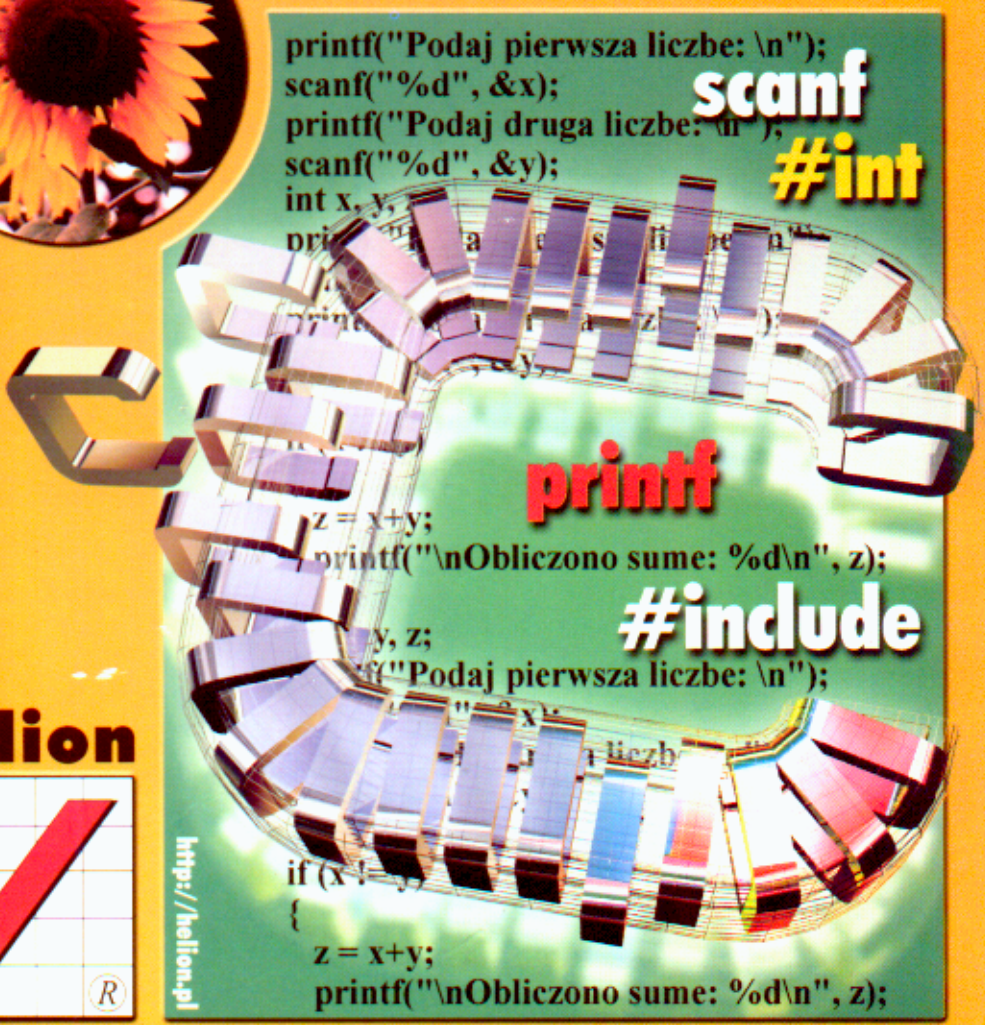
Marek Tłuczek

Programowanie w języku C

Ćwiczenia praktyczne



Helion

Spis treści

Wstęp

Dlaczego C? Takie pytanie stawiają sobie autorzy książek o programowaniu. Ale przecież, jeżeli kupiłeś tę książkę, to pewnie miałeś wyraźny powód. Może jesteś studentem lub uczniem, dla którego nauka C to nudy i tortury umysłowe, a może programowanie to Twoje hobby i chciałeś dobrze poznać ten język. W każdym razie, ja mogę się tylko domyślać. Pisząc tę książkę staram się myśleć o wszystkich czytelnikach. Pragnę zarazić programowaniem w C wszystkich, dla których poznanie tego języka jest przymusem, jak również pomóc pozostałym w rozwijaniu swoich umiejętności programistycznych.

Język C jest używany zarówno przez szerokie grono profesjonalistów, jak i przez początkujących. C nie jest skomplikowanym językiem programowania i posiada wielkie możliwości. Po dobrym jego opanowaniu będziecie mogli napisać nawet najbardziej skomplikowane programy. Nawet Wy – biedni uczniowie, zadręczani przez nauczycieli informatyki programowaniem i związanymi z nim trudnymi obliczeniami matematycznymi, cierpiący na różnorakie depresje z tego powodu – możecie polubić C. Musicie jedynie dać sobie szansę, uzmysłowić, że programowanie nie jest takie straszne. Pozwólcie pomóc sobie!!! Nie spocznę, dopóki nie nauczę was programować w C. Pamiętajcie o tym! Kto wie... może, rzeczywiście, moje metody nauczania pomogą co niektórym poszerzyć horyzonty..., odkryć swoje powołanie, poznać potęgę języka C!!! Wyobraźcie sobie, że namówiłem moją babcię do przeczytania tej książki. O dziwo, zrozumiała większość pojęć i niedługo ma zamiar zająć się programowaniem sieciowym. Widać język C może okazać się ciekawszy nawet od robienia na drutach. Tym optymistycznym akcentem kończę mój manifest i zachęcam do nauki.

Rozdział 1.

Podstawy języka C

C jest językiem programowania o potężnych możliwościach. Jak wspomniałem, nie jest trudny do opanowania i przeznaczony dla wszystkich. C jest elastycznym językiem programowania, tzn. przy użyciu kilku jego funkcji i naszej wyobraźni jesteśmy w stanie stworzyć nawet najbardziej skomplikowane programy.

Dodatkowo, programy napisane w środowisku jednego systemu operacyjnego (np. MS-DOS) mogą być kompilowane i uruchamiane na innych (np. Linux). Jest to możliwe dzięki standardowi ANSI, który jest zbiorem reguł stworzonym dla wszystkich kompilatorów języka C.

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, bo na pewno nie możecie się doczekać napisania swojego pierwszego programu.

Tworzenie programu w C

Tworzenie programu w C można podzielić na 4 etapy:

1. Napisanie kodu źródłowego w dowolnym edytorze tekstu.

   Kod źródłowy to tylko ciąg instrukcji – tekst. W przeciwieństwie do nas, komputer go nie zrozumie. Procesor rozumie tylko instrukcje binarne, których zbiór nazywamy językiem maszynowym. Aby przetłumaczyć instrukcje w trybie tekstowym na odpowiedniki binarne potrzebny nam jest kompilator.

2. 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 pozwoliłem sobie zaprezentować przykłady tworzenia programów przy użyciu kompilatora systemu UNIX, uruchamianego poleceniem `gcc`.

3. Łączenie za pomocą programu „linker".

   Kompilator przekształca jedynie kod źródłowy w pliki obiektowe. Te z kolei muszą być „połączone" za pomocą programu `linker` i dopiero w tym momencie powstaje plik, który możemy uruchomić.

4. Uruchomienie programu.

Ćwiczenie 1.1.

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

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

Proces kompilacji za pomocą `gcc` (system operacyjny UNIX):

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

Uruchamiamy plik:

```
./cw1.1
```

Pozwolę sobie teraz wyjaśnić budowę naszego programu. Wiersze 1 i 2 to komentarz, rozpoczynamy go znakiem `/*`, a kończymy za pomocą `*/`. Kompilator ignoruje tekst znajdujący się pomiędzy tymi specjalnymi znakami.

