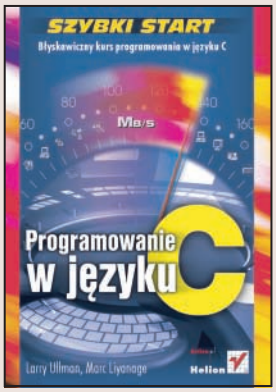
Programowanie w języku C. Szybki start



Autorzy: Larry Ullman, Marc Liyanage
Tłumaczenie: Rafał Jońca
ISBN: 83-7361-808-2
Tytuł oryginału: C Programming : Visual QuickStart Guide (Visual Quickstart Guides)
Format: B5, stron: 400

Błyskawiczny kurs programowania aplikacji w języku C

Język C, mimo prawie 30-letniej historii, cieszy się niesłabnącą popularnością wśród programistów. Wszyscy jego użytkownicy cenią w nim prostą i czytelną składnię, niewielki i łatwy do zapamiętania zakres słów kluczowych oraz duże możliwości, dzięki którym za pomocą C można rozwiązać niemal każde zagadnienie programistyczne. Zewnętrzne biblioteki, dostępne w sieci oraz dołączane do najpopularniejszych narzędzi programistycznych, dodatkowo rozszerzają możliwości C.

Książka „Programowanie w języku C. Szybki start" to podręcznik dla tych, którzy chcą poznać C w praktyce i nie chcą przebijać się przez dziesiątki stron opisów teoretycznych. Każde zagadnienie – od najprostszych, związanych ze strukturą programu i zasadami kompilacji aż do złożonych typów danych i obsługi plików zewnętrznych – jest przedstawione w postaci bogato ilustrowanej sekwencji czynności. Wykonując kolejne przykłady z książki, poznasz wszystkie podstawowe wiadomości o programowaniu w języku C.

- Struktura programu w języku C
- Typy danych
- Sterowanie działaniem programu
- Korzystanie ze standardowego wejścia i wyjścia
- Deklarowanie i stosowanie funkcji
- Dyrektywy preprocesora
- Wskaźniki i zarządzanie pamięcią
- Obsługa plików
- Złożone typy danych

Poznaj zalety i możliwości najpopularniejszego języka programowania

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Struktury sterujące 4

Choć przykłady przedstawione do tej pory korzystały ze zmiennych, przeprowadzały obliczenia i wyświetlały wyniki, brakowało im dynamicznej natury prawdziwego programowania. Jednym z kluczowych elementów prawdziwych aplikacji jest ich elastyczność zapewniaqna przez struktury sterujące takie jak pętle i instrukcje warunkowe.

Najczęściej stosowaną strukturą sterującą jest warunek `if` (jeśli) i jego odmiany: `if-else`, `if-elseif` oraz `if-elseif-else`. Rozdział rozpoczniemy od omówienia właśnie tego warunku. W tym czasie natkniemy się na nowe operatory języka C (poza już wymienionym operatorami arytmetycznymi i przypisania). Następnie przejdziemy do dwóch pozostałych instrukcji warunkowych — operatora trójargumentowego i konstrukcji `switch`. Na końcu rozdziału zajmiemy się dwoma typowymi rodzajami pętli: `while` i `for`. Po przeczytaniu tego rozdziału będziesz potrafił pisać bardziej funkcjonalne aplikację w języku C.

Instrukcja warunkowa if

Warunki to struktury sterujące pozwalające na **rozgałęzianie** przebiegu programu, czyli uzależnianie jego działania od różnych parametrów. Ze wszystkich rodzajów rozgałęzień najczęściej stosuje się instrukcję `if`. Jej składnia jest bardzo prosta:

```
if (warunek) {
  instrukcje;
}
```

Warunek umieszcza się wewnątrz nawiasów okrągłych, a instrukcje, które mają zostać wykonane, wewnątrz nawiasów klamrowych. Jeśli warunek jest prawdziwy, instrukcje zostaną wykonane. W przeciwnym razie zostaną pominięte.

