Stephen Prata

Język C

Szkoła programowania

Wydanie VI

Naucz się C, a zrozumiesz istotę programowania!

Helion

Tytuł oryginału: C Primer Plus, 6th Edition

Tłumaczenie: Przemysław Szeremiota

ISBN: 978-83-283-1470-2

Authorized translation from the English language edition, entitled:
C PRIMER PLUS, Sixth Edition; ISBN 0321928423; by Stephen Prata;
published by Pearson Education, Inc, publishing as Addison Wesley.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/jcszp6.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/jcszp6
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

SPIS TREŚCI

FUNKCJE

Zagadnienia poruszone w tym rozdziale:

- Słowo kluczowe: `return`
- Operatory jednoargumentowe: `*` i `&`
- Funkcje i metody ich definiowania
- Wykorzystanie argumentów i zwracanych wartości
- Zastosowanie wskaźników jako argumentów funkcji
- Typy funkcji
- Prototypy ANSI C
- Rekurencja

a czym polega organizacja programu? Częścią filozofii języka C jest stosowanie funkcji jako podstawowych elementów budulcowych. Do tej pory korzystałeś głównie ze standardowych funkcji bibliotecznych, takich jak `printf()`, `scanf()`, `getchar()`, `putchar()` czy `strlen()`. Teraz jesteś gotowy, aby wziąć na siebie bardziej twórcze zadanie — tworzenie własnych funkcji. Niektórym aspektom tego procesu miałeś okazję przyjrzeć się w poprzednich rozdziałach — niniejszy rozdział systematyzuje i rozszerza podane wcześniej informacje.

Przypomnienie

Czym jest funkcja? **Funkcja** jest wydzielonym fragmentem kodu programu, spełniającym określone zadanie. Strukturę funkcji i sposób ich używania wyznaczają reguły składniowe języka C. Funkcje w języku C odgrywają tę samą rolę co funkcje, podprogramy i procedury w innych językach programowania, choć szczegóły ich stosowania mogą być inne. Niektóre funkcje powodują wykonanie jakiejś czynności. Na przykład funkcja `printf()` powoduje wyświetlenie danych na ekranie. Zadaniem innych funkcji jest obliczenie wartości potrzebnej w programie. Na przykład funkcja `strlen()` informuje program o długości danego łańcucha. Ogólnie rzecz biorąc, funkcja może zarówno wykonywać czynności, jak i zwracać wartości.

Dlaczego powinieneś korzystać z funkcji? Po pierwsze, pozwalają one uniknąć powtarzania tych samych fragmentów kodu. Jeśli określone zadanie ma zostać wykonane kilkakrotnie, wystarczy napisać jedną funkcję i wywoływać ją tam, gdzie jest to potrzebne. Jedną funkcję można ponadto wykorzystywać w wielu programach tak samo, jak robiliśmy to z funkcjami `putchar()` czy `printf()`. Po drugie, w postaci funkcji warto jest przedstawić nawet czynność wykonywaną raz i tylko w jednym programie, ponieważ zwiększa to modularność programu, a tym samym czyni go czytelniejszym i łatwiejszym w modyfikacji. Załóżmy, że chcesz napisać program, który wykonuje następujące czynności:

- Wczytaj listę liczb
- Uporządkuj liczby
- Znajdź wartość średnią
- Narysuj wykres słupkowy

Mógłbyś użyć poniższego kodu:

```
#include <stdio.h>
#define ROZMIAR 50
int main(void)
{
  float lista[ROZMIAR];
  wczytajliste(lista, ROZMIAR);
  uporzadkuj(lista, ROZMIAR);
  srednia(lista, ROZMIAR);
  wykres(lista, ROZMIAR);
  return 0;
}
```

Oczywiście musiałbyś napisać jeszcze — drobny szczegół — cztery użyte w programie funkcje: `wczytajliste()`, `uporzadkuj()`, `srednia()` i `wykres()`. Zauważ, że opisowe nazwy funkcji pokazują w jasny sposób działanie i organizację programu. Dzięki podejściu modularnemu możesz pracować nad każdą funkcją z osobna, dopóki nie będzie wykonywała ona swojego zadania w należyty sposób, a jeśli Twoje funkcje będą wystarczająco wszechstronne, będziesz mógł wykorzystać je również w innych programach.

Wielu programistów lubi wyobrażać sobie funkcje jako „czarne skrzynki", określone przez wchodzące do nich dane (wejście) oraz wykonywane przez nie działania lub zwracane wartości (wyjście). To, co dzieje się wewnątrz funkcji „czarnej skrzynki", nie jest Twoim zmartwieniem, jeśli tylko nie jesteś osobą, która musi ją napisać. Gdy korzystasz na przykład z funkcji `printf()`, wiesz, że musisz przekazać jej łańcuch sterujący i argumenty. Wiesz również, jakie skutki pociągnie za sobą jej wywołanie. Nie musisz natomiast zastanawiać się nad kodem, z jakiego składa się funkcja `printf()`. Traktowanie funkcji w ten sposób pomaga skoncentrować się na ogólnej budowie programu, bez zagłębiania się w szczegóły. Zanim zaczniesz myśleć o tym, jak napisać funkcję, zastanów się dokładnie nad tym, jakie zadania ma ona realizować i jakie jest jej miejsce w programie jako całości.

Co musisz wiedzieć o funkcjach? Musisz wiedzieć, jak je właściwie definiować, jak je wywoływać oraz jak zrealizować wymianę informacji między różnymi funkcjami. Aby odświeżyć Twoją pamięć, rozpoczniemy od bardzo prostego przykładu, do którego dodawać będziemy kolejne elementy aż do otrzymania pełnego obrazu.

Tworzenie i wykorzystanie prostej funkcji

Naszym pierwszym skromnym celem jest utworzenie funkcji, która wyświetla rząd 40 gwiazdek (*). Aby zaopatrzyć ją w jakiś kontekst, umieścimy ją w programie wyświetlającym prosty nagłówek listu. Gotowy program, składający się z dwóch funkcji — `main()` i `gwiazdki()` — przedstawiony jest na listingu 9.1.

LISTING 9.1. Program naglowek1.c

```
/* naglowek1.c */
#include <stdio.h>
#define NAZWA "MEGATHINK, INC."
#define ADRES "10 Megabuck Plaza"
#define MIEJSCOWOSC "Megapolis, CA 94904"
#define LIMIT 40
void gwiazdki(void);  /* prototyp funkcji */
int main(void)
{
    gwiazdki();
    printf("%s\n", NAZWA);
    printf("%s\n", ADRES);
    printf("%s\n", MIEJSCOWOSC);
    gwiazdki();           /* wywolanie funkcji */
    return 0;
}
void gwiazdki(void)       /* definicja funkcji */
{
    int licznik;
    for (licznik = 1; licznik <= LIMIT; licznik++)
        putchar('*');
    putchar('\n');
}
```

Oto wynik działania programu:

```
****************************************
MEGATHINK, INC.
10 Megabuck Plaza
Megapolis, CA 94904
****************************************
```

Analiza programu

- Identyfikator `gwiazdki` występuje w trzech różnych kontekstach: w *prototypie funkcji*, który dostarcza kompilatorowi ogólnych informacji o funkcji `gwiazdki()`, w *wywołaniu funkcji*, które uruchamia funkcję oraz w *definicji funkcji*, która zawiera jej kod.

- Podobnie jak zmienne, funkcje należą do typów. Każdy program wykorzystujący funkcję powinien deklarować jej typ, zanim zostanie ona wywołana. Dlatego przed definicją funkcji `main()` umieszczony został następujący prototyp ANSI C:

  ```
  void gwiazdki(void);
  ```

  Nawiasy wskazują, że `gwiazdki` jest nazwą funkcji. Pierwsze słowo `void` określa typ funkcji — oznacza ono, że funkcja nie zwraca żadnej wartości. Drugie słowo `void` (zawarte w nawiasie) wskazuje, że funkcja nie pobiera argumentów. Średnik oznacza, że wiersz jest deklaracją funkcji, a nie jej definicją. Wiersz `void gwiazdki(void);` informuje zatem kompilator, że program wykorzystuje funkcję o nazwie `gwiazdki`, która nie przyjmuje argumentów i nie zwraca wartości, i że definicji funkcji należy szukać w innym miejscu. Jeśli posiadasz kompilator, który nie rozpoznaje prototypów ANSI C, po prostu zadeklaruj typ funkcji, tak jak poniżej:

  ```
  void gwiazdki();
  ```

  Niektóre bardzo stare kompilatory nie rozpoznają również typu `void`. W takim przypadku skorzystaj z typu `int`. I koniecznie rozejrzyj się za kompilatorem z obecnego stulecia.

- Co do zasady, prototyp określa zarówno typ wartości zwracanej przez funkcję, jak i typy wszystkich argumentów oczekiwanych w wywołaniu funkcji; łącznie ta informacja to tak zwana *sygnatura* funkcji. W tym konkretnym przypadku sygnatura mówi, że funkcja nie zwraca wartości i nie przyjmuje argumentów.

- W naszym programie prototyp funkcji `gwiazdki()` znajduje się przed słowem `main()`; mógłby on znajdować się wewnątrz funkcji `main()`, w miejscu, gdzie umieszcza się deklaracje zmiennych.

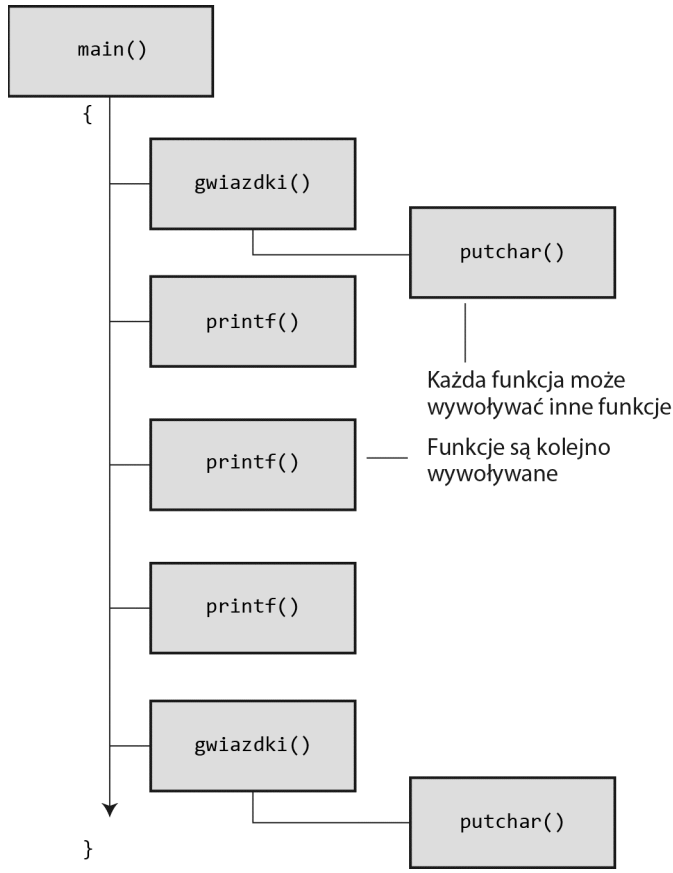
- Program wywołuje funkcję `gwiazdki()` w funkcji `main()` przez podanie jej nazwy wraz z nawiasem i średnikiem:

  ```
  gwiazdki();
  ```

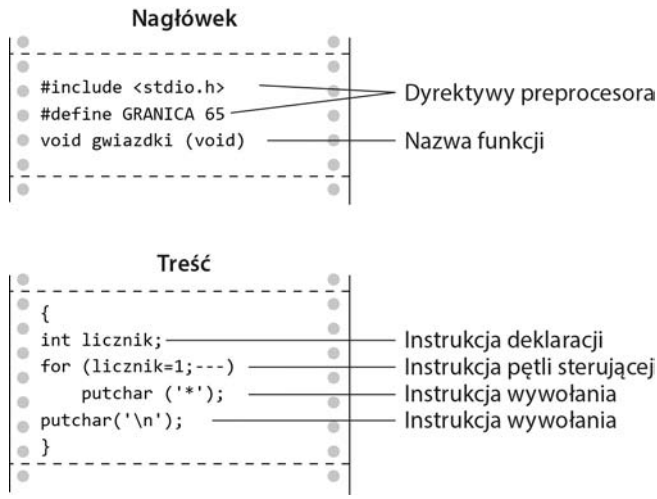
  Jest to jeden ze sposobów wywołania funkcji. Zawsze, gdy komputer napotka instrukcję `gwiazdki();`, odszukuje funkcję `gwiazdki()` i wykonuje zawarte w niej instrukcje. Po zakończeniu wykonywania kodu wewnątrz funkcji `gwiazdki()` komputer wraca do kolejnego wiersza *funkcji wywołującej* — w tym przypadku `main()` (patrz rysunek 9.1). (Dokładnie rzecz biorąc, kompilator C tłumaczy kod funkcji i programu na kod maszynowy, który zachowuje się zgodnie z powyższym opisem).

- Definicja funkcji `gwiazdki()` wygląda tak samo, jak funkcji `main()`. Rozpoczyna się ona typem funkcji, jej nazwą i nawiasami. Dalej następuje klamra otwierająca, deklaracja użytych zmiennych, instrukcje składające się na funkcję oraz klamra zamykająca (patrz rysunek 9.2). Zauważ, że w tym wypadku po nazwie `gwiazdki()` nie występuje średnik. Brak średnika informuje kompilator, że funkcja jest *definiowana*, a nie *deklarowana*.

RYSUNEK 9.1. Przebieg programu naglowek1.c (patrz listing 9.1)



RYSUNEK 9.2. Budowa prostej funkcji



- W naszym przykładzie funkcje `gwiazdki()` i `main()` znajdują się w tym samym pliku, mógłbyś jednak użyć dwóch oddzielnych plików. Model jednoplikowy jest nieco łatwiejszy w kompilacji, z kolei umieszczenie funkcji `gwiazdki()` w osobnym pliku ułatwia wykorzystanie jej w innych programach. Jeśli zdecydujesz się na ten drugi wariant, w pliku zawierającym funkcję `gwiazdki()` musisz również umieścić wszystkie potrzebne dyrektywy `#define` i `#include`. Zagadnieniem korzystania z dwóch lub więcej plików kodu źródłowego zajmiemy się później; tymczasem będziemy trzymać się opcji jednoplikowej. Klamra zamykająca pokazuje kompilatorowi koniec definicji funkcji `main()`; następujący za nim nagłówek `gwiazdki()` to dla kompilatora informacja, że `gwiazdki()` jest kolejną, odrębną funkcją.
- Zmienna `licznik` w funkcji `gwiazdki()` jest *zmienną lokalną*. Oznacza to, że jest ona znana tylko funkcji `gwiazdki()`. Mógłbyś zadeklarować zmienną o nazwie `licznik` w innej funkcji, włącznie z `main()`, bez narażania się na konflikt. Otrzymałbyś po prostu dwie niezależne zmienne o tej samej nazwie.