Dyrektywa `#include` w wierszu 3 powoduje włączenie do programu informacji zawartych w pliku nagłówkowym `stdio.h`. `main()` to funkcja główna naszego 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 nam jeszcze instrukcja `return 0`. Zwraca ona wartość 0 do systemu operacyjnego informując, iż jest to koniec programu.

printf() – funkcja wyjścia

Funkcja `printf()` służy do wyświetlania tekstu na monitorze – urządzeniu wyjścia. Dlatego też możemy stwierdzić, iż `printf()` jest funkcją wyjścia. Jest to podstawowa funkcja biblioteczna języka C, dlatego też zaczniemy naszą naukę programowania 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: /* Wypisuje na ekranie dane osobowe */
3: #include <stdio.h>
4: main()
5: {
6:       printf("Jozek \nMarchewa \nWilcze Dolki 21");
7:       printf("\n45-680 \nKurzy Zdroj \n");
8:       return 0;
9: }
```

Jak widzisz, 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 2 wyrazy oddzielone */
3: /* od siebie dwoma znakami tabulatora */
4: #include <stdio.h>
5: main()
6: {
7:       printf("Kocham \t\tprogramowanie\n");
8:       return 0;
9: }
```

Wykorzystujemy tutaj kolejną sekwencję wyjściową `\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: 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 cudzysłowiu.

Ćwiczenie 1.5.

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

```
1: /* Przykład 1.5 */
2: /* Wypisuje na ekranie tekst oraz wywoluje dźwięk */
3: #include <stdio.h>
4: 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. Wyobraźmy sobie pełen magazyn małych pudełek. Załóżmy, że będą one tworzyły pamięć komputera. Każde z nich to pewne miejsce w tej pamięci. Chcemy zadeklarować zmienną, więc wybieramy jedno pudełko i naklejamy na nim nalepkę (z nazwą zmiennej) oraz wrzucamy do niego jakąś rzecz, którą możemy później swobodnie wymienić na inną (nie zmieniamy nalepki).

Widzimy więc, iż pudełko to zmienna, która posiada swój adres w pamięci oraz nazwę (umieszczoną na nalepce). Możemy przypisywać jej różne wartości (wkładać różne rzeczy do pudełka) bez zmiany nazwy i adresu (nalepka ma pozostać 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). Poniżej przedstawiam podstawowe typy zmiennych:

Integer (`int`)	Liczba stałoprzecinkowa z zakresu –32 768..32767
Char (`char`)	Pojedynczy znak ASCII oraz liczby od –127 do 127
Short integer (`short`)	Liczba stałoprzecinkowa z zakresu –32 768..32767
Long integer (`long`)	Liczba stałoprzecinkowa z zakresu –2 147 483 648... 2 147 483 647
Unsigned integer (`unsigned int`)	Liczba stałoprzecinkowa z zakresu 0..65535
Unsigned short integer (`unsigned short`)	Liczba stałoprzecinkowa z zakresu 0..65535
Unsigned long integer (`unsigned long`)	Liczba stałoprzecinkowa z zakresu 0..4 294 967 295
Single-precision floating point (`float`)	Liczba zmiennoprzecinkowa z zakresu 1.2E–38..3.4E38 (pamiętane 7 cyfr)
Double-precision floating point (`double`)	Liczba zmiennoprzecinkowa z zakresu 2.2E–308..1.8E308 (pamiętane 19 cyfr)

Ćwiczenie 1.6.

Zadeklaruj 5 zmiennych i dobierz dla nich odpowiednie typy. Powinny one zawierać np.:

a) procent miesięcznego zarobku, który wydajemy 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 użyłem typu double, ponieważ wartość będzie tak minimalna, że musimy podać ją z dokładnością przynajmniej 8 miejsc po przecinku (np. 0,00000001%). Dla zmiennej `MojWiek` przypisujemy 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=100000 cm; wartość wykracza poza granicę typu `int`), dlatego też zalecam użycie typu `unsigned long` (dodatkowo, odległość jest zawsze dodatnia). Temperatura może przyjmować wartości zarówno dodatnie, jak i ujemne, dlatego też odpowiedni będzie w tym przypadku typ `int`. Natomiast cena nawozu musi być podana z dokładnością do 2 miejsc po przecinku (dobry rolnik zawsze prowadzi dokładne rozliczenia, np. 12,35 zł – cena aktualna w Pysznicy, wiosce koło Stalowej Woli).

Zastanawiacie się pewnie, jakie nazwy możemy 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 piszemy programy w C, ważny jest również styl (możliwe, że inni ludzi chcieliby zrozumieć, na czym polega działanie Twojego programu; przypomnij sobie swojego nauczyciela matematyki i jego liczne problemy nerwowe spowodowane sprawdzaniem klasówek – skutki tego nie były chyba zbyt przyjemne).

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

Dobra... koniec zanudzania!!! Piszemy programik...

Ćwiczenie 1.7.

Napisz programik, który jeszcze raz deklaruje zmienne z poprzedniego przykładu oraz przypisuje im odpowiednie wartości. A żeby było śmieszniej 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: 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 °C, ",
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: }
```

Teraz należy Ci się krótkie wyjaśnienie. Zmienne deklarujemy na początku programu (przed funkcją `main()`). Wartości, jak widać powyżej, przypisujemy 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, lecz ma jedynie przypisaną taką wartość).

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

Są to tzw. specyfikatory konwersji (z ang. *conversion specifiers*). Nakazują one 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, możemy przypisywać zmiennym wartości podczas ich deklaracji.

Stałe w C

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

Przypomnijmy sobie przykład z pudełkami z poprzedniego rozdziału. Założyliśmy, że umieszczamy w pudełku pewną rzecz, naklejamy na nim nalepkę, na której wypisujemy nazwę zmiennej. W przypadku stałych, po włożeniu pewnej rzeczy do pudełka zaklejamy je taśmą, której już nie wolno nam zerwać (nie możemy zmienić wartości stałej po jednorazowym przypisaniu wartości).

Rozróżniamy 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, z reguły to wartość reprezentowana w programie przez pewien symbol. Stałą tego typu deklarujemy na początku naszego 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
```

Widzimy, że wygląda to, jak deklaracja zmiennej, jednak słowo `const` „zakleja pudełko" (powoduje, iż wartości zmiennej `NaszaLiczba` nie będziemy już mogli zmodyfikować). Pamiętajmy również, 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: main()
8: {
9:       PoleKuli = 4*PI*R*R;
10:      printf("Pole Kuli wynosi %f\n", PoleKuli);
11:      return 0;
12: }
```

Deklarujemy stałą `PI` w wierszu 4 naszego programu przy pomocy `#define`. Następnie deklarujemy kolejno zmienną `PoleKuli` oraz stałą `R` (promień kuli, za pomocą słowa kluczowego `const`). Teraz, w bardzo prosty sposób, możemy obliczyć pole kuli w wierszu 9, od razu przypisując otrzymaną wartość zmiennej `PoleKuli`. W wierszu 10 stosujemy funkcję `printf()`, aby wyświetlić wynik na ekranie.

Ale to proste !!!

