# Język C. Szkoła programowania. Wydanie V


Autor: Stephen Prata
Tłumaczenie: Tomasz Szynalski, Grzegorz Joszcz
ISBN: 83-246-0291-7
Tytuł oryginału: C Primer Plus (5th Edition)
Format: B5, stron: 976

„Język C. Szkoła programowania. Wydanie V" jest uaktualnioną wersją bestsellerowej książki wydanej pod tym samym tytułem w roku 1999 nakładem wydawnictwa Robomatic. Podręcznik ten w czytelny sposób prezentuje kolejne tematy, ilustrując je przykładowymi programami. Każdy rozdział kończą pytania sprawdzające wraz z odpowiedziami oraz zadania programistyczne.

W książce znajdziemy pełny opis standardu (C99) języka C, w tym m.in. szczegółową charakterystykę:

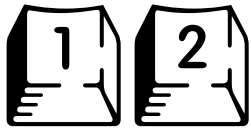
- rozszerzonych typów całkowitych i zbiorów znaków,
- tablic o zmiennej długości (VLA),
- złożonych literałów,
- rozszerzonych zbiorów znaków oraz typów logicznych,
- funkcji wplatanych (inline),
- inicjalizatorów oznaczonych struktur.

Autor nie ogranicza się do opisu instrukcji języka C. Ujawnia także techniki efektywnego programowania oraz przedstawia wybrane algorytmy i struktury danych.

Potwierdzeniem jakości książki jest sukces, jaki odniosła w Stanach Zjednoczonych – pięć wydań i ponad 400 tys. sprzedanych egzemplarzy. Na rynku amerykańskim zaliczana jest już do klasyki.

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

# SPIS TREŚCI



# KLASY ZMIENNEJ, ŁĄCZNOŚĆ I ZARZĄDZANIE PAMIĘCIĄ

**W tym rozdziale poznasz:**

- Słowa kluczowe: `auto`, `extern`, `static`, `register`, `const`, `volatile`, `restrict`
- Funkcje: `rand()`, `srand()`, `time()`, `malloc()`, `calloc()`, `free()`
- Sposób, w jaki język C pozwala decydować o zasięgu zmiennej, (czyli określić jej dostępność w poszczególnych miejscach w programie) i jej czasie trwania, (czyli jak długo będzie ona istnieć)
- Metody projektowania bardziej złożonych programów

Jedną z mocnych stron języka C jest to, że pozwala on na kontrolę nad szczegółami programu. Przykładem może tu być system zarządzania pamięcią, pozostawiający programiście decyzje dostępności poszczególnych zmiennych dla określonych funkcji oraz jak długo dane zmienne mają istnieć w programie. Wybór odpowiednich klas zmiennych jest kolejnym etapem projektowania programu.

## Klasy zmiennych

Język C dostarcza pięciu modeli czy też *klas* (*ang. storage class*) zmiennych. Istnieje też szósty model, oparty na wskaźnikach, do którego wrócimy w dalszej części rozdziału (w sekcji „Przydział pamięci: funkcje `malloc()` i `free`()”).

Zmienną (albo ogólniej: *obiekt danych* (*ang. data object*)) można opisać w kategoriach *czasu trwania* (*ang. storage duration*), określającego jak długo zmienna pozostaje w pamięci programu, jej *zasięgu* (*ang. scope*) i *łączności* (*ang. linkage*). Razem wzięte, atrybuty te decydują, które moduły programu mają dostęp do danej zmiennej. Poszczególne klasy pozwalają na rozmaite kombinacje zasięgów, łączności i czasu trwania. W programie można używać zmiennych, które mogą być współdzielone przez kilka plików źródłowych, a także takich, które mogą być wykorzystywane przez dowolną funkcję w obrębie określonego pliku. Możemy ponadto używać zmiennych, które będą dostępne tylko wewnątrz określonej funkcji. Wreszcie mamy do dyspozycji zmienne dostępne tylko w określonych obszarach danej funkcji. Program może używać zmiennych, które będą istniały przez cały czas jego działania i takich, które będą istnieć tylko podczas wykonywania zawierających je funkcji. Dodatkowo programista może przechowywać dane bezpośrednio w pamięci, przydzielając i zwalniając ją jawnie przez wywołania odpowiednich funkcji.

Zanim przeanalizujemy klasy zmiennych, musimy zrozumieć sens wspomnianych wyżej pojęć: *zasięg*, *łączność* (*ang. linkage*) i *czas trwania* (*ang. storage duration*). Następnie wrócimy do omawiania poszczególnych klas.

## Zasięg zmiennej

Zasięg zmiennej określa region lub regiony programu, które mają dostęp do identyfikatora zmiennej. Zmienna w języku C może posiadać jeden z następujących zasięgów: *blokowy*, *prototypowy* lub *plikowy*. Do tej pory w naszych przykładach używaliśmy zmiennych o zasięgu blokowym. *Blok* (*ang. block*), jak sobie zapewne przypominasz, to fragment programu objęty klamrami, czyli zawarty między klamrą otwierającą a klamrą zamykającą. Przykładowo, całe ciało funkcji stanowi blok. Podobnie, każde złożone polecenie wewnątrz funkcji jest blokiem. Zmienna zdefiniowana wewnątrz bloku ma *zasięg blokowy* i jest dostępna od miejsca, w którym została zdefiniowana aż do końca bloku zawierającego jej definicję. Tak samo, argumenty formalne funkcji, choć znajdują się przed klamrą otwierającą funkcję, mają zasięg blokowy i należą do bloku obejmującego ciało tej funkcji. Zatem zmienne lokalne, które stosowaliśmy dotychczas, włącznie z argumentami formalnymi funkcji, miały *zasięg blokowy*.

Stąd też obie zmienne `kleofas` i `patryk` w podanym niżej przykładzie mają zasięg blokowy, rozciągający się aż do klamry zamykającej funkcję:

```
double blok(double kleofas)
{
double patryk = 0.0;
…
return patryk;
}
```

Zmienne zadeklarowane w wewnętrznym bloku mają zasięg ograniczony jedynie do niego:

```
double blok(double kleofas)
{
double patryk = 0.0;
int i;
for(i = 0; i < 10; i++)
{
      double q = kleofas * i;         // poczatek zasiegu zmiennej q
      …
      patryk *= q;                    // koniec zasiegu zmiennej q
}
…
return patryk;
}
```

W podanym przykładzie zasięg zmiennej `q` jest ograniczony do wewnętrznego bloku i wyłącznie kod znajdujący się w jego obrębie ma dostęp do tej zmiennej. Tradycyjnie, zmienne o takim zasięgu musiały być deklarowane na samym początku bloku. Standard C99 złagodził tę regułę, zezwalając na deklarację zmiennych w dowolnym miejscu bloku. Nowości pojawiły się też w wyrażeniach sterujących pętli `for`. Dzięki nim poprawny jest następujący kod:

```
for(int i = 0; i < 10; i++)
      printf("Nowość w standardzie C99: i = %d", i);
```

Jedną z nowych cech standardu języka C w wersji C99 jest rozszerzenie pojęcia bloku tak, by obejmował on kod nadzorowany przez pętle `for`, `while`, `do while` oraz instrukcję warunkową `if`, nawet jeśli nie jest on ujęty w klamry. Dzięki temu, w poprzednim przykładzie zmienna `i` jest traktowana jako część bloku pętli `for`. W konsekwencji jej zasięg jest ograniczony do bloku pętli. Po wyjściu z niej, program traci dostęp do zmiennej `i`.

*Zasięg prototypowy funkcji* (*ang. function prototype scope*) dotyczy nazw zmiennych użytych w prototypach funkcji tak, jak w następującym przykładzie:

```
int potezna(int mysz, double wielka);
```

Zasięg prototypowy obejmuje obszar od miejsca, w którym zdefiniowano zmienną, aż do końca deklaracji prototypu. Oznacza to, że kompilator, przetwarzając argumenty prototypu funkcji, zwraca uwagę wyłącznie na ich typy; nazwy, o ile jakieś podano, nie mają żadnego znaczenia i nie muszą odpowiadać nazwom argumentów użytych w definicji funkcji. Jedynym przypadkiem, w którym nazwy mają jakieś znaczenie, są argumenty tablic o zmiennym rozmiarze (VLA):

```
void uzyjVLA(int n, int m, tbl[n][m]);
```

Jeżeli w nawiasach kwadratowych (określających rozmiar tablicy) znajdują się nazwy zmiennych, to muszą one zostać wcześniej zadeklarowane.

Zmienna, której definicja znajduje się poza blokami funkcji ma *zasięg plikowy* (*ang. file scope*). Jest ona dostępna od miejsca, w którym została zdefiniowana aż do końca pliku zawierającego jej definicję. Spójrzmy na następujący przykład:

```
#include <stdio.h>
int jednostki = 0;
void krytyka(void);
int main(void)
{
      …
}
void krytyka(void)
{
      …
}
```

Jak widać zmienna `jednostki` ma zasięg plikowy i może być używana zarówno w funkcji `main`(), jak i w funkcji `krytyka`(). Ponieważ zmienne o *zasięgu plikowym* mogą zostać użyte w wielu funkcjach, nazywane bywają również *zmiennymi globalnymi.*

Jest jeszcze jeden typ zasięgu, nazywany *funkcyjnym* (*ang. function scope*), jednak dotyczy on tylko etykiet wykorzystywanych z poleceniem `goto`. Zasięg funkcyjny oznacza, że etykieta `goto` znajdująca się w danej funkcji jest dostępna wszędzie w tej funkcji, niezależnie od bloku, w którym się pojawia.

## Łączność zmiennej

W dalszej kolejności przyjrzymy się pojęciu *łączności zmiennej*. Zmienna w języku C może charakteryzować się jednym z następujących typów łączności: *łącznością zewnętrzną* (*ang. external linkage*), *łącznością wewnętrzną* (*ang. internal linkage*) lub *brakiem łączności*. Zmienne o zasięgu blokowym lub prototypowym cechuje brak łączności. Oznacza to, że są one prywatne w obrębie bloku lub prototypu, w którym zostały zdefiniowane.

Zmienna o zasięgu plikowym może mieć albo łączność wewnętrzną, albo zewnętrzną. Zmienna o łączności zewnętrznej może być użyta w dowolnym miejscu złożonego z wielu plików programu. Zmienna o łączności wewnętrznej może być zastosowana w dowolnym miejscu, ale tylko w obrębie jednego pliku.

Jak zatem stwierdzić, czy zmienna o zasięgu plikowym ma łączność wewnętrzną, czy zewnętrzną? Należy sprawdzić, czy w zewnętrznej definicji użyto specyfikatora klasy zmiennych `static`:

```
int gornik = 5;              // zasieg plikowy, lacznosc zewnetrzna
static int legia = 3;        // zasieg plikowy, lacznosc wewnetrzna
int main()
{
      …
}
```

Zmienna `gornik` jest dostępna w dowolnym pliku, stanowiącym część tego samego programu. Zmienna `legia` natomiast jest prywatna w obrębie danego pliku, ale może być użyta przez dowolną funkcję w nim zawartą.

## Czas trwania zmiennej

Zmienna w języku C charakteryzuje się również jednym z dwóch czasów trwania: *statycznym* (*ang. static storage duration*) lub *automatycznym* (*ang. automatic storage duration*). Jeśli zmienna ma statyczny czas trwania, to istnieje przez cały czas wykonywania programu. Taki właśnie charakter mają zmienne o zasięgu plikowym. Zauważ, że przy tego typu zmiennych, słowo kluczowe `static` wskazuje na typ łączności, a nie na statyczny czas trwania. Zmienna o zasięgu plikowym zadeklarowana ze słowem kluczowym `static` ma łączność wewnętrzną, ale wszystkie zmienne o zasięgu plikowym, posiadające łączność wewnętrzną albo zewnętrzną, mają statyczny czas trwania.

Zmienne o zasięgu blokowym zwykle mają automatyczny czas trwania. Pamięć dla nich jest przydzielana (alokowana) w chwili, gdy program wchodzi do bloku, w którym zostały zdefiniowane, po czym jest zwalniana z chwilą jego opuszczenia. Pamięć przeznaczona dla zmiennych automatycznych jest traktowana jak tymczasowy obszar roboczy, który może być ponownie użyty. Dla przykładu, kiedy wywołana funkcja zakończy pracę, pamięć zajmowana do tej pory przez jej zmienne, może być wykorzystana przez zmienne kolejnej wywoływanej funkcji.

Zmienne lokalne, z których do tej pory korzystaliśmy, należały do kategorii zmiennych automatycznych. W podanym fragmencie programu, zmienne `liczba` i `indeks` tworzone są za każdym razem, gdy wywoływana jest funkcja `nuda`() i niszczone zawsze, gdy funkcja kończy swoje działanie.

```
void nuda(int liczba)
{
      int indeks;
for (indeks = 0; indeks < liczba; indeks++)
      puts("Nic juz nie jest takie jak kiedys.\n");
return 0;
}
```

Język C używa pojęć: zasięgu, łączności i czasu trwania, by zdefiniować pięć klas zmiennych: *automatyczną* (*ang. automatic*), *rejestrową* (*ang. register*), *statyczną o zasięgu blokowym* (*ang. static with block scope*), *statyczną o łączności zewnętrznej* (*ang. static with external linkage*) i *statyczną o łączności wewnętrznej* (*ang. static with internal linkage*). Tabela 12.1 przedstawia wszystkie klasy. Znając pojęcia podstawowe, możemy przejść do szczegółowego opisu klas zmiennych.

## Zmienne automatyczne

Zmienne należące do *klasy zmiennych automatycznych* charakteryzowane są przez: automatyczny czas trwania, zasięg blokowy i brak łączności. Do tej klasy należą domyślnie wszystkie zmienne zadeklarowane w bloku albo nagłówku funkcji. Jako programista możesz jednak, choć nie jest to konieczne, użyć słowa kluczowego `auto`, by podkreślić swoje zamiary, tak jak pokazano niżej:

**TABELA 12.1.** Pięć klas zmiennych

| Klasa zmiennych | Czas trwania | Zasięg | Łączność | Jak deklarujemy |
|---|---|---|---|---|
| Automatyczna | Automatyczny | Blokowy | Brak | W bloku |
| Rejestrowa | Automatyczny | Blokowy | Brak | W bloku ze słowem kluczowym `register` |
| Statyczna o łączności zewnętrznej | Statyczny | Plik | Zewnętrzna | Poza obszarem funkcji |
| Statyczna o łączności wewnętrznej | Statyczny | Plik | Wewnętrzna | Poza obszarem funkcji ze słowem kluczowym `static` |
| Statyczna bez łączności | Statyczny | Blokowy | Brak | W bloku ze słowem kluczowym `static` |

```
int main(void)
{
      auto int zmienna;
```

Możesz to zrobić choćby po to, by udokumentować, że przesłonięcie zewnętrznej definicji funkcji jest zgodne z Twoim zamiarem, albo że ważne jest, by klasa zmiennej nie została zmieniona. Słowo kluczowe `auto` to *specyfikator klasy zmiennych* (*ang. storage class specifier*).

Zasięg blokowy i brak łączności powodują, że dostęp do zmiennej poprzez jej nazwę jest możliwy tylko w obrębie bloku zawierającego jej deklarację. (Oczywiście, można przekazać wartość lub adres zmiennej poprzez parametry do innej funkcji, nie będzie to jednak bezpośredni dostęp). Inna funkcja może korzystać ze zmiennej o takiej samej nazwie, będzie to jednak zupełnie niezależna zmienna, zajmująca osobne miejsce w pamięci.

Przypomnijmy, że automatyczny czas trwania oznacza, że zmienna zaczyna istnieć, gdy program wchodzi do bloku zawierającego deklarację zmiennej. Kiedy program opuszcza ten blok, zmienna automatyczna przestaje istnieć. Jej miejsce w pamięci może być odtąd wykorzystywane do innych potrzeb.