Jeśli potraktować funkcję `gwiazdki()` jako czarną skrzynkę, wykonywaną przez nią czynnością jest wyświetlenie rzędu gwiazdek. Funkcja nie posiada wejścia, ponieważ nie potrzebuje ona żadnych informacji z funkcji wywołującej. Nie posiada również wartości zwracanej, a więc nie dostarcza żadnych informacji do funkcji `main()`. Jednym słowem, funkcja `gwiazdki()` nie musi komunikować się z funkcją wywołującą.

Przyjrzyjmy się teraz sytuacji, w której komunikacja jest potrzebna.

Argumenty funkcji

Pokazany wcześniej nagłówek listu wyglądałby ładniej, gdyby tekst był w nim wyśrodkowany. Wyśrodkowanie tekstu odbywa się przez wyświetlenie przed nim odpowiedniej liczby odstępów. Przypomina to zadanie realizowane przez funkcję `gwiazdki()` — rzecz jasna z tą różnicą, że chodzi o wyświetlenie odstępów. Zamiast pisać dwie oddzielne funkcje dla gwiazdek i dla odstępów, utworzymy jedną wszechstronną funkcję, która będzie w stanie wykonać oba zadania. Nazwiemy ją `n_znak()` (aby zaznaczyć, że wyświetla ona znak *n* razy). Zamiast wbudowywać wyświetlany znak oraz liczbę powtórek w kodzie funkcji, wartości te będziemy przekazywać jako argumenty.

Przejdźmy do konkretów. Załóżmy, że w terminalu mamy dokładnie 40 kolumn znaków wyświetlanych, więc rząd gwiazdek ma mieć szerokość 40 znaków; stąd do jego wyświetlenia powinno posłużyć wywołanie `n_znak('*', 40);`. Co z odstępami? Tekst `MEGATHINK, INC.` ma szerokość 15 znaków, a więc w pierwszej wersji programu następowało po nim 25 odstępów. Aby go wyśrodkować, należy go przesunąć o 12 znaków w prawo, co spowoduje, że po jego obu stronach znajdować się będzie podobna liczba (12 po jednej i 13 po drugiej) odstępów. Należy więc skorzystać z wywołania `n_znak(' ', 12);`.

Poza tym, że wykorzystuje ona argumenty, funkcja `n_znak()` jest zupełnie podobna do funkcji `gwiazdki()`. Istotną różnicą jest fakt, iż `n_znak()` nie powinna wyświetlać znaku nowej linii, ponieważ po jej wywołaniu konieczne będzie wyświetlenie tekstu w tym samym wierszu. Nowa wersja programu przedstawiona jest na listingu 9.2. Aby zilustrować, jak działają argumenty, program przekazuje do funkcji `n_znak()` bardzo różnorodne wyrażenia.

LISTING 9.2. Program naglowek2.c

```
/* naglowek2.c */
#include <stdio.h>
#include <string.h>          /* zawiera prototyp strlen()     */
#define NAZWA "MEGATHINK, INC."
#define ADRES "10 Megabuck Plaza"
#define MIEJSCOWOSC "Megapolis, CA 94904"
#define LIMIT 40
#define ODSTEP ' '
void n_znak(char ch, int num);
int main(void)
{
  int odstepy;
  n_znak('*', LIMIT);        /* stale jako argumenty          */
  putchar('\n');
  n_znak(ODSTEP, 12);        /* stale jako argumenty          */
  printf("%s\n", NAZWA);
  odstepy = (LIMIT - strlen(ADRES)) / 2;
                             /* program oblicza, ile odstepow */
                             /* nalezy wyswietlic             */
  n_znak(ODSTEP, odstepy);   /* zmienna jako argument         */
  printf("%s\n", ADRES);
  n_znak(ODSTEP, (LIMIT - strlen(MIEJSCOWOSC)) / 2);
                             /* wyrazenie jako argument       */
  printf("%s\n", MIEJSCOWOSC);
  n_znak('*', LIMIT);
  putchar('\n');
  return 0;
}
/* definicja funkcji n_znak() */
void n_znak(char ch, int num)
{
  int licznik;
  for (licznik = 1; licznik <= num; licznik++)
    putchar(ch);
}
```

Oto wynik uruchomienia powyższego programu:

```
****************************************
            MEGATHINK, INC.
           10 Megabuck Plaza
          Megapolis, CA 94904
****************************************
```

Przypomnijmy sobie teraz sposób tworzenia funkcji przyjmującej argumenty, a następnie przyjrzyjmy się, jak należy z niej korzystać.

Definiowanie funkcji pobierającej argument — argumenty formalne

Definicja funkcji rozpoczyna się poniższą deklaracją:

```
void n_znak(char ch, int num)
```

Wiersz ten informuje kompilator, że funkcja `n_znak()` pobiera dwa argumenty o nazwach `ch` i `num`, że `ch` należy do typu `char` oraz że `num` należy do typu `int`. Zmienne `ch` i `num` nazywamy **argumentami formalnymi**. Podobnie jak zmienne zadeklarowane wewnątrz funkcji, argumenty formalne są zmiennymi lokalnymi, stanowiącymi prywatną własność funkcji. Oznacza to, że w innych funkcjach mogą istnieć niezależne zmienne o tych samych nazwach. Zmienne `ch` i `num` otrzymują wartości przy każdym wywołaniu funkcji `n_znak()`.

Zauważ, że składnia ANSI C wymaga, aby każda zmienna była poprzedzona nazwą swojego typu. W odróżnieniu od zwykłej deklaracji nie wolno stosować list zmiennych należących do tego samego typu:

```
void ping(int x, y, z)          /* nieprawidlowy naglowek funkcji */
void pong(int x, int y, int z)  /* prawidlowy naglowek funkcji    */
```

Standard ANSI C dopuszcza również starszą formę, ale uznaje ją za przestarzałą:

```
void n_znak(ch, num)
char ch;
int num;
```

Nawiasy zawierają tu jedynie listę nazw argumentów — ich typy deklarowane są poniżej. Zwróć uwagę, że deklaracja argumentów znajduje się przed klamrą otwierającą, a deklaracja zwykłych zmiennych lokalnych — po niej. Ta odmiana definicji funkcji, w przeciwieństwie do odmiany ANSI C, pozwala korzystać z list zmiennych należących do tego samego typu (nazwy zmiennych rozdzielone są przecinkami):

```
void ping(x, y, z)
int x, y, z;          /* prawidlowe */
```

Powyższa postać definicji funkcji wychodzi z użycia. Powinieneś o niej wiedzieć, abyś był w stanie zrozumieć starszy kod, ale pisząc nowe programy, powinieneś trzymać się składni zgodnej z ANSI C (w C99 i C11 również pojawią się ostrzeżenia o składni przeznaczonej do wycofania).

Funkcja `n_znak()` przyjmuje dane z funkcji `main()`, ale nie zwraca żadnej wartości. Dlatego też należy ona do typu `void`.

Zobaczmy teraz, jak z niej korzystać.

Prototyp funkcji pobierającej argumenty

Funkcję `n_znak()` zadeklarowaliśmy z wykorzystaniem następującego prototypu ANSI C:

```
void n_znak(char ch, int num);
```

Prototyp określa liczbę i typy argumentów przyjmowanych przez funkcję. Argumenty rozdzielone są przecinkami. Nazwy zmiennych w prototypie mogą zostać pominięte:

```
void n_znak(char, int);
```

Użycie konstrukcji `char ch` i `int num` w prototypie nie powoduje bowiem utworzenia żadnych zmiennych.

Tak jak poprzednio, standard ANSI C zezwala na korzystanie ze starszej formy deklaracji niezawierającej listy argumentów:

```
void n_znak();
```

Postać bez listy argumentów w przyszłości zostanie wycofana ze standardu C; ale nawet gdyby się tak nie stało, stosowanie pełnego prototypu jest o wiele lepszym rozwiązaniem, o czym wkrótce się przekonasz. Wersję bez listy argumentów warto znać głównie na ewentualność pracy z jakimś mocno leciwym kodem.

Wywoływanie funkcji pobierającej argumenty — argumenty faktyczne

Zmienne `ch` i `num` otrzymują swoje wartości dzięki użyciu w wywołaniu funkcji **argumentów faktycznych**. Przyjrzyj się pierwszemu wywołaniu funkcji `n_znak()` w naszym programie:

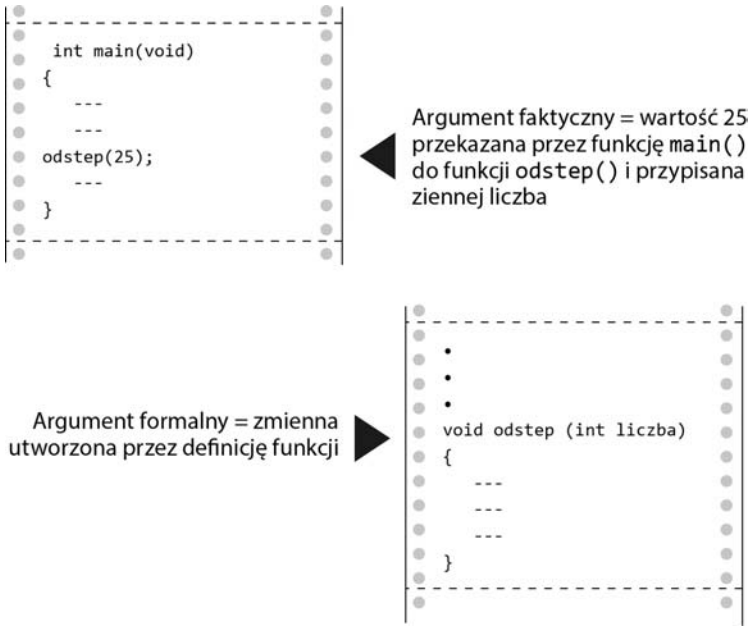
```
n_znak(ODSTEP, 12);
```

Argumentami faktycznymi są tu znak odstępu oraz liczba `12`. Wartości te zostają przypisane odpowiadającym im argumentom formalnym funkcji `n_znak()`, czyli zmiennym `ch` i `num`. Mówiąc w skrócie, argument formalny jest zmienną w wywoływanej funkcji, a argument faktyczny — konkretną wartością przypisaną tej zmiennej przez funkcję wywołującą. Jak pokazaliśmy w naszym przykładzie, argument faktyczny może być stałą, zmienną lub wyrażeniem. W każdym przypadku jest on obliczany, a jego wartość zostaje skopiowana do argumentu formalnego funkcji. Zastanów się na przykład nad następującym wywołaniem funkcji `n_znak()` w programie *naglowek2.c*:

```
n_znak(ODSTEP, (LIMIT – strlen(MIEJSCOWOSC)) / 2);
```

Długie wyrażenie stanowiące drugi argument faktyczny zostaje obliczone, a jego wartość (`10`) zostaje przypisana zmiennej `num`. Wywoływanej funkcji nie interesuje, czy liczba `10` pochodzi ze stałej, zmiennej czy też ze skomplikowanego wyrażenia. Powtórzmy: argument faktyczny jest konkretną wartością przypisywaną zmiennej — argumentowi formalnemu (patrz rysunek 9.3). Argumenty formalne są kopiami danych z funkcji wywołującej, a więc wszelkie modyfikacje, jakich dokonuje na nich funkcja wywoływana, nie pociągają za sobą zmiany danych wyjściowych.

RYSUNEK 9.3. Argumenty formalne i faktyczne



Argumenty faktyczne a argumenty formalne

Argument faktyczny to wyrażenie podane w nawiasach wywoływanej funkcji. Argument formalny to zmienna zadeklarowana w nagłówku definicji funkcji. W chwili wywoływania funkcji zmienne zadeklarowane jako argumenty formalne są tworzone, a następnie inicjowane wartościami wynikającymi z przetworzenia argumentów faktycznych. Na listingu 9.2 `'*'` i `LIMIT` są argumentami faktycznymi podczas pierwszego wywołania funkcji `n_znak()`, natomiast stałe `ODSTEP` i `12` w czasie drugiego wywołania. Zmienne `ch` i `num` są argumentami formalnymi w definicji funkcji.

Punkt widzenia czarnej skrzynki

Przyjmując perspektywę czarnej skrzynki, na wejściu funkcji `n_znak()` mamy znak oraz liczbę razy, jaką należy go wyświetlić. Dane wejściowe są przekazywane do funkcji w postaci argumentów. Te informacje w pełni wystarczają do korzystania z funkcji `n_znak()`, mogą też stanowić podstawę do jej implementacji.

Tym, co umożliwia przyjęcie punktu widzenia czarnej skrzynki, jest fakt, iż `ch`, `num` i `licznik` są zmiennymi lokalnymi, prywatnymi dla funkcji `n_znak()`. Gdybyś miał użyć zmiennych o tej samej nazwie w funkcji `main()`, byłyby one oddzielnymi, niezależnymi obiektami. Gdyby więc funkcja `main()` zawierała zmienną `licznik`, zmiana jej wartości nie powodowałaby modyfikacji zmiennej `licznik` w funkcji `n_znak()`, i odwrotnie. To, co dzieje się w środku czarnej skrzynki, jest niewidoczne dla funkcji wywołującej.

Zwracanie wartości z wykorzystaniem instrukcji return

Wiesz już, w jaki sposób przekazywać dane z funkcji wywołującej do wywoływanej. Do przesyłania informacji w przeciwnym kierunku służy wartości zwracana funkcji. Aby odświeżyć Twoją pamięć, skonstruujemy funkcję, która zwraca mniejszy z dwóch argumentów. Nazwiemy ją `imin()`, ponieważ obsługuje ona wartości typu `int`. Utworzymy również prostą funkcję `main()`, której jedynym zadaniem będzie sprawdzenie, czy funkcja `imin()` działa prawidłowo. Częstą praktyką poprzedzającą wykorzystanie funkcji w „prawdziwym" programie, na potrzeby którego została ona napisana, jest napisanie prostego programu testującego. Przykład takiego podejścia ilustruje listing 9.2 z funkcją `imin()` i możliwie prostym programem wywołującym funkcję.

LISTING 9.3. Program mniejsze.c

```
/* mniejsze.c -- znajduje mniejsze zlo */
#include <stdio.h>
int imin(int, int);
int main(void)
{
  int zlo1, zlo2;
  printf("Podaj dwie liczby calkowite (q konczy program):\n");
  while (scanf("%d %d", &zlo1, &zlo2) == 2)
  {
     printf("Mniejsza liczba sposrod %d i %d jest %d.\n",
            zlo1, zlo2, imin(zlo1,zlo2));
     printf("Podaj dwie liczby calkowite (q konczy program):\n");
  }
  printf("Gotowe.\n");

  return 0;
}
int imin(int n,int m)
{
  int min;
  if (n < m)
     min = n;
  else
     min = m;
  return min;
}
```

Jak pamiętamy, funkcja `scanf()` zwraca liczbę skutecznie odczytanych elementów, więc każde wejście inne niż para liczb całkowitych spowoduje przerwanie wykonywania pętli. Oto wynik działania programu:

```
Podaj dwie liczby calkowite (q konczy program):
509 333
Mniejsza liczba sposrod 509 i 333 jest 333.
Podaj dwie liczby calkowite (q konczy program):
-9393 6
Mniejsza liczba sposrod -9393 i 6 jest -9393.
Podaj dwie liczby calkowite (q konczy program):
q
Gotowe
```

Słowo kluczowe `return` sprawia, że wartość następującego po nim wyrażenia staje się wartością zwracaną funkcji. W tym przypadku funkcja zwraca wartość zmiennej `min`. Ponieważ zmienna `min` należy do typu `int`, do tego typu należy również funkcja `imin()`.

