony" z powrotem do strumienia wejściowego). Liczba znaków odstępu w ciągu formatującym nie musi dokładnie odpowiadać liczbie takich znaków w ciągu wejściowym. Pojedynczy znak odstępu w ciągu formatującym zostanie dopasowany do dowolnie długiego podciągu takich znaków w ciągu wejściowym (co więcej, obecność znaku odstępu w ciągu formatującym nie wymusza obecności takiego znaku w ciągu wejściowym — znak odstępu w ciągu formatującym odpowiada *dowolnie* długiemu podciągowi takich znaków w ciągu wejściowym, a więc również podciągowi zerowemu).
- **Pozostałe znaki.** Kiedy w ciągu formatującym znajduje się znak nienależący do specyfikatorów konwersji i niebędący znakiem odstępu, funkcja `scanf` porównuje go wprost z następnym znakiem wejścia. Jeśli znaki są zgodne, funkcja przechodzi do przetwarzania następnego znaku ciągu wejściowego. Przy braku zgodności funkcja odkłada niepasujący znak z powrotem do ciągu wejściowego i przerywa dalsze przetwarzanie ciągu formatującego.

Dla przykładu niech funkcja `scanf` otrzyma ciąg formatujący w postaci `"%d/%d"`. Jeśli na wejściu programu pojawi się ciąg:

```
·5/·96
```

funkcja `scanf` pominie pierwszy znak odstępu, szukając liczby całkowitej, następnie dopasuje do `%d` podciąg `5`, dopasuje bezpośrednio znak `/`, pominie spację w poszukiwaniu kolejnej liczby całkowitej, a później dopasuje do `%d` podciąg `96`. Ale jeśli na wejściu pojawi się:

```
·5·/·96
```

to funkcja `scanf` pominie pierwszy znak odstępu, szukając liczby całkowitej, następnie dopasuje do `%d` podciąg `5`, a później spróbuje dopasować do wejścia znak `/`. Ponieważ w bieżącym miejscu wejścia zamiast tego znaku znajduje się inny (tu: znak odstępu), funkcja przerwie przetwarzanie wejścia, odkładając spację z powrotem do ciągu wejściowego. Na wejściu pozostanie ciąg `·/·96` czekający na ewentualne kolejne wywołanie `scanf`. Aby umożliwić dopasowanie wejścia ze spacją (spacjami) po pierwszej liczbie całkowitej, ciąg formatujący powinien mieć postać `"%d /%d"`.

Skutki mylenia printf ze scanf

Wywołania funkcji `printf` i `scanf` bywają bardzo podobne, ale w ich działaniu zachodzą daleko idące różnice. Zignorowanie tych różnic może być bardzo niebezpieczne dla działania programu.

Jedną z częstych pomyłek jest umieszczenie `&` przed zmienną w wywołaniu funkcji `printf`:

```
printf("%d %d\n", &i, &j);      /*** ŹLE ***/
```

Na szczęście taka omyłka jest stosunkowo prosta do wytropienia — w toku działania programu funkcja `printf` wyświetli „śmieci" zamiast oczekiwanych wartości `i` i `j`.

Ponieważ funkcja `scanf` normalnie pomija znaki odstępów przy poszukiwaniu podciągu do dopasowania do wartości liczbowej, rzadko kiedy pojawia się potrzeba umieszczania w ciągu formatującym `scanf` czegokolwiek poza specyfikatorami konwersji. Nieuprawnione założenie, że ciąg formatujący `scanf` powinien ściśle odpowiadać analogicznemu ciągowi formatującemu `printf` — to kolejny częsty błąd — może doprowadzić do niepoprawnego działania funkcji `scanf`. Zobaczmy, co się stanie, jeśli w programie znajdzie się następująca instrukcja:

```
scanf("%d, %d", &i, &j);
```

Funkcja `scanf` będzie najpierw szukać podciągu wartości typu `int` i wpisze tę wartość pod zmienną `i`. Potem będzie próbowała dopasować w ciągu wejściowym znak przecinka. Jeśli w ciągu wejściowym zamiast przecinka pojawi się spacja, działanie funkcji zostanie przerwane i zmienna `j` nie otrzyma oczekiwanej wartości.

Ciągi formatujące dla funkcji `printf` często kończą się znakiem sterującym `\n`, wymuszającym wstawienie do wyjścia nowego wiersza. Taki sam znak na końcu ciągu formatującego `scanf` to zazwyczaj zły pomysł. Dla `scanf` znak nowego wiersza w ciągu formatującym jest równoważny ze znakiem spacji; pomija go, szukając następnego znaku niebędącego znakiem odstępu. Jeśli na przykład ciąg formatujący ma postać `"%d\n"`, `scanf` pominie ewentualne początkowe znaki odstępu, wczyta wartość całkowitą, a następnie będzie w ciągu wejściowym szukać następnego znaku innego niż odstęp. W przypadku programu interaktywnego może to doprowadzić do „zawieszenia" programu do czasu, kiedy użytkownik wprowadzi na wejście znak inny niż odstęp.

PROGRAM

Dodawanie ułamków

Aby zilustrować zdolność funkcji `scanf` do dopasowywania wzorców w ciągach znaków, weźmiemy na warsztat problem wczytania ułamka wprowadzanego przez użytkownika. Ułamki są zwyczajowo reprezentowane przez zapis *licznik/mianownik*. Zamiast zmuszać użytkownika do nieintuicyjnego wprowadzania ułamka jako dwóch osobnych liczb całkowitych, dzięki funkcji `scanf` pozwolimy mu wprowadzić ułamek w klasycznej postaci. Oto program ilustrujący tę technikę przy dodawaniu dwóch ułamków:

addfrac.c

```
/* Dodawanie dwóch ułamków zwykłych */

#include <stdio.h>

int main(void)
{
  int num1, denom1, num2, denom2, result_num, result_denom;

  printf("Podaj pierwszy ułamek: ");
  scanf("%d/%d", &num1, &denom1);

  printf("Podaj drugi ułamek: ");
  scanf("%d/%d", &num2, &denom2);

  result_num = num1 * denom2 + num2 * denom1;
  result_denom = denom1 * denom2;
  printf("Suma ułamków wynosi %d/%d\n",
         result_num, result_denom);

  return 0;
}
```

Przykładowa sesja z takim programem wyglądała tak:

```
Podaj pierwszy ułamek: 5/6
Podaj drugi ułamek: 3/4
Suma ułamków wynosi 38/24
```

Zauważmy, że program nie przewiduje znormalizowania wynikowego ułamka.

Pytania i odpowiedzi

***P: Widywałem specyfikatory w postaci `%i`, przeznaczone do wczytywania i wypisywania liczb całkowitych. Czym różnią się od `%d` (s. 71)?**

O: W ciągu formatującym funkcji `printf` nie ma pomiędzy nimi żadnej różnicy. Za to w funkcji `scanf` `%d` dopasuje wyłącznie liczbę całkowitą zapisaną w postaci dziesiętnej, natomiast `%i` może dopasować także liczby całkowite zapisane w innych systemach liczbowych (przy innych podstawach), na przykład liczby ósemkowe czy szesnastkowe. Otóż jeśli ciąg wejściowy zawiera przedrostek `0` (jak w `056`), przy konwersji `%i` taki ciąg zostanie potraktowany jako ósemkowy. Przedrostek `0x` albo `0X` (jak w `0x56`) przy konwersji `%i` spowoduje konwersję liczby jako wartości szesnastkowej. Zastosowanie `%i` zamiast `%d` do wczytywania liczb może dać zaskakujące wyniki, jeśli użytkownik omyłkowo wprowadzi `0` przed właściwą liczbą; z tego powodu zaleca się trzymanie się konwersji `%d`.

liczby ósemkowe ➤ 7.1
liczby szesnastkowe ➤ 7.1

P: Skoro `printf` traktuje znak `%` jako początek specyfikatora konwersji, jak można wypisać na wyjściu programu znak procenta?

O: Funkcja `printf` wypisze na wyjściu znak `%` tam, gdzie w ciągu formatującym pojawi się para przylegających znaków procenta (`%%`). Na przykład instrukcja:

```
printf("Zysk netto: %d%%\n", profit);
```

może spowodować wypisanie:

```
Zysk netto: 10%
```

P: Znak sterujący `\t` ma wymusić na `printf` przesunięcie kursora do następnej pozycji tabulacji. Skąd wiadomo, jaka jest szerokość tabulatora (s. 74)?

O: Nie wiadomo. Efekt wypisania na wyjściu znaku `\t` nie jest ściśle zdefiniowany w standardzie języka C. Jest on zależny od reakcji systemu operacyjnego na żądanie wypisania znaku tabulacji. Zazwyczaj szerokość tabulatora to osiem znaków, ale C sam w sobie nie daje takiej gwarancji.

P: Co zrobi funkcja `scanf`, jeśli będzie miała wczytać liczbę, a użytkownik wprowadzi ciąg nieliczbowy?