Proponuję napisanie podobnych programików, które obliczałyby np. pole walca, stożka, czy też obwód koła. Wiem, nie chce Ci się, ale spróbuj, postaraj się, na pewno pomoże Ci to w zrozumieniu trudniejszych tematów.

Ćwiczenie 1.10.

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

```
1: /* Przyklad 1.10 */
2: /* Oblicza ilosc sekund w ciagu doby */
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: 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, nie ma znaczenia, której z tych deklaracji użyjemy. Można użyć dwa razy `#define`, czy też `const`, nie wpłynie to w żaden sposób na działanie naszego 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 **czyta** 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 nam „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 obrębie cudzysłowu oraz jej nazwę poprzedzoną znakiem `&`. W ten sposób, program w trakcie działania, zapyta nas o wartość, którą następnie przypisze odpowiedniej zmiennej – w naszym 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: main()
6: {
7:       scanf("%d", &Wiek);
8:       printf("Masz %d lat", Wiek);
9:       return 0;
10: }
```

Ale banalne!!! Deklarujesz zmienną, przydzielając jej miejsce w pamięci (wiersz 4). Używasz funkcji `scanf()` (wierz 7), która pobiera wartość z klawiatury i wpisuje ją w podane miejsce, a następnie stosujesz `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, proszę Cię, nie wnikaj w to. Powód jest prosty – na razie nie jest Ci to potrzebne. Nauczysz się więcej o `scanf()` i innych funkcjach wejścia w następnych rozdziałach tej książeczki. A teraz następne zadanka. Ale fajnie..., mogę teraz wymyślać ich więcej. Cieszysz się, prawda?? Widzę, że już pokochałeś C!! Wspaniale!!! Podziękujesz mi później !

Ćwiczenie 1.12.

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

```
1: /* Przyklad 1.12 */
2: /* Oblicza Twoj wiek po uplywie 480-u miesiecy */
3: #include <stdio.h>
4: #define PrzedzialCzasu 480
5: #define ile_mies_w_roku 12
6: 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: }
```

Hmmm.. trochę bardziej skomplikowane??? Należy Wam się dokładne wyjaśnienie tego, co tu się dzieje. Aby zamienić 480 miesięcy na lata, musiałem zadeklarować kolejną stałą – `ile_mies_w_roku` (wiersz 5). Następnie wprowadziłem potrzebne zmienne – `Wiek` (miejsce na wartość wprowadzoną z klawiatury), `ObliczonyWiek` (w celu przechowania wyniku zadania), `IleLat` (tymczasową zmienną zawierającą obliczoną ilość lat, która ma odpowiadać 480 miesiącom). Kolejnym krokiem jest obliczenie zmiennej

IleLat. Dzielimy w tym celu stałą PrzedzialCzasu przez ile_mies_w_roku (używamy operatora / – na ten temat dowiemy się więcej w następnym rozdziale). Następnie korzystamy z funkcji scanf(), aby uzyskać wartość zmiennej Wiek, do której później dodamy IleLat, i otrzymamy końcowy wynik. Ostatecznie wyświetlamy otrzymaną wartość na ekranie przy użyciu printf(). I wszystko jasne...

Instrukcja warunkowa if

Instrukcja warunkowa umożliwia nam kontrolę wykonywania poszczególnych instrukcji w obrębie 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 możemy również wprowadzić polecenie else, które umożliwi wykonanie innego bloku instrukcji (w wypadku, 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 2 liczby oraz decyduje ktora z nich */
3: /* jest wieksza */
4: #include <stdio.h>
5: int x, y;
```

```
6: 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 deklarujemy 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 zauważyliście, iż ćwiczenie zostało rozwiązane niezgodnie z poleceniem, gdyż wartości zmiennych `x` i `y` mogą być sobie równe. Oczywiście zrobiłem to celowo, aby zmusić Was do myślenia. Waszą misją bojową będzie ulepszenie tego programu.

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

Ćwiczenie 1.14.

Napisz program, który dokonuje mnożenia dwóch, wcześniej podanych przez Ciebie liczb, 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 2 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: 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*). W celu ułatwienia Wam lepszego zrozumienia pojęcia *nesting*, słowa kluczowe głównej instrukcji warunkowej zaznaczone są grubą czcionką (wiersze 14 i 19). W wierszu 14 testowany jest omawiany wcześniej przypadek, kiedy wartość zmiennej z jest równa liczbie 100. Następnie, w 21 wierszu 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 (wiesze 23–24). W przeciwnym razie wykonana zostanie funkcja `printf()`, która zawiadomi nas o tym, iż wartość zmiennej z jest mniejsza od liczby 100 (wiersze 28–29).

W instrukcji, w 14 wierszu zastosowany został nieznany Wam dotąd operator ==. W języku C oznacza on *równość*. Musicie pamiętać, iż znak = jest operatorem służącym do przypisywania wartość 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: `&&`) – suma dwóch wyrażeń,

OR (symbol: `||`) – iloraz dwóch wyrażeń,

NOT (symbol: `!`) – zaprzeczenie wyrażenia.

Ćwiczenie 1.15.

Napisz program, który wykona dodawanie dwóch wpisanych z klawiatury liczb, 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, x;
7: 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 widzicie, problem został rozwiązany poprzez użycie operatora zaprzeczenia w 9 wierszu. 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 3 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 w przypadku, 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 3 zmiennych */
3: /* w zaleznosci od spelnienia danych warunkow */
4: #include <stdio.h>
5: int a, b, c, d;
6: 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", 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", 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", d);
31:     }
32:     return 0;
33: }
```

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

Poznaliście już operatory logiczne i relacje. Teraz nadeszła pora na zapoznanie Was z kolejną ich grupą – operatorami przypisania. Domyśliliście się pewnie, iż 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 tworzymy programy w C, co to jest kod źródłowy, kompilator?
- Do czego służy funkcja `printf()`, jaka jest jej konstrukcja?
- Co to są sekwencje wyjściowe?
- Co to jest zmienna i typ zmiennej, jak deklarujemy 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 rozwiązania

Ćwiczenie 1.

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

Ćwiczenie 2.

Zdefiniuj stałą symboliczną WIEK o wartości 20 przy użyciu dwóch metod, które poznałeś 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 podaną wartość temperatury w stopniach Celsjusza na wartość w stopniach Fahrenheita.

100 stopni Celsjusza = 212 stopni Fahrenheita.

Ćwiczenie 7.

Napisz program, który pobierze z klawiatury 5 liczb, a następnie wypisze na ekranie największą z nich.

Rozdział 2.
Programowanie strukturalne

W poprzednim rozdziale zapoznaliście się z podstawowymi instrukcjami języka C. Po opanowaniu dotychczas zaprezentowanego materiału jesteście już w stanie napisać proste programy. Jednak wciąż nie zdajecie sobie sprawy, jakie możliwości daje język C.

W tym rozdziale zapoznacie się z funkcjami – podstawowym elementem każdego programu. Poznacie również różne rodzaje pętli oraz struktury. Po zapoznaniu się z wszystkimi tymi pojęciami oraz po wykonaniu załączonych ćwiczeń zdobędziecie wiedzę, dzięki której będziecie mogli napisać bardziej skomplikowane programy i z pewnością docenicie 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 zyskujemy wiele czasu, program jest lepiej zorganizowany, i w związku z tym łatwiej jest go później odczytać i zrozumieć jego działanie. Co więcej, w tego typu programach łatwiej można znaleźć ewentualne błędy, testując indywidualnie każdą z funkcji. Programowanie strukturalne jest wykorzystywane w programach, wykonujących zadanie wykonywać bardziej skomplikowane zadania. Dzięki podziałowi programu na mniejsze, niezależne podprogramiki, łatwiejsze jest zaplanowanie jego konstrukcji i działania.

Funkcje

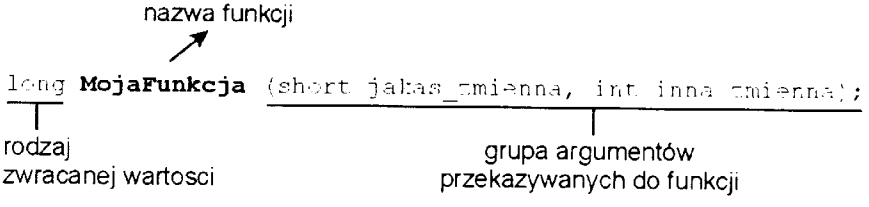
Czym jest funkcja? Funkcja jest niezależnym podprogramem, wykonującym pewne zadanie, lub zadania dla potrzeb programu głównego. Dla każdej funkcji w programie muszą być spełnione poniższe warunki:

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 instrukcję powrotu (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 widzimy, iż argumenty to nic innego, jak definicje zmiennych zawarte w nawiasie. Jeżeli funkcja zwraca jakąś wartość, musimy określić typ zwracanej wartości. W tym przypadku jest to typ `int`.

- definicja funkcji:

```
long MojaFunkcja (short zmienna1, int zmienna2)
{
     long ilocz /* deklaracja zmiennej, która będzie zawierać
                                    zwracaną wartość */
     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 naszą 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**.

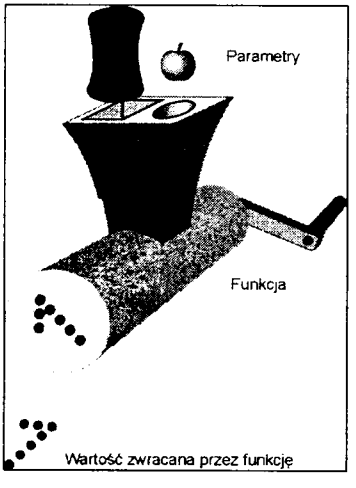
- wywołanie funkcji:

```
long x;
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 musimy 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 pow. prz. zmiennej `x`).

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

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



Jak ma wyglądać prototyp i nagłówek definicji takiej funkcji?

Otóż w polu, w którym umieszczany jest typ zwracanej wartości wpisujemy `void`.

Przykład:

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

Zauważmy, że przykładem takiej funkcji jest `main()` – nie zwraca ona żadnej wartości, ani nie pobiera jakichkolwiek argumentów.

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: 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 widzicie, program ten jest trochę bardziej skomplikowany niż poprzednie. W wierszu 4 umieszczamy prototyp naszej 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 musimy zadeklarować dwie dodatkowe zmienne globalne tego samego typu (`liczba1` oraz `liczba2`), które przekażemy jako argumenty dla naszej funkcji. W wierszu 4 deklarujemy zmienną globalną, która będzie rozpoznawalna w programie głównym i która będzie ostatecznie przechowywać wynik operacji mnożenia (zmienna `wynik`). Zauważcie, iż musi ona być tego samego typu co wartość zwracana przez funkcję. Wiersze 8–10 – wywołanie `main()` oraz pobranie dwóch liczb. W wierszu 11 wywołujemy funkcję `MojaFunkcja`. W tym momencie egzekucja programu zostaje przerwana i wykonywane są wszystkie instrukcje naszej funkcji – wiersze 15–20. W tym momencie zmienne `liczba1` i `liczba2` „przekształcane" są w zmienne lokalne `x` i `y`. W wierszu 17 deklarujemy zmienną lokalną `z` typu `long` - będzie ona użyta w celu „tymczasowego" przechowania wartości zwracanej (iloczynu `x` i `y`). Ostatecznie, po zakończeniu wykonywania instrukcji funkcji `MojaFunkcja`, egzekucja programu zostaje wznowiona w wierszu 11. W tym momencie wartość zwracana zostaje przypisana zmiennej globalnej `wynik`. Pozostaje jeszcze tylko wypisanie rezultatu na ekranie monitora (funkcja `printf()` – wiersz 12).

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 odpowiednio do tego celu skonstruowanej przez was funkcji).*

```
1: /* Przyklad 2.2 */
2: /* Przyklad funkcji, ktora nie zwraca wartosci */
3: #include <stdio.h>
4: void funkcja(int x);
5: int liczba;
6: 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 macie 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 stałych literalnych, zapisze wynik w dowolnej zmiennej, a następnie wyświetli odpowiednią informację na ekranie (wszystkie działania muszą być wykonane w odpowiednio skonstruowanej do tego celu funkcji).

```
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: 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 Was z przykładem funkcji, która ani nie pobiera żadnych argumentów z programu głównego, ani nie zwraca żadnej wartości.

Ćwiczenie 2.4.

Napisz program, który pobierze 3 wartości, przypisze je pewnym zmiennym, które 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 wypadku 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: 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 Wam 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 przypisywany jest zmiennej x (modyfikując jej dotychczasową wartość).

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

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

Pętle w języku C

Wstęp do tablic

Zanim rozpocznę opisywanie działania pętli, powinienem zapoznać Was 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 możemy przypisywać różne wartości. Tablice stosujemy w przypadku, gdy chcemy przechowywać w jednym miejscu grupę zmiennych tego samego typu – np. wiek wszystkich osób w naszej klasie (każdy uczeń ma przydzielony numer w dzienniku). Każda tablica musi być zadeklarowana, np.:

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

W powyższym przykładzie zadeklarowaliśmy tablicę złożoną z 30 elementów (**indeks pierwszego elementu tablicy jest równy 0**), każdemu z nich możemy przypisywać wartości typu int, np.:

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

Możemy również zadeklarować tablicę bez określania jej rozmiaru, wtedy będziemy mogli przypisywać wartości tylu elementom, ile tylko będziemy potrzebować w naszym programie. Np.

```
int tablica[];
tablica[345] = 10;
```

Ale zastanówcie się, w jaki sposób można szybko przypisywać różne wartości wszystkim elementom danej tablicy?

Oczywiście, najłatwiejszym i najpopularniejszym sposobem jest użycie pętli *for*.

[1] Wartość zmiennej c pozostaje oczywiście taka sama pomimo, 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).

Pętla for

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

Konstrukcja pętli *for*:

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

for (n=0; n<31; n++)
      {
          instrukcja 1;
          instrukcja 2;
          ............;
          instrukcja n;
      }
```

Pętlę wywołujemy przy użyciu słowa kluczowego `for`, po którym określamy mechanizm działania naszej pętli. Zmienna `n` jest tzw. licznikiem, służy ona do określania liczby powtórzeń danego ciągu instrukcji. Pisząc programy w C musimy pamiętać, iż przed użyciem pętli *for* należy zadeklarować taką zmienną.

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

Za pomocą drugiego wyrażenia określamy warunek (w naszym przypadku używamy wyrażenia z operatorem relacji, określamy maksymalną wartość licznika). Jeżeli dany warunek zostanie spełniony, pętla będzie kontynuowała swoje działanie. W przeciwnym razie pętla ulegnie zakończeniu (kolejne powtórzenia nie będą już miały miejsca).

