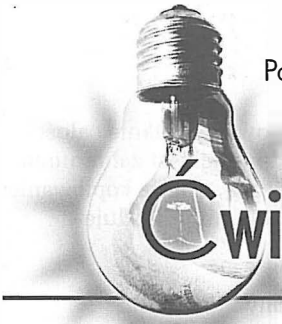


Poznaj w praktyce podstawowe narzędzie pracy profesjonalnych programistów!

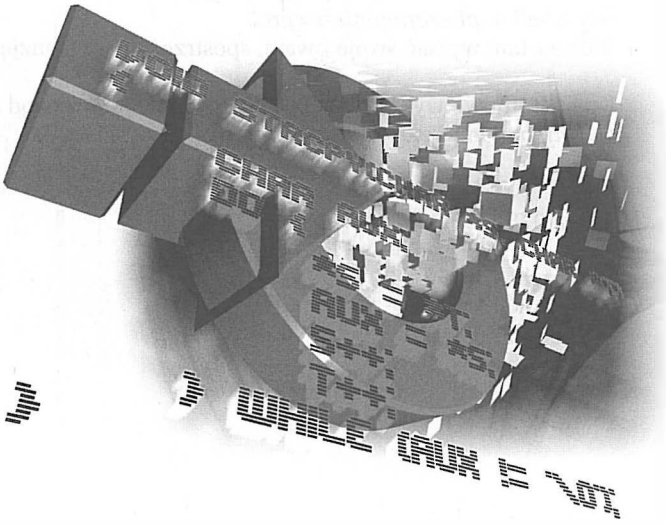
Ćwiczenia praktyczne

Programowanie w języku C

Marek Tłuczek

Wydanie II

- **Poznaj** podstawy języka C
- **Naucz się** programowania strukturalnego
- **Przećwicz** swoje umiejętności



Helion

Spis treści

Wstęp

Dlaczego C? Takie pytanie zadałem we wstępie do pierwszego wydania tej książki. Łatwo było na nie odpowiedzieć dziesięć lat temu i przekonać nawet moją babcię do odstawienia drutów i zajęcia się programowaniem. Dzisiaj już nie jest tak prosto uzasadnić potrzebę nauki tego języka. Babcia zaczęła programować w Javie aplikacje na swój telefon z systemem Android. Młodzi karierowicze oprogramowują swoje strony w językach skryptowych typu PHP, JavaScript czy nawet Flex (albo kopiują i przerabiają gotowe skrypty, które bez problemów można znaleźć w Internecie). Tym, którzy marzą o tworzeniu gier komputerowych, często wystarcza tylko Flash. Kiedyś łatwo było napisać, że C jest szybki i programy nie wymagają dużej pamięci. Tyle że wtedy mój komputer wyposażony był w procesor 350 MHz i 256 MB pamięci. Teraz większość użytkowników ma pewnie minimum 4 rdzenie po 2,7 GHz i 4 GB pamięci, więc nikt aż tak bardzo nie przejmuje się np. rozmiarem programu czy czasem jego działania.

Do kogo więc adresowana jest ta książka? C to język, w którym napisano jądra (i nie tylko) wszystkich najpopularniejszych systemów operacyjnych (np. UNIX, Linux, Windows, MacOS). Jest zatem przydatny zarówno dla programistów systemów operacyjnych, jak i dla wszystkich, którzy chcą te systemy poznać (np. dla hakerów). W C napisano również inne języki programowania (np. PHP, Ruby, Perl czy Python). C jest świetny dla programistów tzw. systemów wbudowanych (tzn. oprogramowania niskopoziomowego dla różnych urządzeń, sterowników itp.). Umożliwia bezpośredni dostęp do pamięci i jednocześnie zakłada, że programista wie, co robi, nie chroniąc go na siłę

od wszelkich możliwych błędów. Na koniec można też dodać oczywiste stwierdzenie, że programy w C działają szybciej i zajmują mniej pamięci.

A co, jeśli nie jesteś programistą systemów wbudowanych, masz gdzieś możliwość bezpośredniego dostępu do pamięci, a programowanie jąder uznajesz za wstydliwe zajęcie, o którym nie należy opowiadać w gronie kolegów? Na pewno dałeś zarobić autorowi na piwo. Ale nie tylko! Możesz pochwalić się kumplom, jakim jesteś mistrzem, ponieważ programujesz w języku, w którym napisane są programy stworzone w językach przez nich używanych. Znajomość budowy języka C to elementarny warunek zrozumienia innych języków programowania (np. Javy i C++). Podstawa składni języka jest prawie taka sama, ale wiedza, którą posiądziesz, pisząc programy w języku niskopoziomowym, okaże się niezbędna do lepszego zrozumienia działania aplikacji, które napiszesz w innych językach.

Podstawy języka C

C jest językiem programowania o potężnych możliwościach. Przy użyciu tylko kilku jego funkcji i własnej wyobraźni można stworzyć nawet najbardziej skomplikowane programy. C jest używany przy tworzeniu takich projektów jak systemy operacyjne, edytory tekstu czy nawet kompilatory innych języków programowania. Świadczy to o tym, że C potęgą jest i basta. Zaczynajmy więc — na pewno nie możesz się doczekać napisania swojego pierwszego programu.

Tworzenie programu w C

Tworzenie programu w C można podzielić na cztery etapy:

Napisanie kodu źródłowego w dowolnym edytorze tekstu.

Kod źródłowy to tylko ciąg instrukcji — tekst. W przeciwieństwie do człowieka komputer go nie zrozumie. Procesor rozumie tylko instrukcje binarne, których zbiór nazywamy językiem maszynowym. Do przetłumaczenia instrukcji w trybie tekstowym na odpowiedniki binarne potrzebny jest kompilator.

Kompilacja kodu źródłowego.

Niestety nie istnieje jeden uniwersalny kompilator języka C. Każdy system operacyjny posiada inny system plików, więc jest to praktycznie niemożliwe. W tej książce prezentuję przykłady tworzenia programów przy użyciu kompilatora systemu UNIX uruchamianego poleceniem

gcc. Rezultatem procesu kompilacji jest utworzenie dla każdego pliku programu z kodem źródłowym odpowiadającego mu pliku z instrukcjami maszynowymi. Dla każdego pliku z rozszerzeniem *.c* tworzony jest więc tzw. plik obiektu z rozszerzeniem *.o*.

Łączenie za pomocą programu „linker".

Kompilator jedynie przekształca kod źródłowy w pliki obiektów. Nie można ich na razie uruchomić. Muszą one jeszcze zostać „połączone" za pomocą programu `linker`. Proces łączenia, jak sama nazwa wskazuje, łączy program złożony z wielu plików w jedną całość. Często różne funkcje zdefiniowane są w różnych plikach. Czasami też programista zapomina o zdefiniowaniu danej funkcji lub o dołączeniu odpowiedniego pliku. Zadaniem linkera jest więc również sprawdzenie, czy wszystkie funkcje, do których należy odwołać się w kodzie, zostały zdefiniowane, i poinformowanie o wystąpieniu takiego błędu.

Uruchomienie programu.

Programy w książce zostały przetestowane zarówno kompilatorem gcc w systemie Linux, jak i kompilatorem gcc z pakietu MinGW w systemie Windows. Użytkownikom wszelkiej maści systemów Windows szczególnie polecam ten pakiet. Instrukcje instalacji znajdują się na stronie internetowej:

http://www.mingw.org/wiki/Getting_Started

ĆWICZENIE

1.1 Napisz program, który wyświetli na ekranie dowolny tekst, następnie skompiluj go i uruchom.

```
1: /* Przykład 1.1 — pierwszy program */
2: #include <stdio.h>
3: int main()
4: {
5:       printf("Czyz nie jestem wspanialy? Napisalem swoj pierwszy
6:             program!!! \n");
7:       return 0;
8: }
```

Proces kompilacji za pomocą gcc:

```
gcc -o cw1.1 cw1.1.c
```

Uruchomienie pliku:

```
./cw1.1
```

Teraz kilka słów wyjaśnienia, jak zbudowany jest program. **Wiersz 1** zawiera komentarz, który rozpoczyna się znakiem /*, a kończy za pomocą */. Kompilator ignoruje tekst znajdujący się pomiędzy tymi specjalnymi znakami.

Dyrektywa `#include` w **wierszu 2** powoduje włączenie do programu informacji zawartych w pliku nagłówkowym `stdio.h`. `main()` to główna funkcja tego kodu, instrukcje w programie są wywoływane, począwszy od pierwszej linijki, a skończywszy na ostatniej.

Funkcja `main()` jest niezbędna w każdym programie pisanym w języku C. Ciąg instrukcji musi być zawarty pomiędzy nawiasami { i }.

Funkcję `printf()`, zastosowaną w **wierszu 5**, opiszę w kolejnym rozdziale. Pozostała jeszcze instrukcja `return 0`. Zwraca ona wartość 0 do systemu operacyjnego, informując, że program zakończył się pomyślnie. Wpisując wartość mniejszą od 0 (np. `return -1`), informujemy, że wystąpił błąd.

printf() — funkcja wyjścia

Funkcja `printf()` służy do wyświetlania tekstu na monitorze — urządzeniu wyjścia. Dlatego można stwierdzić, że `printf()` jest funkcją wyjścia. To podstawowa funkcja biblioteczna języka C, naukę programowania rozpoczniemy od zapoznania się z jej działaniem.

ĆWICZENIE

1.2 Napisz program, który wyświetli na ekranie Twoje dane osobowe. Każda informacja musi być podana w oddzielnej linijce.

```
1: /* Przyklad 1.2 */
2: /* Wypisywanie danych osobowych na ekranie */
3: #include <stdio.h>
4: int main()
5: {
6:       printf("Jozek \nMarchewa \nWilcze Dolki 21");
7:       printf("\n45-680 \nKurzy Zdroj \n");
8:       return 0;
9: }
```

Jak widać, tekst wyświetlany na ekranie musi być zawarty w cudzysłowie. Dodatkowo znak \n wywołuje przeskok do następnej linijki. Należy on do grupy tzw. sekwencji wyjściowych. Niektóre z nich zademonstruję w kolejnych przykładach.

ĆWICZENIE

1.3 Napisz program, który wyświetli na ekranie tekst „Kocham programowanie". Wyrazy te powinny być oddzielone od siebie dwoma tabulatorami.

```
1: /* Przyklad 1.3 */
2: /* Wypisuje na ekranie dwa wyrazy oddzielone */
3: /* od siebie dwoma znakami tabulatora */
4: #include <stdio.h>
5: int main()
6: {
7:       printf("Kocham \t\tprogramowanie\n");
8:       return 0;
9: }
```

Wykorzystana tu została kolejna sekwencja wyjściowa: \t. Wstawia ona odstęp wielkości jednego tabulatora.

ĆWICZENIE

1.4 Napisz program, który wyświetli na ekranie cytat „Litwo, Ojczyzno moja".

```
1: /* Przyklad 1.4 */
2: /* Wypisuje na ekranie cytat */
3: #include <stdio.h>
4: int main()
5: {
6:       printf("\"Litwo, Ojczyzno moja\"\n");
7:       return 0;
8: }
```

Sekwencja wyjściowa \" wyświetla na ekranie znak cudzysłowu.

Pozostałe znaki tego typu to:

\a — wywołuje dźwięk;

\b — powoduje wymazanie pojedynczego znaku (backspace);

\\ — wstawia znak \;

\? — wstawia znak zapytania;

\' — wstawia znak ';

\" — wstawia znak cudzysłowu.

ĆWICZENIE

1.5 Napisz program, który wyświetli na ekranie tekst „Ale wnerwiający dźwięk!!!" oraz wywoła dźwięk 4 razy.

```
1: /* Przyklad 1.5 */
2: /* Wypisuje na ekranie tekst oraz wywoluje dzwiek */
3: #include <stdio.h>
4: int main()
5: {
6: printf( "Ale wnerwiajacy dzwiek!!! \a\a\a\a\n");
7: return 0;
8: }
```

Zmienne w języku C

Zmienna jest to pewne miejsce w pamięci komputera, któremu można przypisywać różne wartości. Wyobraź sobie magazyn pełny małych pudełek — to one będą tworzyły pamięć komputera. Każde z nich to pewne miejsce w tej pamięci. Chcesz zadeklarować zmienną, więc wybierasz jedno pudełko i naklejasz na nim nalepkę (z nazwą zmiennej) oraz wrzucasz do niego jakąś rzecz, którą możesz później swobodnie wymienić na inną (nie zmieniając nalepki).

A zatem pudełko to zmienna, która posiada swój adres w pamięci oraz nazwę (umieszczoną na nalepce). Można przypisywać jej różne wartości (wkładać różne rzeczy do pudełka) bez zmiany nazwy i adresu (nalepka pozostaje ta sama).

Przed użyciem zmiennej w programie należy ją oczywiście zadeklarować. Trzeba więc podać jej typ (rodzaj wartości, jakie mogą być do niej przypisywane).

Deklaracja nie przypisuje wartości zmiennej, ale przydziela jej adres w pamięci (rezerwuje dla niej miejsce). W tabeli 1.1, poniżej, zostały zebrane podstawowe typy zmiennych.

ĆWICZENIE

1.6 Zadeklaruj pięć zmiennych i dobierz do nich odpowiednie typy.

Tabela 1.1.

Integer (`int`)	Liczba stałoprzecinkowa z zakresu od –2 147 483 648 do 2 147483647
Char (`char`)	Pojedynczy znak ASCII oraz liczby od –128 do 127
Short integer (`short`)	Liczba stałoprzecinkowa z zakresu od –32 768 do 32 767
Long integer (`long`)	Liczba stałoprzecinkowa z zakresu od –2 147 483 648 do 2 147 483 647
Unsigned integer (`unsigned int`)	Liczba stałoprzecinkowa z zakresu od 0 do 4 294 967 295
Unsigned short integer (`unsigned short`)	Liczba stałoprzecinkowa z zakresu od 0 do 65 535
Unsigned long integer (`unsigned long`)	Liczba stałoprzecinkowa z zakresu od 0 do 4 294 967 295
Single-precision floating point (`float`)	Liczba zmiennoprzecinkowa z zakresu od 1.2E–38 do 3.4E38 (pamiętane 7 cyfr)
Double-precision floating point (`double`)	Liczba zmiennoprzecinkowa z zakresu od 2.2E–308 do 1.8E308 (pamiętane 16 cyfr)

Powinny one zawierać np.:

a) procent miesięcznego zarobku, który wydajesz na kobiety;

b) Twój wiek;

c) odległość od Twojego domu do szkoły w centymetrach (powinieneś znać);

d) temperaturę powietrza (wartość stałoprzecinkowa);

e) cenę 1 kilograma nawozu w najbliższej wsi (dokładną).

```
double procent;
unsigned int MojWiek;
unsigned long odleglosc;
int temperatura;
float cena_nawozu;
```

W przykładzie a) należy użyć typu `double`, ponieważ wartość będzie tak minimalna, że trzeba podać ją z dokładnością przynajmniej 8 miejsc po przecinku (np. 0,00000001%). Zmiennej `MojWiek` przypisana zosta-

nie wartość typu `unsigned int` — wiek jest zawsze dodatni. W przykładzie c) odległość w centymetrach od domu do szkoły będzie dosyć duża (załóżmy, że wynosi ona 1 km; 1 km = 1000 m = 100 000 cm; wartość wykracza poza granicę typu `int`), dlatego zalecam użycie typu `unsigned long` (dodatkowo wiadomo, że odległość jest zawsze dodatnia). Temperatura może przyjmować wartości zarówno dodatnie, jak i ujemne, więc w tym przypadku odpowiedni będzie typ `int`. Natomiast cena nawozu musi być podana z dokładnością do 2 miejsc po przecinku (dobry rolnik zawsze prowadzi dokładne rozliczenia, i tak np. 12,35 zł to aktualna cena w Pysznicy, wiosce koło Stalowej Woli).

Zastanawiasz się pewnie, jakie nazwy można nadawać zmiennym. Otóż istnieją pewne reguły:

1. Nazwy mogą zawierać cyfry, litery oraz znak podkreślenia _.
2. Słowa kluczowe języka C nie mogą być używane jako nazwy zmiennych.
3. Pierwszym znakiem nazwy musi być litera.

Kiedy pisze się programy w C, ważny jest również styl (możliwe, że inni chcieliby zrozumieć, na czym polega działanie Twojego programu; przypomnij sobie choćby swojego nauczyciela matematyki i jego irytację przy sprawdzaniu klasówek — skutki tego nie były chyba zbyt przyjemne).

W powyższym przykładzie zastosowałem dwa popularne rodzaje nadawania nazw zmiennym — oddzielenie dwóch słów znakiem podkreślenia (`cena_ nawozu`) oraz tzw. *camel notation* (`MojWiek`). Wybór metody należy do Ciebie!

Dobra... koniec zanudzania! Piszemy program...

ĆWICZENIE

1.7 Napisz program, który jeszcze raz deklaruje zmienne z poprzedniego przykładu oraz przypisuje im odpowiednie wartości. Aby było ciekawiej, spraw, aby zostały wyświetlone na ekranie.

```
1: /* Przyklad 1.7 */
2: /* Przypisuje zmiennym wartosci oraz je wyswietla */
3: #include <stdio.h>
4: double procent;
5: unsigned int MojWiek;
6: unsigned long odleglosc;
```

```
7: int temperatura;
8: float cena_nawozu;
9: int main()
10: {
11:     procent = 0.000001;
12:     MojWiek = 20;
13:     odleglosc = 100000;
14:     temperatura = -5;
15:     cena_nawozu = 12.35;
16:     printf("Nazywam sie Zdzichu, mam %u".MojWiek);
17:     printf(" lat, chodze do szkoly oddalonej ");
18:     printf("od mojego domu o %lu cm", odleglosc);
19:     printf("\nDziewczyny z mojej szkoly maja ");
20:     printf("temperature ciala rzedu %d stopni Celsjusza, ",
21:           temperatura);
22:     printf("dlatego tez wydaje na nie zaledwie ");
23:     printf("%f procent moich dochodow ", procent);
24:     printf("z pracy dodatkowej, jaka jest handel ");
25:     printf("hurtowa nawozem w cenie %f zl za kg\n",
26:           cena_nawozu);
27:     return 0;
28: }
```

Należy Ci się krótkie wyjaśnienie. Zmienne można deklarować na początku programu (przed funkcją `main()`), a także na początku każdej funkcji. Wartości, jak widać powyżej, są przypisywane w bardzo prosty sposób. Służy do tego znak równości =. Pamiętaj jednak — **znak równości w języku C służy tylko do przypisywania wartości** (np. zmienna `MojWiek` nie jest równa 20, ma jedynie przypisaną taką wartość).

Zastanawiasz się, co znaczą te dziwne znaczki: `%u`, `%lu` itp.?

Są to tzw. specyfikatory konwersji (z ang. *conversion specifiers*). Nakazują funkcji `printf()` wyświetlać na ekranie wartości zmiennych określonego typu. Oto znaki odpowiadające typom poszczególnych zmiennych:

`char`	`%c`
`int, short`	`%d`
`long`	`%ld`
`float, double`	`%f`
`unsigned int, unsigned short`	`%u`
`unsigned long`	`%lu`

ĆWICZENIE

1.8 Zadeklaruj dwie zmienne typu `int` oraz `float`, przypisując im jakieś wartości. Następnie spraw, aby zostały one wyświetlone na ekranie.

```
1: /* Przyklad 1.8 */
2: /* Przypisuje zmiennym wartosci oraz je wyswietla */
3: #include <stdio.h>
4: int waga = 100;
5: float promien = 10.3;
6: main()
7: {
8:        printf("Waze %d kg i wszystkie kobiety", waga);
9:        printf(" w promieniu %f metrow nie mogą",
10:              promien);
11:       printf(" oderwac ode mnie wzroku\n");
12:       return 0;
13: }
```

Jak widać, można przypisywać zmiennym wartości podczas ich deklaracji.

Stałe w C

Podobnie jak zmienna, stała to pewne miejsce w pamięci komputera, któremu można przypisywać różne wartości, niestety nie można ich zmieniać po jednorazowym przypisaniu.

Przypomnij sobie przykład z pudełkiem (z poprzedniego rozdziału), w którym umieściłeś pewną rzecz i na które nakleiłeś nalepkę z nazwą zmiennej. W przypadku stałych po włożeniu owej rzeczy do pudełka zaklejasz je taśmą, której nie wolno Ci już zerwać (nie można zmienić wartości stałej po jednorazowym przypisaniu wartości).

Rozróżnia się dwa typy stałych — stałą literalną (*literal constant*) oraz stałą symboliczną (*symbolic constant*). Stała literalna to po prostu wartość wpisana w kod programu.

Przykład:

```
godzina = 60; /* 60 to stała literalna oznaczająca liczbę minut
↪w godzinie */
```

Stała symboliczna, jak sama nazwa wskazuje, to z reguły wartość reprezentowana w programie przez pewien symbol. Stałą tego typu deklarujemy na początku kodu w następujący sposób:

```
#define NaszaLiczba 13
```

`NaszaLiczba` to symbol stałej, a `13` — przypisana jej wartość.

Innym sposobem deklaracji stałej jest użycie słowa kluczowego `const`, np.:

```
const int NaszaLiczba 13
```

Wygląda to jak deklaracja zmiennej, jednak słowo `const` „zakleja pudełko" (powoduje, że wartości zmiennej `NaszaLiczba` nie będzie można już zmodyfikować). Należy również pamiętać, że stała deklarowana w taki sposób może być umieszczana w różnych miejscach programu — wpływa to w znaczący sposób na jego poprawne działanie.

ĆWICZENIE

1.9 Napisz program, który oblicza pole kuli.

```
1: /* Przyklad 1.9 */
2: /* Oblicza pole kuli */
3: #include <stdio.h>
4: #define PI 3.14
5: float PoleKuli;
6: const int R = 5;
7: int main()
8: {
9:      PoleKuli = 4*PI*R*R;
10:     printf("Pole Kuli wynosi %f\n", PoleKuli);
11:     return 0;
12: }
```

Najpierw należy zadeklarować stałą `PI` w **wierszu 4** programu za pomocą `#define`, a następnie — kolejno — zmienną `PoleKuli` oraz stałą `R` (promień kuli, za pomocą słowa kluczowego `const`). Teraz w bardzo prosty sposób można obliczyć pole kuli w **wierszu 9** i od razu przypisać otrzymaną wartość zmiennej `PoleKuli`. W wierszu 10 pozostaje zastosować funkcję `printf()`, aby wyświetlić wynik na ekranie.

Ależ to proste!

Proponuję napisanie podobnych programów, które obliczałyby np. pole walca, stożka czy też obwód koła. Nawet jeśli Ci się nie chce, spróbuj, na pewno pomoże Ci to w zrozumieniu trudniejszych zagadnień.

ĆWICZENIE

1.10 Napisz program, który przelicza liczbę sekund w 24 godzinach oraz wyświetla wynik na ekranie.

```
1: /* Przyklad 1.10 */
2: /* Oblicza ilosc sekund w 24 godzinach */
3: #include <stdio.h>
4: #define ile_min_w_godz 60
5: int ile_sek_w_dobie;
6: const int ile_sek_w_min = 60;
7: int main()
8: {
9:       ile_sek_w_dobie=24*ile_min_w_godz*ile_sek_w_min;
10:      printf("Ilosc sekund w 24 godzinach wynosi %d",
11:             ile_sek_w_dobie);
12:      return 0;
13: }
```

Jak widać, kolejny raz użyłem dwóch typów deklaracji stałych (**wiersz 4** oraz **6**). Oczywiście, w tym ćwiczeniu nie ma znaczenia, która z tych deklaracji zostanie użyta. Można wykorzystać dwa razy `#define` czy też `const`, nie wpłynie to istotnie na działanie programu.

scanf() — funkcja wejścia

A teraz coś zupełnie z innej beczki — `scanf()` — kolejna funkcja biblioteki `stdio.h`. W przeciwieństwie do `printf()` jest ona funkcją wejścia. Co to znaczy? Nie wyświetla niczego na ekranie monitora, lecz **pobiera** informacje z klawiatury, a następnie przydziela je odpowiednim zmiennym.

Wartości tym razem nie są przypisywane przez twórcę kodu, ale podawane w trakcie działania programu. Program udostępnia „pudełko" (zmienną) i prosi o włożenie do niego pewnych rzeczy (zależnie od typu zadeklarowanej zmiennej).

Przykład:

```
int x;
scanf ("%d", &x);
```

Jak widać, należy podać typ zmiennej (wcześniej zadeklarowanej) w cudzysłowie oraz jej nazwę poprzedzoną znakiem &. Dzięki temu program w trakcie działania zapyta Cię o wartość, którą następnie przypisze odpowiedniej zmiennej — w tym przypadku zmiennej x.

ĆWICZENIE

1.11 Napisz program, który zapyta Cię o wiek, a następnie wyświetli podaną przez Ciebie wartość na ekranie.

```
1: /* Przyklad 1.11 */
2: /* Pyta o wiek oraz wyswietla podana wartosc*/
3: #include <stdio.h>
4: int Wiek;
5: int main()
6: {
7:       scanf("%d", &Wiek);
8:       printf("Masz %d lat", Wiek);
9:       return 0;
10: }
```

Banalne, prawda? Deklaruje się zmienną, przydzielając jej miejsce w pamięci (**wiersz 4**). Potem zostaje użyta funkcja `scanf()` (**wiersz 7**), która pobiera wartość z klawiatury i wpisuje ją w podane miejsce, a następnie — funkcja `printf()` (**wiersz 8**), aby wyświetlić daną wartość na ekranie. Zapytasz, dlaczego przed nazwą zmiennej znajduje się ten śmieszny znaczek &. Ma to związek z adresowaniem pamięci. Na razie w to nie wnikaj — nie jest Ci to teraz potrzebne: o `scanf()` i innych funkcjach wejścia dowiesz się więcej z następnych rozdziałów. Tymczasem przejdź do następnych zadań. Proponuję Ci to z przyjemnością... bo mogę wymyślać ich więcej. I Ty się cieszysz, prawda? Zapewne już pokochałeś C! Wspaniale! Podziękujesz mi później!

ĆWICZENIE

1.12 Napisz program, który zapyta Cię o wiek, a następnie obliczy, ile będziesz miał lat za 480 miesięcy.

```
1: /* Przyklad 1.12 */
2: /* Program oblicza Twoj wiek po uplywie 480 miesiecy */
3: #include <stdio.h>
4: #define PrzedzialCzasu 480
5: #define ile_mies_w_roku 12
6: int main()
7: {
8:       int Wiek;
```

```
9:      int ObliczonyWiek;
10:     int IleLat;
11:     IleLat = PrzedzialCzasu/ile_mies_w_roku;
12:     scanf("%d", &Wiek);
13:     ObliczonyWiek = Wiek+IleLat;
14:     printf("Za 480 miesiecy, czyli %d lat, ", IleLat);
15:     printf("mial %d lat\n", ObliczonyWiek);
16:     return 0;
17: }
```

Hm... trochę bardziej skomplikowane? Należy Ci się dokładne wyjaśnienie tego, co tu się dzieje. Aby zamienić 480 miesięcy na lata, trzeba zadeklarować kolejną stałą — `ile_mies_w_roku` (**wiersz 5**). Następnie należy wprowadzić potrzebne zmienne — `Wiek` (miejsce na wartość wprowadzoną z klawiatury), `ObliczonyWiek` (w celu przechowania wyniku zadania), `IleLat` (tymczasową zmienną zawierającą obliczoną liczbę lat, która ma odpowiadać 480 miesiącom). Kolejnym krokiem jest obliczenie zmiennej `IleLat`. W tym celu dzieli się stałą `PrzedzialCzasu` przez `ile_mies_w_roku` (przy użyciu operatora / — o tym więcej w następnym rozdziale). Następnie trzeba skorzystać z funkcji `scanf()`, aby uzyskać wartość zmiennej `Wiek`, do której później zostanie dodana zmienna `IleLat` — i tak otrzymujesz końcowy wynik. Ostatecznie otrzymana wartość zostaje wyświetlona na ekranie przy użyciu `printf()`. I wszystko jasne.

