Język C. Programowanie

Autor: Steve Oualline
Tłumaczenie: Piotr Pilch
ISBN: 83-7197-914-2
Tytuł oryginału: Practical C Programming
Format: B5, stron: 460
Przykłady na ftp: 216 kB

Programowanie w języku C jest czymś więcej niż tylko wprowadzaniem kodu zgodnego z regułami składni; styl i wykrywanie błędów odgrywają równie istotną rolę w procesie tworzenia oprogramowania. Istotnie przyczyniają się do prawidłowego działania programów oraz ułatwiają ich konserwację. W książce omówiono więc nie tylko działanie języka C, ale również cały proces tworzenia programu w tym języku (uwzględniając założenia projektowe programu, kodowanie, metody wykrywania błędów, dokumentację, konserwację oraz aktualizację).

Wbrew powszechnemu przekonaniu większość programistów nie spędza całego swojego czasu przy pisaniu kodu. Poświęcają oni wiele czasu na wprowadzanie zmian i poprawek do aplikacji napisanych przez kogoś innego. Po przeczytaniu tej książki będziesz w stanie tworzyć programy, z którymi inni programiści nie będą musieli staczać bojów. Posiądziesz również umiejętność analizowania kodu napisanego przez innego programistę.

Książka „Język C. Programowanie" omawia także popularne zintegrowane środowiska programistyczne dla systemu Windows (Borland C++, Turbo C++ oraz Microsoft Visual C++) oraz narzędzia programistyczne dla systemu UNIX (cc i gcc). Zawarto w niej również kod programu generującego statystyki, który ma za zadanie zademonstrowanie zagadnień omówionych w książce. Książka jest przeznaczona dla osób, które nie miały dotąd doświadczenia w programowaniu oraz dla programistów znających już język C, a pragnących poprawić swój styl i czytelność tworzonego kodu. Jeśli chcesz sprawnie posługiwać się C, „Język C. Programowanie" to idealny podręcznik dla Ciebie.

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

Spis treści

12

Typy złożone

Wielki majestat wielkiego gmachu,
Wybrany przez inkwizytora struktur.

— Wallace Stevens

W tym rozdziale zostaną poruszone następujące zagadnienia:

- Struktury
- Unie
- Instrukcja typedef
- Typ wyliczeniowy enum
- Konwersja typów (rzutowanie)
- Struktury upakowane ora pola bitowe
- Tablice struktur
- Podsumowanie
- Ćwiczenia z programowania

Język C daje programiście do dyspozycji bogaty zestaw typów danych. Poprzez zastosowanie struktur, unii oraz typów wyliczeniowych programista może rozszerzyć język o nowe typy danych.

Struktury

Załóżmy, że piszemy program obsługujący magazyn. W magazynie znajdują się skrzynie, które zawierają różne części. Wszystkie części znajdujące się w jednej skrzyni są jednakowe, dlatego też nie musimy się martwić o to, że zawartość skrzyń zostanie wymieszana.

Na temat każdej skrzyni znane są następujące informacje:

- Nazwa części w niej przechowywanej (łańcuch o długości 30 znaków).
- Liczba części określona ręcznie (liczba całkowita).
- Cena (grosze — liczba całkowita).

W poprzednich rozdziałach do przechowywania grupy podobnych typów danych były stosowane tablice. Jednak w tym przykładzie mamy do czynienia z mieszanką dwóch liczb całkowitych i tekstu.

Zamiast stosować tablicę zostanie użyty nowy typ danych określany terminem *struktury*. W przypadku tablicy wszystkie elementy są tego samego typu i są ponumerowane. W strukturze każdy element lub *pole* posiada nazwę oraz własny typ danych.