Przyjrzyjmy się bliżej zagnieżdżonym blokom. Zmienna jest znana tylko blokowi, w którym została zadeklarowana i każdemu blokowi w nim zagnieżdżonemu.

```
int petla(int n)
{
      int m;          // m jest w zasięgu
      scanf("%d", &m);
      {
int i;        // w zasiegu sa zmienne m i i
for(i = m; i < n; i++)
      puts("i jest zmienna lokalna w podbloku\n");
}
return m;             // w zasiegu jest m, a i zniknelo
}
```

W powyższym przykładzie zmienna `i` dostępna jest tylko w obrębie bloku objętego wewnętrznymi klamrami. Próba jej użycia przed lub po tym bloku, zakończy się błędem kompilacji. Zwykle nie wykorzystuje się tego faktu podczas projektowania programu. Czasami jednak opłaca się zdefiniować zmienną w podbloku, jeśli nie będzie ona używana nigdzie indziej. W ten sposób można umieścić komentarz do zmiennej blisko miejsca jej użycia. Ponadto zmienna nie będzie zajmować niepotrzebnie pamięci, gdy nie będzie już potrzebna. Zmienne `n` i `m`, zdefiniowane w nagłówku funkcji i poza jej blokiem, są w zasięgu dla całej funkcji i istnieją dopóki funkcja nie skończy działania.

Co się stanie, gdy zadeklarujesz zmienną o takiej samej nazwie zarówno wewnątrz bloku jak i poza nim? Wewnątrz bloku będzie używana zmienna zdefiniowana w jego obrębie. Mówimy, że zmienna *przesłania* zewnętrzną definicję. Jednak, gdy wykonywany program opuści blok wewnętrzny, zmienna zewnętrzna ponownie znajdzie się w zasięgu. Listing 12.1 ilustruje tą właściwość:

**LISTING 12.1.** Program zasiegi.c

```
/* zasiegi.c -- zmienne w bloku */
#include <stdio.h>
int main()
{
    int x = 30;      /* oryginalna zmienna x          */

    printf("x w zewnetrznym bloku: %d\n", x);
    {
        int x = 77;  /* nowe x przeslania oryginalne x */
        printf("x w wewnetrznym bloku: %d\n", x);
    }
    printf("x w zewnetrznym bloku: %d\n", x);
    while (x++ < 33) /* oryginalne x          */
    {
        int x = 100; /* nowe x przeslania oryginalne x */
        x++;
        printf("x w petli loop: %d\n", x);
    }
    printf("x w zewnetrznym bloku %d\n", x);

    return 0;
}
```

Oto, co otrzymujemy na wyjściu:

```
x w zewnetrznym bloku: 30
x w wewnetrznym bloku: 77
x w zewnetrznym bloku: 30
x w zewnetrznym bloku: 101
x w zewnetrznym bloku: 30
x w zewnetrznym bloku: 30
x w zewnetrznym bloku: 30
```

Na początku program tworzy zmienną x o wartości 30, co pokazuje pierwsza instrukcja printf(). Następnie definiuje nową zmienną x o wartości 77, co wyświetla druga instrukcja printf(). To, że jest to nowa zmienna przesłaniająca pierwszą, widzimy dzięki trzeciej instrukcji printf(). Znajduje się ona za pierwszym wewnętrznym blokiem i wyświetla pierwotną wartość zmiennej x, dowodząc, że oryginalna wartość tej zmiennej, ani nie znikła, ani też nie została zmieniona.

Prawdopodobnie najbardziej intrygującą częścią programu jest pętla while. Sprawdza ona warunek wykonania pętli, używając pierwszej zmiennej x:

```
while(x++ < 33)
```

Jednak wewnątrz pętli program widzi trzecią zmienną x, zdefiniowaną wewnątrz bloku pętli while. Zatem, gdy kod używa wyrażenia x++ w ciele pętli, to dotyczy on już nowej zmiennej x, która zwiększona o 1, przyjmuje wartość 101, a następnie zostaje wyświetlona. Nowa zmienna x znika, gdy wszystkie instrukcje pętli zostaną wykonane. Wówczas sprawdzany jest warunek wykonania pętli, który przy tym zwiększa o 1 oryginalną wartość x, a program rozpoczyna kolejny cykl, w którym ponownie tworzy wewnętrzną zmienną x. W tym przykładzie zmienna ta jest tworzona i niszczona aż trzy razy. Zauważ, że pętla, by się zakończyć, musiała inkrementować x w warunku testowym, gdyż zmienna x inkrementowana w ciele pętli była całkiem inną zmienną niż w warunku pętli.

Celem tego przykładu nie było propagowanie pisania nieprzejrzystych programów, lecz przybliżenie idei zasięgu zmiennych w języku C.

## Bloki bez klamr

Cechą standardu C99, o której wspomnieliśmy już wcześniej, jest to, że kod będący częścią pętli albo instrukcji if, traktowany jest jak blok, nawet jeśli nie zawiera nawiasów klamrowych (to znaczy: { i }). Mówiąc ściślej, taka pętla stanowi podblok w stosunku do bloku, w którym występuje. Ciało pętli z kolei jest podblokiem względem całego bloku pętli. Podobnie instrukcja if jest blokiem, a związane z nią wyrażenia są jej podblokami. Powyższe reguły określają miejsca w których można zadeklarować zmienną i jej zasięg. Listing 12.2 ilustruje to na przykładzie pętli for.

**LISTING 12.2.** Program forc99.c

```
// forc99.c -- nowe zasady dla zasiegu blokowego w petli for(C99)
#include <stdio.h>
int main()
{
    int n = 10;

    printf("Poczatkowo n = %d\n", n);
    for (int n = 1; n < 3; n++)
        printf("petla 1: n = %d\n", n);
    printf("Po petli 1, n = %d\n", n);
    for (int n = 1; n < 3; n++)
```

```
    {
        printf("petla 2 indeks n = %d\n", n);
        int n = 30;
        printf("petla 2: n = %d\n", n);
        n++;
    }
    printf("Po petli 2, n = %d\n", n);

    return 0;
}
```

Jeśli użyjemy kompilatora zgodnego ze standardem C99, wynik działania programu będzie następujący:

```
poczatkowo n = 10
petla 1 n = 1
petla 1 n = 2
po petli 1 n = 10
petla 2 indeks n = 1
petla 2: n = 30
petla 2 indeks n = 2
petla 2: n = 30
po petli 2 n = 10
```

### Obsługa standardu C99

Niektóre kompilatory mogą nie spełniać reguł standardu C99 odnośnie zasięgu zmiennych. Inne mogą ich przestrzegać jedynie opcjonalnie. Na przykład, w czasie gdy ta książka była pisana, kompilator **gcc** obsługiwał wiele nowych możliwości C99, ale ich użycie wymagało zastosowania przełącznika `–std=c99`:

```
gcc –std=c99 forc99.c
```

Zmienna `n` zadeklarowana w wyrażeniu sterującym pierwszej pętli `for` ma zasięg aż do końca tej pętli i przesłania wcześniej zadeklarowaną zmienną `n`. Kiedy po wykonaniu pętla jest opuszczana, wcześniejsze `n` znowu znajduje się w zasięgu.

W drugiej pętli `for` zmienna `n` zadeklarowana jako zmienna indeksowa pętli, przykrywa wcześniejszą zmienną `n`. Indeks pętli przykrywa z kolei inna zmienna `n` zadeklarowana w ciele pętli. Znika ona za każdym razem, gdy program kończy wykonywanie kodu pętli i sprawdzany jest warunek wykonania przy zmiennej indeksowej `n`. Pierwotna zmienna `n` ponownie znajdzie się w zasięgu, gdy program zakończy wykonywanie wewnętrznej pętli.

## Inicjalizowanie zmiennych automatycznych

Zmiennym automatycznym nie są nadawane wartości początkowe, czyli inaczej mówiąc nie są one inicjalizowane, jeśli nie zostanie to jawnie nakazane przez programistę. Rozważmy następującą deklarację:

```
int main(void)
{
      int raport;
      int namioty = 5;
```

Zmienna `namioty` jest inicjalizowana wartością 5, ale zmienna `raport` będzie posiadać przypadkową wartość zależną od zawartości przydzielonego jej obszaru pamięci. Nie wolno zakładać, że będzie to wartość 0. Zmienną automatyczną można inicjalizować przy pomocy dowolnego wyrażenia złożonego z wcześniej zdefiniowanych zmiennych.

```
int main(void)
{
      int renata = 1;
      int robert = 5 * renata;       // uzyto wczesniej zdefiniowanej
zmiennej
```

## Zmienne rejestrowe

Zmienne przechowywane są zazwyczaj w pamięci komputera. Przy odrobinie szczęścia zmienne rejestrowe trafiają do rejestru procesora, albo ogólniej, do najszybszej dostępnej pamięci, gdzie mogą być odczytywane i przetwarzane znacznie szybciej, niż w przypadku zwykłych zmiennych.

W związku z tym, że zmienne rejestrowe znajdują się zwykle w rejestrach procesora, a nie w pamięci, nie możemy pobierać ich adresów. Poza tymi wyjątkami zmienne rejestrowe możemy traktować jak zmienne automatyczne. Oznacza to, że charakteryzuje je zasięg blokowy, automatyczny czas trwania i brak łączności. Zmienne tej klasy deklarowane są za pomocą specyfikatora `register`:

```
int main(void)
{
      register int szybkazmien;
```

Stwierdziliśmy „przy odrobinie szczęścia", gdyż deklaracja zmiennej jako rejestrowej jest bardziej prośbą niż nakazem. Kompilator musi rozważyć nasze życzenie, biorąc pod uwagę liczbę rejestrów lub dostępną szybką pamięć. Jednym słowem nasza prośba nie musi zostać spełniona. W takim przypadku zmienna zostanie zwyczajną zmienną automatyczną, choć wciąż nie będzie można pobrać jej adresu.

Możemy także wyrazić życzenie, by argumenty formalne funkcji były zmiennymi rejestrowymi. W nagłówku funkcji należy wówczas użyć słowa kluczowego `register`:

```
void macho(register int n)
```

Zależnie od systemu, zbiór typów, jakie można deklarować ze specyfikatorem `register` może być ograniczony. Dla przykładu, rejestry procesora mogą nie być wystarczająco pojemne, by przechowywać zmienne typu `double`.

## Zmienne statyczne o zasięgu blokowym

Określenie *zmienne statyczne* brzmi jak oksymoron; zmienna, która się nie zmienia. W tym przypadku `static` oznacza jednak to, że zmienna pozostaje zawsze w zasięgu. Zmienne o zasięgu plikowym automatycznie (i obowiązkowo) mają statyczny czas trwania. Możliwe jest również tworzenie zmiennych lokalnych — to znaczy, zmiennych o zasięgu blokowym — które mają trwałość statyczną. Zmienne te mają taki sam zasięg, co zmienne automatyczne, lecz nie przestają istnieć, gdy zawierająca je funkcja skończy swoje działanie. Tak więc, zmienne te mają zasięg blokowy oraz brak łączności, ale i statyczny czas trwania. Program pamięta ich wartości niezależnie od funkcji, w której się właśnie znajduje. Zmienne te są tworzone poprzez ich deklarację ze specyfikatorem klasy zmiennych `static` (oznaczającym statyczny czas trwania) w bloku (co zapewnia zasięg blokowy i brak łączności). Przykład z listingu 12.3 przybliża opisane tu zasady.

**LISTING 12.3.** Program lok_stat.c

```
/* lok_stat.c -- uzywanie statycznych zmiennych lokalnych */
#include <stdio.h>
void sprawdz_stat(void);
int main(void)
{
    int licznik;
    for (licznik = 1; licznik <= 3; licznik++)
    {
        printf("Iteracja nr: %d:\n", licznik);
        sprawdz_stat();
    }

    return 0;
}
void sprawdz_stat(void)
{
    int znikam = 1;
    static int trwam = 1;
    printf("znikam = %d, a trwam = %d\n", znikam++, trwam++);
}
```

Zauważ, że funkcja `sprawdz_stat()` zwiększa o 1 każdą zmienną, zaraz po tym, gdy wypisze jej bieżącą wartość. Dlatego program wyświetla następujący wynik:

```
Iteracja nr: 1:
znikam = 1, a trwam =1
Iteracja nr: 2:
znikam = 1, a trwam =2
Iteracja nr: 3:
znikam = 1, a trwam =3
```

Zmienna statyczna `trwam` pamięta, że jej wartość została zwiększona o 1, natomiast zmienna `znikam` za każdym razem zaczyna liczenie od nowa. Wynika to z różnicy w deklaracjach: `znikam` jest inicjalizowana za każdym razem, gdy wywoływana jest

funkcja `sprawdz_stat()`, natomiast zmienna `trwam` jeden jedyny raz, podczas kompilacji funkcji `sprawdz_stat()`. Zmiennym statycznym domyślnie nadawana jest wartość 0, o ile samemu nie przypiszesz im jakiejś innej wartości.

Obie deklaracje wyglądają podobnie:

```
int znikam = 1;
static int trwam = 1;
```

Jednak pierwsza linia w rzeczywistości należy do funkcji `sprawdz_stat()` i jest wykonywana za każdym razem, gdy ta funkcja jest wywoływana. Można powiedzieć, że druga linia właściwie nie jest jej częścią. Jeśli użyjesz debugera, by uruchomić ten program krok po kroku, to zauważysz, że zdaje się on omijać tę instrukcję. Dzieje się tak dlatego, że zmienne statyczne i zmienne zewnętrzne są inicjalizowane w chwili, gdy program jest ładowany do pamięci. Z kolei umieszczenie deklaracji zmiennej statycznej `trwam` w obrębie funkcji `sprawdz_stat()` informuje kompilator, że wyłącznie ta funkcja ma mieć do niej dostęp; nie jest to instrukcja wykonywana podczas pracy programu.

Słowa kluczowego `static` nie można używać w argumentach formalnych funkcji.

```
int zdolny_do_pracy(static int ma_grype);    // niedozwolona instrukcja
```

W starszej literaturze traktującej o języku C ta klasa zmiennych bywała nazywana klasą *statycznych zmiennych wewnętrznych* (*ang. internal static storage class*). Jednakże słowo *wewnętrzne* stosowano wówczas dla podkreślenia faktu występowanie zmiennej wewnątrz funkcji, a nie wskazania na jej łączność wewnętrzną.

## Zmienne statyczne o łączności zewnętrznej

Zmienna statyczna o łączności zewnętrznej ma zasięg plikowy, łączność zewnętrzną i statyczny czas trwania. Taka klasa jest często określana mianem *zewnętrznej klasy zmiennych* (*ang. external storage class*), a zmienne tego typu *zmiennymi zewnętrznymi* (*ang. external variables*).

Aby stworzyć zmienną zewnętrzną, należy umieścić jej deklarację poza obszarem funkcji. Dla celów dokumentacji, zmienne zewnętrzne mogą być ponownie zadeklarowane wewnątrz funkcji, która z nich korzysta, poprzez zastosowanie słowa kluczowego: `extern`. Jeżeli zmienna została zadeklarowana w innym pliku, wówczas ponowna deklaracja ze słowem `extern` jest obowiązkowa. Wspomniane deklaracje wyglądają następująco:

```
int Wybuch;                  /* zmienna zdefiniowana zewnetrzne */
double Gora[100];            /* tablica zdefiniowna zewnetrzne */
extern char Wegiel;          /* musi istniec definicja zmiennej Wegiel */
                             /* w innym pliku */
void nastepny(void);
int main(void)
{
```

```
        extern int Wybuch;      /* deklaracja opcjonalna */
        extern double Gora[];   /* deklaracja opcjonalna */
        …
}
void nastepny(void)
{
…
}
```

Obecność dwóch deklaracji zmiennej `Wybuch` jest przykładem na łączność, ponieważ obie odnoszą się do tej samej zmiennej. Zmienne zewnętrzne mają łączność zewnętrzną; do tego zagadnienia wrócimy w dalszej części rozdziału.

Zauważ, że nie trzeba podawać rozmiaru tablicy w drugiej, opcjonalnej, deklaracji zmiennej `double Gora`. Dzieje się tak dlatego, że pierwsza z nich podała już tą informację. Deklaracje ze słowem `extern` w obrębie funkcji `main()` mogą być pominięte w całości, gdyż zmienne zewnętrzne i tak mają zasięg plikowy, stąd też znane są od miejsca ich deklaracji aż do końca pliku. Ich wystąpienie ma na celu jedynie udokumentowaniu ich użycia w funkcji `main()`.