Zmienna `min` jest wprawdzie prywatna dla funkcji `imin()`, ale jej wartość jest przekazywana do funkcji wywołującej za pośrednictwem słowa `return`. Efektem poniższej instrukcji jest więc nadanie zmiennej `mniejszej` wartości zmiennej `min`:

```
mniejsze = imin(n,m);
```

Czy moglibyśmy użyć zamiast tego następującego kodu?

```
imin(n,m);
mniejsze = min;
```

Nie, ponieważ funkcja `main()` nie ma pojęcia o istnieniu czegoś takiego jak zmienna `min`. Pamiętaj, że zmienne lokalne funkcji `imin()` są znane tylko funkcji `imin()`. Wywołanie `imin(zlo1,zlo2)` kopiuje wartości zestawu zmiennych funkcji `main()` (czyli `zlo1` i `zlo2`) do zestawu zmiennych funkcji `imin()` (czyli `n` i `m`).

Zwrócona wartość może zostać nie tylko przypisana zmiennej, ale także użyta w wyrażeniu. Prawidłowe są na przykład poniższe instrukcje:

```
odpowiedz = 2 * imin(z, zstar) + 25;
printf("%d\n", imin(-32 + odpowiedz, LIMIT));
```

Wartość zwracana funkcji może być dostarczona przez dowolne wyrażenie, nie tylko zwykłą zmienną. Funkcję `imin()` można na przykład skrócić w następujący sposób:

```
/* funkcja wartosci minimalnej, druga wersja */
imin(int n,int m)
{
     return (n < m) ? n : m;
}
```

Wyrażenie warunkowe otrzymuje wartość `n` lub `m` (w zależności od tego, który z dwóch argumentów jest mniejszy) i wartość ta zostaje zwrócona do funkcji wywołującej. Jeśli chcesz, dla zwiększenia czytelności wartość zwracaną możesz ująć w nawias — nie jest to jednak wymagane.

Co się dzieje, gdy funkcja zwraca typ inny niż zadeklarowano?

```
int co_gdy(int n)
{
    double z = 100.0 / (double) n;
    return z;  // co sie stanie?
}
```

Wówczas zwracana liczba jest wartością, jaką otrzymasz, gdy przypiszesz otrzymaną wartość do zmiennej typu, który ma być zwracany. Tak więc w powyższym przykładzie końcowy skutek będzie taki sam, jakbyś przypisał zmienną `z` do zmiennej typu `int` i taką wartość zwrócił. Przypuśćmy, że mamy następujące wywołanie funkcji:

```
wynik = co_gdy(64);
```

Zmiennej z przypisana zostanie wartość 1.5625. Instrukcja return zwróci jednak wynik w postaci liczby 1 typu int.

Użycie instrukcji return ma jeszcze jeden skutek. Powoduje ono zakończenie funkcji i przejście do kolejnej instrukcji w funkcji wywołującej. Ma to miejsce, nawet jeśli return nie jest ostatnią instrukcją w funkcji. Funkcję imin() można więc zapisać następująco:

```
/* funkcja wartosci minimalnej, trzecia wersja */
imin(int n,int m)
{
   if (n < m)
       return n;
   else
       return m;
}
```

Wielu programistów w C uważa, że lepiej jest użyć instrukcji return tylko raz na końcu funkcji, gdyż ułatwia to śledzenie przebiegu programu. Nie jest jednak wielkim grzechem użycie kilku instrukcji return w funkcji tak krótkiej, jak powyższa. Tak czy owak, z punktu widzenia użytkownika wszystkie trzy wersje funkcji imin() są identyczne, ponieważ wszystkie przyjmują te same dane wyjściowe i dają taki sam wynik. Różnią się tylko wewnętrzną budową. Nawet poniższa wersja działa tak samo:

```
/* funkcja wartosci minimalnej, czwarta wersja */
imin(int n,int m)
{
   if (n < m)
       return n;
   else
       return m;
   printf("Profesor Fleppard to kretyn.\n");
}
```

Instrukcje return sprawiają, że instrukcja printf() nie zostanie nigdy wykonana. Profesor Fleppard może przez całe życie wykorzystywać skompilowaną wersję funkcji imin() w swoich programach i nigdy nie dowiedzieć się, co o nim sądzi jego uczeń.

Możesz również użyć następującej instrukcji:

```
return;
```

Powoduje ona zakończenie funkcji i powrót do funkcji wywołującej. Ponieważ po słowie return nie znajduje się żadne wyrażenie, nie ma również wartości zwracanej — forma ta powinna być więc stosowana tylko w funkcji typu void.

Typy funkcji

Deklaracja funkcji musi zawierać jej typ. Funkcja powinna należeć do tego samego typu co zwracana przez nią wartość. Funkcja, która nie zwraca wartości, powinna należeć do typu void. Jeśli w deklaracji funkcji nie podano typu, język C zakłada, że

funkcja należy do typu `int` (to konwencja z wczesnego C, kiedy większość funkcji i tak zwracała `int`). Standard C99 porzucił jednak domniemanie niejawnego typu int dla funkcji.

Deklaracja typu jest częścią definicji funkcji. Pamiętaj, że odnosi się ona do wartości zwracanej, a nie do argumentów. Na przykład poniższy nagłówek definiuje funkcję, która przyjmuje dwa argumenty typu `int`, ale zwraca wartość typu `double`:

```
double klink(int a, int b)
```

Aby móc poprawnie korzystać z funkcji, program musi znać jej typ, zanim zostanie ona użyta po raz pierwszy. Można to osiągnąć przez umieszczenie pełnej definicji funkcji przed miejscem jej pierwszego wywołania. Takie rozwiązanie może jednak zmniejszyć czytelność programu, a ponadto nie można go zastosować w przypadku, gdy funkcja jest częścią biblioteki lub znajduje się w osobnym pliku. Stąd ogólnie przyjętą metodą informowania kompilatora o funkcjach jest ich deklarowanie. Na przykład funkcja `main()` w listingu 9.3 zawiera następujące wiersze:

```
#include <stdio.h>
int imin(int, int);
int main(void)
{
  int zlo1, zlo2;
```

Drugi wiersz ustala, że `imin` jest nazwą funkcji, która przyjmuje dwa argumenty typu `int` i zwraca wartość typu `int`. Dzięki tej informacji kompilator będzie wiedział, jak traktować funkcję `imin()`, gdy pojawi się ona w programie.

Do tej pory wszystkie deklaracje funkcji umieszczaliśmy poza funkcją, która z nich korzystała. Dozwolone jest jednak umieszczenie ich wewnątrz funkcji. Na przykład początek programu *mniejsze.c* można by zmienić w następujący sposób:

```
#include <stdio.h>
int main(void)
{
  int imin(int, int);  /* deklaracja funkcji imin() */
  int zlo1, zlo2;
```

Niezależnie od wybranego zapisu istotne jest to, aby deklaracja znajdowała się przed pierwszym wywołaniem funkcji.

W standardowej bibliotece ANSI C funkcje pogrupowane są w rodziny, z których każda posiada swój plik nagłówkowy. Plik nagłówkowy zawiera między innymi deklaracje funkcji. Na przykład plik *stdio.h* zawiera deklaracje standardowych funkcji wejścia-wyjścia, takich jak `printf()` i `scanf()`, a *math.h* — deklaracje funkcji matematycznych. Plik *math.h* zawiera na przykład następujący wiersz:

```
double sqrt(double);
```

Informuje on kompilator, że funkcja `sqrt()` posiada parametr typu `double` i zwraca wartość typu `double`. Deklaracji funkcji nie należy mylić z definicją. Deklaracja określa,

do jakiego typu należy funkcja, ale kod funkcji znajduje się w definicji. Dołączenie pliku *math.h* informuje jedynie kompilator, że funkcja `sqrt()` zwraca typ `double`; kod funkcji `sqrt()` znajduje się w oddzielnym pliku bibliotecznym.

Prototypy ANSI C

Tradycyjna forma deklaracji funkcji sprzed czasów ANSI C była niekompletna, ponieważ określała jedynie typ wartości zwracanej, milcząc na temat argumentów. Zobaczmy, jakie problemy mogą wyniknąć z korzystania z tej postaci deklaracji.

Poniższa deklaracja stwierdza, że funkcja `imin()` zwraca wartość typu `int`:

```
int imin();
```

Nie mówi ona jednak nic o liczbie i typach przyjmowanych przez nią argumentów. Stąd jeśli wywołasz funkcję `imin()`, przekazując jej niewłaściwe argumenty, kompilator nie zorientuje się, że popełniasz błąd.

Problem

Przyjrzyjmy się kilku przykładom użycia funkcji `imax()`, bliskiej krewnej `imin()`. Listing 9.4 przedstawia program, który deklaruje funkcję `imax()`, a następnie używa jej w nieprawidłowy sposób.

LISTING 9.4. Program blad.c

```
/* blad.c -- korzysta z funkcji w niewlasciwy sposob */
#include <stdio.h>
int imax();      /* deklaracja w starym stylu */
int main(void)
{
  printf("Wieksza liczba z %d i %d jest %d.\n",
       3, 5, imax(3));
  printf("Wieksza liczba z %d i %d jest %d.\n",
       3, 5, imax(3.0, 5.0));
  return 0;
}
int imax(n, m)
int n, m;
{
  return (n > m ? n : m);
}
```

Pierwsze wywołanie funkcji `imax()` pomija jeden z argumentów, a drugie przekazuje wartości zmiennoprzecinkowe zamiast całkowitych. Pomimo tych błędów program kompiluje się i uruchamia. Oto przykładowe uruchomienie po użyciu kompilatora Xcode 4.6:

```
Wieksza liczba z 3 i 5 jest 1606416656.
Wieksza liczba z 3 i 5 jest 3886.
```

Uruchomienie po skompilowaniu przez GCC dało wartości 1359379472 oraz 1359377160. Kompilatory zadziałały prawidłowo; padły jedynie ofiarą braku prototypów funkcji.

O co chodzi? Szczegóły zależą od systemu, ale oto co dzieje się na komputerach PC i VAX: funkcja wywołująca umieszcza argumenty w tymczasowym obszarze pamięci zwanym **stosem** (ang. *stack*), skąd pobiera je funkcja wywoływana. Oba procesy *nie* są ze sobą skoordynowane. Funkcja wywołująca określa typ przekazywanej wartości w oparciu o typy argumentów faktycznych, natomiast funkcja wywoływana odczytuje wartości, kierując się typami argumentów formalnych. Tym samym, choć wywołanie `imax(3)` umieszcza na stosie tylko *jedną* liczbę całkowitą, to funkcja `imax()` pobiera ze stosu *dwie* takie liczby. Pierwsza odczytana wartość jest przekazanym argumentem, a druga — czymkolwiek, co znajdowało się na stosie.

W drugim wywołaniu funkcja `imax()` otrzymuje dwa argumenty typu `float`. Jest to równoznaczne z umieszczeniem na stosie dwóch wartości typu `double`. (Jak pamiętasz, wartości typu `float` są awansowane do `double`, gdy są przekazywane jako argumenty). W naszym systemie typ `double` ma 64 bity, a więc na stosie znalazło się 128 bitów danych. Odczytując ze stosu dwie wartości typu `int`, funkcja `imax()` pobrała 64 bity, ponieważ na naszym komputerze typ `int` mieści 32 bity. Bity te zupełnie przypadkiem odpowiadały dwóm liczbom całkowitym, z których większa wynosiła 3886.

ANSI C na ratunek!

Proponowanym przez standard ANSI C rozwiązaniem problemu niedopasowanych argumentów jest uzupełnienie deklaracji funkcji o typy zmiennych. Rezultatem jest **prototyp funkcji** — deklaracja, która określa typ wartości zwracanej oraz liczbę i typy argumentów. Aby zasygnalizować, że funkcja `imax()` wymaga dwóch argumentów typu `int`, można skorzystać z każdego z poniższych prototypów:

```
int imax(int, int);
int imax(int a, int b);
```

Pierwsza postać zawiera listę typów; druga wzbogaca ją o nazwy zmiennych. Pamiętaj, że nazwy te są w zasadzie atrapami i nie muszą odpowiadać nazwom użytym w definicji funkcji.

Mając te informacje, kompilator może sprawdzić, czy wywołanie funkcji pasuje do jej prototypu. Czy przekazywana jest właściwa liczba argumentów? Czy należą one do właściwych typów? Gdy typy w wywołaniu i w prototypie nie zgadzają się i obydwa są liczbami lub wskaźnikami, kompilator dokonuje rzutowania, które dostosowuje argumenty faktyczne do typu argumentów formalnych. Na przykład `imax(3.0, 5.0)` zostaje zamienione na `imax(3, 5)`. Listing 9.5 jest wynikiem rozszerzenia listingu 9.4 o prototypy funkcji.

LISTING 9.5. Program proto1.c

```
/* proto1.c -- wykorzystuje prototyp funkcji */
#include <stdio.h>
int imax(int, int);      /* prototyp */
```

```
int main(void)
{
  printf("Wieksza liczba z %d i %d jest %d.\n",
       3, 5, imax(3));
  printf("Wieksza liczba z %d i %d jest %d.\n",
       3, 5, imax(3.0, 5.0));
  return 0;
}
int imax(n, m)
{
  return (n > m ? n : m);
}
```

Gdy spróbowaliśmy skompilować listing 9.5, nasz kompilator wyświetlił komunikat o tym, że wywołanie funkcji `imax()` zawiera za mało parametrów.

Co z błędami typów? Aby to sprawdzić, wywołanie `imax(3)` zamieniliśmy na `imax(3, 5)` i ponownie dokonaliśmy próby kompilacji. Tym razem nie było komunikatów o błędach, a po uruchomieniu programu uzyskaliśmy następujący wynik:

```
Wieksza liczba z 3 i 5 jest 5.
Wieksza liczba z 3 i 5 jest 5.
```

Wartości `3.0` i `5.0` w drugim wywołaniu zostały zgodnie z obietnicą przetworzone na `3` i `5`, aby funkcja mogła je prawidłowo obsłużyć.

Choć nie wyświetlił się komunikat o błędzie, kompilator zgłosił jednak ostrzeżenie o konwersji wartości typu `double` na `int` i możliwej utracie danych. Na przykład wywołanie funkcji:

```
imax(3.9, 5.4);
```

staje się równoważne wywołaniu:

```
imax(3, 5);
```

Różnica pomiędzy błędem a ostrzeżeniem polega na tym, że błąd przerywa kompilację, a ostrzeżenie nie. Niektóre kompilatory przeprowadzają rzutowanie typów, nie informując Cię nawet o tym. Dzieje się tak, ponieważ standard nie wymaga stosowania ostrzeżeń. Jednak wiele kompilatorów pozwala na ustawienie poziomu ostrzegania, co umożliwia dostosowanie szczegółowości wydawanych przez kompilator informacji.