Ogólna definicja struktury ma następującą postać:

```
struct nazwa_struktury {
      typ_pola nazwa_pola;   /* komentarz */
      typ_pola nazwa_pola;   /* komentarz */
      ...
} nazwa_zmiennej;
```

Na przykład zdefiniujemy skrzynię, która będzie przechowywać kable do drukarki. Definicja struktury ma następującą postać:

```
struct bin {
       char     name[30];     /* nazwa czesci */
       int      quantity;     /* liczba czesci w skrzyni */
       int      cost;         /* cena pojedynczej czesci (w groszach) */
}  printer_cable_bin;         /* miejsce przechowywania kabli do drukarki */
```

Powyższa definicja informuje język C o dwóch rzeczach. Pierwsza z nich określa format instrukcji `struct bin`. Instrukcja ta definiuje nowy typ danych, który może posłużyć do deklaracji innych zmiennych. Zmienna `sprinter_cable_bin` również jest deklarowana z użyciem tej instrukcji. Ponieważ struktura `bin` została zdefiniowana można ją zastosować w celu deklaracji dodatkowych zmiennych:

```
struct bin terminal_cable_box;   /* miejsce przechowywania kabli do terminala */
```

Część definicji o nazwie *nazwa_struktury* może być pominięta:

```
struct {
       char   name[30];        /* nazwa czesci */
       int      quantity;      /* liczba czesci w skrzyni */
       int      cost;          /* cena pojedynczej czesci (w groszach) */
}  printer_cable_bin;          /* miejsce przechowywania kabli do drukarki */
```

Zmienna `printer_cable_bin` została zdefiniowana, ale nie został utworzony żaden typ danych. Innymi słowy, zmienna ta będzie jedyną zmienną w programie zdeklarowaną w powyższej strukturze. Typem danych takiej zmiennej jest *struktura anonimowa* (ang. *anonymous structure*).

Część *nazwa_zmiennej* może również być pominięta. Poniższy przykład definiuje strukturę typu nie zawierającą zmiennych:

```
struct bin {
       char   name[30];       /* nazwa czesci */
       int      quantity;     /* liczba czesci w skrzyni */
       int      cost;         /* cena pojedynczej czesci (w groszach) */
};
```

Można teraz w celu deklaracji zmiennych, takich jak `printer_cable_bin`, zastosować nowy typ danych (`struct bin`).

W skrajnym przypadku zarówno część *nazwa_zmiennej*, jak i część *nazwa_struktury* mogą być pominięte. Części te są pod względem składni poprawne, ale zupełnie nieprzydatne.

Zdeklarowaliśmy zmienną `printer_cable_bin` zawierającą trzy pola o nazwach: `name`, `quantity` i `cost`. Aby uzyskać do nich dostęp, użyj poniższej instrukcji:

```
zmienna.pole
```

Na przykład jeśli nagle okaże się, że cena kabli wzrosła do 12.95 złotych, musisz wykonać następującą instrukcję:

```
printer_cable_bin.cost = 1295;  /* 12.95 zlotych jest nowa cena */
```

Aby obliczyć wartość całej zawartości skrzyni, wykonaj poniższą instrukcję:

```
total_cost = printer_cable_bin.cost * printer_cable_bin.quantity;
```

Struktury mogą być inicjalizowane w chwili deklaracji poprzez umieszczenie listy elementów w nawiasach klamrowych (`{}`):

```
/*
* Kable do drukarki
*/
struct bin {
       char   name[30];           /* nazwa czesci */
       int      quantity;         /* liczba czesci w skrzyni */
       int      cost;             /* cena pojedynczej czesci (w groszach) */
}  printer_cable_bin = {
       "Kable do drukarki",      /* nazwa czesci znajdujacej sie w skrzyni */
       0,                        /* na poczatku skrzynia jest pusta */
       1295                      /* cena -- 12.95 zlotych */
};
```

Unie

Struktura jest stosowana do definicji typu danych zawierającego kilka pól. Każde pole ma przydzieloną oddzielną przestrzeń, którą zajmuje. Przykładowo poniższa struktura:

```
struct rectangle {
       int   width;
       int    height;
};
```