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;
      }
```

Zastanawiacie się pewnie, kiedy stosować pętlę *for*, a kiedy *while*. Pętli *for* używamy w przypadkach, gdy musimy określić liczbę powtórzeń danego bloku instrukcji. Natomiast pętli *while* używamy w przypadku, gdy należy określić jedynie warunek. Dopóki będzie on spełniany, dopóty odpowiednie instrukcje będą powtarzane.

Pętla do...while

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

W wierszu 4 zadeklarowana jest 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 jest przerywana i wykonywane są 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[39];
6: int licznik;
7: main()
8: {
```

```
9:      for (licznik=0; licznik < 39; licznik++)
10:         tablica[licznik] = 0;
11:     printf("Tablica zostala wyzerowana\n");
12:     return 0;
13: }
```

Jak widać, rozwiązanie problemu jest banalne. Deklarujemy tablicę złożoną z 40 elementów (pamiętajmy, iż indeks pierwszego elementu każdej tabeli jest równy 0), a następnie wywołujemy 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 (stuelementów).

```
1: /* Przyklad 2.8 */
2: /* Przyklad petli for z wykorzystaniem tablic */
3: #include <stdio.h>
4: int tablica[100];
5: int n;
6: 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 deklarujemy tablicę stuelementową. W wierszu 8 wywołujemy pętlę *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: 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 i na pewno niektórzy z Was mają problemy ze zrozumieniem działania powyższego programu. W wierszu 10 deklarujemy tablicę, w której umieszczone zostaną wartości średniej ocen każdego ucznia naszej klasy. W wierszu 5 deklarujemy 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 przypisujemy wartość 0 zmiennej `suma_sr`. Następnie wywołujemy pętlę *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: /* Przykład 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: 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: /* Przykład 2.10 */
2: /* Przykład uzycia petli while */
3: #include <stdio.h>
4: int licznik;
5: 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 na powyższym przykładzie, w przypadku pętli *while* musimy sami określić wstępną wartość licznika (wiersz 7) oraz jego zmianę (wiersz 11). Pamiętamy, iż korzystając z pętli *for* grupujemy 3 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: /* Przykład 2.11 */
2: /* Przykład petli nieskonczonej z wykorzystaniem */
3: /* instrukcji break */
4: #include <stdio.h>
5: unsigned int liczba;
6: unsigned long suma;
7: 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, ćwiczenie to jest trochę bardziej skomplikowane. (Abyście zrozumieli działanie powyższego programu, muszę zapoznać Was z dwoma pojęciami: pętla nieskończona, nagłe przerwanie pętli).

Pętla nieskończona jest pętlą, która 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 na powyższym przykładzie (wiersz 9). Jak widzicie, 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 możemy wywołać w następujący sposób:

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

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

Co powinieneś zapamiętać z tego cyklu ćwiczeń?

- Co to jest funkcja?
- Jak wygląda prototyp i definicja funkcji oraz jak ją wywołujemy?
- 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 deklarujemy oraz jak przypisujemy im wartości?
- Jaka jest konstrukcja pętli *for* oraz do jakich celów możemy 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 możemy ją wywołać?
- Do czego służy instrukcja `break`?

Ćwiczenia do samodzielnego rozwiązania

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 s2.2 tak, aby wykorzystana została pętla while zamiast pętli for.

Rozdział 3.
Język C dla wtajemniczonych

Jeżeli dobrze opanowaliście cały materiał przedstawiony w poprzednich rozdziałach książki, możecie nazwać siebie „wtajemniczonymi programistami" i zapoznać się z kolejnymi zagadnieniami. W tym rozdziale nauczycie się m.in. czym są wskaźniki, tablice wielowymiarowe, łańcuchy znaków i struktury. Pojęcia te są troszeczkę trudniejsze i tym samym trudniej jest dobrze je wytłumaczyć. Dlatego też, przed rozpoczęciem czytania, 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 kofeina „wpłynęła" już do waszej krwi, zapraszam do wykonania następnych kroków w niesamowitym świecie języka C.

Tablice wielowymiarowe

Z pojęciem tablicy zapoznaliście się już w jednym z poprzednich rozdziałów (patrz: Pętla *for*). Tablice, które omawialiśmy do tej pory posiadały tylko jeden wymiar – jeden indeks (numer w nawiasach) Pamiętajmy, że indeksy numerowane są 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 naszej szkole są 24 klasy, a w każdej z nich 36 uczniów.

Rozmiar tablicy to 24×36 = 864 (liczba uczniów w szkole – 864). Jeśli chcemy przypisać wartość jednej z komórek, np. średnią ocen 1. ucznia z 1. klasy w tablicy, musimy 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;
```

25 uczeń 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: /* Przykład 3.1 */
2: /* Przykład uzycia tabeli wielowymiarowej */
3: #include <stdio.h>
4: float MojaSzkola[4][5];
5: int licz1, licz2;
6: 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: }
```

Niektórzy z Was zapewne zastanawiają się jak to się dzieje, że wartości są przypisywane odpowiednim komórkom w tablicy. Postaram się ułatwić Wam zrozumienie tego ćwiczenia, opisując wszystkie czynności krok po kroku.

Wiersz 4: deklarujemy wielowymiarową tablicę dwudziesto elementową typu *float*.

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

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

Wiersz 9: wywołujemy 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 wyświetlane są na ekranie.

Prześledźmy poszczególne kroki pętli:

Kiedy `licz1 = 0` wywoływana jest po raz pierwszy podrzędna pętla *for*. Dla `licz1 = 0` podrzędna pętla powtarzana jest 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*.

Wyświetlanie 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: 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 == 0)
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 deklarujemy dwudziestoczteroelementową tablicę wielowymiarową `tablica[]` i jednocześnie przypisujemy jej odpowiednie wartości. Nasza tablica przechowuje liczbę dni w każdym miesiącu, zarówno roku przestępnego, jak i nieprzestępnego. Następnie, w wierszu 12 wywołujemy pętlę *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 nam wypisanie odpowiednich wartości na ekranie (sprawdza ona, czy dany rok jest przestępny, czy też nie).

Wskaźniki

Każda zmienna zadeklarowana w programie posiada swój adres w pamięci komputera. Jeżeli znamy adres danej zmiennej, możemy zadeklarować drugą zmienną, która będzie zawierać adres pierwszej. W takim przypadku druga zmienna wie, gdzie jest umiejscowiona w pamięci pierwsza zmienna. Możemy stwierdzić, iż druga zmienna wskazuje na pierwszą – jest wskaźnikiem. Może to być dla Was trochę skomplikowane, dlatego też przedstawię przykład, który ułatwi zrozumienie tego pojęcia.

Janek, Zbychu i Ola chodzą do tej samej szkoły w mieście Jakaśdziura. Janek i Zbychu są dobrymi kumplami. Zbychu 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 przyszedł 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.*

Podsumowując: Zbychu jest wskaźnikiem do Oli. Zna on jej imię oraz wie, gdzie mieszka (zna jej adres).

Wskaźnik do zmiennej musimy zadeklarować. Robimy to w następujący sposób:

```
int *wskaznik;
```

Jak widać, dodajemy jedynie gwiazdkę przed nazwą zmiennej.

Do danego wskaźnika musimy dodatkowo przypisać odpowiedni adres zmiennej, na którą ma on wskazywać. Adres zmiennej uzyskujemy stosując znak &. Przypisanie adresu zmiennej do wskaźnika wykonujemy w następujący sposób:

```
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 *Zbychu;
6: main()
7: {
8:      Ola = 180;
9:      Zbychu = &Ola;
10:     printf("Wzrost Oli: %d \n", Ola);
11:     printf("Wzrost Oli wg. Zbycha: %d \n", *Zbychu);
12:     printf("Adres Oli: %d \n", &Ola);
13:     printf("Adres Oli wg.Zbycha: %d "\n, Zbychu);
14:     return 0;
15: }
```

Widzimy, że zmienna `*Zbychu` zawiera wartość zmiennej `Ola`, natomiast zmienna `Zbychu` zawiera adres `Oli`.

Tabela 3.1. *Rozmiary typów zmiennych*

typ zmiennej	rozmiar
`Char`	1 bajt
`Int`	2 bajty
`Short`	2 bajty
`Long`	4 bajty
`Unsigned char`	1 bajt
`Unsigned int`	2 bajty
`Unsigned short`	2 bajty
`Unsigned long`	4 bajty
`Float`	4 bajty
`Double`	8 bajtów

Podsumowując:

`*Zbychu` oraz `Ola` zawierają wartość przypisaną zmiennej `Ola`.

`Zbychu` oraz `&Ola` zawierają adres zmiennej `Ola`.

Wskaźniki i tablice

Wskaźniki są bardzo przydatne, gdy używamy ich w połączeniu z tablicami. Na razie nie zdajecie sobie z tego sprawy, ale z pewnością, posiadając większe doświadczenie w programowaniu, dojdziecie do wniosku, iż wskaźniki są często nieodłączną częścią każdego dobrego programu.

Zapamiętajcie, iż nazwa zadeklarowanej tablicy użyta bez nawiasów jest wskaźnikiem do tej tablicy:

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

Widzimy więc, iż, 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 wpisze na ekranie wartość zawartą w pierwszym elemencie tabeli (wcześniej zadeklarowanej przez Ciebie) – użyj wskaźnika.

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

W wierszu 5 deklarujemy dwudziestoelementową tablicę `Moja_Tablica[20]`. W wierszu 6-tym deklarujemy wskaźnik `wskaznik`. Wiersze 9–10: przypisujemy pierwszemu elementowi tablicy wartość 5 oraz przypisujemy adres tego elementu naszemu wskaźnikowi.

Wskaźniki są niezbędne w przypadku, gdy musimy przekazać tablicę jako argument do funkcji. Pamiętajmy, iż **cała** tablica nie może być argumentem danej funkcji, ponieważ nie jest ona traktowana jako jedna zmienna. Więc w jaki sposób funkcja może mieć dostęp do wszystkich elementów tablicy?

Możemy przekazać jako argument wskaźnik do pierwszego elementu tablicy. W takim przypadku, funkcja, znając adres pierwszego elementu, będzie miała dostęp do pozostałych. Ale co, jeśli funkcja musi znać rozmiar danej tablicy?

Wtedy będziemy musieli przekazać do naszej funkcji wielkości danej tablicy jako drugi argument.

Ćwiczenie 3.5.

Napisz program, który obliczy największą wartość zawartą w którymś z elementów wcześniej zadeklarowanej przez Ciebie tablicy. 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: 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 napisaliśmy. Postaram się teraz opisać krok po kroku wszystkie czynności, jakie należało wykonać, aby rozwiązać to ćwiczenie.

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

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

Wiersz 7: deklarujemy zmienną `licznik`.

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

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

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

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

Wiersz 22: deklarujemy lokalną zmienną `licznik`.

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

Wiersz 24–28: wywołujemy 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 zostaje warunek i jeśli kolejny element jest 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: /* Przyklad 3.6 */
2: /* Przyklad demonstruje przekzywanie 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: 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;
31:       for ( licz = 0; licz < z; licz ++)
32:           suma += x[licz] + y[licz];
33: return suma;
34: }
```

Ćwiczenie to jest bardzo podobne do poprzedniego. Istotną różnicą jest to, iż tym razem przekazujemy do funkcji dwie tablice oraz wspólny rozmiar (tablice mają ten sam rozmiar, w przeciwnym wypadku kod zajmowałby o koło 40 linijek). Jeżeli zrozumieliście, na czym polegało rozwiązanie poprzedniego ćwiczenia, zrozumienie powyższego nie powinno sprawić wam żadnych problemów.

Znaki oraz łańcuchy znaków

Dotychczas w zmiennych umieszczaliśmy jedynie liczby. Nadszedł czas, abyście nauczyli się, jak można przechowywać w nich znaki. W rozdziale tym zapoznacie się również z pojęciem łańcuchów znaków oraz nauczycie się nimi manipulować.

Znaki

Typem zmiennych, który jest używanym do przechowywania pojedynczych znaków jest typ `char`. Zmienną taką deklarujemy w identyczny sposób, jak pozostałe. Natomiast przypisywanie wartości (znaku) odbywa się w nieco odmienny sposób. Przykład:

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

W naszym przykładzie widzimy, że znak a musi być zamknięty w obrębie znaków „cytowania". W taki właśnie sposób są przypisywane 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 254
5: unsigned char licznik;
6: main()
7: {
8:       for (licznik =0; licznik < MAX; licznik++)
9:                 printf("Kod ASCII: %d /t Znak: %c",
10:                licznik, licznik);
11: return 0;
12: }
```

Hmm... dawno nie napisaliśmy tak prostego programiku. Wyjaśnienia wymaga tylko użyty typ zmiennej. Otóż nie jest to `char` ale `unsigned char`. Zmienne typu `char` mogą przyjmować wartości tylko z zakresu od –127...127. Typ `unsigned char` może natomiast przyjmować tylko wartości dodatnie z zakresu od 0...255, więc jest idealny dla potrzeb naszego programu. Musimy 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

Zmienne typu `char` mogą przechowywać, niestety, tylko po jednym znaku. Zastanawiacie się, jak możemy przechowywać całe zdania w naszych zmiennych. 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:

Jeżeli chcemy przechować wyraz „programowanie" jako łańcuch znaków, musimy 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 widzicie, 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' };
```

Jeszcze lepszym sposobem jest przypisanie tablicy całego wyrazu zawartego w cudzysłowie:

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

Lub jeszcze lepiej:

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

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

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

Istnieje również możliwość użycia łańcuchów znaków nie korzystając z tablic. Z czego mamy skorzystać?

Oczywiście skorzystamy ze wskaźników. Mianowicie, deklarujemy wskaźnik i jednocześnie przypisujemy 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. Możemy również zadeklarować wskaźnik, któremu nie przypiszemy żadnego łańcucha znaków, ale chcemy, aby w dalszej części programu wskazywał na jakieś zdanie (np. wprowadzone z klawiatury). Zauważmy, że wskaźnik taki nie ma na co wskazywać, jeżeli nie przypisano mu jakiegokolwiek adresu. Załóżmy, że chcemy, aby później wskazywał na jakieś zdanie. W takim przypadku, musimy udostępnić mu odpowiednio duży blok adresów w pamięci komputera. W przeciwnym razie, nie byłoby miejsca dla przypisywanego zdania w późniejszej części programu.

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

1. Przekazujemy funkcji `malloc()` liczbę potrzebnych bajtów pamięci jako argument.
2. `malloc()` znajduje i rezerwuje dla nas blok pamięci o określonym rozmiarze oraz zwraca adres pierwszego, zarezerwowanego bajtu.