W język C najprostszą reprezentacją fałszu jest wartość 0. Wszystko inne traktowane jest jako prawda.

```
if (1) {
  printf("Ten warunek zawsze jest prawdziwy.");
  printf("To jest kolejna instrukcja.");
}
```

W wyniku prawdziwości warunku wykonana może zostać dowolną liczba instrukcji. Jeżeli warunek dotyczy tylko jednej instrukcji, można pominąć nawiasy klamrowe. Oto przykład:

```
if (1) printf("Ten warunek zawsze jest
➥prawdziwy.");
```

Kolejny przykład zobrazuje, w jaki sposób korzystać z warunków w języku C.

Listing 4.1. *Podstawowa instrukcja warunkowa sprawdza, czy zmienna test ma wartość różną od 0*

Listing

```
1  /* if.c - listing 4.1 */
2
3  #include <stdio.h>
4
5  int main(void) {
6
7    int test = 1; // Zmienna będąca warunkiem.
8
9    // Wyświetlenie tekstu wprowadzenia.
10   printf("Testowane zmiennej test.\n");
11
12   // Sprawdzenie wartości zmiennej i wyświetlenie
     ➥tekstu (jeśli test = 0, tekst się nie wyświetli).
13   if (test) {
14     printf("*** Zmienna test zawiera
       ➥wartość %d.\n",test);
15   }
16
17   getchar(); /* Zatrzymaj, aż użytkownik naciśnie
     ➥klawisz Enter lub Return. */
18
19   return 0;
20
21 }
```

Aby utworzyć warunek if:

1. Utwórz nowy dokument w edytorze tekstu lub IDE.

2. Dodaj początkowy komentarz i kod (patrz listing 4.1).

```
/* if.c - listing 4.1 */
#include <stdio.h>
int main(void) {
```

3. Zadeklaruj i zainicjalizuj zmienną.

```
int test = 1;
```

   Zmienna test jest liczbą całkowitą (równie dobrze mogłaby być typu `unsigned short int`, jeśli chcielibyśmy być minimalistami). Posłuży nam ona jako element warunku w strukturze sterującej.

4. Dodanie komunikatu początkowego.

```
printf("Testowane zmiennej test.\n");
```

   Komunikat ten zapewni odpowiedni kontekst dla dowolnego kolejnego tekstu. Gwarantuje, że aplikacja wyświetli jakiś tekst niezależnie od spełnienia warunku (w przeciwnym razie przy fałszywym warunku tekst nie zostałby wyświetlony).

5. Zdefiniuj warunek `if`.

```
if (test) {
  printf("*** Zmienna test zawiera wartość
  ➥%d.\n",test);
}
```

   Jeśli warunek jest prawdziwy, zostanie wyświetlony dodatkowy tekst wraz z wartością zmiennej `test`. Tak sytuacja wystąpi, gdy zmienna `test` będzie zawierała wartość różną od 0. W przeciwnym razie (`test = 0`) instrukcja `printf()` w ogóle nie zostanie wykonana.

6. Dokończ funkcję `main()`.

```
    getchar();
    return 0;
}
```

7. Zapisz projekt w pliku *if.c*.

8. Skompiluj i sprawdź poprawność kompilacji.

9. Uruchom aplikację (patrz rysunek 4.1).

10. Aby sprawdzić działanie aplikacji dla zmiennej `test` równej 0, ponownie skompiluj aplikację, ustawiając `test` na 0 (patrz rysunek 4.2).

    W tym przypadku warunek jest fałszywy, więc druga instrukcja `printf()` nie zostanie wykonana.

Rysunek 4.1. *Jeśli zmienna test ma wartość różną od 0, zostanie wyświetlony dodatkowy komunikat*

Rysunek 4.2. *Jeśli zmienna test ma wartość równą 0, pojawi się tylko jeden komunikat*