Jeśli w deklaracji wewnątrz funkcji pomija się słowo `extern`, to tworzona jest wówczas całkiem nowa zmienna automatyczna o podanej nazwie. Stąd zastąpienie instrukcji:

```
extern int Wybuch;
```

przez:

```
int Wybuch;
```

spowoduje, że kompilator stworzy zmienną automatyczną o nazwie `Wybuch`. Będzie to nowa zmienna lokalna, różna od oryginalnej zmiennej zewnętrznej `Wybuch`. Zmienna lokalna znajduje się w zasięgu funkcji `main()`, natomiast zewnętrzna jest widoczna dla pozostałych funkcji w tym samym pliku, takich jak `nastepny()`. Krótko mówiąc, zmienna o zasięgu blokowym przesłania zmienną o zasięgu plikowym o takiej samej nazwie, gdy program wykonuje instrukcje w obrębie jej bloku.

Zmienne zewnętrzne mają statyczny czas trwania. Dlatego, tablica `Gora` istnieje i zachowuje wartości niezależnie od tego, czy program wykonuje właśnie funkcje `main()`, `nastepny()` czy też jakąkolwiek inną.

Poniższe trzy przykłady demonstrują cztery możliwe kombinacje zmiennych zewnętrznych i automatycznych. Przykład 1 zawiera zmienną zewnętrzną: `Hokus`. Jest ona dostępna w obrębie funkcji `main()` i `czarodziej()`.

```
/* Przyklad 1 */
int Hokus;
int czarodziej();
int main(void)
{
        extern int Hokus;       // zmienna Hokus zostala zadeklarowana zewn.
        …
```

```
}
int czarodziej()
{
      extern int Hokus;       // ta sama zmienna Hokus co wyzej
      …
}
```

Przykład 2 przedstawia zmienną zewnętrzną, `Hokus`, dostępną obu funkcjom. Tym razem jednak funkcja `czarodziej()` ma do niej dostęp z definicji.

```
/* Przyklad 2 */
int Hokus;
int czarodziej();
int main(void)
{
      extern int Hokus;       // zmienna Hokus zostala zadeklarowana
                                 zewnetrznie
      …
}
int czarodziej()
{
…     // zmienna Hokus nie zostala zadeklarowana, wiec nie jest dostępna
}
```

W przykładzie 3 stworzono cztery różne zmienne. Zmienna `Hokus` w funkcji `main()` jest domyślnie jej zmienną lokalną, ale jest także zmienną automatyczną. Z kolei zmienna `Hokus` jest jawnie zadeklarowaną zmienną automatyczną w funkcji `czarodziej()` i jest dostępna tylko w jej obrębie. Zmienna zewnętrzna `Hokus` jest zatem niedostępna funkcjom `main()` i `czarodziej()`, ale może być dostępna dla każdej innej funkcji w danym pliku, która nie posiada własnej lokalnej zmiennej `Hokus`. Wreszcie mamy zmienną zewnętrzną `Pokus`, która jest dostępna w obrębie funkcji `czarodziej()`, ale wciąż jest niedostępna zadeklarowanej po niej funkcji `main()`.

```
/* Przyklad 3 */
int Hokus;
int czarodziej();
int main(void)
{
      int Hokus;      // zmienna Hokus domyslnie automatyczna.
      …
}
int Pokus;
int czarodziej()
{
      auto int Hokus; // zmienna automatyczna lokalna Hokus
      …
}
```

Powyższe przykłady ilustrują zasięg zmiennych zewnętrznych: od miejsca ich deklaracji do końca pliku. Ponadto pokazują także czas trwania takich zmiennych. Zmienne zewnętrzne `Hokus` i `Pokus` istnieją tak długo, jak długo działa program, a ponieważ nie są ograniczone do żadnej funkcji, więc nie znikają, gdy te kończą swoje działanie.

### Inicjalizacja zmiennych zewnętrznych

Podobnie jak w przypadku zmiennych automatycznych, zmiennym zewnętrznym można przypisać wartość podczas deklaracji. W przeciwieństwie jednak do nich, zmienne zewnętrzne są domyślnie inicjalizowane wartością 0, o ile nie podasz jawnie innej wartości. Dotyczy to również elementów zewnętrznie zdefiniowanych tabel. Taka inicjalizacja może zawierać jednak wyłącznie wyrażenia stałe, inaczej niż w przypadku zmiennych automatycznych.

```
int x = 10;                  // ok, 10 jest wartoscia stala
int y = 3 + 20               // ok, wyrazenie jest stale
size_t z = sizeof(int);      // ok, wyrazenie jest stale
int x2 = 2 * x;              // zle, x jest zmienna
```

(Zwróć uwagę na fakt, że tak długo jak typ nie jest zmienną tablicową, wyrażenie sizeof jest traktowane jak wyrażenie stałe).

### Używanie zmiennych zewnętrznych

Spójrzmy na prosty przykład korzystający ze zmiennej zewnętrznej. Załóżmy, że chcemy, aby dwie funkcje, nazwijmy je `main()` i `krytyka()` miały dostęp do zmiennej `jednostki`. Możemy tego dokonać deklarując zmienną `jednostki` poza i przed tymi funkcjami, jak pokazano na listingu 12.4.

**LISTING 12.4.** Program global.c

```
/* global.c  -- uzycie zmiennych globalnych */
#include <stdio.h>
int jednostki = 0;         /* zmienna zewnetrzna      */
void krytyka(void);
int main(void)
{
    extern int jednostki;  /* powtorna (opcjonalna) deklaracja */
    printf("Ile funtow masla miesci sie w barylce?\n");
    scanf("%d", &jednostki);
    while ( jednostki != 56)
        krytyka();
    printf("Musiales podejrzec!\n");

    return 0;
}
void krytyka(void)
{
    /* pominieto powtorna (opcjonalna) deklaracje */

    printf("Nie masz szczescia, sprobuj ponownie.\n");
    scanf("%d", &jednostki);
}
```

Oto fragment wyniku wyświetlanego przez ten program:

```
Ile jaj ma kopa?
14
Nie masz szczescia, sprobuj ponownie.
56
Musiales podejrzec!
```

(Właśnie to zrobiliśmy).

Zwróć uwagę, że choć druga wartość zmiennej `jednostki` została odczytana przez funkcję `krytyka()`, to funkcja `main()` również miała dostęp do tej nowej wartości w momencie zakończenia pętli `while`. Tak więc obie funkcje `main()` i `krytyka()` korzystały z tej samej zmiennej `jednostki`. W terminologii języka C powiedzielibyśmy, że zmienna `jednostki` ma zasięg plikowy, łączność zewnętrzną i statyczny czas trwania.

Uczyniliśmy zmienną `jednostki` zmienną zewnętrzną poprzez zdefiniowanie jej poza (tzn. na zewnątrz) jakąkolwiek funkcją. I to wszystko czego potrzeba, by udostępnić zmienną `jednostki` wszystkim zdefiniowanym w dalszej części danego pliku funkcjom.

Warto przyjrzeć się szczegółom. Po pierwsze, deklaracja zmiennej `jednostki` w miejscu, w którym została ona zadeklarowana czyni ją dostępną dla późniejszych funkcji bez żadnych dodatkowych działań. Stąd funkcja `krytyka()` może korzystać ze zmiennej `jednostki`.

Podobnie, nie trzeba niczego więcej, by funkcja `main()` miała dostęp do zmiennej `jednostki`. Pomimo to funkcja `main()` zawiera następującą deklarację:

```
extern int jednostki;
```

W tym przykładzie, deklaracja taka ma znaczenie wyłącznie dokumentacyjne. Specyfikator klasy zmiennych `extern` informuje kompilator, że każda wzmianka o zmiennej `jednostki` w tej określonej funkcji odnosi się do zmiennej zdefiniowanej gdzieś poza tą funkcją, a być może nawet i w innym pliku. Ponownie, obie funkcje `main()` i `krytyka()` korzystają ze zdefiniowanej zewnętrznie zmiennej `jednostki`.

## Nazwy zmiennych zewnętrznych

Standard C99 wymaga, by kompilator rozróżniał pierwsze 63 znaki nazw zmiennych lokalnych i pierwszych 31 znaków zmiennych zewnętrznych. To zmiana w stosunku do poprzednich wymogów, mówiących o rozróżnianiu pierwszych 31 znaków dla zmiennych lokalnych i sześciu dla zewnętrznych. Ponieważ standard C99 jest stosunkowo młody, możliwe, że dysponujesz kompilatorem działającym jeszcze zgodnie ze starszymi regułami. Przyczyna, dla której reguły nazewnictwa zmiennych zewnętrznych są bardziej restrykcyjne niż zmiennych lokalnych, tkwi w tym, że nazwy zewnętrzne muszą być zgodne z ograniczeniami narzucanymi przez lokalne środowisko.

### Definicje a deklaracje

Poświęćmy dłuższą chwilę, by przyjrzeć się różnicom pomiędzy definiowaniem zmiennej a jej deklarowaniem. Rozważmy następujący przykład:

```
int tern = 1;                // definicja zmiennej tern
main()
{
      external int tern;     // uzycie zmiennej tern zdefiniowanej gdzie
                             // indziej
```

Zmienna `tern` jest zadeklarowana dwa razy. Pierwsza z deklaracji rezerwuje pamięć dla tej zmiennej. Stanowi więc jej definicję. Druga deklaracja informuje kompilator wyłącznie o tym, że ma on użyć już istniejącej zmiennej `tern`; nie jest to zatem definicja. Pierwsza deklaracja określana jest *deklaracją definiującą* (*ang. defining declaration*), albo po prostu *definicją*, druga zaś nazywana jest *deklaracją nawiązującą* (*ang. referencing declaration*). Słowo kluczowe `extern` wskazuje, że deklaracja nie jest *definicją* zmiennej, gdyż instruuje ona kompilator, by szukał definicji gdzie indziej.

Jeśli napiszemy następujący kod:

```
extern int tern;
main()
{
```

to kompilator założy, że definicja zmiennej `tern` znajduje się w innym miejscu danego programu (być może w innym pliku). Deklaracja taka nie powoduje jednak przypisania (zajęcia) pamięci. Dlatego też, nie używaj słowa `extern`, gdy chcesz stworzyć definicję zewnętrzną; stosuj je tylko wtedy, gdy zmienna deklarowana *nawiązuje* do już istniejącej definicji zmiennej zewnętrznej.

Zmienna zewnętrzna może być zainicjalizowana wyłącznie jeden raz i to tylko w definicji. Instrukcja:

```
extern char dozwol = 'Y';     // blad
```

jest błędem, gdyż obecność słowa `extern` oznacza deklarację nawiązującą do danej zmiennej, a nie jej definicję.

## Zmienne statyczne o łączności wewnętrznej

Zmienne należące do tej klasy mają statyczny czas trwania, zasięg plikowy i łączność wewnętrzną. Tworzymy je, definiując zmienną poza kodem którejkolwiek z funkcji (czyli tak jak zmienne zewnętrzne) i ze specyfikatorem klasy pamięci `static`:

```
static int stwewl = 1;        // zmienna statyczna o lacznosci wewnetrznej
int main(void)
{
```

Takie zmienne nazywano kiedyś *statycznymi zmiennymi zewnętrznymi* (*ang. external static*), co było nieco mylące, gdyż mają one łączność wewnętrzną. Niestety żaden nowy, krótki termin nie zastąpił sformułowania *zewnętrzny statyczny*, tak więc pozostajemy przy określeniu *statyczna zmienna o łączności wewnętrznej.* Różnica jest taka,

że zwykła zmienna zewnętrzna może być używana przez funkcje znajdujące się w dowolnym pliku danego programu, natomiast zmienna statyczna o łączności wewnętrznej może być wykorzystywana wyłącznie przez funkcje znajdujące się w tym samym pliku. Można powtórzyć deklarację zmiennej o zasięgu plikowym wewnątrz funkcji wykorzystując specyfikator klasy pamięci `extern`. Taka deklaracja nie wpływa jednak na typ łączności. Rozważmy następujący przykład:

```
int podroz = 1;         // lacznosc zewnetrzna
static int domek = 1;   // lacznosc wewnetrzna
int main()
{
   extern int podroz;   // uzycie zmiennej globalnej podroz
   extern int domek;    // uzycie zmiennej globalnej domek
```

Zarówno zmienna `podroz`, jak i zmienna `domek` są zmiennymi globalnymi w obrębie danego pliku. Ale tylko zmienna `podroz` może być używana w pozostałych plikach programu. W obu deklaracjach widnieje słowo `extern`, by utrwalić fakt, że funkcja `main()` korzysta ze zmiennych globalnych. Zmienna `domek` wciąż ma łączność wewnętrzną.

## Programy wieloplikowe

Różnica między łącznością wewnętrzną a zewnętrzną jest istotna tylko wówczas, gdy mamy program złożony z wielu plików. Przyjrzyjmy się więc pokrótce temu zagadnieniu.

Złożone programy napisane w języku C często składają z wielu plików źródłowych. Czasami muszą one współdzielić zmienne zewnętrzne. Standard ANSI C proponuje zdefiniować zmienną w jednym pliku, a w pozostałych korzystać z deklaracji ze słowem `extern`. Tak więc, wszystkie deklaracje z wyjątkiem deklaracji definiującej, powinny używać tego słowa kluczowego, i tylko deklaracja definiująca może zostać wykorzystana do inicjalizacji zmiennej.

Zwróć uwagę, że zmienna zewnętrzna zdefiniowana w jednym pliku jest niedostępna w innych, jeśli nie zostanie w nich zadeklarowana (za pomocą słowa `extern`). Zewnętrzna deklaracja sama przez się czyni zmienną jedynie potencjalnie dostępną w innych plikach.

Historycznie rzecz biorąc, wiele kompilatorów postępowało zgodnie z innymi regułami niż podane tutaj. Dla przykładu, liczne systemy uniksowe zezwalają na deklarowanie zmiennej w wielu plikach bez użycia słowa kluczowego `extern`, pod warunkiem, że w co najwyżej jednej deklaracji występuje inicjalizacja. Deklaracja połączona z inicjalizacją, o ile wystąpi, jest traktowana jako definicja funkcji.

## Specyfikatory klasy zmiennych

Zapewne zauważyłeś, że znaczenie słów kluczowych `static` i `extern` zależy od kontekstu. Język C posiada pięć słów kluczowych, które razem określane są jako *specyfikatory klasy zmiennych*. Należą do nich: `auto`, `register`, `static`, `extern` i `typedef`. Co prawda słowo kluczowe `typedef` niewiele (a w zasadzie nic) mówi o klasie zmiennych, ale znalazło się tutaj ze względów syntaktycznych, czyli z uwagi na składnię języka C. Nie możesz użyć więcej niż jednego specyfikatora klasy zmiennej w deklaracji (lub definicji) zmiennej, a co z tego wynika, nie możesz także zastosować żadnego z pozostałych specyfikatorów klasy zmiennych jako części wyrażenia `typedef`.

Specyfikator `auto` oznacza zmienną z automatycznym czasem trwania. Może być używany wyłącznie przy deklaracji zmiennej o zasięgu blokowym, które już mają automatyczny czas trwania, zatem jego główne znaczenie polega na udokumentowaniu intencji autora programu.

Specyfikator `register` również może być używany wyłącznie ze zmiennymi o zasięgu blokowym. Umieszcza on zmienną w rejestrze (o ile jest taka możliwość), by skrócić czas dostępu do niej. Uniemożliwia to jednak pobranie adresu zmiennej.

Specyfikator `static`, użyty w deklaracji zmiennej o zasięgu blokowym, nadaje jej statyczny czas trwania, tak więc istnieje ona i utrzymuje swoją wartość tak długo, jak długo działa program, nawet wówczas, gdy zawierający ją blok nie jest wykonywany. Zmienna zachowuje zasięg blokowy i brak łączności. Jeśli natomiast specyfikatora `static` użyjesz w deklaracji zmiennej o zasięgu plikowym, będzie mieć ona łączność wewnętrzną.

Specyfikator `extern` oznacza, że deklarowana przez Ciebie zmienna została już wcześniej zdefiniowana w innym miejscu programu. Jeżeli deklaracja ze słowem `extern` ma zasięg plikowy, to zmienna, do której się odwołuje, musi mieć łączność zewnętrzną. Jeżeli natomiast deklaracja zawierająca słowo `extern` ma zasięg blokowy, zmienna do której się odwołuje może mieć łączność albo zewnętrzną, albo wewnętrzną, w zależności od definicji tejże zmiennej.