Przykład z życia:

Jest piątek. Józek I Zbychu idą na imprezę. Wybierają się do Rycha ponieważ wiedzą, iż tamten pędził bimber. Przed wyjściem zastanawiają się, czy wystarczą im dwa małe plecaki. Nie wiedzą, ile bimbru Rychu wyprodukował i jakim dobrym kolegą się okaże oraz jaką ilość butelek im odstąpi. Zbychu proponuje aby zadzwonili do Romka, który może im pożyczyć samochód. Wtedy będą mogli przetransportować więcej bimbru. Zbychu dzwoni do Romka:

– *Hej Romek, pożyczysz nam furę? Musimy jakoś zabrać bimber od Rycha, a nie wiem ile tego wyprodukował, plecaki nam chyba nie wystarczą?*

– *Spoko stary, powiedz tylko, ile maksymalnie może tego być?*

– *No, około 50 butelek.*

– *Super, to w takim razie wystarczy wam maluszek. Pogadam ze starym, może pożyczy, moment zapytam go... (...) ojciec się zgodził, pożyczę wam.*

– *Wielkie dzięki Romek!*

Na podstawie naszego przykładu spróbuję Wam przybliżyć działanie funkcji `malloc()`. Potraktujmy Janka i Zbycha jako jeden wskaźnik. Nie wiedzą ile bimbru Rychu mógł wyprodukować, więc nie przypisujemy wskaźnikowi żadnego adresu. Romek jest

naszą funkcją `malloc()`. Zbychu zwraca się do niego, aby udostępnił miejsce na bimber – pamięć. Romek pyta, ile będzie butelek bimbru – ile bajtów pamięci musi udostępnić. Zbychu 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. Typem zwracanej wartości jest `void`, ponieważ jest on kompatybilny z wszystkimi innymi typami zmiennych. Dzięki zastosowaniu `void`, jesteśmy w stanie 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 znaków. 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 uzycie funkcji malloc() */
3: #include <stdlib.h>
4: #include <stdio.h>
5: 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 zauważyliście już, że funkcja `malloc()` wymaga dodania do programu kolejnego pliku załącznika – `stdlib.h`. W wierszu 7 deklarujemy wskaźnik `lancuch`, któremu nie przydzielamy adresu. W wierszu 8 wywołujemy funkcję `malloc()`, przypisując wartość zwróconą przez nią naszemu wskaźnikowi oraz sprawdzamy, czy w ogóle udało się przydzielić wymaganą ilość pamięci (`NULL` oznacza 0, czyli fałsz w języku C). Jeżeli operacja przebiegła nie pomyślnie, 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: 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 wam dotąd 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). Zaznaczamy również, 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: 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: main()
8: {
9:       for (licznik = 0; licznik < 10; licznik++)
10:           printf( "%s\n", zdanie);
11:      return 0;
12: }
```

Jeżeli chcemy wyświetlić łańcuch znaków na ekranie, używając funkcji `printf()`, musimy użyć odpowiedniego *specyfikatora konwersji*, którym jest `%s`.

Ćwiczenie 3.12.

Napisz program, który tym razem 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: 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 powyżej, 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 chcemy umieścić dany ciąg znaków (w naszym przypadku `bufor` – czyli wskaźnik do tablicy `bufor[100]`). Po wywołaniu funckji `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: 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. Zauważmy, iż 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. Wiemy też, że wskaźnik zawiera również adres, na który wskazuje. Dlatego też, w powyższym przykładzie nie używamy znaku & (patrz: wiersz 10), tylko korzystamy z nazwy tablicy, czyli wskaźnika.

Ćwiczenie 3.14.

Napisz program, wywołujący funkcję, która, z kolei, 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: int *funkcja(char x[], char y[]);
7: 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: int *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 a;
24:       else
25:          return b;
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, iż zwracany jest wskaźnik (w naszym przykładzie – wskaźnik do dłuższego łańcucha znaków). `strlen()` jest funkcją obliczającą ilość znaków w łańcuchu. Argumentem jest oczywiście wskaźnik do odpowiedniego bufora (wiersz 20 i 21).

Struktury w języku C

Struktury, podobnie jak tablice, są grupą zmiennych zebranych razem pod wspólną nazwą. W strukturach ponadto możemy 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 rozpoczynamy od słowa kluczowego `struct`, po którym występuje nazwa struktury. Następnie, w obrębie klamr wymieniamy zmienne, które mają wchodzić w skład danej struktury. Zmienne są nazywane *członkami struktury*. Struktura musi być również zadeklarowana – należy utworzyć jej tzw. *przykłady*.

Przykład:

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

Powyższe instrukcje definiują strukturę `wiek_kotka` oraz tworzą jej dwa przykłady: `koci_wiek` i `ludzki_wiek`. Zarówno pierwszy, jak i drugi przykład 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. Ale pamiętaj – dopóki nie zostaną zadeklarowane przykłady danej struktury, dopóty nie można ich używać (miejsce w pamięci komputera zostaje przydzielone dopiero po zadeklarowaniu przykładów 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;
```

Widzimy więc, iż odwołujemy się do poszczególnych członków struktury za pomocą „kropki" oddzielającej nazwę przykładu od nazwy członka. Możemy 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 3.15.

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

```
1: /* Przyklad 3.15 */
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: 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 definiujemy strukturę `wiek_kota` złożoną z trzech elementów oraz deklarujemy jej dwa przykłady – `koci_wiek` i `ludzki_wiek`. W wierszach 11–16 przypisujemy wartości poszczególnym członom, a w pozostałej części programu wyświetlamy wartości wszystkich elementów na ekranie.

Ćwiczenie 3.16.

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

```
1: /* Przyklad 3.16 */
2: /* Przyklad demonstruje uzycie struktur, */
3: /* ktorych czlonkami sa tablice */
4: #include <stdio.h>
5: struct data {
6:     char dzien[20];
7:     char miesiac[20];
8:     int rok;
9: } d_data;
10: main()
11: {
12:       d_data.dzien[20] = "poniedzialek ";
13:       d_data.miesiac[20] = "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 wykorzystałem tablice (łańcuchy znaków) jako elementy struktury. Jak widać nie jest to skomplikowane. Musicie pamiętać, iż podczas przypisywania wartości należy podać liczbę elementów tablicy, do której się odwołujecie (wiersze 12 i 13).

Ćwiczenie 3.17.