Wskazówki

- Tworzenie instrukcji warunkowej dla jednego polecenia bez stosowania nawiasów klamrowych jest co prawda w pełni poprawne, ale może prowadzić do błędów, jeśli w przyszłości będzie się dokonywać modyfikacji warunku. Choć to rozwiązanie jest bardzo kuszące i można się na nie natknąć, przeglądając kod napisany przez inne osoby, nie warto go stosować.
- Standard C99 wprowadza nowy typ danej o nazwie `_Bool` — wartość logiczna. Zmienne tego typu zawsze będą posiadały wartość 0 (fałsz) lub 1 (prawda). Wpisanie do zmiennej jakiejkolwiek wartości różnej od 0 spowoduje zamianę tej zmiennej na wartość 1. Środowisko Dev-C++ w wersji 4. nie obsługuje typu `_Bool`. Jest on dostępny dopiero w wersji beta Dev-C++ 5.
- Dodatkowo standard C99 definiuje plik *stdbool.h*, który zawiera deklarację trzech słów kluczowych: `bool` (odpowiednik `_Bool`), `true` (odpowiednik liczby 1) oraz `false` (odpowiednik liczby 0). Czyni to kod języka C lepiej przenośnym do języka C++.

Tabela 4.1. *Operatory logiczne i porównania bardzo często występują w warunkach i innych strukturach sterujących*

Operatory logiczne i porównania

Operator	Znaczenie
>	większy od
<	mniejszy od
>=	mniejszy od lub równy
<=	większy od lub równy
==	równy
!=	różny od
&&	iloczyn logiczny (*and*)
||	suma logiczna (*or*)
!	negacja (*not*)

Operatory logiczne i porównania

Korzystanie z prostych zmiennych (jak w poprzednim przykładzie) raczej nie pozwoli nikomu zajść daleko w programowaniu w języku C. Aby móc tworzyć bardziej zaawansowane instrukcje warunkowe, trzeba wykorzystać operatory porównania i logiczne (patrz tabela 4.1).

Operatory porównania wykorzystuje się dla wartości liczbowych, na przykład by wskazać, czy dana wartość jest wyższa od innej. Wykorzystanie wartości takiego wyrażenia (prawda lub fałsz), pozwala konstruować bardziej przydatne instrukcje warunkowe.

```
if (wiek >= 18) {
  printf("Masz wystarczającą liczbę lat,
  ➥by móc głosować.");
}
```

Operatory logiczne bardzo często stosuje się w połączeniu z nawiasami, aby utworzyć bardziej złożone warunki, na przykład zakresy.

```
if ((wiek > 12) && (wiek < 20)) {
  printf("Uwaga - nastolatek!");
}
```

Szczególną uwagę należy zwracać na operator równości (==). Jednym z najczęstszych błędów popełnianych przez programistów (także tych wprawionych) jest przypadkowe stosowanie w miejscu operatora równości operatora przypisania (=).

```
if (zm = 190) {...
```

Programista w rzeczywistości chciał sprawdzić, czy wartość zmiennej `zm` jest **równa** 190 (chciał napisać `zm == 190`), co może, ale nie musi być prawdą. Niestety, brak jednego znaku powoduje, iż przedstawiony warunek zawsze będzie prawdziwy (zmienna przyjmie wartość 190, więc będzie różna od 0, co przełoży się na prawdziwość warunku).

W kolejnym przykładzie wykorzystamy operatory do wyświetlenia różnych tekstów w zależności od średniej ocen studenta.

Aby wykorzystać operatory logiczne i porównania:

1. Utwórz nowy dokument w edytorze tekstu lub IDE.
2. Dodaj początkowy komentarz i kod (patrz listing 4.2).

```
/* srednia.c - listing 4.1 */
#include <stdio.h>
int main(void) {
```

3. Zadeklaruj i zainicjalizuj zmienną.

```
float srednia = 3.8;
```

Zmienna `srednia` przechowuje średnią ocen studenta. Na początku zostanie ustawiona na wartość `3.8`.

4. Sprawdź, czy studentowi należy się stypendium za wyniki w nauce.

```
if (srednia >= 4.0) {
  printf("Otrzymujesz stypendium za wyniki
  ➥w nauce, ponieważ masz średnią %0.2f!\n",
  ➥srednia);
}
```

Jeśli wartość `srednia` jest wyższa lub równa `4.0`, wykonujemy instrukcję wyświetlenia tekstu na ekranie. Jeżeli średnia jest mniejsza, pomijamy instrukcję `printf()`.