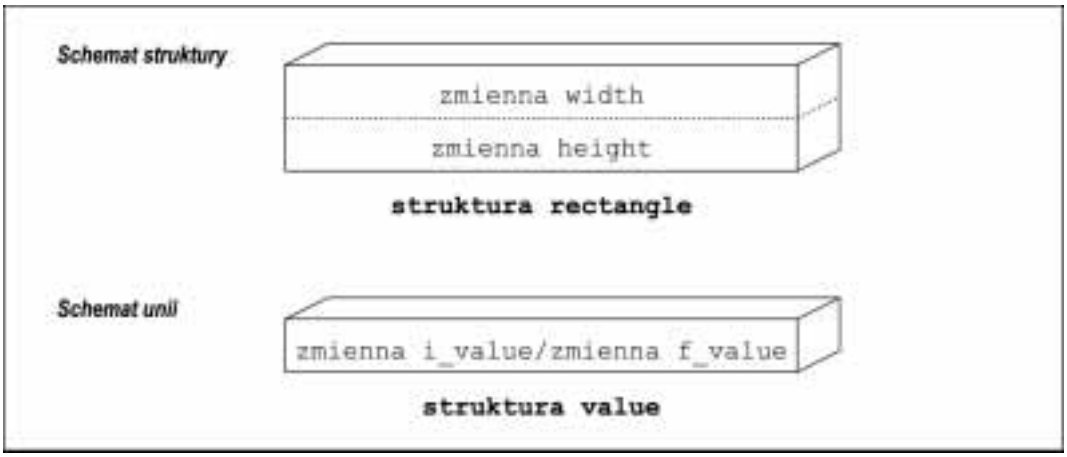
jest przechowywana w pamięci. *Unia* jest podobna do struktury, ale ma przydzielone jedno wspólne miejsce, w których jest przechowywanych wiele różnych nazw pól:

```
union value {
       long int i_value;         /* zmienna typu calkowitego */
       float f_value;            /* zmienna typu zmiennoprzecinkowego */
};
```

Pole `i_value` oraz `f_value` dzielą tę samą przestrzeń pamięci.

Strukturę można sobie wyobrazić jako wielkie pudło podzielone na kilka różnych komórek, z których każda ma inną nazwę. Unia jest też takim pudłem, ale tylko z jedną komórką i kilkoma różnymi etykietami umieszczonymi w jej wnętrzu.

Rysunek 12.1 przedstawia strukturę z dwoma polami. Każdemu polu jest przypisana inna część struktury. Unia zawiera tylko jedną komórkę, do której są przypisane różne nazwy.



Rysunek 12.1. Struktura i unia

W przypadku struktury pola nie oddziałują wzajemnie na siebie. Modyfikacja jednego pola nie wpływa na pozostałe. W przypadku unii wszystkie pola zajmują tę samą przestrzeń, a więc tylko jedno z nich w danej chwili może być aktywne. Innymi słowy, jeśli jakaś wartość zostanie przypisana zmiennej `i_value`, przypisanie wartości zmiennej `f_value` spowoduje usunięcie starej wartości, którą przechowuje zmienna `i_value`.

W wydruku 12.1 pokazano przykład zastosowania unii.

Wydruk 12.1. Zastosowanie unii

```
/*
 * Deklaracja zmiennej przechowujacej liczbe calkowita
 * lub rzeczywista (ale nie obydwie)
 */
```

```
union value {
    long int i_value;      /* Liczba rzeczywista */
    float f_value;         /* Liczba zmiennoprzecinkowa */
} data;
int i;                     /* Losowa liczba calkowita */
float f;                   /* Losowa liczba zmiennoprzecinkowa */
main()
{
    data.f_value = 5.0;
    data.i_value = 3;      /* nadpisana wartosc data.f_value */
    i = data.i_value;      /* prawidlowo */
    f = data.f_value;      /* nieprawidlowo, wywola nieprzewidywalne wyniki */
    data.f_value = 5.5;    /* przypisanie jakiejs wartosci f-value/i-value */
    i = data.i_value;      /* nieprawidlowo, wywola nieprzewidywalne wyniki */
    return(0);
}
```

Unie są często wykorzystywane w branży telekomunikacyjnej. Załóżmy, że posiadamy taśmę umieszczoną w innym miejscu i chcemy wysłać do niej cztery komunikaty: „otwórz", „zamknij", „przeczytaj" oraz „zapisz". Dane zawarte w tych czterech komunikatach zmieniają się w szerokim zakresie zależnie od samych komunikatów.