O: Wyjaśnimy to na przykładzie:

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

Załóżmy, że użytkownik wprowadzi poprawną wartość liczbową, za którą umieści znaki inne niż cyfry:

```
Podaj liczbę: 23bla
```

W takim przypadku funkcja `scanf` wczyta znaki `2` oraz `3` i w zmiennej `i` umieści wartość `23`. Reszta znaków (`bla`) będzie w strumieniu wejściowym czekała na kolejne wywołanie `scanf` (albo innej funkcji obsługującej wejście). Jeśli natomiast użytkownik wprowadzi ciąg niepoprawny od pierwszego znaku:

```
Podaj liczbę: bla
```

zmienna `i` nie otrzyma wartości wczytywanej z wejścia programu, a w strumieniu wejściowym ciąg `bla` poczeka na kolejne wywołania funkcji wejścia.

wykrywanie błędów w `scanf` ➤ *22.3*

Co można zrobić w takich smutnych przypadkach? Później dowiesz się, jak sprawdzić skuteczność wywołania funkcji `scanf`. Jeśli wywołanie będzie nieskuteczne, możemy zakończyć program albo spróbować naprawić sytuację, na przykład odrzucając niepoprawne wejście i ponownie prosząc użytkownika o wprowadzenie danych (sposoby odrzucania danych wejściowych będą omawiane w sekcji pytań i odpowiedzi rozdziału 22.).

P: Nie rozumiem, w jaki sposób funkcja `scanf` „odkłada” znak z powrotem do ciągu wejściowego i potem ponownie wczytuje go z wejścia (s. 77)?

O: Okazuje się, że program nie wczytuje danych wejściowych bezpośrednio przy ich wprowadzaniu. Ciąg wejściowy programu jest przechowywany w ukrytym buforze, do którego odwołuje się funkcja `scanf`. W takim układzie funkcja `scanf` może „odłożyć” znak z powrotem do bufora. Buforowanie wejścia będzie omawiane w rozdziale 22.

P: Jak zadziała funkcja `scanf`, kiedy użytkownik umieści pomiędzy liczbami znaki przestankowe (np. przecinki)?

O: Zobaczmy to na prostym przykładzie. Załóżmy, że zamierzamy wczytać funkcją `scanf` parę liczb:

```
printf("Podaj dwie liczby: ");
scanf("%d%d", &i, &j);
```

Jeśli użytkownik wprowadzi:

```
4,28
```

funkcja `scanf` wczyta znak `4` i podstawi wartość 4 do zmiennej `i`. W poszukiwaniu następnego podciągu reprezentującego liczbę całkowitą znajdzie przecinek. Ponieważ przecinek nie może rozpoczynać zapisu liczby, funkcja przerwie działanie. Sam przecinek i druga wprowadzona liczba zostaną do dyspozycji następnych wywołań `scanf`.

Oczywiście można temu łatwo zaradzić, instruując użytkownika co do oczekiwanego formatu danych wejściowych, np. nakazując mu zawsze oddzielanie liczb przecinkiem:

```
printf("Podaj dwie liczby, oddzielone przecinkiem: ");
scanf("%d,%d", &i, &j);
```

Ćwiczenia

Podrozdział 3.1

1. Co pojawi się na wyjściu po wykonaniu poniższych wywołań `printf`?
   (a) `printf("%6d,%d4", 86, 1040);`
   (b) `printf("%12.5e", 86, 30.253);`
   (c) `printf("%.4f", 83.162);`
   (d) `printf("%-6.2g", .0000009979);`

W 2. Napisz wywołanie funkcji `printf` wyświetlającej wartość zmiennej typu `float` w następujących formatach:
   (a) w zapisie wykładnikowym, wyrównaną do lewej w polu o szerokości 8, z jedną cyfrą po przecinku;
   (b) w zapisie wykładnikowym, wyrównaną do prawej w polu o szerokości 10, z sześcioma cyframi po przecinku;
   (c) w zapisie dziesiętnym z przecinkiem, wyrównaną do lewej w polu o szerokości 8, z trzema cyframi po przecinku;
   (d) w zapisie dziesiętnym z przecinkiem, wyrównaną do prawej w polu o szerokości 6, bez cyfr po przecinku.

Podrozdział 3.2

3. Dla każdej z poniższych par ciągów formatujących `scanf` wskaż, czy oba ciągi pary są sobie równoważne, czy nie. Jeśli nie są, wskaż różnicę w ich działaniu.
   (a) `"%d"` `" %d"`
   (b) `"%d-%d-%d"` `" %d -%d -%d"`
   (c) `"%f"` `"%f "`
   (d) `"%f,%f"` `"%f, %f"`

4. Weźmy następujące wywołanie funkcji `scanf`:

---

* Ćwiczenia z gwiazdką są podchwytliwe — poprawna odpowiedź zazwyczaj jest różna od odpowiedzi narzucającej się na pierwszy rzut oka. Przeczytaj pytanie uważnie, dokładnie prześledź kod, w razie potrzeby powtórz lekturę odpowiedniego podrozdziału. Powodzenia! — *przyp. autora.*

```
scanf("%d%f%d", &i, &x, &j);
```

Jakie będą wartości zmiennych `i`, `x` i `j` po wykonaniu wywołania (zakładamy, że zmienne `i` i `j` są zmiennymi typu `int`, a `x` jest zmienną typu `float`), jeśli użytkownik wprowadzi:

```
10.3 5 6
```

W*5. Weźmy następujące wywołanie funkcji `scanf`:

```
scanf("%f%d%f", &x, &i, &y);
```

Jakie będą wartości zmiennych `x`, `i` i `y` po wykonaniu wywołania (zakładamy, że zmienne `x` i `y` są zmiennymi typu `float`, a `i` jest zmienną typu `int`), jeśli użytkownik wprowadzi:

```
12.3 45.6 789
```

6. Pokaż, jak można przerobić program *addfrac.c* z podrozdziału 3.2, aby użytkownik mógł wprowadzać ułamki ze spacjami wokół znaku dzielenia `/`.

Zadania programistyczne

W 1. Napisz program, który przyjmuje na wejście datę w postaci *dd/mm/rrrr*, a następnie wypisuje ją w formacie *rrrrmmdd*:

```
Podaj datę (dd/mm/rrrr): 17/2/2011
Podałeś datę 20110217
```

2. Napisz program formatujący informacje o produkcie wprowadzone przez użytkownika. Sesja z programem powinna wyglądać tak:

```
Podaj numer towaru: 583
Podaj cenę jednostkową: 13.5
Podaj datę zakupu (dd/mm/rrrr): 24/10/2010
Towar           Cena            Data
                jed.            zakupu
583             $  13.50        24/10/2010
```

Numer towaru i data powinny być wyrównane do lewej. Cena jednostkowa powinna być wyrównana do prawej. Program powinien dopuszczać kwoty do 9999,99. *Podpowiedź:* Do wyrównania kolumn zastosuj tabulatory.

W 3. Książki są oznaczane międzynarodowym numerem ISBN (International Standard Book Number). Numery ISBN nadawane po 1 stycznia 2007 roku składają się z 13 cyfr, podzielonych na pięć grup, np. 978-0-393-97950-3 (starsze numery ISBN miały 10 cyfr). Pierwsza grupa (przedrostek *GSI*) to obecnie albo 978 albo 979. *Identyfikator grupy* określa język kraju wydania (np. w krajach anglojęzycznych stosuje się kody 0 i 1). *Kod wydawcy* identyfikuje wydawcę (393 to kod wydawnictwa W.W. Norton). *Numer publikacji* to numer nadawany przez wydawcę konkretnej książce (tutaj 97950). Numer ISBN kończy się sumą kontrolną umożliwiającą weryfikację poprawności numeru ISBN. Napisz program, który podzieli na grupy numer ISBN wprowadzony przez użytkownika:

```
Podaj numer ISBN: 978-0-393-97950-3
Przedrostek GSI: 978
Identyfikator grupy: 0
```

```
Kod wydawcy: 393
Numer publikacji: 97950
Suma kontrolna: 3
```

Uwaga: Liczba cyfr w każdej z grup może być różna. Nie można założyć, że grupy mają akurat takie rozmiary jak w tym przykładzie. Przetestuj program na innych, prawdziwych numerach ISBN (znajdziesz je na okładkach książek i na stronach z informacjami o wydaniu).

4. Napisz program, który zapyta użytkownika o numer telefoniczny w formacie `(xx) xxx-xxxx`, a potem wyświetli ten numer w zapisie `0-xx xxx-xx-xx`:

```
Podaj numer telefonu [(xx) xxx-xxxx]: (61) 817-6900
Podany numer: 0-61 817-69-00
```

5. Napisz program, który będzie monitował, aby użytkownik wprowadził liczby od 1 do 16 (w dowolnej kolejności), potem wyświetli te liczby czwórkami (w postaci macierzy 4×4), a następnie wypisze sumy w wierszach, kolumnach i po przekątnych:

```
Podaj liczby od 1 do 16 (w dowolnej kolejności):
16 3 2 13 5 10 11 8 9 6 7 12 4 15 14 1
16  3  2 13
 5 10 11  8
 9  6  7 12
 4 15 14  1

Sumy w wierszach: 34 34 34 34
Sumy w kolumnach: 34 34 34 34
Sumy po przekątnych: 34 34
```

Jeśli sumy w wierszach, kolumnach i po przekątnych są identyczne (jak w przykładzie), to takie liczby stanowią tak zwany **magiczny kwadrat**. Ten magiczny kwadrat pojawił się w 1514 roku na rycinie artysty i matematyka Albrechta Dürera (zauważ, że środkowe liczby w ostatnim wierszu składają się na datę ryciny).

6. Przerób program *addfrac.c* z podrozdziału 3.2 tak, aby użytkownik za jednym zamachem wprowadzał oba ułamki oddzielone znakiem +:

```
Podaj dwa ułamki oddzielone znakiem plusa: 5/6+3/4
Suma ułamków wynosi 38/24
```

4 Wyrażenia

Używanie kalkulatorka to jeszcze nie programowanie,
a już nie matematyka.