### Podsumowanie: klasy zmiennych

Zmienne automatyczne mają zasięg blokowy, brak łączności, automatyczny czas trwania. Są lokalne i prywatne dla danego bloku (zwykle jakiejś funkcji), w którym zostały zdefiniowane. Zmienne rejestrowe mają te same właściwości co zmienne automatyczne, ale kompilator może do ich przechowywania użyć szybkiej pamięci albo rejestru. Nie można jednak pobrać adresu takiej zmiennej.

Zmienne o statycznym czasie trwania mogą mieć łączność zewnętrzną, wewnętrzną albo brak łączności. Kiedy zmienna jest zadeklarowana jako zewnętrzna w stosunku do każdej z funkcji w jakimś pliku, jest zmienną zewnętrzną i ma zasięg plikowy, łączność zewnętrzną i statyczny czas trwania. Jeśli dodać słowo kluczowe

`static` do takiej deklaracji, otrzymamy zmienną o statycznym czasie trwania, zasięgu plikowym i łączności wewnętrznej. Jeżeli deklarujemy zmienną wewnątrz funkcji i użyjemy słowa kluczowego `static`, zmienna ta ma statyczny czas trwania, zasięg blokowy i brak łączności.

Pamięć dla zmiennych o automatycznym czasie trwania jest przydzielana, gdy wykonywany program wchodzi do bloku zawierającego deklaracje takich zmiennych, a następnie zwalniana z chwilą jego opuszczenia. Jeśli taka zmienna nie zostanie zainicjalizowana, to otrzyma wartość, znajdującą się uprzednio w przydzielanym jej miejscu pamięci. Pamięć dla zmiennych o statycznym czasie trwania jest alokowana podczas kompilacji i utrzymywana tak długo, jak długo działa program. Jeśli zmienna nie została zainicjalizowana, to przypisana jest jej wartość 0.

Zmienna o zasięgu blokowym jest lokalna w bloku zawierającym jej deklarację. Zmienna o zasięgu plikowym jest znana wszystkim funkcjom następującym po jej deklaracji w danym pliku. Jeśli zmienna o zasięgu plikowym ma łączność zewnętrzną, może być użyta przez inne pliki danego programu. Jeżeli zmienna o zasięgu plikowym ma łączność wewnętrzną, może być użyta tylko w ramach pliku, w którym została zadeklarowana.

Oto krótki program, który wykorzystuje wszystkie opisane wyżej klasy zmiennych. Jest on podzielony na dwa pliki (listing 12.5 i listing 12.6), dlatego wymaga przeprowadzenia kompilacji wielu plików. (Zobacz rozdział 9. lub pomoc kompilatora). Głównym celem przykładu jest użycie wszystkich pięciu klas zmiennych. Nie jest to wzorcowo zaprojektowany program; lepszy projekt nie miałby potrzeby korzystać ze zmiennych o zasięgu plikowym.

**LISTING 12.5.** Program czesca.c

```
// czesca.c --- rozne klasy zmiennych
#include <stdio.h>
void podaj_liczbe();
void sumuj(int k);
int liczba = 0;       // zasieg plikowy, lacznosc zewnetrzna
int main(void)
{
    int wartosc;       // zmienna automatyczna
    register int i;    // zmienna rejestrowa

    printf("Podaj dodatnia liczbe calkowita (0 to koniec): ");
    while (scanf("%d", &wartosc) == 1 && wartosc > 0)
    {
        ++count;       // zmienna o zasiegu plikowym
        for (i = wartosc; i >= 0; i--)
            sumuj(i);
        printf("Podaj dodatnia liczbe calkowita (0 to koniec): ");
    }
    podaj_liczbe();

    return 0;
}
```

```
void podaj_liczbe()
{
    printf("Petle opuszczono po %d cyklach\n", liczba);
}
```

**LISTING 12.6.** Program czescb.c

```
// czescb.c -- dalsza czesc programu
#include <stdio.h>
extern int liczba;       // deklaracja nawiazujaca, lacznosc zewnetrzna
static int suma = 0;     // definicja static, lacznosc wewnetrzna
void sumuj(int k);       // prototyp funkcji
void sumuj(int k)        // k ma zasieg blokowy i brak lacznosci
{
    static int podsuma = 0;  // statyczna, brak lacznosci

    if (k <= 0)
    {
        printf("Cykl petli: %d\n", liczba);
        printf("Podsuma: %d; Suma: %d\n", podsuma, suma);
        podsuma = 0;
    }
    else
    {
        podsuma += k;
        suma += k;
    }
}
```

W powyższym programie zmienna statyczna o zasięgu blokowym `podsuma` zawiera podsumę wartości przekazywanych do funkcji `sumuj()`, a zmienna `suma`, o zasięgu plikowym i łączności wewnętrznej, zawiera bieżącą sumę całości. Funkcja `sumuj()` wypisuje wartości `suma` i `podsuma` za każdym razem, gdy trafia do niej wartość niedodatnia. Funkcja wypisując wyniki, jednocześnie zeruje wartość zmiennej `podsuma`. Obecność prototypu funkcji `sumuj()` w pliku `czesca.c` jest obowiązkowa, gdyż w pliku tym są odwołania do tej funkcji. Z kolei w pliku `czescb.c` obecność prototypu jest już opcjonalna, gdyż plik ten definiuje funkcję `sumuj()`, choć nie jest ona w nim ani razu wywołana. Funkcja korzysta także ze zmiennej zewnętrznej `liczba`, która zlicza ile razy wykonano pętlę `while` w funkcji `main()`. Swoją drogą stanowi to świetny przykład na to, jak nie używać zmiennych zewnętrznych. W naszym programie niepotrzebnie i sztucznie splata się w ten sposób kody z plików `czesca.c` i `czescb.c`. W pliku `czesca.c` funkcje `main()` i `ile_razy()` mają wspólny dostęp do zmiennej `liczba`.

Oto przykładowy wynik działania programu:

```
Podaj dodatnia liczbe calkowita (0 to koniec): 5
Cykl petli: 1
Podsuma: 15; Suma: 15
Podaj dodatnia liczbe calkowita (0 to koniec): 10
Cykl petli: 2
Podsuma: 55; Suma: 70
```

```
Podaj dodatnia liczbe calkowita (0 to koniec): 2
Cykl petli: 3
Podsuma: 3; Suma: 73
Podaj dodatnia liczbe calkowita (0 to koniec): 0
Petla opuszczona po 3 cyklach
```

# Klasy zmiennych a funkcje

Funkcje w języku C również mają swoje klasy. Funkcja może być albo *zewnętrzna* (i tak jest domyślnie) albo *statyczna* (*ang. static*). (Standard C99 dodaje jeszcze trzecią możliwość, funkcje `inline`, omówione w rozdziale 16). Funkcja zewnętrzna jest dostępna w innych plikach, natomiast statyczna może być wywołana jedynie w obrębie pliku, w którym została zdefiniowana. Rozważmy następujący przykład:

```
double gamma();       /* funkcja domyslnie zewnetrzna */
static double beta();
extern double delta();
```

Funkcje `gamma`() i `delta`() mogą być używane przez funkcje w pozostałych plikach należących do danego programu. Funkcja `beta`() natomiast nie może być użyta w żadnym innym pliku. Skoro `beta`() jest przypisana do jednego pliku, w pozostałych możemy zdefiniować funkcje o tej samej nazwie i z nich korzystać. Jednym z powodów, dla których korzysta się ze *statycznej* klasy funkcji, jest możliwość stworzenia funkcji prywatnych dla określonego modułu, a przez to uniknięcia potencjalnego konfliktu nazw.

Przyjęło się używać słowa kluczowego `extern`, podczas deklarowania funkcji zdefiniowanych w innych plikach. Zwyczaj ten ma na celu głównie zwiększenie czytelności kodu, gdyż deklaracja funkcji i tak jest uważana domyślnie za `extern`, o ile nie zawiera słowa `static`.

## Którą klasę wybrać?

Odpowiedź na pytanie „którą klasę wybrać?" brzmi zwykle „automatyczną". Jest to zasadniczy powód, dla którego właśnie tę klasę uczyniono domyślną. Tak, wiemy, że na pierwszy rzut oka zmienne zewnętrzne wydają się atrakcyjne. Wystarczy tworzyć jedynie zmienne zewnętrze i już nie trzeba się przejmować używaniem argumentów i wskaźników do wymiany danych między funkcjami. W tym rozumowaniu kryje się jednak subtelna pułapka. Programista musiałby się zacząć martwić o to, czy jakaś funkcja `A()`, wbrew jego intencjom, przez przypadek nie zmodyfikuje zmiennych używanych w funkcji `B()`.

Długie lata wspólnych doświadczeń programistów na całym świecie dostarczają niepodważalnych dowodów, że to jedno subtelne niebezpieczeństwo ma dużo większe znaczenie niż powierzchowna atrakcyjność niepohamowanego korzystania ze zmiennych klasy zewnętrznej.

Jedną ze najważniejszych zasad bezpiecznego programowania jest reguła minimalnej wiedzy, która mówi, że wewnętrzne mechanizmy funkcji powinny być tak ukryte, jak to tylko możliwe: na zewnątrz widoczne powinny być tylko te zmienne, które koniecznie muszą być udostępnione. Inne klasy są użyteczne i są dostępne. Jednak zanim z nich skorzystasz, zastanów się czy naprawdę jest to konieczne.

# Funkcje pseudolosowe i zmienne statyczne

Teraz gdy znamy już podstawy dotyczące różnych klas zmiennych, przyjrzyjmy się kilku, korzystającym z nich, programom. Na początku, spójrzmy na funkcję, która używa zmiennej statycznej o łączności wewnętrznej: funkcję pseudolosową. Biblioteka ANSI C udostępnia funkcję `rand()`, generującą liczby pseudolosowe. Takich algorytmów jest wiele. Biblioteka pozwala poszczególnym implementacjom na dobór najlepszego dla konkretnej platformy systemowej. Standard ANSI C zawiera również przenośny algorytm standardowy, który generuje te same liczby pseudolosowe niezależnie od systemu. Funkcja `rand()` jest generatorem liczb pseudolosowych, co oznacza, że generowana sekwencja liczb jest przewidywalna (komputery nie są znane ze spontaniczności), z tym że liczby te są rozłożone równomiernie w ramach dostępnego przedziału wartości.

Zamiast użyć wbudowanej w kompilator funkcji `rand()`, skorzystamy z przenośnej wersji ANSI, tak by zobaczyć, co dzieje się w jej wnętrzu. Wszystko zaczyna się od liczby zwanej *ziarnem* (*ang. seed*), czyli wartości inicjującej generator. Funkcja korzysta z ziarna, by wygenerować nową liczbę, które staje się nowym ziarnem, które generuje kolejne ziarno, itd. Aby taki schemat mógł działać prawidłowo, funkcja pseudolosowa musi pamiętać ostatnio użyte ziarno. Ano właśnie! Aż prosi się o skorzystanie ze zmiennej statycznej. Listing 12.7 stanowi wersję nr 0 naszego programu. (Wersja nr 1 już niedługo).

**LISTING 12.7.** Program rand0.c

```
/* rand0.c -- generuje liczby losowe            */
/*      stosujac przenosny algorytm ANSI C   */
static unsigned long int nast = 1;  /* ziarno (ang. seed)  */
int rand0(void)
{
/* magiczna formula generujace liczby pseudolosowe */
     nast = nast * 1103515245 + 12345;
     return (unsigned int) (nast /65536) % 32768;
}
```

Na listingu 12.7 zmienna statyczna `nast` na początku przyjmuje wartość 1 i jest modyfikowana przez magiczną formułę przy każdym wywołaniu funkcji `rand0()`. W wyniku zwracana jest wartość z przedziału od 0 do 32 767. Zauważ, że `nast` jest zmienną statyczną o łączności wewnętrznej; a nie zmienną statyczną z brakiem łączności. Jest tak, ponieważ rozbudujemy potem nasz przykład tak, że będzie ona współdzielona przez dwie funkcje w tym samym pliku.

Sprawdźmy funkcję `rand0()` z prostym programem testowym pokazanym na listingu 12.8.

**LISTING 12.8.** Program r_test0.c

```
/* r_test0.c -- sprawdza funkcje rand0()  */
/* nalezy kompilowac z plikiem rand0.c    */
#include <stdio.h>
extern int rand0(void);
int main(void)
{
    int liczba;
    for (liczba = 0; liczba < 5; liczba++)
        printf("%hd\n", rand0());

    return 0;
}
```

Znów mamy okazję, by poćwiczyć kompilację wielu plików. Zawartość listingu 12.7 wstawmy do jednego pliku, a listingu 12.8 do drugiego. Słowo kluczowe `extern` wskazuje, że funkcja `rand0()` została zdefiniowana w innym pliku.

Wynik działania programu jest następujący:

```
16838
5758
10113
17515
31051
```

Wynik wygląda na zbiór przypadkowych (losowych) liczb. Uruchommy go zatem ponownie. Tym razem wynik jest następujący:

```
16838
5758
10113
17515
31051
```

Hmmm, wygląda to znajomo; oto czynnik „pseudo" wyrażenia „pseudolosowy". Za każdym razem, gdy program jest uruchamiany, rozpoczyna od tego samego ziarna o wartości 1.

Możemy obejść ten problem tworząc funkcję o nazwie `srand1()`, która pozwoli nadać ziarnu wartość 0. Cała sztuka w tym, by uczynić zmienną `nast` zmienną statyczną o łączności wewnętrznej dostępną tylko funkcjom `rand1()` i `srand1()`. (Funkcji `srand1()` w bibliotece języka C odpowiada funkcja `srand()`). Dodajmy zatem funkcję `srand1()` do pliku zawierającego `rand1()`. Listing 12.9 zawiera tę modyfikację.

**LISTING 12.9.** Program s_i_r.c

```
/* s_i_r.c -- plik z funkcjami rand1() i srand1()    */
/*    uzywa przenosnego algorytmu ANSI C */
static unsigned long int nast = 1;  /* ziarno (ang. seed)  */
int rand1(void)
{
/* magiczna formula generujaca liczby pseudolosowe */
    nast = nast * 1103515245 + 12345;
    return (unsigned int) (next/65536) % 32768;
}
void srand1(unsigned int ziarno)
{
    nast = ziarno;
}
```

Zauważmy, że `nast` jest zmienną statyczną o zasięgu plikowym i łączności wewnętrznej. Oznacza to, że może być użyta zarówno przez funkcję `rand1()` jak i `srand1()`, lecz nie może być wykorzystywana w innych plikach. Aby sprawdzić działanie tych funkcji, użyjmy programu testującego z listingu 12.10.

**LISTING 12.10.** Program r_test1.c

```
/* r_test1.c -- sprawdza funkcje rand1() i srand1() */
/* nalezy kompilowac z plikiem s_i_r.c       */
#include <stdio.h>
extern void srand1(unsigned int x);
extern int rand1(void);
int main(void)
{
    int liczba;
    unsigned ziarno;
    printf("Podaj wartosc ziarna:\n");
    while (scanf("%u", &ziarno) == 1)
    {
        srand1(ziarno);    /* reset ziarna */
        for (liczba = 0; liczba < 5; liczba++)
            printf("%hd\n", rand1());
        printf("Podaj nastepna wartosc ziarna (k to koniec):\n");
    }
    printf("Koniec\n");
     return 0;
}
```

Ponownie skorzystajmy z obu plików i uruchommy program.

```
Podaj wartosc ziarna:
1
16838
5758
10113
17515
31051
Podaj nastepna wartosc ziarna (k to koniec):
```

```
513
20067
23475
8955
20841
15324
Podaj nastepna wartosc ziarna (k to koniec):
k
Koniec
```

Dla wartości ziarna równej 1 otrzymujemy taki sam wynik jak wcześniej, ale już ziarno o wartości 3 generuje inny zbiór liczb.