Brak argumentów a argumenty nieokreślone

Załóżmy, że użyjesz następującego prototypu:

```
void wyswietl_imie();
```

Kompilator zgodny z ANSI C przyjmie, że po prostu zdecydowałeś się nie korzystać z prototypu, i nie będzie sprawdzał poprawności argumentów w wywołaniach funkcji. Aby zasygnalizować, że funkcja nie pobiera żadnych argumentów, w nawiasie należy umieścić słowo kluczowe `void`:

```
void wyswietl_imie(void);
```

ANSI C zinterpretuje powyższe wyrażenie jako informację, że funkcja `wyswietl_imie()` nie przyjmuje argumentów, i dopilnuje, aby wywołania funkcji rzeczywiście nie zawierały żadnych parametrów.

Kilka funkcji, takich jak `printf()` i `scanf()`, przyjmuje zmienną liczbę argumentów. Na przykład w funkcji `printf()` pierwszy argument jest łańcuchem, ale liczba i typy pozostałych argumentów są nieznane. W takich przypadkach ANSI C dopuszcza stosowanie prototypów częściowych. Funkcja `printf()` mogłaby mieć na przykład następujący prototyp:

```
int printf(const char *, ...);
```

Prototyp ten ustala, że pierwszy argument jest łańcuchem (wyjaśnia to rozdział 11., „Łańcuchy znakowe i funkcje łańcuchowe") oraz że funkcja może przyjmować dalsze argumenty o dowolnych typach.

Biblioteka C w pliku nagłówkowym *stdarg.h* udostępnia standardowy sposób definiowania funkcji o zmiennej liczbie argumentów. Rozdział 16. zawiera więcej informacji na ten temat.

Potęga prototypów

Prototypy stanowią mocną stronę języka. Pozwalają kompilatorowi wyłapywać błędy lub przeoczenia, jakich możesz się dopuścić podczas używania funkcji. Tego rodzaju problemy, jeśli zostaną przeoczone, mogą być trudne do wykrycia. Czy musisz używać prototypów? Nie, możesz zamiast nich stosować stary sposób deklarowania funkcji (bez podawania argumentów), ale nie wiążą się z tym żadne korzyści, natomiast może wyniknąć wiele kłopotów.

Istnieje sposób, aby nie używając prototypu, zachować jego zalety. Celem stosowania prototypów jest poinformowanie kompilatora, w jaki sposób powinno się używać funkcji, zanim natrafi on na jej pierwsze zastosowanie. Ten sam efekt możesz uzyskać, umieszczając definicję funkcji przed jej pierwszym użyciem. W ten sposób definicja działa jak swój własny prototyp. Postępowanie takie stosuje się zwykle w przypadku krótkich funkcji:

```
// mamy tu i definicje, i prototyp
int imax(int a, int b) { return a >b ? a : b;}
int main()
{
  int x,z;
  …
  z = imax(x, 50);
  …
}
```

Rekurencja

Język C pozwala, aby funkcja wywoływała samą siebie. Proces ten nosi nazwę **rekurencji** (ang. *recursion*). Rekurencja jest narzędziem przydatnym, ale niekiedy trudnym w użyciu. Najwięcej problemów sprawia zakończenie rekurencji z uwagi na to, że funkcja, która wywołuje samą siebie, robi to bez końca, jeśli nie zawiera odpowiednio sformułowanej instrukcji warunkowej.

Rekurencja może być stosowana zamiast pętli. Czasami pętla stanowi rozwiązanie czytelniejsze, a czasami jednak przejrzystsza jest rekurencja. Implementacje rekurencyjne są bardzo często implementacjami eleganckimi, ale niekoniecznie bardziej wydajnymi niż pętle.

Rekurencja bez tajemnic

Aby zobaczyć, o co chodzi, przyjrzyjmy się przykładowi. Funkcja `main()` w listingu 9.6 wywołuje funkcję `gora_i_dol()`. Będziemy to nazywać „pierwszym poziomem rekurencji". Następnie funkcja `gora_i_dol()` wywołuje sama siebie, co będziemy określać „drugim poziomem rekurencji". Drugi poziom rekurencji wywołuje trzeci poziom itd. Poniższy przykład dochodzi do czwartego poziomu. Aby umożliwić lepszy wgląd w swoje działanie, oprócz wyświetlenia wartości `n` program podaje również adres `&n`, pod którym przechowywana jest zmienna `n`. (W dalszej części rozdziału omówiono dokładniej zagadnienie wskaźników. Funkcja `printf()` do wyświetlania wskaźników używa specyfikatora `%p`; jeśli dany system nie obsługuje tego specyfikatora, można użyć `%u` lub `%lu`).

LISTING 9.6. Program rekur.c

```
/* rekur.c -- ilustracja rekurencji */
#include <stdio.h>
void gora_i_dol(int);
int main(void)
{
  gora_i_dol(1);
  return 0;
}
void gora_i_dol(int n)
{
  printf("Poziom: %d: adres zmiennej n: %p\n", n, &n);
  if (n < 4)
       gora_i_dol(n+1);
  printf("POZIOM %d: adres zmiennej n: %p\n", n, &n);
}
```

Dane wyjściowe wyglądają tak:

```
Poziom 1: adres zmiennej n: 0x0012ff48
Poziom 2: adres zmiennej n: 0x0012ff3c
Poziom 3: adres zmiennej n: 0x0012ff30
Poziom 4: adres zmiennej n: 0x0012ff24
```

```
POZIOM 4: adres zmiennej n: 0x0012ff24
POZIOM 3: adres zmiennej n: 0x0012ff30
POZIOM 2: adres zmiennej n: 0x0012ff3c
POZIOM 1: adres zmiennej n: 0x0012ff48
```

Prześledźmy program, aby zobaczyć, jak działa rekurencja. Na początku funkcja `main()` wywołuje funkcję `gora_i_dol()`, przekazując jej argument `1`. W wyniku tego argument formalny `n` otrzymuje wartość `1`, co powoduje wyświetlenie tekstu `Poziom 1` przez pierwszą instrukcję `printf()`. Następnie, ponieważ `n` jest mniejsze od `4`, funkcja `gora_i_dol` (poziom 1) wywołuje funkcję `gora_i_dol` (poziom 2) z argumentem `n + 1`, czyli `2`. Powoduje to przypisanie zmiennej `n` na drugim poziomie wartości `2` oraz wyświetlenie przez pierwszą instrukcję `printf()` łańcucha `Poziom 2`. W podobny sposób kolejne dwa wywołania prowadzą do wyświetlenia tekstów `Poziom 3` i `Poziom 4`.

Gdy osiągnięty zostaje poziom 4, zmienna `n` jest równa `4` i warunek instrukcji `if` nie jest spełniony. Funkcja `gora_i_dol()` nie zostaje ponownie wywołana. Zamiast tego funkcja poziomu 4 przechodzi do drugiej instrukcji pisania, która wyświetla tekst `POZIOM 4` (ponieważ `n` wynosi 4). Następnie program wykonuje instrukcję `return`. W tym momencie funkcja na poziomie 4 się kończy, a program wraca do funkcji, która ją wywołała, czyli funkcji `gora_i_dol()` poziomu 3. Ostatnią instrukcją wykonaną na poziomie 3 było wywołanie poziomu 4 w ramach instrukcji `if`. Poziom 3 wznawia więc działanie od kolejnej instrukcji, którą jest druga instrukcja `printf()`. Powoduje to wyświetlenie tekstu `POZIOM 3`. Następnie kończy się poziom 3, program wraca do poziomu 2, który wyświetla tekst `POZIOM 2` itd.

Zauważ, że każdy poziom rekurencji operuje na swojej prywatnej zmiennej `n`. Można to rozpoznać po różnicy w adresach kolejnych zmiennych (w różnych systemach będą miały one różne wartości albo nawet będą inaczej sformatowane; ważne, że adres na poziomie `Poziom 1` jest taki sam jak na poziomie `POZIOM 1` itd.).

Jeśli wydaje Ci się to trochę niejasne, wyobraź sobie sytuację, w której mamy ciąg wywołań funkcji — funkcja `fun1()` wywołuje `fun2()`, `fun2()` wywołuje `fun3()` i `fun3()` wywołuje `fun4()`. Gdy `fun4()` kończy działanie, program wraca do funkcji `fun3()`. Gdy `fun3()` kończy działanie, program wraca do `fun2()`. A gdy działanie kończy `fun2()`, program wraca do `fun1()`. Rekurencja działa tak samo, z tym że zamiast `fun1()`, `fun2()`, `fun3()` i `fun4()` mamy jedną i tę samą funkcję.

Podstawy rekurencji

Z początku rekurencja może Ci się wydawać zagmatwana, przyjrzyjmy się więc kilku podstawowym elementom, które pomogą Ci ją zrozumieć.

Po pierwsze, każdy poziom funkcji posiada swoje własne zmienne. Zmienna `n` na poziomie 1 jest inną zmienną niż `n` na poziomie 2 — program utworzył więc cztery niezależne zmienne o tej samej nazwie i różnych wartościach. Gdy program powrócił do pierwszego poziomu funkcji `gora_i_dol()`, zmienna `n` wciąż miała wartość `1`, którą otrzymała na początku (patrz rysunek 9.4).

RYSUNEK 9.4.
Zmienne w rekurencji

Po drugie, każdemu wywołaniu funkcji odpowiada jeden powrót. Po wykonaniu instrukcji `return` na końcu ostatniego poziomu rekurencji program przechodzi do poprzedniego poziomu rekurencji, nie zaś do funkcji `main()`. Aby dotrzeć do miejsca pierwotnego wywołania funkcji `gora_i_dol()` w `main()`, program musi przejść przez wszystkie kolejne poziomy rekurencji, wracając z każdego poziomu funkcji `gora_i_dol()` do poziomu, który go wywołał.

Po trzecie, instrukcje w funkcji rekurencyjnej, znajdujące się przed miejscem, w którym wywołuje ona samą siebie, wykonywane są w kolejności wywoływania poziomów. Na przykład na listingu 9.6 pierwsza instrukcja pisania znajduje się przed wywołaniem rekurencyjnym. Została ona wykonana czterokrotnie w kolejności wywołań: Poziom 1, Poziom 2, Poziom 3 i Poziom 4.

Po czwarte, instrukcje w funkcji rekurencyjnej, znajdujące się po miejscu, w którym wywołuje ona samą siebie, wykonywane są w kolejności odwrotnej do kolejności wywoływania poziomów. Na przykład po wywołaniu rekurencyjnym znajduje się druga instrukcja `printf()`. Została ona wykonana w następującej kolejności: Poziom 4, Poziom 3, Poziom 2, Poziom 1. Ta cecha rekurencji jest użyteczna w przypadku problemów programistycznych wymagających odwrócenia kolejności działań (wkrótce przedstawimy przykład).

Po piąte, chociaż każdy poziom rekurencji posiada swój własny zestaw zmiennych, kod funkcji nie jest zwielokrotniany. Kod funkcji jest ciągiem instrukcji, a wywołanie funkcji jest poleceniem nakazującym przejście do jego początku. Wywołanie rekurencyjne powoduje więc powrót programu do początku funkcji. Poza tym, że wywołania rekurencyjne tworzą nowe zmienne, w dużym stopniu przypominają one pętlę. W wielu przypadkach rekurencję i pętlę można stosować zamiennie.

Po szóste i ostatnie, niezwykle istotne jest, aby funkcja rekurencyjna posiadała element umożliwiający zakończenie sekwencji wywołań rekurencyjnych. Zazwyczaj funkcja rekurencyjna używa instrukcji warunkowej `if` albo jej odpowiednika, aby przerwać rekurencję, gdy argument funkcji osiągnie określoną wartość. Aby ten mechanizm zadziałał, każde wywołanie funkcji wymaga innej wartości argumentu.

W ostatnim z przykładów, funkcja `gora_i_dol(n)` wywołuje `gora_i_dol(n+1)`. W końcu argument faktyczny `n` osiąga wartość 4, powodując, że `if(n < 4)` zwraca false.

Rekurencja końcowa

W najprostszej postaci rekurencji wywołanie rekurencyjne znajduje się na końcu funkcji, tuż przed instrukcją `return`. Nazywamy to **rekurencją końcową** (ang. *tail recursion* lub *end recursion*). Jest to najprostsza forma rekurencji, ponieważ działa tak samo jak pętla.

Przyjrzyjmy się dwóm wersjom funkcji obliczającej silnię — jednej, wykorzystującej pętlę, i drugiej, stosującej rekurencję. **Silnia** liczby całkowitej jest iloczynem wszystkich liczb od 1 do tej liczby. Na przykład 3 silnia (zapisujemy: `3!`) to tyle, co `1*2*3`. `0!` ma z definicji wartość `1`; silnia nie jest określona dla liczb ujemnych. Listing 9.7 przedstawia funkcję obliczającą silnię za pomocą pętli `for`.

LISTING 9.7. Program silnia.c

```
// silnia.c — oblicza silnie za pomoca rekurencji i petli
#include <stdio.h>
long silnia(int n);
long rsilnia(int n);
int main(void)
{
    int num;
    printf("Ten program liczy silnie.\n");
    printf("Podaj liczbe z przedzialu 0-12 (k - koniec):\n");
    while (scanf("%d", &num) == 1)
    {
        if (num < 0)
            printf("Prosze nie podawac liczb ujemnych.\n");
        else if (num > 12)
            printf("Prosze podac wartosc mniejsza od 13.\n");
        else
        {
            printf("petla: %d silnia = %ld\n",
                    num, silnia(num));
            printf("rekurencja: %d silnia = %ld\n",
                    num, rsilnia(num));
        }
        printf("Podaj liczbe z przedzialu 0-12 (k - koniec):\n");
    }
    printf("Gotowe.\n");

    return 0;
}
long silnia(int n)     // wersja oparta na petlach
{
    long wyn;
    for (wyn = 1; n > 1; n--)
        wyn *= n;
```

```
    return wyn;
}
long rsilnia(int n)    // wersja rekurencyjna
{
    long wyn;
    if (n > 0)
        wyn = n * rsilnia(n-1);
    else
        wyn = 1;

    return wyn;
}
```

Program ogranicza dane wejściowe do liczb całkowitych z przedziału od 0 do 12. Wynika to z faktu, że `12!` ma wartość nieco poniżej pół miliarda, co oznacza, że liczba `13!` znacznie przekroczy zakres typu `long` na naszym komputerze. Aby wykroczyć poza 12!, musiałbyś użyć typu o większym zakresie, takiego jak `double` czy `long long`.

Oto przykład wykonania programu:

```
Ten program liczy silnie.
Podaj liczbe z przedzialu 0-12 (k - koniec):
5
petla: 5 silnia = 120
rekurencja: 5 silnia = 120
Podaj liczbe z przedzialu 0-12 (k - koniec):
10
petla: 10 silnia = 3628800
rekurencja: 10 silnia = 3628800
Podaj liczbe z przedzialu 0-12 (k - koniec):
k
Gotowe
```

Pętla nadaje zmiennej `odp` wartość początkową 1, a następnie mnoży ją kolejno przez liczby od `n` do 2. W zasadzie zgodnie z definicją silni należałoby wykonać jeszcze mnożenie przez 1, ale ten krok możemy pominąć, ponieważ nie wpływa on na wynik.