Jedną z ważniejszych cech języka C jest nacisk, jaki jest w nim kładziony na wyrażenia — inaczej „wzory", które mówią o sposobie obliczania wartości. Wyrażenia są tu ważniejsze niż instrukcje. Najprostsze wyrażenia to zmienne i stałe programu. Zmienna reprezentuje wartość, która jest obliczana w czasie działania programu poprzez pobranie wartości z pamięci skojarzonej ze zmienną. Stała reprezentuje wartość znaną już w czasie kompilacji. Bardziej rozbudowane wyrażenia obejmują operandy i operatory (przy czym operatory same w sobie są również wyrażeniami). W wyrażeniu `a + (b * c)` widzimy zastosowanie operatora + do operandów `a` oraz `(b + c)`, natomiast obie strony operatora + (oba operandy) są pełnoprawnymi wyrażeniami.

Operatory są podstawowymi narzędziami budowania wyrażeń, a język C posiada bardzo bogaty zbiór operatorów. Przede wszystkim C obsługuje podstawowe operatory obecne w większości innych języków programowania:

- Operatory arytmetyczne, w tym dodawanie, odejmowanie, mnożenie i dzielenie.
- Operatory relacji do obliczania wartości wyrażeń logicznych, takich jak „i jest większe od 0".
- Operatory logiczne do budowania warunków logicznych, jak „i jest większe od 0 i jest mniejsze niż 10".

Na tym jednak nie koniec. W języku C mamy do dyspozycji dziesiątki innych operatorów. Jest ich tak wiele, że będziemy zmuszeni do ich stopniowego wprowadzania na przestrzeni aż dwudziestu rozdziałów tej książki. Opanowanie takiej liczby operatorów wydaje się zadaniem niepomiernym, ale w istocie większość z nich ma zasadnicze znaczenie dla efektywności programowania w języku C.

W tym rozdziale zajmiemy się jednymi z najważniejszych operatorów języka C, mianowicie weźmiemy na warsztat operatory arytmetyczne (podrozdział 4.1), operatory przypisania (podrozdział 4.2) oraz operatory inkrementacji i dekrementacji

(podrozdział 4.3). W podrozdziale 4.1 zajmiemy się też pierwszeństwem i łącznością operatorów — te cechy operatorów są niezwykle istotne dla poprawności wyrażeń obejmujących więcej niż jeden operator. W podrozdziale 4.4 opisany zostanie sposób, w jaki następuje obliczanie wyrażeń w języku C. Wreszcie w podrozdziale 4.5 zajmiemy się tak zwaną instrukcją wyrażeniową — konstrukcją sprawiającą, że dowolne wyrażenie może odgrywać rolę instrukcji w programie.

4.1. Operatory arytmetyczne

Operatory arytmetyczne — czyli operatory realizujące operacje dodawania, odejmowania, mnożenia i dzielenia — to istne woły robocze w większości języków programowania. C nie jest tu wyjątkiem. Listę operatorów arytmetycznych języka C wymienia tabela 4.1.

Tabela 4.1. Operatory arytmetyczne

Z jednym operandem	*Z dwoma operandami*	
	Addytywne	*Multiplikatywne*
+ zachowanie znaku liczby	+ dodawanie	* mnożenie
– zmiana znaku liczby	– odejmowanie	/ dzielenie
		% reszta z dzielenia

Operatory addytywne i multiplikatywne to tak zwane operatory **dwuargumentowe** (ang. *binary*), ponieważ operują na *dwóch* operandach. Operatory **jednoargumentowe** (ang. *unary*) wymagają tylko jednego operandu:

```
i = +1;      /* + jako operator jednoargumentowy */
j = -i;      /* + jako operator jednoargumentowy (negacja) */
```

Jednoargumentowy operator + nie ma żadnego działania. W rzeczy samej w specyfikacji K&R w ogóle go nie uwzględniono. Służy głównie do zaznaczenia, że stała liczbowa jest dodatnia.

Operatory dwuargumentowe nie powinny budzić wątpliwości. Może poza operatorem `%`, który oblicza resztę z dzielenia całkowitego. Wartość wyrażenia `i % j` jest to reszta z całkowitego dzielenia `i` przez `j`. Na przykład wartością wyrażenia `10 % 3` jest 1, a wyrażenie `12 % 4` ma wartość 0.

PiO

Operatory dwuargumentowe z tabeli 4.1 — z wyjątkiem operatora `%` — mogą być stosowane zarówno do operandów będących liczbami całkowitymi, jak i do operandów zmiennoprzecinkowych; dopuszczalne jest też mieszanie typów operandów. Kiedy w zasięgu działania operatora występują operandy typu `int` i `float`, wynikiem działania operatora jest wartość typu `float`. Dlatego wyrażenie `9 + 2.5f` daje 11,5, a `6.7f / 2` to 3.35.

Przy stosowaniu operatorów `/` i `%` należy zachować pewną ostrożność:

- Operator dzielenia `/` może dawać nieoczekiwane wyniki. Otóż kiedy oba operandy są wartościami całkowitymi, operator `/` obcina iloraz do najbliższej mniejszej wartości całkowitej. Dlatego wyrażenie `1 / 2` da wartość 0.

- Operator `%` wymaga, aby operandy były wartościami całkowitymi. Jeśli którykolwiek z operandów `%` jest wartością inną niż całkowita, program nie da się skompilować.
- Jeśli prawym operandem operatora `/` albo `%` będzie 0, działanie programu będzie niezdefiniowane.

niezdefiniowane zachowanie programu ➤ 4.4

PiO

- Największe pułapki czają się przy stosowaniu operatorów `/` i `%` z operandami ujemnymi. Standard C89 stwierdza, że kiedy którykolwiek z operandów jest ujemny, wynik dzielenia może zostać zaokrąglony w górę albo w dół (np. wartością wyrażenia `-9 / 7` może być albo –1, albo –2). Również kiedy operand `i` albo `j` wyrażenia `i % j` jest ujemny, według standardu C89 wynik działania operatora jest zależny od implementacji (np. wyrażenie `-9 % 7` może dać wartość –2 albo 5). Z kolei w C99 wynik dzielenia całkowitego jest zawsze zaokrąglany w kierunku zera (`-9 / 7` da wynik –1), a wartość wyrażenia `i % j` będzie miała znak taki jak operand `i` (czyli `-9 % 7` to –2).

C99

Zachowanie zależne od implementacji

Pojęcie zachowania albo działania **zależnego od implementacji** będzie się pojawiać w naszym omówieniu na tyle często, że warto przedstawić je zawczasu. W standardzie języka C celowo pozostawiono specyfikację działania niektórych elementów języka jako nie do końca sprecyzowaną, z założeniem, że „implementacja" — czyli konkretne wcielenie kompilatora i programu konsolidującego dla konkretnej platformy maszynowej i systemowej — wypełni lukę po swojemu. W efekcie zachowanie programu w toku wykonania może być różne na różnych platformach. Przykładem zachowania zależnego od implementacji jest właśnie działanie operatorów `/` i `%` według standardu C89.

Niedoprecyzowanie niektórych elementów języka wydaje się dziwactwem, a nawet działaniem szkodliwym, ale w istocie dobrze odzwierciedla filozofię przyświecającą twórcom języka C. Jednym z założeń projektowych była przecież możliwie duża wydajność programów pisanych w tym języku, co często oznacza konieczność dostosowania implementacji poszczególnych konstrukcji do zachowania danej maszyny. Tak więc niektóre procesory, wykonując operację dzielenia całkowitego –9 przez 7, dają wynik –1, a inne –2. Standard C89 zwyczajnie odzwierciedlał różnorodność maszynowej implementacji podstawowych operacji arytmetycznych.

Przy pisaniu programów najlepiej unikać polegania na zachowaniach zależnych od implementacji. Jeśli nie da się takiej zależności uniknąć, warto przynajmniej „ręcznie" sprawdzić wyniki wątpliwych operacji — standard C wymaga na szczęście, aby sposób realizacji zachowań zależnych od implementacji był udokumentowany.

Pierwszeństwo i łączność operatorów

Kiedy wyrażenie zawiera więcej niż jeden operator, interpretacja wyrażenia może być wątpliwa. Czy na przykład `i + j * k` oznacza „dodanie `i` do `j` i pomnożenie sumy przez `k`", czy może „pomnożenie `j` przez `k` i dodanie do iloczynu `i`"? Wątpliwość można usunąć, stosując nawiasy, a więc zapisując jawnie albo `(i + j) * k`, albo `i + (j * k)`. Język C co do zasady pozwala na grupowanie podwyrażeń w wyrażeniach właśnie za pomocą nawiasów.

No dobrze, ale jak zinterpretować wyrażenie bez nawiasów? Czy dla kompilatora `i + j * k` to jest `(i + j) * k`, czy może jednak `i + (j *k)`? Tak jak w wielu innych językach, w C potencjalne wątpliwości tego rodzaju rozstrzyga się na bazie reguł **pierwszeństwa** (albo inaczej priorytetów) operatorów (ang. *precedence*). Dla operatorów arytmetycznych pierwszeństwo przedstawia się tak (od najwyższego priorytetu):

```
+  -   (jednoargumentowe)
*  /          %
+  -   (dwuargumentowe)
```

Operatory wymienione w tym samym wierszu (jak jednoargumentowe + i –) cechują się identycznym pierwszeństwem.