5. Sprawdzenie, czy średnia kwalifikuje się do otrzymania książki.

```
if ((srednia >= 3.9) && (srednia<4.0)) {
  printf("Otrzymujesz książkę, ponieważ masz
  ➥średnią %0.2f!\n", srednia);
}
```

Listing 4.2. *Dzięki operatorom logicznym i porównania możliwe jest jednoczesne sprawdzanie kilku warunków*

Listing

```
1 /* srednia.c - listing 4.2 */
2
3 #include <stdio.h>
4
5 int main(void) {
6
7   float srednia = 3.8; // Średnia ocen studenta.
8
9   // Sprawdzenie, czy przysługuje stypendium (średnia
    ➥co najmniej 4.0).
10   if (srednia >= 4.0) {
11     printf("Otrzymujesz stypendium
       ➥za wyniki w nauce, ponieważ masz
       ➥średnią %0.2f!\n", srednia);
12   }
13
14   // Sprawdzenie, czy przysługuje książka za wyniki.
15   if ((srednia >= 3.9) && (srednia<4.0))
{
16     printf("Otrzymujesz książkę, ponieważ
       masz średnią %0.2f!\n", srednia);
17   }
18
19   // Sprawdzenie, czy przysługuje pochwała za wyniki.
20   if ((srednia >= 3.75) && (srednia<3.9)) {
21     printf("Otrzymujesz pochwałę, ponieważ
       ➥masz średnią %0.2f!\n", srednia);
22   }
23
24   getchar(); /* Zatrzymaj, aż użytkownik naciśnie
     ➥klawisz Enter lub Return. */
25
26   return 0;
27
28 }
```

Rysunek 4.3. *W zależności od zawartości zmiennej srednia na ekranie pojawiają się różne komunikaty*

Korzystając z operatorów logicznych i porównania, sprawdzamy, czy student osiągnął średnią większą od 3,9, ale jednocześnie nie przekroczył średniej 4,0. Z powodu zastosowania operatora iloczynu logicznego (`&&`), aby cały warunek był prawdziwy, oba warunki składowe muszą zwrócić wartość prawdy. Jeśli średnia jest mniejsza od 3,9 lub też równa albo większa od 4,0, instrukcja z warunku nie zostanie wykonana.

6. Sprawdzenie, czy student kwalifikuje się do otrzymania pochwały.

```
if ((srednia >= 3.75) && (srednia<3.9)) {
  printf("Otrzymujesz pochwałę, ponieważ
  ➥masz średnią %0.2f!\n", srednia);
}
```

Ostatni z warunków działa dokładnie tak samo jak poprzedni, ale korzysta z innych wartości w trakcie testów.

7. Dokończ funkcję `main()`.

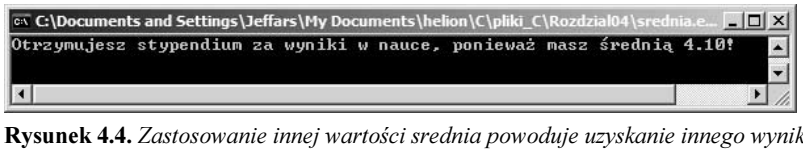
```
  getchar();
  return 0;
}
```

8. Zapisz program w pliku *srednia.c*.

9. Skompiluj i sprawdź poprawność kompilacji (ewentualnie popraw błędy).

10. Uruchom aplikację (patrz rysunek 4.3).

11. Dla porównania zmień wartość zmiennej `srednia` i ponownie skompiluj program (patrz rysunek 4.4).



Rysunek 4.4. *Zastosowanie innej wartości srednia powoduje uzyskanie innego wyniku*