Komunikat „otwórz" musi zawierać nazwę taśmy, natomiast komunikat „zapisz" musi przenosić dane, które zostaną zapisane. Komunikat „przeczytaj" musi zawierać jak największą liczbę znaków, które zostaną odczytane, natomiast komunikat „zamknij" nie zawiera żadnych dodatkowych informacji.

```
#define DATA_MAX 1024      /* Maksymalna ilosc danych do zapisu i odczytu */

struct open_msg {
     char name[30];        /* Nazwa tasmy */
};

struct read_msg {
     int length;           /* Maksymalna predkosc odczytu */
};

struct write_msg {
     int length;           /* Ilosc bajtow do zapisania */
     char data[DATA_MAX];  /* Zapisywane dane */
};

struct close_msg {
      };

const int OPEN_CODE=0;     /* Kod otwieranego komunikatu */
const int READ_CODE=1;     /* Kod odczytywanego komunikatu */
const int WRITE_CODE=2     /* Kod zapisywanego komunikatu */
const int CLOSE_CODE=3;    /* Kod zamykanego komunikatu */

struct msg {
    int msg;               /* Typ komunikatu */
    union {
      struct open_msg open_data;
      struct read_msg read_data;
      struct write_msg write_data;
      struct close_msg close_data
    } msg_data;
};
```

Instrukcja typedef

Język C umożliwia programiście zdefiniowanie swoich własnych typów danych poprzez zastosowanie instrukcji typedef. Instrukcja ta jest środkiem, dzięki któremu program może rozszerzyć podstawowe typy danych języka C. Ogólna postać instrukcji typedef jest następująca,

```
typedef deklaracja_typu;
```

gdzie *deklaracja_typu* jest identyczna jak deklaracja zmiennej, poza tym że zamiast *nazwy_zmiennej* jest użyta nazwa typu. Na przykład instrukcja

```
typedef int count;
```

definiuje nowy typ `count`, który jest taki sam jak typ całkowity.

Tak więc poniższa deklaracja

```
count flag;
```

jest identyczna jak ta deklaracja

```
int flag;
```

Na pierwszy rzut oka powyższa instrukcja niewiele różni się od tej:

```
#define count int
count flag;
```

Jednak instrukcje typedef mogą być stosowane do definiowania bardziej złożonych obiektów, co wykracza poza zakres możliwości prostej instrukcji #define. Oto przykład:

```
typedef int group[10];
```

Został teraz zdefiniowany nowy typ danych o nazwie `group` określający tablicę dziesięciu liczb całkowitych:

```
main()
{
    typedef int group[10];     /* Utworzenie nowego typu 'group' */
    group totals;              /* Przypisanie nowego typu danych zmiennej */
    for (i = 0; i < 10; i++)
         totals[i] = 0;
    return (0);
}
```

Częstym przykładem użycia instrukcji typedef jest definiowanie nowej struktury. Oto przykład:

```
struct complex_struct  {
    double real;
    double imag;
};
typedef struct complex_struct complex;

complex voltag1 = {3.5, 1.2};
```

Typ wyliczeniowy enum

Wyliczeniowy typ danych został stworzony dla zmiennych, które przechowują tylko ograniczony zbiór wartości. Dostęp do tych wartości odbywa się poprzez nazwę (etykietę). Kompilator wewnętrznie przypisuje każdej takiej nazwie liczbę całkowitą. Rozważmy przykład aplikacji, w której zmienna będzie przechowywać dni tygodnia. Aby utworzyć wartości odpowiadające kolejnym dniom tygodnia w typie danych `week_days`, możemy zastosować deklarację const. Ma ona następującą postać:

```
typedef int week_day;    /* definicja typu week_days */
const int SUNDAY    =0;
const int MONDAY    =1;
const int TUESDAY   =2;
const int WEDNESDAY =3;
const int THURSDAY  =4;
const int FRIDAY    =5;
const int SATURDAY  =6;
/* zastosowanie typu */
week_day today = TUESDAY;
```