Kiedy w jednym wyrażeniu występuje wiele operatorów, możemy określić sposób ich interpretowania (kolejność obliczania podwyrażeń) przez kompilator, posiłkując się nawiasami grupującymi podwyrażenia, od operatorów o najwyższym priorytecie po operatory o priorytecie najniższym. Oto przykłady:

```
i + j * k     interpretuje się jako   i + (j * k)
-i * -j       interpretuje się jako   (-i) * (-j)
+i + j / k    interpretuje się jako   (+i) + (j / k)
```

Reguły pierwszeństwa operatorów nie są jednak wystarczające do rozstrzygnięcia wątpliwości co do kolejności obliczania podwyrażeń, kiedy w wyrażeniu występuje wiele operatorów o tym samym priorytecie. W takiej sytuacji zastosowanie ma reguła **łączności** operatorów (ang. *associativity*). O operatorze mówimy, że jest **lewostronnie łączny**, kiedy grupuje operandy od lewej do prawej. Dwuargumentowe operatory arytmetyczne (*, /, %, + i –) są łączne lewostronnie, więc:

```
i - j - k     interpretuje się jako   (i - j) - k
i * j / k     interpretuje się jako   (i * j) / k
```

Operator jest prawostronnie łączny, kiedy grupuje operandy od prawej strony do lewej. Prawostronnie łącznymi operatorami są jednoargumentowe operatory arytmetyczne (+ i –), więc:

```
- + i          interpretuje się jako   -(+i)
```

Reguły pierwszeństwa i łączności operatorów są bardzo ważne także w innych językach programowania, ale w C ich znaczenie jest szczególne. Z drugiej strony przy liczbie operatorów dostępnych w języku C (jest ich niemal pięćdziesiąt!) mało który programista jest w stanie zapamiętać łączność i pierwszeństwo wszystkich operatorów. Pisząc program w języku C, warto więc mieć pod ręką tabelkę operatorów i zaglądać do niej w razie wątpliwości. Można też je eliminować poprzez jawne grupowanie podwyrażeń w nawiasach.

tabela operatorów ➤ *Dodatek A*

PROGRAM

Obliczanie cyfry kontrolnej kodu kreskowego

W latach siedemdziesiątych producenci dóbr szybko zbywalnych w Stanach Zjednoczonych i Kanadzie zaczęli znakować swoje towary, umieszczając na nich kody kreskowe. Taki kod kreskowy albo inaczej kod UPC (*universal product code*)

identyfikuje zarówno producenta, jak i konkretny produkt. Każdy kod kreskowy reprezentuje dwunastocyfrową liczbę (wypisaną zresztą najczęściej pod kodem kreskowym). Oto przykładowy kod kreskowy pizzy Stouffer:

Cyfry:

```
0 13800 15173 5
```

są jawnie wypisane pod właściwym kodem kreskowym. Pierwsza z nich określa typ towaru (dla większości towarów jest to `0` albo `7`, dla towarów ważonych `2`, dla lekarstw i produktów farmaceutycznych `3`, a dla kuponów `5`). Pierwsza grupa pięciu cyfr identyfikuje producenta towaru (13800 to kod Nestle USA Frozen Food Division, czyli działu mrożonek amerykańskiego koncernu Nestle). Druga grupa cyfr (znów pięć) to identyfikator produktu (określający między innymi rozmiar opakowania). Ostatnia cyfra to cyfra kontrolna, której jedynym zadaniem jest pomoc w weryfikowaniu poprawności kodów kreskowych (poprawności poprzednich cyfr). Kiedy taki kod zostanie niepoprawnie zeskanowany, cyfra kontrolna nie będzie się zgadzać z jedenastoma pierwszymi cyframi, więc skaner kasowy odrzuci kod.

Oto jedna z metod obliczania cyfry kontrolnej kodu kreskowego UPC:

Dodaj cyfrę pierwszą, trzecią, piątą, siódmą, dziewiątą i jedenastą.
Dodaj cyfrę drugą, czwartą, szóstą, ósmą i dziesiątą.
Pomnóż pierwszą sumę przez 3 i dodaj ją do drugiej sumy.
Od wyniku odejmij 1.
Oblicz resztę z dzielenia pomniejszonego wyniku przez 10.
Odejmij resztę z dzielenia od 9.

Na przykładzie pizzy Stouffer otrzymamy pierwszą sumę o wartości (0+3+0+1+1+3) = 8; druga suma powinna wynosić (1+8+0+5+7) = 21. Suma iloczynu pierwszej sumy przez 3 i drugiej sumy daje 45. 45 odjąć 1 daje 44. Reszta z dzielenia 44 przez 10 to 4. Różnica 9 – 4 wynosi 5. Oto kilka innych przykładów liczbowych kodów kreskowych UPC, które możemy sprawdzić — lepiej liczyć, zamiast szukać tych produktów w lodówkach:

```
Jif Creamy Peanut Butter (18 uncji):          0 51500 24128    ?
Ocean Spray Jellied Cranberry Sauce (8 uncji): 0 31200 01005    ?
```

Odpowiedzi znajdują się u dołu strony[1].

[1] Brakujące cyfry kontrolne to 8 (Jif) i 6 (Ocean Spray) — *przyp. autora.*

Napiszmy program, który będzie obliczał cyfrę kontrolną dla dowolnego kodu UPC. Program będzie wymagał od użytkownika wprowadzenia pierwszych 11 cyfr kodu UPC, a następnie wypisze brakującą cyfrę kodu. Aby uniknąć omyłek, nakażemy użytkownikowi wprowadzanie cyfr kodu partiami: najpierw wprowadzi samotną cyfrę z lewej, potem grupę pięciu cyfr kodu producenta, a na koniec piątkę cyfr kodu produktu. Sesja z programem będzie wyglądała mniej więcej tak:

```
Podaj pierwszą cyfrę kodu: 0
Podaj pierwszą grupę pięciu cyfr kodu: 13800
Podaj następną grupę pięciu cyfr kodu: 15173
Cyfra kontrolna: 5
```

Zamiast wczytywać poszczególne grupy cyfr jako liczby pięciocyfrowe, będziemy wczytywać je jako piątkę liczb *jedno*cyfrowych. Wczytanie cyfr jako osobnych liczb będzie dla nas potem znacznie wygodniejsze. Nie trzeba będzie się też martwić na przykład o to, że jedna z dwóch pięciocyfrowych liczb nie zmieści się w zmiennej `int` (w niektórych starszych kompilatorach graniczna wartość typu `int` to 32 767). Aby wczytać pojedynczą cyfrę, wykorzystamy funkcję `scanf` ze specyfikatorami konwersji `%1d`, odpowiadającymi liczbom jednocyfrowym (w zapisie dziesiętnym).

upc.c

```
/* Obliczanie cyfry kontrolnej kodu kreskowego UPC */

#include <stdio.h>

int main(void)
{
  int d, i1, i2, i3, i4, i5, j1, j2, j3, j4, j5,
  first_sum, second_sum, total;

  printf("Podaj pierwszą cyfrę kodu: ");
  scanf("%1d", &d);
  printf("Podaj pierwszą grupę pięciu cyfr kodu: ");
  scanf("%1d%1d%1d%1d%1d", &i1, &i2, &i3, &i4, &i5);
  printf("Podaj drugą grupę pięciu cyfr kodu: ");
  scanf("%1d%1d%1d%1d%1d", &j1, &j2, &j3, &j4, &j5);

  first_sum = d + i2 + i4 + j1 + j3 + j5;
  second_sum = i1 + i3 + i5 + j2 + j4;
  total = 3 * first_sum + second_sum;

  printf("Cyfra kontrolna: %d\n", 9 - ((total - 1) % 10));

  return 0;
}
```

Zauważmy, że wyrażenie `9 - ((total - 1) % 10)` można by zapisać jako `9 - (total - 1) % 10`, ale dodatkowa para nawiasów nie zaciemnia wyrażenia — wręcz odwrotnie, czyni je czytelniejszym.

4.2. Operatory przypisania

Po obliczeniu wartości wyrażenia często chcemy zachować tę wartość w zmiennej do późniejszego użycia. W języku C służy do tego celu **operator przypisania** (ang. *assignment*) w postaci symbolu =. Aby dało się wygodnie aktualizować wartość zmiennej wartością wyrażenia, C oferuje również zestaw tak zwanych złożonych operatorów przypisania (ang. *compound assignment operators*).

Przypisania proste

Efektem przypisania *v* = *e* jest obliczenie wyrażenia *e* i skopiowanie wartości wyrażenia do *v*. Jak widać na przykładach poniżej, *e* może być zmienną, stałą albo dowolnym wyrażeniem:

```
i = 5;              /* i ma teraz wartość 5 */
j = i;              /* j ma teraz wartość 5 */
k = 10 * i + j;     /* k ma teraz wartość 55 */
```

Jeśli *e* i *v* nie są wartościami tego samego typu, w toku realizacji przypisania wartość *e* zostanie skonwertowana na typ *v*:

```
int i;
float f;

i = 72.99f;         /* i ma teraz wartość 72 */
f = 136;            /* f ma teraz wartość 136.0 */
```

przypisania z konwersją ➤ 7.4

Do tematu konwersji wartości przypisywanej wrócimy nieco później.

W wielu innych językach programowania przypisanie jest *instrukcją*. W języku C przypisanie jest jednak *operatorem*, tak jak +. Innymi słowy, akt przypisania jest wyrażeniem posiadającym wartość, tak samo jak akt dodawania jest wyrażeniem posiadającym wartość (tu równą sumie operandów). Wartością przypisania *v* = *e* jest wartość *v*, obliczana już po wykonaniu przypisania. Wartością przypisania `i = 72.99f` jest więc nie 72,99, ale 72.

Efekty uboczne

Zazwyczaj nie oczekuje się od operatorów, aby modyfikowały wartości operandów — w matematyce operatory nie mają przecież takich właściwości. Działanie `i + j` nie modyfikuje ani `i`, ani `j`, a jedynie oblicza sumę `i` oraz `j`.