Spróbujmy stworzyć teraz wersję wykorzystującą rekurencję. Kluczową rolę odgrywa tu fakt, iż `n! = n * (n-1)!`. Jest to prawdą, ponieważ `(n-1)!` jest iloczynem wszystkich dodatnich liczb całkowitych aż do `n-1`; pomnożenie go przez `n` daje więc iloczyn wszystkich liczb od 1 do `n`. Jeśli zatem naszą funkcję nazwiemy `rsilnia()`, to `rsilnia(n)` jest równe `n * rsilnia(n-1)`. Wartość `rsilnia(n)` można więc obliczyć, korzystając z wywołania `rsilnia(n-1)`, jak pokazano na listingu 9.7. Rzecz jasna, w pewnym momencie ciąg wywołań rekurencyjnych musi się zakończyć — w tym celu wystarczy zwrócić wartość 1, gdy `n` jest równe 0.

Wersja rekurencyjna z listingu 9.7 daje w wyniku takie same dane wyjściowe, jak wersja poprzednia. Zauważ, że choć wywołanie rekurencyjne funkcji `rsilnia()` nie jest ostatnim wierszem w funkcji, jest ono ostatnią wykonywaną instrukcją dla `n > 0` — mamy więc do czynienia z rekurencją końcową.

Jeśli funkcję można napisać z wykorzystaniem zarówno pętli, jak i rekurencji, to której metody należy użyć? Zazwyczaj lepszym rozwiązaniem jest pętla. Po pierwsze, rekurencja zużywa więcej pamięci, ponieważ każdy poziom otrzymuje oddzielny zestaw zmiennych; ponadto każde wywołanie umieszcza na stosie nową porcję danych, a systemowe ograniczenia rozmiaru stosu programu mogą ograniczyć osiągalną głębokość rekurencji. Po drugie, rekurencja jest wolniejsza, ponieważ każde wywołanie funkcji zabiera czas. Po co więc pokazaliśmy powyższy przykład? Ponieważ rekurencja końcowa jest najłatwiejszą do zrozumienia postacią rekurencji. Rekurencja jest zaś czymś, co warto znać po prostu dlatego, że w niektórych przypadkach zastąpienie jej pętlą jest nieopłacalne.

Rekurencja i odwracanie kolejności działań

Przyjrzyjmy się teraz problemowi, w którym przydatna jest zdolność rekurencji do odwracania kolejności działań (co oznacza, że rekurencja rozwiązuje go prościej niż pętla). Zadanie jest następujące: należy napisać funkcję, która wyświetla dwójkowy (binarny) odpowiednik liczby całkowitej. System binarny przedstawia liczby za pomocą potęg dwójki. Tak jak 234 w systemie dziesiętnym oznacza $2\times10^2+3\times10^1+4\times10^0$, tak w systemie dwójkowym 101 oznacza $1\times2^2+0\times2^1+1\times2^0$. Liczby w systemie dwójkowym składają się wyłącznie z cyfr 0 i 1.

Potrzebujemy metody, *algorytmu*. W jaki sposób możemy znaleźć dwójkowy odpowiednik np. liczby 5? No cóż, liczby nieparzyste w systemie binarnym kończą się cyfrą 1, a parzyste — cyfrą 0, ostatnią cyfrę możemy więc obliczyć za pomocą wyrażenia `5 % 2`. Jeśli ma ono wartość 1, oznacza to, że 5 jest liczbą nieparzystą i kończy się cyfrą 1. Ogólnie rzecz biorąc, jeśli `n` jest liczbą, to jej ostatnią cyfrą w systemie dwójkowym jest `n % 2` — pierwsza cyfra, jaką możemy obliczyć, jest więc ostatnią cyfrą, którą należy wyświetlić. Sugeruje to użycie funkcji rekurencyjnej, w której `n % 2` byłoby obliczane przed wywołaniem rekurencyjnym, ale wyświetlane po tym wywołaniu. W ten sposób wartość obliczona jako pierwsza zostałaby wyświetlona na końcu.

Aby otrzymać kolejną cyfrę, dzielimy wyjściową liczbę przez 2. W systemie dwójkowym jest to równoważne usunięciu ostatniej cyfry (jeśli nie operujemy na liczbach zmiennoprzecinkowych). Jeśli otrzymany wynik jest parzysty, kolejną cyfrą postaci binarnej jest 0; w przeciwnym wypadku cyfrą tą jest 1. Na przykład 5/2 jest równe 2 (dzielenie całkowite!), a więc drugą cyfrą od końca jest 0. Na razie mamy więc 01. Następnie powtarzamy proces. Dzielimy 2 przez 2, otrzymując 1. Obliczenie `1 % 2` daje wynik 1 — kolejną cyfrą jest więc 1. Mamy już trzy cyfry: 101. Kiedy powinniśmy się zatrzymać? Kończymy pracę, gdy wynik dzielenia przez 2 jest mniejszy niż 2, ponieważ w przeciwnym wypadku została nam jeszcze przynajmniej jedna cyfra binarna. Każde dzielenie przez 2 odcina jedną cyfrę dwójkową, do momentu, gdy nie zostanie już nic. (Jeśli jest to dla Ciebie niejasne, spróbuj przeanalizować to samo w systemie dziesiętnym. Reszta z dzielenia 628 przez 10 jest równa 8, a więc ostatnią cyfrą jest 8. Dzielenie całkowite przez 10 daje wynik 62. Reszta z dzielenia 62 przez 10 wynosi 2, a więc następną cyfrą jest 2 itd.). Listing 9.8 stanowi implementację naszego algorytmu.

LISTING 9.8. Program binar.c

```
/* binar.c -- wyswietla liczbe calkowita w postaci dwojkowej */
#include <stdio.h>
void do_binar(unsigned long n);
int main(void)
{
  unsigned long liczba;
  printf("Podaj liczbe calkowita (q konczy program):\n");
  while (scanf("%ld", &liczba) == 1)
  {
     printf("Odpowiednik dwojkowy: ");
     do_binar(liczba);
     putchar('\n');
     printf("Podaj liczbe calkowita (q konczy program):\n");
  }
  printf("Gotowe.\n");

  return 0;
}
void do_binar(unsigned long n)   /* funkcja rekurencyjna */
{
  int r;
  r = n % 2;
  if (n >= 2)
     do_binar(n / 2);
  putchar(r == 0 ? '0' : '1');
  return;
}
```

Funkcja `do_binar()` powinna wypisać znak `'0'`, jeśli `r` ma wartość liczbową 0, albo znak `'1'`, jeśli `r` wynosi 1. Wyrażenie `r == 0 ? '0' : '1'` stanowi więc naszą zminimalizowaną metodę konwersji pomiędzy cyframi binarnymi a ich reprezentacją znakową.

Oto przykładowy przebieg działania programu:

```
Podaj liczbe calkowita (q konczy program):
9
Odpowiednik dwojkowy: 1001
Podaj liczbe calkowita (q konczy program):
255
Odpowiednik dwojkowy: 11111111
Podaj liczbe calkowita (q konczy program):
1024
Odpowiednik dwojkowy: 10000000000
Podaj liczbe calkowita (q konczy program):
q
Gotowe
```

Czy moglibyśmy zaimplementować ten algorytm znajdowania postaci dwójkowej bez korzystania z rekurencji? Tak. Jednak ze względu na to, że ostatnia cyfra jest w nim obliczana jako pierwsza, musielibyśmy zapisywać gdzieś kolejne cyfry (na przykład w tablicy) przed wyświetleniem wyniku. Przykład podejścia nierekurencyjnego jest przedstawiony w rozdziale 15., „Manipulowanie bitami”.

Za i przeciw rekurencji

Rekurencja ma swoje plusy i minusy. Do plusów można zaliczyć między innymi to, że rekurencja proponuje najprostsze rozwiązania pewnych problemów programistycznych. Za minus natomiast trzeba uznać to, że niektóre algorytmy rekurencyjne potrafią błyskawicznie wyczerpać zasoby pamięci komputera. Rekurencja może być również trudna do dokumentowania i późniejszych przeróbek. Spójrzmy na przykład, który ilustruje zarówno złe, jak i dobre aspekty rekurencji.

Ciąg Fibonacciego można zdefiniować następująco: pierwsza liczba Fibonacciego to 1, druga liczba to 1, każda następna jest sumą swoich dwóch poprzedniczek. Zatem pierwszych kilka elementów ciągu to: 1, 1, 2, 3, 5, 8, 13. Ciąg Fibonacciego jest jednym z ulubionych przez matematyków ciągów; poświęcono mu nawet specjalne czasopismo. Nie wnikajmy w to jednak tutaj. Zajmijmy się lepiej utworzeniem funkcji, która dla danej liczby całkowitej `n` zwraca wartość `n`-tej liczby z ciągu Fibonacciego.

Najpierw zalety rekurencji: rekurencja umożliwia proste definiowanie. Jeśli nazwiemy funkcję `Fibonacci(), Fibonacci(n)` powinno zwrócić `1`, jeśli `n` równa się `1` lub `2`, w przeciwnym razie funkcja zwraca sumę `Fibonacci(n-1) + Fibonacci(n-2)`:

```
unsigned long Fibonacci(unsigned n)
{
   if(n>2)
      return Fibonacci(n-1)+ Fibonacci(n-2);
   else
      return 1;
}
```

Rekurencyjna funkcja w języku C zaledwie przeformułowuje rekurencyjną definicję matematyczną. Funkcja używa **podwójnej rekurencji** (ang. *double recursion*) — tzn. funkcja wywołuje samą siebie dwa razy. A to powoduje, że musimy przejść do omawiania wad.

Aby uwidocznić naturę tego problemu, przypuśćmy, że wywołujesz funkcję `Fibonacci(40)`. Jest to pierwszy poziom rekurencji, w którym zmiennej `n` przydzielona zostaje pamięć. Następnie funkcja ponownie wywołuje `Fibonacci()`, tworząc dwie dodatkowe zmienne o nazwie `n` na drugim poziomie rekurencji. Każde z tych dwóch wywołań generuje dwa kolejne wywołania, wymagające czterech dodatkowych zmiennych `n` na trzecim poziomie rekurencji. Każdy poziom wymaga dwa razy więcej zmiennych niż poprzedni, liczba zmiennych rośnie więc wykładniczo! Jak widziałeś na przykładzie z ziarnami pszenicy w rozdziale 5., wzrost wykładniczy prowadzi błyskawicznie do wielkich wartości. W tym przypadku, wzrost wykładniczy szybko sprawi, że wymagana pamięć przekroczy dostępną, co spowoduje zawieszenie się programu.

Cóż, jest to może skrajny przykład, ale dobrze demonstruje konieczność zachowania ostrożności podczas używania rekurencji, szczególnie gdy liczy się wydajność.

Wszystkie funkcje języka C są równe

Każda funkcja języka C istnieje w programie na równych prawach. Każda może wywołać kolejną funkcję albo sama być wywołana przez inną. Sprawia to, że funkcje w języku C różnią się nieco od procedur w Pascalu czy Moduli-2, które mogą być zagnieżdżane w procedurach. Procedury w jednym zagnieżdżeniu nie mogą korzystać z procedur z innego zagnieżdżenia.

Czy funkcja `main()` nie jest tu wyjątkiem? Tak, jest trochę wyjątkowa, gdyż wykonanie składającego się z wielu funkcji programu rozpoczyna się od pierwszej instrukcji wewnątrz funkcji `main()`, ale na tym kończą się jej przywileje. Nawet `main()` może być wywoływana rekurencyjnie przez siebie lub też przez inne funkcje, choć zdarza się to raczej rzadko.

Kompilowanie programów zawierających więcej niż jedną funkcję

Najprostszym sposobem na korzystanie z wielu funkcji jest umieszczenie ich w jednym pliku. Kompilacja pliku źródłowego przebiega wówczas dokładnie tak samo jak w przypadku pliku zawierającego jedną funkcję. Inne warianty są w większym stopniu uzależnione od systemu, o czym przekonasz się po przeczytaniu poniższych podrozdziałów.

Unix

Przyjmujemy, że system Unix zawiera standardowy kompilator `cc` oraz że *plik1.c* i *plik2.c* są dwoma plikami zawierającymi funkcje języka C (klasyczny kompilator `cc` już raczej nie jest w użyciu, ale samo polecenie `cc` powinno być dostępne jako alias dla faktycznej implementacji kompilatora, na przykład `gcc` czy `clang`). Poniższe polecenie spowoduje skompilowanie obu plików i utworzenie pliku wykonywalnego o nazwie *a.out*:

```
cc plik1.c plik2.c
```

Dodatkowo tworzone są dwa pliki obiektowe *plik1.o* i *plik2.o*. Jeśli później wprowadzisz zmiany w pliku *plik1.c*, ale nie w *plik2.c*, będziesz mógł skompilować pierwszy plik i połączyć go ze skompilowaną wersją pliku drugiego za pomocą następującego polecenia:

```
cc plik1.c plik2.o
```

W Uniksie stosuje się też polecenie `make`, które automatyzuje kompilację programów składających się z wielu plików kodu źródłowego, ale jego omówienie to już temat na osobną książkę.

Pamiętajmy, że w systemie OS X program Terminal pozwala na pracę w powłoce uniksowej, ale programy kompilatorów (Clang lub GCC) trzeba zainstalować samodzielnie).

Linux

Zakładamy, że system Linux ma zainstalowany kompilator GNU C gcc. Przypuśćmy, że *plik1.c* i *plik2.c* są dwoma plikami zawierającymi funkcje języka C. Poniższe polecenie skompiluje obydwa pliki i utworzy plik wykonywalny o nazwie *a.out*:

```
gcc plik1.c plik2.c
```

Dodatkowo powstają dwa pliki obiektowe *plik1.o* i *plik2.o*. Jeśli później wprowadzisz zmiany w pliku *plik1.c*, ale nie w pliku *plik2.c*, możesz skompilować pierwszy plik i połączyć go ze skompilowaną wersją drugiego pliku, używając następującego polecenia:

```
gcc plik1.c plik2.o
```

DOS (kompilatory wiersza poleceń)

Większość kompilatorów wiersza poleceń dla systemu DOS działa podobnie jak polecenie `cc` w systemie Unix (różnica sprowadza się często do nazwy polecenia). Jedna z różnic polega na tym, że pliki obiektowe otrzymują w DOS-ie rozszerzenie *.obj*, a nie *.o*. Niektóre kompilatory zamiast plików kodu obiektowego tworzą pliki przejściowe w języku asemblera lub we własnym, specjalnym języku.

Środowiska IDE dla Windows i OS X

Zintegrowane środowiska programistyczne dla systemów Windows i OS X są *oparte na koncepcji projektu*. **Projekt** opisuje zasoby wykorzystywane przez dany program. Należą do nich także pliki z Twoim kodem źródłowym. Jeśli używałeś któregoś z tych kompilatorów, prawdopodobnie byłeś zmuszony tworzyć projekty nawet po to, aby uruchomić jednoplikowy program. W przypadku programów składających się z kilku plików musisz znaleźć w menu kompilatora polecenie, które pozwoli Ci dodać plik z kodem źródłowym do projektu. Powinieneś się upewnić, że wszystkie pliki z kodem źródłowym (z rozszerzeniem *.c*) zostały wymienione jako składniki projektu. W większości IDE plików nagłówkowych (tych z rozszerzeniem *.h*) nie dołącza się jawnie do listy plików projektu, a jedynie włącza się je do plików kodu źródłowego (*.c*) dyrektywą `#include`; wyjątkiem jest Xcode, gdzie pliki nagłówkowe włączane do programu stanowią część projektu.