Taka metoda jest niewygodna. Lepsza metoda polega na zastosowaniu typu wyliczeniowego **enum**:

```
enum week_day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
        FRIDAY, SATURDAY};
/* zastosowanie typu */
enum week_day today = TUESDAY;
```

Ogólna postać instrukcji **enum** jest następująca:

```
enum nazwa_typu { etykieta-1, etykieta-2, ... }nazwa_zmiennej
```

Podobnie jak w przypadku struktur, część *nazwa_typu* lub *nazwa_zmiennej* może być pominięta. Etykiety mogą być dowolnymi identyfikatorami dopuszczalnymi w języku C, ale zazwyczaj zawierają same małe litery.

Język C implementuje typ wyliczeniowy enum jako zgodny z typem całkowitym, dlatego też całkowicie poprawne jest następujące określenie,

```
today = 5;     /* 5 nie jest typu week_day */
```

chociaż niektóre kompilatory po napotkaniu takiej instrukcji wyświetlą ostrzeżenie. W języku C++ typ wyliczeniowy enum jest oddzielnym typem i nie jest kompatybilny z typem całkowitym.

Konwersja typów (rzutowanie)

Czasem konieczna jest zamiana jednego typu zmiennej na inny. Zadanie to jest realizowane poprzez *operację rzutowania* (ang. *typecast operation*). Ogólna postać rzutowania jest następująca:

```
(typ) wyrażenie
```

Operacja rzutowania nakazuje językowi C obliczenie wartości *wyrażenia*, a następnie konwersję jej typu na określony przez element *typ*. Rzutowanie jest szczególnie przydatne przy przetwarzaniu liczb całkowitych i zmiennoprzecinkowych:

```
int won, lost;      /* liczba gier dotad wygranych/przegranych */
float ratio;        /* wspolczynnik wygrane/przegrane */
won = 5;
lost = 3;
ratio = won / lost; /* wspolczynnik ma wartosc 1.0 (niepoprawna wartosc) */
/* Ponizsze wyrazenie obliczy prawidlowa wartosc wspolczynnika */
ratio = ((float) won) / ((float) lost);
```

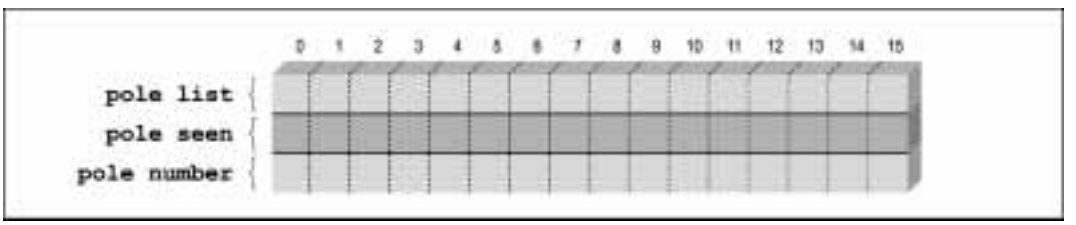
Kolejnym powszechnym zastosowaniem operacji rzutowania jest konwersja wskaźników z jednego typu na inny.

Struktury upakowane oraz pola bitowe

Struktury upakowane umożliwiają deklarowanie struktur w taki sposób, że zajmują one minimalną ilość miejsca. Na przykład poniższa struktura zajmuje 6 bajtów (na platformie 16-bitowej).

```
struct item {
  unsigned int list;   /* jesli pozycja jest na liscie, pole ma wartosc true */
  unsigned int seen;   /* jesli pozycja zostala przypisana polu,
                       /* wtedy ma ono wartosc true */
  unsigned int number; /* numer pozycji */
};
```

Na rysunku 12.2 został pokazany schemat przechowywania zmiennych struktury upakowanej. Każda struktura zajmuje sześć bajtów dostępnej przestrzeni (dwa bajty dla każdego pola typu całkowitego).