Większość operatorów języka C również nie modyfikuje operandów, ale niektóre to robią. O takich operatorach mówimy, że mają **efekty uboczne**, ponieważ ich działanie nie ogranicza się tylko do jawnego wyliczenia wartości. Pierwszym operatorem, jaki poznajemy od strony efektów ubocznych, jest prosty operator przypisania — nie tylko oblicza wartość wyrażenia prawego operandu, ale także modyfikuje wartość lewego operandu. Obliczenie wartości wyrażenia `i = 0` daje wartość 0, a efektem ubocznym obliczenia wyrażenia jest przypisanie 0 do `i`.

Ponieważ przypisanie jest operatorem, możemy konstruować łańcuchowe wyrażenia z operatorami przypisania:

```
i = j = k = 0;
```

Operator = jest łączny prawostronnie, więc takie przypisanie jest interpretowane jako:

```
i = (j = (k = 0));
```

W efekcie w pierwszej kolejności nastąpi przypisanie 0 do `k`, następnie wynik przypisania zostanie przypisany do `j`, a wynik tego przypisania — do `i`.

Należy się wystrzegać nieoczekiwanych wyników w łańcuchowych przypisaniach, spowodowanych konwersją typów operandów:

```
int i;
float f;

f = i = 33.3f;
```

Zmienna `i` otrzyma tutaj wartość 33 i przez to do zmiennej `f` przypiszemy 33.0, a nie 33.3.

Zasadniczo przypisanie w postaci *v* `=` *e* jest dozwolone wszędzie tam, gdzie byłaby dozwolona wartość typu *v*. W poniższym przykładzie wyrażenie `j = i` kopiuje wartość `i` do zmiennej `j`. Nowa wartość `j` jest potem dodawana do 1 i tak obliczona wartość jest przypisywana do `k`:

```
i = 1;
k = 1 + (j = i);
printf("%d %d %d\n", i, j, k);      /* wypisuje "1 1 2" */
```

Takie stosowanie operatora przypisania trudno jednak uznać za dobrą praktykę programistyczną. Przede wszystkim dlatego, że „zagnieżdżone przypisania" zmniejszają czytelność programu. Mogą być również przyczyną subtelnych błędów, o których będzie mowa w podrozdziale 4.4.

L-wartości

PiO

Większość operatorów języka C pozwala, aby w roli operandów występowały zmienne, stałe albo wyrażenia (również zawierające inne operatory). Operator przypisania jest o tyle wyjątkowy, że wymaga, aby lewy operand był tak zwaną **l-wartością** (ang. *lvalue*). L-wartość reprezentuje obiekt przechowywany w pamięci komputera. Nie może to być stała ani na przykład wynik porównania. L-wartościami są wszystkie zmienne. Wyrażenia w rodzaju `10` albo `2 * i` l-wartościami nie są. Na razie jedyne l-wartości, które znamy, to właśnie zmienne. W dalszych rozdziałach powiemy sobie także o innych l-wartościach.

Z wymagania, aby lewym operandem operatora przypisania była l-wartość, wynika, że po lewej stronie operatora przypisania nie wolno stosować żadnych wyrażeń:

```
12 = i;          /*** ŹLE ***/
i + j = 0;       /*** ŹLE ***/
-i = j;          /*** ŹLE ***/
```

Takie błędne przypisania są wykrywane przez kompilator — próba skompilowania powyższych instrukcji zaowocuje błędem kompilacji z komunikatem „invalid lvalue in assignment" („niepoprawna l-wartość w przypisaniu").

Przypisania złożone

W programach pisanych w języku C często widzi się przypisania, które przy obliczaniu wartości przypisania bazują na poprzedniej wartości modyfikowanej zmiennej. Oto przykładowe przypisanie dodające 2 do bieżącej wartości zmiennej `i`:

```
i = i + 2;
```

Takie i tym podobne instrukcje można w języku C skracać do postaci **przypisań złożonych** (ang. *compound assignments*). Możemy więc do analogicznej operacji wykorzystać operator złożony `+=`:

```
i += 2;      /* to samo, co i = i + 2 */
```

Operator `+=` dodaje do wartości lewego operandu wartość prawego operandu.

W języku C mamy do dyspozycji dziewięć złożonych operatorów przypisania, w tym:

```
-=      *=      /=      %=
```

inne operatory przypisania ➤ 20.1

(o pozostałych złożonych operatorach przypisania powiemy sobie w jednym z dalszych rozdziałów). Wszystkie operatory przypisań złożonych działają na podobnych zasadach:

`v += e`	dodaje `e` do `v` i zapisuje sumę w `v`
`v -= e`	odejmuje `e` do `v` i zapisuje różnicę w `v`
`v *= e`	mnoży `v` przez `e` i zapisuje iloczyn w `v`
`v /= e`	dzieli `v` przez `e` i zapisuje iloraz w `v`
`v %= e`	oblicza resztę z dzielenia `v` przez `e` i zapisuje wynik `v`

Wcale nie oznacza to, że zapis `v += e` jest zupełnie „tożsamy" z `v = v + e`. Przede wszystkim ze względu na pierwszeństwo operatorów `i *= j + k` to nie to samo co `i = i * j + k`. Są też rzadkie przypadki, kiedy `v += e` jest różne od `v = v + e`, gdy `v` posiada efekty uboczne. Podobne zastrzeżenia dotyczą również pozostałych operatorów przypisania.

PiO

Przy stosowaniu przypisań złożonych trzeba uważać, żeby nie zamienić miejscami znaków w symbolu operatora. Przestawienie znaków może doprowadzić do utworzenia wyrażenia, które będzie poprawne z punktu widzenia składni programu

(kompilator nie zgłosi błędu), ale niepoprawne ze względu na oczekiwane działanie. Jeśli na przykład zamierzamy napisać `i += j`, ale omyłkowo napiszemy `i =+ j`, program jak najbardziej się skompiluje. Niestety, zamiast dodać do `i` wartość `j`, wymusimy wykonanie wyrażenia `i = (+j)`, czyli skopiujemy `j` do `i`.

Operatory przypisań złożonych mają te same właściwości co operatory przypisań prostych. W szczególności są prawostronnie łączne, więc instrukcja:

```
i += j += k;
```

oznacza:

```
i += (j += k);
```

4.3. Operatory inkrementacji i dekrementacji

Do najczęściej używanych operatorów języka C należą operatory inkrementacji (dodania jedynki) i dekrementacji (odjęcia jedynki). Tego rodzaju operacje można, rzecz jasna, zapisać za pomocą zwyczajnych operatorów arytmetycznych:

```
i = i + 1;
j = j - 1;
```

Można też wykorzystać operatory przypisań złożonych:

```
i += 1;
j -= 1;
```

PiO

Ale C oferuje jeszcze krótszy zapis tych operacji przy użyciu operatorów `++` (**inkrementacja**) i `--` (**dekrementacja**).

Na pierwszy rzut oka operatory inkrementacji i dekrementacji to wcielona prostota: `++` dodaje jeden, a `--` odejmuje jeden od operandu. Niestety, prostota jest tu myląca. Operatory inkrementacji i dekrementacji bywają nieoczywiste w użyciu. Wynika to z tego, że operatory inkrementacji (`++`) i dekrementacji (`--`) mogą występować w postaci **przedrostkowej** (np. `++i`, `--i`) albo **przyrostkowej** (np. `i++`, `i--`). Właściwy wybór rodzaju operatora silnie wpływa na poprawność programu.

Kolejna komplikacja tkwi w fakcie, że operatory `++` i `--` posiadają efekty uboczne (tak jak operatory przypisania) — modyfikują wartość operandu. Obliczenie wyrażenia `++i` (inkrementacja przedrostkowa albo tzw. „przed-inkrementacja” `i`) daje wartość `i + 1` oraz — w ramach efektu ubocznego — zwiększa `i` o jeden:

```
i = 1;
printf("i to %d\n", ++i);      /* wypisuje "i to 2" */
printf("i to %d\n", i);        /* wypisuje "i to 2" */
```

Z kolei obliczenie wyrażenia `i++` („po-inkrementacja”) daje wartość `i`, ale w ramach efektu ubocznego `i` jest zwiększane o 1:

```
i = 1;
printf("i to %d\n", i++);      /* wypisuje "i to 1" */
printf("i to %d\n", i);        /*  wypisuje "i to 2" */
```

Pierwsza instrukcja z wywołaniem `printf` wypisze na wyjściu pierwotną wartość `i`, jeszcze sprzed inkrementacji. Drugie wywołanie `printf` wypisze nową wartość `i`. Jak widać, `++i` oznacza „zwiększ natychmiast wartość `i`", a `i++` oznacza „daj bieżącą wartość `i`, a potem zwiększ `i`". Co to znaczy „potem"? Standard języka C nie precyzuje dokładnego momentu wykonania inkrementacji operatora przyrostkowego, ale można bezpiecznie założyć, że `i` będzie miało nową wartość jeszcze przed wykonaniem następnej instrukcji.