Korzystanie z plików nagłówkowych

Jeśli funkcja `main()` znajduje się w jednym pliku, a definicje pozostałych funkcji — w drugim, pierwszy plik wciąż potrzebuje prototypów. Aby nie musieć wpisywać ich za każdym razem, gdy będziesz chciał skorzystać ze swoich funkcji, możesz umieścić je w pliku nagłówkowym. Tak właśnie postąpili autorzy standardowej biblioteki C — jak pamiętasz, prototypy funkcji we-wy znajdują się w pliku *stdio.h*, a prototypy funkcji matematycznych — w *math.h*. Analogicznie możesz postąpić ze swoimi funkcjami.

Ponadto programy często zawierają stałe zdefiniowane z wykorzystaniem preprocesora. Takie definicje obejmują jedynie plik, w którym znajdują się dyrektywy

`#define`. Jeśli wykorzystywane przez program funkcje umieścisz w kilku plikach, definicje stałych będziesz musiał udostępnić każdemu plikowi z osobna. Najbardziej bezpośrednim sposobem, aby to osiągnąć, jest przepisanie dyrektyw do każdego pliku — pochłania to jednak dużo czasu i zwiększa możliwość popełnienia błędu. Rozwiązanie to powoduje również powstanie poważnego problemu z modyfikacją programu: jeśli kiedyś zmienisz jedną ze stałych, będziesz musiał pamiętać, aby zrobić to we wszystkich plikach programu. Znacznie lepszym wyjściem jest umieszczenie dyrektyw `#define` w pliku nagłówkowym i użycie dyrektywy `#include` we wszystkich plikach źródłowych.

Umieszczanie prototypów funkcji i definicji stałych w pliku nagłówkowym świadczy o dobrej technice programowania. Przyjrzyjmy się przykładowi. Załóżmy, że jesteś zarządcą sieci czterech hoteli. Pokój w każdym z hoteli ma inną cenę, ale wszystkie pokoje w danym hotelu kosztują tyle samo. Jeśli gość zostaje na dłużej, za drugą noc płaci 95% ceny pierwszej nocy, za trzecią — 95% ceny drugiej nocy itd. (Pomińmy zagadnienie, czy taka polityka cenowa byłaby opłacalna). Potrzebujesz programu, który umożliwia wybór jednego z hoteli oraz liczby nocy i na podstawie tych danych oblicza całkowitą opłatę za pobyt. Program powinien wyświetlać menu, które pozwoli obliczać opłaty dopóty, dopóki nie zdecydujesz się zakończyć pracy.

Listingi 9.9, 9.10 i 9.11 składają się na jedną z możliwych wersji takiego programu. Pierwszy listing zawiera funkcję `main()`, która odzwierciedla ogólną organizację programu. Drugi listing zawiera wszystkie pozostałe funkcje. Listing 9.11 przedstawia plik nagłówkowy, przechowujący definicje stałych i prototypy funkcji dla wszystkich plików źródłowych. Jak być może pamiętasz, w środowiskach Unix i DOS cudzysłów w dyrektywie `#include "hotel.h"` wskazuje, że dołączany plik znajduje się w bieżącym katalogu roboczym (czyli zazwyczaj w katalogu zawierającym kod źródłowy programu; jeśli używasz IDE, musisz dowiedzieć się sam, czy i jak dołączać pliki nagłówkowe do projektu).

LISTING 9.9. Program oplaty.c

```
/* oplaty.c -- program obliczajacy oplate za pokoj */
/* kompiluj razem z listingiem 9.10               */
#include <stdio.h>
#include "hotel.h" /* definiuje stale, deklaruje funkcje */
int main(void)
{
   int noce;
   double hotel;
   int kod;
   while ((kod = menu()) != KONIEC)
   {
      switch(kod)
      {
      case 1 : hotel = HOTEL1;
               break;
      case 2 : hotel = HOTEL2;
               break;
```

```
            case 3 : hotel = HOTEL3;
                     break;
            case 4 : hotel = HOTEL4;
                     break;
            default: hotel = 0.0;
                     printf("Ups!\n");
                     break;
            }
            noce = pobierz_noce();
            pokaz_cene(hotel, noce);
        }
        printf("Dziekuje i do widzenia\n");
        return 0;
}
```

LISTING 9.10. Moduł wspomagający hotel.c

```
/* hotel.c -- funkcje dla zarzadzajacych hotelami */
#include <stdio.h>
#include "hotel.h"
int menu(void)
{
    int kod, stan;
    printf("\n%s%s\n", GWIAZDKI, GWIAZDKI);
    printf("Podaj numer hotelu:\n");
    printf("1) Marek Antoniusz          2) Olimpijski\n");
    printf("3) U Marynarza              4) Savoy\n");
    printf("5) koniec\n");
    printf("%s%s\n", GWIAZDKI, GWIAZDKI);
    while ((stan = scanf("%d", &kod)) != 1  ||
              (kod < 1 || kod > 5))
    {
        if (stan != 1)
           scanf("%*s"); // odrzucamy wejście nieliczbowe
        printf("Podaj liczbe z przedzialu od 1 do 5.\n");
    }
    return kod;
}
int pobierz_noce(void)
{
    int noce;
    printf("Ile nocy bedzie potrzebne? ");
    while (scanf("%d", &noce) != 1)
    {
        scanf("%*s"); // odrzucamy wejście nieliczbowe
        printf("Podaj liczbe calkowita, np. 2.\n");
    }
    return noce;
}
void pokaz_cene(double hotel, int noce)
{
    int n;
    double suma = 0.0;
    double przelicznik = 1.0;
```

```
    for (n = 1; n <= noce; n++, przelicznik *= RABAT)
        suma += hotel * przelicznik;
    printf("Calkowity koszt pobytu wyniesie %0.2f $.\n", suma);
}
```

LISTING 9.11. Plik nagłówkowy hotel.h

```
/* hotel.h -- stale i deklaracje dla hotel.c */
#define KONIEC       5
#define HOTEL1    80.00
#define HOTEL2   125.00
#define HOTEL3   155.00
#define HOTEL4   200.00
#define RABAT   0.95
#define GWIAZDKI "*********************************"
// pokazuje liste wyborow
int menu(void);
// zwraca zadana liczbe nocy
int pobierz_noce(void);
// oblicza cene na podstawie stawki i liczby noclegów
// i wyswietla wynik
void pokaz_cene(double hotel, int noce);
```

Oto przykładowy przebieg działania programu:

```
*********************************************************************
Podaj numer hotelu:
1) Marek Antoniusz          2) Olimpijski
3) U Marynarza              4) Savoy
5) koniec
*********************************************************************
3
Ile nocy bedzie potrzebne? 1
Calkowity koszt pobytu wyniesie 155.00 $.
*********************************************************************
Podaj numer hotelu:
1) Marek Antoniusz          2) Olimpijski
3) U Marynarza              4) Savoy
5) koniec
*********************************************************************
4
Ile nocy bedzie potrzebne? 3
Calkowity koszt pobytu wyniesie 570.50 $.
*********************************************************************
Podaj numer hotelu:
1) Marek Antoniusz          2) Olimpijski
3) U Marynarza              4) Savoy
5) koniec
*********************************************************************
5
Dziekuje i do widzenia
```

Oprócz wieloplikowej organizacji program zawiera kilka innych interesujących elementów. Funkcje `menu()` i `pobierz_noce()` pomijają dane nienumeryczne, testując wartość zwracaną `scanf()` i wykorzystując wywołanie `scanf("%*s")` w celu porzucenia łańcucha znakowego, jeśli takowy został wprowadzony. Przedstawiony niżej fragment funkcji `menu()` sprawdza równocześnie, czy dane są numeryczne oraz czy mieszczą się one w zadanych granicach:

```
while ((stan = scanf("%d", &kod)) != 1  ||
       (kod < 1 || kod > 5))
```

Kod ten wykorzystuje gwarancje, jakie daje język C: po pierwsze, wyrażenia logiczne obliczane są od lewej do prawej; po drugie, obliczanie ulega zatrzymaniu w momencie, gdy wartość wyrażenia staje się jasna. W tym przykładzie wartość zmiennej `kod` jest sprawdzana tylko wtedy, gdy funkcji `scanf()` udało się odczytać liczbę całkowitą.

Przydzielenie poszczególnych zadań różnym funkcjom sprzyja udoskonalaniu programu. Pierwsza wersja funkcji `menu()` lub `pobierz_noce()` mogłaby wykorzystywać proste wywołanie `scanf()` pozbawione elementów weryfikacji danych. Następnie, gdyby okazało się, że wersja podstawowa działa poprawnie, mógłbyś rozpocząć jej ulepszanie.

Uzyskiwanie adresów: operator &

Jednym z najważniejszych (i czasami najtrudniejszych) pojęć języka C jest **wskaźnik** (ang. *pointer*), czyli zmienna przechowująca adres w pamięci. W jednym z wcześniejszych rozdziałów stwierdziliśmy, że argumenty funkcji `scanf()` są adresami. Mówiąc ogólniej, każda funkcja, która modyfikuje dane w funkcji wywołującej bez użycia wartości zwracanej, wykorzystuje adresy. Nasze omówienie rozpoczniemy od prezentacji jednoargumentowego operatora & (zagadnienia używania i nadużywania wskaźników będą kontynuowane w następnym rozdziale).

Jednoargumentowy operator & pozwala uzyskać adres, pod którym przechowywana jest zmienna. Jeśli `ach` jest nazwą zmiennej, to `&ach` jest jej adresem. Adres możesz wyobrażać sobie jako miejsce w pamięci. Załóżmy, że mamy następującą instrukcję:

```
ach = 24;
```

a adresem, pod którym zapisana jest zmienna `ach` jest `0B76`. (Adresy na komputerach PC są często wyrażane w postaci wartości szesnastkowych). Wówczas instrukcja:

```
printf("%d %p\n", ach, &ach);
```

dałaby następujący wynik (`%p` jest specyfikatorem wyświetlającym adresy):

```
24 0B76
```

W listingu 9.12 wykorzystujemy operator &, aby sprawdzić, gdzie przechowywane są zmienne o tych samych nazwach należące do różnych funkcji.

LISTING 9.12. Program sprmiejs.c

```
/* sprmiejs.c -- sprawdza, gdzie przechowywane sa zmienne */
#include <stdio.h>
void mikado(int);                   /* deklaracja funkcji */
int main(void)
{
    int ach = 2, och = 5;           /* lokalne wzgledem main() */
    printf("W funkcji main() ach = %d, a &ach = %p\n", ach, &ach);
    printf("W funkcji main() och = %d, a &och = %p\n", och, &och);
    mikado(ach);
    return 0;
}
void mikado(int och)                 /* definicja funkcji */
{
    int ach = 10;                    /* lokalna wzgledem mikado() */
    printf("W funkcji mikado() ach = %d, a &ach = %p\n",
            ach, &ach);
    printf("W funkcji mikado() och = %d, a &och = %p\n",
            och, &och);
}
```

Powyższy program do wyświetlania adresów wykorzystuje specyfikator formatu `%p` ze standardu ANSI. Na naszym komputerze dane wyjściowe listingu 9.12 wyglądają następująco:

```
W funkcji main() ach = 2, a &ach = 0x7fff5fbff8e8
W funkcji main() och = 5, a &och = 0x7fff5fbff8e4
W funkcji mikado() ach = 10, a &ach = 0x7fff5fbff8b8
W funkcji mikado() och = 2, a &och = 0x7fff5fbff8bc
```

Sposób zapisu adresów przez specyfikator `%p` zależy od implementacji. Wiele z nich stosuje zapis szesnastkowy; tutaj, skoro każda cyfra szesnastkowa odpowiada 4 bitom, dwunastocyfrowe adresy szesnastkowe wskazują na adresowanie z użyciem 48 bitów.

O czym świadczą powyższe dane wyjściowe? Po pierwsze, obie zmienne `ach` mają różne adresy; to samo tyczy się zmiennych `och`. Oznacza to, że — jak wspomnieliśmy wcześniej — komputer traktuje je jako cztery niezależne zmienne. Po drugie, wywołanie `mikado(ach)` przekazało wartość (2) argumentu faktycznego (zmienna `ach` z funkcji `main()`) do argumentu formalnego (zmienna `och` z funkcji `mikado()`). Zauważ, że przekazana została tylko wartość. Obie zmienne (`ach` w `main()` i `och` w `mikado()`) pozostają niezależne.

Zwracamy uwagę na drugi fakt, ponieważ nie zachodzi on we wszystkich językach. Na przykład w języku FORTRAN podprogram wpływa na zmienne podprogramu, który go wywołał. Zmienna lokalna w podprogramie może mieć inną nazwę, ale jej adres jest zawsze taki sam jak zmiennej w podprogramie wywołującym. Język C działa inaczej. Każda funkcja posiada swoje własne zmienne. Jest to pożądane, ponieważ zapobiega zmianom pierwotnej zmiennej pod wpływem jakiegoś skutku ubocznego wywołanej funkcji — może jednak również sprawiać trudności, o czym przekonasz się w kolejnym podrozdziale.

Modyfikacja zmiennych w funkcji wywołującej

Czasami pożądane jest, aby funkcja dokonywała modyfikacji zmiennych należących do innej funkcji. Na przykład powszechną czynnością przy sortowaniu jest zamiana wartości dwóch zmiennych. Załóżmy, że mamy dwie zmienne o nazwach x i y i chcemy zamienić ich wartości. Najbardziej oczywiste rozwiązanie

```
x = y;
y = x;
```

nie działa, ponieważ zanim program wykona drugi wiersz, początkowa wartość zmiennej x zostanie zastąpiona początkową wartością zmiennej y. Potrzebny jest dodatkowy wiersz, który tymczasowo przechowa pierwotną wartość zmiennej x.

```
temp = x;
x = y;
y = temp;
```

Kod ten możemy umieścić w funkcji, a następnie napisać program, który ją przetestuje. Rezultatem będzie program podobny do listingu 9.13. Aby zaznaczyć, które zmienne należą do funkcji `main()`, a które do funkcji `zamiana()`, w pierwszej z nich użyliśmy oznaczeń x i y, a w drugiej — u i v.

LISTING 9.13. Program zamien1.c

```
/* zamien1.c -- pierwsza proba wykonania funkcji zamieniajacej */
#include <stdio.h>
void zamiana(int u, int v); /* deklaracja funkcji */
int main(void)
{
    int x = 5, y = 10;
    printf("Poczatkowo x = %d, a y = %d.\n", x, y);
    zamiana(x, y);
    printf("A teraz x = %d, a y = %d.\n", x, y);
    return 0;
}
void zamiana(int u, int v)  /* definicja funkcji */
{
   int temp;
   temp = u;
   u = v;
   v = temp;
}
```

Uruchomienie programu daje następujący wynik:

```
Poczatkowo x = 5, a y = 10.
A teraz x = 5, a y = 10.
```

Fatalnie! Wartości nie zostały zamienione! Aby zobaczyć, co poszło nie tak, dodajmy do funkcji `zamiana()` kilka instrukcji pisania (patrz listing 9.14).