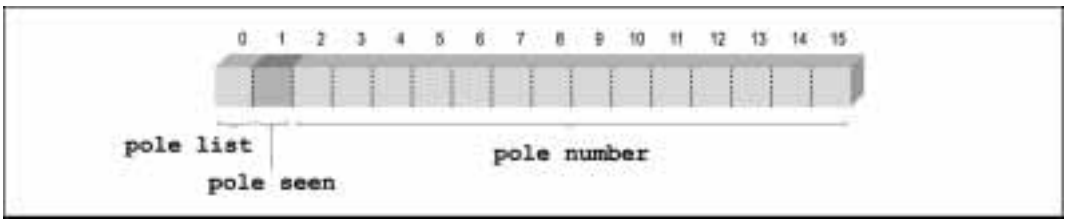
Rysunek 12.2. Struktura zwykła

Pola `list` oraz `seen` mogą przechowywać tylko dwie wartości — `0` i `1` — tak więc do ich reprezentacji jest wymagany tylko jeden bit. Nie planujemy nigdy przekroczyć granicy `16383` pozycji (`0x3fff` lub 14 bitów). W takim razie można zmodyfikować definicję struktury przy użyciu pól bitowych, co polega na umieszczeniu na końcu każdego pola znaku dwukropka oraz liczby określającej liczbę bitów, które zostaną przypisane polu. W efekcie struktura będzie zajmować tylko dwa bajty przestrzeni. Struktura ma postać:

```
struct item {
    unsigned int list:1;         /* jesli pozycja jest na liscie */
                                 /* wtedy pole ma wartosc true */
    unsigned int seen:1;         /* jesli pozycja zostala przypisana polu,*/
```

```
                                    /* wtedy ma ono wartosc true */
    unsigned int number:14;         /* numer pozycji */
};
```

W tym przykładzie poinstruujemy kompilator, aby wykorzystał po jednym bicie dla pól `list` i `seen` oraz 14 bitów dla pola `number`. Tym sposobem dane mogą być upakowane na obszarze tylko dwóch bajtów. Zostało to pokazane na rysunku 12.3.



Rysunek 12.3. Struktura upakowana

Struktury upakowane powinny być stosowane z umiarem. Wynikowy kod, który wykonuje operację rozpakowania danych przechowywanych przez pola bitowe, ma stosunkowo dużą objętość i działa wolno. Jeśli nie występuje problem z brakiem wolnej przestrzeni, struktury upakowane nie powinny być stosowane.

W rozdziale 10., *Preprocesor języka C*, konieczne było zapisanie danych tekstowych oraz pięciu znaczników statusu liczących łącznie 8000 znaków. W takim przypadku zastosowanie dla każdego znacznika oddzielnego bajta pochłonęłoby sporą część przestrzeni (pięć bajtów dla każdego wprowadzonego znaku). Aby pomieścić pięć znaczników w jednym bajcie, zastosowano operacje bitowe. Inna metoda polegałaby na wykorzystaniu struktury upakowanej. Zostało to pokazane poniżej:

```
struct char_and_status {
   char character;          /* Znak przekazany przez urzadzenie */
   int error:1;             /* Po wystapieniu dowolnego bledu przyjmuje wartosc
                               ↳true */
   int framing_error:1;     /* Wystapil blad ramki */
   int parity_error:1;      /* Znak mial nieprawidlowa parzystosc */
   int carrier_lost:1;      /* Sygnal nosny zanikl */
   int channel_down:1;      /* Urzadzenie komunikacyjne stracilo zasilanie */
};
```

Zastosowanie struktur upakowanych w przypadku znaczników jest bardziej czytelne i generuje mniej błędów niż przy użyciu operatorów bitowych. Jednak operatory bitowe dają programiście większą elastyczność, dlatego też sam powinieneś zdecydować, jaką metodę zastosować w zależności od tego, która z nich jest dla Ciebie bardziej zrozumiała i przystępna.