PiO

Podobne właściwości ma operator dekrementacji `--`:

```
i = 1;
printf("i to %d\n", --i);      /* wypisuje "i to 0" */
printf("i to %d\n", i);        /* wypisuje "i to 0" */

i = 1;
printf("i to %d\n", i--);      /* wypisuje "i to 1" */
printf("i to %d\n", i);        /* wypisuje "i to 0" */
```

Kiedy operator ++ albo `--` wystąpi wielokrotnie w jednym wyrażeniu, wartość wyrażenia i faktyczne wartości operandów mogą być trudne do ustalenia. Weźmy następujący przykład:

```
i = 1;
j = 2;
k = ++i + j++;
```

Jakie są wartości `i`, `j` i `k` po wykonaniu tych instrukcji? Ponieważ zmienna `i` była inkrementowana *przed* użyciem jej wartości w wyrażeniu, a zmienna `j` była inkrementowana *po* użyciu jej wartości w wyrażeniu, powyższe instrukcje są równoważne następującym:

```
i = i + 1;
k = i + j;
j = j + 1;
```

więc wynikowe wartości zmiennych `i`, `j` i `k` to odpowiednio 2, 3 i 4. Dla porównania wykonanie instrukcji:

```
i = 1;
j = 2;
k = i++ + j++;
```

zostawi zmienne `i`, `j` i `k` z wartościami (odpowiednio) 2, 3 i 3.

Dla porządku wypada dopowiedzieć, że przyrostkowe wersje operatorów ++ i `--` mają wyższy priorytet niż jednoargumentowe operatory +, – i są łączne lewostronnie. Wersje przedrostkowe mają priorytet taki sam jak jednoargumentowe operatory +, – i są łączne prawostronnie.

4.4. Obliczanie wartości wyrażeń

W tabeli 4.2 zestawiono operatory omówione w poprzednich podrozdziałach (podobną tabelę, ale z *kompletem* operatorów, zawiera dodatek A). Pierwsza kolumna określa pierwszeństwo operatorów względem pozostałych operatorów w tabeli (1 to największy priorytet, 5 to priorytet najmniejszy). Ostatnia kolumna opisuje łączność operatorów.

Tabela 4.2. Częściowa lista operatorów języka C

Priorytet	*Nazwa*	*Symbol*	*Łączność*
1	inkrementacja (przyrostkowa)	`++`	lewostronna
	dekrementacja (przyrostkowa)	`--`	
2	inkrementacja (przedrostkowa)	`++`	prawostronna
	dekrementacja (przedrostkowa)	`--`	
	jednoargumentowy plus	`+`	
	jednoargumentowy minus	`-`	
3	multiplikatywne	`*` `/` `%`	lewostronna
4	addytywne	`+` `-`	lewostronna
5	przypisania	`=` `*=` `/=` `%=` `+=` `-=`	prawostronna

Tabela 4.2 (albo jej uzupełniony odpowiednik z dodatku A) będzie bardzo przydatna. Załóżmy, że w kodzie źródłowym (na przykład cudzym) napotkamy taką instrukcję:

```
a = b += c++ - d + --e / -f
```

Takie wyrażenie byłoby znacznie czytelniejsze, gdyby zawierało nawiasy grupujące podwyrażenia. Z pomocą tabeli 4.2 możemy łatwo samodzielnie uzupełnić pogrupować wyrażenie — wystarczy znaleźć w wyrażeniu operator o najwyższym priorytecie i wraz z operandami ująć go w nawias. Od tej pory będzie dla nas pojedynczym operandem. Technikę powtarzamy do momentu rozpoznania wszystkich podwyrażeń.

W naszym przykładzie operatorem o najwyższym priorytecie jest ++, występujący tu jako operator przyrostkowy. Ujmujemy operator z operandami w nawias:

```
a = b += (c++) - d + --e / -f
```

Następny według pierwszeństwa jest operator -- (przedrostkowy) oraz jednoargumentowy minus (oba mają priorytet 2.):

```
a = b += (c++) - d + (--e) / (-f)
```

Drugi znak minusa w wyrażeniu ma operand po lewej stronie, więc jest operatorem odejmowania, a nie kolejnym jednoargumentowym minusem.

Następnie odnajdujemy operator / (priorytet 3.):

```
a = b += (c++) - d + ((--e) / (-f))
```

Wyrażenie zawiera jeszcze dwa operatory o priorytecie 4. (według tabeli 4.2): dodawanie i odejmowanie. Kiedy w wyrażeniu sąsiadują ze sobą dwa operatory o równym priorytecie, należy bardzo uważnie zastosować regułę łączności. W naszym przykładzie – i + otaczają `d`; łączność dwuargumentowych operatorów + i – jest lewostronna, więc stawiamy nawiasy najpierw wokół odejmowania, a potem wokół dodawania:

```
a = b += (((c++) - d) + ((--e) / (-f)))
```

Zostały już tylko przypisania, oba przy operandzie `b`, więc znów trzeba się posiłkować łącznością. Operatory przypisania są łączne prawostronnie (od prawej do lewej), zatem najpierw ujmujemy w nawias podwyrażenie z operatorem +=, a potem podwyrażenie z operatorem =:

```
(a = (b += (((c++) - d) + ((--e) / (-f)))))
```

W ten sposób dokonaliśmy pełnego pogrupowania wyrażenia.

Kolejność obliczania podwyrażeń

Reguły pierwszeństwa i łączności operatorów pozwalają na skuteczne rozbicie dowolnie skomplikowanego wyrażenia na podwyrażenia — możemy dzięki nim pogrupować poszczególne podwyrażenia w nawiasy. Paradoksalnie obie reguły nie zawsze pozwalają na określenie wartości wyrażenia, która może zależeć od kolejności obliczania wartości poszczególnych podwyrażeń.

operatory logiczne `and` i `or` ➤ 5.1
operator warunkowy ➤ 5.2
operator przecinka ➤ 5.3

Język C nie definiuje kolejności obliczania wartości podwyrażeń (z wyjątkiem podwyrażeń angażujących operatory logiczne *and* i *or*, operatory warunkowe i operatory przecinka). Przez to w przypadku wyrażenia `(a + b) * (c – d)` nie możemy mieć pewności, czy jako pierwsze zostanie obliczone podwyrażenie `(a + b)`, czy może `(c –d)`.

Większość wyrażeń ma tę samą wartość niezależnie od tego, w jakiej kolejności zostały obliczone ich podwyrażenia. Bywa jednak, że podwyrażenie modyfikuje jeden ze swoich operandów. Weźmy poniższy przykład:

```
a = 5;
c = (b = a + 2) - (a = 1);
```

Efekt wykonania drugiej instrukcji jest niezdefiniowany. Standard języka C nie mówi nic jednoznacznego na temat wartości tak zbudowanego wyrażenia. W przypadku większości kompilatorów zmienna `c` będzie miała wartość 6 albo 2. Jeśli jako pierwsze zostanie obliczone podwyrażenie `(b = a + 2)`, `b` otrzyma wartość 7, a `c` zostanie obliczone jako 6. Ale jeśli pierwszym obliczonym podwyrażeniem będzie `(a = 1)`, wtedy `b` otrzyma wartość 3, a w `c` znajdzie się wartość 2.

Należy unikać konstruowania wyrażeń, w których odwołujemy się do wartości zmiennej, a równocześnie (w innym podwyrażeniu) bazujemy na tej wartości. Wyrażenie `(b = a + 2) – (a = 1)` zawiera odwołanie do wartości `a` w pierwszym podwyrażeniu oraz modyfikację wartości do `a` w drugim podwyrażeniu (przez przypisanie jedynki). Niektóre kompilatory oznaczają takie wyrażenia komunikatami z ostrzeżeniami w rodzaju „operacja na ‘a’ może być niezdefiniowana”.

Aby zapobiec tego rodzaju problemom, najlepiej unikać stosowania operatora przypisania w podwyrażeniu. Zamiast tego należy wykonać przypisania jako osobne instrukcje. Na przykład nasze wątpliwe wyrażenie może zostać rozbite następująco:

```
a = 5;
b = a + 2;
a = 1;
c = b - a;
```

W ten sposób zagwarantujemy, że zmienna `c` otrzyma wartość 6.

Poza operatorami przypisania mamy jeszcze inne operatory modyfikujące wartość operandu — chodzi o operatory inkrementacji i dekrementacji. Przy korzystaniu z tych operatorów również trzeba uważać, aby nie doprowadzić do powstania wyrażenia, którego poprawna wartość jest zależna od konkretnej kolejności obliczania podwyrażeń, jak tutaj, gdzie do `j` może zostać przypisana jedna z dwóch wartości:

```
i = 2;
j = i * i++;
```

Pozornie jest oczywiste, że `j` otrzyma wartość 4. Niestety, w efekcie wykonania takich instrukcji zmienna `j` może mieć również wartość 6. Oto możliwy scenariusz: (1) najpierw pobierany jest drugi operand (pierwotna wartość `i`); następnie `i` podlega inkrementacji (2) i pobierany jest drugi operand (już nowa wartość `i`); (3) obliczany jest iloczyn obu operandów: liczba 6. „Pobranie" wartości zmiennej oznacza odczytanie wartości z pamięci skojarzonej ze zmienną. Późniejsza zmiana wartości zmiennej w pamięci nie wpływa już na wartość pobraną, bo pobrana wartość jest kopiowana w specjalne miejsce (tzw. *rejestr*) wewnątrz procesora.

Niezdefiniowane zachowanie programu

Według standardu języka C instrukcje takie jak `c = (b = a + 2) - (a = 1)` oraz `j = i * i++` powodują tzw. **niezdefiniowane zachowanie** (ang. *undefined behavior*) programu (patrz podrozdział 4.1). Kiedy program zawiera zachowania niezdefiniowane, nie można już nic powiedzieć o jego wykonaniu. W zależności od użytego kompilatora może się on zachowywać różnie, ale to nie wyczerpuje pojęcia „niezdefiniowanego zachowania". Przede wszystkim program może się w ogóle nie skompilować, skompilowany może się nie uruchomić, a uruchomiony może się wyłożyć, działać niepoprawnie bądź dawać nieznaczące wyniki (np. za każdym uruchomieniem inne). Innymi słowy, przestaje być „programem" — zachowań niezdefiniowanych trzeba więc unikać jak ognia.