Wskazówki

- Sposób, w jaki został napisany przykład, powoduje, że dla `sredniej` mniejszej od 3,75 nie zostanie wyświetlony żaden komunikat. Poprawimy tę kwestię w następnym przykładzie. Można też dodać kolejny warunek, który wyświetli komunikat dla wartości mniejszej od 3,75.
- Pamiętaj, aby nigdy nie sprawdzać równości dwóch zmiennych typu `float` lub `double`. Z racji takiego, a nie innego sposobu reprezentacji liczb zmiennoprzecinkowych w komputerze, dwie wartości wydające się być identycznymi mogą w rzeczywistości różnić się na jednej pozycji. Co więcej, liczba całkowita 2 może nie być równa liczbie zmiennoprzecinkowej 2,0.
- Niektórzy programiści zalecają odwracanie wartości w warunku wykorzystującym równość, gdy jednym ze sprawdzanych elementów jest stała. Oto przykład:

  ```
  if (24 == godziny) {...
  ```

  Zaletą takiego rozwiązania jest to, że gdy przypadkowo opuścimy jeden ze znaków równości, kompilator zgłosi błąd (ponieważ nie można przypisać zmiennej do konkretnej wartości).
- Aby sprawdzić, czy dwa ciągi znaków są identyczne, używa się specjalnej funkcji `strcmp()`. Funkcja ta zwraca liczbę różnic między obu sprawdzanymi tekstami. Jeśli zwróconą wartością jest 0, oba teksty są identyczne.

  ```
  if (strcmp(zm1, zm2) == 0) {...
  ```

  Więcej informacji na ten temat znajduje się w rozdziale 11., „Ciągi znaków”.

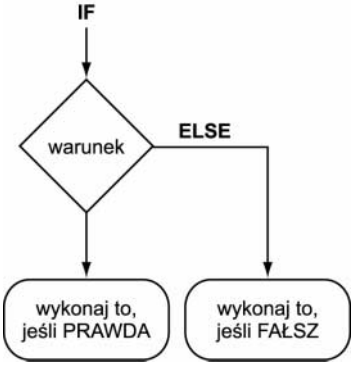
Dalszy ciąg kolejności operatorów

Podobnie jak operatory arytmetyczne także operatory logiczne i porównania mają swoją ściśle określoną kolejność wykonywania. Na przykład operatory `<`, `>`, `<=` i `>=` mają wyższy priorytet niż operatory `==` i `!=`.

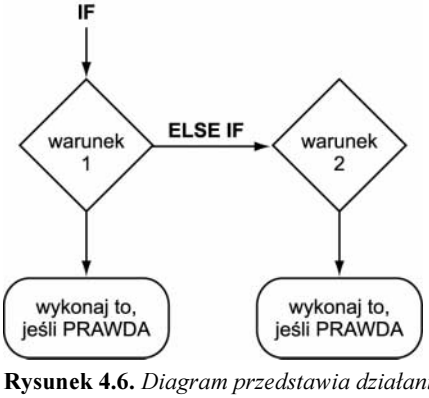
Jeśli uwzględnić wszystkie poznane do tej pory operatory, operatory porównania mają większy priorytet niż operator przypisania (`=`), ale niższy priorytet niż operatory arytmetyczne.

Co więcej, operator negacji (`!`) ma większy priorytet niż mnożenie i dzielenie, natomiast operatory iloczynu i sumy logicznej (`&&` i `||`) mają większy priorytet niż operator przypisania (`=`), ale niższy priorytet niż operatory porównań. Poza tym operator `&&` jest ważniejszy od operatora `||`.

Pogmatwane? Na pewno. Można zapamiętać kolejność wykonywania wszystkich operatorów (pełna lista priorytetów operatorów znajduje się w dodatku B) lub też po prostu stosować nawiasy.



Rysunek 4.5. *Diagram przedstawia działanie warunku if-else*



Rysunek 4.6. *Diagram przedstawia działanie warunku if-else if (elementów else if może być nieskończenie dużo)*

Klauzule else i else if

Instrukcja warunkowa `if` jest bardzo użyteczna, ale może zostać wzbogacona o dodatkowe elementy za pomocą klauzul `else` i `else if`. Składnia konstrukcji `if-else` jest następująca:

```
if (warunek) {
  /* Zrób coś. */
} else {
  /* W przeciwnym razie wykonaj to. */
}
```

Zauważ, że instrukcje z klauzuli `else` zostaną wykonane tylko wtedy, gdy główny warunek nie będzie prawdziwy. Innymi słowy, klauzula `else` działa jak odpowiedź domyślna (patrz rysunek 4.5).