### Automatyczna zmiana ziarna

Jeżeli nasza implementacja języka C pozwala na skorzystanie z jakichś zmiennych czynników zewnętrznych, takich jak zegar systemowy, możemy wykorzystać taką wartość (być może odpowiednio przekształconą) do inicjalizowania wartości ziarna. Na przykład, standard ANSI C zawiera funkcję `time()`, która zwraca czas systemowy. Jednostki miary czasu zależą od konkretnego systemu, ale w tym kontekście istotne jest tylko to, że zwracana jest wartość o typie liczbowym i że zmienia się ona wraz z upływem czasu. Dokładny typ zwracanej wartości zależy od systemu i jest określony etykietą `time_t`, ale możemy użyć rzutowania typów. Oto prosty przykład:

```
#include <time.h>        /* prototyp funkcji time() z ANSI C */
srand1((unsigned int) time(0));    /* inicjalizowanie ziarna */
```

Ogólnie funkcja `time()` pobiera argument będący adresem obiektu o typie `time_t`. W takim przypadku wartość czasu jest także przechowywana pod tym adresem. Można również przekazać do funkcji jako argument wskaźnik zerowy (0). Wtedy wartość jest przekazywana wyłącznie poprzez mechanizm wartości zwracanej.

Tę samą metodę możesz zastosować w przypadku standardowych funkcji ANSI C: `srand()` i `rand()`. Aby ich użyć, musisz załączyć plik nagłówkowy `stdlib.h`. Teraz, gdy już wiemy jak działają funkcje `srand1()` i `rand1()` korzystające ze zmiennych statycznych o łączności wewnętrznej, możemy korzystać z ich odpowiedników z biblioteki standardowej. Spróbujmy na poniższym przykładzie.

# Rzut kostką

W tym podrozdziale spróbujemy zasymulować bardzo popularną czynność o charakterze losowym: rzut kostką. Najpowszechniejsza jest kostka sześciościenna, choć istnieją też inne. W grach fabularnych typu RPG, korzysta się na przykład ze wszystkich pięciu geometrycznie dopuszczalnie kości: 4, 6, 8, 12 i 20- ściennych.

Starożytni Grecy dowodzili, że istnieje dokładnie pięć brył foremnych, czyli mających określoną liczbę, jednakowych co do kształtu i pól, ścian. Oczywiście kostki mogą mieć również inną liczbę ścian, lecz szanse wylosowania każdej z nich byłyby wówczas nierówne.

Komputer nie jest ograniczony zasadami geometrii, stąd też możemy obmyślić elektroniczną kostkę o właściwie dowolnej liczbie ścian. Zacznijmy jednak od sześciu, a potem przykład uogólnimy.

Chcemy otrzymać losową wartość z przedziału od 1 do 6. Funkcja `rand()` generuje liczby całkowite z przedziału od 0 do `RAND_MAX`; `RAND_MAX` jest zdefiniowane w pliku `stdlib.h`. Jest zwykle równe wartości `INT_MAX`, czyli największej liczbie całkowitej. Dlatego też, musimy przeprowadzić kilka modyfikacji. Oto jedna z możliwości:

1. Wyliczamy resztę z dzielenia przez 6 liczby pseudolosowej (czyli wykonujemy działanie modulo 6). Da nam to w wyniku liczbę całkowitą z przedziału od 0 do 5.
2. Dodajmy 1. Nowa wartość będzie z przedziału od 1 do 6.
3. Aby proces uogólnić, po prostu zastąpmy liczbę 6 w kroku 1 wymaganą liczbą ścian.

Oto program, który realizuje przedstawiony wyżej pomysł:

```
#include <stdlib.h> /* dla rand() */
int rzucaj(int scianki)
{
      int rzut;
      rzut = rand() % scianki + 1;
      return rzut;
}
```

Będziemy jednak ambitniejsi i stwórzmy funkcję, która pozwoli na rzut dowolną liczbą kostek i zwróci sumę wyrzuconych oczek. Prezentuje to listing 12.11.

**LISTING 12.11.** Program rzutkosc.c

```
/* rzutkosc.c -- symulacja rzutu koscmi*/
#include "rzutkosc.h"
#include <stdio.h>
#include <stdlib.h>              /* potrzebujemy funkcji rand()   */
int liczba_rzutow  = 0;          /* lacznosc zewnetrzna */
static int rzucaj(int scianki)   /* prywatne w ramach pliku*/
{
    int oczka;
    oczka = rand() % scianki + 1;
    ++liczba_rzutow;             /* zlicza wywolania funkcji*/

    return oczka;
}
int rzucaj_n_razy(int rzuty, int scianki)
{
    int k;
    int suma = 0;
    if (scianki < 2)
    {
        printf("Wymagane sa co najmniej 2 scianki.\n");
        return -2;
    }
    if (rzuty < 1)
```

```
    {
        printf("Wymagany co najmniej 1 rzut.\n");
        return -1;
    }

    for (k = 0; k < rzuty; k++)
        suma += rzucaj(scianki);

    return suma;
}
```

W powyższym pliku użyliśmy kilku sztuczek. Po pierwsze, zmieniliśmy funkcję `rzucaj()` w funkcję prywatną dla tego pliku. Służy ona jako funkcja pomocnicza dla funkcji `rzucaj_n_razy()`. Po drugie, aby zilustrować jak działa łączność zewnętrzna, w pliku zadeklarowaliśmy zmienną zewnętrzną `liczba_rzutow`, która przechowuje liczbę wywołań funkcji `rzucaj()`. Przykład jest nieco sztuczny, ale pokazuje za to cechy zmiennych zewnętrznych. Po trzecie wreszcie, plik zawiera następującą instrukcję:

```
#include "rzutkosc.h"
```

Kiedy korzystamy z funkcji biblioteki standardowej, takich jak `rand()`, załączamy plik nagłówkowy (`stdlib.h dla rand()`) zamiast deklarować funkcję. Dzieje się tak dlatego, że plik nagłówkowy zawiera już odpowiednie poprawne deklaracje.

W naszym programie naśladujemy tę praktykę przez załączenie pliku `rzutkosc.h`, by skorzystać z funkcji `rzucaj_n_razy()`. Umieszczenie nazwy pliku w znakach cudzysłowu, zamiast standardowych < i >, instruuje kompilator, by szukał pliku lokalnie, a nie w miejscach, w których szuka standardowych plików nagłówkowych. Przy czym znaczenie słowa „lokalnie" zależy już od konkretnej implementacji. Zazwyczaj jest przez to rozumiany katalog albo folder, w którym znajduje się plik źródłowy, albo katalog, w którym znajduje się plik projektu (jeśli kompilator go używa). Listing 12.12 przedstawia zawartość pliku nagłówkowego `rzutkosc.h`.

**LISTING 12.12.** Program rzutkosc.c

```
// rzutkosc.h
extern int liczba_rzutow;
int rzucaj_n_razy(int rzuty, int scianki);
```

Plik zawiera prototypy funkcji i deklarację typu `extern`. Plik `rzutkosc.c`, który załącza ten nagłówek, posiada w zasadzie dwie deklaracje zmiennej `liczba_rzutow`.

```
extern int liczba_rzutow;            // z pliku naglowkowego
int liczba_rzutow = 0;               // z pliku zrodlowego
```

Jest to poprawne. Zmienną można co prawda zdefiniować tylko raz, ale deklarację nawiązującą ze słowem kluczowym `extern` można umieścić w programie dowolną ilość razy.

Program wywołujący funkcję `rzucaj_n_razy ()` powinien zawierać odpowiedni plik nagłówkowy.

Nie tylko udostępnia on prototyp dla funkcji `rzucaj_n_razy()`, ale również zmienną `liczba_rzutow`. Listing 12.13 ilustruje to zagadnienie.

**LISTING 12.13.** Program wielerzut.c

```
/* wielrzut.c -- wielokrotny rzut koscmi                  */
/* kompilowac razem z rzutkosc.c                          */
#include <stdio.h>
#include <stdlib.h>              /* potrzeba funkcji srand() */
#include <time.h>                /* potrzeba funkcji time()  */
#include "rzutkosc.h"            /* potrzeba funkcji rzucaj_n_razy()   */
                                 /* i zmiennej liczba_rzutow */
int main(void)
{
  int rzuty, wynik;
  int scianki;
  srand((unsigned int) time(0)); /* losowe ziarno      */
  printf("Podaj liczbe scianek, 0 oznacza koniec.\n");
  while (scanf("%d", &scianki) == 1 && scianki > 0)
  {
      printf("Ile rzutow?\n");
      scanf("%d", &rzuty);
      wynik = rzucaj_n_razy(rzuty, scianki);
      printf("Wyrzucono razem %d uzywajac %d %d-sciennych kosci.\n",
              wynik, rzuty, scianki);
      printf("Podaj liczbe scianek, 0 oznacza koniec.\n");
  }
  printf("Funkcja rzucaj() zostala wywolana %d razy.\n",
          liczba_rzutow);           /* zmienna zewnetrzna */

  printf("ZYCZE DUZO SZCZESCIA!\n");

  return 0;
}
```

Skompilujmy listing 12.13 z plikiem zawierającym listing 12.11. Dla ułatwienia sprawy umieśćmy listingi 12.11, 12.12 i 12.13 w tym samym katalogu. Uruchommy otrzymany program. Rezultat powinien wyglądać mniej więcej tak:

```
Podaj liczbe scianek, 0 oznacza koniec.
6
Ile rzutow?
2
Wyrzucono razem 12 uzywajac 2 6-sciennych kosci.
Podaj liczbe scianek, 0 oznacza koniec.
6
Ile rzutow?
2
Wyrzucono razem 4 uzywajac 2 6-sciennych kosci.
Podaj liczbe scianek, 0 oznacza koniec.
6
Ile rzutow?
```

```
2
Wyrzucono razem 5 uzywajac 2 6-sciennych kosci.
Podaj liczbe scianek, 0 oznacza koniec.
0
Funkcja rzucaj() zostala wywolana 6 razy.
ZYCZE DUZO SZCZESCIA!
```

Ponieważ program używa funkcji `srand()`, by wylosować ziarno pseudolosowe, nie powinieneś otrzymać takich samych wyników, nawet podając taką samą liczbę rzutów. Zauważ, że funkcja `main()` w pliku `wielrzut.c` nie ma dostępu do zmiennej `liczba_rzutow` zdefiniowanej w pliku `rzutkosc.c`.

Możemy użyć funkcji `rzucaj_n_razy()` na wiele sposobów. Dla zmiennej `scianki` równej 2 program symuluje rzut monetą: 1 to reszka, 2 to orzeł (albo odwrotnie, w zależności od twojego upodobania). Możesz łatwo zmodyfikować ten program, by wyświetlał kolejne wyniki, a nie tylko ich sumę, możesz też stworzyć prawdziwy symulator gry w kości. Jeśli potrzebujesz wielu rzutów, tak jak w niektórych grach RPG, możesz w prosty sposób zmodyfikować program, aby wyświetlał wyniki tak jak poniżej:

```
Podaj liczbe kolejek, k oznacza koniec.
18
Podaj liczbe scianek i liczbe kostek.
6 3
Oto 18 kolejek 3 6-sciennych rzutow.
      12      10      6       9       8       14      8       15      9
      14      12      17      11      7       10      13      8       14
Podaj liczbe kolejek, k oznacza koniec.
k
```

Jeszcze innym zastosowaniem dla funkcji `rand1()` i `rand()` (choć nie dla `rzucaj()`) może być zabawa w zgadywanie pomyślanej liczby. Komputer „pomyśli sobie", czyli wylosuje liczbę, a użytkownik zgaduje. Spróbuj samemu napisać taki program.

# Przydział pamięci: funkcje malloc() i free()

Opisanych pięć klas zmiennych ma jedną cechą wspólną. Kiedy już podejmiemy decyzję, której klasy użyć, decyzje o zasięgu i czasie trwania podejmowane są automatycznie. Programista ma jednak do dyspozycji narzędzia w postaci funkcji bibliotecznych do przydzielania i zarządzania pamięcią, które dają mu większą swobodę działania.

Na początek, przyjrzyjmy się pewnych kwestiom ogólnym, dotyczącym przydziału pamięci. Każdemu programowi należy zapewnić wystarczającą ilość pamięci na dane, z których korzysta. Część przydziałów pamięci dokonywana jest automatycznie. Na przykład, gdy zadeklarujesz:

```
float x;
char miejsce[] = "Jakies tam miejsce";
```

to obu zmiennym zostanie przydzielona pamięć określona przez ich typy. Można określić również jawnie ilość wymaganej pamięci:

```
int talerze[100];
```

Powyższa deklaracja przydziela 100 komórek pamięci, z których każda przechowuje wartość typu `int`. W podanych przypadkach deklaracja zawiera również identyfikator pamięci, dzięki czemu można odwoływać się do danych poprzez nazwy `x` czy `miejsce`.

Język C pozwala pójść jeszcze dalej. Możemy przydzielać pamięć także podczas działania programu. Podstawowym narzędziem jest funkcja `malloc`(), która wymaga tylko jednego argumentu: wymaganej liczby bajtów pamięci. Następnie `malloc`() szuka bloków wolnej pamięci o odpowiedniej wielkości. Pamięć pozostaje anonimowa; `malloc`() przydziela pamięć, nie przypisując jej nazwy. Jednakże funkcja zwraca adres pierwszego bajtu przydzielonego bloku, który można przypisać wskaźnikowi. Ponieważ typ `char` zajmuje 1 bajt, `malloc`() tradycyjnie definiowany był jako wskaźnik do typu `char` (*ang. pointer-to-char*). Standard ANSI C używa do tego celu nowego typu: wskaźnika do typu `void` (*ang. pointer-to-void*). Typ ten powinien być traktowany jako wskaźnik ogólnego przeznaczenia. Pamięć przydzielona przez `malloc`() może być użyta jako tablica, struktura itp., więc zwykle zwracana wartość będzie rzutowana na właściwy typ docelowy. W ramach standardu ANSI C wciąż powinniśmy rzutować typy dla zachowania przejrzystości, choć przypisanie wskaźnika do `void` wskaźnikowi do innego typu nie spowoduje konfliktu typów. Jeśli funkcji `malloc`() nie uda się znaleźć wystarczająco dużego wolnego obszaru, zwróci wskaźnik zerowy.

Dla przykładu użyjmy funkcji `malloc`(), by stworzyć tablicę. Możemy wykorzystać `malloc`(), by zgłosić zapotrzebowanie na odpowiednio duży blok pamięci już w trakcie działania programu. Potrzebujemy ponadto wskaźnika, który przechowa informację o tym, gdzie w pamięci znajduje się przydzielony właśnie blok.

```
double * wsk;
wsk = (double *) malloc (30 * sizeof(double));
```

Powyższa instrukcja żąda przydziału pamięci dla 30 wartości typu `double`; zmienna `wsk` wskazuje na przydzielony obszar. Zauważ, że zmienna `wsk` jest zadeklarowana jako wskaźnik do pojedynczego typu `double`, a nie do bloku 30 wartości `double`. Jak zapewne pamiętasz, nazwa tablicy jest adresem pierwszego jej elementu. Zatem, jeśli zmienna `wsk` jest wskaźnikiem do pierwszego elementu tego bloku, możemy traktować ją na równi z tablicą. Dlatego właśnie, możemy stosować wyrażenie `wsk[0]`, by uzyskać dostęp do pierwszego elementu bloku, `wsk[1]` do drugiego itd. Widać zatem, że możesz stosować notację wskaźnikową dla nazw tablicowych oraz używać notacji tablicowej w przypadku wskaźników.

Znamy już trzy sposoby tworzenia tablic w języku C:

- Pierwszy to zadeklarować tablicę, stosując wyrażenia stałe dla rozmiarów tablic i odwoływać się do elementów tablicy za pomocą jej nazwy,

- Drugi to zadeklarować tablicę o zmiennym rozmiarze, stosując zmienne wyrażenia dla rozmiarów tablicy i odwoływać się do elementów tablicy za pomocą jej nazwy,
- Trzeci to zadeklarować wskaźnik, wywołać funkcję `malloc`() i odwoływać się do elementów tablicy za pomocą wskaźnika.

Dwa ostatnie sposoby pozwalają na coś niemożliwego przy użyciu zwyczajnej tablicy — na stworzenie dynamicznej tablicy (*ang. dynamic array*), pamięć dla której zostanie przydzielana w trakcie działania programu.i której rozmiar nie musi być znany podczas kompilacji. Załóżmy, na przykład, że `n` jest zmienną typu `int`. W standardach starszych od C99 nie mogliśmy użyć następującej deklaracji:

```
double elem[n]; /* niedozwolone, jeśli n jest zmienna w kompilatorach
sprzed standardu C99
```

Natomiast poniższa instrukcja jest poprawna nawet dla starszych kompilatorów:

```
wsk = (double *) malloc(n * sizeof(double)); /* ok */
```

Jak się za chwilę przekonamy, instrukcja ta nie tylko działa, ale przede wszystkim daje większe możliwości niż tablice o zmiennym rozmiarze.

Zazwyczaj, każdemu wystąpieniu funkcji `malloc`() powinna towarzyszyć funkcja `free`(), która pobiera jako argument adres zwrócony wcześniej przez `malloc`() i uwalnia przydzieloną pamięć. Tak więc czas trwania pamięci przydzielonej pamięci trwa od momentu wywołania funkcji `malloc()` do użycia funkcji `free()`, która zwalnia używany blok i umożliwia jego powtórne wykorzystanie . Powinniśmy traktować `malloc`() i `free`() jak funkcje zarządzania pulą pamięci. Każde wywołanie `malloc`() przydziela pamięć na potrzeby programu, a każde wywołanie `free`() zwalnia obszar i zwraca go do puli pamięci. Funkcja `free`() potrzebuje argumentu w postaci wskaźnika do bloku pamięci przydzielonej przez `malloc`(); nie można użyć funkcji `free`(), by zwolnić pamięć, przydzieloną w inny sposób, na przykład w zadeklarowanej tablicy. Prototypy obu funkcji znajdują się w pliku `stdlib.h`.

Stosując funkcję `malloc`() w programie, możemy na bieżąco decydować jaki rozmiar tablicy jest potrzebny i przydzielać pamięć dynamicznie. Ilustruje to listing 12.14. Program przypisuje adres bloku pamięci do wskaźnika `wsk`, a następnie używa go jak nazwy tablicy. Funkcja `exit`(), która ma swój prototyp także w pliku `stdlib.h`, wywoływana jest by przerwać pracę programu, jeśli przydział pamięci się nie powiedzie. Wartość `EXIT_FAILURE` zdefiniowana jest w tym samym pliku nagłówkowym. Standard definiuje dwie zwracane wartości, które są dostępne we wszystkich systemach operacyjnych: `EXIT_SUCCESS` (albo po prostu 0), oznaczającą zwyczajne zakończenie programu, oraz `EXIT_FAILURE` sygnalizującą niespodziewane przerwanie programu. Niektóre systemy operacyjne, w tym Unix, Linux i Windows, akceptują jeszcze pewne dodatkowe wartości.

**LISTING 12.14.** Program tab_dyn.c

```
/* tab_dyn.c -- dynamicznie konstruowanie tablic */
#include <stdio.h>
#include <stdlib.h> /* potrzebujemy funkcji: malloc() i free() */
int main(void)
{
    double * wsk;
    int max;
    int liczba;
    int i = 0;
puts("Podaj maksymalna liczbe elementow (typu double):");
scanf("%d", &max);
    wsk = (double *) malloc(max * sizeof (double));
    if (wsk == NULL)
    {
        puts("Przydzial pamieci nie powiodl sie.");
        exit(EXIT_FAILURE);
    }
    /* wsk wskazuje na tablice o liczbie elementow rownej max */
    puts("Podaj elementy (k to koniec):\n");
    while (i < max && scanf("%lf", &wsk[i]) == 1)
        ++i;
    printf("Oto %d wprowadzonych elementow:\n", liczba = i);
    for (i = 0; i < liczba; i++)
    {
        printf("%7.2f ", wsk[i]);
        if (i % 7 == 6)
            putchar('\n');
    }
    if (i % 7 != 0)
        putchar('\n');
    puts("Koniec.");
    free(wsk);
    return 0;
}
```

Oto przykładowy przebieg programu. Podaliśmy sześć liczb, lecz program przetworzył jedynie pięć z nich, gdyż ograniczyliśmy rozmiar tablicy do 5.

```
Podaj maksymalna liczbe elementow (typu double):
5
Podaj elementy (k to koniec):");
20 30 35 25 40 80
Oto 5 wprowadzonych elementow:
       20.00        30.00        35.00        25.00        40.00
80.00
Koniec.
```

Spójrzmy jeszcze na poniższy kod. Program pobiera rozmiar tablicy za pomocą instrukcji:

```
puts("Podaj maksymalna liczbe elementow (typu double):");
scanf("%d", &max);
```

Następnie, poniższa instrukcja przydziela ilość pamięci konieczną do przechowania wymaganej liczby elementów i przypisuje adres bloku wskaźnikowi `wsk`.

```
wsk = (double *) malloc(max * sizeof (double));
```

Rzutowanie typu do (`double *`) jest w języku C opcjonalne, natomiast wymagane w języku C++. Tak więc stosowanie rzutowania ułatwia przeniesienie kodu z C do C++.

Pamiętaj, że istnieje możliwość, iż funkcji `malloc`() nie uda się zarezerwować wymaganej ilości pamięci. Wówczas zwróci ona wskaźnik zerowy, a program zostanie przerwany:

```
if( wsk == NULL )
{
      puts("Przydzial pamieci nie powiodl sie.\n");
      exit(EXIT_FAILURE);
}
```

O ile program nie natrafi na ten problem, może potraktować zmienną `wsk` jak nazwę tablicy o liczbie elementów równej `max`, i tak też się dzieje.

Zwróć uwagę na funkcję `free`() znajdującą się pod koniec programu. Zwalnia ona pamięć przydzieloną przez `malloc`(). Zwalnia jednak tylko blok pamięci, na który wskazuje jej argument. W tym konkretnym przykładzie, wywołanie `free`() nie jest konieczne, gdyż cała przydzielona pamięć jest zwalniana automatycznie z chwilą zakończenia programu. W bardziej złożonych programach zwalnianie pamięci do ponownego wykorzystania jest bardzo ważne.

Co zyskujemy stosując tablice dynamiczne? Przede wszystkim, elastyczność. Załóżmy, że pisząc program wiesz, że w większości przypadków będzie on potrzebować nie więcej niż 100 elementów, ale czasami może potrzebować nawet 10 000. Jeżeli zadeklarujesz tablicę, to to musisz być przygotowany na najgorszy przypadek i zadeklarować ją dla 10 000 elementów. Zauważ, że przez większość czasu program będzie marnować sporo pamięci. Co gorsza, jeśli pewnego razu zajdzie potrzeba użycia 10 001 elementów to program nie zadziała. Rozwiązaniem wszystkich tych problemów jest użycie tablicy dynamicznej.

## Znaczenie funkcji free()

Wymaga ilość pamięci statycznej znana jest już podczas kompilacji i nie zmienia się w czasie działania programu. Ilość pamięci poświęconej zmiennym automatycznym rośnie i kurczy się automatycznie podczas pracy programu. Natomiast ilość pamięci przydzielonej funkcją `malloc`() rośnie, jeśli nie zwalniamy jej funkcją `free`(). Dla przykładu przypuśćmy, że mamy funkcję, która tworzy tymczasową kopię tablicy tak, jak w poniższym programie:

```
...
int main()
{
double zadowolony[2000];
int i;
…for(i = 0; i < 1000 ; i++)
zarlok(zadowolony, 2000);
…}
void zarlok(double tab[], int n)
{
      double * temp = (double *) malloc( n * sizeof(double) );
…     /* free(temp);          // zapomnielismy uzyc funkcji free() */
}
```

Kiedy funkcja `zarlok`() zostaje wywołana po raz pierwszy, tworzy wskaźnik `temp` i wywołuje `malloc`(), przydzielając 16 000 bajtów pamięci (przy założeniu, że typ `double` jest 8-bajtowy). Przypuśćmy, że ani razu nie wywołamy funkcji `free`().

Za każdym razem, gdy funkcja kończy swoje działanie, wskaźnik `temp`, będący zmienną automatyczną, znika. Jednak wskazywane przez niego 16 000 bajtów pamięci nie zostało zwolnione. Ze zniknięciem zmiennej `temp` niestety straciliśmy do nich dostęp, gdyż nie znamy już adresu. Pamięć nie może być ponownie wykorzystana, gdyż nie zwolniliśmy jej wywołaniem funkcji `free`().

Kiedy ponownie wywołamy funkcję `zarlok`(), ta na nowo stworzy zmienną `temp` i wywoła funkcję `malloc`(), aby znowu przydzielić 16 000 bajtów pamięci. Skoro jeden blok 16 000 bajtów nie jest już dostępny, to `malloc`() szuka innego. I ponownie, gdy funkcja skończy swoje działanie, tracimy dostęp do tego bloku pamięci i nie możemy go ponownie wykorzystać.

Załóżmy, że pętla wykona się 1000 razy. Zatem gdy się skończy, 16 000 000 bajtów pamięci zostanie usuniętych z puli pamięci. W rzeczywistości programowi może zabraknąć pamięci na długo przed teoretycznym ukończeniem pętli. Problemy tego typu nazywane są wyciekami pamięci (*ang. memory leak*), a można im zapobiec, stosując funkcję `free`().

## Funkcja calloc()

Innym sposobem dynamicznego przydziału pamięci jest funkcja `calloc()`. Jej wykorzystanie wygląda zazwyczaj następująco:

```
long * nowapam;
nowapam = (long *)calloc(100, sizeof(long));
```

Podobnie do funkcji `malloc`(), `calloc`() zwraca albo wskaźnik do typu `char` — w wersjach sprzed standardu ANSI C, albo wskaźnik do typu `void` — dla ANSI C. Jeżeli chcesz korzystać z innych typów zmiennych, musisz zastosować operator rzutowania. Funkcja `calloc`() przyjmuje dwa argumenty, które powinny być typu całkowitego bez znaku (`unsigned int` – `size_t` według ANSI). Pierwszym argumentem jest

wymagana ilość komórek pamięci. Drugi argument określa wielkość pojedynczej komórki w bajtach. W naszym przypadku, typ `long` ma 4 bajty, zatem powyższa instrukcja przydzieli 100 czterobajtowych komórek, co daje łącznie 400 bajtów.

Używamy `sizeof(long)` zamiast po prostu 4, aby program był przenośny. Dzięki temu będzie on działał również na systemach, w których typ `long` ma rozmiar inny niż 4 bajty.

Funkcja `calloc()` różni się od `malloc`() jeszcze tym, że przypisuje wszystkim bitom danego bloku pamięci wartość 0. (Zauważmy jednak, że na niektórych platformach sprzętowych, wartość zmiennoprzecinkowa 0 nie jest reprezentowana w postaci wszystkich bitów ustawionych na 0).

Pamięć przydzieloną przez `calloc`() zwalniamy także funkcją `free`().

Dynamiczny przydział pamięci jest podstawą wielu zaawansowanych technik programistycznych. Niektórym z nich przyjrzymy się w rozdziale 17.

Biblioteki różnych implementacji języka C mogą zawierać także dodatkowe funkcje do zarządzania pamięcią, niektóre przenośne, niektóre nie. Warto poświęcić trochę czasu i zapoznać się z nimi.

## Dynamiczny przydział pamięci a tablice o zmiennym rozmiarze

Tablice o zmiennym rozmiarze (VLA) i funkcja `malloc`() w sensie funkcjonalnym mają wiele cech wspólnych. Na przykład obie pozwalają tworzyć tablice o dynamicznie zmieniającym się rozmiarze.

```
int vlamal()
{
int n;
      int * pi;
      scanf("%d", &n);
      pi = (int *) malloc (n * sizeof(int));
      int tab[n];     // vla
      pi[2] = tab[2] = -5;
…
}
```

Jedną z różnic jest to, że VLA ma pamięć automatyczną. Jedną z konsekwencji zastosowania pamięci automatycznej jest fakt, że używana przez nią pamięć zwalniana jest automatycznie, gdy program wychodzi z bloku, w którym tablica została zdefiniowana — w powyższym przykładzie wtedy, gdy funkcja `vlamal`() skończy swoje działanie. Nie musimy więc pamiętać o funkcji `free`(). Z drugiej jednak strony, dostęp do tablicy stworzonej za pomocą funkcji `malloc`() nie jest ograniczony do pojedynczej funkcji. Jedna funkcja może stworzyć tablicę, a drugą, używając wskaźnika do tej tablicy, może ją modyfikować, a po zakończeniu pracy, zwolnić przydzieloną jej pamięć wywołując funkcję `free`(). Zmienne wskaźnikowe użyte w funkcjach `malloc`()

i free() mogą być różne, choć oczywiście wskazywany przez nie adres musi być w obu przypadkach ten sam.

VLA są wygodniejsze w użytkowaniu w przypadku tablic wielowymiarowych. Możemy oczywiście stworzyć tablicę dwuwymiarową używając funkcji malloc(), lecz składnia w tym przypadku jest mniej przejrzysta. Jeżeli nasz kompilator nie pozwala używać tablic o zmiennym rozmiarze, to jeden z wymiarów musi być stałą, jak w poniższym kodzie:

```
int n = 5;
int m = 6;
int tab2[n][m];       // n x m VLA
int (* p2) [6];       // przed standardem C99
int (* p3)[m];        // wymaga obslugi VLA
p2 = (int (*)[6]) malloc(n * 6 * sizeof(int) ); // tablica o rozmiarze
                                                // n * 6
p3 = (int (*)[m]) malloc(n * m * sizeof(int) ); // tablica o rozmiarze
                                                // n * m
// powyzsze wyrazenie wymaga obslugi VLA
tab2[1][2] = p2[1][2] = 12;
```

Warto przyjrzeć się umieszczonym wyżej deklaracjom zmiennych wskaźnikowych. Funkcja malloc() zwraca wskaźnik, zatem p2 musi być wskaźnikiem o odpowiednim typie. Deklaracja:

```
int (* p2) [6];       // przed standardem C99
```

stwierdza, że zmienna p2 wskazuje na tablicę sześciu liczb całkowitych (int). To znaczy, że p2[i] można interpretować jako element złożony z sześciu liczb całkowitych oraz że p2[i][j] jest pojedynczą wartością typu int.

Druga deklaracja używa do określenia rozmiarów wskazywanej przez p3 tablicy zmiennej. Oznacza to, że p3 jest traktowane jako wskaźnik do VLA, przez co program nie będzie zgodny ze standardem C90.

## Klasy zmiennych a dynamiczny przydział pamięci

Możesz zastanawiać się nad związkami między klasami zmiennych a dynamicznym przydziałem pamięci. Spójrzmy na pewien idealny model. Można wyobrazić sobie, że program dzieli dostępną mu pamięć na trzy oddzielne przestrzenie: pierwszą — na zmienne statyczne o łączności zewnętrznej, wewnętrznej lub jej braku; drugą — na zmienne automatyczne; oraz trzecią — na dynamicznie przydzielaną pamięć.

Wymagana ilość pamięci dla klas zmiennych o statycznym czasie trwania jest znana już podczas kompilacji, a dane są dostępne do końca działania programu. Zmienne takie powstają przy uruchomieniu programu i trwają do końca jego działania.

Zmienne automatyczne powstają, gdy program wstępuje w blok kodu zawierającego ich definicję i znikają, gdy program go opuszcza. Zatem, kiedy program wywołuje funkcje i funkcje kończą swoje działanie, ilość pamięci wykorzystanej przez zmienne

automatyczne na przemian to rośnie to maleje. Ten obszar pamięci zwykle przechowywany jest w postaci stosu. Oznacza to, że nowe zmienne są dodawane kolejno do pamięci, gdy są tworzone i następnie z niej usuwane w odwrotnym porządku, gdy są zwalniane.

Z dynamicznym przydziałem pamięci mamy do czynienia wówczas, gdy zostanie wywołana funkcja `malloc`() (albo inna funkcja o podobnym charakterze). Pamięć tak przydzielona zwalniana jest przez wywołanie funkcji `free`(). Czas trwania zmiennej kontrolowany jest zatem przez programistę, a nie przez sztywny mechanizm. Dzięki temu blok pamięci może być tworzony przez jedną funkcję, a usuwany przez inną. Z tego powodu, sekcja pamięci dla dynamicznie przydzielonej pamięci może ulec fragmentacji, co oznacza, że nieużywane kawałki mogą być rozsiane pomiędzy aktywnymi blokami pamięci. Trzeba zauważyć, że korzystanie z dynamicznej pamięci jest zwykle wolniejsze niż korzystanie ze stosu.

# Kwalifikatory typu ANSI C

Jak widzieliśmy, zmienne są opisywane poprzez ich typ i klasę pamięci. Standard C90 dodał dwie właściwości: *stałość* (*ang. constancy*) i *ulotność* (*ang. volatility*), deklarowane za pomocą słów `const` i `volatile`, które tworzą tak zwane *typy kwalifikowane* (*ang. qualified types*). Komitet C99 uzupełnił je jeszcze trzecim kwalifikatorem: `restrict`, zaprojektowanym, by ułatwić kompilatorowi proces optymalizacji kodu.

C99 nadał kwalifikatorom jeszcze jedną właściwość — są one niezmienne. Tak naprawdę oznacza to jedynie, że możesz użyć danego kwalifikatora częściej niż raz w jednej deklaracji, gdyż nadmiarowe wystąpienia zostaną zignorowane:

```
const const const int n = 6;  // rownowazne wyrazeniu const int n = 6;
```

Dzięki temu kompilator uznaje poniższe wyrażenia za poprawne:

```
typedef const int zip;
const zip q = 8;
```

## Kwalifikator typu const

Kwalifikator `const` został już przedstawiony w rozdziałach 4. i 10. Słowo kluczowe `const` sprawia, że wartość zmiennej nie może zostać zmodyfikowana za pomocą przypisania, inkrementacji lub dekrementacji.

Jeśli kompilator spełnia wymogi standardu ANSI, to następujący kod:

```
const int brakzmian;   /* sprawia, że brakzmian jest stala */
brakzmian = 12;        /* instrukcja przypisania jest nieprawidlowa */
```

spowoduje wyświetlenie komunikatu o błędzie. Zmienną typu `const` można oczywiście zainicjalizować. Dzięki temu poniższa instrukcja jest całkowicie poprawna:

```
const int brakzmian = 12;     /* ok */
```

Powyższa deklaracja powoduje, że `brakzmian` staje się zmienną tylko do odczytu (*ang. read-only*). Innymi słowy, taka zmienna nie może zostać zmieniona po inicjalizacji.

Słowa kluczowego `const` można użyć na przykład, by stworzyć tablicę danych, których program nie będzie mógł później modyfikować:

```
const int dni_mies[12] ={31, 28, 31, 30, 31, 30, 31, 30, 31, 30, 31};
```

### Stosowanie kwalifikatora const ze wskaźnikami i deklaracje argumentów

Korzystanie ze słowa kluczowego `const` do zadeklarowania zwykłej zmiennej czy tablicy jest bardzo proste. W przypadku wskaźników jest to bardziej skomplikowane, gdyż trzeba rozróżnić sytuacje, gdy `const` odnosi się do samego wskaźnika, od tych kiedy odnosi się do wartości przezeń wskazywanej. Deklaracja:

```
const float * pf;            /* pf wskazuje na stala wartosc float*/
```

sprawia, że zmienna `pf` wskazuje na wartość, która musi pozostać stała. Natomiast wartość samego wskaźnika `pf` może ulec zmianie. Może, na przykład, wskazywać na inną wartość `const`. Z kolei deklaracja:

```
float * const pt;            /* pt jest stalym wskaznikiem*/
```

powoduje, że wskaźnik `pt` nie może ulec zmianie. Musi zawsze wskazywać na to samo miejsce pamięci, choć sama wartość, które się tam znajduje, może się zmienić. Wreszcie deklaracja:

```
const float * const ptr;
```

oznacza, że zarówno wskaźnik `ptr` musi wskazywać zawsze to samo miejsce pamięci, jak i wartość tam przechowywana nie może się zmienić.

Jest jeszcze jedno miejsce, w którym możemy postawić słowo kluczowe `const`:

```
float const * pfc;           // rownowazne wyrazeniu const float * pfc
```

Jak na to wskazuje komentarz, umieszczenie `const` za typem zmiennej i przed znakiem * oznacza, że wskaźnik nie może być użyty do zmiany wartości, na którą wskazuje. Krótko mówiąc, słowo `const`, postawione na lewo od * powoduje, że stałe stają się dane, a. `const` na prawo od * oznacza, że stały ma być wskaźnik.

Powszechnym zastosowaniem słowa kluczowego `const` są deklaracje wskaźników, będących formalnymi argumentami funkcji. Przyjrzyjmy się, dla przykładu, funkcji `wyswietl`(), która wypisuje zawartość tablicy. Aby jej użyć należy przekazać do niej jako argument faktyczny nazwę tablicy, która, jak pamiętamy, jest adresem do pierwszego jej elementu. Przekazanie wskaźnika pozwala zmieniać dane w funkcji wywołującej, czemu można zapobiec stosując następujący prototyp:

```
void wyswietl(const int tab[], int granica);
```

W prototypie i nagłówku funkcji, deklaracja parametru `const int tab[]` jest równoważna deklaracji `const int * tab`. Zatem dane w tablicy nie mogą ulec zmianie.

Biblioteka ANSI C stosuje powyższą praktykę. Jeżeli wskaźnik jest używany wyłącznie po to, by funkcja mogła odczytać dane, to jest deklarowany jako wskaźnik do stałej. Jeżeli wskaźnik służy do zmiany danych w funkcji wywołującej, jest on deklarowany bez kwalifikatora `const`. Na przykład, deklaracja ANSI C dla funkcji `strcat()` wygląda następująco:

```
char * strcat(char *, const char *);
```

Przypomnijmy, że funkcja `strcat()` dodaje kopię drugiego łańcucha na koniec pierwszego. Modyfikuje więc tylko pierwszy z nich, drugi pozostawiając nietknięty. Jak widać, deklaracja podkreśla ten fakt.

## Stosowanie const z danymi globalnymi

Wcześniej wspomnieliśmy, że stosowanie zmiennych globalnych wiąże się z ryzykiem modyfikacji danych przez dowolny fragment programu. Ryzyko jednak znika, gdy dane są stałymi, nie ma więc powodu, by unikać zmiennych globalnych z kwalifikatorem `const`. Można tworzyć zmienne `const`, tablice `const` i struktury `const`. (Struktura jest typem złożonym omawianym w następnym rozdziale).

Kwestią wymagającą uwagi jest współdzielenie stałych przez kilka plików. Do wyboru mamy tu dwie strategie. Pierwsza nakazuje stosować zwykłe reguły dla zmiennych zewnętrznych — definicja w jednym pliku, a deklaracje nawiązujące (używające słowa kluczowe `extern`) w pozostałych:

```
/* plik1.c – definicja kilku zmiennych globalnych */
const double PI = 3.14159;
const char * MIESIACE[12] =
      { "Styczen", "Luty", "Marzec", "Kwiecien", "Maj", "Czerwiec",
      "Lipiec", "Sierpień", "Wrzesien", "Pazdziernik", "Listopad",
      "Grudzień"};
/* plik2.c – definicja stalych globalnych znajduje sie gdzie indziej */
extern const double PI;
extern const char * MIESIACE[];
```

Druga możliwość to umieścić stałe w pliku nagłówkowym. Należy wtedy pamiętać, by dodatkowo użyć statycznej zewnętrznej klasy zmiennych:

```
/* stale.c – definicja kilku stalych globalnych */
static const double PI = 3.14159;
static const char * MIESIACE[12] =
      { "Styczen", "Luty", "Marzec", "Kwiecien", "Maj", "Czerwiec",
      "Lipiec", "Sierpień", "Wrzesien", "Pazdziernik", "Listopad",
"Grudzień"};
/* plik1.c – definicja stalych globalnych znajduje sie gdzie indziej */
#include "stale.h"
/* plik2.c – definicja stalych globalnych znajduje sie gdzie indziej */
#include "stale.h"
```

Gdybyś pominął słowo kluczowe `static`, załączenie pliku `constant.h` w plikach `plik1.c` i `plik2.c` spowodowałoby, że w każdym pliku znalazłaby się definicja tego

samego identyfikatora, na co standard ANSI nie zezwala (choć niektóre kompilatory na to przyzwalają). Nadanie zmiennym klasy zmiennych statycznych zewnętrznych sprawia, że każdy plik dostaje oddzielną kopię danych. Takie podejście nie sprawdziłoby się, gdyby zmienne miały służyć do wymiany informacji — każdy plik widziałby bowiem tylko swoją kopię danych. Ponieważ jednak dane są stałe (dzięki użyciu słowa kluczowego `const`) i identyczne (obydwa pliki korzystają z tego samego pliku nagłówkowego) we wszystkich plikach, nie stanowi to problemu.

Zaletą metody z zastosowaniem pliku nagłówkowego jest to, że nie musisz pamiętać, by użyć definicji tylko w jednym pliku, a deklaracji nawiązujących w pozostałych; wszystkie pliki po prostu korzystają z tego samego pliku nagłówkowego. Wadą jest natomiast powielanie danych. W powyższym przykładzie nie jest to problemem, ale sytuacja byłaby inna, gdybyśmy chcieli użyć stałych tablic o naprawdę dużym rozmiarze.

## Kwalifikator typu volatile

Kwalifikator `volatile` informuje kompilator, że wartość zmiennej może zostać zmodyfikowana przez czynniki inne niż sam program. Zwykle jest on wykorzystywany do adresów sprzętowych i podczas współdzielenia danych z innymi pracującymi równolegle programami. Na przykład pod adresem wskazywanym przez wskaźnik może znajdować się bieżący czas systemowy. Wartość zmienia się z upływem czasu, niezależnie od tego, co robi Twój program. Można użyć także adresu pamięci, by otrzymywać informacje przesyłane, dajmy na to, z innego komputera.

Składnia deklaracji jest taka sama jak w przypadku kwalifikatora `const`:

```
volatile int adr1;    /* adr1 stanowi "ulotna pamiec" */
volatile int * wadr;  /* wadr wskazuje na ulotna pamiec */
```

Instrukcje powyższe deklarują zmienną `adr1` jako zmienną ulotną (`volatile`) oraz zmienną `wadr` jako wskaźnik do wartości ulotnej.

Oczywiście zgodzimy się, że funkcjonalność `volatile` jest bardzo ciekawym pomysłem. Możemy się jednak dziwić, dlaczego komitet ANSI uznał za konieczne, by dodać `volatile` jako słowo kluczowe. Stało się tak, aby umożliwić optymalizację kodu przez kompilator. Spójrzmy na następujący przykład:

```
war1 = x;
/* jakis kod nie uzywajacy x */
war2 = x;
```

Inteligentny (optymalizujący) kompilator może zauważyć, że używamy dwukrotnie zmiennej `x` bez zmiany jej wartości. Tymczasowo umieści więc wartość `x` w rejestrze, a następnie, gdy będzie potrzebna zmiennej `war2`, oszczędzi czas odczytując tę wartość z rejestru zamiast z oryginalnego miejsca w pamięci. Taka procedura określana jest mianem *buforowania* (*ang. caching*). Zwykle buforowane stanowi dobrą optymalizację, ale nie wtedy, gdy zmienna `x` ulega zmianie pomiędzy dwiema instrukcjami

za sprawą czynników zewnętrznych. Gdyby nie wprowadzono słowa kluczowego `volatile`, kompilator nie miałby podstaw, by stwierdzić, że tak się nie stanie i dla bezpieczeństwa musiałby całkowicie zrezygnować z buforowania. Tak było przed wprowadzeniem standardu ANSI. Obecnie jednak, jeśli słowo kluczowe `volatile` nie jest użyte w deklaracji, kompilator zakłada, że wartość między dwiema instrukcjami nie ulega zmianie i optymalizuje kod, stosując buforowanie.

Wartość może mieć równocześnie atrybuty `const`, jak i `volatile`, czyli być jednocześnie stałą i ulotną. Na przykład czas zegara systemowego nie powinien być modyfikowany przez program, co osiągamy wstawiając słowo `const`, a jednocześnie jest zmieniany przez czynniki zewnętrzne, co oznaczamy słowem kluczowym `volatile`. W takim przypadku wystarczy użyć obu kwalifikatorów (ich kolejność nie ma znaczenia) tak jak w poniższej deklaracji:

```
volatile const int adr1;
const volatile int * wadr1;
```

## Kwalifikator typu restrict

Słowo kluczowe `restrict` zwiększa wydajność obliczeniową, pozwalając kompilatorowi zoptymalizować pewne rodzaje kodu. Można je stosować wyłącznie w odniesieniu do wskaźników. Jego użycie oznacza, że wskaźnik tak zdeklarowany ma *wyłączny* dostęp do określonego obszaru pamięci. Aby przekonać się o użyteczności tego kwalifikatora przyjrzyjmy się kilku przykładom:

```
int tab[10];
int * restrict rest_tab = (int *) malloc(10 * sizeof(int));
int * wtab = tab;
```

Wskaźnik `rest_tab` ma tu wyłączny dostęp do przydzielonego przez funkcję `malloc()` obszaru pamięci. Stąd definicja zmiennej `rest_tab` zawiera słowo `restrict`. Z kolei wskaźnik `wtab` nie ma wyłączności na obszar pamięci tablicy `tab`, na którą wskazuje, dlatego nie może być zakwalifikowany jako zmienna `restrict`.

Rozważmy jeszcze następujący, trochę sztuczny przykład, w którym zmienna `n` jest liczbą całkowitą:

```
for( n = 0; n < 10; n++)
{
      wtab[n] += 5;
      rest_tab[n] += 5;
      tab[n] *= 2;
      wtab[n] += 3;
      rest_tab[n] += 3;
}
```

Kompilator, wiedząc że zmienna `rest_tab` ma wyłączny dostęp do wskazywanego obszaru pamięci, może zamienić dwie dotyczące jej instrukcje w jedną, dającą ten sam rezultat:

```
rest_tab[n] += 8;     /* zamiana poprawna */
```

Z kolei dla zmiennej `wtab` możemy otrzymać błędne wyliczenie, jeśli złączymy obie instrukcje w jedną:

```
wtab[n] += 8; /* uwaga, nieprawidlowy wynik */
```

Przyczyną, dla której otrzymujemy niepoprawny rezultat jest taka, że w pętli dane są zmieniane przy pomocy tablicy `tab` pomiędzy dwoma wywołaniami w których poprzez wskaźnik `wtab` odwołujemy się do nich.

Bez słowa kluczowego `restrict` kompilator zmuszony jest przyjąć, że ma miejsce najgorszy możliwy przypadek, a mianowicie, że coś mogło zmienić dane pomiędzy dwoma odwołaniami do wskaźnika. Dzięki słowu `restrict`, kompilator może bezpiecznie optymalizować kod.

Można stosować `restrict` do tych argumentów funkcji, które są wskaźnikami. Kompilator może wówczas przyjąć założenie, że żaden inny wskaźnik nie może zmodyfikować danych wskazywanych przez wskaźniki wewnątrz ciała funkcji i podjąć próbę optymalizacji, jakiej w innym przypadku nie mógłby przeprowadzić. Na przykład biblioteka języka C zawiera dwie funkcje kopiujące bajty z jednej lokalizacji do drugiej. W standardzie C99 funkcje mają następujące prototypy:

```
void * memcpy(void * restrict s1, const void * restrict s2, size_t n);
void * memmove(void * s1, const void * s2, size_t n);
```

Każda kopiuje `n` bajtów z obszaru spod adresu `s2` do obszaru pod adresem `s1`. Funkcja `memcpy`() wymaga, aby obszary te się nie pokrywały, natomiast funkcje `memmove`() nie ma takiego ograniczenia. Deklaracja wskaźników `s1` i `s2` ze słowem `restrict` informuje kompilator, że tylko one mają dostęp do określonego obszaru pamięci, więc wskazywane przezeń bloki pamięci nie mogą się pokrywać. Z kolei funkcja `memmove`(), która pozwala na pokrywanie się kopiowanych obszarów, musi wykazać się większą ostrożnością podczas kopiowania danych, aby nie nadpisać poprzednich wartości.

Słowo `restrict` kierowane jest do dwóch adresatów. Pierwszym jest sam kompilator, który może spokojnie optymalizować kod. Drugim jest użytkownik, który jest informowany, by jako argumenty wstawiać tylko wartości spełniające wymogi słowa kluczowego `restrict`. W ogólności kompilator nie potrafi rozstrzygnąć, czy ograniczenia są rzeczywiście spełnione, więc odpowiedzialność za ich ewentualne naruszenie spoczywa na programiście.