4.5. Instrukcje wyrażeniowe

Język C posiada niezwykłą właściwość — tutaj *każde* wyrażenie może być użyte jako instrukcja programu. Każde wyrażenie (niezależnie od jego typu i od tego, co oblicza) może zostać zamienione na instrukcję — wystarczy zakończyć je średnikiem. Na przykład na instrukcję możemy zamienić wyrażenie `++i`:

```
++i;
```

Kiedy dochodzi do wykonania tej instrukcji, następuje zwiększenie wartości `i` o jeden, a później pobierana jest nowa wartość `i` (tak jakby miała za chwilę zostać wykorzystana przy obliczaniu wyrażenia nadrzędnego). Ale skoro `++i` nie jest częścią większego wyrażenia, pobrana wartość jest odrzucana, a program przechodzi do wykonania następnej instrukcji (ale inkrementacja `i` jest oczywiście trwała).

PiO

Skoro wartość instrukcji wyrażeniowej jest odrzucana, nie ma większego sensu wykorzystywanie wyrażeń jako instrukcji, chyba że są to wyrażenia z efektami ubocznymi; one zostaną przecież wykonane mimo odrzucenia obliczonej wartości wyrażenia. Weźmy trzy przykłady. W pierwszym do `i` przypisywana jest wartość 1 — nowa wartość `i` jest pobierana, ale zaraz odrzucana:

```
i = 1;
```

W drugim przykładzie wartość `i` jest pobierana, ale znów nie będzie nigdzie wykorzystana. Za to samo `i` już po pobraniu wartości zostanie zwiększone o jeden:

```
i++;
```

W trzecim przykładzie zostanie obliczona wartość wyrażenia `i * j - 1`, ale obliczona wartość zaraz będzie odrzucona:

```
i * j - 1;
```

Taka instrukcja nie ma żadnego efektu ubocznego, nie zmienia żadnego z operandów, więc jest zwyczajnie bezcelowa.

Bezcelową, a więc pustą instrukcję wyrażeniową można łatwo popełnić przez prostą literówkę. Wystarczy, że zamiast:

```
i = j;
```

przypadkiem napiszemy:

```
i + j;
```

(taki błąd jest prawdopodobny tym bardziej, że znaki + i = zajmują ten sam klawisz na klawiaturze). Niektóre kompilatory wykrywają bezcelowe instrukcje wyrażeniowe, generując przy nich ostrzeżenia w rodzaju „instrukcja bez efektu” („statement with no effect”).

Pytania i odpowiedzi

P: Zauważyłem, że język C nie posiada operatora potęgowania. Jak mam podnosić liczby do potęgi?

O: Jeśli wykładnik potęgi jest niewielką dodatnią liczbą całkowitą, potęgowanie najlepiej zrealizować przez wielokrotne mnożenie (np. `i * i * i` dla obliczenia sześcianu `i`). Do obliczania potęg o wykładnikach niecałkowitych najlepiej wykorzystać funkcję `pow`.

funkcja `pow` ➤ *23.3*

P: Chciałem zastosować operator `%` przy operandzie typu `float`, ale program nie daje się skompilować. Co mogę zrobić (s. 86)?

funkcja `fmod` ➤ 23.3

O: Operator `%` wymaga operandów całkowitych. Spróbuj użyć funkcji `fmod`.

P: Dlaczego działanie operatorów dzielenia (`/`) i reszty z dzielenia (`%`) dla ujemnych operandów jest tak zagmatwane (s. 87)?

O: Zasady działania tych operatorów nie są tak zagmatwane, jakby się wydawało. W obu wersjach standardu celem jest zapewnienie, żeby wartość `(a / b) * b + a % b` zawsze była równa `a` (i faktycznie, oba standardy gwarantują taką zależność, o ile tylko wartość `a / b` jest wartością „reprezentowalną"). Problem polega na tym, że założoną zależność można spełnić na dwa sposoby, przy różnych metodach obliczania `a / b` i `a % b`. Według C89 albo `-9 / 7` to –1 i `-9 % 7` to –2 (równość jest spełniona), albo `-9 / 7` to –2 i `-9 % 7` to 5 (i znów równość jest spełniona). W pierwszym przypadku `(-9 / 7) * 7 + -9 % 7` daje –1×7+–2 = –9, w drugim przypadku `(-9 / 7) * 7 + -9 % 7` to –2×7+ 5 = –9. Do czasu pojawienia się standardu C99 większość procesorów wykonywała już dzielenie całkowite z obcinaniem w kierunku zera, więc taką właśnie regułę dzielenia zapisano w standardzie C99 jako jedyną dozwoloną wartość ilorazu z operandem ujemnym.

C99

P: Skoro w C są l-wartości, czy są też r-wartości (s. 92)?

O: W rzeczy samej. *L-wartość* jest wyrażeniem, które jest dozwolone po *lewej* stronie operatora przypisania; *r-wartość* to wyrażenie, które jest dozwolone po *prawej* stronie. *R-wartość* może więc być zmienną, stałą albo dowolnym wyrażeniem. W niniejszej książce, podobnie jak w standardzie języka C, będziemy trzymać się określenia „wyrażenie", które jednak dość dobrze oddaje istotę r-wartości.

***P: Była mowa o tym, że `v += e` nie jest odpowiednikiem `v = v + e`, jeśli `v` ma efekty uboczne. Jak to rozumieć (s. 93)?**

O: Obliczenie wartości `v += e` powoduje, że wartość `v` jest obliczana tylko raz. `v` w wyrażeniu `v = v + e` jest obliczane dwa razy. Więc jeśli w tym drugim przypadku `v` posiada efekt uboczny, zostanie on wykonany dwukrotnie. W tym przykładzie `i` jest inkrementowane raz:

```
a[i++] += 2;
```

ale jeśli zamiast += użyjemy przypisania =, otrzymamy:

```
a[i++] = a[i++] + 2;
```

Wartość `i` jest równocześnie pobierana i modyfikowana w obrębie jednej instrukcji, więc wynik wykonania takiej instrukcji jest niezdefiniowany. Jest prawdopodobne, że `i` zostanie zwiększone dwukrotnie, ale w istocie nie można nic pewnego powiedzieć o działaniu takiego programu.

P: W jakim celu w C udostępniono operatory ++ i --? Czy są one szybszą metodą inkrementacji i dekrementacji zmiennej, czy są jedynie wygodniejsze (krótsze w zapisie) (s. 94)?

O: Język C odziedziczył operatory ++ i -- w spadku po języku B Kena Thompsona. Thompson wprowadził te operatory, ponieważ jego kompilator B najwyraźniej potrafił efektywniej przetłumaczyć zapis `++i` niż `i = i + 1`. Operatory te

stały się solą języka C (bazuje na nich wiele jego sławnych idiomów). W nowoczesnych kompilatorach stosowanie ++ i -- zapewne ani bardzo nie przyspieszy programu, ani nie zmniejszy bardzo rozmiaru pliku wynikowego. Nieustająca popularność tych operatorów wynika chyba z ich zwartości.

P: Czy operatory ++ i -- działają ze zmiennymi typu `float`?

O: Tak, operacje inkrementacji i dekrementacji można stosować do wartości całkowitych i zmiennoprzecinkowych, jednak w praktyce mało kto próbuje inkrementować albo dekrementować zmienną typu `float`.

***P: Kiedy dokładnie następuje zwiększenie wartości operandu w przypadku przyrostkowych wersji ++ i -- (s. 95)?**

O: Świetne pytanie. Niestety, nie można na nie łatwo odpowiedzieć. Standard języka C wprowadza pojęcie tak zwanego „punktu sekwencji" i mówi, że „aktualizacja składowanej wartości operandu powinna odbyć się pomiędzy poprzednim a następnym punktem sekwencji". W języku C określono kilka różnych punktów sekwencji. Jednym z nich jest koniec instrukcji wyrażeniowej — na końcu instrukcji wyrażeniowej wszystkie opóźnione inkrementacje i dekrementacje powinny zostać wykonane; nie może dojść do rozpoczęcia wykonywania następnej instrukcji z pominięciem tego kroku.

Niektóre operatory, o których powiemy sobie w dalszej części książki (logiczny operator *and*, logiczny operator *or*, operator warunkowy i operator przecinka), również stanowią punkty sekwencji. To samo dotyczy wywołań funkcji — argumenty wywołania funkcji muszą być w pełni obliczone przed wykonaniem wywołania. Jeśli argument wywołania jest wyrażeniem zawierającym przyrostkowy operator ++ albo --, inkrementacja bądź dekrementacja musi zostać wykonana jeszcze przed wykonaniem wywołania funkcji.

P: Co oznacza „odrzucenie" wartości instrukcji wyrażeniowej (s. 99)?

O: Z definicji wyrażenie reprezentuje wartość. Jeśli np. `i` ma wartość 5, to obliczenie wyrażenia `i + 1` daje wartość 6. Zamieńmy to wyrażenie na instrukcję wyrażeniową, dodając średnik na końcu:

```
i + 1;
```

W ramach wykonywania tej instrukcji dochodzi do obliczenia wartości wyrażenia `i + 1`. Ponieważ jednak ta wartość nie jest nigdzie wykorzystywana (nie została przypisana do zmiennej ani nie jest wykorzystywana jako podwyrażenie) — przepada.

P: A co z instrukcjami typu `i = 1;`? Nie widzę, żeby coś tu było tracone.