Klauzula `else if` jest bardziej rozbudowana, gdyż pozwala dodatkowo sprawdzić kolejny warunek, jeśli okazało się, że pierwszy warunek nie był prawdziwy (patrz rysunek 4.6).

```
if (warunek 1) {
  /* Zrób coś. */
} else if (warunek 2) {
  /* Zrób coś innego. */
}
```

Liczba klauzul `else if` jest nieograniczona. Często łączy się klauzule `else if` i `else`. Ważne jest tylko, by klauzula `else` była ostatnia (gdyż jest rozwiązaniem domyślnym). Zmodyfikujmy wcześniejszy przykład z ocenami, aby uwzględniał nowe instrukcje.

Aby skorzystać z klauzul else i else if:

1. Otwórz plik *srednia.c* (listing 4.2) w edytorze tekstu lub środowisku IDE.
2. Usuń wszystkie istniejące warunki (patrz listing 4.3).

   Trzy osobne warunki `if` zostaną zastąpione jedną dużą konstrukcją `if-else if-else if-else`, więc nie będą już potrzebne.
3. Utwórz główną pętlę.

```
if (srednia >= 4.0) {
  printf("Otrzymujesz stypendium za wyniki
  ➥w nauce, ponieważ masz średnią %0.2f!\n",
  ➥srednia);
} else if (srednia >= 3.9) {
  ➥printf("Otrzymujesz książkę, ponieważ
  ➥masz średnią %0.2f!\n", srednia);
} else if (srednia >= 3.75) {
  ➥printf("Otrzymujesz pochwałę, ponieważ
  ➥masz średnią %0.2f!\n", srednia);
} else {
  printf("Może w przyszłym roku będzie
  ➥lepiej, średnia to nie wszystko.\n");
}
```

Same warunki są bardzo podobne do tych z poprzedniego przykładu, ale teraz wszystko stanowi jedną dużą instrukcję warunkową. Aplikacja będzie sprawdzała warunki tak długo, aż któryś z nich nie okaże się prawdziwy. W przeciwnym razie zostanie wykonana klauzula `else`.

Listing 4.3. *Korzystając z klauzul else i else if, zmodyfikowaliśmy wcześniejszy przykład ze średnią ocen*

Listing

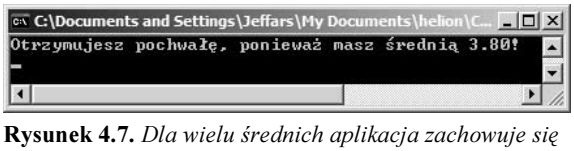
```
1  /* srednia2.c - listing 4.3 - przeróbka listingu 4.2
   ➥ (srednia.c) */
2
3  #include <stdio.h>
4
5  int main(void) {
6
7    float srednia = 3.8; // Średnia ocen studenta.
8
9    // Raport na temat średniej.
10   if (srednia >= 4.0) {
11     printf("Otrzymujesz stypendium
       ➥za wyniki w nauce, ponieważ masz
       ➥średnią %0.2f!\n", srednia);
12   } else if (srednia >= 3.9) {
13     printf("Otrzymujesz książkę, ponieważ
       ➥masz średnią %0.2f!\n", srednia);
14   } else if (srednia >= 3.75) {
15     printf("Otrzymujesz pochwałę, ponieważ
       ➥masz średnią %0.2f!\n", srednia);
16   } else {
17     printf("Może w przyszłym roku będzie
       ➥lepiej, średnia to nie wszystko.\n");
18   }
19
20   getchar(); /* Zatrzymaj, aż użytkownik naciśnie
     ➥klawisz Enter lub Return. */
21
22   return 0;
23
24 }
```

Zauważ, że drugi i trzeci warunek nie muszą sprawdzać, czy wartość jest mniejsza od konkretnej liczby. Na przykład, jeśli `srednia` wynosi 3,8, pierwszy i drugi warunek jest fałszywy, a dopiero trzeci jest prawdziwy. W trzecim warunku nie trzeba już sprawdzać, czy `srednia` jest mniejsza od 3,9, ponieważ zostało to stwierdzone już w drugim warunku. Skoro program sprawdza dany warunek, mamy pewność, że poprzednie warunki z tej samej konstrukcji były fałszywe.