Instrukcja warunkowa if

Instrukcja warunkowa umożliwia kontrolę wykonywania poszczególnych instrukcji w ramach programu. Sprawdza ona dany warunek i, w przypadku jego spełnienia, wykonuje blok instrukcji.

Instrukcja warunkowa `if` przyjmuje zazwyczaj następującą postać:

```
if (warunek)
  {
       instrukcja 1;
       instrukcja 2;
       .............
       .............
       instrukcja n;
  }
```

A co dzieje się w przypadku, gdy warunek nie zostanie spełniony? Blok instrukcji nie jest wykonywany, natomiast program powraca do swojego normalnego biegu, czyli przeskakuje do dalszej jego części. Opcjonalnie można również wprowadzić polecenie `else`, które umożliwi wykonanie innego bloku instrukcji (kiedy warunek nie zostanie spełniony). Wówczas instrukcja `if` przyjmie postać:

```
if (warunek)
  {    instrukcja 1;
       instrukcja 2;
       ............;
       ............;
       instrukcja n;
  }
else
  {    instrukcja 1;
       ............;
       instrukcja n;
  }
```

ĆWICZENIE

1.13

Napisz program, który poprosi Cię o podanie dwóch liczb, a następnie porówna je i wyświetli na ekranie tekst informujący o tym, która z nich jest większa.

```
1: /* Przyklad 1.13 */
2: /* Porownuje dwie liczby oraz decyduje, ktora z nich */
3: /* jest wieksza */
4: #include <stdio.h>
5: int x, y;
6: int main()
7: {
8:      printf("Podaj pierwsza liczbe: \n");
9:      scanf("%d", &x);
10:     printf("Podaj druga liczbe: \n");
11:     scanf("%d", &y);
12:     if (x>y)
13:         printf("Liczba %d jest wieksza\n", x);
14:     else
15:     {
16:         printf("Liczba %d jest wieksza lub rowna ", y);
17:         printf("liczbie %d\n", x);
18:     }
19:     return 0;
20: }
```

W drugim wierszu zostają zadeklarowane dwie zmienne: x oraz y. Następnie wykonywana jest funkcja printf() oraz scanf(), która czeka na podanie pierwszej liczby. W **wierszach 10 i 11** ponownie wykonywane są te funkcje (w tym przypadku scanf() czeka na wprowadzenie drugiej liczby). Instrukcja warunkowa sprawdza, czy warunek x>y jest spełniony, a następnie wykonuje jedną z funkcji printf(). Na pewno zauważyłeś, że ćwiczenie zostało rozwiązane niezgodnie z poleceniem, gdyż wartości zmiennych x i y mogą być sobie równe. Oczywiście to celowe działanie, aby zmusić Cię do myślenia. Twoją misją bojową będzie ulepszenie tego programu.

Podpowiedź: do rozwiązania tego problemu będzie potrzebny materiał zawarty w dalszej części podrozdziału.

ĆWICZENIE

1.14 Napisz program, który dokonuje mnożenia dwóch liczb podanych wcześniej przez Ciebie, przypisuje wynik pewnej zmiennej, a następnie sprawdza, czy dana liczba jest większa, równa, czy też mniejsza od liczby 100.

```
1:  /* Przyklad 1.14 */
2:  /* Dokonuje mnozenia dwoch podanych liczb oraz */
3:  /* sprawdza czy wynik jest mniejszy, wiekszy, czy */
4:  /* rowny liczbie 100 */
5:  #include <stdio.h>
6:  int x, y, z;
7:  int main()
8:  {
9:       printf("Podaj pierwsza liczbe: \n");
10:      scanf("%d", &x);
11:      printf("Podaj druga liczbe: \n");
12:      scanf("%d", &y);
13:      z = x*y;
14:      if (z==100)
15:      {
16:           printf("Wartosc zmiennej z jest rowna ");
17:           printf("liczbie 100. \n");
18:      }
19:      else
20:      {
21:           if (z>100)
22:           {
23:                printf("Wartosc zmiennej z jest ");
24:                printf("wieksza od liczby 100. \n");
25:           }
```

```
26:          else
27:          {
28:               printf("Wartosc zmiennej z jest ");
29:               printf("mniejsza od liczby 100. \n");
30:          }
31:     }
32:     return 0;
33: }
```

Aby rozwiązać powyższe ćwiczenie, należy w głównej instrukcji warunkowej zagnieździć kolejną (z ang. *nesting*). Żeby łatwiej było Ci zrozumieć pojęcie *nesting*, słowa kluczowe głównej instrukcji warunkowej zostały zaznaczone pogrubioną czcionką (**wiersze 14 i 19**). W **wierszu 14** testowany jest omawiany wcześniej przypadek, kiedy wartość zmiennej `z` jest równa liczbie 100. Następnie w **wierszu 21** wprowadzona jest „zagnieżdżona" druga instrukcja warunkowa, w której sprawdzamy, czy wartość zmiennej `z` jest większa od liczby 100. Jeśli warunek jest spełniony, na ekranie zostanie wyświetlona informacja stwierdzająca ten fakt (**wiersze 23 – 24**). W przeciwnym razie wykonana zostanie funkcja `printf()`, która zawiadomi Cię o tym, że wartość zmiennej `z` jest mniejsza od liczby 100 (**wiersze 28 – 29**).

W instrukcji, w **wierszu 14**, zastosowany został nieznany Ci dotąd operator ==. W języku C oznacza on *równość*. Warto pamiętać, że znak = jest operatorem służącym do przypisywania wartości i nie należy go mylić z wprowadzonym w powyższym ćwiczeniu operatorem ==.

Znak == wchodzi w skład grupy operatorów porównania. Pozostałe z nich to:

> — większość,

< — mniejszość,

>= — większość lub równość,

<= — mniejszość lub równość,

!= — zaprzeczenie.

Inna grupa operatorów to operatory logiczne:

AND (symbol: &&) — iloczyn logiczny wyrażeń,

OR (symbol: ||) — suma logiczna wyrażeń,

NOT (symbol: !) — zaprzeczenie wyrażenia.

ĆWICZENIE

1.15 Napisz program, który wykona dodawanie dwóch liczb wpisanych z klawiatury, ale tylko w przypadku gdy NIE będą sobie równe. W przeciwnym razie dokona operacji dzielenia pierwszej przez drugą.

```
1: /* Przyklad 1.15 */
2: /* Sprawdza, czy podane liczby sa sobie rowne */
3: /* i w zaleznosci od wyniku porownania dokonuje */
4: /* ich dodawania lub dzielenia */
5: #include <stdio.h>
6: int x, y, z;
7: int main()
8: {
9:      printf("Podaj pierwsza liczbe: \n");
10:     scanf("%d", &x);
11:     printf("Podaj druga liczbe: \n");
12:     scanf("%d", &y);
13:     if (x != y)
14:     {
15:          z = x+y;
16:          printf("\nObliczono sume: %d\n", z);
17:     }
18:     else
19:     {
20:          z = x/y;
21:          printf("\nObliczono iloraz: %d\n", z);
22:     }
23:     return 0;
24: }
```

Jak widać, problem został rozwiązany poprzez użycie operatora zaprzeczenia w **wierszu 9**. W tym przypadku wyrażenie jest prawdziwe wtedy i tylko wtedy, kiedy wartość zmiennej x NIE jest równa wartości zmiennej y.

ĆWICZENIE

1.16 Napisz program, który pobiera wartości dla trzech zmiennych i wykonuje:

- *mnożenie liczby pierwszej oraz drugiej, gdy liczba pierwsza jest większa od trzeciej i liczba druga jest większa od pierwszej,*
- *dzielenie liczby drugiej przez trzecią, gdy liczba druga jest mniejsza od trzeciej albo mniejsza od pierwszej,*

- *dodawanie wszystkich trzech liczb, gdy liczba trzecia jest większa od pierwszej i liczba druga nie jest równa 5 LUB liczba druga jest większa od trzeciej oraz liczba pierwsza nie jest równa 0.*

```
1: /* Przyklad 1.16 */
2: /* Wykonuje rozne dzialania na grupie trzech zmiennych */
3: /* w zaleznosci od spelnienia danych warunkow */
4: #include <stdio.h>
5: int a, b, c, d;
6: int main()
7: {
8:      printf("Podaj pierwsza liczbe: \n");
9:      scanf("%d", &a);
10:     printf("Podaj druga liczbe: \n");
11:     scanf("%d", &b);
12:     printf("Podaj trzecia liczbe: \n");
13:     scanf("%d", &c);
14:     if (a > c && b > a)
15:     {
16:          d = a*b*c;
17:          printf("Dokonano mnozenia wartosci ");
18:          printf("trzech zmiennych, iloczyn: %d\n", d);
19:     }
20:     if (b < c || b < a)
21:     {
22:          d = b/a;
23:          printf("Dokonano dzielenia wartosci ");
24:          printf("zmiennej b przez a, iloraz: %d\n", d);
25:     }
26:     if ((c > a && b != 5) || (b > c && a != 0))
27:     {
28:          d = a+b+c;
29:          printf("Dodano wartosci wszystkich trzech ");
30:          printf("zmiennych, suma: %d\n", d);
31:     }
32:     return 0;
33: }
```

Ćwiczenie ma na celu zapoznać Cię z podstawowymi operatorami relacji oraz priorytetem każdego z nich. Jak widać, w **wierszu 26** trzeba było użyć wielu nawiasów, aby poszczególne warunki były sprawdzane w odpowiedniej kolejności. Operator iloczynu (&&) ma większy priorytet od operatora sumy (||), dlatego działanie iloczynu dwóch wyrażeń oddzielone zostało za pomocą dodatkowych nawiasów.

Poznałeś już operatory logiczne i relacje. Teraz pora na zapoznanie się z kolejną grupą — operatorami przypisania. Domyślasz się pewnie, że jednym z nich jest operator =. Poniżej zamieszczam przykłady pozostałych:

`x += 3` Operator powoduje dodanie do wartości zmiennej `x` liczby `3`

`x *= 9` Operator powoduje pomnożenie wartości zmiennej `x` przez `9`

`x /= 2` Operator powoduje podzielenie wartości zmiennej `x` przez `2`

`x -= 4` Operator powoduje odjęcie `4` od wartości zmiennej `x`

`x += y*9` Przykładowa operacja przypisania powoduje dodanie do wartości zmiennej `x` iloczynu wartości zmiennej `y` i liczby `9` (`x = x+y*9`)

Co powinieneś zapamiętać z tego cyklu ćwiczeń?

- Jak tworzyć programy w C, co to jest kod źródłowy, kompilator i linker?
- Do czego służy funkcja `printf()`, jaka jest jej konstrukcja?
- Co to są sekwencje wyjściowe?
- Co to jest zmienna i typ zmiennej, jak deklarować zmienne?
- Jakie są podstawowe typy zmiennych w języku C?
- Co to są specyfikatory konwersji (*conversion specifiers*)?
- Jaka jest różnica pomiędzy stałą literalną a symboliczną?
- Jakie są dwa sposoby deklarowania stałych?
- Do czego służy funkcja `scanf()`, jaka jest jej konstrukcja?
- Do czego służy instrukcja warunkowa `if`, jaka jest jej konstrukcja?
- Jakie operatory logiczne stosuje się w języku C?
- Jakie operatory porównania stosuje się w języku C?

Ćwiczenia do samodzielnego wykonania

ĆWICZENIE 1.

Napisz program, który wypisze na ekranie tekst: `/Jakis tekst/`.

ĆWICZENIE 2.

Zdefiniuj stałą symboliczną `WIEK` o wartości 20 przy użyciu dwóch metod poznanych w tym rozdziale.

ĆWICZENIE 3.

Napisz program, który obliczy pole koła.

Wykorzystaj stałą symboliczną.

ĆWICZENIE 4.

Napisz program, który obliczy pole walca.

Wykorzystaj stałą symboliczną.

ĆWICZENIE 5.

Napisz program, który sprawdzi, czy jesteś pełnoletni.

ĆWICZENIE 6.

Napisz program, który przekształci wartość temperatury podaną w stopniach Celsjusza na wartość w stopniach Fahrenheita.

100 stopni Celsjusza = 212 stopni Fahrenheita.

ĆWICZENIE 7.

Napisz program, który pobierze z klawiatury pięć liczb, a następnie wypisze na ekranie największą z nich.

Programowanie strukturalne

W poprzednim rozdziale zapoznałeś się z podstawowymi instrukcjami języka C. Po opanowaniu dotychczas zaprezentowanego materiału jesteś już w stanie pisać proste programy. Jednak wciąż nie zdajesz sobie sprawy, jakie możliwości daje język C.

W tym rozdziale zapoznasz się z funkcjami — podstawowym elementem każdego programu. Poznasz również różne rodzaje pętli oraz struktury. Dzięki rozeznaniu się wśród tych wszystkich pojęć oraz wykonaniu załączonych ćwiczeń zdobędziesz wiedzę pozwalającą napisać bardziej skomplikowane programy i z pewnością docenisz wysiłki twórców wspaniałego języka programowania, jakim jest C.

Programowanie strukturalne jest to pisanie programu podzielonego na wiele niezależnych podprogramów (funkcji), z których każdy wykonuje pewne określone zadanie. W ten sposób zyskuje się wiele czasu, program jest lepiej zorganizowany — w związku z tym łatwiej go później odczytać i zrozumieć jego działanie. Co więcej, w tego typu programach prościej odnajduje się ewentualne błędy — można bowiem przetestować każdą funkcję z osobna. Programowanie strukturalne jest wykorzystywane do bardziej skomplikowanych zadań. Dzięki podziałowi programu na mniejsze, niezależne podprogramy łatwiejsze okazuje się zaplanowanie jego konstrukcji i działania.

Funkcje

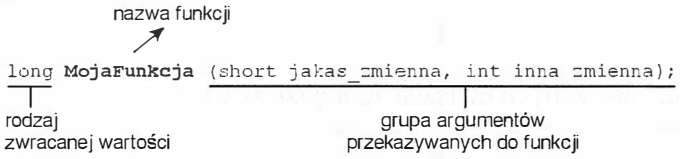
Czym jest funkcja? Funkcja to niezależny podprogram, wykonujący pewne zadanie lub zadania na potrzeby programu głównego. Dla każdej funkcji w programie muszą być spełnione poniższe warunki:

1. Funkcja musi posiadać **nazwę**.
2. Dla każdej funkcji musi być stworzony **prototyp** — model, pod którym będzie ona rozpoznawalna w programie.
3. Funkcja musi być zdefiniowana; **definicja** funkcji musi posiadać nagłówek, szkielet (zawierający wszystkie instrukcje wykonywane w obrębie danej funkcji) oraz opcjonalnie instrukcję powrotu, jeżeli funkcja zwraca jakąś wartość (część kończącą każdą funkcję).

Opcjonalnie funkcja może pobierać pewne argumenty z programu głównego, wykorzystywane przez nią w celu wykonania określonych zadań. Funkcja może zwracać jakąś wartość (np. rezultat działania wykonanego w obrębie danego podprogramu) do programu głównego.

Przykłady:

- prototyp funkcji:



  Prototyp funkcji musi być umieszczony przed wywołaniem funkcji `main()`. W podanym przykładzie widać, że argumenty to nic innego, jak definicje zmiennych zawarte w nawiasie. Jeżeli funkcja zwraca jakąś wartość, trzeba określić typ zwracanej wartości. W tym przypadku jest to typ `long`.

- definicja funkcji:

```
long MojaFunkcja (short zmienna1, int zmienna2)
{
     long ilocz /* deklaracja zmiennej, ktora bedzie zawierac
                   zwracana wartosc */
     ilocz = zmienna1 * zmienna2;
     return ilocz;                    /* instrukcja powrotu */
}
```

Nagłówek definicji danej funkcji wygląda tak samo jak jej prototyp, poza jednym drobnym szczegółem — **prototyp funkcji zakończony jest średnikiem w przeciwieństwie do jej definicji.**

W powyższym przykładzie szkielet funkcji to wszystkie instrukcje (łącznie z instrukcją powrotu) umieszczone pomiędzy klamrami.

Pierwsza instrukcja deklaruje lokalną zmienną `ilocz` w celu przechowania w niej wartości zwracanej przez funkcję. **Zmienna lokalna jest rozpoznawana jedynie w obrębie danej funkcji.**

Druga instrukcja to przypisanie zmiennej ilości wartości równej iloczynowi dwóch argumentów przekazanych **z funkcji głównej**. Trzecia instrukcja to instrukcja powrotu, która zwraca wartość zawartą w zmiennej `ilocz` **do funkcji głównej**.

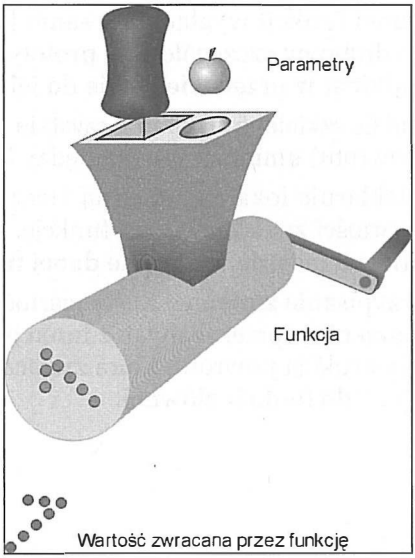
- wywołanie funkcji:

```
long x;
int main()
{
.............
.............
x = MojaFunkcja (zmienna1, zmienna2);
.............
.............
}
```

Sposób wywoływania funkcji zależy od jej typu. Zanim ją wywołamy, musimy wiedzieć, czy przyjmuje ona argumenty z programu głównego i czy zwraca jakąś wartość. W naszym przykładzie funkcja zarówno przyjmuje argumenty, jak i zwraca wartość. W tym przypadku trzeba zadeklarować zmienną globalną (przed wywołaniem funkcji `main()`), która będzie rozpoznawana w obrębie programu głównego; jej typ musi odpowiadać rodzajowi zwracanej przez funkcję wartości (w powyższym przykładzie jest to typ `long`). Każda funkcja, która zwraca wartość, **musi** być wywoływana przy jednoczesnym przypisaniu danej wartości pewnej zmiennej globalnej (w powyższym przykładzie zmiennej `x`).

W przypadku funkcji, które nie zwracają żadnej wartości, mechanizm wywołania ogranicza się tylko do wpisania nazwy danej funkcji oraz (opcjonalnie) przyjmowanych argumentów. Na przykład:

```
........
........
PewnaFunkcja (argument);
........
```



Jak ma wyglądać prototyp i nagłówek definicji takiej funkcji?

Otóż w polu, w którym umieszczany jest typ zwracanej wartości, należy wpisać void.

Przykład:

```
void PewnaFunkcja (int argument);
```

ĆWICZENIE

2.1

Napisz program, który pobiera z klawiatury dwie liczby, wykonuje ich mnożenie i umieszcza ich iloczyn w dowolnej zmiennej (skorzystaj z odpowiednio skonstruowanej funkcji).

```
1: /* Przyklad 2.1 */
2: /* Proste wykorzystanie funkcji */
3: #include <stdio.h>
4: long MojaFunkcja( int x, int y );
5: int liczba1, liczba2;
6: long wynik;
7: int main()
8: {
9:      printf("Podaj pierwsza liczbe: \n");
10:     scanf("%d", &liczba1);
11:     printf("Podaj druga liczbe: \n");
12:     scanf ("%d", &liczba2);
13:     wynik = MojaFunkcja(liczba1, liczba2);
```

```
14:     printf("Iloczyn dwoch liczb: %ld\n", wynik);
15:     return 0;
16: }
17: long MojaFunkcja (int x, int y)
18: {
19:       long z;
20:       z = x*y;
21:       return z;
22: }
```

Jak widać, program ten jest trochę bardziej skomplikowany niż poprzednie. W **wierszu 4** umieszczony jest prototyp funkcji. Ma ona zwracać wartość typu `long` oraz pobierać dwa argumenty typu `int`. Argumenty w nawiasach to deklaracje zmiennych **lokalnych**, rozpoznawalnych tylko w obrębie danej funkcji. Z tego powodu trzeba zadeklarować dwie dodatkowe zmienne globalne tego samego typu (`liczba1` oraz `liczba2`), które zostaną przekazane jako argumenty dla naszej funkcji. W **wierszu 6** deklarowana jest zmienna globalna, która będzie rozpoznawalna w programie głównym i która będzie ostatecznie przechowywać wynik operacji mnożenia (zmienna `wynik`). Zauważ, że musi ona być tego samego typu co wartość zwracana przez funkcję. **Wiersze 7 – 12** zawierają wywołanie `main()` oraz pobranie dwóch liczb. W **wierszu 13** zostaje wywołana funkcja `MojaFunkcja`. W tym momencie wykonanie programu zostaje przerwane i wykonywane są wszystkie instrukcje naszej funkcji — **wiersze 17 – 22**. W tym momencie zmienne `liczba1` i `liczba2` „przekształcane" są w zmienne lokalne `x` i `y`. W **wierszu 19** następuje deklaracja zmiennej lokalnej `z` typu `long` — będzie ona użyta w celu „tymczasowego" przechowania wartości zwracanej (iloczynu `x` i `y`). Ostatecznie, po zakończeniu wykonywania instrukcji funkcji `MojaFunkcja`, program zostaje wznowiony w **wierszu 13**. W tym momencie wartość zwracana zostaje przypisana zmiennej globalnej `wynik`. Pozostaje jeszcze tylko wypisanie rezultatu na ekranie monitora (funkcja `printf()` — **wiersz 14**).

Pewnie zastanawiasz się, jaka jest różnica między zmiennymi globalnymi a lokalnymi. Zmienne globalne są widoczne w każdej funkcji programu, natomiast lokalne tylko w obrębie funkcji, w której są zdefiniowane. Zalecane jest używanie zmiennych lokalnych, ponieważ są one usuwane z pamięci w momencie zakończenia wykonywania danej funkcji. Zmienne globalne istnieją natomiast aż do zakończenia programu.

ĆWICZENIE

2.2

Napisz program, który pobiera jedną liczbę z klawiatury i w zależności od tego, czy jest ona większa od liczby 20, czy też mniejsza, wypisuje na ekranie odpowiednią informację (**uwaga:** zarówno instrukcja warunkowa, jak i oba wywołania funkcji `printf()` muszą być egzekwowane w obrębie funkcji odpowiednio skonstruowanej do tego celu).

```
1: /* Przyklad 2.2 */
2: /* Przyklad funkcji, ktora nie zwraca wartosci */
3: #include <stdio.h>
4: void funkcja(int x);
5: int liczba;
6: int main()
7: {
8:      printf("Podaj liczbe: \n");
9:      scanf("%d", &liczba);
10:     funkcja(liczba);
11:     return 0;
12: }
13: void funkcja(int x)
14: {
15:      if (x < 20)
16:      {
17:           printf("Liczba jest mniejsza od 20.\n");
18:      }
19:      else if ( x==20)
20:           {
21:                printf("Liczba jest rowna 20.\n");
22:           }
23:           else
24:           {
25:                printf("Liczba wieksza od 20.\n");
26:           }
27: }
```

W tym ćwiczeniu masz okazję zapoznać się z przykładem funkcji, która nie zwraca żadnej wartości. W tym przypadku funkcja jedynie pobiera jeden argument (może również więcej) i korzystając z przekazanej przez niego wartości, wykonuje odpowiednie operacje.

ĆWICZENIE

2.3

Napisz program, który dokona obliczenia sumy dwóch dowolnych wartości, zapisze wynik w dowolnej zmiennej, a następnie wyświetli odpowiednią informację na ekranie (wszystkie działania muszą być wykonane w funkcji odpowiednio skonstruowanej do tego celu).

```
1: /* Przyklad 2.3 */
2: /* Przyklad funkcji, ktora nie zwraca wartosci, */
3: /* ani nie pobiera zadnych argumentow */
4: #include <stdio.h>
5: void funkcja();
6: int x;
7: int main()
8: {
9:       funkcja();
10:      return 0;
11:}
12:void funkcja()
13:{
14:      x = 10+40;
15:      printf("Suma stalych literalnych 10 i 40 ");
16:      printf("wynosi: %d \n", x);
17:}
```

Rozwiązanie jest ekstremalnie łatwe. Ćwiczenie ma za zadanie zapoznać Cię z przykładem funkcji, która ani nie pobiera żadnych argumentów z programu głównego, ani nie zwraca żadnej wartości.

ĆWICZENIE

2.4 Napisz program, który pobierze trzy wartości, przypisze je pewnym zmiennym, a te następnie przekaże jako argumenty do pewnej funkcji. Później, w zależności od spełnienia poniższego warunku, zwróci do programu głównego odpowiednią wartość:

- jeżeli wartość pierwszej zmiennej pomnożonej przez wartość drugiej zmiennej będzie większa od 100-krotności trzeciej zmiennej — zwrócona zostanie zmodyfikowana wartość pierwszej zmiennej; w przeciwnym razie zwrócona zostanie wartość drugiej zmiennej.

```
1: /* Przyklad 2.4 */
2: /* Przyklad funkcji, ktora moze zwracac wartosci */
3: /* roznych zmiennych w zaleznosci od spelnienia */
4: /* poszczegolnych warunkow */
5: #include <stdio.h>
6: int funkcja(int x, int y, int z);
7: int a, b, c, wynik;
8: int main()
9: {
10:     printf("Pierwsza liczba: \n");
11:     scanf("%d", &a);
12:     printf("Druga liczba: \n");
```

```
13:     scanf("%d", &b);
14:     printf("Trzecia liczba: \n");
15:     scanf("%d", &c);
16:     wynik = funkcja(a, b, c);
17:     printf("Wynik: %d \n", wynik);
18:     return 0;
19: }
20: int funkcja(int x, int y, int z)
21: {
22:      if ((x*=y) > (z*=100))
23:          return x;
24:      else
25:          return y;
26: }
```

W rozwiązaniu tego ćwiczenia została zaprezentowana możliwość zwracania różnych wartości przez funkcję. Oczywiście, zastosowano tutaj — mam nadzieję, że doskonale Ci znaną — instrukcję warunkową.

W **wierszu 22** sprawdzany jest odpowiedni warunek i wykonywane są jednocześnie dwie instrukcje przypisania. Wartość zmiennej `x` mnożona jest przez wartość zmiennej `y`, a iloczyn zostaje przypisany zmiennej `x` (modyfikuje jej dotychczasową wartość).