LISTING 9.14. Program zamien2.c

```
/* zamien2.c -- badanie programu zamien1.c */
#include <stdio.h>
void zamiana(int u, int v);
int main(void)
{
    int x = 5, y = 10;
    printf("Poczatkowo x = %d, a y = %d.\n", x, y);
    zamiana(x, y);
    printf("A teraz x = %d, a y = %d.\n", x, y);
    return 0;
}
void zamiana(int u, int v)
{
   int temp;
   printf("Poczatkowo u = %d, a v = %d.\n", u, v);
   temp = u;
   u = v;
   v = temp;
   printf("A teraz u = %d, a v = %d.\n", u, v);
}
```

Oto nowe dane wyjściowe:

```
Poczatkowo x = 5, a y = 10.
Poczatkowo u = 5, a v = 10.
A teraz u = 10, a v = 5.
A teraz x = 5, a y = 10.
```

Jak widać, funkcja `zamiana()` działa prawidłowo — zamienia ona wartości zmiennych `u` i `v`. Problem dotyczy przekazania zmienionych wartości do funkcji `main()`. Jak zwracaliśmy uwagę wcześniej, funkcja `zamiana()` używa innych zmiennych niż `main()`, a więc zamiana wartości `u` i `v` nie ma absolutnie żadnego wpływu na `x` i `y`! Czy nie moglibyśmy w jakiś sposób wykorzystać instrukcji `return`? No cóż, moglibyśmy zakończyć funkcję `zamiana()` następującym wierszem:

```
return(u);
```

a następnie zmienić wywołanie w funkcji `main()` tak, jak pokazano poniżej:

```
x = zamiana(x,y);
```

Nadaje to wprawdzie zmiennej `x` nową wartość, ale zupełnie ignoruje zmienną `y`. Instrukcja `return` pozwala wysłać do funkcji wywołującej tylko jedną wartość — tutaj zaś potrzebujemy zwrócić dwie wartości. Jak to osiągnąć? Wystarczy skorzystać ze wskaźników.

Wskaźniki: pierwsze spojrzenie

Wskaźniki? Cóż to takiego? Zasadniczo **wskaźnik** (ang. *pointer*) jest zmienną (lub mówiąc ogólniej, obiektem danych), której wartość jest adresem. Tak jak wartością zmiennej typu `char` jest znak, a wartością zmiennej typu `int` — liczba całkowita, tak wartością zmiennej wskaźnikowej jest adres w pamięci. Jeśli zmienna wskaźnikowa nosi nazwę `wsk`, prawidłowa jest na przykład następująca instrukcja:

```
wsk = &ach;  // przypisuje zmiennej wsk adres zmiennej ach
```

Mówimy, że `wsk` „wskazuje na" `ach`. Różnica między `wsk` a `&ach` polega na tym, iż `wsk` jest zmienną, a `&ach` stałą (inaczej: `wsk` jest modyfikowalną l-wartością, a wyrażenie `&ach` jest r-wartością). Jeśli chcesz, możesz sprawić, aby zmienna `wsk` wskazywała gdzie indziej:

```
wsk = &och;  // powoduje, ze wsk wskazuje na och zamiast na ach
```

Teraz wartością `wsk` jest adres zmiennej `och`.

Aby utworzyć zmienną wskaźnikową, musisz zadeklarować jej typ. Załóżmy, że chcesz zadeklarować wskaźnik `wsk` tak, aby mógł on przechowywać adres zmiennej typu `int`. Aby to zrobić, musisz skorzystać z nowego operatora, przedstawionego w następnym podrozdziale.

Operator dereferencji: *

Załóżmy, że wiesz, że `wsk` wskazuje na `ach`:

```
wsk = &ach;
```

Wówczas możesz skorzystać z operatora **dereferencji** *, zwanego również operatorem **pośredniości**, (ang. *dereference*, *indirection*), aby znaleźć wartość przechowywaną przez zmienną `ach`. (Nie pomyl tego jednoargumentowego operatora z dwuargumentowym operatorem mnożenia — symbole są te same, ale działanie jest zasadniczo odmienne).

```
wart = *wsk;  // znajduje wartosc, na ktora wskazuje wsk
```

Powyższe dwie instrukcje (`wsk = &ach;` i `wart = *wsk;`) są równoważne następującej instrukcji:

```
wart = ach;
```

Użycie adresu i operatora dereferencji jest raczej pośrednim sposobem osiągnięcia tego rezultatu — stąd nazwa „operator pośredniości".

Deklarowanie wskaźników

Wiesz już, w jaki sposób deklarować zmienne typu `int` i innych typów podstawowych. Jak deklarujemy wskaźniki? Być może przypuszczasz, że deklaracja wygląda następująco:

```
pointer wsk;    // tak nie deklarujemy wskaznika!
```

Podsumowanie. Operatory związane ze wskaźnikami

Operator adresu:

&

Opis ogólny:

Operator & pozwala uzyskać adres zmiennej, która po nim następuje.

Przykład:

`&siostra` jest adresem zmiennej `siostra`.

Operator dereferencji (pośredniości):

*

Opis ogólny:

Operator * zwraca wartość przechowywaną pod adresem wskazywanym przez zmienną wskaźnikową.

Przykład:

```
siostra = 22;
wsk = &siostra;  // wskaźnik do siostra
wart = *wsk;     // przypisanie wart wartości spod adresu wsk
```

Efektem powyższych instrukcji jest przypisanie zmiennej `wart` wartości 22.

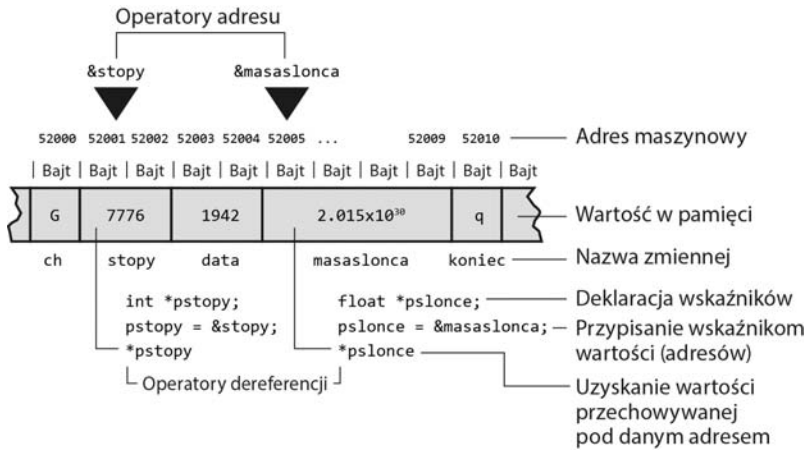
Dlaczego nie? Ponieważ nie wystarczy stwierdzić, że zmienna jest wskaźnikiem; należy również określić, na jaki typ zmiennej będzie ona wskazywała. Powodem tego jest fakt, iż zmienne różnych typów zajmują różne ilości pamięci, a niektóre operacje na wskaźnikach wymagają wiedzy o rozmiarze wskazywanej zmiennej. Ponadto program powinien wiedzieć, jaki rodzaj danych jest przechowywany pod określonym adresem. Typy `long` i `float` zajmują na niektórych systemach tyle samo pamięci, ale wyrażają liczby w zupełnie różny sposób. Oto jak powinna wyglądać prawidłowa deklaracja wskaźnika:

```
int * pi;         // pi jest wskaznikiem do zmiennej calkowitej
char * pc;        // pc jest wskaznikiem do zmiennej znakowej
float * pf, * pg; // pf i pg sa wskaznikami do zmiennych typu float
```

Słowa kluczowe `int`, `char` i `float` określają typ wskazywanej zmiennej, a gwiazdka (*) sygnalizuje, że deklarowana zmienna jest wskaźnikiem. Deklaracja `int * pi;` ustala więc, że `pi` jest wskaźnikiem oraz że `*pi` należy do typu `int` (patrz rysunek 9.5).

Odstęp pomiędzy symbolem * a nazwą wskaźnika nie jest wymagany. Wielu programistów używa odstępu w deklaracji, ale pomija go przy dereferencji zmiennej.

Wartość (`*pc`) zmiennej, na którą wskazuje `pc`, jest typu `char`. A czym jest sama zmienna `pc`? Nazywamy ją „wskaźnikiem do zmiennej typu `char`" lub krócej „wskaźnikiem do `char`". Jej wartość (adres) jest na większości systemów dodatnią liczbą całkowitą, nie należy jednak tych reprezentacji utożsamiać: na liczbach można wykonywać operacje, których nie można wykonywać na wskaźnikach (i odwrotnie).



RYSUNEK 9.5. Deklarowanie i korzystanie ze wskaźników

Na przykład dozwolone jest mnożenie dwóch liczb, ale nie można pomnożyć przez siebie dwóch wskaźników. Tak więc typ wskaźnikowy jest faktycznie istotnie różny od typu liczbowego. Dlatego też standard ANSI C definiuje dla wskaźników osobny specyfikator `%p`.

Wykorzystanie wskaźników do komunikacji pomiędzy funkcjami

Ponieważ naszym zamiarem jest rozwiązanie problemu z przekazywaniem danych między funkcjami, zaledwie dotknęliśmy tu powierzchni bogatego i fascynującego świata wskaźników. Listing 9.15 przedstawia program, który wykorzystuje wskaźniki w celu uzyskania poprawnego działania funkcji `zamien()`. Przyjrzyjmy mu się i spróbujmy zrozumieć, jak działa.

LISTING 9.15. Program zamien3.c

```
/* zamien3.c -- zamiana z wykorzystaniem wskaznikow */
#include <stdio.h>
void zamiana(int * u, int * v);
int main(void)
{
    int x = 5, y = 10;
    printf("Poczatkowo x = %d, a y = %d.\n", x, y);
    zamiana(&x, &y);  // wyslanie adresow do funkcji
    printf("A teraz x = %d, a y = %d.\n", x, y);
    return 0;
}
void zamiana(int * u, int * v)
{
   int temp;
   temp = *u;    // temp otrzymuje wartosc, na ktora wskazuje u
   *u = *v;
   *v = temp;
}
```

Czy listing 9.15 działa prawidłowo?

```
Poczatkowo x = 5, a y = 10.
A teraz x = 10, a y = 5.
```

Tak — tym razem wszystko jest w porządku.

Zobaczmy teraz, w jaki sposób działa nasz program. Wywołanie funkcji wygląda następująco:

```
zamiana(&x, &y);
```

Zamiast *wartości* zmiennych `x` i `y` przekazaliśmy ich *adresy*. Oznacza to, że wartościami argumentów formalnych `u` i `v` w funkcji `zamiana()` będą adresy. Zmienne `u` i `v` należy więc zadeklarować jako wskaźniki. Ponieważ `x` i `y` są liczbami całkowitymi, `u` i `v` powinny być wskaźnikami do zmiennych typu `int`. Deklaracja wygląda zatem następująco:

```
void zamiana(int * u, int * v)
```

Następnie w części głównej funkcji deklarujemy potrzebną nam zmienną tymczasową:

```
int temp;
```

Ponieważ chcemy nadać `temp` wartość zmiennej `x`, piszemy:

```
temp = *u;
```

Wskaźnik `u` ma bowiem wartość `&x` — wskazuje więc na zmienną `x`. Oznacza to, że `*u` jest wartością zmiennej `x`. Nie moglibyśmy napisać

```
temp = u;  /* ZLE */
```

ponieważ spowodowałoby to przypisanie zmiennej `temp` adresu — a nie wartości — zmiennej `x`.

Podobnie, aby przypisać wartość `y` zmiennej `x`, piszemy:

```
*u = *v;
```

co przekłada się na

```
x = y;
```

Podsumujmy działanie naszego przykładu. Chcieliśmy napisać funkcję zmieniającą wartości `x` i `y`. Przekazując funkcji `zamiana()` adresy `x` i `y`, daliśmy jej dostęp do tych zmiennych. Dzięki wskaźnikom i operatorowi * funkcja może odczytać wartości zapisane pod tymi adresami i dokonać ich zmiany.

W prototypie ANSI C nazwy zmiennych mogą zostać pominięte:

```
void zamiana(int *, int *);
```

Ogólnie rzecz biorąc, do funkcji możesz przekazać dwa rodzaje informacji o zmiennej. Jeśli użyjesz wywołania w postaci

```
funkcja1(x);
```

przekazujesz wartość x. Jeśli zaś skorzystasz z wywołania o postaci

```
funkcja2(&x);
```

przekazujesz adres x. Pierwsza postać wymaga, aby definicja funkcji zawierała argument formalny takiego samego typu jak zmienna x.

```
int funkcja1(int num)
```

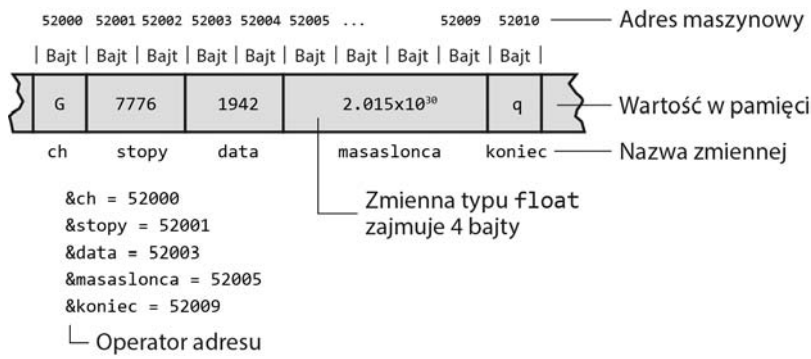
Druga postać wymaga, aby definicja funkcji zawierała argument formalny, będący wskaźnikiem do właściwego typu:

```
int funkcja2(int * wsk)
```

Użyj pierwszej postaci, jeśli funkcja potrzebuje wartości, aby wykonać jakieś obliczenia lub czynności. Użyj drugiej postaci, jeśli zadaniem funkcji jest modyfikacja zmiennych w funkcji wywołującej. Postać tę stosowaliśmy od początku, korzystając z funkcji `scanf()`. Jeśli chcemy na przykład pobrać wartość dla zmiennej `num`, używamy wywołania `scanf("%d", &num);`.

Wskaźniki pozwoliły nam obejść ograniczenia stwarzane przez fakt, iż zmienne funkcji `zamiana()` są lokalne. Dzięki nim mogliśmy sięgnąć do funkcji `main()` i zmodyfikować jej zmienne.

Użytkownicy języków Pascal i Modula-2 być może rozpoznają pierwszą postać jako odpowiednik parametru stałego, a drugą postać — jako odpowiednik parametru zmiennego. Programiści C++ poznają zmienne wskaźnikowe i zapytają, czy C też (jak C++) ma typy referencyjne (odpowiedź brzmi: nie). Z kolei użytkownicy języka BASIC mogą poczuć się nieco zakłopotani przedstawionymi tu technikami. Jeśli wskaźniki wydają Ci się dziwne i skomplikowane, możesz być pewny, że odrobina praktyki sprawi, iż przynajmniej niektóre z ich zastosowań staną się proste i wygodne (patrz rysunek 9.6).