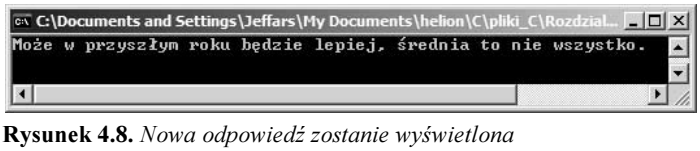
4. Zapisz program w pliku *srednia2.c*, skompiluj i uruchom go (patrz rysunek 4.7).
5. Zmień wartość zmiennej `srednia` na inną i ponownie skompiluj oraz uruchom aplikację (patrz rysunek 4.8).

Wskazówki

- Choć nie jest to wymagane, warto wprowadzać wcięcia dla instrukcji znajdujących się wewnątrz klauzul `if`, `else` i `else if`. Zwiększa to czytelność kodu.
- Podobnie jak zwykła konstrukcja `if`, także `else` i `else if` nie wymagają stosowania nawiasów klamrowych, gdy występuje po nich tylko jedna instrukcja. Warto jednak dodawać te nawiasy, aby w przyszłości nie popełnić błędów.



Rysunek 4.7. *Dla wielu średnich aplikacja zachowuje się dokładnie tak samo jak poprzednio (patrz rysunki 4.3 i 4.4) pomimo zmian w warunkach*



Rysunek 4.8. *Nowa odpowiedź zostanie wyświetlona dla średnich poniżej 3,75*

Operator trójargumentowy

Język C posiada składnię alternatywną dla konstrukcji `if-else`. Jest to tak zwany operator **trójargumentowy**. Nazwa wynika z faktu, że operator ten wymaga trzech parametrów (składa się z trzech części). Oto jego podstawowa składnia:

```
(warunek) ? wynik_dla_prawdy : wynik_dla_falszu;
```

Zauważmy, że operator ten zwraca jedną z dwóch wartości w zależności od warunku. Zwracaną wartość można przypisać do zmiennej lub wyświetlić. Na przykład poniższy kod zwróci informację o tym, czy liczba jest wartością parzystą czy nieparzystą. Do sprawdzenia parzystości używamy operatora reszty z dzielenia.

```
char parzysty_nieparzysty;
parzysty_nieparzysty = ( (liczba % 2) == 0) ?
➥'p' : 'n';
```

Gdyby w tym samym celu zastosować warunek `if`, wyglądałby on następująco:

```
char parzysty_nieparzysty;
if ( (liczba % 2) == 0) {
  parzysty_nieparzysty = 'p';
} else {
  parzysty_nieparzysty = 'n';
}
```

W następnym przykładzie operator trójargumentowy zostanie wykorzystany do wyświetlenia odpowiedniego komunikatu w zależności od temperatury.

Aby wykorzystać operator trójargumentowy:

1. Utwórz nowy, pusty dokument w edytorze tekstu lub IDE.
2. Dodaj początkowy komentarz i kod (listing 4.4).

```
/* pogoda.c - listing 4.4 */
#include <stdio.h>
int main(void) {
```

Listing 4.4. *Operator trójargumentowy często stosuje się jako krótszą wersję warunku if-else, jeśli trzeba zwrócić jedną z dwóch wartości*

Listing

```
1  /* pogoda.c - listing 4.4 */
2
3  #include <stdio.h>
4
5  int main(void) {
6
7    int temperatura = 38; // Temperatura
   ➥w stopniach Celsjusza.
8
9    if (temperatura > 30) { // Sprawdź,
   ➥czy gorąco.
10
11     printf("Jest %d stopni, czyli na dworze
       ➥jest bardzo gorąco.\n", temperatura);
12
13   } else if (temperatura < 10) { // Wyświetl
   ➥'zimno' lub 'lodowato' w zależności
   ➥ od temperatury.
14
15     printf("Jest %d stopni, czyli na dworze
       ➥jest %s!\n", temperatura,
       ➥(temperatura < 0 ) ? "lodowato" :
       ➥"zimno");
16
17   } else { // Nie jest ani za gorąco, ani za zimno.
18
19     printf("Jest %d stopni, czyli na dworze
       ➥jest całkiem przyjemnie.\n",
       ➥temperatura);
20
21   }
22
23
24   getchar(); /* Zatrzymaj, aż użytkownik naciśnie
   ➥klawisz Enter lub Return. */
25
26   return 0;
27
28 }
```