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

```
1: /* Przyklad 3.17 */
2: /* Przyklad demonstruje uzycie struktur, */
3: /* ktorych czlonkami 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: 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 deklarujemy wewnątrz definicji głównej struktury (wiersze: 18, 19). Pamiętajmy, 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). Zwróćcie uwagę na funkcję `scanf()` – jeśli „zmienna wejściowa" jest typu „łańcuchowego", nie piszemy znaku `&` (nazwa tablicy jest wskaźnikiem; patrz: np. wiersze: 24, 27, 29); musimy 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 deklarujemy?
- Jak uzyskujemy adres zmiennej?
- Jakie wartości mogą przechowywać wskaźniki?
- Jak przypisujemy wartości wskaźnikom?
- Jakimi sposobami możemy 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?
- Co to są struktury, jak je definiujemy, a jak deklarujemy?
- Jakiego typu zmienne mogą być przechowywane w strukturach?
- Jakie dane, oprócz zmiennych, możemy przechowywać w strukturach?
- Jak odwołujemy się do poszczególnych członów struktury?
- Jak tworzymy struktury, których elementami są inne struktury?
- Jak odwołujemy się do struktur będących elementami struktury?

Ćwiczenia do samodzielnego rozwiązania

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.

Rozdział 4.

Język C dla guru

Drogi czytelniku!!! Czyżbyś przerobił cały materiał z poprzednich części książki? Rozwiązałeś wszystkie ćwiczenia? Nie masz żadnych wątpliwości? Jesteś pewien, że nie masz żadnych wątpliwości? Hmm..., w takim razie możesz przekroczyć kolejne wrota fascynującej krainy języka C i zatopić się w bezmiernej głębinie wiedzy. Pamiętaj – stąd już nie ma powrotu. Jeśli opanujesz cały materiał zawarty w tej książce, z pewnością sięgniesz po książki 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. Gotowy? Czy już naładowałeś swoją mózgownicę odpowiednimi używkami? Jeśli tak, to zapraszam do rozdziału 4. Poznasz tu takie pojęcia, jak: programowanie strumieni wejścia–wyjścia, wskaźniki do wskaźników oraz łańcuchy znaków i struktury dla zaawansowanych.

Strumienie wejścia–wyjścia

Każdy program w C musi wykonywać operacje na strumieniach wejścia–wyjścia. Poznaliśmy już podstawowe funkcje wejścia–wyjścia: `printf()`, `puts()` – funkcje wyjścia oraz `scanf()` i `gets()` – funkcje 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 też 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: strumienie 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. Są one używane 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 każdego.

Funkcje wejścia

W tym rozdziale omówimy tylko te funkcje które pobierają dane ze standardowego wejścia (stdin). Dzielą się ona na:

- funkcje pobierające pojedyncze znaki z klawiatury, np. `getchar();`
- funkcje, które pobierają z klawiatury całe wiersze, np. `gets();`
- funkcje, które pozwalają na manipulację strumieniami wejścia – `scanf()`.

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

```
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: main()
6: {
7:       znak = getchar();
8:       printf("Wpisales znak: ");
```

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

Zauważmy, 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 używamy 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()`. Zapoznaliście 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 znana Wam 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 łańuchem formatowania. Przypomnijcie sobie, iż znak `&` przed zmienną jest tzw. „*operatorem adresu*" i umieszczony przed zmienną powoduje otrzymanie jej adresu. Widzimy, iż łańcuch formatowania składa się ze specyfikatorów konwersji. Każdemu z nich musi odpowiadać adres zmiennej. Każdy specyfikator konwersji składa się z następujących znaków:

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

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, oktetowej lub szesnastkowej.
o	`int *`	Wartość typu `integer` w postaci oktetowej.
u	`unsigned int *`	Dodatnia wartość typu `integer` w postaci dziesiętnej.

Tabela 4.1. c.d. *Specyfikatory typu*

Typ	Argument	Znaczenie specyfikatora typu
x	`int *`	Wartość typu `integer` w postaci heksadecy-malnej.
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, iż argument jest wskaźnikiem do typu `short`.
l	Znak ten umieszczony przed specyfikatorami typu `d`, `i`, `o`, `u` lub `x`, informuje funkcję o tym, iż argument jest wskaźnikiem do typu `long`.
L	Znak ten umieszczony przed specyfikatorami typu `e`, `f` lub `g`, informuje funkcję o tym, iż 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: /* Przyklad 4.2 */
2: /* Przyklad demonstruje uzycie specyfikatorow */
3: /* typu w funkcji scanf() */
4: #include <stdio.h>
5: int liczba1, liczba2;
6: char napis[30];
7: 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 funkcji scanf() */
4: #include <stdio.h>
5: #define rozmiar 30
6: char napis[rozmiar];
7: long liczba;
8: 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, iż argument nie jest wskaźnikiem do typu `integer` (`%d`), ale do typu `long`.

Funkcje wyjścia

W tym rozdziale omówimy jedynie dwie funkcje: `putchar()` oraz `puts()`.

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

Prototyp funkcji `putchar()`:

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

Zauważmy, iż zarówno argument funkcji, jak i wartość zwracana są typu `int`. Na pewno domyśliliście się, iż są 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 funkcji putchar() */
3: #include <stdio.h>
4: 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 użyliśmy pętli *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()`, z która zapoznaliście się już 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 chcemy wyświetlić 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 poznaliście już w poprzednich rozdziałach tej książki. Nauczyliście się jak je deklarować, jak przydzielać im pamięć (`malloc()`) oraz jak wyświetlać je na ekranie. W tym rozdziale pragnę przedstawić Wam zaawansowane operacje na łańcuchach. Nauczycie się, jak je kopiować oraz łączyć ze sobą.

Kopiowanie łańcuchów znaków

W książce tej postanowiłem opisać 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( *przeznaczenie, *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.

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: 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 jest trochę skomplikowane, postanowiłem więc opisać je krok po kroku.

Wiersz 3–5: dołączamy trzy pliki nagłówkowe – `stdio.h`, `stdlib.h` – bo używamy funkcji `malloc()`, `string.h` – plik niezbędny przy wykonywaniu operacji na łańcuchach znaków.

Wiersz 6: deklarujemy źródłowy łańcuch znaków.

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

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

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

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

Wiersz 16: kopiujemy źródłowy łańcuch do miejsca przeznaczenia (`dest2`).

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( *przeznaczenie, *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: 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: 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 deklarujemy dwa łańcuchy znaków (zauważcie, iż `lancuch1` musi być umieszczony w większej tablicy, aby później znalazło się miejsce dla kopii drugiego łańcucha). W wierszu 11 wywołujemy funkcję `strcat()`, która dodaje kopię drugiego łańcucha na koniec pierwszego. Ostatecznie wyświetlamy 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()`. Jedyną różnicą jest 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: 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: }
```

Widzicie, iż w wierszu 11 wywołujemy funkcję `strncat()`, która skopiuje tylko 5 znaków z łańcucha `lancuch2` do łańcucha `lancuch1`.

Instrukcja switch

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

Konstrukcja instrukcji switch:

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

W polu `wyrażenie` najczęściej wpisujemy 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, wykonywany jest blok instrukcji po słowie kluczowym `default`.

Ćwiczenie 4.10.

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

```
1: /* Przyklad 4.10 */
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:               puts("Wpisales 1");
13:               break;
14:          }
15:          case 2:
16:          {
17:               puts("Wpisales 2");
18:               break;
19:          }
20:          case 3:
21:          {
22:               puts("Wpisales 3");
23:               break;
24:          }
25:          default:
26:          {
27:               puts("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 są strumienie?
- Jakie standardowe strumienie wejścia/wyjścia wyróżnia-my w języku C?
- Jaki jest prototyp oraz działanie funkcji `getchar()`?
- Jaki jest prototyp oraz działanie funkcji `gets()`?
- Z czego składa się funkcja `scanf()`?
- Jakie specyfikatory typu wyróżniam w języku C?
- Jakie modyfikatory precyzji wyróżniamy w języku C?
- Jaki jest prototyp oraz działanie funkcji `putchar()`?
- Jaki jest prototyp oraz działanie funkcji `puts()`?
- Na jakiej zasadzie odbywa się kopiowanie łańcuchów znaków ?
- Jaki jest prototyp oraz działanie funkcji `strcpy()`?
- Jaki jest prototyp oraz działanie funkcji `strncpy()`?
- 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?
- 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 rozwiązania

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.

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