## Stare słowa kluczowe w nowych miejscach

Standard C99 pozwala na umieszczanie kwalifikatorów typu i kwalifikator klasy zmiennej `static` jako argumentów formalnych w prototypach i nagłówkach funkcji. W przypadku kwalifikatorów typu można skorzystać więc z alternatywnej składni, aby uzyskać znane wcześniej efekty. Poniżej mamy deklarację w starym stylu:

```
void slownie(int * const a1, int * restrict a2, int n); // stary sposob
```

Wynika z niej, że zmienna `a1` jest wskaźnikiem `const` do wartości o typie `int`, co jak sobie zapewne przypominasz, oznacza, że wskaźnik jest stały w przeciwieństwie do danych, na które wskazuje. Ponadto zmienna `a2` jest wskaźnikiem o kwalifikatorze `restrict`, opisanym w poprzedniej sekcji. Nowa, równoważna składnia pozwala zapisać to samo w inny sposób:

```
void slownie(int a1[const], int a2[restrict], int n); // nowy sposob podany
                                                      // przez C99
```

W przypadku słowa `static` mamy do czynienia z czymś innym, gdyż wprowadza ono całkiem nową jakość. Przeanalizujmy następujący przykład:

```
double patyk(double tab[static 20]);
```

Takie użycie słowa `static` powoduje, że argument faktyczny w wywołaniu funkcji będzie wskaźnikiem do pierwszego elementu tablicy mającej przynajmniej 20 elementów. Kompilator wykorzystuje tę informację, by zoptymalizować kod tej funkcji.

Podobnie jak w przypadku słowa kluczowego `restrict`, `static` ma dwoje adresatów. Pierwszym jest sam kompilator, który może dzięki temu zoptymalizować kod. Drugim jest użytkownik, który jest informowany, by jako argumenty wstawiać tylko wartości spełniające wymogi słowa kluczowego `static`.

## Kluczowe zagadnienia

Język C udostępnia kilka modeli zarządzania pamięcią. Powinieneś zapoznać się z dostępnymi możliwościami. Musisz także nauczyć się wybierać właściwy dla danego zastosowania typ. Zwykle najlepszym wyborem są zmienne automatyczne. Jeśli się zdecydujesz na inną klasę zmiennych, to powinieneś mieć po temu dobry powód. Również do przekazywania danych między funkcjami lepiej zazwyczaj użyć zmiennych automatycznych, argumentów funkcji i zwracanych wartości, niż zmiennych globalnych. Z drugiej strony, zmienne globalne sprawdzają się, gdy trzeba przechowywać dane stałe.

Powinieneś zrozumieć właściwości pamięci statycznej i automatycznej oraz zagadnienia przydziału pamięci. W szczególności, trzeba pamiętać, że ilość pamięci statycznej jest ustalana już w czasie kompilacji, a dane statyczne są ładowane do pamięci w chwili uruchamiania programu. W przypadku zmiennych automatycznych pamięć jest przydzielana, a następnie zwalniana już podczas pracy programu. Dlatego też ilość pamięci zajmowanej przez zmienne automatyczne może ulegać zmianom. Możesz traktować pamięć automatyczną jako przestrzeń roboczą, którą można wciąż modyfikować. Podobnie pamięć przydzielana (funkcjami z rodziny `malloc`()), może rosnąć i maleć. Jednakże w tym przypadku, proces ten jest kontrolowany wywołaniami odpowiednich funkcji, nie jest więc automatyczny.

# Podsumowanie rozdziału

Pamięć, jakiej używamy w programach, charakteryzowana jest przez czas trwania, zasięg i łączność zmiennych. Jeśli pamięć jest statyczna, to jest przydzielana w momencie uruchamiania programu i istnieje do jego zakończenia. Pamięć automatyczna przydzielana jest zmiennej, gdy program wchodzi do bloku jej deklaracji. Z chwilą wyjścia z bloku pamięć taka jest zwalniana. Pamięć może być przydzielana przez programistę za pomocą funkcji `malloc`() (albo innych podobnych) i zwalniana przez użycie funkcji `free`().

Zasięg decyduje o tym, które części programu będą miały dostęp do zmiennej. Zmienna, zdefiniowana poza ciałem którejkolwiek z funkcji, ma zasięg plikowy i jest dostępna w obrębie każdej funkcji zadeklarowanej po jej definicji. Zmienna, której deklarację umieszczono w bloku albo w argumentach funkcji ma zasięg blokowy i jest dostępna wyłącznie w obrębie tego bloku i w blokach kodu w nim zagnieżdżonych.

Łączność określa stopień w jakim zmienna zdefiniowana w jednej części programu może być powiązana z innymi. Zmienne o zasięgu blokowym, będąc zmiennymi lokalnymi, charakteryzuje brak łączności. Zmienne o zasięgu plikowym mogą mieć łączność zewnętrzną bądź wewnętrzną. Łączność wewnętrzna oznacza, że zmienna może być użyta tylko w obrębie pliku, w którym została zdefiniowana. Łączność zewnętrzna sprawia, że zmienna jest dostępna także w innych plikach.

W języku C zdefiniowano następujące klasy zmiennych:

- automatyczna — zmienna zadeklarowana w bloku (albo jako parametr funkcji) bez operatora klasy zmiennych, albo z operatorem `auto`, należy do klasy zmiennych automatycznych. Ma automatyczny czas trwania, zasięg blokowy i brak łączności. Jeśli nie zostanie zainicjalizowana, jej wartość będzie nieokreślona.
- rejestrowa — zmienna zadeklarowana w bloku (albo jako parametr funkcji) ze specyfikatorem klasy zmiennych `register`, należy do klasy zmiennych rejestrowych. Posiada automatyczny czas trwania, zasięg blokowy, nie ma łączności. Nie można pobrać jej adresu. Zadeklarowanie zmiennej jako rejestrowej informuje kompilator, by użył najszybszej dostępnej pamięci. Jej wartość, jeśli nie jest zainicjalizowana, jest przypadkowa.
- statyczna bez łączności — zmienna zadeklarowana w bloku ze specyfikatorem `static` należy do klasy zmiennych *statyczne bez łączności*. Ma statyczny czas trwania, zasięg blokowy, nie ma łączności. Jest inicjalizowana tylko raz, podczas kompilacji. Jeśli nie zostanie zainicjalizowana, to bajty ją tworzące ustawiane są na 0.
- statyczna z łącznością zewnętrzną — zmienna zadeklarowana poza obrębem którejkolwiek funkcji bez użycia specyfikatora klasy `static` należy do klasy zmiennych statycznych o łączności zewnętrznej. Posiada statyczny czas trwania, zasięg plikowy i łączność zewnętrzną. Jest inicjalizowana tylko raz, podczas kompilacji. Jeśli nie zostanie zainicjalizowana, to bajty ją tworzące ustawiane są na 0.

- statyczna z łącznością wewnętrzną — zmienna zadeklarowana poza obrębem którejkolwiek funkcji z użyciem specyfikatora klasy `static` należy do klasy zmiennych statyczne o łączności wewnętrznej. Posiada statyczny czas trwania, zasięg plikowy i łączność wewnętrzną. Jest inicjalizowana tylko raz podczas kompilacji. Jeśli nie zostanie zainicjalizowana, to bajty ją tworzące ustawiane są na 0.

Przydział pamięci następuje po wywołaniu funkcji `malloc`() (albo innych podobnych), która zwraca wskaźnik do bloku pamięci o żądanej ilości bajtów. Pamięć ta jest dostępna ponownie poprzez jej zwolnienie za pomocą funkcji `free()` z parametrem w postaci wskaźnika do bloku pamięci zwróconego przez funkcję `malloc`().

Język C posiada kwalifikatory typu `const`, `volatile` i `restrict`. Kwalifikator `const` sprawia, że dana jest stała. Zmienna wskaźnikowa ze słowem `const` sprawia, że albo stały jest wskaźnik (czyli może wskazywać tylko jeden jedyny adres) albo stałą jest wartość, na którą on wskazuje. Kwalifikator `volatile` oznacza, że zmienna może być modyfikowana przez czynniki zewnętrzne inne niż sam program. Jest ono używane, aby ustrzec kompilator przed dokonaniem optymalizacji, które w tym przypadku są niewskazane. Kwalifikator `restrict` został wprowadzony również z uwagi na optymalizację. Wskaźnik oznaczony tym słowem ma wyłączny dostęp do określonego bloku pamięci.

# Pytania sprawdzające

1. Które klasy tworzą zmienne lokalne dla zawierającej je funkcji?
2. Które klasy tworzą zmienne, które istnieją przez cały czas działania programu?
3. Która klasa tworzy zmienne, które mogą być wykorzystywane przez kilka plików?
4. Jaką typ łączności mają zmienne o zasięgu blokowym?
5. Jak działa słowo kluczowe `extern`?
6. Rozważmy następujący fragment programu

```
int * p1 = (int *) malloc (100 * sizeof(int));
```

   Czym różnią się podane instrukcje, jeśli chodzi o rezultaty?

```
int * p1 = (int *) calloc (100, sizeof(int));
```

7. Które zmienne są znane którym funkcjom w poniższym przykładzie? Czy kod zawiera jakieś błędy?

```
/* plik 1 */
int stokrotka;
int main(void)
{
   int lilia;
   …;
}
int platek()
{
```

```
   extern int stokrotka, lilia
   …;
}
/* plik 2 */
extern int stokrotka;
static int lilia;
int roza;
int lodyga()
{
   int roza;
   …;
}
void korzen()
{
   …;
}
```

**8.** Co wyświetli poniższy program?

```
#include <stdio.h>
char kolor = 'B';
void pierwsza(void);
void druga(void);
int main(void)
{
   extern char kolor;

   printf("kolor w main() wynosi %c\n", kolor);
   pierwsza();
   printf("kolor w main() wynosi %c\n", kolor);
   druga();
   printf("kolor w main() wynosi %c\n", kolor);
   return 0;
}
void pierwsza()
{
char kolor;
kolor = "R";
printf("kolor w pierwsza() wynosi %c\n", kolor);
}
void druga()
{
kolor = 'G';
printf("kolor w pierwsza() wynosi %c\n", kolor);
}
```

**9.** Plik rozpoczyna się następującymi deklaracjami:

```
static int plink;
int wart_licz(const int tabl[], int wartosc, int n);
```

**a)** Co można powiedzieć o zamiarach programisty spoglądając na te deklaracje?

**b)** Czy zamiana `int wartosc` i `int n` na odpowiednio `const int wartosc` i `const int n` wzmocni ochronę wartości w tym programie?

# Ćwiczenia

1. Przepisz program z listingu 12.4 tak, aby nie korzystał ze zmiennych globalnych.
2. Zużycie paliwa jest zwykle podawane w USA w milach na galon a w Europie w litrach na 100 km. Oto część programu, który pyta użytkownika o wybór trybu (metryczny bądź US), a następnie gromadzi dane i wylicza zużycie paliwa:

```
// pe12-2b.c
#include <stdio.h>
#include "pe12-2a.h"
int main(void)
{
   int tryb;

   printf("Wybierz: 0 – system metryczny, 1 – system US: ");
   scanf("%d", &tryb);
   while( tryb >= 0 )
   {
           wybierz_tryb(tryb);
           pobierz_dane();
           wyswietl_dane();
           printf("Wybierz: 0 – system metryczny, 1 – system US");
           printf(" (-1 aby zakonczyc):" );
           scanf("%d", &tryb);
   }
   printf(Koniec.\n");
   return 0;
}
```

Oto przykładowe dane wyjściowe programu:

```
Wybierz: 0 – system metryczny, 1 – system US: 0
Wprowadz przebyta odleglosc w kilometrach: 600
Wprowadz zuzyte paliwo w litrach: 78.8
Zuzycie paliwa wynioslo 13.13 litrow na 100 km.
Wybierz: 0 – system metryczny, 1 – system US (-1 aby zakonczyc): 1
Wprowadz przebyta odleglosc w milach: 434
Wprowadz zuzyte paliwo w galonach: 12.7
Zuzycie paliwa wynioslo 34.2 mil na galon.
Wybierz: 0 – system metryczny, 1 – system US (-1 aby zakonczyc): 3
Podano nieprawidlowa wartosc. Wybrano system 1(US).
Wprowadz przebyta odleglosc w milach: 388
Wprowadz zuzyte paliwo w galonach: 15.3
Zuzycie paliwa wynioslo 25.4 mil na galon.
Wybierz: 0 – system metryczny, 1 – system US (-1 aby zakonczyc): -1
Koniec.
```

Jeśli użytkownik wybierze niewłaściwy tryb, program powinien to skomentować i użyć ostatnio wybranego trybu. Program powinien działać po dołączeniu do niego pliku nagłówkowego `pe12-2a.h` i pliku `pe12-2a.c`. Plik z kodem źródłowym powinien definiować trzy zmienne o zasięgu plikowym i łączności wewnętrznej. Niech jedna reprezentuje tryb, druga dystans, a trzecia zużyte paliwo. Funkcja `pobierz_dane()` ma prosić o dane zgodnie z wybranym trybem i przechowywać

wprowadzone przez użytkownika dane w zmiennych o zasięgu plikowym. Funkcja `wyswietl_dane()` niech oblicza i wyświetla zużycie paliwa w oparciu o wybrany tryb.

3. Zmodyfikuj ponownie program opisany w ćwiczeniu 2 tak, by używał wyłącznie zmiennych automatycznych. Niech program posiada taki sam interfejs użytkownika, to znaczy powinien prosić użytkownika, by wpisał tryb itd. Będziesz musiał skorzystać jednak z innych wywołań funkcji.
4. Napisz i sprawdź w pętli funkcję, która zwraca ile razy została wywołana.
5. Napisz program, który generuje listę 100 losowych liczb z przedziału od 1 do 10 w porządku malejącym. (Możesz w tym celu wykorzystać algorytm przedstawiony w rozdziale 11 dla typu `int`. W tym przypadku po prostu posortuj same liczby).
6. Napisz program, który generuje 1000 losowych liczb z przedziału od 1 do 10. Nie zachowuj ani nie pokazuj liczb, tylko wyświetl liczbę razy dana liczba została wybrana. Niech program korzysta z 10 różnych ziaren. Czy liczby te pojawiają się w równej ilości? Możesz użyć funkcji z tego rozdziału albo standardowych funkcji ANSI C: `rand()` i `srand()`, które działają tak jak nasze funkcje. Jest to jeden ze sposobów, by sprawdzić losowość określonego generatora liczb losowych.
7. Napisz program, który zachowuje się jak modyfikacja listingu 12.13, którą omówiliśmy po pokazaniu jego wyników. Niech program zwraca następujące wartości:

```
Wprowadz liczbe kolejek; wybierz q aby zakonczyc.
18
Ile scian i ile kosci?
6 3
Oto 18 kolejek rzutow 3 6-sciennymi kostkami:
   12 10 6 9 8 14 8 15 9 14 12 17 11 7 10
   13 8 14
Wprowadz liczbe kolejek; wybierz q aby zakonczyc.
q
```

8. Oto fragment programu:

```
// pe12-8.c
#include <stdio.h>
int * stworz_tablice (int elem, int wart);
void pokaz_tablice (const int tab[], int n);
int main (void)
{
int * wt;
int rozmiar;
int wartosc;
printf("Podaj liczbe elementow: ");
scanf(“%d", &rozmiar);
while (rozmiar > 0)
{
   printf("Podaj wartosc poczatkowa: ");
   scanf("%d", &wartosc);
   wt = stworz_tablice(rozmiar, wartosc);
```

```
    if (wt)
    {
            pokaz_tablice(wt, rozmiar);
            free(wt);
    }
    printf("Podaj liczbe elementow (<1 – koniec): ");
    scanf("%d", &rozmiar);
}
printf("Koniec.\n");
return 0;
}
```

Uzupełnij program przez udostępnienie definicji funkcji `stworz_tablice()` i `pokaz_tablice()`. Funkcja `stworz_tablice()` wymaga dwóch argumentów. Pierwszy z nich jest liczbą elementów tablicy liczb całkowitych (`int`), a drugi oznacza wartość, która jest przypisywana każdemu z elementów. Funkcja używa `malloc()`, by stworzyć tabelę o odpowiednim rozmiarze, przypisać każdemu elementowi wskazaną wartość i zwrócić wskaźnik do tablicy. Funkcja `pokaz_tablice()` wyświetla zawartość — po osiem liczb w wierszu.