Otrzymana w ten sposób zmodyfikowana wartość zmiennej `x` porównywana jest z wartością zmiennej `z`, która z kolei została pomnożona przez sto, a wynik iloczynu jest przypisany do zmiennej `z`. Jeżeli warunek zostanie spełniony, zwracana jest zmodyfikowana już wartość `x`.

Jak myślisz, czy po powrocie z funkcji zmienna `c` pozostanie taka sama?[1]

[1] Wartość zmiennej `c` pozostaje oczywiście taka sama, mimo że wartość zmiennej "`z`" pozostała zmodyfikowana. Zmienna `z` jest zmienną lokalną dla naszej funkcji, nie jest rozpoznawana w programie głównym ani nie jest zwracana. W przypadku zmiennej `x` jej zmodyfikowana wartość jest zwracana z funkcji, czyli przypisywana zmiennej globalnej "`wynik`". Uwaga: jeżeli przekazujemy zmienne jako argumenty do funkcji, pobieramy tylko ich wartość i przypisujemy ją do zmiennych lokalnych danej funkcji (tych, które są umieszczane w prototypie).

Pętle w języku C

Wstęp do tablic

Zanim rozpocznę opisywanie działania pętli, powinienem zapoznać Cię z pojęciem tablic, które są bardzo często wykorzystywane w tego typu konstrukcjach.

Tablica jest grupą komórek pamięci. Każda komórka posiada swój numer (indeks) i każdej z nich można przypisywać różne wartości. Tablice stosujemy w przypadku, gdy chcemy przechowywać w jednym miejscu grupę zmiennych tego samego typu — np. wiek wszystkich osób w klasie (każdy uczeń ma przydzielony numer w dzienniku). Każda tablica musi być zadeklarowana, np.:

```
int wiek_ucznia[31];
```

W powyższym przykładzie zadeklarowana została tablica złożona z 31 elementów **(indeks pierwszego elementu tablicy jest równy 0, ostatniego — 30)**, każdemu z nich można przypisać wartości typu `int`, np.:

```
Int wiek_ucznia[31];
Wiek_ucznia[25] = 15;
Wiek_cznia[0] = 14;
```

W jaki sposób można szybko przypisać różne wartości wszystkim elementom danej tablicy?

Oczywiście najłatwiejszym i najpopularniejszym sposobem jest użycie pętli `for`.

Pętla for

Pętla `for` pozwala na wielokrotne powtarzanie pewnego ciągu instrukcji. Z pewnością będziesz stosować tę konstrukcję (oraz inne rodzaje pętli opisane w dalszej części książki) w większości tworzonych przez siebie programów.

Konstrukcja pętli `for`:

```
for (n=0; n<31; n++)
      Instrukcja;

for (n=0; n<31; n++)
      {
```

```
        instrukcja 1;
        instrukcja 2;
        .............;
        instrukcja n;
    }
```

Pętlę wywołuje się przy użyciu słowa kluczowego `for`, po którym należy określić mechanizm działania pętli. Zmienna `n` jest tzw. licznikiem, służy ona do określania liczby powtórzeń danego ciągu instrukcji. Kiedy pisze się programy w C, trzeba pamiętać, że przed użyciem pętli `for` należy zadeklarować taką zmienną.

Za pomocą pierwszego wyrażenia w nawiasie określa się wstępną wartość licznika (zazwyczaj `n=0`).

Za pomocą drugiego wyrażenia określany jest warunek (w naszym przypadku używamy wyrażenia z operatorem porównania — określamy maksymalną wartość licznika). Jeżeli dany warunek zostanie spełniony, pętla będzie kontynuowała swoje działanie. W przeciwnym razie pętla ulegnie zakończeniu (nie nastąpią kolejne powtórzenia).

Trzecie wyrażenie służy do określania zmiany wartości licznika przy każdym powtórzeniu pętli (zwykle licznik ulega zwiększeniu; w naszym przypadku ulega on zwiększeniu o wartość równą 1).

Pozostała część to instrukcja lub ciąg instrukcji, które mają być wykonywane przy każdorazowym powtórzeniu pętli.

Pętla while

Można powiedzieć, że pętla `while` jest uproszczoną wersją pętli `for`. W nawiasach, po słowie kluczowym `while`, określamy jedynie warunek. Dopóki będzie on spełniany, dopóty powtarzany będzie blok odpowiednich instrukcji.

Pętlę `while` konstruujemy w następujący sposób:

```
while ( warunek )
    {
        instrukcja 1;
        instrukcja 2;
        .............;
        instrukcja n;
    }
```

Zastanawiasz się pewnie, kiedy stosować pętlę `for`, a kiedy `while`. Pętli `for` używa się wtedy, gdy trzeba określić liczbę powtórzeń danego

bloku instrukcji. Natomiast pętlę while stosuje się wtedy, gdy należy określić jedynie warunek. Dopóki będzie on spełniany, dopóty odpowiednie instrukcje będą powtarzane.

Pętla do...while

Do omówienia pozostała jedynie pętla do...while, która (w przeciwieństwie do for i while) sprawdza warunek dopiero po wykonaniu bloku instrukcji.

Pętlę do...while konstruujemy w następujący sposób:

```
do
  {
       instrukcja 1;
       instrukcja 2;
       ............;
       instrukcja n;
  }
while (warunek);
```

ĆWICZENIE

2.5 Napisz program, który 10 razy wypisze na ekranie napis „kocham lato".

```
1: /* Przykład 2.5 */
2: /* Proste wykorzystanie petli for */
3: #include <stdio.h>
4: int n;
5: int main()
6: {
7:     for (n=0; n<10; n++)
8:         printf("Kocham lato \n");
9:     return 0;
10:}
```

W **wierszu 4** zadeklarowana zostaje zmienna n, która ma służyć jako licznik. W **wierszu 7** wywoływana jest pętla for. Powtarzana jest ona 10 razy (0 – 9). Przy każdorazowym powtórzeniu licznik zwiększany jest o 1 (n++). Kiedy licznik osiągnie wartość 10 (warunek nie zostanie spełniony), pętla zostanie przerwana i wykonywane będą pozostałe instrukcje programu (począwszy od **wiersza 9**).

ĆWICZENIE

2.6 Napisz program, który wyzeruje tablicę złożoną z 40 elementów.

```
1: /* Przyklad 2.6 */
2: /* Przyklad uzycia petli for z wykorzystaniem */
3: /* tablic */
4: #include <stdio.h>
5: int tablica[40];
6: int licznik;
7: int main()
8: {
9:       for (licznik=0; licznik < 40; licznik++)
10:          tablica[licznik] = 0;
11:      printf("Tablica zostala wyzerowana\n");
12:      return 0;
13: }
```

Jak widać, rozwiązanie problemu jest nader proste. Deklaruje się tablicę złożoną z 40 elementów (pamiętaj, że indeks pierwszego elementu każdej tabeli jest równy 0), a następnie wywołuje pętlę `for` powtarzaną 40 razy. Przy każdorazowym powtórzeniu pętli wykonywana jest instrukcja przypisania wartości 0 elementowi tablicy o numerze równym wartości, która w danej chwili umieszczona jest w zmiennej `licznik`.

ĆWICZENIE

2.7 Napisz program, który przypisuje wartość zero co piątemu elementowi tablicy (stuelementowej).

```
1: /* Przyklad 2.7 */
2: /* Przyklad petli for z wykorzystaniem tablic */
3: #include <stdio.h>
4: int tablica[100];
5: int n;
6: int main()
7: {
8:       for (n=0; n<100; n += 5)
9:          tablica[n] = 0;
10:      printf("Tablica zostala wyzerowana\n");
11:      return 0;
12: }
```

W **wierszu 4** deklarowana jest tablica stuelementowa. W **wierszu 8** zostaje wywołana pętla `for`, która przy każdym powtórzeniu powiększa wartość licznika o 5 (`n += 5`). W ten sposób zerowany jest co piąty element tablicy.

ĆWICZENIE

2.8 Napisz program, który poprosi Cię o wprowadzenie z klawiatury średniej ocen każdego ucznia w Twojej klasie. Następnie program powinien odpowiednio przypisać podane wartości poszczególnym elementom zadeklarowanej wcześniej tablicy. Na koniec spraw, aby została obliczona średnia ocen całej klasy.

```
1: /* Przyklad 2.8 */
2: /* Przyklad uzycia petli for z wykorzystaniem */
3: /* funkcji */
4: #include <stdio.h>
5: float srednia_ucznia, suma_sr, sr_klasy;
6: int n;
7: float JakaSrednia (float x, int y);
8: int main()
9: {
10:     float uczniowie[21];
11:     suma_sr = 0;
12:     for (n=0; n<21; n++)
13:     {
14:          printf("Podaj srednia ucznia numer %d \n", n);
15:          scanf("%f", &srednia_ucznia);
16:          uczniowie[n] = srednia_ucznia;
17:          suma_sr += uczniowie[n];
18:     }
19:     sr_klasy = JakaSrednia (suma_sr, n);
20:     printf("Srednia ocen Twojej klasy wynosi: %f",
21:          sr_klasy);
22:     return 0;
23: }
24: float JakaSrednia( float x, int y )
25: {
26:      float z;
27:      z = x/y;
28:      return z;
29: }
```

Rozwiązanie tego zadania jest nieco bardziej skomplikowane, problemem może być zrozumienie działania powyższego programu. W **wierszu 10** jest zdefiniowana tablica, w której umieszczone zostaną wartości średniej ocen każdego ucznia. W **wierszu 5** definiuje się trzy zmienne typu `float`: `srednia_ucznia` (przechowuje tymczasowo wartości wprowadzone z klawiatury), `suma_sr` (przechowuje sumę średnich wszystkich uczniów), `sr_klasy` (przechowuje średnią ocen całej klasy). W **wierszu 11** zmiennej `suma_sr` zostaje przypisana wartość 0. Następuje wywołanie pętli `for`, która jest powtarzana 21 razy. Przy każdym

powtórzeniu program przerywa i czeka na wprowadzenie odpowiedniej wartości dla kolejnych elementów tablicy. Wartości te są dodawane do zmiennej suma_sr. Po zakończeniu wykonywania pętli wywoływana jest funkcja, która oblicza średnią ocen dla całej klasy.

ĆWICZENIE

2.9 Zmodyfikuj poprzedni program, tak aby można było wpisywać z klawiatury jedynie średnią ocen w zakresie 1,00 – 6,00.

```
1: /* Przyklad 2.9 */
2: /* Modyfikacja przykladu 2.8 */
3: #include <stdio.h>
4: float sr_ucz, suma_sr, sr_klasy;
5: int n;
6: float JakaSrednia (float x, int y);
7: int main()
8: {
9:      float uczniowie[21];
10:     suma_sr = 0;
11:     for (n=0; n<21; n++)
12:     {
13:          printf("Podaj srednia ucznia numer %d ", n);
14:          printf("z zakresu 1.00-6.00  \n");
15:          scanf("%f", &sr_ucz);
16:          if (sr_ucz < 1 || sr_ucz > 6)
17:              exit(0);
18:          uczniowie[n] = sr_ucz;
19:          suma_sr += uczniowie[n];
20:     }
21:     sr_klasy = JakaSrednia (suma_sr, n);
22:     printf("Srednia ocen Twojej klasy ");
23:     printf("wynosi: %f", sr_klasy);
24:     return 0;
25: }
26: float JakaSrednia(float x, int y)
27: {
28:      int z;
29:      z = x/y;
30:      return z;
31: }
```

ĆWICZENIE

2.10 Napisz program wykorzystujący pętlę while, który policzy od 25 do 200.

```
1: /* Przyklad 2.10 */
2: /* Przyklad uzycia petli while */
3: #include <stdio.h>
```

```
4: int licznik;
5: int main()
6: {
7:     licznik = 25;
8:     while (licznik <=200)
9:     {
10:         printf( "Licznik: %d \n", licznik );
11:         licznik++;
12:    }
13:    return 0;
14: }
```

Jak widać w powyższym przykładzie, w przypadku pętli `while` samodzielnie trzeba określić wstępną wartość licznika (**wiersz 7**) oraz jego zmianę (**wiersz 11**). Warto pamiętać, że korzystając z pętli `for`, grupuje się trzy wyrażenia (wstępna wartość, warunek, zmiana) w obrębie jednej pary nawiasów.

ĆWICZENIE

2.11 Napisz program, który będzie czytał dodatnie liczby z klawiatury i kolejno je sumował. Jedynym sposobem przerwania nieskończonej pętli ma być wpisanie liczby 1. Spraw, aby program wyświetlił na ekranie sumę wszystkich wprowadzonych liczb.

```
1: /* Przyklad 2.11 */
2: /* Przyklad petli nieskonczonej z wykorzystaniem */
3: /* instrukcji break */
4: #include <stdio.h>
5: unsigned int liczba;
6: unsigned long suma;
7: int main()
8: {
9:    for( ; ; )
10:   {
11:       printf("Podaj dodatnia liczbe z zakresu ");
12:       printf("2 - 65535, wpisz 1, aby zakonczyc\n");
13:       scanf("%u", &liczba);
14:           if (liczba == 1)
15:               break;
16:           else
17:               suma += liczba;
18:   }
19:   printf("Suma wszystkich wpisanych liczb ");
20:   printf("wynosi: %lu\n", suma);
21:   return 0;
22: }
```

Jak widać, i to ćwiczenie jest trochę bardziej skomplikowane. Do pełnego zrozumienia działania powyższego programu potrzeba znajomości dwóch pojęć: pętla nieskończona, nagłe przerwanie pętli.

Pętla nieskończona może powtarzać się bez końca. Jej charakterystyczną cechą jest brak jakichkolwiek warunków oraz liczników. Przerwanie takiej pętli jest możliwe poprzez wykorzystanie instrukcji `break` (w naszym przykładzie: **wiersz 15**). Pętlę nieskończoną można wywołać w sposób przedstawiony w powyższym przykładzie (**wiersz 9**). Jak widać, pola przeznaczone dla wstępnej wartości licznika, warunków oraz zmiany wartości licznika zawierają jedynie puste znaki oddzielone średnikami.

W przypadku pętli `while` oraz `do...while` nieskończoną liczbę powtórzeń można wywołać w następujący sposób:

```
while (1)
{
.....
.....
}

do
{
.....
.....
} while (1)
```

Instrukcja switch

Instrukcja `switch` jest instrukcją wyboru, która pozwala sprawdzić wartość zmiennej `i` i w zależności od wyniku wykonać różne działania.

Konstrukcja instrukcji `switch`:

```
switch (wyrazenie)
{
     case wartosc 1: { blok instrukcji break;};
     case wartosc 2: { blok instrukcji break;};
     case wartosc 3: { blok instrukcji break;};
     case wartosc n: { blok instrukcji break;};
     default: { blok instrukcji };
}
```

W polu wyrazenie najczęściej wpisuje się nazwę jakiejś zmiennej lub funkcję, która zwraca wartość typu: int, long lub char. W zależności od wartości zwróconej przez wyrażenie wykonywany jest jeden z podanych bloków instrukcji. Jeżeli wartość nie spełni żadnego warunku, wykonany zostanie blok instrukcji po słowie kluczowym default.

ĆWICZENIE

2.12 Napisz program, który pobierze z klawiatury wartość od 1 – 3, a następnie, w zależności od podanej wartości, wykona odpowiedni blok instrukcji.

```
1: /* Przyklad 2.12 */
2: /* Przyklad demonstruje uzycie instrukcji switch */
3: #include <stdio.h>
4: int wartosc;
5: main()
6: {
6:     printf("Wpisz wartosc od 1-3: ");
7:     scanf("%d", &wartosc);
8:     switch (wartosc)
9:     {
10:         case 1:
11:         {
12:             printf("Wpisales 1");
13:             break;
14:         }
15:         case 2:
16:         {
17:             printf("Wpisales 2");
18:             break;
19:         }
20:         case 3:
21:         {
22:             printf ("Wpisales 3");
23:             break;
24:         }
25:         default:
26:         {
27:             printf ("Cos ty wpisal...?");
28:         }
29:     }
30:     return 0;
31: }
```

W **wierszu 7** wykonywana jest funkcja scanf(), która pobiera z klawiatury pewną wartość. Następnie, w zależności od podanej wartości, wykonywany jest odpowiedni blok instrukcji. W każdym bloku instrukcji zawarta jest instrukcja break, która przerywa działanie switch.

W przypadku gdy wpisana wartość nie należy do zakresu 1 – 3, wykonywany jest blok instrukcji po słowie kluczowym `default`. Spróbuj zmodyfikować powyższy program, wymazując wszystkie instrukcje `break`, a następnie skompiluj i uruchom program. Co się dzieje?

Co powinieneś zapamiętać z tego cyklu ćwiczeń?

- Co to jest funkcja?
- Jak wygląda prototyp i definicja funkcji oraz jak ją wywoływać?
- Co to są argumenty funkcji i co to jest wartość zwracana?
- Jaka jest różnica pomiędzy zmiennymi globalnymi a lokalnymi?
- Co to są tablice, jak je deklarować oraz jak przypisywać im wartości?
- Jaka jest konstrukcja pętli `for` oraz do jakich celów można ją wykorzystać?
- Jaka jest konstrukcja pętli `while` i w jaki sposób ona działa?
- Jaka jest konstrukcja pętli `while...do` i w jaki sposób ona działa?
- Jaka jest różnica pomiędzy trzema poznanymi rodzajami pętli?
- Co to jest pętla nieskończona i jak można ją wywołać?
- Do czego służy instrukcja `break`?
- Na jakiej zasadzie działa instrukcja warunkowa `switch`?
- Jaka jest konstrukcja instrukcji warunkowej `switch`?
- Kiedy wykonywany jest blok instrukcji po słowie kluczowym `default` w instrukcji warunkowej `switch`?

Ćwiczenia do samodzielnego wykonania

ĆWICZENIE 1.

Napisz program, który wykorzysta funkcję do obliczenia różnicy dwóch zmiennych.

ĆWICZENIE 2.

Napisz program, który wyświetli na ekranie napis „Kocham język C".

Użyj funkcji, która nie pobiera żadnych argumentów i nie zwraca żadnej wartości.

ĆWICZENIE 3.

Napisz program, który przypisze każdemu elementowi dowolnej tablicy wartość 1.

ĆWICZENIE 4.

Napisz program, który pobierze z klawiatury 10 wartości, a następnie obliczy ich sumę i wyświetli odpowiednią informację na ekranie.

ĆWICZENIE 5.

Zmodyfikuj program z ćwiczenia 2.2 tak, aby wykorzystana została pętla `while` zamiast pętli `for`.

ĆWICZENIE 6.

Za pomocą instrukcji `switch` stwórz proste menu złożone z 8 opcji. Przy wybieraniu kolejnych opcji na ekranie powinien wyświetlać się odpowiedni tekst.

Język C dla wtajemniczonych

Jeżeli dobrze opanowałeś cały materiał przedstawiony w poprzednich rozdziałach książki, możesz nazywać siebie programistą wtajemniczonym i przejść do kolejnych zagadnień. Z tego rozdziału dowiesz się m.in., czym są wskaźniki, tablice wielowymiarowe, łańcuchy znaków i struktury. Pojęcia te są trochę trudniejsze i niełatwo je dobrze wyjaśnić. Dlatego przed rozpoczęciem lektury zalecam pochłonięcie co najmniej trzech dużych filiżanek kawy lub pięciu puszek napojów energetycznych (moja dzienna dawka podczas pisania tej książki — włożyłem w nią dużo serca).

Jeśli odczuwasz już działanie kofeiny, zapraszam do wykonania następnych kroków w niesamowitym świecie języka C.

Tablice wielowymiarowe

Z pojęciem tablicy spotkałeś się już w jednym z poprzednich rozdziałów (patrz: „Pętla for"). Tablice omawiane do tej pory posiadały tylko jeden wymiar — jeden indeks (numer w nawiasach). Pamiętaj, że indeksy numeruje się, począwszy od zera — numer pierwszego elementu jest równy zero.

Tablice wielowymiarowe mają więcej niż jeden indeks. Przykład:

```
float MojaSzkola[24][36];
```

Załóżmy, że komórki powyższej tablicy to uczniowie w Twojej szkole. Pierwszy indeks to klasy, drugi natomiast to uczniowie w każdej z tych klas. W szkole są 24 klasy, a w każdej z nich 36 uczniów.

Rozmiar tablicy to: 24×36 = 864 (liczba uczniów w szkole — 864). Jeśli chcesz przypisać wartość jednej z komórek, np. średnią ocen 1. ucznia z 1. klasy w tablicy, musisz napisać:

```
MojaSzkola[0][0] = 5.6; /*pierwszy element-indeks[0]*/
```

Natomiast przypisanie średniej ocen dla 21. ucznia w 13. klasie powinno się odbywać w następujący sposób:

```
MojaSzkola[12][20] = 4.3;
```

Dla 25. ucznia w 11. klasie:

```
MojaSzkola[10][24] = 3.4;
```

ĆWICZENIE

3.1 Napisz program, który poprosi Cię o wpisanie średniej ocen dla każdego ucznia w szkole (4 klasy, a w każdej z nich 5 uczniów) oraz wyświetli wpisane wartości na ekranie.

```
1: /* Przyklad 3.1 */
2: /* Przyklad uzycia tabeli wielowymiarowej */
3: #include <stdio.h>
4: float MojaSzkola[4][5];
5: int licz1, licz2;
6: int main()
7: {
8:      for (licz1 = 0; licz1 < 4; licz1++)
9:      {
10:         for (licz2 = 0; licz2 < 5; licz2++)
11:         {
12:              printf("\n\nWpisz srednia ocen %d-ego ",
13:                    licz2+1);
14:              printf("ucznia %d-ej klasy: ", licz1+1);
15:              scanf("%f", &MojaSzkola[licz1][licz2]);
16:         }
17:     }
18:     for (licz1 = 0; licz1 < 4; licz1++)
19:     {
```

```
20:          for (licz2 = 0; licz2 < 5; licz2++)
21:          {
22:              printf("\n\nSrednia ocen %d-ego ucznia\n ",
23:                     licz2+1);
24:             printf("%d-ej klasy wynosi: %f\n", licz1+1,
25:                     MojaSzkola[licz1][licz2]);
26:          }
27:     }
28:     return 0;
29: }
```

Być może zastanawiasz się, jak to się dzieje, że wartości są przypisywane odpowiednim komórkom w tablicy. Postaram się objaśnić tę kwestię, opisując wszystkie czynności krok po kroku.

Wiersz 4: deklarujesz wielowymiarową tablicę dwudziestoelementową typu `float`.

Wiersz 5: deklarujesz dwie zmienne, które mają posłużyć jako licznik pętli oraz wskazywać na kolejne elementy tabeli.

Wiersz 8: wywołujesz nadrzędną pętlę `for`, która liczy komórki pierwszego indeksu (numer klasy).

Wiersz 10: wywołujesz podrzędną (zagnieżdżoną) pętlę `for`, która liczy komórki drugiego indeksu (numer ucznia).

Wiersze 12 – 15: wykonywane są instrukcje dla każdego elementu tablicy (ucznia).

Wiersze 18 – 27: przypisane wcześniej wartości są wyświetlane na ekranie.

Prześledźmy poszczególne kroki pętli:

Kiedy `licz1 = 0`, wywoływana jest po raz pierwszy podrzędna pętla `for`, która zostaje powtórzona 5 razy (`licz2 = 0`, `licz2 = 1`, …. `licz2 = 4`).

Dla `licz1 = 1` pętla `for` powtarzana jest 5 razy (`licz2 =0`, … `licz2 =4`).

Dla `licz1 = 3` odbywa się ostatnie, pięciokrotne powtórzenie podrzędnej pętli `for`.

Wyświetlenie na ekranie wprowadzonych wcześniej danych odbywa się w identyczny sposób.

ĆWICZENIE

3.2

Napisz program, który wyświetli na ekranie liczbę dni w każdym miesiącu roku przestępnego oraz nieprzestępnego.

```
1: /* Przyklad 3.2 */
2: /* Przyklad wykorzystania tabeli wielowymiarowej */
3: /* w polaczeniu z instrukcja warunkowa if */
4: #include <stdio.h>
5: int a, b;
6: int main()
7: {
8:  int tablica[2][12] = {{31,28,31,30,31,30,31,31,
9:  30,31,30,31}, {31,29,31,30,31,30,31,31,30,31,30,
10:  31}};
11:
12:      for (a = 0; a < 2; a++)
13:      {
14:           for (b = 0; b < 12; b++)
15:           {
16:                if (a == 1)
17:                {
18:                     printf("\nMiesiac %d roku ", b+1);
19:                     printf("przestepnego ma %d dni.",
20:                          tablica[a][b]);
21:                }
22:                else
23:                {
24:                     printf("\nMiesiac %d roku ", b+1);
25:                     printf("nieprzestepnego ma %d dni.",
26:                          tablica[a][b]);
27:                }
28:           }
29:      }
30:      return 0;
31: }
```

Rozwiązanie powyższego ćwiczenia jest jeszcze bardziej skomplikowane. Nie dość, że zagnieżdżamy pętlę `for` w innej pętli `for`, to jeszcze dodajemy instrukcję warunkową `if`. W **wierszu 8** deklarowana jest dwudziestoczteroelementowa tablica dwuwymiarowa `tablica[][]`, jednocześnie zostają jej przypisane odpowiednie wartości. Ta tablica przechowuje liczbę dni w każdym miesiącu, zarówno roku przestępnego, jak i nieprzestępnego. Następnie w **wierszu 12** zostaje wywołana pętla `for`, w której zagnieżdżona jest kolejna pętla (mechanizm ten działa na tej samej zasadzie, co konstrukcja pętli z poprzedniego ćwiczenia).

Dodatkowym utrudnieniem jest instrukcja warunkowa `if` w **wierszu 16**, która ułatwia wypisanie odpowiednich wartości na ekranie (sprawdza, czy dany rok jest przestępny, czy też nie).

Wskaźniki

Każda zmienna zadeklarowana w programie posiada swój adres w pamięci komputera. Jeżeli znany jest adres danej zmiennej, można zadeklarować drugą zmienną, która będzie zawierać adres pierwszej. W takim przypadku druga zmienna wie, gdzie jest umiejscowiona w pamięci pierwsza zmienna. Można stwierdzić, że druga zmienna wskazuje na pierwszą — jest wskaźnikiem. Jeśli to wydaje się zawiłe, przedstawię przykład, który ułatwi zrozumienie tego zagadnienia.

Janek, Zbych i Ola chodzą do tej samej szkoły w mieście Jakaśdziura. Janek i Zbych są dobrymi kumplami. Zbych poznał Olę na lekcji angielskiego, natomiast Janek jej nie zna. Ola wpadła Jankowi w oko po tym, jak ją raz zobaczył na korytarzu rozmawiającą ze Zbychem. Po lekcjach Janek podchodzi do Zbycha i pyta:

— Zbychu, co to za laska, z którą rozmawiałeś na korytarzu, jak ma na imię?

— Ma na imię Ola. Niezłe ma nogi, no nie?

— Jasne, nieziemskie, znasz może jej adres?

— Tak, Ogrodowa 28.