RYSUNEK 9.6. Nazwy, adresy i wartości na komputerze adresowalnym bajtowo, takim jak PC

Zmienne — nazwy, adresy i wartości

Powyższe omówienie wskaźników obracało się wokół zależności pomiędzy nazwami, adresami i wartościami zmiennych. Przyjrzyjmy się temu zagadnieniu nieco bliżej.

Pisząc program, koncentrujemy się głównie na dwóch cechach zmiennych: nazwie i wartości (inne cechy, jak typ, możemy tu zignorować). Po skompilowaniu i załadowaniu programu komputer również zwraca uwagę na dwie cechy zmiennych: wartość i adres. Adres możemy określić jako komputerowy odpowiednik nazwy.

W wielu językach adres zmiennej jest prywatną sprawą komputera, niewidoczną dla programisty. Język C pozwala natomiast uzyskać adres zmiennej za pomocą operatora &.

`&trawa` jest adresem zmiennej `trawa`.

Aby uzyskać wartość, wystarczy podać nazwę zmiennej: `printf("%d\n", trawa)` wyświetli wartość zmiennej `trawa`.

Aby uzyskać wartość przechowywaną pod danym adresem, korzystamy z operatora *.

Jeśli `ptrawa == &trawa`, to `*ptrawa` jest wartością przechowywaną pod adresem `&trawa`.

Mówiąc w skrócie, podstawową cechą zwykłej zmiennej jest wartość; adres jest zaś cechą drugorzędną, uzyskiwaną za pośrednictwem operatora &. W przypadku zmiennej wskaźnikowej jest dokładnie odwrotnie: cechą drugorzędną jest wartość — aby ją bowiem otrzymać, musimy skorzystać z operatora *.

Choć możesz chcieć sprawdzić adres zmiennej tylko po to, aby zaspokoić swoją ciekawość, nie jest to główne zastosowanie operatora &. Operator ten (a także *) pozwala przeprowadzać operacje, w których adresy wyrażone są w sposób symboliczny, tak jak zrobiliśmy to w programie *zamien3.c* (patrz listing 9.15).

Podsumowanie. Funkcje

Postać:

Typowa definicja funkcji w ANSI C ma następującą postać:

```
typ zwracany nazwa(lista argumentów)
treść funkcji
```

Lista argumentów to lista deklaracji zmiennych, rozdzielonych przecinkami. Zmienne niebędące argumentami deklarujemy w części głównej funkcji, ograniczonej klamrami.

Przykłady:

```
int roznica(int x, int y)     // wersja ANSI C
{                             // poczatek tresci funkcji
  int z;                      // deklaracja zmiennej lokalnej
  z = x – y;
  return z;                   // zwrocenie wartosci
}                             // koniec tresci funkcji
```

Przekazywanie wartości:

Do przekazywania wartości z funkcji wywołującej do funkcji wywoływanej służą argumenty. Jeśli zmienne `a` i `b` mają wartości `5` i `2`, to wywołanie

```
c = roznica(a,b);
```

przesyła te wartości do zmiennych `x` i `y`. Wartości `5` i `2` nazywamy *argumentami faktycznymi*, a zmienne `x` i `y` w funkcji `roznica()` *argumentami formalnymi*. Słowo kluczowe `return` przekazuje pojedynczą wartość z funkcji wywoływanej do funkcji wywołującej. W powyższym przykładzie `c` otrzymuje wartość zmiennej `z`, czyli `3`. Aby możliwa była modyfikacja więcej niż jednej zmiennej w funkcji wywołującej, argumenty formalne muszą być wskaźnikami.

Typ zwracany funkcji:

Typ zwracany funkcji określa typ wartości zwracanych do wywołującego. Jeśli zwracana wartość jest innego typu niż deklarowany typ zwracany, wartość zwracana jest rzutowana na odpowiedni typ.

Sygnatura funkcji:

Na sygnaturę funkcji składa się typ zwracany funkcji wraz z listą parametrów funkcji; sygnatura określa więc typy wartości przekazywanych do funkcji w jej wywołaniu i typ wartości odbierany z funkcji po jej zakończeniu.

Przykład:

```
double duff(double, int);  // prototyp funkcji
int main()
{
  double q, x;
  int n;
...
  q = duff(x,n);             // wywolanie funkcji
...
}
double duff(double u, int k) // definicja funkcji
{
  double tor;
...
  return tor;          // zwraca wartosc typu double
}
```

Kluczowe zagadnienia

Jeśli chcesz programować w języku C z powodzeniem i efektywnie, musisz zrozumieć funkcje. Podział programu na kilka funkcji jest nie tylko przydatny, ale nawet konieczny. Jeśli skorzystasz z praktyki przydzielania każdej funkcji dokładnie do jednego zadania, Twoje programy staną się łatwiejsze do zrozumienia i poprawiania. Upewnij się, że wiesz, jak funkcje przekazują między sobą dane — to znaczy, że rozumiesz, jak działa mechanizm argumentów i zwracania wartości przez funkcje. Wiedz, że argumenty i zmienne lokalne mają wewnątrz funkcji zasięg prywatny; dlatego też zadeklarowanie dwóch zmiennych o tej samej nazwie w różnych funkcjach

utworzy dwie różne zmienne. Funkcja nie ma bezpośredniego dostępu do zmiennych zadeklarowanych w innej funkcji. Takie ograniczenie ułatwia zachowanie integralności danych. Jeśli jednak chcesz, aby funkcja miała dostęp do danych innej funkcji, możesz użyć (jako argumentów) wskaźników.

Podsumowanie rozdziału

Funkcje są elementami, z których złożone są programy. Każda funkcja powinna mieć jedno dobrze określone zadanie. Do przekazywania wartości do funkcji służą argumenty, a do zwracania wartości — słowo kluczowe `return`. Jeśli funkcja zwraca wartość nienależącą do typu `int`, musisz określić jej typ zarówno w definicji, jak i w obszarze deklaracyjnym funkcji wywołującej. Jeśli chcesz, aby funkcja mogła wpływać na zmienne w funkcji wywołującej, użyj adresów i wskaźników.

Standard ANSI C udostępnia *prototypy funkcji* — przydatne narzędzie, które daje kompilatorom możliwość sprawdzania poprawności wywołań funkcji pod kątem liczby i typów przekazywanych parametrów.

Funkcja w języku C może wywoływać samą siebie; zjawisko to nosi nazwę *rekurencji*. Niektóre problemy obliczeniowe dobrze pasują do rekurencji, ale sama rekurencja może być nieefektywna czasowo i pamięciowo.

Pytania sprawdzające

Odpowiedzi na pytania sprawdzające znajdziesz w dodatku A.

1. Na czym polega różnica między argumentem faktycznym i formalnym?

2. Napisz nagłówki funkcji (w stylu ANSI) odpowiadające poniższym opisom (pomiń treść funkcji):

a) funkcja `pakiet()` wyświetla liczbę zer określoną przez przekazany jej argument typu `int`.

b) funkcja `bieg()` pobiera dwa argumenty typu `int` i zwraca typ `int`.

c) funkcja `zgadula()` nie pobiera argumentów i zwraca wartość typu `int`.

d) funkcja `podloga()` pobiera zmienną typu `double` oraz adres zmiennej typu `double` i zapisuje przekazaną zmienną pod podanym adresem.

3. Napisz nagłówki funkcji (w stylu ANSI) odpowiadające poniższym opisom (pomiń treść funkcji):

a) funkcja `n_znak()` pobiera argument typu `int`, a zwraca argument typu `char`.

b) funkcja `cyfry()` pobiera argument typu `double` oraz argument typu `int`, a zwraca liczbę typu `int`.

c) funkcja `ktory()` pobiera dwa adresy wartości typu `double` i zwraca adres wartości typu double.

d) funkcja `los()` nie pobiera argumentów, a zwraca wartość `int`.

4. Zaprojektuj funkcję zwracającą sumę dwóch liczb całkowitych.
5. Jakie zmiany (jeśli w ogóle) musiałbyś wprowadzić w funkcji z pytania 4, aby sumowała ona dwie liczby typu `double`?
6. Zaprojektuj funkcję `zmien()`, która pobiera dwie zmienne typu `int` o nazwach `x` i `y` i przypisuje im odpowiednio wartość ich sumy i różnicy.
7. Czy poniższa definicja funkcji jest poprawna?

```
void salami(num)
{
   int num, licznik;
   for (licznik = 1; licznik <= num; num++)
       printf(" o salami mio!\n");
}
```

8. Napisz funkcję zwracającą największy z trzech argumentów całkowitych.
9. Zakładając poniższe dane wyjściowe:

```
Wybierz jedna z ponizszych mozliwosci:
1) kopiowanie plikow          2) przenoszenie plikow
3) usuwanie plikow            4) koniec
Podaj numer wybranej opcji:
```

a) Napisz funkcję, która wyświetla menu składające się z czterech ponumerowanych opcji (powinno ono wyglądać tak, jak powyżej).

b) Napisz funkcję pobierającą dwa argumenty typu `int`: granicę dolną i górną. Powinna ona odczytać z klawiatury liczbę całkowitą. Jeśli liczba nie mieści się w granicach, funkcja powinna ponownie wyświetlić menu (korzystając z funkcji z punktu a) i pobrać nową wartość. W przypadku wpisania liczby mieszczącej się w granicach funkcja powinna zwracać ją do funkcji wywołującej. Wpisanie wartości nieliczbowej powinno spowodować zwrócenie z funkcji wartości 4.

c) Napisz program atrapę wykorzystujący funkcje z punktów a. i b. tego pytania. Słowo „atrapa" oznacza, że program nie musi naprawdę wykonywać czynności zapowiadanych w menu; powinien on po prostu wyświetlać opcje i pobierać odpowiedź użytkownika.

Ćwiczenia

1. Zaprojektuj funkcję `min(x,y)`, zwracającą mniejszą z dwóch wartości typu `double`, i przetestuj ją za pomocą prostego programu.
2. Zaprojektuj funkcję `rzad_zn(ch,i,j)`, wyświetlającą znak `ch` w kolumnach od `i` do `j`. Wypróbuj ją w prostym programie.

3. Napisz funkcję, która pobiera trzy argumenty: znak oraz dwie liczby całkowite. Pierwsza liczba określa liczbę razy, jaką należy wyświetlić znak w jednym wierszu; druga liczba określa liczbę wierszy. Napisz program, który wykorzystuje tę funkcję.
4. Średnią harmoniczną dwóch liczb uzyskujemy przez znalezienie odwrotności danych liczb, wyciągnięcie z nich średniej arytmetycznej i obliczenie odwrotności otrzymanego wyniku. Napisz funkcję, która pobiera dwa argumenty typu `double` i zwraca ich średnią harmoniczną.
5. Napisz i sprawdź funkcję o nazwie `wieksze_od()`, która zamienia zawartość obu zmiennych typu double większą z nich. Na przykład `wieksze_od(x,y)` przypisze obu zmiennym x i y wartość większej z nich.
6. Napisz i sprawdź funkcję, która pobiera adresy trzech wartości typu `double` i przepisuje najmniejszą z tych wartości pod pierwszy adres, wartość środkową pod drugi adres, a wartość największą pod trzeci adres.
7. Napisz program, który odczytuje znaki z wejścia standardowego aż do wystąpienia końca pliku. Dla każdego znaku program powinien informować, czy jest on literą. Jeśli tak, program powinien również wyświetlić numer litery w alfabecie. Na przykład litery *c* i *C* obie mają numer 3. Wykorzystaj funkcję, która pobiera znak jako argument i zwraca jego numer w alfabecie, jeśli jest on literą; w przeciwnym wypadku wartością zwracaną powinno być –1.
8. W rozdziale 6. napisaliśmy funkcję `potega()`, która zwracała wynik podniesienia liczby typu `double` do potęgi naturalnej (patrz listing 6.20). Ulepsz tę funkcję tak, aby poprawnie obsługiwała potęgi ujemne. Ponadto wbuduj w funkcję założenie, że 0 do dowolnej potęgi wynosi 0 oraz że podniesienie dowolnej liczby do potęgi 0 daje wynik 1 (funkcja powinna zasygnalizować, że 0 do potęgi 0 nie zadziała i że funkcja zamiast tego użyje wartości 1). Użyj pętli. Przetestuj funkcję w programie.
9. Ponownie wykonaj ćwiczenie 8. — tym razem użyj funkcji rekurencyjnej.
10. Uogólnij funkcję `do_binar` z listingu 9.8 do postaci `do_podst_n()`, pobierającej jako drugi argument wartość z przedziału od 2 do 10. Następnie powinna ona wyświetlić liczbę pobraną jako pierwszy argument w odpowiadającym drugiemu argumentowi systemie liczbowym. Na przykład funkcja `do_podst_n(129, 8)` wyświetli `201`, co jest ósemkowym odpowiednikiem liczby `129`. Sprawdź działanie funkcji, pisząc wykorzystujący ją program.
11. Napisz i przetestuj funkcję `Fibonacci()`, która zamiast rekurencji do obliczania kolejnych wyrazów ciągu Fibonacciego używa pętli.

SKOROWIDZ

D

E

F

G

I

J

K

L

Ł

M

N

O

Q

R

S

Ś

T

U

V

W

Z

Naucz się C, a zrozumiesz istotę programowania!

Język C niewątpliwie należy do kanonu języków programowania. Cechuje się elegancją i prostotą, jest wszechstronny i elastyczny, jednak uważa się go za trudny i wymagający. Na pewno warto opanować C — jeśli nauczysz się tworzyć solidny kod w tym języku, poradzisz sobie z każdym innym językiem programowania.

Trzymasz w dłoni kolejne wydanie niezwykle popularnego podręcznika do nauki C. Podobnie jak poprzednie wydania, także to zostało zaktualizowane i uzupełnione, między innymi o elementy standardu C11. Książka stanowi przemyślane, przejrzyste i wnikliwe wprowadzenie do języka C. Czytelnie wyjaśnia zasady programowania, zawiera opisy licznych rozwiązań programistycznych, setki przykładów kodu oraz ćwiczenia do samodzielnego wykonania. Dzięki takiemu układowi treści wiele osób nauczyło się C właśnie z tej książki, a jej kolejne wydania są przyjmowane z entuzjazmem.

W tej książce znajdziesz:

- kompletne omówienie podstaw języka C i najważniejszych paradygmatów programowania
- wyczerpujące informacje o nowych elementach C
- jasne wskazówki dotyczące wyboru poszczególnych metod programowania w różnych sytuacjach
- setki przykładowych fragmentów kodu
- pytania sprawdzające i ćwiczenia utrwalające w każdym rozdziale
- liczne informacje o możliwościach języka C

Stephen Prata — jest emerytowanym wykładowcą astronomii, fizyki i programowania w College of Marin w Kentfield w Kalifornii. Obronił doktorat na Uniwersytecie Kalifornijskim w Berkeley. Jego przygoda z programowaniem komputerów rozpoczęła się od modelowania ruchu gwiazd. Jest autorem i współautorem licznych książek, między innymi na temat języka C i Uniksa.

Helion

44764 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

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

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

ISBN 978-83-283-1470-2

9 788328 314702

Informatyka w najlepszym wydaniu

cena: 99,00 zł

Addison Wesley

ALWAYS LEARNING

PEARSON