Tablice struktur

Struktury i tablice mogą być ze sobą łączone. Załóżmy, że chcemy zarejestrować czas, jaki potrzebuje biegacz na jedno okrążenie w biegu składającym się z czterech rund. Aby zapisać czas, należy zdefiniować następującą strukturę:

```
struct time {
    int hour;    /*godzina (zegar 24-godzinny ) */
    int minute;  /* 0-59 */
    int second;  /* 0-59 */
};
const int MAX_LAPS = 4; /* tylko cztery okrazenia */
/* najlepszy czas okrazenia w ciagu dnia */
struct time lap[MAX_LAPS];
```

Zastosujemy powyższą strukturę w taki oto sposób:

```
/*
*  Biegacz wlasnie minal punkt pomiarowy
*/
lap[count].hour = hour;
lap[count].minute = minute;
lap[count].second = second;
++count;
```

Powyższa tablica może być zainicjalizowana przy uruchomieniu programu.

Inicjalizacja tablicy struktur jest podobna do inicjalizacji tablic wielowymiarowych. Ma ona następującą postać:

```
struct time start_stop[2] = {
       {10, 0, 0},
       {12, 0, 0}
};
```

Załóżmy, że chcemy napisać program obsługujący listę wysyłkową. Etykiety kopert składają się z 5 wierszy i mają szerokość 60 znaków. Należy zdefiniować strukturę służącą do przechowywania nazwisk i adresów. Lista wysyłkowa dla większości wydruków będzie sortowana według nazwiska, natomiast na potrzeby samej listy zostanie zastosowane sortowanie według kodu pocztowego. Struktura takiej listy wysyłkowej ma następującą postać:

```
struct mailing {
       char   name[60];          /* nazwisko, imie */
       char   address1[60];      /* dwa wiersze zawierajace ulice */
       char   address2[60];
       char   city[40];
       char   state[2];          /* dwuliterowy skrot */
       long int zip;             /* kod pocztowy */
};
```

Można teraz zdeklarować tablicę przechowującą zawartość listy wysyłkowej:

```
/* Lista wysylkowa */
struct mailing list[MAX_ENTRIES];
```

Pole `state` składa się z dwóch elementów, ponieważ zostało przewidziane do przechowywania dwóch znaków. Pole to nie jest typu łańcuchowego, ponieważ nie została przydzielona wystarczająca ilość miejsca dla znaku końca łańcucha (`'\0'`).

Podsumowanie

Struktury i unie są zaliczane do grupy bardziej efektywnych elementów języka C. Dzięki nim nie jesteś już ograniczony tylko do wbudowanych typów danych języka C — możesz definiować własne typy. Jak się okaże w następnych rozdziałach, w celu tworzenia bardzo złożonych i wydajnych struktur danych struktury mogą być łączone ze wskaźnikami.

Ćwiczenia z programowania

Ćwiczenie 12.1. Zaprojektuj strukturę przechowującą dane listy wysyłkowej. Napisz funkcję wyświetlającą dane.

Ćwiczenie 12.2. Zaprojektuj strukturę przechowującą datę i czas. Napisz funkcję znajdującą przesunięcie w minutach pomiędzy dwoma czasami.

Ćwiczenie 12.3. Zaprojektuj strukturę danych dla celów obsługi rezerwacji biletów lotniczych, która zawiera następujące dane:

- Numer lotu
- Kod lotniska odlotu (trzy znaki)
- Kod lotniska przylotu (trzy znaki)
- Czas odlotu
- Czas przylotu

Ćwiczenie 12.4. Napisz program, który generuje listę zawierającą wszystkie samoloty odlatujące z dwóch lotnisk określonych przez użytkownika.