Podsumujmy: Zbych jest wskaźnikiem do Oli. Zna jej imię oraz wie, gdzie dziewczyna mieszka (zna jej adres).

Wskaźnik do zmiennej trzeba zadeklarować. Robi się to w następujący sposób:

```
int *wskaznik;
```

Jak widać, dodaje się jedynie gwiazdkę przed nazwą zmiennej.

Do danego wskaźnika trzeba dodatkowo przypisać odpowiedni adres zmiennej, na którą ma on wskazywać. Adres zmiennej uzyskujemy, stosując znak `&`. Przypisanie adresu zmiennej do wskaźnika wykonuje się tak:

```
wskaznik = &zmienna;
```

ĆWICZENIE

3.3 Napisz program, w którym zadeklarujesz wskaźnik do zmiennej oraz przypiszesz mu odpowiedni adres. Następnie spraw, aby wartość zmiennej została wypisana na ekranie na dwa sposoby: poprzez bezpośrednie odniesienie do zmiennej oraz poprzez wskaźnik. Spraw również, aby na ekranie został wyświetlony adres zmiennej (na dwa sposoby).

```
1: /* Przyklad 3.3 */
2: /* Przyklad prezentujacy dzialanie wskaznikow */
3: #include <stdio.h>
4: int Ola;
5: int *Zbych;
6: int main()
7: {
8:      Ola = 180;
9:      Zbych = &Ola;
10:     printf("Wzrost Oli: %d\n", Ola);
11:     printf("Wzrost Oli wg Zbycha: %d\n", *Zbych);
12:     printf("Adres Oli: %p\n", &Ola);
13:     printf("Adres Oli wg Zbycha: %p\n", Zbych);
14:     return 0;
15: }
```

Jak widać, odwołanie do wskaźnika `*Zbych` pozwala na wyłuskanie wartości zmiennej `Ola`, natomiast wskaźnik `Zbych` zawiera adres `Oli`.

Podsumujmy:

Odwołanie do wskaźnika `*Zbych` oraz zmienna `Ola` zawierają wartość przypisaną zmiennej `Ola`.

Wskaźnik `Zbych` oraz odwołanie do wskaźnika `&Ola` zawierają adres zmiennej `Ola`.

Wskaźniki i tablice

Wskaźniki są bardzo przydatne, gdy używa się ich w połączeniu z tablicami. Na razie nie zdajesz sobie z tego sprawy, ale z pewnością, gdy zyskasz większe doświadczenie w programowaniu, dojdziesz do wniosku, że wskaźniki są nieodłączną częścią każdego dobrego programu.

Pamiętaj, że **nazwa zadeklarowanej tablicy użyta bez nawiasów jest wskaźnikiem do tej tablicy**:

```
int tablica[20];
tablica == &tablica[0];
```

Jak widać, rzeczywiście zmienna `tablica` nie zawiera nic innego, ale adres pierwszego elementu tablicy `tablica[20]`, zgodnie z definicją, jest wskaźnikiem.

ĆWICZENIE

3.4 Napisz program, który wypisze na ekranie wartość zawartą w pierwszym elemencie tabeli (wcześniej zadeklarowanej przez Ciebie) — użyj wskaźnika.

```
1: /* Przyklad 3.4 */
2: /* Przyklad uzycia wskaznikow z wykorzystaniem */
3: /* tablic */
4: #include <stdio.h>
5: int Moja_Tablica[20];
6: int *wskaznik;
7: int main()
8: {
9:      Moja_Tablica[0] = 5;
10:     wskaznik = Moja_Tablica;
11:     printf("1. element tablicy zawiera wartosc:%d \n",
12:          *wskaznik);
13:     return 0;
14: }
```

W **wierszu 5** zostaje zadeklarowana dwudziestoelementowa tablica `Moja_Tablica[20]`. W **wierszu 6** jest deklarowany wskaźnik `wskaznik`. **Wiersze 9 – 10**: pierwszemu elementowi tablicy zostaje przypisana wartość 5, a następnie adres tego elementu jest przypisywany wskaźnikowi.

Wskaźniki są niezbędne, gdy trzeba przekazać tablicę jako argument do funkcji. Warto pamiętać, że cała tablica nie może być argumentem danej funkcji, ponieważ nie jest ona traktowana jako jedna zmienna. Więc w jaki sposób funkcja może zyskać dostęp do wszystkich elementów tablicy?

Można przekazać jako argument wskaźnik do pierwszego elementu tablicy. W takim przypadku funkcja, znając adres pierwszego elementu, będzie miała dostęp do pozostałych. A co się stanie, jeśli funkcja będzie musiała znać rozmiar danej tablicy?

Wtedy trzeba będzie wielkości danej tablicy przekazać do funkcji jako drugi argument.

ĆWICZENIE

3.5

Napisz program, który obliczy największą wartość zawartą w którymś z elementów tablicy zadeklarowanej przez Ciebie wcześniej. Skonstruuj odpowiednią funkcję.

```
1: /* Przyklad 3.5 */
2: /* Przyklad demonstruje przekazywanie tablicy */
3: /* do funkcji */
4: #include <stdio.h>
5: #define rozm 20
6: int tab[rozm];
7: int licznik;
8: int maks(int x[], int y);
9: int main()
10:{
11:       for (licznik = 0; licznik < rozm; licznik++)
12:       {
13:            printf("Wprowadz wartosc z zakresu: ");
14:            printf("-32000 - 32000 \n");
15:            scanf("%d", &tab[licznik]);
16:       }
17:       printf("Najwieksza wartosc: %d\n", maks(tab, rozm));
18:       return 0;
19:}
20:int maks( int x[], int y )
21:{
22:       int licz;
23:       int maksimum = -32000;
24:       for (licz = 0; licz < y; licz++)
25:       {
26:            if (x[licz] > maksimum)
27:            maksimum = x[licz];
28:       }
29:       return maksimum;
30:}
```

To z pewnością najtrudniejszy program, jaki dotąd zaproponowałem do napisania. Oto opis krok po kroku wszystkich czynności, które należało wykonać, aby rozwiązać to zadanie.

Wiersz 5: definiujesz stałą `rozm`, która ma przechowywać wartość równą wielkości tablicy.

Wiersz 6: definiujesz tablicę ze zmiennymi typu `int` o rozmiarze równym stałej `rozm`.

Wiersz 7: definiujesz zmienną `licznik`.

Wiersz 8: deklarujesz funkcję, która ma obliczyć największą wartość w tablicy.

Wiersze 11 – 16: wywołujesz pętlę `for`, która z kolei ma przyjmować wartości wpisane przez Ciebie z klawiatury dla wszystkich elementów tablicy `tab[]`.

Wiersz 17: wywołujesz funkcję `printf`, która wywołuje funkcję `maks()` oraz wyświetla na ekranie zwróconą przez nią wartość.

Wiersz 20: to początek definicji funkcji `maks( int x[], int y)`.

Wiersz 22: definiujesz lokalną zmienną `licznik`.

Wiersz 23: definiujesz lokalną zmienną `maksimum`, która ma tymczasowo przechowywać największą wartość w tablicy.

Wiersz 24 – 28: wywołujesz pętlę `for`, która przegląda wszystkie elementy tabeli w celu znalezienia największej wartości; mechanizm działania: przy pierwszym kroku spełniony jest warunek instrukcji `if` (ponieważ wartość pierwszego elementu funckji **musi** być większa od wartości zmiennej maksimum, która ma tymczasowo przypisaną wartość równą najmniejszej, możliwej do wpisania wartości), wartość pierwszego elementu tablicy zostaje przypisana zmiennej `maksimum`; przy następnych powtórzeniach sprawdzany jest warunek i jeśli kolejny element okazuje się większy od dotychczas przypisanej maksymalnej wartości, nowa wartość zostaje przypisana zmiennej `maksimum`; po zakończeniu pętli zmienna `maksimum` zawiera największą wartość występującą w tablicy.

Wiersz 29: instrukcja `return` zwraca wartość do programu głównego (do funkcji `printf()`).

ĆWICZENIE

3.6 Napisz program, który zsumuje wartości wszystkich elementów dwóch tablic (tablice powinny mieć taki sam rozmiar). Wykorzystaj odpowiednią funkcję.

```
1: /* Przykład 3.6 */
2: /* Przykład demonstruje przekazywanie dwoch tablic */
3: /* do funkcji */
4: #include <stdio.h>
5: #define rozmiar 20
6: int tab1[rozmiar];
7: int tab2[rozmiar];
```

```
8: int licznik;
9: int sumuj( int x[], int y[], int z );
10:int main()
11:{
12:      for (licznik = 0; licznik < rozmiar; licznik++)
13:      {
14:          printf("\nPodaj wartosc %d-ego elementu ",
15:                licznik);
16:          printf("pierwszej tablicy:  ");
17:          scanf("%d", &tab1[licznik]);
18:          printf("\nPodaj wartosc %d-ego elementu ",
19:                licznik);
20:          printf("drugiej tablicy: ");
21:          scanf("%d", &tab2[licznik]);
22:      }
23:      printf("Suma wszystkich elementow obydwu ");
24:      printf("tablic:%d\n",sumuj(tab1,tab2,rozmiar));
25:      return 0;
26:}
27:int sumuj( int x[], int y[], int z )
28:{
29:      int licz;
30:      int suma = 0;
31:      for ( licz = 0; licz < z; licz ++)
32:          suma += x[licz] + y[licz];
33:      return suma;
34:}
```

To ćwiczenie jest bardzo podobne do poprzedniego. Istotna różnica jest taka, że tym razem do funkcji zostały przekazane dwie tablice oraz wspólny rozmiar (tablice mają ten sam rozmiar, w przeciwnym razie kod zajmowałby około 40 wierszy). Jeżeli pojąłeś, na czym polegało rozwiązanie poprzedniego ćwiczenia, zrozumienie powyższego nie powinno sprawić Ci żadnych problemów.

Znaki oraz łańcuchy znaków

Dotychczas w zmiennych umieszczaliśmy jedynie liczby. Nadszedł czas, aby się dowiedzieć, jak przechowywać w nich znaki. W tym rozdziale poznasz również pojęcie łańcuchów znaków oraz nauczysz się nimi manipulować.

Znaki

Typ zmiennych, który jest używany do przechowywania pojedynczych znaków, to typ char. Zmienną taką deklaruje się w identyczny sposób jak pozostałe. Przypisywanie wartości (znaku) odbywa się natomiast w nieco odmienny sposób. Przykład:

```
Char JakisZnak;     /* deklaracja zmiennej typu char */
JakisZnak = 'a';    /* przypisanie wartosci (znaku) */
```

W tym przykładzie widać, że znak a musi być zamknięty pomiędzy apostrofami. Tak właśnie przypisuje się wartości (znaki) w języku C.

Typ char jest właściwie typem liczbowym, a w pamięci nie są przechowywane znaki, lecz odpowiadające im wartości w kodzie ASCII. Wartości te należą do zbioru liczb od 0 do 255.

Każdej liczbie odpowiada dana litera (dużym i małym literom odpowiadają odmienne liczby w kodzie ASCII).

ĆWICZENIE

3.7 Napisz program, który wyświetli na ekranie wszystkie znaki kodu ASCII.

```
1: /* Przyklad 3.7 */
2: /* Przyklad demonstruje uzycie typu char */
3: #include <stdio.h>
4: #define MAX 256
5: unsigned char licznik;
6: int main()
7: {
8:      for (licznik =0; licznik < MAX; licznik++)
9:               printf("Kod ASCII: %d /t Znak: %c\n",
10:              licznik, licznik);
11:     return 0;
12:}
```

Dawno nie było tak prostego zadania! Wyjaśnienia wymaga tylko to, jakiego typu zmiennej tu użyto. Otóż nie jest to char, ale unsigned char. Zmienne typu char mogą przyjmować wartości tylko z zakresu od –128 do 127. Typ unsigned char może natomiast przyjmować tylko wartości dodatnie z zakresu od 0 do 255, jest zatem idealny dla potrzeb tego programu. Musisz również pamiętać, że dla typów char istnieją dwa specyfikatory konwersji (z ang. *Conversion specifiers*): %d — dla liczb kodu ASCII oraz %c — dla znaków im odpowiadających.

Łańcuchy znaków

Niestety, zmienne typu `char` mogą przechowywać tylko po jednym znaku. Jak zatem możliwe jest przechowywanie w zmiennych całych zdań? Do tego celu służą właśnie łańcuchy znaków — tworzymy je, deklarując tabele typu `char`. Każdy element tabeli musi zawierać po jednym znaku danego wyrazu lub zdania.

Przykład:

Aby przechować wyraz „programowanie" jako łańcuch znaków, trzeba wykonać następujące kroki:

1. Zadeklarować tablicę składającą się z liczby elementów równej liczbie liter w wyrazie „programowanie" (13) plus 1 element przeznaczony dla znaku „końca zdania" (jest to znak `\0`):

```
char zdanie[14];
```

2. Przypisać kolejnym elementom tablicy poszczególne znaki:

```
zdanie[0] = 'p';
zdanie[1] = 'r';
zdanie[2] = 'o';
zdanie[3] = 'g';
.....
zdanie[13] = '\0'
```

Jak widać, taki sposób przypisywania znaków poszczególnym elementom tablicy nie jest zbyt użyteczny.

Łatwiej będzie przypisać wszystkie znaki przy deklaracji tablicy (łańcucha znaków):

```
char zdanie[14] = { 'p', 'r', 'o', 'g', 'r', 'a', 'm', 'o', 'w', 'a',
↪'n', 'i', 'e', '\0' };
```

Lepszym sposobem jest przypisanie tablicy całego wyrazu zawartego w cudzysłowie:

```
char zdanie[14] = "programowanie";
```

Lub jeszcze prościej:

```
char zdanie[] = "programowanie";
```

W obu przypadkach znak „końca zdania" jest automatycznie dopisywany przez kompilator; dodatkowo w drugim przypadku liczba elementów tablicy jest automatycznie dobierana tak, aby mogła przechować wystarczającą liczbę znaków.

Aby łatwo było korzystać z całych zdań przechowywanych w tablicach jako łańcuchy znaków, trzeba użyć wskaźników. Wiemy, że nazwa tablicy (bez nawiasów) jest wskaźnikiem jej początku, dlatego jeżeli znamy adres pierwszego elementu, automatycznie mamy dostęp do całego łańcucha.

Istnieje również możliwość użycia łańcuchów znaków bez konieczności korzystania z tablic. Z czego zatem należy skorzystać?

Oczywiście ze wskaźników. To znaczy, że deklaruje się wskaźnik i jednocześnie przypisuje mu ciąg znaków (zamkniętych w cudzysłowie):

```
char *zdanie = "Jakies zdanie";
```

Tak wprowadzone zdanie jest automatycznie lokowane w pewnym miejscu pamięci komputera (wraz ze znakiem `\0`) podczas kompilacji programu. Można również zadeklarować wskaźnik, któremu nie zostanie przypisany żaden łańcuch znaków, ale który w dalszej części programu wskazywałby na jakieś zdanie (np. wprowadzone z klawiatury). Warto zauważyć, że taki wskaźnik nie ma na co wskazywać, jeżeli nie przypisano mu żadnego adresu. Załóżmy, że miałby on później wskazywać na jakieś zdanie. W takim przypadku trzeba udostępnić mu odpowiednio duży blok adresów w pamięci komputera. W przeciwnym razie nie będzie miejsca dla zdania przypisywanego w późniejszej części programu.

Do tego celu posłuży funkcja `malloc()`. Działa ona w następujący sposób:

1. Funkcji `malloc()` przekazuje się liczbę potrzebnych bajtów pamięci jako argument.
2. `malloc()` znajduje i rezerwuje blok pamięci o określonym rozmiarze oraz zwraca adres pierwszego zarezerwowanego bajtu.

Przykład (zaczerpnięty z życia):

> Jest piątek. Józek i Zbych idą na imprezę. Wybierają się do Rycha, ponieważ wiedzą, że tamten pędził bimber. Przed wyjściem zastanawiają się, czy wystarczą im dwa małe plecaki. Nie wiedzą bowiem, ile bimbru Rychu wyprodukował i jakim dobrym kolegą się okaże, a więc ile butelek im odstąpi. Zbych proponuje, aby zadzwonić do Romka, który mógłby im pożyczyć samochód. Dzięki temu będą mogli przetransportować więcej alkoholu. Zbych dzwoni do Romka:

— Hej, Romek, pożyczysz nam furę? Musimy jakoś zabrać bimber od Rycha, a nie wiem, ile tego wyprodukował, plecaki mogą nam nie wystarczyć.

— Spoko, stary, powiedz tylko, ile maksymalnie może tego być.

— No, około 50 butelek.

— Super, w takim razie wystarczy wam maluszek. Pogadam ze starym, może pożyczy, moment, zapytam go... Ojciec się zgodził, pożyczę wam.

— Wielkie dzięki, Romek!

Na podstawie tego przykładu spróbuję przybliżyć działanie funkcji `malloc()`. Potraktujmy Janka i Zbycha jako jeden wskaźnik. Nie wiedzą, ile bimbru Rychu mógł wyprodukować, więc wskaźnikowi nie zostanie przypisany żaden adres. Romek jest funkcją `malloc()`. Zbych zwraca się do niego, aby udostępnił miejsce na bimber — pamięć. Romek pyta, ile to będzie butelek — ile bajtów pamięci musi udostępnić. Zbych odpowiada, że potrzeba im 50 butelek (przekazuje argument do funkcji `malloc()`). Romek pyta ojca (komputer), czy może udostępnić tyle miejsca. Ojciec się zgadza — miejsce w pamięci zostaje udostępnione.

Konstrukcja funkcji `malloc()`:

```
void *malloc( size_t rozmiar );
```

Funkcja zwraca wskaźnik do zarezerwowanego bloku pamięci. Typ zwracanej wartości to `void`, ponieważ jest on kompatybilny z wszystkimi innymi typami zmiennych. Dzięki zastosowaniu `void` można przypisywać pamięć dla dowolnych typów zmiennych. W przypadku gdy `malloc()` nie jest w stanie znaleźć żadnego wolnego bloku pamięci, wartością zwracaną jest 0.

ĆWICZENIE

3.8

Napisz program, który spróbuje udostępnić pamięć dla łańcucha o dwustu znakach. W zależności od tego, czy operacja zostanie wykonana pomyślnie, czy też nie, spraw, aby na ekranie został wyświetlony odpowiedni komunikat. Zastosuj funkcję `malloc()`.

```
1: /* Przykład 3.8 */
2: /* Przykład demonstruje użycie funkcji malloc() */
3: #include <stdlib.h>
4: #include <stdio.h>
5: int main()
```

```
6: {
7:      char *lancuch;
8:      if ((lancuch = (char *) malloc(200)) == NULL)
9:      {
10:          printf( "Za malo pamieci!!! Sorry...\n");
11:          exit(1);
12:     }
13:     printf("Operacja przydzielenia pamieci ");
14:     printf("zostala pomyslnie zakonczona!\n" );
15:     return 0;
16:}
```

Pewnie zauważyłeś już, że funkcja `malloc()` wymaga dodania do programu kolejnego pliku załącznika — *stdlib.h*. W **wierszu 7** jest deklarowany wskaźnik `lancuch`, któremu nie przydziela się adresu. W **wierszu 8** następuje wywołanie funkcji `malloc()` — poprzez przypisanie wskaźnikowi zwróconej przez nią wartości — tu także sprawdzasz, czy w ogóle udało się przydzielić wymaganą ilość pamięci (`NULL` oznacza 0, czyli fałsz w języku C). Jeżeli operacja przebiegła niepomyślnie, wykonywane są instrukcje w **wierszach 10 – 11** (instrukcja `exit(1)` powoduje nagłe zakończenie programu).

ĆWICZENIE

3.9 Napisz program, który spróbuje udostępnić pamięć tablicy złożonej z 50 zmiennych typu `int`.

```
1: /* Przyklad 3.9 */
2: /* Przyklad demonstruje uzycie funkcji malloc() */
3: /* w celu udostepnienia pamieci dla tablic */
4: #include <stdlib.h>
5: #include <stdio.h>
6: int main()
7: {
8:      int *tablica;
9:      if((tablica=(int *)malloc(50*sizeof(int)))== NULL)
10:     {
11:         printf( "Za malo pamieci... sorry \n" );
12:         exit(1);
13:     }
14:     printf("Pamiec zostala przydzielona \n");
15:     return 0;
16:}
```

W rozwiązaniu powyższego ćwiczenia wykorzystano nieznaną Ci na razie funkcję — `sizeof()`. Służy ona do obliczania liczby bajtów zajmowanych przez przekazany do niej argument. W naszym przykładzie argumentem jest typ `int` (zwykle każda zmienna typu `int` zajmuje

2 bajty, więc `malloc()` udostępni blok pamięci wielkości 50*2 = 100 bajtów). Należy również zaznaczyć, że `malloc()` zwróci wskaźnik do typu `int`.

ĆWICZENIE

3.10 Napisz program, który 10 razy wyświetli na ekranie zdanie: „Ale piękne zdanie".

```
1: /* Przyklad 3.10 */
2: /* Przyklad demonstruje uzycie funkcji puts() */
3: /* w celu wyswietlenia na ekranie lancuchow znakow*/
4: #include <stdio.h>
5: char *zdanie = "Ale piekne zdanie";
6: int licznik;
7: int main()
8: {
9:       for (licznik = 0; licznik < 10; licznik++)
10:      {
11:           puts(zdanie);
12:      }
13:      return 0;
14:}
```

Funkcja `puts()` służy do wyświetlania łańcuchów znaków na ekranie. W tym przypadku jej argumentem musi być jedynie wskaźnik do odpowiedniego łańcucha.

ĆWICZENIE

3.11 Zmodyfikuj program z poprzedniego ćwiczenia tak, aby wyświetlał to samo zdanie za pomocą funkcji `printf()`.

```
1: /* Przyklad 3.11 */
2: /* Przyklad demonstruje uzycie funkcji printf() */
3: /* w celu wyswietlenia na ekranie lancuchow znakow*/
4: #include <stdio.h>
5: char *zdanie = "Ale piekne zdanie";
6: int licznik;
7: int main()
8: {
9:       for (licznik = 0; licznik < 10; licznik++)
10:           printf( "%s\n", zdanie);
11:      return 0;
12:}
```

Aby wyświetlić łańcuch znaków na ekranie przy użyciu funkcji `printf()`, trzeba użyć odpowiedniego *specyfikatora konwersji*, którym jest `%s`.

ĆWICZENIE

3.12 Napisz program, który pobierze łańcuch znaków z klawiatury, a następnie wyświetli go 5 razy.

```
1: /* Przyklad 3.12 */
2: /* Przyklad demonstruje uzycie funkcji gets() */
3: #include <stdio.h>
4: char bufor[100];
5: int licznik;
6: int main()
7: {
8:      puts("Wpisz dowolny tekst: ");
9:      gets(bufor);
10:     for (licznik = 0; licznik < 5; licznik++)
11:         puts(bufor);
12      return 0;
13:}
```

Jak widać w powyższym przykładzie, funkcja `gets()` służy do pobierania łańcuchów znaków z klawiatury (również z innych rodzajów wejścia, ale o tym w dalszej części książki). Jako argument pobiera wskaźnik do bufora, w którym ma być umieszczony dany ciąg znaków (w tym przypadku `bufor` — czyli wskaźnik do tablicy `bufor[100]`). Po wywołaniu funkcji `gets()` wprowadzony łańcuch znaków znajdzie się w odpowiednim buforze. Aby wyświetlić go na ekranie, wystarczy wywołać funkcję `printf()` lub `puts()` z argumentem — wskaźnikiem do bufora (**wiersz 11**).

ĆWICZENIE

3.13 Zmodyfikuj poprzedni program tak, aby pobierał łańcuch znaków z klawiatury za pomocą funkcji `scanf()`.

```
1: /* Przyklad 3.13 */
2: /* Przyklad demonstruje uzycie funkcji scanf() */
3: /* w celu pobrania lancuchow znakow z klawiatury */
4: #include <stdio.h>
5: char bufor[100];
6: int licznik;
7: int main()
8: {
9:      puts("Wpisz dowolny tekst: ");
10:     scanf("%s", bufor);
11:     for (licznik = 0; licznik < 5; licznik++)
12:          printf("%s\n", bufor);
13:     return 0;
14:}
```

Funkcja `scanf()`, podobnie jak `printf()`, używa znaku `%s` do pobierania łańcuchów z klawiatury. Zauważ, że zazwyczaj przy używaniu `scanf()` musieliśmy podawać zmienną ze znakiem & (np. `scanf("%d", &liczba)`).

Zmienna poprzedzona znakiem & zwraca jej adres w pamięci komputera, a nie wartość. Wskaźnik — jak wiadomo — zawiera również adres, na który wskazuje, dlatego w powyższym przykładzie nie użyto znaku & (patrz: **wiersz 10**), tylko skorzystano z nazwy tablicy, czyli wskaźnika.

ĆWICZENIE

3.14 Napisz program wywołujący funkcję, która pobierze dwa łańcuchy znaków jako argumenty, policzy liczbę znaków w każdym z nich oraz zwróci wskaźnik do dłuższego łańcucha.

```
1: /* Przyklad 3.14 */
2: /* Przyklad demonstruje przekazywanie lanuchow */
3: /* znakow do funkcji */
4: #include <stdio.h>
5: char bufor1[100], bufor2[100];
6: char *funkcja(char x[], char y[]);
7: int main()
8: {
9:       puts( "Wpisz pierwszy tekst: " );
10:      gets(bufor1);
11:      puts( "Wpisz drugi tekst: " );
12:      gets(bufor2);
13:      printf( "Dluzszy lancuch znakow: %s\n",
14:            funkcja( bufor1, bufor2 ));
15       return 0;
16:}
17:char *funkcja( char x[], char y[] )
18:{
19:       size_t a, b;
20:       a = strlen(x);
21:       b = strlen(y);
22:       if ( a > b )
23:          return x;
24:       else
25:          return y;
26:}
```

Program pobiera dwa łańcuchy znaków za pomocą funkcji `gets()` (**wiersz 10 i 12**), a następnie wyświetla na ekranie dłuższy łańcuch. Funkcja `int *funkcja` wywoływana jest w celu porównania długości dwóch ciągów znaków. Gwiazdka umieszczona przed nazwą funkcji oznacza, że zwracany jest wskaźnik (w naszym przykładzie — wskaź-

nik do dłuższego łańcucha znaków). `strlen()` to funkcja obliczająca liczbę znaków w łańcuchu. Argumentem jest oczywiście wskaźnik do odpowiedniego bufora (**wiersz 20 i 21**).

Zastosowanie wskaźników

Trzy podstawowe zastosowania wskaźników to:

1. przekazywanie przez wskaźnik zmiennej jako argumentu funkcji,
2. dynamiczny przydział pamięci,
3. operacje arytmetyczne na wskaźnikach.

Poniżej przedstawione są przykłady oraz opis każdego z tych praktycznych zastosowań wskaźników. Zrozumienie ich działania jest niezbędne do poznania prawdziwej potęgi C. Najlepsi specjaliści opanowali te zastosowania do perfekcji i potrafią ze swobodą operować wskaźnikami, niemal jak specjalistycznymi narzędziami. Gdyby wrócić do przykładu z Rychem (wcześniej w tym rozdziale), można by to porównać do procesu produkcji bimbru. Wskaźniki są jak aparatura do destylacji — bez zrozumienia jej podstaw oraz nauczenia się operowania profesjonalnymi narzędziami nie ma szans na wytworzenie porządnego produktu, którego smak podziwialiby koledzy i koleżanki. A zatem, drogi Czytelniku, jeśli chcesz tworzyć programy, które będą budzić szacunek kolegów (szczególnie tych, którzy programują tylko w PHP), naucz się porządnie operować wskaźnikami.

Przekazywanie przez wskaźnik zmiennej jako argumentu funkcji

Są dwa sposoby przekazywania parametrów do funkcji: przez wartość oraz przez wskaźnik. W pierwszym przypadku, jak sama nazwa wskazuje, przekazuje się tylko wartość (kopię) zmiennej. Prototyp funkcji wygląda wtedy tak jak we wszystkich przykładach z poprzedniego rozdziału, np.:

```
int funkcja( int x )
```

natomiast jej wywołanie:

```
y = funkcja(x)
```

Przy przekazywaniu zmiennej x do funkcji **nie można zmienić jej wartości**. Utworzona zostaje tylko lokalna kopia, która jest używana do obliczania np. zwracanej wartości zapisywanej w zmiennej y.

Prawdziwe możliwości przy przekazywaniu zmiennej jako wskaźnika (odwołanie). Oznacza to, że funkcja adresu zmiennej przekazywana jest jako parametr. Wtedy prototyp oraz wywołanie wyglądają następująco:

```
int funkcja( int *x )

y = funkcja (&x)
```

Wszelkie operacje wewnątrz funkcji **zmieniają wartość zmiennej** x (przekazanej jako wskaźnik) w programie.

Najlepiej tę różnicę ilustruje poniższy przykład.

ĆWICZENIE

3.15

Napisz program, który zdefiniuje dwie funkcje ilustrujące przekazywanie zmiennej przez wartość oraz wskaźnik. Wypisz wartość zmiennej po wywołaniu każdej z funkcji.

```
1: /* Przyklad 3.15 */
2: /* Przyklad demonstruje przekazywanie zmiennej do funkcji */
3: /* przez wartosc oraz wskaznik */
4: #include <stdio.h>
5: int wartosc( int x );
6: void wskaznik (int *x);
7: int main()
8: {
9:      int x = 1;
10:     int y;
11:     y = wartosc(x);
12:     printf("Wartosc x po wywolaniu funkcji wartosc: %d\n", x);
13:     wskaznik(&x);
14:     printf("Wartosc x po wywolaniu funkcji wskaznik: %d\n", x);
15:     return 0;
16: }
15: int wartosc( int x )
16: {
17:      x++;
18:      return x;
19: }
```

```
20: void wskaznik (int *x)
21: {
22:       (*x)++;
23: }
```

Wiersz 5 – 6: definiujesz funkcje `wartosc` oraz `wskaznik`. Zauważ, że funkcja `wskaznik` jako argument przyjmuje wskaźnik do zmiennej.

Wiersz 11 – 12: wywołujesz funkcję `wartosc`. Wypisujesz jej wartość po wywołaniu i okazuje się, że pomimo zwiększenia wartości `x` wewnątrz funkcji nie zmieniła się ona w funkcji `main` — `printf` wypisuje wciąż wartość 1.

Wiersz 13 – 14: wywołujesz funkcję `wskaznik`. Konieczne jest przekazanie jako argumentu adresu zmiennej `x`. Do jego wydobycia służy operator &. Jak widać, `wskaznik` zmienia wartość zmiennej `x` w programie — `printf` wypisuje wartość 2.

Wiersz 20 – 22: wewnątrz funkcji `wskaznik` zwiększasz wartość zmiennej `x` o jeden. Nie zapomnij o operatorze * i nawiasach! Bez nawiasów zwiększony zostanie adres, a nie wartość pod tym adresem. Pamiętaj, że `x` jest wewnątrz funkcji `wskaznik` wskaźnikiem! Przy wywołaniu takiej funkcji następuje niewidzialna operacja przypisania adresu zmiennej (przekazywanego jako argument) do lokalnego wskaźnika wewnątrz funkcji:

```
*x_wewnatrz_funkcji  = &x_z_programu
```

Tak, faktycznie, ćwiczenie 3.5 ilustrowało przekazywanie zmiennej przez wskaźnik. Wtedy jako parametr przekazywany był wskaźnik do początku tablicy, czyli `int x[]`, oznaczający to samo co `int *x`. Prototyp funkcji w postaci `int maks( int x[], int y )` można również zapisać jako `int maks ( int *x, int y)`.

Tablice zawsze są przekazywane przez wskaźnik! Praktycznie nie da się tego zrobić inaczej, ponieważ należałoby przekazać każdy z jej elementów jako parametr.

Dynamiczny przydział pamięci

Dynamiczny przydział pamięci jest najważniejszym zastosowaniem wskaźników. Zostało to zademonstrowane w ćwiczeniu 3.8 — przy użyciu funkcji `malloc`. Każdą przydzieloną pamięć należy zwolnić za pomocą metody `free`. Jak sama nazwa wskazuje, przydział jest

dynamiczny, ponieważ w dowolnym momencie można zwalniać i udostępniać, nawet w czasie działania programu, wybraną ilość pamięci. W przypadku tablic trzeba określić ich wielkość przy deklaracji. Taki przydział jest statyczny, ponieważ rozmiar tablicy należy określić jeszcze przed kompilacją. W tej sytuacji możesz zostać władcą pamięci i robić z nią, co chcesz, kiedy chcesz. Pamiętaj jednak, żeby każdą pamięć, której zawartości nie będziesz dalej używać, zwolnić przy użyciu funkcji free!

ĆWICZENIE

3.16 Napisz program, który poprosi o podanie ilości bimbru dostępnej do sprzedaży (w litrach) oraz na podstawie tej informacji udostępni odpowiednią ilość pamięci dla tablicy dynamicznej.

```
1: /* Przyklad 3.16 */
2: /* Przyklad demonstruje dynamiczny przydzial pamieci */
3: /* z uzyciem funkcji malloc() */
4: #include <stdio.h>
5: int main()
6: {
7:     int *kontener;
8:     int n;
9:     printf("Podaj mozliwa ilosc bimbru do sprzedazy w litrach: ");
10:    scanf("%d", &n);
11:    if (kontener = (int *)malloc(sizeof(int)*n))
12:        printf("Przygotowalem kontener o wielkosci %d\n", n);
13:    else
14:        printf("Nie mam takiego kontenera, nie bedzie dzisiaj
               imprezy, sorry!");
15:    free(kontener);
16:    return 0;
17:}
```

Operacje arytmetyczne na wskaźnikach

Operacje arytmetyczne na wskaźnikach, oferowane przez język C, pozwalają wprawnym programistom dokonywać cudów na pamięci. Sztuka ta nie jest prosta i wymaga dużo praktyki, by można było wykorzystać jej prawdziwą moc. Warto jednak zainwestować swój czas w ćwiczenia i zostać prawdziwym guru C.

Dostępne operacje arytmetyczne to: przypisanie wskaźnika do wskaźnika, operatory porównania wskaźników oraz dodawanie liczby do wskaźnika i odejmowanie liczby od wskaźnika.

Wskaźnik można przypisać do innego wskaźnika w poniższy sposób:

```
int x = 0;
int *ptr1 = &x;
int *ptr2 = ptr1;
```

Rezultatem jest po prostu przypisanie adresu, na który wskazuje `ptr1`, wskaźnikowi `ptr2`. W praktyce jest to bardzo użyteczne, szczególnie gdy przeglądając zawartość pewnej struktury danych (np. tablicy/łańcucha znaków), chcemy zachować wskaźnik do interesującego nas miejsca w pamięci — np. w przypadku zdania przechowywanego jako łańcuch znaków chcemy przechować wskaźniki do miejsc, od których zaczynają się poszczególne słowa w tym zdaniu.

Porównanie wskaźników jest głównie wykorzystywane w celu sprawdzenia, który wskaźnik wskazuje na dalszy element w pamięci. Dobrym przykładem jest implementacja odwrócenia łańcucha znaków za pomocą wskaźników (ćwiczenie 3.17).

Najczęstszym i najważniejszym zastosowaniem operacji arytmetycznych jest przeglądanie (iterowanie) struktur danych — w szczególności tablic/łańcuchów znaków. Wykorzystuje się do tego celu głównie operator inkrementacji `++`. Do wskaźników można też dodawać dowolną liczbę całkowitą, a także odejmować od nich dowolną liczbę całkowitą. Dla przykładu zdefiniujemy wskaźnik, do którego przypiszemy adres początku tablicy.

`int *wskaznik = tablica; //` — trzeba pamiętać, że zapis `tablica` jest równoważny z `&tablica[0]`, czyli oznacza adres jej początku.

W efekcie wykonania operacji `wskaznik + 5` można uzyskać dostęp do 5. elementu tablicy. Dlaczego nie oznacza to dodania liczby 5 do adresu, na który wskazuje wskaźnik? Ponieważ kompilator C jest na tyle sprytny, że automatycznie oblicza różnicę w odległości między elementami danego typu (w tym przypadku typu `int`) i tak naprawdę wykonuje operację `wskaznik + 5 * (sizeof(int))`, ale my jej po prostu nie widzimy. Warto zauważyć, że operacja `wskaznik + 5` jest też równoznaczna z indeksowaniem tablicy — `tablica[5]`. Tak więc `tablica[5] == wskaznik + 5 == tablica + 5`. Po co zatem używać do tego celu wskaźników? W przedstawionym przykładzie faktycznie nie ma różnicy

między operacją indeksowania tablicy a dodawaniem odpowiedniej liczby do wskaźnika. Rozważmy jednak przykład przeglądania tablicy w pętli:

```
int i = 0;
while (i < rozmiar_tablicy)
{
*wskaźnik++ = 1; //Uwaga! operacja rownowazna z *wskaznik = 1; wskaznik++;
  i++;
}
```

oraz:

```
int i = 0;
while (i < rozmiar_tablicy)
{
  tablica[i] = 1;
  i++;
}
```

Na pierwszy rzut oka nie widać żadnej różnicy. A jednak wersja ze wskaźnikami jest szybsza. Warto zauważyć, że w każdej iteracji pętli w przykładzie z indeksowaniem tablicy wykonywane jest mnożenie i dodawanie — zapis `tablica[i]` jest równoznaczny z `tablica + i*sizeof(int)`. W przypadku zwiększania wskaźnika o jeden element wykonywana jest operacja `wskaznik++`, czyli tylko `wskaznik = wskaznik + sizeof(int)`. Nie występuje więc tutaj żadna operacja mnożenia, a mnożenie jest dużo bardziej kosztowne (czasochłonne) niż dodawanie. Przykład ze wskaźnikami można jeszcze ulepszyć, eliminując konieczność definiowania indeksu (`int i`):

```
int *ptr = tablica;
do
  *wskaznik = 1;
while
  (wskaznik++ <= tablica + rozmiar_tablicy)
```

Jest to możliwe, jeśli znamy rozmiar tablicy oraz mamy zachowany wskaźnik do jej początku (`tablica`). Tak więc wykorzystanie operacji arytmetycznych na wskaźnikach pozwala na zwiększenie szybkości wykonywania programu oraz zmniejszenie wymaganej pamięci (poprzez eliminację konieczności wykorzystywania w programie zmiennych — najczęściej typu `int` — służących do iteracji/indeksowania). Często też kod wygląda przejrzyściej — przynajmniej dla bardziej zaawansowanych programistów C. Ciekawostką jest też to, że poprzez operacje arytmetyczne na wskaźnikach można dostać się w obszary niedostępne dla zwykłych śmiertelników. Definiując następującą tablicę:

```
int tablica[20];
```

nie można odczytać wartości `tablica[40]` — ponieważ rezultatem będzie błąd w dostępie do pamięci oraz przerwanie programu. Natomiast jeśli odwołać się do tego miejsca pamięci poprzez `*(tablica+40)`, to będzie można odczytać (lub nawet zmienić) znajdujące się tam informacje (prawdopodobnie widoczne jako tzw. śmieci; choć może to być np. adres powrotu funkcji, który haker potrafi nadpisać tak, żeby wywołać jakąś jego magiczną funkcję). Myślę, że teraz jest już jasne, jaką potęgę dają język C i wskaźniki. Czas zatem przejść do kilku ciekawych ćwiczeń utrwalających zdobytą właśnie wiedzę.

ĆWICZENIE

3.17 Napisz program odwracający łańcuch znaków w miejscu, tzn. bez deklarowania drugiego łańcucha. Zastosuj operacje arytmetyczne na wskaźnikach.

```
1: /* Przyklad 3.17
2:    Przyklad demonstruje operacje arytmetyczne na wskaznikach
3:    poprzez implementacje odwrocenia lancucha znakow */
4: #include <stdio.h>
5: #include <string.h>
6: void odwrocenie (char *s);
7: int main()
8: {
9:      char ciag_znakow[20] = "Rychu ma bimber";
10:     odwrocenie(ciag_znakow);
11:     printf("Ciag znakow po odwroceniu: %s\n", ciag_znakow);
12:     return 0;
13:}
14:void odwrocenie(char *s)
15:{
16:     char *wsk = s + strlen(s)-1;
17:     char temp;
18:     for (;wsk>s;wsk--,s++)
19:     {
20:         temp = *s;
21:         *s = *wsk;
22:         *wsk = temp;
23:     }
24:}
```

Wiersz 5: niezbędne jest załączenie pliku nagłówkowego `<string.h>` w celu wykorzystania funkcji `strlen` wewnątrz funkcji `odwrocenie`.

Wiersz 10: wywołanie funkcji `odwrocenie(char *s)` — funkcja pobiera jako argument wskaźnik do tablicy znaków, więc przekazujesz tylko adres do początku tablicy.

Wiersz 16: definiujesz lokalny wskaźnik *wsk, który ma być nakierowany na koniec łańcucha znaków (nie na koniec tablicy). Do tego celu stosuje się funkcję strlen, która zwraca długość łańcucha. Wskaźnik s jest już nakierowany na pierwszy element, więc trzeba odjąć 1 od długości łańcucha, żeby nakierować wskaźnik na ostatni element, a nie na kończące łańcuch NULL ('\0'). Trzeba pamiętać, że każdy łańcuch znaków w C musi być zakończony znakiem NULL - '\0'.

Wiersz 17: definiujesz tymczasową zmienną typu char w celu przechowywania odpowiednich znaków do skopiowania.

Wiersz 18 – 23: jest to główna pętla programu — w każdym kroku zamieniasz ze sobą znaki z dwóch przeciwległych końców łańcucha, po których przechodzą odpowiednie wskaźniki — wskaźnik s przesuwa się od początku, a wskaźnik wsk od końca. W momencie kiedy wskaźnik wsk będzie mniejszy niż wskaźnik s (czyli będzie wskazywał na element umieszczony wcześniej w tablicy), pętla kończy działanie, a w tablicy ciag_znakow[20] zamiast „Rychu ma bimber" pojawi się łańcuch „rebmib am uhcyR". Zauważ też, że pętla for nie zawiera w ogóle początkowego przypisania (ponieważ niepotrzebna Ci jest zmienna indeksująca) oraz zawiera inkrementacje dwóch wskaźników naraz (wsk-- oraz s++).

ĆWICZENIE

3.18

Napisz program odwracający łańcuch znaków w miejscu, tzn. bez deklarowania drugiego łańcucha. Zastosuj indeksowanie tablic.

```
1: /* Przyklad 3.18
2:    Przyklad demonstruje uzycie tablicy znakow w celu
3:    implementacji odwrocenia lancucha znakow */
4: #include <stdio.h>
5: #include <string.h>
6: void odwrocenie_tab (char *s);
7: int main()
8: {
9:      char ciag_znakow[20] = "Rychu ma bimber";
10:     odwrócenie_tab(ciag_znakow);
11:     printf("Ciag znakow po odwroceniu: %s\n", ciag_znakow);
12:     return 0;
13:}
14:void odwrocenie_tab(char *s)
15:{
16:     int i,j;
17:     char temp;
```

```
18:     j = strlen(s)-1;
19:     for (i=0; i<j; j--, i++)
20:     {
21:         temp = s[i];
22:         s[i] = s[j];
23:         s[j] = temp;
24:     }
25:}
```

Tu kod jest bardzo podobny do tego z poprzedniego ćwiczenia ze wskaźnikami. Jedyne zmiany wprowadzone zostały do funkcji odwracającej łańcuch znaków w miejscu.

Wiersz 16: niezbędna jest deklaracja dwóch zmiennych `int` służących do iteracji w pętli `for` — w przykładzie z operacjami na wskaźnikach wystarczyła tylko jedna zmienna — wskaźnik do końca łańcucha.

Wiersz 18: podobnie jak w poprzednim przykładzie, przypisujesz zmiennej `j` indeks tablicy odpowiadający ostatniemu znakowi. Również wykorzystujesz w tym celu funkcję `strlen`, której prototyp znajduje się w pliku nagłówkowym *string.h*.

Wiersz 19 – 24: główna pętla programu — podstawowe różnice to zamiana wskaźników na dostęp do tablicy poprzez indeksowanie oraz posłużenie się zmiennymi indeksowymi (w tym przypadku konieczna jest inicjalizacja zmiennej `i`).

ĆWICZENIE

3.19 Napisz program obliczający długość łańcucha znaków wpisanego z klawiatury. Zdefiniuj własną funkcję wykonującą taką operację.

```
1: /* Przyklad 3.19
2:    Przyklad zawiera implementacje funkcji strlen obliczajacej dlu
3:    gosc lancucha znakow */
4: #include <stdio.h>
5: int moj_strlen(char *s);
5: int main()
6: {
7:     char str[50];
8:     printf("Podaj lancuch znakow: ");
9:     scanf("%s", str);
10:    printf("\nDlugosc lancucha znakow: %d", moj_strlen(str));
11:    return 0;
12:}
13:int moj_strlen(char *s)
14:{
```

```
15:    int i = 0;
16:    while (*s !='\0')
17:    {
18:        s++;
19:        i++;
20:    }
21:    return i;
22:}
```

Program definiuje operację zliczania znaków w łańcuchu z użyciem operacji na wskaźniku.

Wiersz 16-20: w każdym kroku pętli `while` wskaźnik `*s` jest przesuwany do następnej pozycji w łańcuchu. Pętla wykonywana jest do momentu, w którym wskaźnik `*s` wskaże na koniec łańcucha znaków.

Struktury w języku C

Struktury, podobnie jak tablice, są grupą zmiennych zebranych razem pod wspólną nazwą. Ponadto w strukturach można przechowywać zmienne różnego typu, a nawet całe tablice oraz inne struktury. Struktury **definiujemy** w następujący sposób:

```
struct wiek_kotka {
    int bury_kotek;
    int krasy_kotek;
};
```

Definicję rozpoczyna się od słowa kluczowego `struct`, po którym występuje nazwa struktury. Następnie w obrębie klamer wymieniane są zmienne, które mają wchodzić w skład danej struktury. Zmienne są nazywane *składowymi struktury*. Struktura musi być również zadeklarowana — należy utworzyć jej tzw. *instancje*.

Przykład:

```
struct wiek_kotka {
    int bury_kotek;
    int krasy_kotek;
} koci_wiek, ludzki_wiek;
```

Powyższe instrukcje definiują strukturę `wiek_kotka` oraz tworzą jej dwie instancje: `koci_wiek` i `ludzki_wiek`. Zarówno pierwsza, jak i druga instancja zawierają po dwie zmienne typu `int`.

Istnieje jeszcze drugi sposób deklarowania struktur:

```
struct wiek_kotka {        /* definicja */
    int bury_kotek;        /* definicja */
    int krasy_kotek;       /* definicja */
};
struct wiek_kotka koci_wiek,ludzki_wiek; /*deklaracja*/
```

W tym przypadku deklaracja jest oddzielona od definicji i może być umieszczona w dowolnej części programu. Pamiętaj jednak — dopóki nie zostaną zadeklarowane instancje danej struktury, dopóty nie można ich używać (miejsce w pamięci komputera zostaje przydzielone dopiero po zadeklarowaniu instancji struktury).

Przypisywanie wartości poszczególnym elementom struktury odbywa się w następujący sposób:

```
koci_wiek.bury_kotek = 40;
koci_wiek.krasy_kotek = 30;
ludzki_wiek.bury_kotek = 4;
ludzki_wiek.krasy_kotek = 3;
```

Jak widać, odwołujemy się do poszczególnych elementów struktury za pomocą kropki oddzielającej nazwę przykładu od nazwy członu. Można również jednocześnie zdefiniować i zadeklarować strukturę oraz przypisać wartości jej elementom:

```
struct wiek_kotka {
    int bury_kotek;
    int krasy_kotek;
} koci_wiek = { 4, 3 };
```

ĆWICZENIE

3.20 Napisz program, który wykorzystując strukturę *wiek_kota*, przypisuje poszczególnym elementom zarówno wartości informujące o kocim wieku, jak i o rzeczywistym, ludzkim wieku. Zadeklaruj dwie instancje struktury i niech każdy z nich zawiera po trzy elementy (np. `bury_kot`, `krasy_kot`, `kot_odmieniec`).

```
1: /* Przyklad 3.20 */
2: /* Przyklad demonstruje uzycie struktur */
3: #include <stdio.h>
4: struct wiek_kota {
5:     int bury_kot;
6:     int krasy_kot;
7:     int kot_odmieniec;
8: } koci_wiek, ludzki_wiek;
```

```
9: int main()
10:{
11:       koci_wiek.bury_kot = 3;
12:       koci_wiek.krasy_kot = 4;
13:       koci_wiek.kot_odmieniec = 5;
14:       ludzki_wiek.bury_kot = 30;
15:       ludzki_wiek.krasy_kot = 40;
16:       ludzki_wiek.kot_odmieniec = 50;
17:       printf("Bury kot ma %d lat, ",
18:             ludzki_wiek.bury_kot);
19:       printf("ale po kociemu ma %d lat \n",
20:             koci_wiek.bury_kot);
21:       printf("Krasy kot ma %d lat, ",
22:             ludzki_wiek.krasy_kot);
23:       printf("ale po kociemu ma %d lat \n",
24:           koci_wiek.krasy_kot);
25:       printf("Kot odmieniec ma %d lat, ",
26:           ludzki_wiek.kot_odmieniec);
27:       printf("ale po kociemu ma %d lat. \n",
28:             koci_wiek.kot_odmieniec);
29:       return 0;
30:}
```

W **wierszach 4 – 8** definiowana jest struktura `wiek_kota` złożona z trzech elementów oraz deklarowane są jej dwie instancje — `koci_wiek` i `ludzki_wiek`. W **wierszach 11 – 16** poszczególnym członom zostają przypisane wartości, a w pozostałej części programu następuje wyświetlenie wartości wszystkich elementów na ekranie.

ĆWICZENIE

3.21 Napisz program, który wyświetli na ekranie dzisiejszą datę. Wykorzystaj struktury.

```
1: /* Przyklad 3.21 */
2: /* Przyklad demonstruje uzycie struktur, */
3: /* ktorych elementami sa tablice */
4: #include <stdio.h>
5: #include <string.h>
5: struct data {
6:      char dzien[20];
7:      char miesiac[20];
8:      int rok;
9: } d_data;
10:int main()
11:{
12:       strcpy(d_data.dzien, "poniedzialek");
13:       strcpy(d_data.miesiac, "styczen");
14:       d_data.rok = 2001;
15:       printf("%s, %s, %d", d_data.dzien, d_data.miesiac,
```

```
16:              d_data.rok);
17:       return 0;
18:}
```

W powyższym przykładzie wykorzystane zostały tablice (łańcuchy znaków) jako elementy struktury. Jak widać, nie jest to skomplikowane. Trzeba pamiętać, że nie można po prostu przypisać łańcucha znaków do tablicy (np. `dzien[20] = „wtorek"`). Jest to możliwe tylko przy definicji tablicy (np. `char dzien[20] = „poniedzialek"`). W tym przypadku należy skopiować znak po znaku lub użyć gotowej funkcji `strcpy()` z biblioteki `<string.h>`. Pierwszym argumentem takiej funkcji jest wskaźnik do łańcucha, do którego chcemy skopiować łańcuch znaków przekazany jako drugi argument.

ĆWICZENIE

3.22 Napisz program, który poprosi Cię o podanie swoich danych (`imie`, `nazwisko`, `adres`, `data_urodzenia`), a następnie wyświetli je na ekranie. Wykorzystaj struktury.

```
1: /* Przyklad 3.22 */
2: /* Przyklad demonstruje uzycie struktur, */
3: /* ktorych elementami sa inne struktury */
4: #include <stdio.h>
5: struct adres {
6:     char ulica[30];
7:     int nr_domu;
8:     char miasto[40];
9: };
10: struct data {
11:     int rok;
12:     char miesiac[20];
13:     char dzien[20];
14: };
15: struct dane {
16:     char imie[20];
17:     char nazwisko[30];
18:     struct adres moj_adr;
19:     struct data u_data;
20: } MojeDane;
21: int main()
22: {
23:      printf("\nPodaj imie: ");
24:      scanf("%s", MojeDane.imie);
25:      printf("\n\nPodaj nazwisko: ");
26:      scanf("%s", MojeDane.nazwisko);
27:      printf("\n\nPodaj adres: ");
28:      printf("\n\tUlica: ");
29:      scanf("%s", MojeDane.moj_adr.ulica);
```

```
30:     printf("\n\tNumer domu: ");
31:     scanf("%d", &MojeDane.moj_adr.nr_domu);
32:     printf("\n\tMiasto: ");
33:     scanf("%s", MojeDane.moj_adr.miasto);
34:     printf("\n\nPodaj date urodzenia: ");
35:     printf("\n\tRok (np.1990): ");
36:     scanf("%d", &MojeDane.u_data.rok);
37:     printf("\n\tMiesiac (slownie): ");
38:     scanf("%s", MojeDane.u_data.miesiac);
39:     printf("\n\tDzien (slownie): ");
40:     scanf("%s", MojeDane.u_data.dzien);
41:     printf("\n\n\nTwoje Dane: \n");
42:     printf("\n%s\n%s\n%s\n%d\n%s\n%d\n%s\n%s",
43:           MojeDane.imie, MojeDane.nazwisko,
44:           MojeDane.moj_adr.ulica,
45:           MojeDane.moj_adr.nr_domu,
46:           MojeDane.moj_adr.miasto,
47:           MojeDane.u_data.rok,
48:           MojeDane.u_data.miesiac,
49:           MojeDane.u_data.dzien);
50:     return 0;
51: }
```

W powyższym przykładzie elementami struktury są inne struktury. Struktury takie deklaruje się wewnątrz definicji głównej struktury (**wiersze: 18, 19**). Należy pamiętać, że muszą one być wcześniej zdefiniowane (**wiersze: 5 – 9, 10 – 14**). Do takich elementarnych struktur odwołujemy się przy użyciu jeszcze jednej kropki (np. **wiersze: 29, 31, 33**). Zwróć uwagę na funkcję `scanf()` — jeśli „zmienna wejściowa" jest typu „łańcuchowego", nie pisze się znaku `&` (nazwa tablicy jest wskaźnikiem; patrz np.: **wiersze: 24, 27, 29**); należy natomiast umieścić znak `&` przed zmiennymi numerycznymi (`scanf()` musi jakoś uzyskać ich adres; patrz np.: **wiersze 31, 36**).

Co powinieneś zapamiętać z tego cyklu ćwiczeń?

- Co to są wskaźniki, jak je deklarować?
- Jak uzyskać adres zmiennej?
- Jakie wartości mogą przechowywać wskaźniki?

- Jak przypisywać wartości wskaźnikom?
- Jakimi sposobami można przekazywać tablice do funkcji?
- Ile bajtów zajmują zmienne typu `char`?
- Jakie wartości mogą przyjmować zmienne typu `char`?
- Co to są znaki ASCII?
- Co to jest łańcuch znaków, jakie są sposoby jego deklaracji?
- Do czego służy funkcja `malloc()`, jaka jest jej konstrukcja?
- Do czego służy funkcja `gets()`, jaka jest jej konstrukcja?
- Do czego służy funkcja `puts()`, jaka jest jej konstrukcja?
- Jakie są podstawowe zastosowania wskaźników?
- Czym się różni przekazywanie parametrów do funkcji przez wskaźnik od przekazywania przez wartość?
- Jak przebiega niewidzialna operacja przypisania wskaźnika przekazywanego jako parametr do funkcji? (ćwiczenie 3.15)
- Czy tablice są przekazywane do funkcji przez wartość, czy przez wskaźnik?
- Po co przydzielać dynamicznie pamięć?
- W jaki sposób przydzielać dynamicznie pamięć?
- W jaki sposób należy pamięć zwolnić?
- Dlaczego należy zwalniać pamięć?
- Jakiego typu operacje na wskaźnikach są Ci znane?
- Do czego służy dodawanie liczby do wskaźnika i odejmowanie jej od wskaźnika?
- Jaka jest różnica między przeglądaniem tablicy za pomocą operacji inkrementacji na wskaźniku (`tablica++`) a indeksowaniem tablicy (`tablica[i]`)?
- W jaki sposób można przekazać do programu argumenty z linii poleceń?
- Co to są struktury, jak je definiować, a jak deklarować?
- Jakiego typu zmienne mogą być przechowywane w strukturach?
- Jakie dane, oprócz zmiennych, można przechowywać w strukturach?
- Jak odwoływać się do poszczególnych składowych struktury?
- Jak tworzyć struktury, których elementami są inne struktury?
- Jak odwoływać się do struktur będących elementami struktury?

Ćwiczenia do samodzielnego wykonania

ĆWICZENIE 1.

Napisz program, który wyzeruje dowolną tablicę wielowymiarową.

ĆWICZENIE 2.

Napisz program, który zsumuje wszystkie elementy dwóch dowolnych tablic i umieści wynik w dowolnej zmiennej, której wartość zostanie wyświetlona na ekranie.

Użyj funkcji, która pobiera dwie tablice jako argumenty i zwraca wartość równą sumie wszystkich elementów danych tablic.

ĆWICZENIE 3.

Napisz program, który przydzieli pamięć dla tablicy złożonej z 20 elementów typu `float`.

Użyj funkcji `malloc()` oraz `sizeof()`.

ĆWICZENIE 4.

Napisz program, który wyświetli na ekranie jakikolwiek łańcuch znaków.

Użyj dwóch sposobów!

ĆWICZENIE 5.

Spróbuj napisać program, który posłuży Ci jako mała książka adresowa.

Skorzystaj ze struktur.

ĆWICZENIE 6.

Napisz program obliczający długość łańcucha znaków podanego jako argument programu.

Pamiętaj o odpowiednich parametrach funkcji `main`.

Operacje wejścia-wyjścia

Drogi Czytelniku, w tym rozdziale poznasz narzędzia niezbędne każdemu programiście — czyli operacje wejścia-wyjścia. Choć lektura tego typu rozdziałów bywa nużąca (postaram się rzecz jasna przekazać wszystko w jak najprostszy i najbardziej zwięzły sposób), nie pomijaj tej części książki! Bez podstawowej znajomości operacji wejścia-wyjścia dużo nie zdziałasz. I nawet jeśli przejdziesz dalej, sądzę, że wrócisz do tego rozdziału. Co więcej, myślę, że kartki, na których został on zapisany, najszybciej zaczną wypadać z książki — szczególnie należącej do ambitnego Czytelnika. Dzięki lekturze tego rozdziału poznasz takie pojęcia jak programowanie strumieni wejścia-wyjścia i operacje na plikach, czyli wszystko to, co dotyczy przekazywania danych z i do programu.

Strumienie wejścia-wyjścia

Każdy program w C musi wykonywać operacje na strumieniach wejścia-wyjścia. Była już mowa o podstawowych funkcjach wejścia-wyjścia: `printf()`, `puts()` — funkcjach wyjścia — oraz `scanf()` i `gets()` — funkcjach wejścia. Funkcjami wyjścia nazywamy funkcje, które wysyłają dane poza program (np. na ekran, do drukarki lub do plików). Natomiast funkcjami wejścia nazywamy funkcje, które pobierają dane z zewnątrz (np. z klawiatury czy z pliku) do programu.

Co to jest strumień? Strumień jest ciągiem bajtów danych przesyłanych do programu (strumień wejścia) lub wysyłanych na zewnątrz programu (strumień wyjścia). Rozróżniamy dwa rodzaje strumieni: tekstowe oraz binarne.

Strumienie tekstowe są pogrupowane w wiersze tekstu (do 255 znaków w jednym wierszu) i zakończone znakiem `/0` (znak *końca linii*).

Strumienie binarne to ciąg danych reprezentowanych przez zera i jedynki. Tego typu używa się podczas operacji na plikach.

W języku C istnieje pięć standardowych (zdefiniowanych) strumieni wejścia-wyjścia: `stdin`, `stdout`, `stderr`, `stdprn` i `stdaux`.

`Stdin` — strumień standardowego wejścia. Korzysta z danych tekstowych wpisanych z klawiatury.

`Stdout` — strumień standardowego wyjścia. Służy do przesyłania danych tekstowych na ekran.

`Stderr` — strumień „standardowego błędu". Służy do wyświetlania błędów na ekranie komputera.

`Stdprn` — strumień drukarki. Służy do wysyłania danych tekstowych do portu drukarki.

`Stdaux` — strumień pomocniczy. Służy do przesyłania danych do portu COM1.

W języku C istnieje wiele funkcji operujących na strumieniach wejścia-wyjścia. W tym rozdziale opiszę kilka z nich oraz zaprezentuję po jednym przykładzie dla każdej funkcji.

Funkcje wejścia

W tym rozdziale omówione zostaną tylko te funkcje, które pobierają dane ze standardowego wejścia (`stdin`). Funkcje te dzielą się na:

- pobierające pojedyncze znaki z klawiatury, np. `getchar();`
- pobierające z klawiatury całe wiersze, np. `gets();`
- pozwalające na manipulację strumieniami wejścia — `scanf()`.

Funkcja `getchar()` pobiera pojedyncze znaki ze standardowego wejścia (z klawiatury). Prototyp funkcji `getchar()` wygląda tak:

```
int getchar(void);
```

ĆWICZENIE

4.1 Napisz program, który pobierze jeden znak z klawiatury, a następnie wyświetli go na ekranie. Użyj funkcji `getchar()`.

```
1: /* Przyklad 4.1 */
2: /* Przyklad demonstruje uzycie funkcji getchar() */
3: #include <stdio.h>
4: int znak;
5: int main()
6: {
7:       znak = getchar();
8:       printf("Wpisales znak: ");
9:       putchar(znak);
10:      printf("\n");
11:      return 0;
12: }
```

Zauważ, że funkcja `getchar()` zwraca wartość typu `int`, a więc tylko odpowiednik znaku w kodzie ASCII. Nasza zmienna — znak — musi być zatem typu `int`. Do wyświetlenia na ekranie pobranego wcześniej znaku używa się funkcji wyjścia `putchar()` (jest ona opisana w dalszej części tego rozdziału, patrz: „Funkcje wyjścia").

Przykładem funkcji pobierającej ze standardowego wejścia całe wiersze tekstu jest funkcja `gets()`. Zapoznałeś się z jej użyciem przy okazji omawiania łańcuchów znaków.

Prototyp funkcji `gets()`:

```
char *gets(char *str);
```

Funkcja jako argument pobiera wskaźnik do odpowiedniego łańcucha znaków i również zwraca wskaźnik do innego łańcucha (do typu `char`).

Funkcją pozwalającą na manipulowanie strumieniami wejścia jest dobrze Ci znana funkcja `scanf()`. Mówimy, że funkcja manipuluje strumieniami, ponieważ może interpretować pobrane wartości z klawiatury oraz przypisywać je odpowiednim typom zmiennych. Służy do tego tzw. **łańcuch formatowania** (z ang. *format string*).

```
scanf("%d", &zmienna);
```

Pierwszy argument funkcji `scanf()`, czyli znak `%d` w powyższym przykładzie, jest nazywany łańcuchem formatowania. Przypominasz sobie zapewne, że znak `&` przed zmienną jest tzw. operatorem adresu i umieszczony przed zmienną powoduje otrzymanie jej adresu. Jak

widać, łańcuch formatowania składa się ze specyfikatorów konwersji. Każdemu z nich musi odpowiadać adres zmiennej. W skład każdego specyfikatora konwersji wchodzą następujące znaki:

- znak `%`;
- tzw. specyfikator typu (np. znak `d` w `%d`), który informuje funkcję `scanf()` o typie zmiennej, jakiej ma być przypisana wartość pobrana ze standardowego wejścia (więcej specyfikatorów typu opisano w tabeli 4.1);
- opcjonalnie liczba, która określa szerokość pola, czyli liczbę znaków, jaką `scanf()` ma pobrać ze standardowego wejścia;
- opcjonalnie tzw. modyfikator precyzji (z ang. *precision modificator*), który modyfikuje znaczenie specyfikatora typu (listę modyfikatorów precyzji przedstawiono w tabeli 4.2).

Tabela 4.1. *Specyfikatory typu*

Typ	Argument	Znaczenie specyfikatora typu
d	`int *`	Wartość typu `integer` w postaci dziesiętnej
i	`int *`	Wartość typu `integer` w postaci dziesiętnej, oktalnej lub szesnastkowej
o	`int *`	Wartość typu `integer` w postaci oktalnej
u	`unsigned int *`	Dodatnia wartość typu `integer` w postaci dziesiętnej
x	`int *`	Wartość typu `integer` w postaci heksadecymalnej
c	`char *`	Pojedyncze znaki lub więcej znaków (jeżeli podana jest szerokość pola)
s	`char *`	Łańcuchy znaków
e,f,g	`float *`	Liczba przecinkowa
[...]	`char *`	Łańcuch znaków (ze standardowego wejścia przyjmowane są tylko znaki podane w nawiasach)
[^...]	`char *`	Łańcuch znaków (przyjmowane są wszystkie znaki oprócz tych, które zostały podane w nawiasach)

Tabela 4.2. Modyfikatory precyzji

Modyfikator precyzji	Znaczenie
h	Znak ten umieszczony przed specyfikatorami typu `d`, `i`, `o`, `u` lub `x` informuje funkcję o tym, że argument jest wskaźnikiem do typu `short`
l	Znak ten umieszczony przed specyfikatorami typu `d`, `i`, `o`, `u` lub `x` informuje funkcję o tym, że argument jest wskaźnikiem do typu `long`
L	Znak ten umieszczony przed specyfikatorami typu `e`, `f` lub `g` informuje funkcję o tym, że argument jest wskaźnikiem do typu `long double`

ĆWICZENIE

4.2 Napisz program, który pobierze z klawiatury, a następnie wyświetli na ekranie liczbę pięciocyfrową (3 pierwsze cyfry powinny być zapisane w jednej zmiennej, natomiast 2 pozostałe w drugiej).

```
1: /* Przyk-lad 4.2 */
2: /* Przyklad demonstruje uzycie specyfikatorow */
3: /* typu w fimkcji scanf() */
4: #include <stdio.h>
5: int liczba1, liczba2;
6: char napis[30];
7: int main()
8: {
9:       printf("\nWpisz 5-cyfrowa liczbe: ");
10:      scanf("%3d%2d", &liczba1, &liczba2);
11:      printf("\nWpisales liczby %d i %d.\n",
12:            liczba1, liczba2);
13:      return 0;
14: }
```

W rozwiązaniu powyższego ćwiczenia użyto w funkcji `scanf()` specyfikatora typu `%d` z liczbą określającą szerokość pola — wpisanie pięciocyfrowej liczby powoduje przypisanie pierwszych trzech cyfr zmiennej `liczba1` oraz pozostałych dwóch zmiennej `liczba2`.

ĆWICZENIE

4.3 Napisz program, który pobierze z klawiatury, a następnie wyświetli na ekranie dowolną liczbę dziewięciocyfrową.

```
1: /* Przyklad 4.3 */
2: /* Przyklad demonstruje uzycie specyfikatorow */
3: /* typu w fimkcji scanf() */
4: #include <stdio.h>
5: #define rozmiar 30
6: char napis[rozmiar];
7: long liczba;
8: int main()
9: {
10:     printf("Wpisz dowolna liczbe 9-cyfrowa: ");
11:     scanf("%ld", &liczba);
12:     printf("\nWpisales liczbe: %ld\n", liczba);
13:     return 0;
14: }
```

W rozwiązaniu powyższego ćwiczenia użyto w funkcji `scanf()` specyfikatora typu `%ld`. Jest to typ specyfikatora z dodatkowym modyfikatorem precyzji `l`, który wskazuje, że argument nie jest wskaźnikiem do typu `integer` (`%d`), ale do typu `long`.

Funkcje wyjścia

W tym rozdziale omówione zostaną dwie funkcje: `putchar()` oraz `puts()`.

Funkcja `putchar()` wysyła pojedyncze znaki do standardowego wyjścia (na ekran).

Prototyp funkcji `putchar()`:

```
int putchar(int c);
```

Zauważ, że zarówno argument funkcji, jak i wartość zwracana są typu `int`. Na pewno się domyślasz, że to odpowiedniki liczbowe poszczególnych znaków w kodzie ASCII (0 – 255).

ĆWICZENIE

4.4 Napisz program, który wyświetli na ekranie wszystkie znaki kodu ASCII. Użyj funkcji `putchar()`.

```
1: /* Przyklad 4.4 */
2: /* Przyklad demonstruje uzycie fimkcji putchar() */
3: #include <stdio.h>
4: int main()
5: {
6:     int licznik;
7:     for (licznik = 0; licznik < 256; licznik++)
8:     {
```

```
9:      putchar(licznik);
10:     printf("\n");
11:     }
12:     return 0;
13: }
```

W rozwiązaniu powyższego ćwiczenia użyta została pętla `for`, która odlicza od 0 do 255 i wyświetla poszczególne znaki kodu ASCII odpowiadające każdej z wartości przyjmowanych przez zmienną `licznik`.

Kolejną funkcją wyjścia jest funkcja `puts()`, o której była już mowa w poprzednich rozdziałach książki. Pozwala ona wyświetlać na ekranie całe wiersze tekstu.

Prototyp funkcji `puts()`:

```
int puts(char *cp);
```

`cp` jest wskaźnikiem dla pierwszego znaku łańcucha, który ma być wyświetlony na ekranie. Funkcja zwraca wartość dodatnią, jeśli dana operacja została wykonana pomyślnie lub „–1" (EOF), jeśli miał miejsce jakikolwiek błąd.

ĆWICZENIE

4.5 Napisz program, który wyświetli na ekranie dowolny łańcuch znaków. Użyj funkcji `puts()`.

```
1: /* Przyklad 4.5 */
2: /* Przyklad demonstruje uzycie funkcji puts() */
3: #include <stdio.h>
4: char napis[30] = "Dowolny lancuch znakow";
5: main()
6: {
7:      puts(napis);
8:      return 0;
9: }
```

Operacje na łańcuchach znaków

Łańcuchy znaków pojawiły się już w poprzednich rozdziałach tej książki. Nauczyłeś się je deklarować, przydzielać im pamięć (`malloc`) oraz wyświetlać je na ekranie. W tym rozdziale poznasz zaawansowane operacje na łańcuchach, a także nauczysz się je kopiować oraz łączyć ze sobą.

Kopiowanie łańcuchów znaków

Postanowiłem opisać w tej książce jedynie dwie podstawowe funkcje służące do kopiowania łańcuchów: `strcpy()` oraz `strncpy()`.

Funkcja `strcpy()` kopiuje cały łańcuch znaków (łącznie ze znakiem `\0`) do innego miejsca w pamięci komputera.

Prototyp funkcji `strcpy()`:

```
char *strcpy( char *przeznaczenie, char *zrodlo );
```

Argument *zrodlo* jest wskaźnikiem dla pierwszego znaku łańcucha, który chcemy przekopiować. Natomiast argument *przeznaczenie* to wskaźnik do pierwszego elementu docelowego łańcucha, do którego chcemy przekopiować źródłowy łańcuch. Funkcja zwraca wskaźnik do pierwszego elementu łańcucha docelowego (`char *przeznaczenie`).

ĆWICZENIE

4.6 Napisz program, który przekopiuje dowolny łańcuch znaków do innego łańcucha (pamiętaj o wcześniejszym zadeklarowaniu docelowego łańcucha oraz o przypisaniu mu pamięci).

```
1: /* Przyklad 4.6 */
2: /* Przyklad demonstruje uzycie funkcji strcpy() */
3: #include <stdlib.h>
4: #include <stdio.h>
5: #include <string.h>
6: char zrodlo[] = "Nasz lancuch zrodlowy";
7: char docel1[30];
8: char *docel2;
9: int main()
10:{
11:     printf("\nLancuch zrodlowy: %s", zrodlo);
12:     strcpy(docel1, zrodlo);
13:     printf("\n1-y lancuch docelowy po wykonaniu ");
14:     printf("operacji kopiowania: %s", docel1);
15:     docel2 = (char *)malloc(strlen(zrodlo)+1);
16:     strcpy(docel2, zrodlo);
17:     printf("\n2-gi lancuch docelowy po ");
18:     printf("przekopiowaniu: %s \n", docel2);
19:     return 0;
20:}
```

Rozwiązanie ćwiczenia może wydać się trochę skomplikowane, opiszę więc kolejne działania krok po kroku.

Wiersz 3 – 5: dołączasz trzy pliki nagłówkowe — *stdio.h, stdlib.h* (ponieważ używana jest funkcja `malloc()`) i *string.h* (plik niezbędny przy wykonywaniu operacji na łańcuchach znaków).

Wiersz 6: deklarujesz źródłowy łańcuch znaków.

Wiersz 7: deklarujesz trzydziestoelementową tablicę zmiennych typu `char` jako miejsce docelowe dla kopiowanego łańcucha.

Wiersz 8: deklarujesz wskaźnik do typu `char` (`*docel2`) — ma to być wskaźnik do pierwszego elementu skopiowanego łańcucha znaków, trzeba udostępnić dla niego blok pamięci (patrz: wiersz 13).

Wiersz 12: kopiujesz źródłowy łańcuch do tablicy `docel1[]`.

Wiersz 15: używasz funkcji `malloc()`, aby udostępnić blok pamięci dla kopiowanego łańcucha znaków; liczba bajtów w pamięci musi być równa długości łańcucha źródłowego (używana jest funkcja `strlen()`) plus 1 bajt.

Wiersz 16: kopiujesz źródłowy łańcuch do miejsca przeznaczenia (`docel2`).

Drugą funkcją służącą do kopiowania łańcuchów znaków jest `strncpy()`. W przeciwieństwie do funkcji `strcpy()` funkcja `strncpy()` pozwala na określenia liczby znaków, które mają być przekopiowane do miejsca przeznaczenia.

Prototyp funkcji `strncpy()`:

```
char *strncpy(char *przeznaczenie, char *zrodlo, size_t n);
```

Argument `zrodlo` jest wskaźnikiem do pierwszego znaku łańcucha, który chcemy przekopiować. Natomiast argument `przeznaczenie` to wskaźnik do pierwszego elementu docelowego łańcucha, do którego chcemy przekopiować źródłowy łańcuch. `strncpy()` kopiuje pierwsze `n` znaków łańcucha źródłowego do miejsca przeznaczenia. Funkcja zwraca wskaźnik do pierwszego elementu łańcucha docelowego.

ĆWICZENIE

4.7 Napisz program, który skopiuje pierwsze 5 znaków dowolnego łańcucha do określonego miejsca przeznaczenia.

```
1: /* Przyklad 4.7 */
2: /* Przyklad demonstruje uzycie funkcji strncpy() */
3: #include <stdio.h>
4: #include <string.h>
```

```
5: char zrodlo[] = "To jest lancuch zrodlowy";
6: char docel[10];
7: int main()
8: {
9:      printf("\nLancuch zrodlowy: %s", zrodlo);
10:     strncpy(docel, zrodlo, 5);
11:     printf("\nSkopiowano znaki: %s", docel);
12:     return 0;
13:}
```

W **wierszach 5 – 6** zadeklarowano źródłowy łańcuch znaków `zrodlo[]` oraz tablicę zmiennych typu `char`, która ma przechowywać skopiowany łańcuch. W **wierszu 10** użyto funkcji `strncpy()` w celu skopiowania pięciu znaków łańcucha źródłowego do miejsca przeznaczenia.

Łączenie łańcuchów znaków

Funkcjami używanymi do łączenia łańcuchów znaków są `strcat()` oraz `strncat()`.

Prototyp funkcji `strcat()`:

```
char *strcat(char *lancuch1, char *lancuch2);
```

Funkcja `strcat()` dodaje kopię łańcucha `lancuch2` na koniec łańcucha `lancuch1`. Wartością zwracaną jest wskaźnik pierwszego elementu łańcucha `lancuch1`.

ĆWICZENIE

4.8 Napisz program, który połączy dwa dowolne łańcuchy znaków.

```
1: /* Przyklad 4.8 */
2: /* Przyklad demonstruje uzycie funkcji strcat() */
3: #include <stdio.h>
4: #include <string.h>
5: char lancuch1[50] = "Pierwszy lancuch";
6: char lancuch2[] = " Drugi lancuch";
7: int main()
8: {
9:      printf("Pierwszy lancuch: %s\n", lancuch1);
10:     printf("Drugi lancuch: %s\n", lancuch2);
11:     strcat(lancuch1, lancuch2);
12:     printf("Pierwszy lancuch po dodaniu do niego ");
13:     printf("drugiego lancucha: %s\n", lancuch1);
14:     return 0;
15: }
```

W **wierszach 5 i 6** deklarowane są dwa łańcuchy znaków (zauważ, że `lancuch1` musi być umieszczony w większej tablicy, aby później znalazło się miejsce dla kopii drugiego łańcucha). W **wierszu 11** wywoływana jest funkcja `strcat()`, która dodaje kopię drugiego łańcucha na koniec pierwszego. Ostatecznie wyświetlony zostaje zmodyfikowany pierwszy łańcuch na ekranie.

Drugą funkcją służącą do łączenia łańcuchów znaków jest funkcja `strncat()`.

Prototyp funkcji `strncat()`:

```
char *strncat(char *lancuch1, char *lancuch 2, size_t n);
```

Funkcja `strncat()` jest bardzo podobna do `strcat()`. Jedyna różnica to możliwość określenia liczby znaków, które mają być skopiowane z jednego łańcucha i dodane na koniec drugiego (w przypadku funkcji `strncat()`).

ĆWICZENIE

4.9 Zmodyfikuj poprzedni program tak, aby do pierwszego łańcucha dodawał jedynie pierwsze 5 znaków z drugiego. Użyj funkcji `strncat()`.

```
1: /* Przyklad 4.9 */
2: /* Przyklad demonstruje uzycie funkcji strncat() */
3: #include <stdio.h>
4: #include <string.h>
5: char lancuch1[30] = "Pierwszy lancuch";
6: char lancuch2[] = " Drugi lancuch";
7: int main()
8: {
9:      printf("Pierwszy lancuch: %s\n", lancuch1);
10:     printf("Drugi lancuch: %s\n", lancuch2);
11:     strncat(lancuch1, lancuch2, 5);
12:     printf("Pierwszy lancuch po dodaniu 5 znakow ");
13:     printf("drugiego lancucha: %s\n", lancuch1);
14:     return 0;
15:}
```

Jak widzisz, w **wierszu 11** wywoływana jest funkcja `strncat()`, która skopiuje tylko 5 znaków z łańcucha `lancuch2` do łańcucha `lancuch1`.

Operacje na plikach

W dotychczasowych przykładach używane były jedynie funkcje wejścia-wyjścia pobierające dane z klawiatury i wysyłające komunikaty na ekran (terminal). Wielkich cudów takimi sposobami nie da się sprawić i prędzej czy później trzeba będzie sięgnąć po bardziej zaawansowane rozwiązania. Operacje na plikach są podstawą w tworzeniu poważnych programów. Prawie każdy program musi korzystać z danych, a nie zawsze są to znaki wprowadzane z klawiatury. Bardzo często stosuje się właśnie pliki, aby można było zapisywać i odczytywać większe porcje danych. Warto więc poznać zestaw funkcji służących do ich wykonywania. W tym rozdziale przedstawię tylko operacje na plikach tekstowych, które są wystarczające dla większości zastosowań. Jestem pewien, że po dobrym opanowaniu operacji na plikach tekstowych żaden Czytelnik nie będzie miał problemu, żeby indywidualnie poszerzyć swoją wiedzę i poznać operacje na plikach binarnych bez pomocy tej książki. Chciałbym podkreślić, że celem tej książki jest łagodne i bezbolesne wprowadzenie wszystkich Czytelników w świat języka C. Zaprezentowanie 20 różnych funkcji do operacji na plikach nie jest najlepszym sposobem na osiągnięcie tego celu.

Aby utworzyć plik w języku C, należy zdefiniować wskaźnik do pliku. W pliku nagłówkowym `<stdio.h>` jest zdefiniowany specjalny wskaźnik do tego celu — `FILE *`.

Deklaracja wskaźnika do pliku wygląda tak:

```
FILE *wskaznik_do_pliku;
```

Otwieranie, tworzenie i zamykanie plików tekstowych

Aby utworzyć nowy plik lub otworzyć już istniejący, używamy funkcji `fopen` o następującym prototypie:

```
FILE* fopen(const char* nazwa_pliku, const char* tryb);
```

Funkcja `fopen()` zwraca wskaźnik do pliku lub `NULL` (jeśli nie można otworzyć pliku). Następnie za pomocą tego wskaźnika można przesuwać się w obrębie pliku i odpowiednio go edytować lub odczytywać.

Nazwa_pliku to ścieżka do pliku lub sama nazwa pliku (jeśli plik znajduje się w tym samym katalogu, co program wywołujący funkcję). Warto

pamiętać, że w systemie Windows do definiowania ścieżki używa się backslasha \ — np. „C:\Programy\nowy_plik.txt". Nie można niestety definiować ścieżek w ten sposób, ponieważ kompilator wyszukuje znaczniki w tekście — np. \n zinterpretuje jako wstawienie nowej linii — i takie wywołanie wywołałoby niepożądane rezultaty. Rozwiązaniem jest zastąpienie jednego backslasha dwoma — \\. Wtedy nazwa pliku musiałaby wyglądać tak: „C:\\Programy\\nowy_plik.txt".

Z kolei tryb oznacza sposób otwarcia pliku. Trzy najważniejsze tryby to:

- „r" — otwarcie pliku do odczytu;
- „w" — otwarcie pliku do zapisu;
- „a" — otwarcie pliku w trybie dopisywania.

Otwarcie pliku do odczytu jest jasne. Tryb zapisu różni się od trybu dopisywania tym, że przy zapisywaniu nadpisywane są ewentualnie już istniejące dane. Natomiast w trybie dopisywania edycja rozpoczynana jest od końca pliku, więc wszystkie nowe dane zostają dopisane do już istniejącego tekstu.

Można wybrać tylko jeden z tych podstawowych trybów (istnieją również bardziej zaawansowane tryby, ale to zagadnienie wykracza poza zakres tej książki).

Jeżeli plik nie istnieje, a wybrana została opcja „w" lub „a", tworzony jest nowy plik. Jeśli operacja się powiedzie, to — tak jak już wspominałem — zwracany jest wskaźnik do początku pliku, a jeśli nie — zwracany jest NULL.

Na koniec plik należy zamknąć za pomocą funkcji o nazwie fclose:

```
int fclose(FILE *wskaznik_do_pliku)
```

Jeżeli operacja się powiedzie, zwracana jest wartość 0, a jeśli nie — zostaje zwrócony tzw. kod błędu EOF (zdefiniowany w pliku nagłówkowym <stdio.h>).

Odczytywanie pliku tekstowego

W celu odczytania pliku tekstowego należy użyć jednej z trzech funkcji: fscanf, fgetc lub fgets.

Odczytywanie pliku za pomocą funkcji fscanf

fscanf działa tak jak scanf, tyle że dodajemy dodatkowo, jako pierwszy parametr, wskaźnik do pliku, który chcemy odczytać, np.:

```
fscanf(wskaznik_do_pliku, „%d", zmienna_int)
```

Takie wywołanie funkcji spowoduje odczytanie elementu z pliku jako typ int i przypisanie go do zmiennej zmienna_int. Jako ciekawostkę dodam, że zapis:

```
fscanf(stdin, „%d", zmienna_int)
```

jest równoznaczny z zapisem:

```
scanf(„%d", zmienna_int)
```

ponieważ jako pierwszy argument podany został tzw. standardowy strumień wejścia — czyli znaki wpisywane z klawiatury.

fscanf zwraca liczbę przeczytanych elementów lub EOF, jeśli wystąpi błąd.

Po odczytaniu kolejnych elementów wskaźnik pliku jest odpowiednio przesuwany na miejsce, w którym kończy się dany element.

Najlepiej korzystać z funkcji fscanf, jeśli odczytujemy plik, który zawiera szczególny rodzaj danych — nie tylko tekst, ale np. liczby różnych typów. Można wtedy bezpośrednio zapisać kolejne elementy pliku do odpowiednich zmiennych, bez konieczności późniejszego skomplikowanego przetwarzania i konwertowania tekstu.

ĆWICZENIE

4.10 Napisz program, który wypisze na ekranie zawartość utworzonego wcześniej pliku tekstowego o nazwie *plik.txt*. Plik ten zawiera tylko liczby typu int oraz float zapisane jedna po drugiej. Wykorzystaj funkcję fscanf.

W dowolnym edytorze tekstowym utwórz plik *plik.txt* o następującej zawartości:

```
1 1.2 2 2.4 3 3.5
```

Program służący do jego odczytania może wyglądać następująco:

```
1: /* Przyklad 4.10 */
2: /* Program wypisujacy na ekranie zawartosc pliku tekstowego */
3: #include <stdio.h>
4: int main()
```

```
5: {
6:     FILE *plik;
7:     int a;
8:     float b;
9:     if ((plik = fopen("plik.txt","r")) != NULL)
10:    {
11:        while (fscanf(plik, "%d %f", &a, &b) == 2)
12:            printf ("%d %f ", a, b);
13:    }
14:    else
15:        return -1;
16:    fclose(plik);
17:    return 0;
18:}
```

W **wierszach 6 – 8** tworzony jest wskaźnik do pliku oraz odpowiednie zmienne do przechowywania odczytanych z niego wartości. W **wierszu 9** otwieramy plik w trybie do odczytu i jednocześnie sprawdzamy, czy zwracana wartość nie jest równa `NULL`. W przypadku błędu wychodzimy z programu. Jeżeli wszystko jest w porządku, funkcja `fopen` zwraca wskaźnik do otwartego pliku *plik.txt*. W **wierszu 11** uruchamiana jest pętla, która pobiera w parach elementy typu `int` i `float` do momentu, aż zwrócona zostanie wartość inna niż 2. Warto przypomnieć, że funkcja `fscanf` zwraca liczbę przeczytanych elementów lub `EOF`. Na koniec należy pamiętać, aby zamknąć plik — **wiersz 16**.

Odczytywanie pliku tekstowego za pomocą funkcji fgetc

```
int fgetc(FILE *wskaznik_do_pliku)
```

Funkcja `fgetc` służy do odczytywania pojedynczych znaków. Zwraca w wyniku znak jako typ `int` (po przekonwertowaniu z typu `unsigned char`) lub — w razie błędu — EOF. Można też użyć makra `getc`, które ma takie samo wywołanie jak funkcja. (Makra opisane są w rozdziale 5.). W tym przypadku makro `getc` jest szybsze od funkcji `fgetc`.

ĆWICZENIE

4.11 Napisz program, który wypisze zawartość pliku *plik.txt* na ekranie. Plik *plik.txt* zawiera tylko tekst. Wykorzystaj funkcję `fgetc`.

```
1: /* Przyklad 4.11 */
2: /* Program wypisujacy na ekranie zawartosc pliku tekstowego */
3: #include <stdio.h>
4: int main()
```

```
5: {
6:     FILE *plik;
7:     char c;
8:     if ((plik = fopen("plik.txt","r")) != NULL)
9:     {
10:        while ((c = fgetc(plik)) != EOF)
11:            printf("%c ", c);
12:    }
13:    else
14:        return -1;
15:    fclose(plik);
16:    return 0;
17:}
```

Podobnie jak w ćwiczeniu 4.10, w **wierszach 6 – 8** deklarujemy odpowiedni wskaźnik do pliku i zmienną do przechowywania odczytywanych znaków oraz otwieramy plik w trybie do odczytu. Istotną zmianą są **wiersze 10 – 11**, w których za pomocą funkcji `getc` odczytujemy, znak po znaku, zawartość pliku. W każdej iteracji odczytywany jest jeden znak, który następnie zostaje zapisany w zmiennej `c` (zwracanej przez funkcję `fgetc`) oraz wypisany na ekran. Pętla wykonuje się aż do momentu, kiedy funkcja `fgetc` zwróci wartość `EOF`.

Odczytywanie pliku tekstowego za pomocą funkcji fgets

```
char* fgets(char *lancuch, int dlugosc, FILE* wsk_do_pliku)
```

Funkcja `fgets` służy do odczytywania z pliku tekstowego kolejnych łańcuchów znaków i zapisywania ich w zmiennej (`char *lancuch`) przekazywanej jako parametr funkcji. Kolejne znaki z pliku zapełniają łańcuch aż do napotkania znaku końca linii lub zapisania liczby znaków przekazanej jako parametr (`int dlugosc`). Funkcja zwraca `NULL` lub wskaźnik do łańcucha, w którym zapisano znaki — czyli do tego samego łańcucha, na który wskazuje wskaźnik przekazany jako parametr.

ĆWICZENIE

4.12

Napisz program, który wypisze zawartość pliku *plik.txt* na ekranie. Plik ten zawiera tylko tekst. Wykorzystaj funkcję `fgets`.

```
1: /* Przyklad 4.12 */
2: /* Program wypisujacy na ekranie zawartosc pliku tekstowego */
3: #include <stdio.h>
```

```
4: int main()
5: {
6:     FILE *plik;
7:     char lancuch[80];
8:     if ((plik = fopen(„plik.txt","r")) != NULL)
9:     {
10:        while (fgets(lancuch, sizeof(lancuch), plik) != NULL)
11:            printf("%s ", lancuch);
12:    }
13:    else
14:        return -1;
15:    fclose(plik);
16:    return 0;
17:}
```

Jak widać, program wygląda bardzo podobnie do poprzednich, różni się jedynie w **wierszach 10 – 11**. Tu uruchamiamy w pętli funkcję `fgets`, która w kolejnych iteracjach odczytuje maksymalnie 80 znaków z pliku (`sizeof(lancuch)`) i zapisuje je w tablicy `lancuch`. Jeżeli napotkany zostanie znak nowej linii (`\n`), do tablicy `lancuch` zostanie zapisana mniejsza liczba znaków z `\n` na końcu. Jeżeli funkcja `fgets` zwróci wartość `NULL`, oznacza to koniec pliku lub błąd.

Zapisywanie pliku tekstowego

Zapisywanie do pliku tekstowego za pomocą funkcji fprintf

`fprintf` to prawie taka sama funkcja jak `printf`. Różnica, podobnie jak w przypadku funkcji `scanf` i `fscanf`, polega na dodaniu strumienia (którym może też być wskaźnik do pliku) jako parametru; przykładowo instrukcja:

```
fprintf(wskaznik_do_pliku, „%d", zmienna_typu_int)
```

spowoduje zapisanie do pliku, na który wskazuje wskaźnik *wskaznik_*
↪do_pliku, wartości typu `int` przechowywanej w zmiennej *zmienna_*
↪typu_int.

Jeżeli wystąpi błąd, funkcja `fprintf` zwraca wartość mniejszą od zera, natomiast w przeciwnym wypadku zwraca liczbę elementów, które zostały zapisane do pliku.

ĆWICZENIE

4.13

Napisz program, który utworzy nowy plik tekstowy o nazwie *plik.txt*, a następnie zapisze do niego w kolejnych liniach liczby od 0 do 10.

```
1: /* Przyklad 4.13 */
2: /* Program zapisujacy do pliku liczby od 0 do 10 przy uzyciu */
3: /* funkcji fprintf() */
4: #include <stdio.h>
5: int main()
6: {
7:     FILE *plik;
8:     int i;
8:     if ((plik = fopen("plik.txt","w")) != NULL)
9:     {
10:        for (i = 0; i <= 10; i++)
11:            if (fprintf(plik, "%d\n", i) < 0)
12:                return -1;
13:    }
15:    else
16:        return -1;
17:    fclose(plik);
18:    return 0;
19:}
```

W **wierszach 7 – 8** deklarowane są odpowiednie zmienne: wskaźnik do pliku oraz zmienna typu `int`, którą posłużymy się do wygenerowania kolejnych liczb od 0 do 10. W **wierszu 6** jednocześnie tworzymy nowy plik o nazwie *plik.txt* i otwieramy go w trybie zapisu. Jeżeli operacja się nie powiedzie, instrukcja `return -1` spowoduje wcześniejsze wyjście z programu. Instrukcje w **wierszach 10 – 11** wywołują funkcję `fprintf` w pętli, co powoduje zapisanie kolejnych liczb oraz znaku nowej linii w pliku *plik.txt*. Jeżeli `fprintf` zwróci błąd (wartość mniejsza od 0), wywołujemy instrukcję `return -1`, aby spowodować wcześniejsze wyjście z programu. Ostatecznie musimy zamknąć plik, wywołując funkcję `fclose` w **wierszu 17**.

Zapisywanie do pliku tekstowego za pomocą funkcji fputc

```
int fputc(int znak, FILE *wskaznik_do_pliku)
```

Każde wywołanie funkcji `fputc` powoduje zapisanie do pliku, na który wskazuje *wskaznik_do_pliku*, pojedynczego znaku (przekazanego jako parametr — `int` *znak*). W przypadku poprawnego wykonania funkcja zwraca zapisany znak (jako `int`). W przeciwnym razie zwracana jest wartość `EOF`.

Świetnym ćwiczeniem zapisywania i odczytywania plików tekstowym jest przedstawiony poniżej przykład z kopiowaniem plików. Przy okazji można się nauczyć, w jaki sposób przekazywać argumenty do programu z linii poleceń.

ĆWICZENIE

4.14 Napisz program, który otworzy istniejący plik tekstowy i skopiuje jego zawartość do innego, nowo utworzonego pliku. Wykorzystaj funkcje `fputc` i `fgetc`. Nazwy pliku źródłowego i docelowego muszą zostać podane z linii poleceń.

```
1: /* Przyklad 4.14 */
2: /* Program kopiujacy plik tekstowy przy uzyciu */
3: /* funkcji fputc i fgetc */
4: #include <stdio.h>
5: int main(int argc, char* argv[])
6: {
7:     FILE *plik_zrodlowy, *plik_docelowy;
8:     int znak;
9:     if (argc != 3)
10:    {
11:        printf("Uruchomienie programu: nazwa_programu plik_zrodlowy
           ↪plik_docelowy \n");
12:        return -1;
13:    }
14:    if ((plik_zrodlowy = fopen(argv[1], „r")) == NULL)
15:        return -1;
16:    if ((plik_docelowy = fopen(argv[2], „w")) == NULL)
17:        return -1;
18:    while ((znak = fgetc(plik_zrodlowy)) != EOF)
19:        fputc(znak, plik_docelowy);
20:    fclose(plik_zrodlowy);
21:    fclose(plik_docelowy);
22:    return 0;
23:}
```

Niejednokrotnie zachodzi potrzeba przekazania argumentów do programu z linii poleceń. W tym celu często stosowane jest właśnie przekazywanie nazw plików wejściowych/wyjściowych. W **wierszu 5** funkcja `main()` posiada dodatkowe parametry. Parametr `char *argv[]` to tablica łańcuchów znaków przekazanych z linii poleceń — łącznie z nazwą programu, która stanowi pierwszy element tej tablicy. Parametr `int argc` to liczba elementów tablicy `argv` — czyli liczba parametrów plus nazwa programu. W naszym programie oczekujemy dwóch argumentów — nazwy pliku źródłowego oraz nazwy pliku docelowego. Parametr `argc`, przy poprawnym wywołaniu programu, powinien

więc być równy 3 (2 parametry+nazwa). Warunek ten sprawdzany jest w **wierszu 9** i w razie złego wywołania programu wypisany zostaje odpowiedni komunikat oraz następuje wcześniejsze wyjście z programu. W **wierszach 14** oraz **16** otwierane są pliki w odpowiednich trybach. `argv[1]` reprezentuje nazwę pliku źródłowego (przekazanego jako pierwszy argument), natomiast `argv[2]` reprezentuje nazwę pliku docelowego (przekazanego jako drugi argument programu). Najważniejszym fragmentem programu są instrukcje zawarte w **wierszach 18 – 19**, gdzie w każdej iteracji przy użyciu funkcji `fgetc` odczytywany jest jeden znak, który zostaje zapisany do pliku docelowego z wykorzystaniem funkcji `fputc`. Ostatecznie, w **wierszach 20 – 21**, obydwa pliki są zamykane za pomocą funkcji `fclose`.

Zapisywanie do pliku tekstowego za pomocą funkcji fputs

```
int fputs(char *lancuch_znakow, FILE *wskaznik_do_pliku)
```

Funkcja powoduje zapisanie łańcucha znaków przekazanego jako parametr do pliku, na który wskazuje *wskaznik_do_pliku*. W przypadku błędu zwracana jest wartość `EOF`, w przeciwnym razie zwrócona zostaje wartość większa od zera.

ĆWICZENIE

4.15

Napisz program, który otworzy istniejący plik tekstowy i skopiuje jego zawartość do innego, nowo utworzonego pliku. Wykorzystaj funkcje `fputs` i `fgets`. Nazwy pliku źródłowego i docelowego muszą zostać podane z linii poleceń.

```
1: /* Przyklad 4.15 */
2: /* Program kopiujacy plik tekstowy przy uzyciu */
3: /* funkcji fputs i fgets */
4: #include <stdio.h>
5: int main(int argc, char* argv[])
6: {
7:       FILE *plik_zrodlowy, *plik_docelowy;
8:       char lancuch[80];
9:       if (argc != 3)
10:      {
11:          printf("Uruchomienie programu: nazwa_programu plik_zrodlowy
             ↪plik_docelowy \n");
12:       return -1;
13:      }
14:      if ((plik_zrodlowy = fopen(argv[1], „r")) == NULL)
```

```
15:         return -1;
16:     if ((plik_docelowy = fopen(argv[2], „w")) == NULL)
17:         return -1;
18:     while (fgets(lancuch, sizeof(lancuch), plik_zrodlowy) != NULL)
19:         fputs(launch, plik_docelowy);
20:     return 0;
21: }
```

Program różni się od ćwiczenia 4.14 tylko instrukcjami w **wierszach 18 – 19**. Pętla `while` w każdej kolejnej iteracji odczytuje za pomocą funkcji `fgets` łańcuch znaków z pliku źródłowego i zapisuje go do pliku docelowego za pomocą funkcji `fputs`.

Co powinieneś zapamiętać z tego cyklu ćwiczeń?

- Co to są strumienie?
- Jakie standardowe strumienie wejścia-wyjścia wyróżnia się w języku C?
- Jaki jest prototyp funkcji `getchar()` oraz jak przebiega jej działanie?
- Jaki jest prototyp funkcji `gets()` oraz jak przebiega jej działanie?
- Z czego składa się funkcja `scanf()`?
- Jakie specyfikatory typu wyróżnia się w języku C?
- Jakie modyfikatory precyzji wyróżnia się w języku C?
- Jaki jest prototyp funkcji `putchar()` oraz jak przebiega jej działanie?
- Jaki jest prototyp funkcji `puts()` oraz jak przebiega jej działanie?
- Na jakiej zasadzie odbywa się kopiowanie łańcuchów znaków?
- Jaki jest prototyp funkcji `strcpy()` oraz jak przebiega jej działanie?
- Jaki jest prototyp funkcji `strncpy()` oraz jak przebiega jej działanie?
- Na jakiej zasadzie odbywa się łączenie łańcuchów znaków?
- Jaki jest prototyp funkcji `strcat()` oraz na jakiej zasadzie ona działa?
- Jaki jest prototyp funkcji `strncat()` oraz na jakiej zasadzie ona działa?

Ćwiczenia do samodzielnego wykonania

ĆWICZENIE 1.

Napisz program, który przekopiuje 5 elementów pierwszego łańcucha znaków do drugiego łańcucha.

ĆWICZENIE 2.

Napisz program, który doda 5 elementów pierwszego łańcucha znaków na koniec drugiego.

ĆWICZENIE 3.

Napisz program, który przekopiuje plik złożony z samych liczb zmiennoprzecinkowych oddzielonych spacjami do innego pliku.

ĆWICZENIE 4.

Napisz program, który doda dowolny tekst do wybranego pliku tekstowego.

Pamiętaj o odpowiednim argumencie funkcji `fopen`.

ĆWICZENIE 5.

Napisz program, który odczyta plik złożony z liczb całkowitych i skopiuje odczytane liczby do komórek tablicy dynamicznej. Tablica powinna mieć wielkość dostosowaną do liczby odczytanych elementów z pliku. Nazwa pliku powinna zostać podana z linii poleceń.

ĆWICZENIE 6.

Napisz program, który odczyta plik tekstowy i skopiuje każdy odczytany wyraz do kolejnych elementów tablicy łańcuchów znaków.

Tablica łańcuchów znaków jest tablicą wielowymiarową, ponieważ każdy element tablicy powinien zawierać tablicę znaków lub wskaźnik do pierwszego znaku w łańcuchu.

Język C dla guru

Drogi Czytelniku, czyżbyś opanował cały materiał z poprzednich części książki? Rozwiązałeś wszystkie ćwiczenia? Nie masz żadnych wątpliwości? Jesteś pewien, że nie masz żadnych wątpliwości? Hm... w takim razie możesz przekroczyć kolejne wrota fascynującej krainy języka C i zanurzyć się w bezmiernej głębinie wiedzy. Pamiętaj — stąd już nie ma powrotu. Z pewnością po lekturze tej książki sięgniesz po opracowania omawiające zaawansowane pojęcia związane z programowaniem w C (np. programowanie sieciowe) lub rozpoczniesz naukę C++. Ale nie mów hop, póki nie przeskoczysz. Najpierw opanuj materiał zawarty w tym rozdziale. Gotów? Jeśli tak, zapraszam do lektury rozdziału 5. Będzie w nim mowa o zaawansowanym zastosowaniu struktur i wskaźników do definicji struktur danych nazywanych listami, wskaźnikach do funkcji oraz dyrektywach preprocesora.

Struktury ze wskaźnikami

Nie jest wielką tajemnicą, zwłaszcza dla guru, że struktury też mogą zawierać wskaźniki jako pola oraz że można tworzyć wskaźniki do struktur. Warto jednak o tym wspomnieć na początku rozdziału, ponieważ w języku C wprowadzono operator ->, który ułatwia dostęp do wskaźników do struktur. Najłatwiej zrozumieć to na poniższym przykładzie.

ĆWICZENIE

5.1

Napisz program, w którym zdefiniujesz typ struktury zawierającej dowolne wskaźniki jako pola. Zdefiniuj również zmienną dla tego typu oraz wskaźnik do takiej struktury. Przypisz wartości odpowiednim polom oraz wypisz je na ekranie poprzez odwołanie się zarówno do zwykłej zmiennej, jak i do wskaźnika do struktury.

```
1: /* Przyklad 5.1 */
2: /* Przyklad ilustruje składnie używana do uzyskania dostepu */
3: /* do wskaznikow do struktur oraz wskaznikow bedacych */
4: /* elementami struktury */
5: #include <stdio.h>
6: typedef struct {
7:       int x;
8:       int *y;
9: } struktura;
10:int main()
11:{
12:     struktura *wsk_strukt;
13:     struktura strukt;
14:     int a = 10;
15:     int b = 20;
16:     strukt.x = a;
17:     strukt.y = &b;
18:     wsk_strukt = &strukt;
19:     printf („Wartosc pola x: %d\n", strukt.x);
20:     printf („Wartosc pola x odwolujac się przez wskaznik
        ↪do struktury: %d\n", wsk_strukt->x );
21:     printf(„Wartosc wskazywana przez pole *y: %d\n", *(strukt.y) );
22:     printf(„Wartosc wskazywana przez pole *y w przypadku
        ↪odwolania sie przez wskaznik do struktury: %d\n",
        ↪*(wsk_strukt->y) );
23:     return 0;
24:}
```

Program ilustruje cztery przypadki użycia wskaźników do struktur oraz struktur ze wskaźnikami. W **wierszach 6 – 9** zadeklarowano typ struktury z dwoma polami, z których jedno jest wskaźnikiem, a drugie zwykłą zmienną typu `int`. W **wierszach 12** oraz **13** zdefiniowano odpowiednio wskaźnik oraz zmienną typu `struktura`.

Pierwszy przypadek (**wiersz 19**) to najprostsze odwołanie się do pola `x` zmiennej typu `struktura`.

Drugi przypadek (**wiersz 20**) to odwołanie się do zmiennej `x` typu `int`, ale poprzez wskaźnik do struktury zawierającej tę zmienną. Tutaj wła-

śnie przydał się wspomniany operator ->. W przypadku braku takiego operatora należałoby się odwołać do tej zmiennej w następujący sposób — `(*wsk_strukt).x` — co nie wyglądałoby zbyt czytelnie.

W **wierszu 21** znajduje się przykład odwołania do wskaźnika `y` wewnątrz zwykłej zmiennej typu `struktura`. Poprzez instrukcję `strukt.y` uzyskujemy wskazywany adres, a następnie poprzez (operator *) — wartość pod tym adresem.

Wiersz 22 demonstruje najtrudniejszy z przypadków — czyli odwołanie się do wartości pola `*y` będącego wskaźnikiem poprzez wskaźnik do struktury go zawierającej. Najpierw, podobnie jak w drugim przypadku, uzyskujemy dostęp do wskaźnika `y` poprzez wskaźnik do struktury — `wsk_strukt->y`. Jednak w ten sposób uzyskany został tylko adres — w celu uzyskania wartości pod tym adresem należy użyć operatora * — `*(wsk_strukt->y)`. Gdyby nie było operatora ->, trzeba by użyć następującej instrukcji: `*((*wsk_strukt).y)`.

Poznałeś zatem już wszystkie sekrety związane ze składnią. Czas na wyjaśnienie, komu i do czego może się to przydać.

Jeśli chodzi o same wskaźniki do struktur — na pewno można je przekazać przez wskaźnik jako parametr do funkcji. Można też definiować dynamiczne tablice struktur. Podobne zastosowania możliwe są także w odniesieniu do wskaźników użytych jako pola struktur — zawsze można sobie zdefiniować dynamiczne tablice jako część struktury, choć pewnie nie jest to zbyt pospolite działanie. Najciekawszym zastosowaniem, zarazem chyba najbardziej praktycznym i popularnym, jest użycie wskaźników do struktur wewnątrz struktur. Co ciekawe, najczęściej typ wskaźnika będący polem struktury jest taki sam jak typ tej struktury. Czyżby takie pole wskazywało na strukturę, w której się znajduje? Odpowiedź brzmi: też, niekoniecznie jednak tylko na siebie, a przede wszystkim na inne elementy takiego samego typu. W ten sposób tworzy się tzw. listę. Lista to ciąg połączonych elementów. Połączone są one w taki sposób, że każdy element wskazuje na kolejny element po nim.

Przykładem listy — jeśli odwołać się do życia realnego — jest pociąg, elementami listy są poszczególne wagony. Do każdego wagonu dołączony jest kolejny (poza lokomotywą, która stanowi szczególny element takiej listy). Lista jest alternatywą dla tablicy i w wielu zastosowaniach okazuje się lepszym rozwiązaniem. Jest szczególnie efektywna w przypadku zarządzania pamięcią.