3. Zadeklaruj i zainicjalizuj zmienną całkowitą.

```
int temperatura = 38;
```

Ta zmienna przechowuje aktualną temperaturę w stopniach Celsjusza.

4. Rozpocznij główny warunek `if`.

```
if (temperatura > 30) {
  printf("Jest %d stopni, czyli na dworze
  ➥jest bardzo gorąco.\n", temperatura);
```

Pierwszy warunek sprawdza, czy na dworze jest ponad 30 stopni Celsjusza. Jeśli tak, wyświetla aktualną temperaturę i informuje o tym, że jest gorąco (jest to program dla osób, które lubią, by komputer informował je o rzeczach oczywistych).

5. Dodaj warunek `else if` wraz z operatorem trójargumentowym wewnątrz instrukcji `printf()`.

```
} else if (temperatura < 10) {
  printf("Jest %d stopni, czyli na dworze
  ➥jest %s!\n", temperatura, (temperatura
  ➥< 0 ) ? "lodowato" : "zimno");
```

Jeśli jest poniżej 10 stopni, aplikacja wyświetli `lodowato` lub `zimno` w zależności od tego, jak bardzo jest zimno. Zamiast używać osobnej konstrukcji `else if`, stosujemy operator trójargumentowy wewnątrz instrukcji `printf()`.

Znacznik `%s` wewnątrz tekstu komunikatu określa miejsce, w którym zostanie wstawiona wartość zwrócona przez operator trójargumentowy (patrz trzeci parametr funkcji `printf()`). Operator sprawdza, czy temperatura ma wartość poniżej 0. Jeśli tak, zwraca wartość `lodowato`.
W przeciwnym razie zwraca wartość `zimno`.
Zauważ, że obie wartości to ciągi znaków (ponieważ znajdują się w cudzysłowach).

6. Dokończ warunek.

```
} else {
  printf("Jest %d stopni, czyli na dworze
  ➥jest całkiem przyjemnie.\n", temperatura);
}
```

Jeżeli temperatura jest między 10 a 30 stopniami, informujemy o przyjemnej pogodzie za oknem.

7. Zakończ funkcję `main()`.

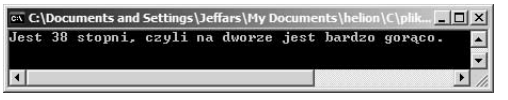
```
  getchar();
  return 0;
}
```

8. Zapisz dokument w pliku *pogoda.c*.

9. Skompiluj kod.

10. Uruchom aplikację (patrz rysunek 4.9).

11. Zmodyfikuj wartość zmiennej `temperatura`, a następnie ponownie skompiluj i uruchom aplikację (patrz rysunek 4.10).



Rysunek 4.9. *Jeśli jest ponad 30 stopni, pojawia się następujący komunikat (patrz listing 4.4)*

Rysunek 4.10. *Jeśli jest poniżej 10 stopni, za pomocą operatora trójargumentowego wybierany jest komunikat lodowato lub zimno*

Wskazówki

- Można zmodyfikować przedstawiony powyżej kod, na przykład zmieniając pierwszy warunek na 22 stopnie i dodając w instrukcji `printf()` operator trójargumentowy rozróżniający stan `ciepło` i `gorąco`.
- Główną zaletą operatora trójargumentowego jest jego bardzo zwięzły zapis w porównaniu z konstrukcją `if-else`. Wadą jest zmniejszona przejrzystość kodu.

Instrukcja switch

Poza operatorem trójargumentowym istnieje również inna odmiana tradycyjnego warunku, a mianowicie instrukcja `switch`. