Rozdział 14

Struktury i inne formy danych

W tym rozdziale poznasz:

Słowa kluczowe	Operatory
`struct, union, typedef`	`. ->`

W tym rozdziale dowiesz się, czym są struktury, jak uzyskiwać dostęp do ich poszczególnych składników, oraz jak pisać funkcje przeznaczone do ich przetwarzania. W końcowej części rozdziału będziesz miał również okazję przyjrzeć się poleceniu `typedef`, uniom oraz wskaźnikom do funkcji.

Jednym z najważniejszych etapów projektowania programu jest wybór dobrego sposobu reprezentacji danych. W wielu przypadkach prosta zmienna lub nawet tablica nie wystarcza. Język C podnosi Twoją zdolność przedstawiania danych o kolejny poziom za pomocą *zmiennych strukturalnych* (ang. *structure variables*). Struktury w języku C są w swojej podstawowej postaci wystarczająco uniwersalne, aby nadawać się do przedstawienia dużego zakresu danych, a ponadto pozwalają one konstruować zupełnie nowe formy reprezentacji. Jeśli jesteś obeznany z rekordami w języku Pascal, struktury nie powinny stanowić dla Ciebie problemu. Jeśli nie, nie przejmuj się – po prostu przeczytaj ten rozdział.

Zacznijmy od przestudiowania konkretnego przykładu, który pokaże, w jakich sytuacjach przydaje się struktura, jak ją tworzyć i jak z niej korzystać.

Przykładowy problem: Tworzenie spisu książek

Beatrycze chciałaby móc wyświetlać na ekranie spis książek ze swojej biblioteczki. Chciałaby mieć dostęp do szeregu informacji o każdej z książek: tytułu, autora, wydawnictwa, daty powstania, liczby stron, liczby egzemplarzy oraz wartości w złotych. Niektóre z tych danych, na przykład tytuły, mogą być zapisane w tablicy łańcuchów. Inne dane wymagają tablicy typu `int` lub tablicy typu `float`. Gdyby utworzyć siedem różnych tablic, łatwo byłoby się w nich pogubić, zwłaszcza jeśli Beatrycze chciałaby otrzymać kilka pełnych list: jedną uporządkowaną według tytułu, jedną według autora, jedną według wartości, i tak dalej. Lepszym rozwiązaniem byłoby skorzystanie tylko z jednej tablicy, za to takiej, w której każdy element zawiera wszystkie potrzebne informacje na temat jednej książki.

Beatrycze potrzebuje więc formy danych, która potrafiłaby przechować równocześnie łańcuchy i liczby, i to tak, aby różne rodzaje informacji nie pomieszały się między sobą. Taką formą danych jest struktura. Aby zademonstrować sposób tworzenia i zasadę działania struktury, rozpoczniemy od prostego przykładu. Dla uproszczenia problemu przyjmiemy dwa ograniczenia. Po pierwsze, uwzględnimy tylko tytuł książki, autora oraz aktualną wartość rynkową. Po drugie, ograniczymy spis do jednej książki. Jeśli masz więcej niż jedną książkę, nie martw się – nasz program wkrótce zostanie rozszerzony.

Przyjrzyj się programowi na listingu 14.1 i jego danym wyjściowym. Następnie przeczytaj nasze objaśnienia podstawowych zagadnień.

Listing 14.1. Program `ksiazka.c`.

```
/* ksiazka.c -- spis jednej ksiazki */
#include <stdio.h>
#define MAXTYT  41       /* maksymalna dlugosc tytulu + 1              */
#define MAXAUT  31       /* maksymalna dlugosc nazwiska autora + 1     */
struct ksiazka {         /* szablon struktury o nazwie "ksiazka"       */
    char tytul[MAXTYT];
    char autor[MAXAUT];
    float wartosc;
};                       /* koniec szablonu struktury                  */
int main(void)
{
    struct ksiazka bibl; /* declaracja bibl jako zmiennej typu ksiazka */
    printf("Podaj tytul ksiazki.\n");
    gets(bibl.tytul);                  /* dostep do skladnika "tytul" */
    printf("Teraz podaj autora.\n");
    gets(bibl.autor);
    printf("Teraz podaj wartosc.\n");
    scanf("%f", &bibl.wartosc);
    printf("%s, autor: %s, cena: %.2f zl\n", bibl.tytul,
           bibl.autor, bibl.wartosc);
    printf("%s: \"%s\" (%.2f zl)\n", bibl.autor,
           bibl.tytul, bibl.wartosc);
    return 0;
}
```

Oto przykładowy przebieg działania programu:

```
Podaj tytul ksiazki.
Kurczak z Alp
Teraz podaj autora.
Bismo Lapoult
Teraz podaj wartosc.
59.80
Kurczak z Alp, autor: Bismo Lapoult, cena: 59.80 zl
Bismo Lapoult: "Kurczak z Alp" (59.80 zl)
```

Struktura utworzona w listingu 14.1 składa się z trzech części, zwanych *składnikami* lub *polami* (ang. *members*, *fields*), przechowujących tytuł, autora i wartość książki. Trzema głównymi umiejętnościami, jakie będziesz musiał nabyć, są:

Tworzenie formatu (układu) struktury

Deklarowanie zmiennej o tym formacie

Uzyskiwanie dostępu do poszczególnych składników zmiennej strukturalnej

Deklaracja struktury

Deklaracja struktury jest planem, który opisuje budowę struktury. Wygląda ona następująco:

```
struct ksiazka {
```

```
    char tytul[MAXTYT];
    char autor[MAXAUT];
    float wartosc;
};
```

Deklaracja ta opisuje strukturę złożoną z dwóch tablic znakowych i jednej zmiennej typu `float`. Nie tworzy ona rzeczywistego obiektu w pamięci, a jedynie określa, z czego składa się taki obiekt. (Od czasu do czasu deklarację struktury będziemy nazywać *szablonem*, ponieważ w oparciu o jedną strukturę można utworzyć wiele zmiennych. Szablony w języku C++, o których być może słyszałeś, są czymś zupełnie innym.) Przyjrzyjmy się szczegółom. Na początku deklaracji znajduje się słowo kluczowe `struct`. Wskazuje ono, że to, co po nim następuje, jest strukturą. Kolejnym elementem jest opcjonalna etykieta `ksiazka` – jest ona nazwą przyporządkowaną strukturze. Została ona wykorzystana w dalszej części programu w deklaracji

```
struct ksiazka bibl;
```

Deklaracja ta stwierdza, że `bibl` jest zmienną strukturalną o budowie szablonu `ksiazka`.

Następnym elementem deklaracji struktury jest lista składników zawarta w klamrach. Każdy składnik jest opisany przez swoją własną deklarację zakończoną średnikiem. Na przykład, składnik `tytul` jest tablicą typu `char` posiadającą `MAXTYT` elementów. Składnik może należeć do dowolnego typu danych; może być nawet strukturą!

Definicję budowy struktury kończy klamra zamykająca i średnik. Deklaracja struktury może zostać umieszczona poza wszystkimi funkcjami (zewnętrznie), tak jak w naszym przykładzie, lub w ramach definicji funkcji. W tym drugim przypadku, etykieta jest dostępna tylko w funkcji, w której znajduje się deklaracja. W pierwszym przypadku, etykieta jest widoczna dla wszystkich funkcji w pliku następujących po deklaracji. Na przykład, moglibyśmy użyć w innej funkcji definicji

```
struct ksiazka dickens;
```

Spowodowałoby to utworzenie w ramach tej funkcji zmiennej `dickens` o budowie zgodnej z szablonem `ksiazka`.

Etykieta nie jest elementem obowiązkowym, ale jej użycie jest konieczne w sytuacji, kiedy – tak jak w naszym programie – szablon struktury jest zdefiniowany w jednym miejscu, a oparte na nim zmienne – w drugim. Do tego tematu powrócimy po omówieniu definiowania zmiennych strukturalnych.

Definiowanie zmiennej strukturalnej

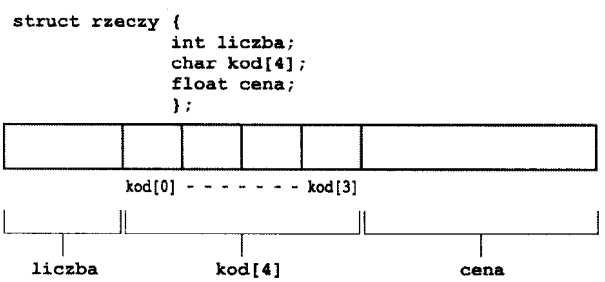
Słowo *struktura* jest używane w dwóch znaczeniach. Pierwszym z nich jest „plan strukturalny", który omówiliśmy przed chwilą. Plan strukturalny informuje kompilator o tym, w jaki sposób mają zostać przedstawione dane, ale nie powoduje przydzielenia tym danym miejsca w pamięci. Kolejnym krokiem jest utworzenie „zmiennej strukturalnej" lub „struktury" w drugim znaczeniu tego słowa. W naszym programie odpowiedzialny jest za to następujący wiersz:

```
struct ksiazka bibl;
```

W odpowiedzi na tę instrukcję kompilator tworzy zmienną `bibl`. Zgodnie z szablonem `ksiazka` rezerwuje on miejsce dla tablicy typu `char` zawierającej `MAXTYT` elementów, takiej samej tablicy zawierającej `MAXAUT` elementów oraz dla zmiennej typu `float`. Wszystkie te dane zostają połączone w jeden obiekt o nazwie `bibl` (patrz rys. 14.1). (Kolejny podrozdział wyjaśnia, w jaki sposób można je „rozłączyć".)

W deklaracji zmiennej strukturalnej `struct ksiazka` pełni dokładnie tę samą rolę, co `int` lub `float` w zwykłych deklaracjach. Na przykład, możliwe jest zadeklarowanie dwóch zmiennych typu `struct ksiazka` na raz lub nawet wskaźnika do tej struktury:

```
struct ksiazka doyle, chandler, *wskks;
```



Rysunek 14.1. Przydzielenie pamięci strukturze.

Każda ze zmiennych strukturalnych `doyle` i `chandler` zawiera pola `tytul`, `autor` i `wartosc`. Wskaźnik `wskks` mógłby wskazywać na `doyle`, `chandler` lub na jakąkolwiek inną strukturę opartą na szablonie `ksiazka`. W istocie rzeczy deklaracja struktury `ksiazka` tworzy nowy typ o nazwie `struct ksiazka`.

Z punktu widzenia komputera deklaracja

```
struct ksiazka bibl;
```

jest skróconą formą deklaracji

```
struct ksiazka {
    char tytul[MAXTYT];
    char autor[MAXAUT];
    float wartosc;
}   bibl;     /* po deklaracji następuje nazwa zmiennej */
```

Innymi słowy, proces deklarowania i definiowania zmiennej strukturalnej może zostać połączony w jedną czynność. Połączenie deklaracji i definicji pozwala pominąć etykietę struktury:

```
struct {          /* brak etytkiety */
    char tytul[MAXTYT];
    char autor[MAXAUT];
    float wartosc;
} bibl;
```

Jeśli jednak masz zamiar użyć szablonu kilka razy, powinieneś użyć etykiety.

Istnieje jeszcze jeden aspekt definiowania zmiennej strukturalnej, który nie jest widoczny w tym przykładzie: inicjalizacja. Przyjrzymy się mu w następnym podrozdziale.

Inicjalizacja struktury

Wiesz już, w jaki sposób inicjalizować zmienne i tablice:

```
int licznik = 0;
int fibo[7] = {0,1,1,2,3,5,8};
```

Czy struktura może również zostać zainicjowana? Tak, chociaż wiele implementacji niezgodnych z ANSI pozwala inicjalizować tylko struktury zewnętrzne i statyczne. To, czy zmienna strukturalna jest zewnętrzna, zależy od miejsca, w którym jest ona zdefiniowana, nie od położenia deklaracji opisującej budowę struktury. W poprzednim przykładzie deklaracja struktury `ksiazka` jest zewnętrzna, w przeciwieństwie do opartej na niej zmiennej `bibl`, która jest zdefiniowana w obrębie funkcji i jako taka należy standardowo do klasy automatycznej.

Strukturę statyczną uzyskujemy w zwykły sposób, czyli za pomocą słowa kluczowego `static`:

```
static struct ksiazka bibl;
```

Aby zainicjalizować strukturę (dowolnej klasy w ANSI C lub klasy nieautomatycznej w starszych kompilatorach), korzystamy ze składni podobnej do tej, którą stosujemy w przypadku tablic:

```
struct book bibl = {
     "Pirat i dziewica",
     "Rene Vivotte"
     7.95
};
```

Mówiąc w skrócie, korzystamy z ujętej w klamry listy wartości rozdzielonych przecinkami. Każda wartość powinna należeć do tego samego typu, co odpowiadający jej składnik struktury. W naszym przypadku oznacza to, że pole `tytul` może zostać zainicjalizowane przez łańcuch, a pole `wartosc` – przez liczbę. Dla zwiększenia czytelności zapisaliśmy każdą wartość w osobnym wierszu, jednak z punktu widzenia kompilatora istotne są tylko przecinki.

Wróćmy teraz do omówienia własności struktur.

Uzyskiwanie dostępu do składników struktury

Struktura jest czymś w rodzaju „supertablicy", w której jeden element może być znakiem, drugi – liczbą zmiennoprzecinkową, a następny – tablicą liczb całkowitych. Poszczególne elementy tablicy wskazujemy za pomocą indeksu. W jaki sposób wskazujemy składniki struktury? Przy pomocy kropki (`.`), operatora przynależności do struktury. Na przykład, `bibl.wartosc` oznacza pole `wartosc` struktury `bibl`. Z wyrażenia `bibl.wartosc` można korzystać dokładnie w taki sam sposób, jak z każdej innej zmiennej typu `float`. Podobnie wyrażenie `bibl.tytul` jest traktowane tak samo, jak zwykła tablica typu `char`. Stąd nasz przykładowy program wykorzystuje wyrażenia, takie jak:

```
gets(bibl.tytul);
```

oraz

```
scanf("%f", &bibl.wartosc);
```

Można powiedzieć, że `.tytul`, `.autor` i `.wartosc` pełnią rolę indeksów struktury `ksiazka`.

Zauważ, że chociaż `bibl` jest strukturą, `bibl.wartosc` jest wartością typu `float` i korzystamy z niej tak samo, jak z każdej innej takiej wartości. Na przykład, instrukcja `scanf("%f",...)` wymaga przekazania adresu wartości typu `float` i właśnie takim adresem jest wyrażenie `&bibl.wartosc`. Kropka ma tutaj wyższy priorytet niż symbol `&`, zatem wyrażenie to jest równoważne `&(bibl.wartosc)`.

Gdyby nasz program zawierał drugą zmienną strukturalną tego samego typu, korzystalibyśmy z niej w taki sam sposób:

```
struct ksiazka adam, juliusz;

gets(adam.tytul);
gets(juliusz.tytul);
```

Przyrostek `.tytul` odnosi się do pierwszego składnika struktury `ksiazka`. Zauważ, że w ostatnim przykładzie wyświetliliśmy zawartość struktury `bibl` w dwóch różnych formatach. Ilustruje to swobodę korzystania ze składników struktury.

Opanowałeś już podstawy, jesteś więc gotów, aby poszerzyć swoje horyzonty i przyjrzeć się tablicom struktur, strukturom struktur, wskaźnikom do struktur oraz funkcjom przetwarzającym struktury.

Tablice struktur

Rozbudujmy nasz program tak, aby mógł on obsługiwać większą liczbę książek. Każda książka może być opisana przez jedną zmienną strukturalną typu `ksiazka`. Aby opisać dwie książki należy więc użyć dwóch takich zmiennych, i tak dalej. Aby przechować dane o kilku książkach, możemy skorzystać z tablicy struktur typu `ksiazka`; to właśnie uczyniliśmy w kolejnej wersji programu, widocznej na listingu 14.2. (Jeśli korzystasz z kompilatora Borland C/C++, zapoznaj się z ramką „Borland C a wartości zmiennoprzecinkowe".)

Struktury a pamięć

Program `ksiazki.c` wykorzystuje tablicę 100 struktur. Ponieważ tablica ta jest obiektem należącym do klasy automatycznej, jest ona zazwyczaj umieszczana na stosie. Może to powodować problemy, ponieważ tablica o tych rozmiarach zajmuje spory obszar pamięci. Jeśli natknąłeś się na błąd wykonania (run-time error), mówiący o rozmiarze stosu lub jego przepełnieniu, domyślna wielkość stosu Twojego kompilatora jest prawdopodobnie zbyt mała dla tego programu. Aby usunąć błąd, możesz: zmienić rozmiar stosu na 10 000 za pomocą opcji kompilatora, uczynić tablicę statyczną lub zewnętrzną (aby nie była ona umieszczana na stosie) lub zmniejszyć ilość elementów tablicy do 16. Dlaczego od razu nie wybraliśmy mniejszego rozmiaru tablicy? Ponieważ powinieneś wiedzieć o potencjalnym problemie z wielkością stosu tak, abyś mógł sobie z nim poradzić, jeśli natkniesz się na niego w toku swoich własnych doświadczeń.

Borland C a wartości zmiennoprzecinkowe

Starsze kompilatory Borland C próbują zmniejszać rozmiar kodu wykonywalnego przez korzystanie z okrojonej wersji funkcji `scanf()`, jeśli program nie używa wartości zmiennoprzecinkowych. Niestety niektóre z nich (Borland C/C++ do wersji 3.1 włącznie) nie radzą sobie z sytuacją, w której wszystkie wartości zmiennoprzecinkowe w programie znajdują się w tablicy struktur (tak jak w listingu 14.2). W rezultacie wyświetlają one następujący komunikat:

```
scanf : floating point formats not linked
Abnormal program termination
```

Jedną z metod obejścia problemu jest dodanie do programu następującego kodu:

```
#include <math.h>
double atrapa = sin(0.0);
```

Zmusza on kompilator do użycia pełnej wersji funkcji `scanf()`.

Listing 14.2. Program `ksiazki.c`.

```
/* ksiazki.c -- spis wielu ksiazek */
#include <stdio.h>
#define MAXTYT   40
#define MAXAUT   40
```

```
#define MAXKS   100              /* maksymalna liczba ksiazek     */
struct ksiazka {                 /* definiuje szablon ksiazka     */
    char tytul[MAXTYT];
    char autor[MAXAUT];
    float wartosc;
};
int main(void)
{
     struct ksiazka bibl[MAXKS]; /* tablica struktur typu ksiazka */
     int licznik = 0;
     int index;

     printf("Podaj tytul ksiazki.\n");
     printf("Aby zakonczyc, wcisnij [enter] na poczatku wiersza.\n");
     while (licznik < MAXKS && gets(bibl[licznik].tytul) != NULL
                         && bibl[licznik].tytul[0] != '\0')
     {
          printf("Teraz podaj autora.\n");
          gets(bibl[licznik].autor);
          printf("Teraz podaj wartosc.\n");
          scanf("%f", &bibl[licznik++].wartosc);
          while (getchar() != '\n')
               continue;               /* czysci wiersz wejsciowy */
          if (licznik < MAXKS)
          printf("Podaj kolejny tytul.\n");
     }
     printf("Oto lista Twoich ksiazek:\n");
     for (index = 0; index < licznik; index++)
     printf("%s, autor: %s, cena: %.2f zl\n", bibl[index].tytul,
          bibl[index].autor, bibl[index].wartosc);
     return 0;
}
```

Oto przykładowy przebieg działania programu:

```
Podaj tytul ksiazki.
Aby zakonczyc, wcisnij [enter] na poczatku wiersza.
Moje zycie jako papuga
Teraz podaj autora.
Mack Zackles
Teraz podaj wartosc.
51.95
Podaj kolejny tytul.
    ...dalsze dane...
Oto lista Twoich ksiazek:
Moje zycie jako papuga, autor: Mack Zackles, cena: 51.95 zl
Teza, antyteza i synteza, autor: Kindra Schlagmeyer, cena: 174.50 zl
Wzrok - nowe spojrzenie, autor: Salome Deschamps, cena: 59.99 zl
Dieta ludzi sukcesu, autor: Buster Downsize, cena: 77.25 zl
Chodzenie krok po kroku, autor: Dr Rubin Thonkwacker, cena: 0.00 zl
Niesforna frywolnosc, autor: Neda McFey, cena: 119.99 zl
Morderstwo w bikini, autor: Mickey Splats, cena: 75.95 zl
Historia Buwanii, Tom 2, autor: Ksiaze Nikoli Buvan, cena: 200.00 zl
Poznaj swoj zegarek elektroniczny, wydanie 2, autor: Miklos Mysz, cena: 75.95 zl
Nadprzewodnictwo dla opornych, autor: Prof. Ed Edison, cena: 23.99 zl
Jak zdobyc przyjaciol i zjednac sobie ludzi, autor: Vlad Dracula, cena 80.00 zl
```

Na początku opiszemy, w jaki sposób deklarować tablice struktur i uzyskiwać dostęp do ich poszczególnych składników. Następnie zwrócimy uwagę na dwa istotne aspekty programu.

Deklarowanie tablicy struktur

Deklaracja tablicy struktur jest taka sama, jak w przypadku każdej innej tablicy:

```
struct ksiazka bibl[MAXKS];
```

Powyższa instrukcja stwierdza, że `bibl` jest tablicą złożoną z `MAXKS` elementów. Każdy element jest strukturą typu `ksiazka`. Stąd `bibl[0]` jest jedną strukturą typu `ksiazka`, `bibl[1]` – drugą taką strukturą, i tak dalej. Jeśli masz kłopoty z wyobrażeniem sobie tego, spójrz na rys. 14.2. Sama nazwa `bibl` nie jest nazwą struktury; jest ona nazwą tablicy, której elementy są strukturami typu `struct ksiazka`.

Wskazywanie składników tablicy struktur

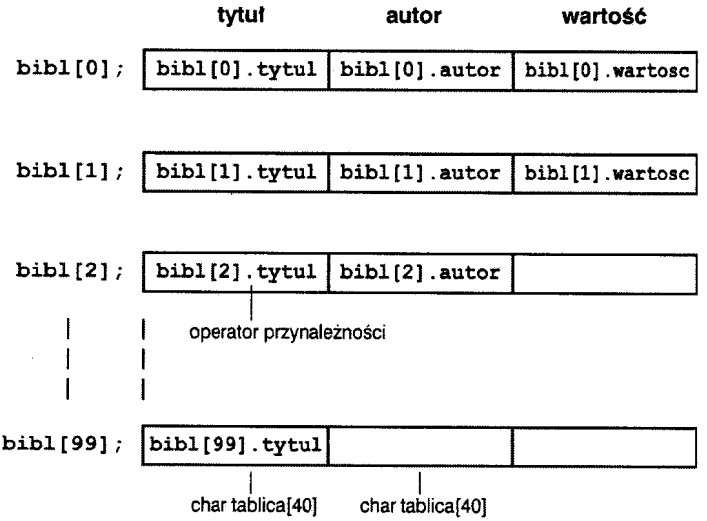
Wskazując składniki tablicy struktur, stosujemy te same zasady, co w przypadku pojedynczej struktury: Do nazwy struktury dodajemy nazwę składnika poprzedzoną kropką.

```
bibl[0].wartosc   /* wartość przechowywana w pierwszym elemencie tablicy */
bibl[4].tytul     /* tytuł przechowywany w piątym elemencie tablicy      */
```

Zauważ, że indeks tablicy znajduje się przed kropką, a nie na końcu wyrażenia:

```
bibl.wartosc[2]   /*  ŹLE   */
bibl[2].wartosc   /* DOBRZE */
```

Przyrostek `.wartosc` możemy bowiem dodać tylko do nazwy struktury; nazwą taką jest `bibl[2]`, ale nie `bibl`.



Rysunek 14.2. Tablica struktur.

Jak sądzisz, co oznacza poniższy zapis?

```
bibl[2].tytul[4]
```

Oznacza on piąty znak w tytule (`tytul[4]`) książki opisanej przez trzecią strukturę (`bibl[2]`). Przykład ten wskazuje, że indeksy położone z prawej strony kropki odnoszą się do składników struktury, a indeksy poprzedzające kropkę – do tablicy struktur.

Szczegóły programu

Główna zmiana w stosunku do pierwszej wersji programu polega na obecności pętli odczytującej dane o kolejnych książkach. Pętla ta rozpoczyna się następującym warunkiem:

```
while (licznik < MAXKS && gets(bibl[licznik].tytul) != NULL
                && bibl[licznik].tytul[0] != '\0')
```

Wyrażenie `gets(bibl[licznik].tytul)` pobiera łańcuch będący tytułem książki; otrzymuje ono wartość `NULL`, jeśli funkcja `gets()` wykryje koniec pliku. Wyrażenie `bibl[licznik].tytul != '\0'` sprawdza, czy pierwszy znak w łańcuchu jest znakiem zerowym – innymi słowy, czy nie wpisano pustego wiersza. Odpowiada to sytuacji, w której użytkownik wcisnął Enter na początku wiersza, i powoduje zakończenie pętli. Pętla zawiera również wyrażenie zapobiegające przepełnieniu tablicy.

W dalszej części programu znajduje się następująca instrukcja:

```
while (getchar() != '\n')
   continue;                /* czysci wiersz wejsciowy */
```

Jak być może pamiętasz z wcześniejszych rozdziałów, powyższy kod wynika z faktu, iż funkcja `scanf()` ignoruje odstępy i znaki nowej linii. Odpowiadając na pytanie o wartość książki, użytkownik wpisuje coś w tym rodzaju:

```
12.50[enter]
```

Powoduje to przesłanie następującego ciągu znaków:

```
12.50\n
```

Funkcja `scanf()` pobiera znaki: `1`, `2`, `.`, `5` oraz `0`, ale pozostawia znak `\n` w strumieniu wejściowym. Gdyby nie pętla `while`, znak ten pobrałoby najbliższe wywołanie funkcji wejścia, czyli instrukcja `gets(bibl[licznik].tytul)`. Zostałby on zinterpretowany jako pusty wiersz, który jest przecież sygnałem powodującym zakończenie programu! Dodany przez nas kod odczytuje znaki do momentu pobrania i porzucenia znaku nowej linii. Nie wykonuje on na znakach żadnych operacji – jego jedynym zadaniem jest usunięcie ich z łańcucha wejściowego tak, aby nie zakłócały one pracy funkcji `gets()`.

Powróćmy teraz do poznawania struktur.

Struktury zagnieżdżone

Czasami zachodzi potrzeba utworzenia struktury zagnieżdżonej, czyli takiej, która zawiera w sobie inną strukturę. Na przykład, Szalala Pieróg buduje strukturę przechowującą dane o jej znajomych. Jednym ze składników takiej struktury są – co oczywiste – dane osobowe znajomego. Dane te mogą być jednak same przedstawione za pomocą struktury złożonej z dwóch składników: imienia i nazwiska. Listing 14.3 jest skondensowanym wynikiem pracy Szalali.

Listing 14.3. Program `znajom.c`.

```
/* znajom.c -- przyklad wykorzystania struktury zagniezdzonej */
#include <stdio.h>
#define DL 20
const char * teksty[5] =
{
    "    Dziekuje za cudowny wieczor, ",
    "Nie przypuszczalam, ze ",
    "moze byc tak wspanialym facetem. Musimy umowic sie",
    "na pyszny ",
    " i spedzic ze soba kilka milych chwil"
};
struct daneos {                     /* pierwszy szablon       */
    char imie[DL];
    char nazw[DL];
};
struct facet {                      /* drugi szablon          */
    struct daneos person;           /* struktura zagniezdzona */
    char ulub_jedz[DL];
    char zawod[DL];
    float dochody;
};
int main(void)
{
    struct facet gosc = {   /* inicjalizacja zmiennej */
        { "Chip", "Hyperlink" },
        "talerz chipsow",
        "makler pamieciowy",
         36827.00
    };

    printf("%s - moj drogi, \n\n", gosc.person.imie);
    printf("%s%s.\n", teksty[0], gosc.person.imie);
    printf("%s%s\n", teksty[1], gosc.zawod);
    printf("%s\n", teksty[2]);
    printf("%s%s%s", teksty[3], gosc.ulub_jedz, teksty[4]);
    if (gosc.dochody > 150000.0)
        puts("!!");
    else if (gosc.dochody > 75000.0)
        puts("!");
    else
        puts(".");
    printf("\n%40s%s\n", " ", "Do zobaczenia wkrotce,");
    printf("%40s%s\n", " ", "Szalala");
    return 0;
}
```

Oto dane wyjściowe:

```
Chip - moj drogi,

    Dziekuje za cudowny wieczor, Chip.
Nie przypuszczalam, ze makler pamieciowy
moze byc tak wspanialym facetem. Musimy umowic sie
na pyszny talerz chipsow i spedzic ze soba kilka milych chwil.
```

```
                                    Do zobaczenia wkrotce,
                                    Szalala
```

Po pierwsze, zauważ sposób, w jaki zagnieżdżona struktura została umieszczona w szablonie. Została ona po prostu zadeklarowana tak samo, jak zwykła zmienna typu `int`:

```
struct daneos person;
```

Deklaracja ta stwierdza, że `person` jest zmienną typu `struct daneos`. Rzecz jasna, plik powinien zawierać również deklarację struktury `daneos`.

Po drugie, zwróć uwagę na sposób uzyskania dostępu do składnika struktury zagnieżdżonej – odbywa się to przez dwukrotne użycie kropki:

```
gosc.person.imie == "Chip"
```

Powyższa konstrukcja jest interpretowana w kierunku od lewej do prawej:

```
(gosc.person).imie
```

Komputer odnajduje strukturę `gosc`, jej składnik o nazwie `person`, a następnie składnik tego składnika o nazwie `imie`.

Wskaźniki do struktur

Miłośników wskaźników z pewnością ucieszy wiadomość, że język C pozwala na tworzenie wskaźników do struktur. Istnieją przynajmniej trzy powody, dla których jest to dobry pomysł. Po pierwsze, wskaźniki do struktur są łatwiejsze do przetwarzania (np. w programie sortującym) niż same struktury, podobnie jak wskaźniki do tablic są bardziej poręczne niż same tablice. Po drugie, w niektórych starszych implementacjach struktura – w przeciwieństwie do wskaźnika – nie może zostać przekazana jako argument do funkcji. Po trzecie, wiele fascynujących form danych wykorzystuje struktury zawierające wskaźniki do innych struktur.

Następny krótki przykład (patrz listing 14.4) pokazuje, w jaki sposób zdefiniować wskaźnik do struktury, oraz jak za jego pomocą uzyskać dostęp do jej składników.

Listing 14.4. Program `znajom.c`.

```
/* znajom.c -- wykorzystuje wskaznik do struktury */
#include <stdio.h>
#define DL 20
struct daneos {
    char imie[DL];
    char nazw[DL];
};
struct facet {
    struct daneos person;
    char ulub_jedz[DL];
    char zawod[DL];
    float dochody;
};
int main(void)
{
    struct facet gosc[2] = {
```

```
        {{ "Chip", "Hyperlink"},
         "talerz chipsow",
         "makler pamieciowy",
         36827.00
        },
        {{"Norbert", "Brzuchacz"},
         "mus lososiowy",
         "redaktor brukowca",
         148500.00
        }
    };
    struct facet * on;        /* oto wskaznik do struktury      */
    printf("adres #1: %p #2: %p\n", &gosc[0], &gosc[1]);
    on = &gosc[0];            /* ustawia wskaznik               */
    printf("wskaznik #1: %p #2: %p\n", on, on + 1);
    printf("on->dochody ma wartosc %.2f $: (*on).dochody ma wartosc %.2f $\n",
         on->dochody, (*on).dochody);
    on++;                     /* wskazuje na nastepna strukture */
    printf("on->ulub_jedz ma wartosc %s:  on->person.nazw ma wartosc %s\n",
         on->ulub_jedz, on->person.nazw);
    return 0;
}
```

Oto dane wyjściowe:

```
adres #1: 0064FD50 #2: 0064FDA4
wskaznik #1: 0064FD50 #2: 0064FDA4
on->dochody ma wartosc 36827.00 $: (*on).dochody ma wartosc 36827.00 $
on->ulub_jedz ma wartosc mus lososiowy:  on->person.nazw ma wartosc Brzuchacz
```

Przyjrzyjmy się najpierw sposobowi utworzenia wskaźnika do struktury `facet`. Następnie wyjaśnimy, w jaki sposób uzyskiwać dostęp do poszczególnych składników struktury za pośrednictwem wskaźnika.

Deklaracja i inicjalizacja wskaźnika do struktury

Deklaracja wskaźnika do struktury jest maksymalnie prosta:

```
struct facet * on;
```

Najpierw słowo kluczowe `struct`, potem etykieta struktury `facet`, a następnie symbol `*` oraz nazwa wskaźnika. Jednym słowem, składnia jest taka sama, jak w przypadku wszystkich innych deklaracji wskaźników, jakie do tej pory widziałeś.

Powyższa deklaracja nie tworzy nowej struktury, a wskaźnik `on`, który może wskazywać na dowolną istniejącą strukturę typu `facet`. Nasz program inicjalizuje wskaźnik `on`, przypisując mu adres struktury `gosc[0]`. Zauważ, że wymaga to użycia operatora adresowego:

```
on = &gosc[0];
```

Pierwsze dwa wiersze danych wyjściowych pokazują, że instrukcja przypisania wykonała swoje zadanie. Jeśli porównasz obydwa wiersze, zobaczysz, że `on` wskazuje na `gosc[0]`, a `on + 1` – na `gosc[1]`. Zauważ, że dodanie `1` do wskaźnika `on` jest równoznaczne z dodaniem 84 do przechowywanego w nim adresu, ponieważ każda struktura `facet` zajmuje 84 bajty pamięci: `person.imie`, `person.nazw`, `ulub_jedz` i `zawod` mają rozmiar 20, a `dochody` – 4, tyle ile

typ `float` na naszym systemie. (W systemie szesnastkowym A4 – 50 = 54 , czyli dziesiętnie 84.) Nawiasem mówiąc, na niektórych komputerach rozmiar struktury może być większy niż suma rozmiarów jej składników. Powodem tego jest wyrównywanie adresów wszystkich składników np. do liczb parzystych lub wielokrotności liczby 4. W takim przypadku struktury mogą zawierać w sobie nieużywane „dziury".

Dostęp do składników za pomocą wskaźnika

Wskaźnik `on` wskazuje na strukturę `gosc[0]`. W jaki sposób można za jego pomocą uzyskać wartość jednego ze składników tej struktury? Trzeci wiersz danych wyjściowych przedstawia dwie metody.

Pierwsza z nich, stosowana najczęściej, wymaga użycia nowego operatora, `->`. Operator ten składa się z myślnika (-), po którym następuje symbol „większy niż" (>). Jego znaczenie ilustruje następująca zależność:

```
on->dochody  jest równe  gosc[0].dochody  jesli  on == &gosc[0]
```

Innymi słowy, wskaźnik do struktury z operatorem `->` działa tak samo, jak nazwa struktury z operatorem `.` (kropką). (Nie można użyć konstrukcji `on.dochody`, ponieważ `on` nie jest nazwą struktury.)

Warto zauważyć, że `on` jest wskaźnikiem, ale `on->dochody` jest składnikiem wskazywanej struktury. W tym przypadku `on->dochody` jest zatem zmienną typu `float`.

Druga metoda uzyskania dostępu do składnika struktury wynika z następującego faktu: jeśli `on == &gosc[0]`, to `*on == gosc[0]`, ponieważ `&` i `*` są operatorami odwrotnymi. Podstawiając nazwę pola, otrzymujemy zatem zależność:

```
gosc[0].dochody == (*on).dochody
```

Nawiasy są wymagane, ponieważ operator `.` ma wyższy priorytet niż `*`.

Podsumowując, jeśli `on` wskazuje na strukturę `gosc[0]`, to równoważne są następujące wyrażenia:

```
gosc[0].dochody == (*on).dochody == on->dochody
```

Przyjrzyjmy się teraz zagadnieniu interakcji między strukturami a funkcjami.

Struktury a funkcje

Jak pamiętasz, argumenty pozwalają przekazywać wartości do funkcji. Każdy argument jest liczbą: wartością `int`, wartością `float`, kodem ASCII znaku, być może adresem. Struktura jest czymś bardziej skomplikowanym niż pojedyncza wartość, nie jest więc zaskoczeniem, że starsze implementacje nie pozwalają przekazywać jej jako argumentu. Ograniczenie to zostało usunięte w nowszych implementacjach, nie ma po nim śladu również w standardzie ANSI C. Tym samym, nowsze kompilatory oferują wybór między przekazaniem do funkcji całej struktury lub jej adresu; w przypadku, gdy istotna jest tylko część struktury, można również przekazać pojedynczy składnik. Przeanalizujemy wszystkie trzy metody, rozpoczynając od przekazywania składników struktur.

Przekazywanie składników struktur

O ile tylko składnik struktury należy do typu podstawowego (czyli jest wartością całkowitą, znakiem, wartością float, wartością double lub wskaźnikiem), może on zostać przekazany jako argument. Ilustruje to amatorski program finansowy przedstawiony na listingu 14.5, który dodaje stan konta bankowego klienta do stanu jego rachunku oszczędnościowo-pożyczkowego.

Listing 14.5. Program fund1.c.

```
/* fund1.c -- przekazywanie skladnikow struktury jako argumentow */
#include <stdio.h>
#define FUNDDL 50
struct fundusze {
    char   bank[FUNDDL];
    double bankfund;
    char   oszcz[FUNDDL];
    double oszczfund;
};
double suma(double, double);
int main(void)
{
    struct fundusze edek = {
        "Bank Czosnkowo-Melonowy",
        2024.72,
        "Kasa Oszczednosciowo-Pozyczkowa \"Debet\"",
        8237.11
    };

    printf("Edek posiada w sumie %.2f zl.\n",
           suma(edek.bankfund, edek.oszczfund) );
    return 0;
}
/* dodaje dwie liczby typu double */
double suma(double x, double y)
{
    return(x + y);
}
```

Oto wynik uruchomienia tego programu:

```
Edek posiada w sumie 10261.83 zl.
```

Ach, wspaniale – program działa. Zauważ, że funkcji suma() nie interesuje fakt, iż argumenty faktyczne są składnikami struktury; wymaga ona jedynie, aby należały one do typu double.

Rzecz jasna, jeśli chcesz, aby wywołana funkcja zmieniła wartość składnika w funkcji wywołującej, możesz przekazać jej adres składnika:

```
zmien(&edek.bankfund);
```

Funkcja zmien() byłaby w stanie zmienić stan konta bankowego Edka.

Korzystanie z adresu struktury

Rozwiążemy ten sam problem, co poprzednio, jednak tym razem jako argument przekażemy adres struktury. Ponieważ funkcja musi znać budowę struktury, aby móc z nią współpracować, nagłówek funkcji musi zawierać odwołanie do szablonu fundusze. Przykładowy program znajduje się na listingu 14.6.

Listing 14.6. Program fund2.c.

```
/* fund2.c -- przekazywanie wskaznika do struktury */
#include <stdio.h>
#define FUNDDL 50
struct fundusze {
    char   bank[FUNDDL];
    double bankfund;
    char   oszcz[FUNDDL];
    double oszczfund;
};
double suma(const struct fundusze *);  /* argument jest wskaznikiem */
int main(void)
{
    struct fundusze edek = {
        "Bank Czosnkowo-Melonowy",
        2024.72,
        "Kasa Oszczednosciowo-Pozyczkowa \"Debet\"",
        8237.11
    };
    printf("Edek posiada w sumie %.2f zl.\n", suma(&edek));
    return 0;
}
double suma(const struct fundusze * pieniadze)
{
    return(pieniadze->bankfund + pieniadze->oszczfund);
}
```

Dane wyjściowe są takie same, jak poprzednio:

```
Edek posiada w sumie 10261.83 zl.
```

Funkcja suma() przyjmuje jeden argument, którym jest wskaźnik (pieniadze) do struktury typu fundusze. Przekazanie jej adresu &edek sprawia, że wskaźnik pieniadze wskazuje na strukturę edek. Wartości edek.bankfund i edek.oszczfund są uzyskiwane za pomocą operatora ->. Ponieważ funkcja nie zmienia zawartości wskazywanej struktury, deklaruje ona pieniadze jako wskaźnik do stałej.

Funkcja suma() ma również dostęp do nazw instytucji finansowych, choć ich nie wykorzystuje. Zauważ, że aby otrzymać adres struktury, musisz skorzystać z operatora &. W odróżnieniu od nazwy tablicy, sama nazwa struktury nie jest bowiem synonimem jej adresu.

Przekazywanie struktury jako argumentu

W kompilatorach, które dopuszczają przekazywanie struktur jako argumentów, nasz przykład może zostać napisany w jeszcze jeden sposób, przedstawiony na listingu 14.7.

Listing 14.7. Program fund3.c.

```
/* fund3.c -- przekazywanie struktury */
#include <stdio.h>
#define FUNDDL 50
struct fundusze {
    char   bank[FUNDDL];
    double bankfund;
    char   oszcz[FUNDDL];
    double oszczfund;
};
double suma(struct fundusze mamona);  /* argument jest struktura */
int main(void)
{
    struct fundusze edek = {
        "Bank Czosnkowo-Melonowy",
        2024.72,
        "Kasa Oszczednosciowo-Pozyczkowa \"Debet\"",
        8237.11
    };
    printf("Edek posiada w sumie %.2f zl.\n", suma(edek));
    return 0;
}
double suma(struct fundusze mamona)
{
    return(mamona.bankfund + mamona.oszczfund);
}
```

Zgodnie z oczekiwaniami, dane wyjściowe wyglądają następująco:

```
Edek posiada w sumie 10261.83 zl.
```

Zmienną pieniadze, która była wskaźnikiem do `struct fundusze`, zastąpiliśmy zmienną `mamona` typu `struct fundusze`. Przy wywołaniu funkcji `suma()` utworzona zostaje zmienna automatyczna `mamona` oparta na szablonie `fundusze`. Następnie składniki struktury `mamona` otrzymują wartości odpowiadających im składników w strukturze `edek`. Tym samym, obliczenia dokonywane są na kopii pierwotnej struktury, podczas gdy w poprzednim programie wykonywane były na oryginale. Ponieważ `mamona` jest strukturą, a nie wskaźnikiem do struktury, funkcja `suma()` korzysta z wyrażenia `mamona.bankfund`, a nie `mamona->bankfund`.

Więcej o nowym, ulepszonym statusie struktury

W nowoczesnych implementacjach języka C, włącznie z implementacjami zgodnymi z ANSI C, struktury mogą być nie tylko przekazywane do funkcji jako argumenty, ale także zwracane za pośrednictwem słowa kluczowego `return`. Aby mechanizm zwracania mógł działać, wartość jednej struktury może zostać przypisana drugiej strukturze. Innymi słowy, jeśli `n_dane` i `o_dane` są strukturami tego samego typu, to w nowszych kompilatorach prawidłowa jest instrukcja:

```
o_dane = n_dane;   /* przypisanie jednej struktury drugiej strukturze */
```

Powoduje ona przypisanie każdemu składnikowi struktury `o_dane` wartości odpowiadającego mu składnika struktury `n_dane`. Podobny manewr jest możliwy przy inicjalizacji struktury:

```
struct imiona koledzy = {"Rudolf", "Jerzy"};
struct imiona znajomi = koledzy;  /* inicjalizacja struktury */
```

Użycie struktury jako argumentu pozwala przekazać do funkcji dane zawarte w strukturze, a zwrócenie struktury za pomocą mechanizmu `return` pozwala przekazać te dane z funkcji wywołanej do funkcji wywołującej. Wskaźniki do struktur również pozwalają na komunikację dwukierunkową, często więc obu metod można używać zamiennie.

Dla skontrastowania ze sobą obu podejść, napiszemy prosty program przetwarzający struktury za pomocą wskaźników, a następnie zmodyfikujemy go wykorzystując przekazywanie i zwracanie struktur. Program prosi o podanie imienia i nazwiska, a następnie informuje o ich całkowitej długości w znakach. Wprawdzie użycie struktur w tak prostym programie jest grubą przesadą, ale pozwala zobaczyć ich działanie w nieskomplikowanym kontekście. Wersja wykorzystująca wskaźniki znajduje się na listingu 14.8.

Listing 14.8. Program `imienaz1.c`.

```
/* imienaz1.c -- wykorzystuje wskazniki do struktury */
#include <stdio.h>
#include <string.h>
struct daneos {
    char imie[20];
    char nazw[20];
    int litery;
};
void pobierz(struct daneos *);
void oblicz(struct daneos *);
void pokaz(const struct daneos *);
int main(void)
{
    struct daneos osoba;

    pobierz(&osoba);
    oblicz(&osoba);
    pokaz(&osoba);
    return 0;
}

void pobierz (struct daneos * wst)
{
    printf("Podaj swoje imie.\n");
    gets(wst->imie);
    printf("Podaj swoje nazwisko.\n");
    gets(wst->nazw);
}

void oblicz (struct daneos * wst)
{
    wst->litery = strlen(wst->imie) +
                  strlen(wst->nazw);
}

void pokaz (const struct daneos * wst)
{
    printf("%s %s, Twoje imie i nazwisko skladaja sie z %d liter.\n",
        wst->imie, wst->nazw, wst->litery);
}
```

Skompilowanie i uruchomienie programu daje następujący wynik:

```
Podaj swoje imie.
Wiola
Podaj swoje nazwisko.
Plunderfest
Wiola Plunderfest, Twoje imie i nazwisko skladaja sie z 16 liter.
```

Działanie programu opiera się na trzech funkcjach wywoływanych z funkcji `main()`. W każdym przypadku przekazywany jest adres struktury `osoba`.

Funkcja `pobierz()` przesyła dane do funkcji `main()`.Pobiera ona imię i nazwisko użytkownika i umieszcza je w strukturze `osoba`, którą lokalizuje za pomocą wskaźnika `wst`. Jak pamiętasz, `wst->nazw` oznacza składnik `nazw` struktury wskazywanej przez `wst`. Czyni to `wst->nazw` odpowiednikiem nazwy tablicy typu `char`, a tym samym prawidłowym argumentem dla funkcji `gets()`. Zauważ, że choć funkcja `pobierz()` przekazuje informacje do głównego programu, nie wykorzystuje ona w tym celu mechanizmu `return` – dlatego należy ona do typu `void`.

Funkcja `oblicz()` przesyła informacje w dwóch kierunkach. Za pomocą wskaźnika odnajduje ona dwa łańcuchy przechowywane w strukturze `osoba`. Korzystając z funkcji bibliotekowej `strlen()`, oblicza ona całkowitą liczbę liter w imieniu i nazwisku, a następnie przekazuje ją na zewnątrz przy pomocy adresu struktury. Tak jak poprzednia funkcja, `oblicz()` należy do typu `void()`.

Funkcja `pokaz()` również lokalizuje potrzebne informacje za pomocą wskaźnika. Ponieważ nie zmienia ona zawartości struktury, jej nagłówek zawiera słowo `const`.

W czasie wszystkich tych operacji istniała tylko jedna zmienna strukturalna `osoba`, z której – za pośrednictwem adresu – korzystała każda kolejna funkcja. W zależności od funkcji dane były przekazywane od funkcji wywołanej do wywołującej, od funkcji wywołującej do wywołanej lub w obydwu kierunkach.

Zobaczmy teraz, w jaki sposób można wykonać to samo zadanie korzystając z przekazywania i zwracania struktur. Po pierwsze, aby przekazać strukturę do funkcji, należy użyć argumentu `osoba`, a nie `&osoba`. Tym samym, odpowiedni argument formalny powinien *należeć* do typu `struct daneos` zamiast *być wskaźnikiem* do tego typu. Po drugie, aby dostarczyć strukturę do funkcji `main()`, wystarczy po prostu ją zwrócić. Druga wersja programu przedstawiona jest na listingu 14.9.

Listing 14.9. Program `imienaz2.c`.

```
/* imienaz2.c -- przekazuje i zwraca struktury */
#include <stdio.h>
#include <string.h>
struct daneos {
    char imie[20];
    char nazw[20];
    int litery;
};
struct daneos pobierz(void);
struct daneos oblicz(struct daneos);
void pokaz(struct daneos);
int main(void)
{
    struct daneos osoba;

    osoba = pobierz();
    osoba = oblicz(osoba);
    pokaz(osoba);
    return 0;
}
```

```
struct daneos pobierz(void)
{
    struct daneos temp;
    printf("Podaj swoje imie.\n");
    gets(temp.imie);
    printf("Podaj swoje nazwisko.\n");
    gets(temp.nazw);
    return temp;
}

struct daneos oblicz(struct daneos info)
{
    info.litery = strlen(info.imie) + strlen(info.nazw);
    return info;
}

void pokaz(struct daneos info)
{
    printf("%s %s, Twoje imie i nazwisko skladaja sie z %d liter.\n",
        info.imie, info.nazw, info.litery);
}
```

Ta wersja programu daje taki sam wynik, jak poprzednia, ale działa ona w inny sposób. Każda z trzech funkcji tworzy swoją własną kopię struktury `osoba`, a więc program ten wykorzystuje nie jedną, a cztery oddzielne struktury.

Na przykład, zastanówmy się nad funkcją `oblicz()`. W pierwszym programie przekazany został adres struktury `osoba`, zatem funkcja wykonywała wszystkie operacje na prawdziwych, oryginalnych wartościach. W tej wersji programu tworzona jest nowa struktura o nazwie `info`, do której skopiowane zostają wartości zapisane w strukturze `osoba`. Funkcja ma dostęp jedynie do kopii – liczba liter zostaje więc umieszczona w strukturze `info`, a nie `osoba`. Z pomocą przychodzi tu jednak mechanizm `return`. Wiersz

```
return info;
```

w funkcji `oblicz()` w połączeniu z wierszem

```
osoba = oblicz(osoba);
```

kopiują wartości ze struktury `info` do struktury `osoba`. Zauważ, że funkcja `oblicz()` należy do typu `struct daneos`, ponieważ zwraca ona strukturę.

Struktury czy wskaźniki do struktur?

Załóżmy, że chcesz napisać funkcję przetwarzającą struktury. Czy jej argumentem powinien być wskaźnik do struktury czy sama struktura? Każde z podejść ma swoje mocne i słabe strony.

Dwoma zaletami metody opartej na wskaźnikach są: dostępność zarówno w starych, jak i nowych implementacjach C oraz szybkość – przekazywany jest tylko jeden adres. Wadą jest słabsza ochrona danych. Niektóre operacje w funkcji wywołanej są w stanie dokonać nieodwracalnych zmian w wyjściowej strukturze. Na szczęście w ANSI C problem ten rozwiązuje kwalifikator `const`. Na przykład, jeśli w funkcji `pokaz()` umieścisz kod, który zmienia jakikolwiek składnik struktury, kompilator uzna to za błąd.

Zaletą przekazania struktury jako argumentu jest to, iż funkcja przetwarza kopie danych, co jest bezpieczniejsze niż operowanie na danych oryginalnych. Ponadto, podejście to charakteryzuje się czytelniejszym stylem zapisu. Załóżmy, że zdefiniowaliśmy następujący typ strukturalny:

```
struct wektor = {double x; double y;};
```

Chcemy przypisać wektorowi odp sumę wektorów a i b. Możemy w tym celu utworzyć funkcję przekazującą i zwracającą struktury; kod programu wyglądałby wówczas następująco:

```
struct wektor odp, a, b, suma_wekt();
...
odp = suma_wekt(a,b);
```

Powyższy kod wygląda bardziej naturalnie niż wersja wykorzystująca wskaźniki:

```
struct wektor odp, a, b;
void suma_wekt();
...
suma_wekt(&a, &b, &odp);
```

Oprócz tego, w przypadku korzystania ze wskaźników, programista musi pamiętać, czy adres sumy powinien być pierwszym czy ostatnim argumentem.

Przeciwko przekazywaniu struktur przemawia to, iż mogą go nie dopuszczać starsze implementacje, oraz to, że stanowi ono marnotrawstwo czasu i pamięci. Szczególnie rozrzutne jest przekazywanie dużej struktury do funkcji, która wykorzystuje tylko jeden lub dwa jej elementy. W takim przypadku bardziej sensowne jest przekazanie wskaźnika lub tylko wymaganych składników struktury.

Mając na uwadze wydajność, programiści zwykle przekazują do funkcji wskaźniki, a nie całe struktury, w razie potrzeby chroniąc dane za pomocą kwalifikatora const. Przekazywanie struktur jest najczęściej stosowane w przypadku struktur o niewielkich rozmiarach.

Tablice znakowe lub wskaźniki do znaków w strukturze

Przedstawione do tej pory struktury przechowywały łańcuchy w formie tablic znakowych. Być może zastanawiałeś się, czy zamiast tablic można zastosować wskaźniki do char. Na przykład, listing 14.3 zawiera poniższą deklarację:

```
#define DL 20
struct daneos {
    char imie[LEN];
    char nazw[LEN];
};
```

Czy mógłbyś użyć następującego zapisu?

```
struct wdaneos {
    char * imie;
    char * nazw;
};
```

Odpowiedź brzmi „tak", ale możesz narazić się na kłopoty, jeśli nie rozumiesz jego konsekwencji. Zastanów się nad poniższym kodem:

```
struct daneos vprez = {"Natalia", "Lato"};
struct wdaneos skarb = {"Ebenezer", "Scrooge"};
printf("%s i %s\n", vprez.imie, skarb.imie);
```

Jest to poprawny i działający kod, ale zastanów się, gdzie przechowywane są łańcuchy. W przypadku zmiennej vprez typu struct daneos, łańcuchy znajdują się wewnątrz struktury, zajmując w sumie 40 bajtów. W przypadku zmiennej skarb typu struct wdaneos, łańcuchy znajdują się w miejscu, w którym kompilator przechowuje wszystkie inne stałe łańcuchowe. Sama struktura zawiera jedynie dwa adresy, które – na naszym systemie – zajmują w sumie 8 bajtów. Należy zauważyć, że struktura struct wdaneos nie rezerwuje miejsca dla łańcuchów! w związku z tym jej wskaźniki mogą przechowywać tylko adresy łańcuchów, które znajdują się już w pamięci, takich jak stałe łańcuchowe czy łańcuchy w tablicach.

Ograniczenie to jest istotne na przykład w następującej sytuacji:

```
struct daneos ksiegowy;
struct wdaneos adwokat;
puts("Podaj nazwisko twojego ksiegowego:");
scanf("%s", ksiegowy.nazwisko);
puts("Podaj nazwisko twojego adwokata:");
scanf("%s", adwokat.nazwisko);    /* tu jest niebezpieczeństwo */
```

Z punktu widzenia składni powyższy kod jest prawidłowy. Ale gdzie zostają zapisane dane? w przypadku księgowego nazwisko zostaje umieszczone w ostatnim składniku zmiennej ksiegowy, czyli w tablicy znakowej. W przypadku adwokata, program nakazuje funkcji scanf() umieszczenie łańcucha pod adresem adwokat.nazwisko. Ponieważ zmienna adwokat.nazwisko nie została zainicjalizowana, adres ten może być dowolny, a więc program zapisze łańcuch w przypadkowym miejscu. Jeśli będziesz miał szczęście, program będzie działał, przynajmniej przez większość czasu. Jeśli nie, program może zdestabilizować działanie komputera. Prawdę powiedziawszy, byłoby lepiej, gdyby program nie działał, ponieważ w przeciwnym wypadku nie zauważysz, że zawiera on niebezpieczny błąd.

Jeśli więc tworzysz strukturę przechowującą łańcuchy, korzystaj z tablic znakowych. Wskaźniki do char mają swoje zastosowania, ale zostawiają one otwarte pole dla poważnych nadużyć.

Funkcje korzystające z tablic struktur

Załóżmy, że mamy tablicę struktur, którą chcemy przetworzyć za pomocą funkcji. Nazwa tablicy jest synonimem jej adresu, może więc zostać przekazana jako argument. Rzecz jasna, funkcja potrzebuje również dostępu do szablonu struktury. Aby to zilustrować, listing 14.10 rozszerza nasz program finansowy na dwóch ludzi przy wykorzystaniu tablicy dwóch struktur typu fundusze.

Listing 14.10. Program `fund4.c`.

```
/* fund4.c -- przekazywanie tablicy struktur do funkcji */
#include <stdio.h>
#define FUNDDL 50
#define N 2
struct fundusze {
    char   bank[FUNDDL];
    double bankfund;
    char   oszcz[FUNDDL];
    double oszczfund;
};
double suma(const struct fundusze *pieniadze, int n);
int main(void)
{
    struct fundusze kowalski[N] = {
        {
            "Bank Czosnkowo-Melonowy",
```

```
            2024.72,
            "Kasa Oszczednosciowo-Pozyczkowa \"Debet\"",
            8237.11
        },
        {
            "Bank \"Uczciwy Jan\"",
            1834.28,
            "Kasa oszczednosciowa \"Chomik\"",
            2903.89
        }
    };

    printf("Kowalscy posiadaja w sumie %.2f zl.\n",
           suma(kowalski,N));
    return 0;
}

double suma(const struct fundusze *pieniadze, int n)
{
    double kwota;
    int i;

    for (i = 0, kwota = 0; i < n; i++, pieniadze++)
        kwota += pieniadze->bankfund + pieniadze->oszczfund;
    return(kwota);
}
```

Dane wyjściowe wyglądają tak:

```
Kowalscy posiadaja w sumie 15000.00 zl.
```

(Cóż za okrągła suma! Można by pomyśleć, że liczby zostały specjalnie dobrane.)

Nazwa tablicy `kowalski` jest równocześnie jej adresem. W szczególności jest ona adresem pierwszego elementu tablicy, czyli struktury `kowalski[0]`. Stąd, w momencie wywołania funkcji `suma()` wskaźnik `pieniadze` ma wartość `&kowalski[0]`.

Operator `->` umożliwia dodanie dwóch kwot pieniędzy zaoszczędzonych przez pierwszego Kowalskiego. (Odbywa się to tak samo, jak w listingu 14.6.) Następnie pętla `for` zwiększa wskaźnik `pieniadze` o 1; teraz wskazuje on na kolejną strukturę (`kowalski[1]`), dzięki czemu do sumy mogą zostać dodane oszczędności drugiego członka rodziny Kowalskich.

Oto najistotniejsze informacje dotyczące programu:

Argumentem funkcji jest nazwa tablicy, czyli adres pierwszej struktury.

Wskaźnik zawierający adres jest zwiększany tak, aby wskazywał kolejne struktury w tablicy. Zauważ, że użycie adresu pierwszej struktury, np. w wywołaniu

```
suma(&kowalski[0],N)
```

miałoby taki sam efekt, jak użycie nazwy tablicy, ponieważ oba wyrażenia odpowiadają temu samemu miejscu w pamięci.

Ponieważ funkcja `suma()` nie powinna zmieniać danych w funkcji `main()`, jej argument został zadeklarowany ze słowem kluczowym `const`.

Zapisywanie zawartości struktury w pliku

Struktury, z uwagi na ich zdolność przechowywania różnorodnych rodzajów informacji, są ważnym narzędziem w konstruowaniu baz danych. Na przykład, za pomocą struktury mógłbyś przechować wszystkie potrzebne informacje o pracodawcy lub części samochodowej. Prędzej czy później zaistniałaby konieczność zapisu struktur w pliku, a wraz z nią konieczność ich odczytywania. Właśnie tymi zagadnieniami zajmiemy się w tym podrozdziale.

Plik bazy danych może zawierać praktycznie dowolną liczbę struktur. Pełny zestaw informacji przechowywanych w strukturze nazywamy *rekordem* (ang. *record*), a poszczególne składniki – *polami* (ang. *fields*).

Prawdopodobnie najbardziej oczywisty sposób zapisania rekordu – użycie funkcji `fprintf()` – jest równocześnie sposobem najmniej efektywnym. Przypomnij sobie strukturę `ksiazka` użytą w listingu 14.1:

```
#define MAXTYT   40
#define MAXAUT   40
struct ksiazka {
    char tytul[MAXTYT];
    char autor[MAXAUT];
    float wartosc;
};
```

Zakładając, że `pksiazka` jest wskaźnikiem plikowym, informacje zawarte w zmiennej `slownik` typu `struct ksiazka` można zapisać w pliku za pomocą następującej instrukcji:

```
fprintf(pksiazka, "%s %s %.2f\n", slownik.tytul,
        slownik.autor, slownik.wartosc);
```

Metoda ta staje się dość nieporęczna w przypadku struktur zawierających np. 30 składników. Ponadto, stwarza ona problem przy odczytywaniu, ponieważ program musiałby wiedzieć, gdzie kończy się jedno pole, a zaczyna drugie. Problem ten można wyeliminować przez użycie formatu o stałej szerokości pola, np. `"%39s%39s%8.2f\n"`, jednak wówczas trzeba pamiętać o określeniu pól o wystarczającym rozmiarze.

Lepszym rozwiązaniem jest użycie funkcji `fread()` i `fwrite()` do odczytu i zapisu jednostek o rozmiarze struktury. Jak pamiętasz, funkcje te zapisują i odczytują informacje korzystając z tej samej binarnej reprezentacji, której używa program. Na przykład, instrukcja

```
fwrite(&slownik, sizeof (struct ksiazka), 1, pksiazka);
```

kopiuje całość struktury `slownik` do pliku związanego ze wskaźnikiem `pksiazka`. Argument `sizeof (struct ksiazka)` informuje funkcję o długości bloku, a liczba `1` wskazuje, że należy skopiować tylko jeden blok. Funkcja `fread()` w przypadku przekazania jej tych samych argumentów kopiuje porcję danych o rozmiarze struktury z pliku pod adres `&slownik`. Krótko mówiąc, funkcje `fread()` i `fwrite()` odczytują i zapisują dane po jednym rekordzie zamiast po jednym polu.

Aby pokazać, jak funkcje `fread()` i `fwrite()` mogą zostać wykorzystane w programie, zmodyfikowaliśmy listing 14.2 tak, aby tytuły książek były zapisywane w pliku o nazwie `ksiazki.dat`. Jeśli plik o takiej nazwie już istnieje, program wyświetla jego aktualną zawartość i umożliwia dopisanie danych na jego końcu. Program znajduje się na listingu 14.11. (Jeśli korzystasz z jednego ze starszych kompilatorów firmy Borland, zapoznaj się z ramką „Borland C a wartości zmiennoprzecinkowe" położoną w pobliżu listingu 14.2.)

Listing 14.11. Program `ksplik.c`.

```
/* ksplik.c -- zapisuje zawartosc struktury w pliku */
#include <stdio.h>
#include <stdlib.h>
#define MAXTYT   40
#define MAXAUT   40
#define MAXKS    10                  /* maksymalna liczba ksiazek   */
struct ksiazka {                     /* utworzenie szablonu ksiazka */
    char tytul[MAXTYT];
    char autor[MAXAUT];
    float wartosc;
};
int main(void)
{
     struct ksiazka bibl[MAXKS]; /* tablica struktur */
     int licznik = 0;
     int index, licznikp;
     FILE * pksiazki;
     int rozmiar = sizeof (struct ksiazka);

     if ((pksiazki = fopen("ksiazki.dat", "a+b")) == NULL)
     {
         fputs("Nie moge otworzyc pliku ksiazki.dat\n",stderr);
         exit(1);
     }
     rewind(pksiazki);            /* przejscie na poczatek pliku */
     while (licznik < MAXKS &&  fread(&bibl[licznik], rozmiar,
                1, pksiazki) == 1)
     {
         if (licznik == 0)
             puts("Biezaca zawartosc pliku ksiazki.dat:");
         printf("%s by %s: $%.2f\n",bibl[licznik].tytul,
             bibl[licznik].autor, bibl[licznik].wartosc);
         licznik++;
     }
     licznikp = licznik;
     if (licznik == MAXKS)
     {
         fputs("Plik ksiazki.dat jest pelny.", stderr);
         exit(2);
     }
     puts("Podaj nowe tytuly ksiazek.");
     puts("Aby zakonczyc, wcisnij [enter] na poczatku wiersza.");
     while (licznik < MAXKS && gets(bibl[licznik].tytul) != NULL
                       && bibl[licznik].tytul[0] != '\0')
     {
         puts("Teraz podaj autora.");
         gets(bibl[licznik].autor);
         puts("Teraz podaj wartosc.");
         scanf("%f", &bibl[licznik++].wartosc);
         while (getchar() != '\n')
             continue;                /* czysci wiersz wejsciowy */
         if (licznik < MAXKS)
             puts("Podaj nastepny tytul.");
     }
```

```
    puts("Oto lista Twoich ksiazek:");
    for (index = 0; index < licznik; index++)
        printf("%s, autor: %s, cena: %.2f zl\n", bibl[index].tytul,
            bibl[index].autor, bibl[index].wartosc);
    fwrite(&bibl[licznikp], rozmiar, licznik - licznikp,
            pksiazki);
    fclose(pksiazki);
    return 0;
}
```

Przyjrzymy się dwóm przykładowym sesjom z programem, a następnie omówimy jego główne elementy.

```
% ksplik
Podaj nowe tytuly ksiazek.
Aby zakonczyc, wcisnij [enter] na poczatku wiersza.
Metryczna mlodosc
Teraz podaj autora.
Polly Poetica
Teraz podaj wartosc.
75.99
Podaj nastepny tytul.
Spisek dworski
Teraz podaj autora.
Zbyszek Zmorski
Teraz podaj wartosc.
63.99
Podaj nastepny tytul.
[enter]
Oto lista Twoich ksiazek:
Metryczna mlodosc, autor: Polly Poetica, cena: 75.99 zl
Spisek dworski, autor: Zbyszek Zmorski, cena: 63.99 zl
% ksplik
Biezaca zawartosc pliku ksiazki.dat:
Metryczna mlodosc autor:  Polly Poetica: $75.99
Spisek dworski autor:  Zbyszek Zmorski: $63.99
Podaj nowe tytuly ksiazek.
Aby zakonczyc, wcisnij [enter] na poczatku wiersza.
Romansidlo
Teraz podaj autora.
Roman Sidlo
Teraz podaj wartosc.
91.99
Podaj nastepny tytul.

Oto lista Twoich ksiazek:
Metryczna mlodosc, autor: Polly Poetica, cena: 75.99 zl
Spisek dworski, autor: Zbyszek Zmorski, cena: 63.99 zl
Romansidlo, autor: Roman Sidlo, cena: 91.99 zl
%
```

Ponowne uruchomienie programu `ksplik` spowodowałoby wyświetlenie listy trzech książek jako aktualnej zawartości pliku.

Omówienie programu

Po pierwsze, plik został otwarty w trybie "a+b". Człon a+ umożliwia odczyt całego pliku i dopisywanie danych na jego końcu. Litera b oznacza, że program będzie korzystał z trybu binarnego. W systemach uniksowych, które nie przyjmują litery b, litera ta może zostać pominięta, ponieważ w systemach tych tryby tekstowy i binarny są identyczne. W innych środowiskach konieczne może być użycie lokalnego odpowiednika symbolu b. Wybraliśmy tryb binarny, ponieważ do pracy w tym trybie przeznaczone są funkcje `fread()` i `fwrite()`.

Polecenie `rewind()` umieszcza wskaźnik położenia na początku pliku tak, aby możliwe było odczytanie pierwszej wartości.

Pierwsza pętla `while` wczytuje kolejne struktury do tablicy `bibl` aż do momentu przepełnienia tablicy lub osiągnięcia końca pliku. Zmienna `licznikp` przechowuje liczbę odczytanych struktur.

Kolejna pętla `while` pobiera dane od użytkownika. Tak jak w listingu 14.2, kończy ona działanie w przypadku braku miejsca w tablicy lub wciśnięcia klawisza Enter na początku wiersza. Zauważ, że zmienna `licznik` rozpoczyna liczenie od wartości, którą miała na końcu poprzedniej pętli. Sprawia to, że nowe struktury są dopisywane w wolnym obszarze tablicy.

Pętla `for` wyświetla zarówno dane pobrane z pliku, jak i wprowadzone przez użytkownika.

Mogliśmy użyć pętli, aby dodawać do pliku po jednej strukturze, zdecydowaliśmy się jednak wykorzystać zdolność funkcji `fwrite()` do zapisywania wielu porcji danych w jednym wywołaniu. Wyrażenie `licznik - licznikp` stanowi liczbę pozycji (struktur) do dopisania. Wyrażenie `&bibl[licznikp]` jest adresem pierwszej nowej struktury w tablicy – od tego miejsca rozpoczyna się kopiowanie. Ponieważ plik został otwarty w trybie dopisywania, nowe dane są dodawane do jego istniejącej zawartości.

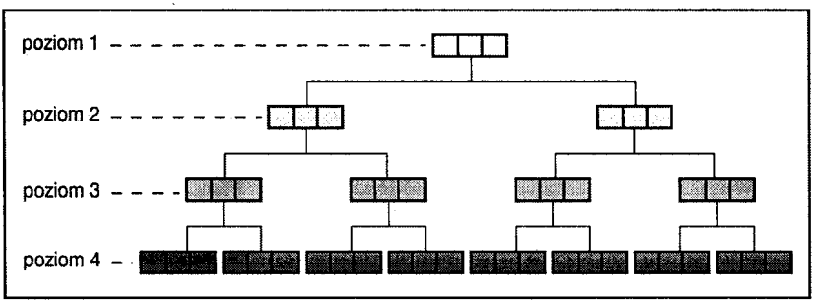
Nasz przykład przedstawia prawdopodobnie najprostszy sposób zapisywania i odczytywania danych z pliku – sposób ten marnuje jednak przestrzeń dyskową, ponieważ powoduje on przechowywanie również nieużywanych części struktury. Rozmiar struktury wynosi `2 x 40 x sizeof(char) + sizeof(float)`, czyli na naszym komputerze 84 bajty. Żadna ze struktur nie potrzebowała w rzeczywistości takiej ilości miejsca. Mimo to, jednakowy rozmiar wszystkich porcji danych bardzo ułatwia odczytywanie informacji.

Inne, bardziej oszczędne podejście polega na korzystaniu z rekordów o różnych rozmiarach. Aby ułatwić odczyt takich rekordów, każdy z nich mógłby rozpoczynać się polem liczbowym przechowującym jego rozmiar. Metoda ta jest oczywiście bardziej skomplikowana niż ta, którą zastosowaliśmy w naszym programie. Najczęściej wymaga ona korzystania ze „struktur łączonych", o których wspomnimy za chwilę, oraz z dynamicznego przydzielania pamięci, które omawiamy w Rozdziale 16 „Preprocesor i biblioteka C".

Struktury: Co dalej?

Zanim zakończymy nasze omówienie struktur, chcielibyśmy wspomnieć o jednym z ich ważniejszych zastosowań: tworzeniu nowych form danych. Istnieją bowiem formy danych, które w pewnych sytuacjach są znacznie efektywniejsze niż przedstawione przez nas tablice i proste struktury. Formy te noszą nazwy, takie jak: kolejki, drzewa binarne, stosy, tablice rozproszone, grafy. Wiele z nich jest budowanych w oparciu o struktury łączone, czyli takie, które oprócz wartości przechowują jeden lub dwa wskaźniki do innych struktur tego samego typu. Wskaźniki te łączą struktury ze sobą i udostępniają ścieżkę, pozwalającą na

przejście przez cały łańcuch struktur. Na przykład, rysunek 14.3 przedstawia drzewo binarne, w którym każda struktura (węzeł) jest połączona z dwoma strukturami znajdującymi się poniżej.



Rysunek 14.3. Budowa drzewa binarnego.

Czy hierarchiczna, drzewiasta struktura przedstawiona na rys. 14.3 jest bardziej efektywna niż tablica? Rozważmy przypadek drzewa o dziesięciu poziomach. Zawiera ono 2^{10}-1 czyli 1023 węzły, w których można przechować maksymalnie 1023 słowa. Jeśli słowa są rozmieszczone w jakimś porządku, to rozpoczynając od najwyższego poziomu każde słowo można znaleźć w co najwyżej dziewięciu ruchach. Gdyby słowa znajdowały się w tablicy, znalezienie żądanego słowa mogłoby w najgorszym wypadku wymagać przejścia przez wszystkie 1023 elementy.

Jeśli jesteś zainteresowany bardziej zaawansowanymi konstrukcjami, takimi jak drzewa binarne, możesz zapoznać się z tekstami informatycznymi poświęconymi strukturom danych. Struktury w języku C pozwalają tworzyć praktycznie każdą formę danych przedstawioną w tych tekstach. Ponadto, niektóre z zaawansowanych form danych oparte na strukturach łączonych omówione są w Rozdziale 17 „Zaawansowana reprezentacja danych".

Na tym kończymy nasze rozważania o strukturach w tym rozdziale. W kolejnych podrozdziałach przyjrzymy się uniom oraz instrukcji `typedef`.

Unie: Szybkie spojrzenie

Unia (ang. *union*) jest typem, który pozwala przechowywać różne rodzaje danych w tym samym obszarze pamięci (jednak nie równocześnie). Dzięki uniom możliwe jest na przykład utworzenie tablicy jednostek o jednakowej długości, z których każda może przechowywać dane innego typu.

Tworzenie unii przebiega podobnie, jak tworzenie struktury. Wyróżniamy szablony i zmienne, które mogą zostać zdefiniowane w jednym lub – za pośrednictwem etykiety – w dwóch krokach. Oto przykład szablonu unii z etykietą:

```
union magazyn {
    int cyfra;
    double duzfl;
    char litera;
};
```

Poniższe instrukcje definiują trzy unie typu `magazyn`:

```
union magazyn fit;      /* unia typu magazyn             */
union magazyn tab[10]; /* tablica 10 unii typu magazyn  */
union magazyn *wu;      /* wskaźnik do unii typu magazyn */
```

Pierwsza deklaracja tworzy pojedynczą zmienną o nazwie `fit`. Kompilator przydziela jej tyle miejsca, aby mogła ona przechować największą ze zmiennych będących częścią szablonu unii. W tym przypadku zmienną o największym rozmiarze jest `duzfl`, należąca do typu double o długości 64 bitów, czyli 8 bajtów (na naszym komputerze). Druga deklaracja tworzy tablicę `tab` złożoną z dziesięciu elementów o rozmiarze 8 bajtów. Trzecia deklaracja tworzy wskaźnik, który może przechowywać adres unii typu `magazyn`.

Unie mogą być inicjalizowane; ponieważ jednak każda unia przechowuje tylko jedną wartość, zasady są nieco inne niż w przypadku struktur. Istnieją dwie możliwości: nadanie unii wartości innej unii tego samego typu lub inicjalizacja pierwszego składnika unii:

```
union magazyn wartA;
wartA.litera = 'R';
union magazyn wartB = wartA;  /* przypisanie wartości innej unii */
union magazyn wartC = {88};   /* inicjalizacja składnika cyfra   */
```

Oto przykłady wykorzystania unii:

```
fit.cyfra = 23;   /* 23 zapisane w fit; zajęte 2 bajty          */
fit.duzfl = 2.0;  /* 23 usunięte, 2.0 zapisane; zajęte 8 bajtów */
fit.litera = 'h'; /* 2.0 usunięte, h zapisane; zajęty 1 bajt    */
```

Przyrostek złożony z kropki i nazwy składnika informuje o typie danej, która ma zostać umieszczona w unii. W danym momencie unia może przechowywać tylko jedną wartość. W naszym przypadku oznacza to, że nie możemy równocześnie przechować zmiennej typu `char` i zmiennej typu `int`, mimo że unia posiada wystarczającą ilość miejsca, aby zmieścić obie te wartości. Należy zauważyć, że pamiętanie o tym, jaki rodzaj informacji znajduje się w unii w każdym momencie, jest obowiązkiem programisty.

Operator -> może być stosowany do wskaźników do unii tak samo, jak w przypadku wskaźników do struktur:

```
wu = &fit;
x = wu->cyfra;  /* równoważne x = fit.cyfra */
```

Poniższy kod pokazuje, czego nie należy robić:

```
fit.litera = 'A';
flnum = 3.02 * fit.duzfl;  /* BŁĄD BŁĄD BŁĄD */
```

Powyższy ciąg instrukcji jest błędny, ponieważ w pierwszym wierszu zapisany został znak, a drugi wiersz zakłada, że zawartość unii `fit` należy do typu `double`.

Mimo to, korzystanie z różnych składników unii przy zapisie i odczycie może czasami być przydatne. Przykład tej techniki przedstawia listing 15.4 w kolejnym rozdziale.

Dobrym miejscem do zastosowania unii może być struktura, w której rodzaj przechowywanych danych zależy od jednego ze składników. Na przykład, załóżmy, że mamy strukturę reprezentującą samochód. Jeśli właścicielem samochodu jest jego użytkownik, potrzebny jest składnik struktury opisujący osobę właściciela. Jeśli zaś samochód jest wypożyczony, składnik powinien opisywać firmę, która jest właścicielem pojazdu. Aby to uzyskać, możemy użyć następującego kodu:

```
struct wlasc {
     char nrdowodu[12];
     ...
};
struct firma {
     char nazwa[40];
```

```
    char siedziba[40];
    ...
};
union dane {
    struct wlasciciel wlasc_sam;
    struct firma firma_sam;
};
struct dane_sam {
    char marka[15];
    int stan; /* 0 = własny, 1 = pożyczony */
    union dane dane_wlasc;
    ...
};
```

Załóżmy, że zmienna oparta na szablonie `dane_sam` nosi nazwę `garbus`. Wówczas jeśli `garbus.stan` wynosi 0, program mógłby korzystać ze struktury `garbus.dane_wlasc.wlasc_sam`, a jeśli `garbus.stan` wynosi 1 – ze struktury `garbus.dane_wlasc.firma_sam`.

Podsumowanie: Operatory w strukturach i uniach

Operator przynależności:

.

Uwagi ogólne:

Operator . w połączeniu z nazwą struktury lub unii określa jej składnik. Jeśli `nazwa` jest nazwą struktury, a `skladnik` – składnikiem zadeklarowanym w szablonie struktury, wówczas składnik ten oznaczamy następująco:

```
nazwa.skladnik
```

Wyrażenie `nazwa.skladnik` należy do tego samego typu, co zmienna `skladnik`. W podobny sposób stosujemy kropkę w odniesieniu do unii.

Przykład:

```
struct {
       int kod;
       float cena;
} artykul;

artykul.kod = 1265;
```

Ostatnia instrukcja nadaje wartość składnikowi `kod` struktury `artykul`.

Operator przynależności pośredniej:

->

Uwagi ogólne:

Operator ten w połączeniu ze wskaźnikiem do struktury lub unii pozwala uzyskać dostęp do jednego ze składników tej struktury lub unii. Załóżmy, że `wskstr` jest wskaźnikiem do struktury, oraz że `skladnik` jest składnikiem zadeklarowanym w szablonie tej struktury. Wówczas wyrażenie

```
wskstr->skladnik
```

określa składnik wskazywanej struktury. W podobny sposób operator -> stosujemy do unii.

Przykład:

```
struct {
       int kod;
       float cena;
} artykul, * wskst;
wskst = &artykul;
wskst->kod = 3451;
```

Ostatnia instrukcja nadaje wartość składnikowi `kod` struktury `artykul`. Poniższe trzy wyrażenia są równoważne:

```
wskst->kod    artykul.kod    (*wskst).kod
```

typedef: Szybkie spojrzenie

Słowo kluczowe `typedef` jest zaawansowanym elementem języka C, który pozwala tworzyć nowe nazwy typów. Przypomina ono dyrektywę `#define` z trzema istotnymi różnicami:

W odróżnieniu od `#define`, `typedef` nadaje nazwy typom, a nie wartościom.

Słowo kluczowe `typedef` jest interpretowane przez kompilator, a nie preprocesor.

W swoim obszarze zastosowań mechanizm `typedef` jest bardziej elastyczny niż `#define`.

Zobaczmy, w jaki sposób działa słowo kluczowe `typedef`. Załóżmy, że chcemy używać terminu `BYTE` w odniesieniu do liczb o rozmiarze jednego bajta. W tym celu wystarczy zdefiniować `BYTE` jak zwykłą zmienną typu `char` i poprzedzić definicję słowem kluczowym `typedef`.

```
typedef unsigned char BYTE;
```

Od tego momentu możemy korzystać z nowego typu `BYTE` przy deklarowaniu zmiennych:

```
BYTE x, y[10], * z;
```

Zasięg definicji typu zależy od położenia instrukcji `typedef`. Jeśli definicja znajduje się wewnątrz funkcji jej zasięg jest lokalny; jeśli znajduje się ona poza funkcją, jej zasięg jest globalny.

Nazwy nowych typów pisane są często wielką literą dla podkreślenia, że są one jedynie symbolicznymi skrótami – równie dobrze można jednak użyć małych liter:

```
typedef unsigned char byte;
```

Zasady nazewnictwa typów są takie same, jak zasady nazewnictwa zmiennych.

Tworzenie drugiej nazwy dla istniejącego typu może wydawać się niepoważne, ale w rzeczywistości bywa przydatne. W poprzednim przykładzie używanie nazwy `BYTE` zamiast `unsigned char` pozwala zaakcentować, że zmienne typu `BYTE` będą wykorzystywane do przechowywania liczb, a nie kodów znaków.

Instrukcja `typedef` pozwala również zwiększyć przenośność programów. Wspomnieliśmy swego czasu o nazwie `size_t`, która reprezentuje typ zwracany przez operator `sizeof`, oraz o nazwie `time_t`, która oznacza typ zwracany przez funkcję `time()`. Standard języka C stwierdza jedynie, że `sizeof` i `time()` zwracają wartość całkowitą – określenie jej typu należy do implementacji. Ten brak zdecydowania wypływa z przekonania komitetu ANSI C, że żaden narzucony z góry typ nie będzie najlepszy dla wszystkich platform. Wprowadzenie nowej nazwy typu, takiej jak `time_t` (której każda implementacja może

przypisać jakiś określony typ za pomocą mechanizmu typedef) umożliwiło ustalenie ogólnego prototypu funkcji, takiego jak poniższy:

```
time_t time(time_t *);
```

Na jednym systemie time_t może oznaczać typ unsigned int, na innym – unsigned long. Jeśli tylko do programu dołączony jest plik nagłówkowy time.h, kompilator może dotrzeć do odpowiedniej definicji, a Twój kod może wykorzystywać typ time_t.

W niektórych sytuacjach instrukcję typedef można zastąpić dyrektywą #define. Na przykład,

```
#define BYTE unsigned char
```

powoduje zastąpienie przez preprocesor słowa BYTE słowami unsigned char. Oto przykład definicji typu, której nie można zastąpić dyrektywą #define:

```
typedef char * STRING;
```

Warto zauważyć, że gdyby nie słowo kluczowe typedef, powyższy wiersz byłby definicją zmiennej STRING typu „wskaźnik do char". Słowo typedef sprawia, że STRING zostaje zdefiniowana jako inna nazwa typu „wskaźnik do char". Stąd,

```
STRING imie, znak;
```

to tyle, co

```
char * imie, * znak;
```

Załóżmy, że użyjemy następującej dyrektywy:

```
#define STRING char *
```

Wówczas

```
STRING imie, znak;
```

zostałoby przełożone na

```
char * imie, znak;
```

W tym przypadku tylko zmienna imie byłaby wskaźnikiem.

Instrukcję typedef można również stosować do struktur:

```
typedef struct zespol {
        float rzecz;
        float uroj;
} ZESPOL;
```

W wyniku powyższego kodu moglibyśmy definiować liczby zespolone za pomocą typu ZESPOL zamiast struct zespol. Jednym z powodów korzystania z mechanizmu typedef jest to, iż pozwala on tworzyć wygodne, łatwo rozpoznawalne nazwy dla często używanych typów. Na przykład, wielu ludzi woli korzystać z nazwy STRING zamiast char *.

Innym zastosowaniem instrukcji typedef jest tworzenie skróconych nazw dla skomplikowanych typów. Na przykład, wiersz

```
typedef char (* FWC ()) [5];
```

definiuje FWC jako typ będący funkcją, która zwraca wskaźnik do pięcioelementowej tablicy typu char. (Patrz poniższe omówienie udziwnionych deklaracji.)

Korzystając z instrukcji typedef miej świadomość, że nie tworzy ona nowych typów, a jedynie wygodne etykiety. Oznacza to na przykład, że zmienne utworzonego przez nas typu STRING mogą być przekazywane jako argumenty do funkcji, które oczekują wskaźnika do char.

Udziwnione deklaracje

Język C pozwala tworzyć bardzo zawiłe formy danych. Chociaż w niniejszej książce korzystamy raczej z tych prostszych form, uważamy, że naszym obowiązkiem jest zwrócenie uwagi na dostępne możliwości.

Jak zapewne wiesz, znaczenie deklaracji może zostać zmienione przez dodanie do nazwy zmiennej tzw. modyfikatora.

Modyfikator	Znaczenie
*	wskaźnik
()	funkcja
[]	tablica

Możliwe jest użycie wielu modyfikatorów w jednej deklaracji, co pozwala tworzyć duży zakres różnych typów:

```
int plansza[8][8];   /* tablica tablic typu int                    */
int ** wsk;          /* wskaźnik do wskaźnika do int               */
int * domy[10];      /* 10-elementowa tablica wskaźników do int    */
int (* tomy)[10];    /* wskaźnik do 10-elementowej tablicy typu int */
int * uff[3][4];     /* tablica 3 x 4 wskaźników do int            */
int (* puff)[3][4];  /* wskaźnik do tablicy 3 x 4 wartości int     */
int (* huff[3])[4];  /* 3-elementowa tablica wskaźników do
                        4-elementowych tablic typu int             */
```

Rozszyfrowanie tych deklaracji nie wymaga nadprzyrodzonych zdolności, a jedynie wiedzy o kolejności stosowania modyfikatorów. Pomocne powinny być następujące zasady:

1. Modyfikatory [] (oznaczający tablicę) oraz () (oznaczający funkcję) mają ten sam priorytet. Priorytet ten jest wyższy niż priorytet operatora *, co oznacza, że poniższa instrukcja deklaruje domy jako tablicę wskaźników, a nie wskaźnik do tablicy:

   ```
   int * domy[10];
   ```

2. Modyfikatory [] i () działają w kierunku od lewej do prawej. Stąd poniższy wiersz deklaruje towary jako tablicę 12 tablic 50 wartości typu int, a nie tablicę 50 tablic 12 wartości typu int:

   ```
   int towary[12][50];
   ```

3. Modyfikator [] ma pierwszeństwo przed *, ale nawiasy (które mają zawsze najwyższy priorytet) łączą ze sobą * i tomy, zanim zastosowane zostaną nawiasy kwadratowe. Oznacza to, że tomy jest wskaźnikiem do tablicy 10 wartości typu int:

   ```
   int (* tomy)[10];
   ```

Spróbujmy zastosować te zasady do następującej deklaracji:

```
int * uff[3][4];
```

Modyfikator [3] ma pierwszeństwo przed *, a także – z powodu kierunku wiązania – przed [4]. uff jest więc tablicą trzech elementów. Następny w kolejności jest modyfikator [4], zatem elementy tablicy są tablicami złożonymi z czterech elementów. Modyfikator * stwierdza, że elementy te są wskaźnikami. Ostatnią częścią układanki jest słowo kluczowe int: uff jest trzyelementową tablicą czteroelementowych tablic wskaźników do int lub krócej tablicą 3 x 4 wskaźników do int. Zajmuje ona tyle samo pamięci, co 12 wskaźników.

Przeanalizujmy teraz tę deklarację:

```
int (* puff)[3][4];
```

Nawiasy sprawiają, że modyfikator * działa jako pierwszy. Tym samym zmienna puff zostaje zadeklarowana jako wskaźnik do tablicy 3 x 4 wartości typu int. Przydzielony obszar pamięci ma długość jednego wskaźnika.

Możliwe są również następujące typy:

```
char * flip();       /* funkcja zwracająca wskaźnik do char */
char (* flap)();     /* wskaźnik do funkcji, która zwraca
                        wartość typu char                   */
char (* flop[3])(); /* tablica 3 wskaźników do funkcji
                        zwracających wartość typu char      */
```

Jeśli dodać do tego jeszcze struktury, możliwe staje się tworzenie naprawdę wyszukanych deklaracji. A zastosowania – no cóż, omówienie zastosowań pozostawimy bardziej zaawansowanym tekstom.

Funkcje a wskaźniki

Jak pokazało to nasze omówienie deklaracji, możliwe jest tworzenie wskaźników do funkcji. Być może zastanawiasz się, czy takie konstrukcje mogą być przydatne. Wyjaśnimy to na przykładzie. Jednym z argumentów funkcji qsort() z biblioteki ANSI C jest wskaźnik do funkcji. Jaka jest jego rola? w niniejszej książce korzystaliśmy do tej pory z dwóch funkcji sortujących: jednej dla liczb całkowitych i drugiej dla łańcuchów. Algorytm był w obu przypadkach identyczny – różnica tkwiła w porównywaniu elementów: liczby całkowite porównywaliśmy za pomocą operatora >, a łańcuchy za pomocą funkcji strcmp(). Funkcja qsort() przyjmuje bardziej uniwersalne podejście. Pobiera ona wskaźnik do określonej przez programistę funkcji porównującej, a następnie korzysta z niej przeprowadzając procedurę sortowania. Dzięki temu funkcję qsort() można zastosować do liczb całkowitych, łańcuchów, struktur, jak i wielu innych typów danych.

Czym jest wskaźnik do funkcji? Wskaźnik np. do int przechowuje adres w pamięci, pod którym rozpoczyna się wartość typu int. Wskaźnik do funkcji przechowuje adres, pod którym rozpoczyna się kod maszynowy funkcji.

Deklarując wskaźnik do danych, należy określić typ wskazywanej wartości. Podobnie deklarując wskaźnik do funkcji, należy określić typ wskazywanej funkcji. Określenie typu funkcji odbywa się przez podanie typów argumentów oraz typu wartości zwracanej. Na przykład, zastanów się nad tym prototypem:

```
void DuzeLit(char *);  /* przetwarza małe litery na duże */
```

Typ funkcji DuzeLit() jest określony przez typ argumentu char * oraz typ wartości zwracanej void. Aby zadeklarować wskaźnik wf do funkcji tego typu, należy użyć następującej instrukcji:

```
void (*wf)(char *);     /* wf jest wskaźnikiem do funkcji */
```

Pierwsza para nawiasów związuje operator * z nazwą wf, co oznacza, że wf jest wskaźnikiem do funkcji. Tym samym wyrażenie (*wf) jest funkcją o argumencie typu (char *) i wartości zwracanej typu void. Prawdopodobnie najprostszym sposobem na zapamiętanie tej deklaracji jest zauważenie, że wyrażenie (*wf) zastępuje w niej nazwę funkcji DuzeLit.

Jak wspomnieliśmy wcześniej, pierwszy nawias jest wymagany z uwagi na kolejność działania operatorów. Pominięcie go prowadzi do czegoś zupełnie innego:

```
void *wf(char *);      /* wf jest funkcją, która zwraca wskaźnik */
```

Wskazówka

Aby zadeklarować wskaźnik do funkcji określonego typu, najpierw zadeklaruj funkcję, a następnie zastąp jej nazwę wyrażeniem postaci (*wf); wf stanie się wskaźnikiem do funkcji.

Po utworzeniu wskaźnika do funkcji, może on otrzymywać adresy funkcji odpowiedniego typu. Adres funkcji symbolizuje w tym kontekście jej nazwa:

```
void DuzeLit(char *);
void MaleLit(char *);
int zaokr(double);
void (*wf)(char *);
wf = DuzeLit;         /* prawidłowe, DuzeLit jest adresem funkcji              */
wf = MaleLit;         /* prawidłowe, MaleLit jest adresem funkcji              */
wf = zaokr;           /* nieprawidłowe, zaokr jest funkcją niewłaściwego typu */
wf = MaleLit();       /* nieprawidłowe, MaleLit() nie jest adresem             */
```

Ostatnia instrukcja jest nieprawidłowa również dlatego, że funkcja typu void nie może być częścią instrukcji przypisania. Zauważ, że wskaźnik wf może wskazywać na każdą funkcję, która przyjmuje argument typu char * i zwraca wartość typu void – nie może jednak wskazywać na żadną funkcję, która nie jest zgodna z tą charakterystyką.

Tak jak wskaźnik do danych pozwala uzyskać dostęp do wskazywanych wartości, tak i wskaźnik do funkcji umożliwia skorzystanie ze wskazywanej przezeń funkcji. Co ciekawe, istnieją dwie równoprawne, ale sprzeczne logicznie składnie, pozwalające to uczynić:

```
void DuzeLit(char *);
void MaleLit(char *);
void (*wf)(char *);
char mis[] = "Nina Metier";
wf = DuzeLit;
(*wf)(mis);       /* stosuje funkcję DuzeLit do zmiennej mis (składnia 1) */
wf = MaleLit;
wf(mis);          /* stosuje funkcję MaleLit do zmiennej mis (składnia 2) */
```

Obie metody zapisu wydają się rozsądne. Oto uzasadnienie pierwszego zapisu: Ponieważ wf wskazuje na funkcję DuzeLit, wyrażenie *wf oznacza funkcję DuzeLit, a więc (*wf)(mis) to tyle, co DuzeLit(mis). Jeśli nie wierzysz, że DuzeLit i (*wf) są równoważne, spójrz na deklaracje funkcji DuzeLit i wskaźnika wf. Oto uzasadnienie drugiego zapisu: Ponieważ nazwa funkcji jest wskaźnikiem, nazwy i wskaźnika można używać zamiennie, zatem wf(mis) jest równoważne MaleLit(mis). Jeśli nie

wierzysz, że `wf` i `MaleLit` są równoważne, spójrz na instrukcję przypisania `wf = MaleLit;`. W czasach początków języka C Laboratoria Bella (które stworzyły C i UNIX) opowiadały się za pierwszą argumentacją, a Uniwersytet w Berkeley (który kontynuował prace nad Uniksem) – za drugą. Definicja K&R C nie dopuszczała drugiej składni, jednak standard ANSI C akceptuje obie formy zapisu.

Podobnie jak wskaźniki do danych, wskaźniki do funkcji są najczęściej wykorzystywane w roli argumentów. Na przykład, zastanów się nad następującym prototypem:

```
void pokaz(void (* fw)(char *), char * lan);
```

Powyższy prototyp – choć wygląda skomplikowanie – deklaruje dwa parametry, `fw` i `lan`. Pierwszy z nich jest wskaźnikiem do funkcji, a drugi – wskaźnikiem do danych. Mówiąc bardziej konkretnie, `fw` wskazuje na funkcję, która przyjmuje argument typu `char *` i nie zwraca wartości, a `lan` wskazuje na znak. Stąd, uwzględniając dokonane przez nas wcześniej deklaracje, możemy użyć wywołań, takich jak poniższe:

```
pokaz(MaleLit, mis); /* pokaz() wykorzystuje funkcję MaleLit(): fw = MaleLit      */
pokaz(wf, mis);      /* pokaz() wykorzystuje funkcję wskazywaną przez wf: fw = wf */
```

W jaki sposób funkcja `pokaz()` może wykorzystać przekazany jej wskaźnik do funkcji? Może ona wywołać wskazywaną funkcję za pomocą jednego z dwóch zapisów – `fw()` lub `(*fw)()`.

```
void pokaz(void (* fw)(char *), char * lan)
{
    (*fw)(lan); /* stosuje wybraną funkcję do lan */
    puts(lan);  /* wyświetla wynik                */
}
```

Zgodnie z powyższym kodem funkcja `pokaz()` najpierw przetwarza łańcuch `lan` przez użycie na nim funkcji wskazywanej przez `fw`, a następnie wyświetla go na ekranie.

Nawiasem mówiąc, funkcje zwracające wartość mogą pełnić dwie różne role jako argumenty innych funkcji. Zastanów się nad następującymi instrukcjami:

```
funkcja1(sqrt);     /* przekazuje adres funkcji sqrt                  */
funkcja2(sqrt(4.0); /* przekazuje wartość zwróconą przez funkcję sqrt */
```

Pierwsza instrukcja przekazuje adres funkcji `sqrt()` – przypuszczalnie `funkcja1()` wykorzysta tę funkcję w swoim kodzie. Druga instrukcja wywołuje funkcję `sqrt()`, a następnie przekazuje zwróconą przez nią wartość (w tym wypadku 2.0) do `funkcji2()`.

Program na listingu 14.12 wywołuje funkcję `pokaz()`, przekazując jej kilka różnych funkcji przetwarzających łańcuchy. Ilustruje on również parę technik przydatnych przy obsłudze menu.

Listing 14.12. Program `fun_wsk.c`.

```
/* fun_wsk.c -- wykorzystuje wskazniki do funkcji */
#include <stdio.h>
#include <string.h>
#include <ctype.h>
char pokazmenu(void);
void usunwiersz(void);  /* usuwa dane do konca wiersza             */
void pokaz(void (* fp)(char *), char * str);
void DuzeLit(char *);   /* przetwarza male litery na duze          */
void MaleLit(char *);   /* przetwarza duze litery na male          */
void Odwroc(char *);    /* zamienia duze litery na male i odwrotnie */
void Atrapa(char *);    /* pozostawia lancuch bez zmian            */
```

```
int main(void)
{
    char wiersz[81];
    char kopia[81];
    char wybor;
    void (*wfun)(char *); /* wskazuje na funkcje przyjmujaca */
                          /* argument typu char *            */
                          /* i nie zwracajaca wartosci       */
    puts("Podaj lancuch (pusty wiersz konczy program):");
    while (gets(wiersz) != NULL && wiersz[0] != '\0')
    {
        while ((wybor = pokazmenu()) != 'n')
        {
            switch (wybor   )      /* switch decyduje o wskazniku */
            {
                case 'd' : wfun = DuzeLit; break;
                case 'm' : wfun = MaleLit; break;
                case 'o' : wfun = Odwroc;  break;
                case 'b' : wfun = Atrapa;  break;
            }
            strcpy(kopia, wiersz); /* tworzy kopie dla funkcji pokaz() */
            pokaz(wfun, kopia);    /* korzysta z wybranej funkcji      */
        }
        puts("Podaj lancuch (pusty wiersz konczy program):");
    }
    puts("Czesc!");
    return 0;
}

char pokazmenu(void)
{
    char odp;
    puts("Wybierz jedna opcje:");
    puts("d) duze litery        m) male litery");
    puts("o) odwrocenie liter   b) bez zmian");
    puts("n) nastepny lancuch");
    odp = getchar();    /* pobiera odpowiedz              */
    odp = tolower(odp); /* przetwarza odp. na mala litere */
    usunwiersz();       /* pozbywa sie reszty wiersza     */
    while (strchr("dmobn", odp) == NULL)
    {
        puts("Wpisz d, m, o, b lub n:");
        odp = tolower(getchar());
        usunwiersz();
    }
    return odp;
}

void usunwiersz(void)
{
    while (getchar() != '\n')
        continue;
}

void DuzeLit(char * lan)
{
    while (*lan != '\0')
```

```
    {
        *lan = toupper(*lan);
        lan++;
    }
}

void MaleLit(char * lan)
{
    while (*lan != '\0')
    {
        *lan = tolower(*lan);
        lan++;
    }
}
void Odwroc(char * lan)
{
    while (*lan != '\0')
    {
        if (islower(*lan))
            *lan = toupper(*lan);
        else if (isupper(*lan))
            *lan = tolower(*lan);
        lan++;
    }
}
void Atrapa(char * lan)
{
    /* pozostawia lancuch bez zmian */
}

void pokaz(void (* fw)(char *), char * lan)
{
    (*fw)(lan); /* stosuje wybrana funkcje do lan */
    puts(lan);  /* wyswietla wynik                */
}
```

Oto przykładowy przebieg działania programu:

```
Podaj lancuch (pusty wiersz konczy program):
Czy programujac w C czujesz sie zapetlony?
Wybierz jedna opcje:
d) duze litery        m) male litery
o) odwrocenie liter   b) bez zmian
n) nastepny lancuch
o
cZY PROGRAMUJAC w c CZUJESZ SIE ZAPETLONY?
Wybierz jedna opcje:
d) duze litery        m) male litery
o) odwrocenie liter   b) bez zmian
n) nastepny lancuch
m
czy programujac w c czujesz sie zapetlony?
Wybierz jedna opcje:
d) duze litery        m) male litery
o) odwrocenie liter   b) bez zmian
n) nastepny lancuch
n
```

```
Podaj lancuch (pusty wiersz konczy program):
[enter]
Czesc!
```

Zauważ, że funkcje `DuzeLit()`, `MaleLit()`, `Odwroc()` i `Atrapa()` wszystkie należą do tego samego typu, a więc adres każdej z nich może być przechowywany we wskaźniku `wfun`. Nasz program przekazuje funkcji `pokaz()` wskaźnik `wfun`, ale – rzecz jasna – możliwe byłoby również bezpośrednie przekazanie nazwy każdej z czterech funkcji, np. `pokaz(Odwroc, wiersz)`.

