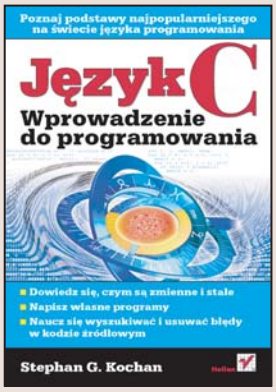
Język C. Wprowadzenie do programowania



Autor: Stephan G. Kochan
Tłumaczenie: Tomasz Żmijewski
ISBN: 83-7361-754-X
Tytuł oryginału: Programming in C. A complete introduction to the C programming language
Format: B5, stron: 504

Język programowania C został stworzony przez Dennisa Ritchie'ego w laboratoriach AT&T Bell na początku lat 70. XX wieku. Jednak dopiero pod koniec lat 70. został spopularyzowany i uzyskał powszechne uznanie. W 1990 roku opublikowana została pierwsza oficjalna wersja standardu ANSI C. Najnowszy standard, znany jako ANSI C99 i ISO/IEC 9899:1999, został opublikowany w 1999 roku.

Język C jest obecnie jednym z najpopularniejszych języków programowania na świecie. Dzięki swoim możliwościom pozwala na stworzenie niemal każdego rodzaju aplikacji. Posiada prostą składnię i niewielki zbiór słów kluczowych, co czyni go stosunkowo łatwym do opanowania. Jednakże dzięki temu, że powstał jako język programowania wykorzystywany do tworzenia systemów operacyjnych, cechuje go ogromna elastyczność i wydajność.

Książka „Język C. Wprowadzenie do programowania" to podręcznik przeznaczony dla osób, które chcą poznać język C i nie mają żadnego doświadczenia w pracy z nim. Opisuje wszystkie elementy języka C zilustrowane krótkimi przykładowymi programami. Przedstawia nie tylko zasady tworzenia programów w C, ale cały proces ich pisania, kompilowania i uruchamiania.

- Języki programowania wysokiego poziomu
- Zmienne, stałe i typy danych
- Pętle i wyrażenia warunkowe
- Definiowanie i stosowanie funkcji
- Wskaźniki i struktury
- Operacje na bitach
- Sterowanie kompilacją za pomocą preprocesora
- Obsługa operacji wejścia i wyjścia
- Usuwanie błędów z programów
- Podstawowe zasady programowania obiektowego

Jeśli chcesz poznać język C, zacznij od tej książki. Znajdziesz w niej wszystko, co musisz wiedzieć o C.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 4.

Zmienne, typy danych i wyrażenia arytmetyczne

W tym rozdziale powiemy więcej o nazwach zmiennych i stałych. Omówimy też dokładnie podstawowe typy danych oraz pewne zasady dotyczące zapisywania wyrażeń arytmetycznych.

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 się 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 3. używaliśmy kilku zmiennych do zapisywania wartości całkowitoliczbowych. W programie 3.4 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 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, 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:

```
tegoroczneePieniadze = 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.

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.

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.

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 `040`. 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"). Więcej informacji na ten temat podajemy w tabeli A 4, w dodatku A.

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 1.7×10^{4}. Wartość przed literą `e` to *mantysa*, zaś 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 $3/16\times2^{10} = 192$.

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, zaś każda wartość niezerowa jako 1.

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

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 6.10 A z rozdziału 6.

W programie 4.1 użyto podstawowych typów danych języka C.

Program 4.1. *Użycie podstawowych typów danych*

```
#include <stdio.h>

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 4.1. *Wyniki*

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

W pierwszej instrukcji programu 4.1 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 kal-

kulator 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. `%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 14.

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 14. dokładnie zajmiemy się typami danych oraz ich konwersjami.

W tabeli 4.1 zestawiono podstawowe typy danych oraz kwalifikatory.

Tabela 4.1. *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`, `5001l`	`%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`

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 4.2 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 4.2. *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 4.2. *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 tym rozdziale 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);
```

może zostać wstawiona do programu 4.2 i ten da taki sam wynik jak poprzednio. Jeśli jednak użyjemy instrukcji:

```
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`.

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 4.2 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 4.3 służy utrwaleniu przekazanej wcześniej wiedzy, pokazuje też zasady arytmetyki całkowitoliczbowej.

Program 4.3. *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 4.3. *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 zauważyli też zapewne, że w pokazywanych dotąd programach każdy operator jest otoczony spacjami. Nie jest to niezbędne, a robione w celu poprawienia estetyki programu. Warto dodać kilka spacji, jeśli czytelność programu na tym zyska.

Wyrażenie z pierwszego wywołania funkcji `printf` w programie 4.3 wskazuje, jak istotne są priorytety operatorów. Wyznaczanie wartości tego wyrażenia odbywa się następująco.

0. Dzielenie ma wyższy priorytet od dodawania, więc wartość zmiennej `a` (25) jest dzielona najpierw przez 5. Wynik pośredni wynosi 5.
0. Mnożenie ma wyższy priorytet od dodawania, więc wynik pośredni 5 jest mnożony przez 2, wartość zmiennej `b`, co daje nowy wynik pośredni 10.
0. 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.

W przedostatniej instrukcji `printf` z programu 4.3 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 role — 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 4.4 spróbujmy zorientować się, jak działa ten operator.

Program 4.4. *Użycie operatora modulo*

```
// Operator modulo

#include <stdio.h>

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

    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 4.4. *Wyniki*

```
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`.

Jak już wiemy, funkcja `printf` wykorzystuje znaki znajdujące się za znakiem procenta jako definicje wyświetlanych wartości. Jeśli jednak za jednym znakiem procenta występuje drugi taki sam znak, funkcja `printf` traktuje ten procent jako zwykły znak do wyświetlenia.

Zgadza się, 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 4.4 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 10.). Czasami dzielenie programu na kilka wierszy może być nie tylko przydatne, ale wprost niezbędne. W programie 4.4 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. W końcu, dodając resztę z dzielenia `a` przez `d` (wyrażenie `a % d`), otrzymujemy ostatecznie 25. Nie jest przypadkiem, że jest to ta sama wartość, jaką 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 4.5 pokazano niektóre rodzaje konwersji. Warto wiedzieć, że niektóre kompilatory mogą generować ostrzeżenia o realizowanych konwersjach.

Program 4.5. *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 4.5. *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 4.5, 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) / (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 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.

Kompilator nie musi obu tych typów obsługiwać[3]. Więcej informacji na ich temat znajdziemy w dodatku A.

Ćwiczenia

0. Przepisz i uruchom pięć programów pokazanych w tym rozdziale. Uzyskane wyniki porównaj z wynikami pokazanymi w tekście.

0. Które z poniższych nazw zmiennych są nieprawidłowe i dlaczego?

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

0. Które z poniższych stałych są nieprawidłowe i dlaczego?

```
123.456     0x10.5    0X0G1
0001        0xFFFF    123L
0Xab05      0L        -597.25
```

[3] W chwili pisania tej książki kompilator `gcc` w wersji 3.3 nie obsługiwał tych typów w pełni.

```
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
```

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

```
C = (F - 32) / 1.8
```

0. 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;
}
```

0. Napisz program wyliczający wartość wielomianu:

$3x^3–5x^2+6$

dla $x = 2.55$.

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

$(3.31\cdot10^{-8}\cdot2.01\cdot10^{-7})/(7.16\cdot10^{-6}+2.01\cdot10^{-8})$

0. 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
```

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

i	j
365	7
12,258	23
996	4