O: Pamiętajmy, że przypisanie jest w języku C operatorem i jak każdy operator generuje wartość. Przypisanie:

```
i = 1;
```

powoduje zapisanie 1 w zmiennej `i`, ale jako wyrażenie ma wartość przypisania (1) i ta właśnie wartość jest odrzucana. Odrzucanie wartości wyrażenia nie jest jakąś wielką stratą, ponieważ w naszej instrukcji chodziło nam przede wszystkim o zmodyfikowanie zmiennej `i`.

Ćwiczenia

Podrozdział 4.1

1. Napisz, co pojawi się na wyjściu programu wykonującego poniższe instrukcje. Załóż, że `i`, `j` i `k` są zmiennymi typu `int`:

   (a)
   ```
   i = 5; j = 3;
   printf("%d %d", i / j, i % j);
   ```
   (b)
   ```
   i = 2; j = 3;
   printf("%d", (i + 10) % j);
   ```
   (c)
   ```
   i = 7; j = 8; k = 9;
   printf("%d", (i + 10) % k / j);
   ```
   (d)
   ```
   i = 1; j = 2; k = 3;
   printf("%d", (i + 5) % (j + 2) / k);
   ```

W *2. Czy wyrażenie `(-i)/j` będzie miało zawsze tę samą wartość co `-(i/j)`, jeśli `i` i `j` są dodatnimi wartościami całkowitymi? Uzasadnij odpowiedź.

3. Jaka będzie wartość poniższych wyrażeń według standardu C89 (jeśli możliwa jest więcej niż jedna wartość, podaj wszystkie warianty):

   (a) `8 / 5`
   (b) `-8 / 5`
   (c) `8 / -5`
   (d) `-8 / -5`

4. Powtórz ćwiczenie 3. dla wytycznych standardu C99.

5. Jaka będzie wartość poniższych wyrażeń według standardu C89 (jeśli możliwa jest więcej niż jedna wartość, podaj wszystkie warianty):

   (a) `8 % 5`
   (b) `-8 % 5`
   (c) `8 % -5`
   (d) `-8 % -5`

6. Powtórz ćwiczenie 5. dla wytycznych standardu C99.

7. Algorytm obliczania cyfry kontrolnej UPC kończy się następującymi krokami:

   Odejmij 1 od sumy.
   Oblicz resztę z dzielenia zmniejszonej sumy przez 10.
   Odejmij resztę z dzielenia od 9.

   Aż korci, żeby ten algorytm uprościć następująco:

   Oblicz resztę z dzielenia sumy przez 10.
   Odejmij resztę z dzielenia od 10.

   Dlaczego taka poprawka nie zadziała?

8. Czy program *upc.c* będzie wciąż poprawny, jeśli wyrażenie *9 – ((total – 1) % 10)* zostanie zastąpione przez `(10 - (total % 10)) % 10`?

Podrozdział 4.2

W 9. Napisz, co pojawi się na wyjściu programu wykonującego poniższe instrukcje. Załóż, że `i`, `j` i `k` są zmiennymi typu `int`:

(a)
```
i = 7; j = 8;
i *= j + 1;
printf("%d %d", i, j);
```
(b)
```
i = j = k = 1;
i += j += k;
printf("%d %d %d", i, j, k);
```
(c)
```
i = 1; j = 2; k = 3;
i -= j -= k;
printf("%d %d %d", i, j, k);
```
(d)
```
i = 2; j = 1; k = 0;
i *= j *= k;
printf("%d %d %d", i, j, k);
```

10. Napisz, co pojawi się na wyjściu programu wykonującego poniższe instrukcje. Załóż, że `i` i `j` są zmiennymi typu `int`:

(a)
```
i = 6;
j = i += i;
printf("%d %d", i, j);
```
(b)
```
i = 5;
j = (i -= 2) + 1;
printf("%d %d", i, j);
```
(c)
```
i = 7;
j = 6 + (i = 2.5);
printf("%d %d", i, j);
```
(d)
```
i = 2; j = 8;
j = (i = 6) + (j = 3);
printf("%d %d", i, j);
```

Podrozdział 4.3

*11. Napisz, co pojawi się na wyjściu programu wykonującego poniższe instrukcje. Załóż, że `i`, `j` i `k` są zmiennymi typu `int`:

(a)
```
i = 1;
printf("%d ", i++ - 1);
printf("%d", i);
```
(b)
```
i = 10; j = 5;
printf("%d ", i++ - ++j);
printf("%d %d", i, j);
```
(c)
```
i = 7; j = 8;
printf("%d ", i++ - --j);
printf("%d %d", i, j);
```
(d)
```
i = 3; j = 4; k = 5;
printf("%d ", i++ - j++ + --k);
printf("%d %d %d", i, j, k);
```

12. Napisz, co pojawi się na wyjściu programu wykonującego poniższe instrukcje. Załóż, że `i` i `j` są zmiennymi typu `int`:

(a)
```
i = 5;
j = ++i * 3 - 2;
printf("%d %d", i, j);
```
(b)
```
i = 5;
j = 3 - 2 * i++;
printf("%d %d", i, j);
```

```
(c) i = 7;
    j = 3 * i-- + 2;
    printf("%d %d", i, j);
(d) i = 7;
    j = 3 + --i * 2;
    printf("%d %d", i, j);
```

W 13. Które z wyrażeń: `++i` czy `i++` jest dokładnie równoważne wyrażeniu `(i += 1)`? Uzasadnij odpowiedź.

Podrozdział 4.4

14. Pogrupuj podwyrażenia w nawiasy tak, aby zilustrować sposób interpretacji poniższych wyrażeń złożonych:

```
(a) a * b - c * d + e
(b) a / b % c / d
(c) - a - b + c - + d
(d) a * - b / c - d
```

Podrozdział 4.5

15. Podaj wartości zmiennych `i` i `j` po wykonaniu każdej z poniższych instrukcji (początkowa wartość `i` to 1, a początkowa wartość `j` to 2):

```
(a) i += j;
(b) i--;
(c) i * j / i;
(d) i % ++j;
```

Zadania programistyczne

1. Napisz program, który będzie wymagał od użytkownika wprowadzenia liczby dwucyfrowej, a następnie wypisze tę liczbę w odwróconej kolejności cyfr. Sesja z programem powinna przebiegać tak:

```
Podaj liczbę dwucyfrową: 28
Wspak: 82
```

W 2. Rozbuduj program z zadania 1. tak, aby obsługiwał liczby trzycyfrowe.

3. Przerób program z zadania 2. tak, żeby program wypisywał odwrotny zapis liczby trzycyfrowej, bez użycia operacji arytmetycznych do podziału liczby na cyfry. *Wskazówka:* Zajrzyj do programu *upc.c* z podrozdziału 4.1.

4. Napisz program, który wczytuje liczbę wprowadzoną na wejście i wyświetla ją w zapisie ósemkowym:

```
Podaj liczbę pomiędzy 0 i 32767: 1953
W zapisie ósemkowym to: 03641
```

Wyjście programu powinno być wyświetlane z użyciem pięciu cyfr, nawet jeśli zapis liczby nie wymaga ich tylu. *Wskazówka:* Aby zamienić liczbę na reprezentację ósemkową, należy podzielić ją przez osiem. Wynik to pierwsza cyfra zapisu ósemkowego (tutaj: 1). Resztę z dzielenia należy znów podzielić przez osiem i powtarzać proces tak długo, jak długo reszta będzie większa od 8. Ostatnia cyfra to reszta z ostatniego dzielenia (jest też prostszy sposób, bo funkcja `printf` potrafi wypisywać liczby całkowite w zapisie ósemkowym — przekonasz się o tym w rozdziale 7.).

5. Przerób program *upc.c* z podrozdziału 4.1 tak, aby użytkownik wprowadzał 11 cyfr kodu UPC za jednym zamachem:

```
Podaj 11 cyfr kodu UPC: 01380015173
Cyfra kontrolna: 5
```

6. W krajach europejskich stosuje się kody kreskowe z 13 cyframi (tzw. kod EAN). Każdy kod EAN kończy się cyfrą kontrolną (tak jak UPC). Algorytm obliczania cyfry kontrolnej kodu EAN również jest dość podobny:

   Dodaj drugą, czwartą, szóstą, ósmą, dziesiątą i dwunastą cyfrę kodu.
   Dodaj pierwszą, trzecią, piątą, siódmą, dziewiątą i jedenastą cyfrę kodu.
   Pomnóż pierwszą sumę przez 3 i dodaj ją do drugiej sumy.
   Od sumy odejmij 1.
   Oblicz resztę z dzielenia pomniejszonej sumy przez 10.
   Odejmij wynik od 9.

   Na przykład tureckie słodycze Güllüoglu Turkish Delight Pistachio & Coconut mają kod EAN 8691484260008. Pierwsza suma to 6+1+8+2+0+0 = 17, a druga suma to 8+9+4+4+6+0 = 31. Suma iloczynu pierwszej sumy przez 3 i drugiej sumy daje 82. Po odjęciu 1 zostaje 81. Reszta z dzielenia 81 przez 10 to 1. 9 – 1 daje 8. Zgadza się, cyfra kontrolna naszego kodu to dokładnie 8. Zadanie polega na przerobieniu programu *upc.c* z podrozdziału 4.1 tak, aby obliczał cyfrę kontrolną kodu EAN. Użytkownik powinien wprowadzać do programu pierwsze 12 cyfr kodu EAN jednym ciągiem:

```
Podaj 12 cyfr kodu EAN: 869148426000
Cyfra kontrolna: 8
```