W sytuacjach, takich jak ta, przydatna bywa instrukcja `typedef`. Na przykład, program mógłby zawierać następujący kod:

```
typedef void (*V_WF_WCHAR)(char *);
void pokaz(V_WF_WCHAR wp, char *);
V_WF_WCHAR wfun;
```

Jeśli masz ochotę na drobne urozmaicenie, możesz zadeklarować i zainicjować tablicę następujących wskaźników:

```
V_WF_WCHAR tabwf[4] = {DuzeLit, MaleLit, Odwroc, Atrapa};
```

Jeśli następnie zmodyfikujesz funkcję `pokazmenu()` tak, aby należała ona do typu `int` i zwracała `0`, gdy użytkownik wpisze `d`, `1`, gdy użytkownik wpisze `m`, i tak dalej, będziesz mógł zastąpić pętlę z instrukcją `switch` następującym, znacznie prostszym kodem:

```
index = pokazmenu();
while (index >= 0 && index <= 3)
{
    strcpy(kopia, wiersz);      /* tworzy kopie dla funkcji pokaz() */
    pokaz(tabwf[index], kopia); /* korzysta z wybranej funkcji      */
    index = pokazmenu();
}
```

Zauważ, że język C dopuszcza tworzenie tablic wskaźników do funkcji, ale nie tablic samych funkcji.

Rysunek 14.4 podsumowuje wszystkie pięć zastosowań nazw funkcji.

nazwa funkcji w prototypie:	`int licz(int x, int y);`
nazwa funkcji w wywołaniu:	`stan = licz(q,r);`
nazwa funkcji w definicji:	`int licz(int x, int y)`
	`{ ...`
nazwa funkcji jako wskaźnik:	`wfun = licz;`
nazwa funkcji jako argument wskaźnikowy:	`slowsort(tab,n,licz);`

Rysunek 14.4. Zastosowania nazwy funkcji.

Funkcja `pokazmenu()` ilustruje kilka technik obsługi menu. Po pierwsze, kod

```
odp = getchar();     /* pobiera odpowiedz              */
odp = tolower(odp); /* przetwarza odp. na mala litere */
```

oraz

```
odp = tolower(getchar());
```

pokazuje dwa sposoby zamiany znaku wejściowego na małą literę tak, aby nie było konieczne uwzględnienie zarówno odpowiedzi 'd', jak i 'D', 'm' oraz 'M', i tak dalej.

Funkcja usunwiersz() pozbywa się reszty wiersza wejściowego. Jest to cenne z dwóch powodów. Po pierwsze, wybierając opcję menu, użytkownik wpisuje literę, a następnie wciska klawisz Enter, co generuje znak nowej linii. Znak ten, jeśli nie zostanie usunięty, zostanie odczytany jako kolejna odpowiedź użytkownika. Po drugie, załóżmy, że użytkownik wpisze duze litery zamiast po prostu d. Gdyby nie funkcja usunwiersz(), program potraktowałby każdy kolejny znak w wyrażeniu duze litery jako oddzielną odpowiedź. Tymczasem, dzięki tej funkcji program przetwarza tylko literę d, porzucając pozostałą część wiersza.

Funkcja usunwiersz() została tak zaprojektowana, aby zwracała tylko prawidłowe odpowiedzi. W tym celu wykorzystuje ona standardową funkcję strchr() z rodziny string.h:

```
while (strchr("dmobn", odp) == NULL)
```

Funkcja ta poszukuje znaku odp w łańcuchu "dmobn" i zwraca wskaźnik do jego położenia. Jeśli znak nie został znaleziony, wartością zwracaną jest wskaźnik zerowy. Powyższe wyrażenie testowe jest więc krótszym i wygodniejszym odpowiednikiem wyrażenia:

```
while (odp != 'd' && odp != 'm' && odp != 'o' && odp != 'b' && odp != 'n')
```

Jak łatwo się domyślić, im więcej możliwych odpowiedzi, tym atrakcyjniejszym rozwiązaniem staje się skorzystanie z funkcji strchr().

Podsumowanie rozdziału

Struktura w języku C udostępnia sposób przechowania wielu wartości – zwykle należących do różnych typów – w jednym obiekcie danych. Szablony struktur oznaczamy za pomocą etykiet, z których korzystamy następnie przy deklarowaniu zmiennych strukturalnych. Operator przynależności . (kropka) pozwala uzyskiwać dostęp do poszczególnych składników (pól) struktury przez podanie ich nazw określonych w szablonie.

W przypadku wskaźnika do struktury, nazwę struktury i kropkę zastępuje nazwa wskaźnika i operator ->. Aby uzyskać adres struktury, korzystamy z operatora &. W odróżnieniu od nazwy tablicy, nazwa struktury nie jest bowiem równoznaczna z jej adresem.

W starszych programach funkcje uzyskiwały dostęp do struktur wyłącznie za pośrednictwem wskaźników. Nowoczesne implementacje (także te zgodne z ANSI C) pozwalają przekazywać struktury do funkcji, zwracać struktury za pomocą mechanizmu return, a także używać ich w instrukcjach przypisania.

Unie wykorzystują tę samą składnię, co struktury, jednak w przypadku unii składniki dzielą między sobą ten sam obszar pamięci. Zamiast przechowywać kilka typów danych jednocześnie, tak jak to czyni struktura, unia przechowuje w danym momencie tylko jeden typ danych z listy określonej przez programistę. Struktura może zatem przechować wartość int, wartość double i wartość char, podczas gdy unia może przechować wartość int, wartość double *lub* wartość char.

Instrukcja typedef pozwala tworzyć nazwy zastępcze dla dowolnych typów możliwych do uzyskania w języku C.

Nazwa funkcji (bez nawiasów) jest równoznaczna z jej adresem. Może ona zostać przekazana do funkcji, która dzięki temu może wykorzystać wskazywaną funkcję w toku swojego działania.

Pytania sprawdzające

1. Czy poniższy szablon jest poprawny?

```
structure {
        char ytatywny;
        int  num[20];
        char * psy
}
```

2. Oto fragment programu. Jaki będzie efekt jego wykonania?

```
#include <stdio.h>
struct dom {
    float mkw;
    int pokoje;
    int pietra;
    char adres[40];
};
int main(void)
{
  struct dom nasz = {140.0, 6, 1, "ul. Kernighana 22"};
  struct dom *znak;

  znak = &nasz;
  printf("%d %d\n", nasz.pokoje, znak->pietra);
  printf("%s\n", nasz.adres);
  printf("%c %c\n", znak->adres[4], nasz.adres[5]);
  return 0;
}
```

3. Zaprojektuj szablon struktury, przechowujący nazwę miesiąca, trzyliterowy skrót nazwy, liczbę dni w miesiącu oraz numer miesiąca.
4. Zdefiniuj tablicę 12 struktur (takich jak w pytaniu nr 3) i zainicjalizuj ją, przyjmując, że rok nie jest przestępny.
5. Napisz funkcję, która po przekazaniu jej numeru miesiąca zwraca całkowitą liczbę dni w roku do tego miesiąca włącznie. Przyjmij, że szablon struktury z pytania nr 3 i tablica struktur z pytania nr 4 są zadeklarowane w innym pliku.
6. Przyjmując poniższą instrukcję `typedef`, zadeklaruj 10-elementową tablicę złożoną ze wskazanych struktur. Następnie, przy pomocy odpowiednich instrukcji przypisania (lub wywołań funkcji łańcuchowych) spraw, aby trzeci element opisywał soczewkę Remarkatar o ogniskowej 500 mm i rozwartości optycznej 2.0.

```
typedef struct soczewka {   /* opis soczewki       */
    float ognisk;           /* ogniskowa w mm      */
    float rozwart;          /* rozwartosc optyczna */
    char marka[30];         /* nazwa soczewki      */
} SOCZEWKA;
```

7. Zastanów się nad następującym fragmentem programu:

```
struct daneos {
```

```
        char imie[20];
        char nazw[20];
};
struct bem {
        int konczyny;
        struct daneos tytul;
        char typ[30];
};
struct bem *wb;
struct bem deb = {
        6,
        {"Berbnazel", "Gwolkapwolk"},
        "Arkturianin"
};

wb = &deb;
```

a. Co wyświetliłaby każda z poniższych instrukcji?

```
printf("%d\n", deb.konczyny);
printf("%s\n", wb->typ);
printf("%s\n", wp->typ + 2);
```

b. W jaki sposób wartość "Gwolkapwolk" można wyrazić w notacji strukturalnej (istnieją dwa sposoby)?

c. Napisz funkcję, która pobiera jako argument adres struktury typu bem i wyświetla jej zawartość w przedstawionym poniżej układzie. Przyjmij, że szablon struktury znajduje się w pliku o nazwie pozaziem.h.

Berbnazel Gwolkapwolk jest Arkturianinem o 6 konczynach.

8. Zastanów się nad poniższymi deklaracjami:

```
struct dane_s {
              char imie[20];
              char nazwisko[20];
              };
struct bard   {
              struct dane_s dane;
              int data_ur;
              int data_sm;
              };
struct bard willie;
struct bard *wsk = &willie;
```

a. Wskaż składnik data_ur struktury willie za pomocą nazwy willie.

b. Wskaż składnik data_ur struktury willie za pomocą nazwy wsk.

c. Za pomocą funkcji scanf() wczytaj wartość do składnika data_ur za pomocą nazwy willie.

d. Za pomocą funkcji scanf() wczytaj wartość do składnika data_ur za pomocą nazwy wsk.

e. Skonstruuj wyrażenie określające trzecią literę imienia osoby opisanej przez strukturę willie.

f. Skonstruuj wyrażenie oznaczające całkowitą liczbę liter w imieniu i nazwisku osoby opisanej przez strukturę `willie`.

9. Zdefiniuj szablon struktury przeznaczony do przechowania następujących pozycji: nazwa samochodu, moc silnika (w koniach mechanicznych), zużycie paliwa na 100 km w warunkach miejskich, rozstaw osi oraz rocznik. Bądź oryginalny i jako etykiety szablonu użyj słowa `samochod`.

10. Załóżmy, że mamy następującą strukturę:

```
struct paliwo {
    float odleglosc;
    float litry;
    float kml;      /* kilometry na litr */
};
```

Opracuj funkcję, która pobiera argument typu `struct paliwo`, oblicza wartość składnika `kml` (liczbę kilometrów, jakie można przejechać na jednym litrze paliwa) i zwraca kompletną strukturę. Przyjmij, że przekazana struktura zawiera wartości składników `odleglosc` i `litry`.

11. Zadeklaruj wskaźnik do funkcji, która zwraca wskaźnik do `char`, a pobiera dwa argumenty: wskaźnik do `char` oraz wartość `char`.

12. Zadeklaruj cztery funkcje i zainicjalizuj tablicę wskaźników tak, aby wskazywały one na funkcje. Każda z funkcji powinna zwracać wartość typu `double` i przyjmować dwa argumenty tego typu.

Ćwiczenia

1. Ponownie wykonaj funkcję z pytania nr 5, ale tym razem niech jej argument będzie nazwą, a nie numerem miesiąca. (Pamiętaj o funkcji `strcmp()`.)

2. Napisz program, który prosi użytkownika o podanie dnia, miesiąca oraz roku. Miesiąc może zostać wskazany za pomocą numeru (np. „9"), nazwy („wrzesien") lub nazwy skróconej („wrz"). Program powinien wyświetlić całkowitą liczbę dni w roku do podanego dnia włącznie.

3. Zmodyfikuj program z listingu 14.2 tak, aby wyświetlał on opisy książek w porządku alfabetycznym (wg tytułu) oraz całkowitą wartość książek.

4. Napisz program, który tworzy szablon struktury o dwóch składnikach zgodnie z poniższymi kryteriami:

   a. Pierwszym składnikiem jest numer PESEL, a drugim – struktura o trzech składnikach. Jej pierwszy składnik zawiera imię, drugi – drugie imię, a trzeci – nazwisko. Utwórz i zainicjuj tablicę złożoną z trzech takich struktur. Program powinien wyświetlić dane w następującej formie:

```
Jordan, Michael J. -- 65092301159
```

   b. Wyświetlana jest tylko pierwsza litera drugiego imienia wraz z kropką. W przypadku, jeśli składnik zawierający drugie imię jest pusty, program nie powinien wyświetlić ani inicjału (co oczywiste), ani kropki. Algorytm wyświetlania zaimplementuj w postaci oddzielnej funkcji, pobierającej tablicę struktur jako argument.

   c. Zmodyfikuj program opisany w punkcie a. tak, aby funkcja wyświetlająca przyjmowała strukturę, a nie adres tablicy struktur.

5. Napisz program, realizujący następujący przepis:

a. Zdefiniuj zewnętrznie szablon struktury o nazwie `daneos`, zawierający dwa składniki: łańcuch przechowujący imię i łańcuch przechowujący nazwisko.

b. Zdefiniuj zewnętrznie szablon struktury o nazwie `student`, zawierający trzy składniki: strukturę typu `daneos`, tablicę `oceny` przechowującą liczby zmiennoprzecinkowe oraz zmienną przechowującą średnią ocen.

c. W funkcji `main()` zadeklaruj tablicę złożoną z `ROZMIAR` (gdzie `ROZMIAR = 4`) struktur typu `student` i zainicjalizuj składniki typu `daneos` przy pomocy dowolnie wybranych imion i nazwisk. Użyj funkcji do wykonania zadań opisanych w punktach d, e, f i g.

d. Poproś użytkownika o podanie ocen kolejnych studentów i umieść je w składniku `oceny` odpowiednich struktur. Potrzebna do tego celu pętla może znajdować się – zgodnie z Twoimi preferencjami – w funkcji `main()` lub w funkcji pobierającej dane.

e. Oblicz średnią ocen dla każdej struktury i przypisz ją odpowiedniemu składnikowi.

f. Wyświetl informacje zawarte w każdej ze struktur.

g. Wyświetl średnią ocen wszystkich studentów.

6. Plik tekstowy zawiera informacje o drużynie koszykarskiej. Każdy z jego wierszy ma następujący układ:

```
10 Maciej Zielinski 14 10 8 5
```

Pierwszą pozycją jest numer gracza, należący do przedziału 0-18, drugą – imię gracza, a trzecią – nazwisko. Zarówno imię, jak i nazwisko składają się z jednego słowa. Kolejnymi wartościami są: liczba oddanych rzutów za 2 punkty, liczba trafień za 2 punkty, liczba asyst oraz liczba fauli. Plik może zawierać dane z więcej niż jednego meczu, a więc statystyki jednego gracza mogą być zapisane w kilku wierszach, przy czym wiersze te mogą przeplatać się ze statystykami innych graczy. Napisz program, który pobiera dane z pliku (aż do jego końca) i wczytuje je do tablicy struktur. Każda struktura powinna opisywać jednego gracza i zawierać składniki przeznaczone do przechowania jego imienia, nazwiska, ilości rzutów za 2 punkty, ilości rzutów trafionych, ilości asyst, ilości fauli (chodzi o całkowite ilości ze wszystkich meczów) oraz skuteczności (która zostanie obliczona później). Numer gracza może zostać użyty jako indeks tablicy.

Najprostszym sposobem, aby to osiągnąć, jest przypisanie wszystkim składnikom struktury wartości 0, wczytywanie danych liczbowych z pliku do zmiennych tymczasowych i dodawanie ich do składników odpowiedniej struktury. Po zakończeniu odczytywania pliku program powinien obliczyć skuteczność każdego gracza i zapisać ją w odpowiednim składniku każdej struktury. Skuteczność otrzymujemy przez podzielenie całkowitej liczby rzutów trafionych przez całkowitą liczbę rzutów oddanych; powinna być ona wartością zmiennoprzecinkową. Program powinien wyświetlić zbiorcze statystyki dla każdego gracza oraz dla całej drużyny.

7. Zmodyfikuj listing 14.11 tak, aby w miarę odczytywania kolejnych rekordów i wyświetlania ich na ekranie, możliwe było usunięcie lub zmiana zawartości każdego rekordu. W przypadku usunięcia rekordu w zwolnionym miejscu tablicy powinien zostać umieszczony następny odczytany rekord. Aby umożliwić zmianę zawartości pliku, będziesz musiał użyć trybu `"r+b"` zamiast `"a+b"`. Będziesz również musiał poświęcić więcej uwagi wskaźnikowi położenia tak, aby dodawane rekordy nie zamazywały rekordów istniejących. Najprostszym wyjściem jest przygotowanie danych w pamięci komputera, a następnie zapisanie ich ostatecznej wersji w pliku.

8. Flota linii lotniczych Colossus składa się z jednego samolotu o 12 miejscach. Samolot ten odbywa jeden rejs dziennie. Napisz program do rezerwacji miejsc spełniający następujące warunki:

a. Program wykorzystuje tablicę 12 struktur. Każda z nich powinna przechowywać numer identyfikacyjny miejsca, znacznik określający, czy miejsce jest wolne, oraz imię i nazwisko osoby, która dokonała rezerwacji.

b. Program wyświetla poniższe menu:

```
Aby wybrac opcje, wpisz jej oznaczenie literowe:
a.    Pokaz liczbe pustych miejsc
b.    Pokaz liste pustych miejsc
c.    Pokaz alfabetyczna liste miejsc
d.    Zarezerwuj miejsce dla klienta
e.    Usun rezerwacje miejsca
f.    Koniec
```

c. Program wykonuje czynności zapowiedziane w menu. Opcje `d)` i `e)` wymagają podania dodatkowych danych; każda z nich powinna umożliwić przerwanie wpisywania.

d. Po wykonaniu jednej z funkcji program powinien ponownie wyświetlić menu (nie dotyczy opcji `f)`).

e. Informacje o rezerwacjach są przechowywane w pliku.

9. Linie lotnicze Colossus nabyły drugi samolot (o tej samej liczbie miejsc) i rozszerzyły swoje usługi do czterech rejsów dziennie (rejsy 102, 311, 444 i 519). Zmodyfikuj program z poprzedniego ćwiczenia tak, aby obsługiwał on cztery rejsy. Dodaj nadrzędne menu, pozwalające wybrać jeden z lotów lub zakończyć program. Po wybraniu rejsu na ekranie powinno pojawić się menu podobne do tego z ćwiczenia nr 8. Powinno ono jednak zawierać dodatkową opcję: potwierdzenie rezerwacji miejsca. Ponadto, opcja „koniec" powinna zostać zastąpiona opcją „powróć do menu głównego". Przez cały czas na ekranie powinien być widoczny numer lotu, który jest aktualnie modyfikowany. Oprócz tego, lista alfabetyczna (opcja `c)`) powinna zawierać informację o potwierdzeniu rezerwacji dla każdego z miejsc.

10. Napisz program, który realizuje menu za pomocą tablicy wskaźników do funkcji. Na przykład, wybranie opcji a powodowałoby uruchomienie funkcji wskazywanej przez pierwszy element tablicy.

Dyrektywa #elif, niedostępna w niektórych starszych implementacjach, pozwala tworzyć konstrukcję przypominającą sekwencję else-if:

```
#if SYS == 1
     #include "ibmpc.h"
#elif SYS == 2
     #include "vax.h"
#elif SYS == 3
     #include "mac.h"
#else
     #include "general.h"
#endif
```

Wiele nowszych implementacji udostępnia alternatywny sposób sprawdzenia, czy dany identyfikator został zdefiniowany. Zamiast dyrektywy

```
#ifdef VAX
```

w implementacjach tych można skorzystać z formy:

```
#if defined (VAX)
```

defined jest tu operatorem preprocesora, który zwraca wartość 1, jeśli jego argument jest z#definiowany, a w przeciwnym wypadku – wartość 0. Zaletą nowszej formy jest to, iż można stosować ją w połączeniu z #elif. Przy jej pomocy poprzedni przykład można sformułować następująco:

```
#if defined (IBMPC)
     #include "ibmpc.h"
#elif defined (VAX)
     #include "vax.h"
#elif defined (MAC)
     #include "mac.h"
#else
     #include "general.h"
#endif
```

Rzecz jasna, w zależności od komputera, na którym uruchamiany jest program, wcześniej w pliku należy umieścić odpowiednią definicję stałej, taką jak poniższa:

```
#define VAX
```

Jednym z celów korzystania z kompilacji warunkowej jest zwiększenie przenośności programu. Dzięki dyrektywom preprocesora, możliwe jest utworzenie kilku wersji kodu dla różnych systemów, a następnie uaktywnianie ich przez prostą zmianę kilku kluczowych definicji na początku pliku.

Standard ANSI C zaleca, aby implementacje z nim zgodne zawierały definicję identyfikatora __STDC__. Dzięki temu, aby sprawdzić, czy kompilator jest zgodny z ANSI C, wystarczy użyć testu:

```
#if defined (__STDC__)
```

Typy wyliczeniowe

Słowo kluczowe enum jest powszechnym rozszerzeniem języka C, uwzględnionym w standardzie ANSI. Pozwala ono utworzyć nowy *typ wyliczeniowy* (ang. *enumerated type*) i określić wartości, jakie może on

przyjmować. (W rzeczywistości wszystkie wartości typu enum są liczbami całkowitymi i można je stosować wszędzie tam, gdzie dozwolone jest użycie wartości typu int.) Zadaniem typów wyliczeniowych jest zwiększenie czytelności programu przez zastąpienie niewiele znaczących liczb bardziej zrozumiałymi nazwami. Składnia definicji typu wyliczeniowego jest podobna, jak w przypadku struktur:

```
enum spektrum {czerwony, pomaranczowy, zolty, zielony, niebieski, fioletowy};
enum spektrum kolor;
```

Pierwsza deklaracja tworzy nowy typ o nazwie spektrum, a druga – należącą do tego typu zmienną kolor. Nazwy zawarte w klamrach są wartościami, jakie może przyjmować zmienna typu spektrum. Tym samym, możliwymi wartościami zmiennej kolor są: czerwony, pomaranczowy, zolty, itd. Po zadeklarowaniu zmiennej wyliczeniowej można korzystać z instrukcji, takich jak poniższe:

```
kolor = niebieski;
if (kolor = zolty)
   ...;
for (kolor = czerwony; kolor <= fioletowy; kolor++)
   ...;
```

Chociaż stałe wyliczeniowe należą zawsze do typu int, zmienne wyliczeniowe są określone mniej precyzyjnie i mogą należeć do dowolnego typu całkowitego, o ile typ ten jest w stanie przechować wszystkie możliwe wartości zmiennej. Na przykład, stałe wyliczeniowe typu spektrum należą do przedziału 0-7, a więc kompilator mógłby wyrazić zmienną kolor za pomocą typu unsigned char.

Stałe enum

Czym są nazwy niebieski i czerwony? z technicznego punktu widzenia są one stałymi całkowitymi. Oznacza to, że można ich użyć na przykład w następującej instrukcji:

```
printf("czerwony = %d, pomaranczowy = %d\n", czerwony, pomaranczowy);
```

Oto dane wyjściowe:

```
czerwony = 0, pomaranczowy = 1
```

Nazwa czerwony jest więc stałą reprezentującą liczbę 0, pozostałe identyfikatory symbolizują zaś liczby od 1 do 5. Stałe wyliczeniowe przypominają zatem stałe symboliczne z tym, że ich tworzeniem zajmuje się kompilator, a nie preprocesor.

Wartości domyślne

Stałe wyliczeniowe otrzymują standardowo wartości całkowite 0, 1, 2, i tak dalej. Stąd, w wyniku deklaracji

```
enum dzieci {antek, basia, cyryl, darek, ela};
```

stała darek reprezentuje wartość 3.

Wartości ustalone

Wartości reprezentowane przez stałe wyliczeniowe mogą również zostać narzucone z góry. W tym celu wystarczy użyć następującej formy deklaracji:

```
enum poziomy {niski = 100, sredni = 500, wysoki = 2000};
```

W przypadku, gdy po zdefiniowanej stałej następuje kilka stałych, którym nie przypisano wartości, stałe te są definiowane automatycznie przez kompilator. Sposób, w jaki się to odbywa, ilustruje poniższy przykład.

```
enum koty {kot, rys = 10, puma, tygrys};
```

W powyższej deklaracji stała `kot` jest równa 0 (domyślnie), a stałe `rys`, `puma` i `tygrys` mają odpowiednio wartości 10, 11 i 12.

Zastosowania

Jak pamiętasz, celem stosowania typów wyliczeniowych jest zwiększenie czytelności programu. Jeśli program zajmuje się kolorami, korzystanie z nazw `czerwony` i `pomaranczowy` jest dużo bardziej intuicyjne niż korzystanie z liczb 0 i 1. Zauważ, że typy wyliczeniowe mogą być wykorzystywane jedynie wewnętrznie przez sam program. Podając wartość dla zmiennej `kolor` należy wpisać liczbę 1, a nie słowo `pomaranczowy` (chyba, że program zawiera odpowiednie instrukcje zamieniające łańcuch `"pomaranczowy"` na wartość `pomaranczowy`).

Ponieważ typ wyliczeniowy jest typem całkowitym, zmienne `enum` mogą być wykorzystywane w wyrażeniach w taki sam sposób, jak zmienne całkowite. Zmienne takie są na przykład wygodnymi etykietami instrukcji `case`.

Listing 16.9 przedstawia krótki program stosujący typ `enum`. Wykorzystuje on domyślną metodę przydzielania wartości stałym wyliczeniowym: stała `czerwony` otrzymuje wartość 0, która jest równocześnie indeksem tablicy `kolory` prowadzącym do łańcucha `"czerwony"`.

Listing 16.9. Program `enum.c`.

```
/* enum.c -- korzysta z wyliczen */
#include <stdio.h>
#include <string.h>
enum BOOL {FALSE, TRUE};
typedef enum BOOL BOOLEAN;
enum spektrum {czerwony, pomaranczowy, zolty, zielony, niebieski, fioletowy};
const char * kolory[] = {"czerwony", "pomaranczowy", "zolty",
                         "zielony", "niebieski", "fioletowy"};
#define DL 30
int main(void)
{
   char wybor[DL];
   enum spektrum kolor;
   BOOLEAN znal = FALSE;

   puts("Podaj kolor (pusty wiersz konczy program):");
   while (gets(wybor) != NULL && wybor[0] != '\0')
   {
      for (kolor = czerwony; kolor <= fioletowy; kolor++)
      {
         if (strcmp(wybor, kolory[kolor]) == 0)
         {
            znal = TRUE;
            break;
```

```
        }
      }
      if (znal == TRUE)
         switch(kolor)
         {
            case czerwony     : puts("Czerwone sa roze.");
                                break;
            case pomaranczowy : puts("Pomaranczowe sa nasturcje.");
                                break;
            case zolty        : puts("Zolte sa sloneczniki.");
                                break;
            case zielony      : puts("Zielona jest trawa.");
                                break;
            case niebieski    : puts("Niebieskie sa chabry.");
                                break;
            case fioletowy    : puts("Fioletowe sa fiolki.");
                                break;
         }
      else
         printf("Kolor %s nic mi nie mowi.\n", wybor);
      znal = FALSE;
      puts("Podaj prosze nastepny kolor (pusty wiersz konczy program):");
   }
   return 0;
}
```

Uwaga

Niektóre kompilatory standardowo zawierają definicję stałych `FALSE` i `TRUE`. Jeśli korzystasz z jednego z takich kompilatorów, w definicji `enum` powinieneś użyć innych nazw, np. `False` i `True`.

Wielu programistów woli tworzyć stałe za pomocą jednej instrukcji `enum` zamiast kilku dyrektyw `#define`.

Biblioteka języka C

Oficjalna biblioteka języka C została opracowana przez komitet ANSI C w oparciu o faktyczny standard, jakim była implementacja C dla systemu UNIX. W stosunku do wersji uniksowej wersja ANSI biblioteki wyróżnia się lepszą przenośnością, która została podyktowana postępującym rozpowszechnianiem się języka C w coraz większej liczbie środowisk.

Niektóre funkcje z biblioteki standardowej zostały już przez nas opisane – w poprzednich rozdziałach znajdują się omówienia m.in. funkcji wejścia/wyjścia, funkcji znakowych i funkcji łańcuchowych. W niniejszym rozdziale przyjrzymy się paru innym funkcjom, najpierw jednak zastanowimy się, na czym polega korzystanie z biblioteki.

Uzyskiwanie dostępu do biblioteki C

Funkcje bibliotekowe znajdują się zwykle w kilku różnych miejscach. Na przykład, funkcja `getchar()` jest zazwyczaj zdefiniowana jako makro w pliku `stdio.h`, a `strlen()` przechowywana jest z reguły w pliku biblioteki. Uzyskanie dostępu do miejsc, w których znajdują się funkcje, odbywa się w różny sposób w różnych implementacjach. Kolejne podrozdziały omawiają trzy najczęściej spotykane możliwości.

Rozdział 17

Zaawansowana reprezentacja danych

W tym rozdziale poznasz:

Funkcje

`więcej o malloc()`

W tym rozdziale dowiesz się więcej o możliwościach języka C w dziedzinie tworzenia różnych typów danych. Poznasz nowe algorytmy i rozwiniesz swoją zdolność koncepcyjnego opracowywania programów. Zapoznasz się również z abstrakcyjnymi typami danych (ATD).

Nauka języka programowania przypomina naukę muzyki, stolarstwa czy inżynierii. Na początku poznajesz narzędzia stosowane w zawodzie, grając gamy, ucząc się, jak trzymać młotek, rozwiązując niezliczone zadania ze spadającymi i zjeżdżającymi obiektami. Nabywaniem i doskonaleniem umiejętności technicznych zajmowałeś się w kolejnych rozdziałach tej książki, ucząc się, jak tworzyć zmienne, struktury, funkcje, itp. Z czasem osiągniesz jednak poziom, na którym korzystanie z narzędzi stanie się Twoją drugą naturą, a prawdziwym wyzwaniem będzie opracowanie projektu. Decydującym czynnikiem będzie wówczas umiejętność patrzenia na projektowany program jak na jedną spójną całość. Niniejszy rozdział koncentruje się na tym wyższym poziomie programowania. Zawarty tutaj materiał może okazać się nieco trudniejszy niż poprzednie rozdziały, ale opanowanie go powinno dać Ci również większą satysfakcję – dzięki niemu bowiem będziesz mógł awansować z czeladnika na mistrza.

Zaczniemy od przyjrzenia się kluczowemu aspektowi budowy programu: sposobowi reprezentacji przetwarzanych danych. Znalezienie dobrej reprezentacji jest w stanie sprawić, że napisanie całej reszty programu będzie lekkie, łatwe i przyjemne. Do tej pory poznałeś typy danych udostępniane bezpośrednio przez język C: proste zmienne, tablice, wskaźniki, struktury i unie.

Jednak określenie właściwej reprezentacji danych bardzo często nie ogranicza się do wyboru typu. Oprócz ustalenia sposobu przechowywania danych, należy również zdecydować o operacjach, jakie będzie można na nich wykonywać. Na przykład, implementacje języka C zazwyczaj przechowują zarówno typ `int`, jak i typ wskaźnikowy jako wartości całkowite, jednak te dwa typy cechują zupełnie inne zbiory dozwolonych działań. Możliwe jest pomnożenie dwóch wartości typu `int`, ale nie dwóch wskaźników. Możliwa jest dereferencja wskaźnika za pomocą operatora `*`, jednak to samo działanie jest pozbawione sensu w przypadku liczby całkowitej. Język C definiuje prawidłowe operacje dla swoich typów podstawowych, jednak projektując swoją własną reprezentację danych, zbiór dostępnych działań trzeba w wielu przypadkach określić samemu. W języku C można to osiągnąć przez opracowanie funkcji realizujących te działania. Ujmując rzecz w skrócie, projektowanie typu danych wymaga zdecydowania o sposobie przechowywania danych i opracowania zbioru funkcji je obsługujących.

W niniejszym rozdziale przyjrzysz się kilku przydatnym *algorytmom*, czyli przepisom na przetwarzanie danych. Jako programista nabędziesz cały repertuar takich przepisów i będziesz je stosował wielokrotnie do rozwiązywania podobnych problemów.

Rozdział ten poświęcony jest projektowaniu typów danych – procesowi, który w dużej mierze polega na dopasowywaniu algorytmów do reprezentacji. Poznasz w nim niektóre powszechnie stosowane formy danych, takie jak kolejki, listy i drzewa binarne.

Zaznajomisz się również z pojęciem abstrakcyjnego typu danych (ATD). ATD przechowuje metody i reprezentacje danych w sposób nastawiony na problemy, a nie na konkretny język programowania. Po zaprojektowaniu ten sam ATD może być z łatwością wykorzystywany w różnych okolicznościach. Zrozumienie abstrakcyjnych typów danych stanowi przygotowanie do wejścia do świata programowania obiektowego (PO) i języka C++.

Poznajemy reprezentację danych

Załóżmy, że chcemy napisać program-książkę adresową. Jakiej formy danych użyjemy do przechowania informacji? Ponieważ z każdą pozycją w książce związane są różnorodne informacje, rozsądnym rozwiązaniem jest przedstawienie jej za pomocą struktury. Ale w jaki sposób przechowamy kilka pozycji? Przy pomocy zwykłej tablicy struktur? Przy pomocy tablicy dynamicznej? Przy pomocy jakiejś innej formy danych? Czy pozycje powinny być ustawione w kolejności alfabetycznej? Czy będziemy chcieli wyszukiwać pozycje przez podanie kodu pocztowego lub numeru kierunkowego? Czynności, jakie będzie można wykonywać na danych, mogą zależeć od sposobu ich przechowywania. Mówiąc krótko, przed rzuceniem się w wir pisania kodu należy podjąć bardzo wiele decyzji na etapie projektowania.

Jak powinna wyglądać reprezentacja bitmapowego (rastrowego) obrazu graficznego, który chcemy przechować w pamięci? Obraz bitmapowy to taki, który składa się z wielu ustawianych niezależnie pikseli. W czasach ekranów czarno-białych stan każdego piksela (włączony lub wyłączony) mógł być wyrażony za pomocą jednego bitu (stąd nazwa *bitmapa*). W przypadku monitorów kolorowych, do opisania jednego piksela potrzeba więcej niż jednego bitu. Na przykład, wyrażenie każdego piksela za pomocą 8 bitów pozwala uzyskać na ekranie 256 różnych kolorów. W obecnych czasach powszechne są palety 65 536 kolorów (16 bitów na piksel), 16 777 216 kolorów (24 bity na piksel), a nawet 2 147 483 648 kolorów (32 bity na piksel). W trybie o 16 milionach kolorów i rozdzielczości 1024 x 768 pikseli, pojedynczy ekran zajmuje 18.9 miliona bitów (2.25 MB). Czy taką ilość danych należy przechować w „czystej" postaci, czy też należy opracować jakiś sposób ich skompresowania? Czy kompresja powinna być *bezstratna* (zachowująca wszystkie dane), czy też *stratna* (powodująca utratę stosunkowo mało istotnych danych)? Tak jak poprzednio, przed rozpoczęciem pisania kodu należy podjąć wiele decyzji projektowych.

Zajmijmy się konkretnym przypadkiem. Załóżmy, że chcesz napisać program, który pozwala wpisać listę wszystkich filmów, które widziałeś w ciągu roku. Dla każdego filmu chciałbyś przechować różne rodzaje informacji, takie jak: tytuł, rok produkcji, reżyser, główni aktorzy, długość w minutach, gatunek (komedia, science fiction, romans, itd.), Twoja ocena, i tak dalej. Takie postawienie problemu sugeruje użycie struktury dla każdego filmu i tablicy struktur dla całej listy. Dla uproszczenia ograniczymy strukturę do dwóch składników: tytułu filmu i oceny w skali od 0 do 10. Dość oszczędna implementacja programu znajduje się na listingu 17.1.

Listing 17.1. Program `filmy1.c`.

```
/* filmy1.c -- korzystanie z tablicy struktur */
#include <stdio.h>
#define ROZT        45      /* rozmiar tablicy przechowujacej tytul */
#define FMAX         5      /* maksymalna liczba filmow             */
struct film {
   char tytul[ROZT];
   int ocena;
};
```

```
int main(void)
{
   struct film filmy[FMAX];
   int i = 0;
   int j;

   puts("Podaj pierwszy tytul filmu:");
   while (i < FMAX && gets(filmy[i].tytul) != NULL &&
      filmy[i].tytul[0] != '\0')
   {
      puts("Podaj Twoja ocene <0-10>:");
      scanf("%d", &filmy[i++].ocena);
      while(getchar() != '\n')
         continue;
      puts("Podaj nastepny tytul filmu (pusty wiersz konczy program):");
   }
   if (i == 0)
      printf("Nie wpisano zadnych danych. ");
   else
      printf ("Oto lista filmow:\n");

   for (j = 0; j < i; j++)
      printf("Film: %s  Ocena: %d\n", filmy[j].tytul,
         filmy[j].ocena);
   printf("Do widzenia!\n");
   return 0;
}
```

Program tworzy tablicę struktur, a następnie wypełnia ją danymi wpisanymi przez użytkownika. Wpisywanie danych trwa do momentu przepełnienia tablicy (warunek i < FMAX), osiągnięcia końca pliku (warunek ze wskaźnikiem NULL) lub wciśnięcia klawisza Enter na początku wiersza (warunek ze znakiem '\0').

Powyższa implementacja kryje w sobie kilka niedostatków. Po pierwsze, program będzie najprawdopodobniej marnował dużo miejsca, ponieważ tytuły większości filmów są krótsze niż 45 znaków, a tylko niektóre są naprawdę długie, np. *Dyskretny urok burżuazji* czy *Won Ton Ton: Pies, który uratował Hollywood.* Po drugie, dla wielu ludzi ograniczenie do pięciu filmów w roku będzie zbyt restrykcyjne. Oczywiście zawsze możemy zwiększyć granicę, ale jaka wartość będzie dobra? Niektórzy ludzie oglądają 500 filmów rocznie, a więc stałą FMAX moglibyśmy zwiększyć do 500, jednak dla niektórych i ta wartość może być zbyt mała, podczas gdy dla innych będzie ona olbrzymim marnotrawstwem pamięci. Ponadto, niektóre kompilatory ustalają domyślne ograniczenie pamięci, jaka jest dostępna dla zmiennych automatycznych, takich jak filmy, i tak duża wartość FMAX może spowodować przekroczenie tego limitu. Z przeszkodą tą można się wprawdzie uporać przez użycie tablicy zewnętrznej lub statycznej bądź przez zwiększenie rozmiaru stosu, ale takie rozwiązanie leczy jedynie objawy, nie usuwając prawdziwego problemu.

A prawdziwym problemem jest tutaj nieelastyczność reprezentacji danych. W czasie pisania kodu jesteśmy zmuszeni do podjęcia decyzji, które powinny zostać podjęte w trakcie pracy programu. Aby zmienić tę niekorzystną sytuację, moglibyśmy użyć reprezentacji opartej na dynamicznym przydzielaniu pamięci:

```
#define ROZT        45      /* rozmiar tablicy przechowujacej tytul */
struct film {
   char tytul[ROZT];
   int ocena;
};
...
```

```
int n, i;
struct film * filmy;        /* wskaznik do struktury                */
...
printf("Podaj maksymalna liczbe filmow:\n");
scanf("%d", &n);
filmy = (struct film *) malloc(n * sizeof(struct film));
```

Tak jak w poprzednim rozdziale, z wskaźnika `filmy` możemy korzystać tak samo, jak ze zwykłej nazwy tablicy:

```
while (i < FMAX && gets(filmy[i].tytul) != NULL &&
         filmy[i].tytul[0] != '\0')
```

Użycie funkcji `malloc()` pozwala odłożyć określenie liczby elementów do czasu uruchomienia programu – nie musimy więc rezerwować 500 elementów, jeśli potrzebne jest tylko 20. Równocześnie jednak podejście to kładzie na użytkowniku obowiązek prawidłowego wpisania liczby filmów.

Listy łączone

Gdyby było to możliwe, chcielibyśmy móc dodawać nowe dane bez końca (lub do wyczerpania się pamięci), nie określając z góry liczby pozycji i nie angażując niepotrzebnie ogromnych obszarów pamięci. Możemy to osiągnąć, wywołując funkcję `malloc()` za każdym razem, kiedy użytkownik wpisze nową pozycję, i zawsze rezerwując tylko tyle miejsca, ile zajmuje ta pozycja. Jeśli użytkownik wpisze trzy filmy, program wywoła `malloc()` trzy razy. Jeśli użytkownik wpisze 300 filmów, funkcja `malloc()` zostanie uruchomiona 300 razy.

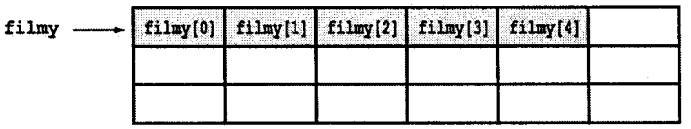
Ten znakomity pomysł rodzi jednak nowy problem. Aby zobaczyć, o co chodzi, porównaj zarezerwowanie miejsca dla 300 struktur typu `film` na raz i rezerwowanie miejsca „po jednej strukturze" za pomocą 300 oddzielnych wywołań funkcji `malloc()`. W pierwszym przypadku przydzielona pamięć tworzy jeden ciągły blok i do odwoływania się do jej zawartości wystarczy pojedyncza zmienna typu „wskaźnik do `struct film`", wskazująca na pierwszą strukturę bloku. Dostęp do poszczególnych struktur odbywa się za pomocą zwykłej notacji tablicowej, co ilustruje ostatni fragment kodu. Problem z drugim przedstawionym podejściem polega na tym, że nie ma żadnej gwarancji, że kolejne wywołania funkcji `malloc()` będą przydzielały kolejne, sąsiadujące ze sobą bloki pamięci. Oznacza to, że struktury nie muszą być rozmieszczone w postaci ciągłego obszaru (patrz rys. 17.1). W efekcie zamiast jednego wskaźnika do bloku 300 struktur potrzebne jest 300 wskaźników, po jednym dla każdej utworzonej niezależnie struktury!

```
struct film * filmy;
filmy = (struct film *) malloc(5*sizeof(struct film));

filmy; filmy[0]; filmy[1]...

int i;
struct film * filmy[s];

for (i = 0; i < 5; i++)
filmy[i] = (struct film *) malloc(sizeof(struct film));
```

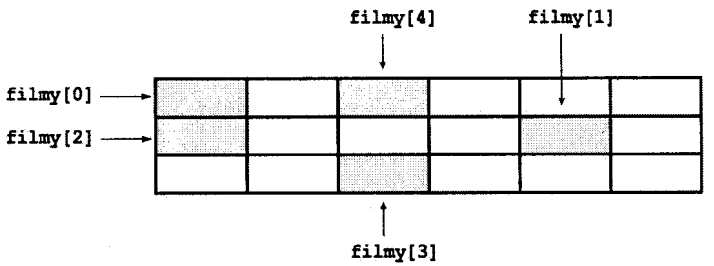
Jednym z możliwych rozwiązań, z którego nie skorzystamy, jest utworzenie dużej tablicy wskaźników i przypisywanie im wartości w miarę tworzenia kolejnych struktur:

```
#define ROZT        45      /* rozmiar tablicy przechowujacej tytul */
#define FMAX         5      /* maksymalna liczba filmow             */
struct film {
   char tytul[ROZT];
   int ocena;
};
...
struct film * filmy[FMAX];  /* tablica wskaznikow do struktur       */
int i;
...
filmy[i] = (struct film *) malloc(n * sizeof(struct film));
```

```
struct film * filmy;
filmy = (struct film *) malloc(5*sizeof(struct film));
```




```
int i;
struct film * filmy[5];

for (i = 0; i < 5; i++)
filmy[i] = (struct film *) malloc(sizeof(struct film));
```



Rysunek 17.1. Alokacja struktur – razem i oddzielnie.

Podejście to oszczędza sporo pamięci, ponieważ tablica 500 wskaźników pochłania dużo mniej miejsca niż tablica 500 struktur. Tym niemniej, obszar zarezerwowany dla nieużywanych wskaźników jest nadal marnowany, a ponadto nadal obowiązuje ograniczenie do 500 struktur.

Istnieje lepszy sposób. Przydzielając miejsce dla nowej struktury, można równocześnie przydzielić miejsce dla nowego wskaźnika. „Ale" – powiesz – „wówczas potrzebny będzie kolejny wskaźnik, który pozwoli uzyskać dostęp do nowego wskaźnika, a potem wskaźnik do tego wskaźnika, i tak dalej." Uniknięcie tego potencjalnego problemu polega na takiej zmianie szablonu `film`, aby każda struktura zawierała wskaźnik do *następnej* struktury. Wówczas, tworząc nową strukturę, możemy zapisać jej adres w strukturze poprzedniej. Krótko mówiąc, definicja szablonu `film` powinna wyglądać następująco:

```
#define ROZT        45   /* rozmiar tablicy przechowujacej tytul */
struct film {
   char tytul[ROZT];
```

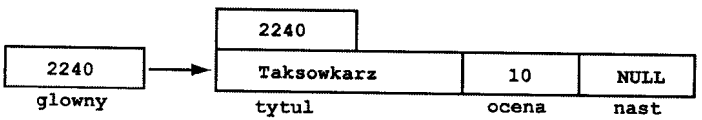
```
   int ocena;
   struct film * nast;
};
```

Co prawda struktura nie może zawierać w sobie struktury tego samego typu, ale *może* zawierać wskaźnik do takiej struktury. Taka definicja jest podstawą *listy łączonej* (ang. *linked list*) – ciągu pozycji, z których każda zawiera informację o miejscu przechowywania następnej pozycji.

Zanim przyjrzymy się implementacji listy łączonej w języku C, spróbujmy wykonać doświadczenie myślowe polegające na przejściu przez taką listę. Załóżmy, że użytkownik wpisał tytuł `Taksowkarz` i ocenę `10`. Program zarezerwowałby miejsce dla struktury typu `film`, skopiowałby łańcuch `Taksowkarz` do składnika `tytul` i przypisałby składnikowi `ocena` wartość `10`. Aby zaznaczyć, że po bieżącej strukturze nie ma już żadnych innych struktur, program nadałby składnikowi `nast` wartość `NULL`. (Jak pamiętasz, `NULL` jest stałą symbolizującą wskaźnik zerowy, zdefiniowaną w pliku `stdio.h`.) Rzecz jasna, musimy wiedzieć, gdzie znajduje się utworzona przez nas pierwsza struktura. W tym celu możemy przypisać jej adres osobnemu wskaźnikowi, który będziemy nazywać *wskaźnikiem głównym* (ang. *head pointer*). Wskaźnik główny wskazuje na pierwszy element listy łączonej. Wygląd utworzonej struktury przedstawia rys. 17.2. (Puste miejsce w składniku `tytul` zostało skrócone tak, aby mogło zmieścić się na rysunku.)

```
#define ROZT 45
struct film {
   char tytul[ROZT];
   int ocena;
   struct film * nast;
};
struct film * glowny;
```

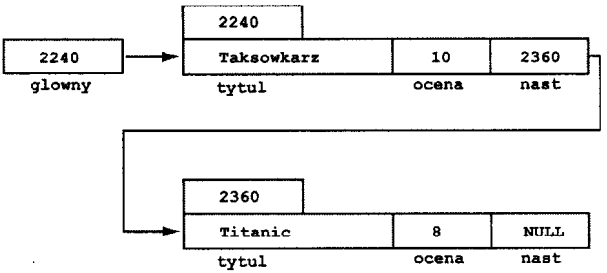
```
#define ROZT 45
struct film {
   char tytul[ROZT];
   int ocena;
   struct film * nast;
};
struct film * glowny;
```



Rysunek 17.2. Pierwsza pozycja listy łączonej.

Załóżmy teraz, że użytkownik wpisał drugi tytuł filmu wraz z oceną – na przykład `Titanic` i `8`. Program przydzieliłby miejsce nowej strukturze typu `film`, zapisując jej adres w składniku `nast` pierwszej struktury (zastępując umieszczony tam wcześniej wskaźnik `NULL`). Wskaźnik `nast` w pierwszej pozycji wskazuje zatem na drugą pozycję listy łączonej. Następnie program skopiowałby wartości `Titanic` i `8` do nowej struktury i nadałby wartość `NULL` jej składnikowi `nast`, aby zasygnalizować, że jest ona ostatnią strukturą w liście. Schemat listy dwóch pozycji jest przedstawiony na rys. 17.3.

Każdy kolejny element zostanie obsłużony w ten sam sposób. Jego adres zostanie zapisany w poprzedniej strukturze, składniki `tytul` i `ocena` nowej struktury otrzymają dane o filmie, a składnik `nast` otrzyma wartość `NULL`. Rezultatem tego procesu jest lista łączona o budowie przedstawionej na rys. 17.4.



Rysunek 17.3. Lista łączona dwóch elementów.

Załóżmy, że chcemy wyświetlić listę łączoną. Wyświetlając dowolną strukturę możemy skorzystać z przechowywanego w niej adresu, aby zlokalizować następną pozycję do wyświetlenia. Jednak aby rozpocząć, potrzebujemy wskaźnika do pierwszego elementu w liście, ponieważ nie zawiera go żaden inny element. Rolę tę pełni wskaźnik główny.

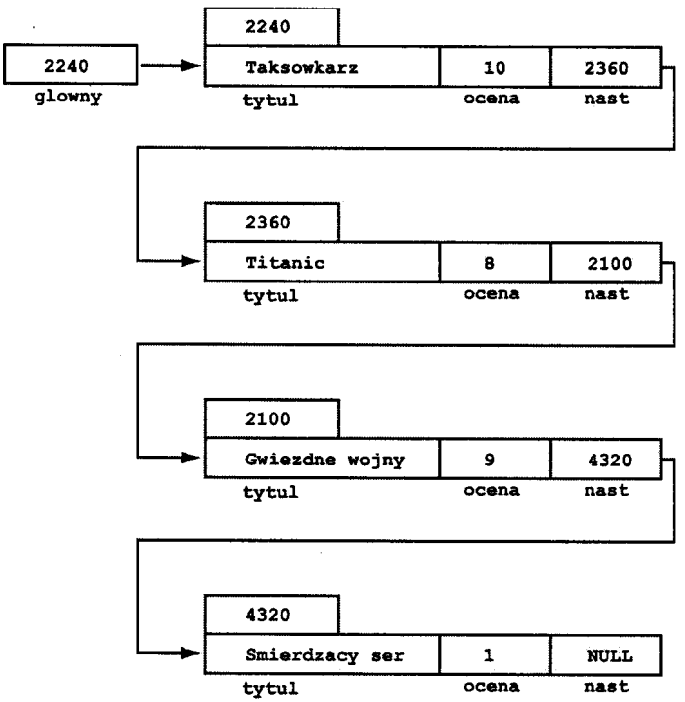
Korzystanie z listy łączonej

Skoro przedstawiliśmy już zasadę działania listy łączonej, spróbujmy ją zaimplementować. Listing 17.2 jest modyfikacją listingu 17.1, przechowującą informacje o filmach w liście łączonej zamiast tablicy.

Listing 17.2. Program `filmy2.c`.

```
/* filmy2.c -- korzystanie z listy laczonej struktur */
#include <stdio.h>
#include <stdlib.h>      /* zawiera prototyp malloc()              */
#include <string.h>      /* zawiera prototyp strcpy()              */
#define ROZT    45       /* rozmiar tablicy przechowujacej tytul   */
struct film {
   char tytul[ROZT];
   int ocena;
   struct film * nast;   /* wskazuje na nastepna strukture w liscie */
};

int main(void)
{
   struct film * glowny = NULL;
   struct film * poprz, * biezacy;
   char wejscie[ROZT];

   puts("Podaj pierwszy tytul filmu:");
   while (gets(wejscie) != NULL && wejscie[0] != '\0')
   {
      biezacy = (struct film *) malloc(sizeof(struct film));
      if (glowny == NULL)         /* pierwsza struktura    */
         glowny = biezacy;
      else                        /* kolejne struktury     */
         poprz->nast = biezacy;
      biezacy->nast = NULL;
      strcpy(biezacy->tytul, wejscie);
      puts("Podaj Twoja ocene <0-10>:");
      scanf("%d", &biezacy->ocena);
      while(getchar() != '\n')
         continue;
```

```
      puts("Podaj nastepny tytul filmu (pusty wiersz konczy program):");
      poprz = biezacy;
   }
   if (glowny == NULL)
      printf("Nie wpisano zadnych danych. ");
   else
      printf ("Oto lista filmow:\n");
   biezacy = glowny;
   while (biezacy != NULL)
   {
      printf("Film: %s  Ocena: %d\n", biezacy->tytul, biezacy->ocena);
      biezacy = biezacy->nast;
   }
   printf("Do widzenia!\n");
   return 0;
}
```



Rysunek 17.4. Lista łączona kilku elementów.

Program wykonuje dwa główne zadania. Na początku tworzy on kolejne elementy listy łączonej i wypełnia je wpisywanymi danymi, a następnie wyświetla całą listę. Ponieważ wyświetlanie jest czynnością prostszą, od niego rozpoczniemy nasze omówienie.

Wyświetlanie listy

Rozpoczynamy od przypisania wskaźnikowi (nazwijmy go biezacy) adresu pierwszej struktury. Ponieważ na strukturę tę wskazuje wskaźnik główny (o nazwie glowny), w tym celu wystarcza następująca instrukcja:

```
biezacy = glowny;
```

Składniki tej pozycji możemy odczytać za pomocą notacji wskaźnikowej:

```
printf("Film: %s  Ocena: %d\n", biezacy->tytul, biezacy->ocena);
```

Kolejny krok polega na takiej zmianie wartości wskaźnika biezacy, aby wskazywał on na następną strukturę w liście. Adres tej struktury jest zapisany w składniku nast struktury bieżącej, zatem odpowiedni kod wygląda następująco:

```
biezacy = biezacy->nast;
```

Następnie cały proces wykonujemy ponownie (od instrukcji printf()). Po wyświetleniu ostatniego elementu listy, wskaźnik biezacy będzie równy NULL, ponieważ taką wartość ma składnik nast ostatniej struktury. Fakt ten możemy wykorzystać do zakończenia pętli wyświetlającej. Oto cały kod odpowiedzialny za wyświetlanie listy w programie filmy2.c:

```
while (biezacy != NULL)
{
   printf("Film: %s  Ocena: %d\n", biezacy->tytul, biezacy->ocena);
   biezacy = biezacy->nast;
}
```

Dlaczego musieliśmy utworzyć nowy wskaźnik o nazwie biezacy, aby przejść przez listę, zamiast po prostu skorzystać z istniejącego wcześniej wskaźnika glowny? Ponieważ użycie wskaźnika głównego spowodowałoby zmianę jego wartości, wskutek czego program utraciłby bezpowrotnie możliwość zlokalizowania początku listy.

Tworzenie listy

Tworzenie listy łączonej składa się z trzech kroków:

1. Wywołanie funkcji malloc() w celu przydzielenia miejsca dla struktury.
2. Zapisanie adresu struktury.
3. Skopiowanie do struktury odpowiednich danych.

Nie ma sensu tworzyć struktury, jeśli jest ona niepotrzebna, dlatego program umieszcza wpisany przez użytkownika tytuł filmu w obszarze tymczasowym (tablicy wejscie). Jeśli użytkownik wpisze pusty wiersz lub znak EOF, pętla odczytująca ulega zakończeniu, a struktura nie zostaje utworzona.

```
while (gets(wejscie) != NULL && wejscie[0] != '\0')
```

Jeśli wpisano zwykłe dane wejściowe, program rezerwuje miejsce dla struktury i przypisuje jej adres zmiennej wskaźnikowej biezacy:

```
biezacy = (struct film *) malloc(sizeof(struct film));
```

Adres pierwszej struktury w liście powinien zostać umieszczony w zmiennej wskaźnikowej glowny, a adres każdej następnej struktury – w składniku nast struktury ją poprzedzającej. Stąd, program musi wiedzieć, czy ma do czynienia z pierwszą strukturą, czy też nie. Prostą metodą jest przypisanie wskaźnikowi glowny wartości NULL na początku programu. Dzięki temu program może podjąć decyzję w oparciu o wartość wskaźnika glowny.

```
if (glowny == NULL)          /* pierwsza struktura    */
    glowny = biezacy;
else                         /* kolejne struktury     */
    poprz->nast = biezacy;
```

W powyższym kodzie poprz jest wskaźnikiem przechowującym adres struktury utworzonej poprzednio.

Następnie wszystkie składniki struktury powinny otrzymać właściwe wartości. W szczególności składnik nast powinien otrzymać wartość NULL, ponieważ bieżąca struktura jest ostatnią w liście. W składniku tytul należy umieścić łańcuch z tablicy wejscie, a w składniku ocena – wpisaną przez użytkownika ocenę. Wszystkie te czynności wykonuje poniższy kod:

```
biezacy->nast = NULL;
strcpy(biezacy->tytul, wejscie);
puts("Podaj Twoja ocene <0-10>:");
scanf("%d", &biezacy->ocena);
```

Ostatnim etapem są przygotowania do następnego cyklu pętli. W szczególności wskaźnik poprz powinien otrzymać adres bieżącej struktury, ponieważ po wpisaniu kolejnego tytułu filmu i utworzeniu nowego elementu listy stanie się ona strukturą poprzednią. Odpowiednia instrukcja przypisania znajduje się na końcu pętli:

```
poprz = biezacy;
```

Czy program działa? Oto odpowiedź:

```
Podaj pierwszy tytul filmu:
Szeregowiec Ryan
Podaj Twoja ocene <0-10>:
7
Podaj nastepny tytul filmu (pusty wiersz konczy program):
The Big Lebowski
Podaj Twoja ocene <0-10>:
8
Podaj nastepny tytul filmu (pusty wiersz konczy program):
Inwazja krwiozerczych pomidorow
Podaj Twoja ocene <0-10>:
1
Podaj nastepny tytul filmu (pusty wiersz konczy program):

Oto lista filmow:
Film: Szeregowiec Ryan  Ocena: 7
Film: The Big Lebowski  Ocena: 8
Film: Inwazja krwiozerczych pomidorow  Ocena: 1
Do widzenia!
```