Kontynuując wcześniejszą analogię listy do pociągu, tablicę można porównać do wagonu. Elementami takiej tablicy są poszczególne, ponumerowane miejsca w wagonie. Do tablicy szybciej można się dostać — wystarczy podać indeks elementu (numer miejsca) i już wiadomo, co się w danym polu znajduje. Jeśli chodzi o listę, można polegać jedynie na wskaźnikach — trzeba szukać danego elementu, przeskakując z jednego do drugiego za pomocą wskaźnika do sąsiada. Lista jest jednak lepsza, jeśli chodzi o zarządzanie pamięcią dla dodawanych i usuwanych dynamicznie elementów. Jeśli usuwany jest element z listy — lub wagon z pociągu — trzeba tylko zmienić jeden wskaźnik poprzedniego elementu tak, aby wskazywał na kolejny element za tym usuniętym. Podobnie po usunięciu wagonu z pociągu spina się tylko sąsiadujące z nim wagony. W przypadku tablicy po usunięciu elementu zostaje puste, nieużywane pole i pamięć nie może być zwolniona. Nie można skurczyć wagonu, jeśli potrzeba 20 miejsc, nie można usunąć połowy miejsc z wagonu 40-osobowego. Możliwa jest oczywiście realokacja pamięci dla tablicy dynamicznej, ale wiązałoby się to z koniecznością podstawienia nowego wagonu (np. 50-osobowego) i przestawienia do niego wszystkich pasażerów. Lepszym rozwiązaniem jest dopięcie 10-osobowego wagonu na koniec pociągu. Może nieco przesadziłem z tą analogią — PKP wszystkie wagony ma bardzo podobne i nowoczesne zarządzanie miejscami w pociągu chyba nie ma zbyt dużego sensu, przecież pasażer może sobie postać w sąsiedztwie komfortowej toalety przez 8 godzin... ale to już temat na inne przykłady. Przejdźmy więc do praktycznej implementacji takiego pociągu PKP:

```
struct {
  struct wagon *nastepny;
  int *miejsca_w_wagonie;
} wagon;
```

Tak może wyglądać typ struktury dla wagonu PKP. Przy definicji każdego elementu należy zacząć od lokomotywy — ustawić wskaźnik `nastepny` jako `NULL` (czyli brak kolejnego elementu). Następnie trzeba utworzyć kolejny element i ustawić wskaźnik `nastepny` tak, aby wskazywał na lokomotywę (np. `wagon1.nastepny = &lokomotywa`). Przy definicji każdego elementu trzeba też udostępnić odpowiednią ilość pamięci na miejsca w danym wagonie (stosując funkcję `malloc()`). Myślę, że jesteś już gotów na wykonanie kolejnego praktycznego ćwiczenia.

ĆWICZENIE

5.2 Napisz program, który utworzy dynamicznie listę reprezentującą pociąg złożony z 3 wagonów (w tym warsu) i lokomotywy; wykorzystaj przy tym strukturę typu wagon. Pamiętaj o udostępnieniu odpowiedniej ilości miejsca dla pasażerów w kolejnych wagonach — w przypadku PKP będzie to 60 miejsc dla każdego wagonu oraz 20 miejsc dla wagonu wars. Następnie usuń wagon1, tak aby zwolnić zużytą pamięć, i podepnij wagon2 do warsu.

```
1: /* Przyklad 5.2 */
2: /* Program tworzacy liste reprezentujaca pociag PKP */
3: #include <studio.h>
4: #include <stdlib.h>
5: struct wagon{
6:      struct wagon *nastepny;
7:      int *miejsca_w_wagonie;
8: } ;
9: typedef struct wagon wagon_PKP;
10:int main()
11:{
12:     wagon_PKP *lokomotywa, *wars, *wagon1, *wagon2;
13:     lokomotywa = (wagon_PKP *)malloc(sizeof(wagon_PKP));
14:     wars = (wagon_PKP *)malloc(sizeof(wagon_PKP));
15:     wagon1 = (wagon_PKP *)malloc(sizeof(wagon_PKP));
16:     wagon2 = (wagon_PKP *)malloc(sizeof(wagon_PKP));
17:     if !(wars && lokomotywa && wagon1 && wagon2) return -1;
18:     lokomotywa->nastepny = NULL;
19:     lokomotywa->miejsca_w_wagonie = NULL;
20:     wars->nastepny = lokomotywa;
21:     wars->miejsca_w_wagonie = (int *)malloc(20*sizeof(int));
22:     wagon1->nastepny = wars;
23:     wagon1->miejsca_w_wagonie = (int *)malloc(60*sizeof(int));
24:     wagon2->nastepny = wagon1;
25:     wagon2->miejsca_w_wagonie =  (int *)malloc(60*sizeof(int));
26:     printf("Usuwamy wagon1 i podpinamy wagon2 do wagonu WARS\n");
27:     wagon2->nastepny = wars;
28:     free(wagon1);
29:     return 0;
30: }
```

Wiersze 5 – 9: tworzysz strukturę wagonu PKP i definiujesz odpowiedni alias — `wagon_PKP` — reprezentujący nowy typ.

Wiersz 12: definiujesz wskaźniki do odpowiednich struktur. Gdyby zostały zdefiniowane zwykłe zmienne zamiast wskaźników, nie można by dynamicznie zwalniać i udostępniać pamięci.

Wiersze 14 – 17: udostępniasz odpowiednią ilość pamięci dla elementów. Jeśli operacja się nie powiedzie, wychodzisz z programu.

Wiersze 18 – 19: inicjalizujesz elementy lokomotywy — oba wskaźniki ustawiasz na `NULL`, ponieważ żaden wagon nie jest dołączony przed lokomotywą ani nie ma tam miejsc dla pasażerów.

Wiersze 20 – 25: zawierają inicjalizację pól innych wagonów, wagon `wars` jest podłączony bezpośrednio do lokomotywy, więc ustawiasz odpowiedni wskaźnik tak, aby na nią wskazywał. Udostępniasz pamięć dla miejsc pasażerskich poprzez proste i znane Ci już dobrze wywołanie funkcji `malloc()`. Analogiczną operację przeprowadzasz dla pozostałych wagonów.

Wiersze 27 – 28: usuwanie wagonu nr 1 jest bardzo proste — przestawiasz odpowiedni wskaźnik z wagonu `wagon2`, tak aby wskazywał na wagon `wars`, a następnie zwalniasz pamięć zajmowaną przez `wagon1` za pomocą funkcji `free()`. Dynamiczna alokacja pamięci za pomocą list jest bezcenna, ponieważ w przeciwieństwie do tablicy nie marnuje się miejsce po usunięciu wybranych elementów.

Wskaźniki do funkcji

Wskaźniki do funkcji to konstrukcje języka C stosowane przez największych guru. Nie są najpopularniejszymi mechanizmami, ale na pewno przydają się w zastosowaniach opisanych w dalszej części tej sekcji.

Wskaźniki definiuje się, aby wskazywać nie tylko na dane w pamięci, ale także na funkcje, które przecież też mają swoje adresy. Wskaźniki do funkcji deklaruje się w poniższy sposób:

```
int (*wsk_do_funkcji)(int)
```

Nawiasy są konieczne, ponieważ w przypadku deklaracji:

```
int *wsk_do_funkcji(int)
```

zadeklarowana zostałaby funkcja o nazwie `wsk_do_funkcji`, która zwracałaby wskaźnik do zmiennej typu `*int`.

Po zadeklarowaniu wskaźnika należy go zdefiniować, przypisując mu adres jakiejś funkcji, np. funkcji o prototypie:

```
int funkcja(int)
```

Trzeba pamiętać, że typ wartości zwracanej i typ parametrów wskaźnika i wskazywanej funkcji muszą być identyczne. Przypisanie adresu funkcji do wskaźnika jest bardzo proste:

```
wsk_do_funkcji = funkcja;
```

Nazwa funkcji jest też jednocześnie jej adresem. Wywołanie funkcji poprzez wskaźnik okazuje się również bardzo proste, np.:

```
int a;
int b = 1;
a = wsk_do_funkcji(b);
```

Wystarczyło tylko zamienić nazwę funkcji na nazwę wskaźnika, który na nią wskazuje.

Tyle wiedzy teoretycznej o składni wskaźników do funkcji. Teraz przydałoby się przedstawić dla nich jakieś zastosowanie. Świetnym przykładem jest mechanizm obsługi zdarzeń. Wyobraź sobie, że musisz napisać program, który będzie służył do przetwarzania danych z czujników (np. poziomu cieczy w zbiorniku) w pewnej fabryce. Czujnik wysyła dane do komputera, zwykle z pewną częstotliwością, ale może też wykrywać pewne krytyczne zdarzenia. Po przesłaniu takiego sygnału i danych na ten temat do komputera potrzebny jest program, który dokładniej przeanalizuje takie niskopoziomowe dane i zadecyduje, jak zareagować na takie zdarzenia. Do tego właśnie służą specjalne funkcje, które zajmują się obsługą tego typu zdarzeń. W programowaniu obiektowym, które być może poznasz przy okazji nauki języka C++, stosuje się pojęcia zdarzenia (z ang. *events*) i obsługi zdarzeń (z ang. *event handlers*).

ĆWICZENIE

5.3 Napisz program, który pozwoli na obsługę menu użytkownika. Zależnie od wyboru użytkownika program uruchomi odpowiednią funkcję. Zastosuj wskaźniki do funkcji.

```
1: /* Przykład 5.3 */
2: /* Napisz program, ktory zapewni obsluge przekroczenia poziomu */
3: /* cieczy w zbiorniku. Zastosuj wskazniki do funkcji */
4: #include <stdio.h>
```

```
5: void handler1(float);
6: void handler2(float);
7: void przekroczony_poziom(float, void (*handler)(float));
8: int main()
9: {
10:     float pomiar = 123.58;
11:     void (*wsk_do_obslugi)(float);
12:     wsk_do_obslugi = handler1;
13:     przekroczony_poziom(pomiar, wsk_do_obslugi);
14:     wsk_do_obslugi = handler2;
15:     przekroczony_poziom(pomiar, wsk_do_obslugi);
16:     return 0;
17: }
18: void przekroczony_poziom(float pomiar, void (*handler)(float x))
19: {
20:     printf("Wskazanie czujnika jest podejrzane, uruchamiam obsluge
        ↪zdarzenia\n");
21:     handler(pomiar);
22: }
23: void handler1(float x)
24: {
25:     if (x < 100)
26:     {
27:         printf("Wskazanie czujnika jest na granicach normy.\n");
28:         printf("Zalecane sprawdzenie czujnika w terminie do 7
            ↪dni.\n");
29:     }
30:     else printf("Wystapila awaria, zalecana natychmiastowa wymiana
        ↪czujnika\n");
31:  }
32: void handler2(float x)
33: {
34:     if (x < 200)
35:     {
36:          printf ("Wskazanie czujnika jest na granicach normy.\n");
37:          printf ("Zalecane sprawdzenie czujnika w terminie do 7
             ↪dni\n");
38:     }
39:     else printf("Wystapila awaria, zalecana natychmiastowa wymiana
        ↪czujnika\n");
40: }
```

Wiersze 5 – 7: deklarowane są funkcje używane w programie. Funkcja `przekroczony_poziom()` jest wywoływana za każdym razem, gdy wystąpi odpowiednie zdarzenie — czyli gdy do programu przesłane zostaną dane z czujnika. Funkcje `handler1()` oraz `handler2()` służą do obsługi tego zdarzenia.

Wiersz 11: następuje zdefiniowanie wskaźnika do funkcji. Ważne jest, aby wstawić nawiasy tam, gdzie trzeba, tak by nie skończyło się na definicji funkcji zwracającej wskaźnik. Parametry oraz wartość zwracana muszą być tego samego typu co funkcje, na które taki wskaźnik będzie wskazywał.

Wiersz 12: przypisanie adresu funkcji do wskaźnika jest bardzo proste, ponieważ nazwa funkcji jest jednocześnie jej adresem.

Wiersz 13: wywoływana jest funkcja, która odpowiada wystąpieniu zdarzenia. W tym przypadku jest to oczywiście duże uproszczenie rzeczywistości. Takie funkcje są zwykle automatycznie wywoływane przez część programu, która odpowiada komunikacji z urządzeniem (czujnikiem/sterownikiem) np. poprzez port szeregowy. Funkcji przekazywany jest parametr (czyli dane odczytane przez czujnik) oraz wskaźnik do funkcji obsługującej zdarzenie.

Wiersze14 – 15: wskaźnikowi do funkcji przypisywany jest adres do innej funkcji, w której zmieniony został sposób obsługi zdarzenia. Dzięki temu przy kolejnym wystąpieniu zdarzenia uruchomiona zostaje już inna funkcja obsługi.

Wiersz 21: wywoływana jest funkcja obsługująca zdarzenie poprzez wskaźnik przekazany jako parametr do funkcji `przekroczony_poziom()`.

Wiersze 23 – 40: definiowane są funkcje obsługujące zdarzenie.

Mam nadzieję, że wszyscy Czytelnicy zrozumieli zalety takiego programu. W powyższym przykładzie warto zauważyć, że gdy konieczna jest zmiana obsługi zdarzenia, wystarczy dodać definicję nowej funkcji obsługi (bez konieczności zmiany czy usuwania już istniejących) i zmienić dwa miejsca w kodzie — czyli przypisanie adresu nowej funkcji do wskaźnika i wywołanie zdarzenia `przekroczony_poziom()` (**wiersze 14 – 15**). Może niektórzy Czytelnicy pomyśleli, że przecież można wykorzystać instrukcję `switch`, by niepotrzebnie nie bawić się jakimiś dziwnymi wskaźnikami do funkcji. Ale co, jeśli mamy 20 rodzajów obsługi zdarzenia, co chwilę coś się zmienia i dodawane są nowe funkcje i mechanizmy? Programy, które pisze się dla profesjonalnych zastosowań, nie są statyczne. Wciąż coś się modyfikuje, poprawia, usuwa i dodaje. Trzeba zatem zadbać, aby zmieniać tylko to, co jest konieczne. W przeciwieństwie do amatorskich instrukcji `switch` nasz kod wygląda przejrzyście i profesjonalnie.

Tablice wskaźników do funkcji

Tablice wskaźników do funkcji mogą służyć do lepszego zarządzania programami podobnymi do tego z ćwiczenia 5.3.

Aby zdefiniować tablicę wskaźników do funkcji, które zarówno nie pobierają żadnych elementów, jak i nie zwracają żadnych wartości, należy ją zadeklarować i zdefiniować w poniższy sposób:

```
void (*wskaźniki_do_funkcji[])(void) = {funkcja1, funkcja2,
                                        funkcja3};
```

Jest to najprostszy przykład jednoczesnej definicji i deklaracji. Można też oddzielnie deklarować i definiować, ale wtedy trzeba się męczyć z ręcznym przydziałem pamięci dla takich wskaźników do funkcji za pomocą funkcji `malloc()`. Aby zatem program był czytelny, zalecam najpierw przypisać jakieś wskaźniki (chociażby `NULL`), a ewentualnie później podmienić je na inne. Trzeba jednak pamiętać, że każda funkcja w tablicy wskaźników musi mieć takie same parametry i wartość zwracaną.

ĆWICZENIE

5.4

Napisz program, w którym zdefiniujesz tablicę wskaźników do funkcji wykonujących podstawowe operacje arytmetyczne. Następnie wywołaj je wszystkie w pętli, odwołując się do poszczególnych elementów tablicy.

```
1: /* Przykład 5.4 */
2: /* Przykład demonstruje użycie tablicy wskaźników do fimkcji */
3: #include <stdio.h>
4: float mnozenie(float, float);
5: float dzielenie(float, float);
6: float dodawanie(float, float);
7: float odejmowanie(float, float);
8: int main()
9: {
10:     int i;
11:     float x, y;
12:     float (*wsk_do_funkcji[])(float, float) = {dodawanie,
        ↪odejmowanie,mnozenie, dzielenie};
14:     printf("Podaj dwie liczby: \n");
15:     scanf("%f", &x);
16:     scanf("%f", &y);
17:     for (i = 0; i < 4; i++)
18:         printf("Wynik: %f\n", wsk_do_funkcji[i](x,y));
```

```
19:     return 0;
20:}
21:float mnozenie(float a, float b)
22:{
23:     printf("Mnozenie\n");
24:     return a*b;
25:}
26:float dzielenie(float a, float b)
27:{
28:     printf("Dzielenie\n");
29:     return a/b;
30:}
31:float dodawanie(float a, float b)
32:{
33:     printf("Dodawanie\n");
34:     return a+b;
35:}
36:float odejmowanie(float a, float b)
37:{
38:     printf("Odejmowanie\n");
39:     return a-b;
40:}
```

Wiersz 12 zawiera definicję tablicy wskaźników do funkcji pobierających jako parametry dwie zmienne typu `float` oraz zwracających wartość również typu `float`. Do tablicy przypisane są od razu wskaźniki do funkcji zadeklarowanych na początku i zdefiniowanych na końcu programu.

Wiersze 17 – 19 zawierają wywołanie w pętli wszystkich funkcji w tablicy. Wywołanie funkcji odbywa się prawie tak samo jak przy pojedynczych wskaźnikach do funkcji. Funkcje wywołujemy poprzez ich nazwę, ale dodajemy tylko odpowiedni indeks tablicy przed parametrami w nawiasach.

Wiersze 21 – 40 zawierają tylko proste definicje funkcji wykonujących podstawowe działania arytmetyczne.

Preprocesor

W przykładowych programach zamieszczonych w poprzednich rozdziałach używane były zapisy typu `#include` oraz `#define`. Są to tzw. dyrektywy preprocesora. Preprocesor to program, który przetwarza

tekst programu, zastępując niektóre instrukcje innymi. W praktyce jest on częścią kompilatora, ale przetwarzanie tekstu przez preprocesor następuje przed samym procesem kompilacji.

Preprocesor, analizując program, wyszukuje różne dyrektywy (rozpoczynające się znakiem #) i w zależności od ich typu zastępuje tekst programu w sposób przez nie zdefiniowany. Przykładowo dyrektywa `#include <stdio.h>` nakazuje preprocesorowi włączyć do tekstu programu zawartość pliku nagłówkowego *stdio.h*. Natomiast dyrektywa `#define PI 3.14`, służąca do definiowania stałych symbolicznych, instruuje procesor, aby zamienił wszystkie wystąpienia słowa `PI` w programie liczbą `3.14`. Przeanalizujmy przykład programu z ćwiczenia 1.9:

```
1: /* Przyklad 1.9 */
2: /* Oblicza pole kuli */
3: #include <stdio.h>
4: #define PI 3.14
5: float PoleKuli;
6: const int R = 5;
7: main()
8: {
9:       PoleKuli = 4*PI*R*R;
10:      printf("Pole Kuli wynosi %f\n", PoleKuli);
11:      return 0;
12: }
```

Po przetworzeniu przez preprocesor program będzie wyglądał następująco:

```
1: /* Przyklad 1.9 */
2: /* Oblicza pole kuli */
3: Zawartosc pliku stdio.h
4: Pusta linia
5: float PoleKuli;
6: const int R = 5;
7: main()
8: {
9:       PoleKuli = 4*3.14*R*R;
10:      printf("Pole Kuli wynosi %f\n", PoleKuli);
11:      return 0;
12: }
```

W miejscu **wiersza 3** pojawi się zawartość pliku nagłówkowego `stdio.h`, **wiersz 4** z dyrektywą `define` zniknie, ponieważ kompilator nie będzie potrzebował tych informacji, natomiast w **wierszu 9** symbol `PI` zostanie zastąpiony wartością stałej symbolicznej — `3.14`.

Sparametryzowane makrodefinicje (makra)

Dyrektywa #define daje większe możliwości niż definicja prostej stałej symbolicznej. Można również tworzyć za jej pomocą tzw. sparametryzowane makrodefinicje (dalej będą zwane po prostu makrami), które są szczególnego rodzaju funkcjami. Prostym przykładem jest poniższe makro funkcji obliczającej maksimum dwóch liczb:

```
#define MAX(x,y) ( (x) > (y) ? (x) : (y) )
```

Ta dziwnie wyglądająca instrukcja ze znakami ? oraz : to nic innego, jak zwykła instrukcja warunkowa zapisana w odmienny sposób. Przykładowo następujący zapis:

```
wynik_operacji = x > y ? x : y
```

odczytujemy jako:

```
if (x > y)
   wynik_operacji = x;
else
   wynik_operacji = y;
```

Preprocesor po napotkaniu takiej dyrektywy zamieni wszystkie wystąpienia MAX(x,y) w programie ciągiem instrukcji ((x) > (y) ? (x) : (y)). Można to nazwać takim bezmyślnym podstawianiem tekstu w miejsce innego i porównać z zachowaniem często obserwowanym wśród leniwych uczniów lub studentów, które nazywa się copy-paste (kopiuj-wklej). Preprocesor to właśnie taki leniwy student. Kiedy napotyka ciąg znaków MAX(x,y), zmazuje go i w jego miejsce bezmyślnie wstawia ((x) > (y) ? (x) : (y)) — nieważne, w jakim kontekście MAX(x,y) wystąpi. Dlatego tak istotne są nawiasy — ich nadmiar nigdy nikomu nie zaszkodził, warto je wstawiać wszędzie tam, gdzie nie ma pewności, czy wystąpi np. prawidłowa kolejność operacji arytmetycznych.

Wyobraź sobie następujący przykład:

```
#define MAX(x,y) ( x > y ? x : y )
if (MAX(a,b) == b)
{
  Dowolny ciąg operacji
}
```

Po przetworzeniu instrukcja warunkowa wyglądałaby tak:

```
if ( a > b ? a : b == a)
```

Co by się stało? Przede wszystkim instrukcja warunkowa nie zwracałaby poprawnych wartości — np. gdyby obie zmienne były większe od zera, byłaby prawdziwa. Również instrukcja `a == b` dawałaby niepożądane wyniki.

Trudno jest jednak przewidzieć, jakie rezultaty mogą dać bezmyślne podstawienia tekstu wykonywane przez preprocesor. Warto używać sparametryzowanych makrodefinicji, ale na pewno trzeba zachować umiar. Z pewnością dobrym zastosowaniem są takie proste funkcje, jak maksimum dwóch liczb.

ĆWICZENIE

5.5

Napisz program, który zdefinuje makro służące do przydzielania pamięci dla tablicy typu *int* o sparametryzowanej liczbie elementów. Wypisz na ekranie liczbę elementów tablicy po udanym przydziale pamięci.

```
1: /* Przyklad 5.5 */
2: /* Demonstruje zastosowanie sparametryzowanych makrodefinicji */
3: /* w celu prostego, dynamicznego przydzialu pamieci */
4: #include <stdio.h>
5: #define NEWINT(n) ((int *)malloc(sizeof(int)*(n)))
6: int main()
7: {
8:       int *tablica;
9:      if (tablica = NEWINT(10))
10:          printf("Pomyslnie zaalokowano pamiec\n");
11:      else
12:          return -1;
13:      return 0;
14:}
```

W **wierszu 5** zawarta jest sparametryzowana makrodefinicja `NEWINT(n)` definiująca funkcję `malloc` przydzielającą pamięć `n` elementom typu `int`. Wykorzystana zostaje ona w **wierszu 9**, gdzie następuje zamiana ciągu znaków `NEWINT(10)` na `int *)malloc(sizeof(int)*(10))`.

Kompilacja warunkowa

Kompilacja warunkowa to inaczej wybór odpowiednich instrukcji w pliku programu, które faktycznie zostaną poddane procesowi kompilacji. Dzięki preprocesorowi i jego dyrektywom mamy więc możliwość stworzenia elastycznego programu, który zmienia się w zależności

od różnych okoliczności przed kompilacją. Najlepszym przykładem jest tryb debuggowania programu. Debuggowanie to proces testowania programu w poszukiwaniu potencjalnych błędów. W przypadku gdy nie można skorzystać z mechanizmów oferowanych przez różne środowiska programistyczne (z ang. IDE — *Integrated Development Environment*), trzeba polegać na prostych rozwiązaniach — np. wypisywaniu wartości zmiennych za pomocą instrukcji `printf`.

Do przeprowadzenia kompilacji warunkowej można zastosować dyrektywy kompilatora `#ifdef` oraz `#infdef`.

Najlepiej zilustruje to poniższy fragment kodu:

```
#define DEBUG

int main()
{
    .....
    #ifdef DEBUG
        printf("Wartosc zmiennej x: %d\n", x);
    #endif
    .....
}
```

W powyższym przykładzie zdefiniowana została stała symboliczna `DEBUG` — nie musi ona mieć żadnej wartości. Dyrektywę `#ifdef DEBUG` należy odczytać w następujący sposób: „jeśli została zdefiniowana stała symboliczna `DEBUG`, to...". Dyrektywa `#endif` kończy dyrektywę służącą do kompilacji warunkowej. W związku z tym, jeśli zdefiniowana jest stała `DEBUG`, kompilacji poddany zostanie fragment kodu wypisujący na ekranie wartość zmiennej `x`. Należy również zauważyć, że dyrektywy preprocesora można stosować także wewnątrz funkcji `main()` — nie tylko na początku programu.

Uważny Czytelnik zauważy pewnie, że kiepski pożytek z takiej kompilacji warunkowej, skoro za każdym razem i tak trzeba edytować plik programu. Można by tak samo wstawić komentarz przy instrukcji `printf()`, a żeby wyświetlić wartość zmiennych w programie, trzeba by prostu ten komentarz usunąć. Kompilatory języka C pozwalają na ustawienie odpowiedniej opcji poprzez wywołanie kompilacji programu, np. w poniższy sposób:

```
gcc -D DEBUG plik.c
```

Nie trzeba w tym przypadku używać w programie dyrektywy `#define DEBUG`.

Dyrektywę #ifndef stosuje się natomiast najczęściej na samym początku plików nagłówkowych w poniższy sposób:

```
#ifndef MOJ_PLIK_NAGLOWKOWY
#define MOJ_PLIK_NAGLOWKOWY

.....
Zawartosc pliku nagłowkowego
....

#endif
```

W powyższy sposób można uniknąć dwukrotnego dołączenia do programu tego samego pliku nagłówkowego.

Co powinieneś zapamiętać z tego cyklu ćwiczeń?

- Co to są struktury ze wskaźnikami i jak je definiować?
- Jakie są zastosowania struktur ze wskaźnikami?
- Jak utworzyć strukturę typu lista i do czego ona służy?
- Jak usuwać i dodawać elementy listy?
- Co to są wskaźniki do funkcji i jak je definiować?
- Jak utworzyć tablice wskaźników do funkcji?
- Jakie jest zastosowanie wskaźników do funkcji?
- Co to jest obsługa zdarzeń?
- Co to są dyrektywy preprocesora?
- Jakie znasz dyrektywy preprocesora?
- Co to są sparametryzowane makrodefinicje i do czego służą?
- W jakim celu używa się dyrektywy #ifndef?

Ćwiczenia do samodzielnego wykonania

ĆWICZENIE 1.

Rozszerz program z ćwiczenia 5.2, tak aby dodawanie i usuwanie wagonów można było wykonywać poprzez wywołanie oddzielnych funkcji.

ĆWICZENIE 2.

Zmodyfikuj program z ćwiczenia 5.2 tak, aby struktura wagon posiadała dodatkowy wskaźnik na poprzedni wagon w pociągu.

Lista, która powstanie w rezultacie tej modyfikacji, jest nazywana listą dwukierunkową.

ĆWICZENIE 3.

Dodaj obsługę nowego zdarzenia do programu z ćwiczenia 5.3.

Zdefiniuj nowe funkcje do obsługi zdarzeń. Pamiętaj, żeby wywoływać je poprzez wskaźniki do funkcji.

ĆWICZENIE 4.

Utwórz plik nagłówkowy — `naglowkowy.h` — i zdefiniuj w nim dwie stałe — `TRUE` oraz `FALSE` — reprezentujące odpowiednie wartości logiczne.

Pamiętaj o zastosowaniu dyrektyw preprocesora: `#ifndef`, `#define` i `#endif`.

ĆWICZENIE 5.

Napisz sparametryzowaną makrodefinicję obliczającą pierwiastek kwadratowy podanego parametru.

Makrodefinicja powinna być wywoływana np. w ten sposób: `SQRT(4)`, jeśli chciałbyś obliczyć pierwiastek kwadratowy z liczby 4.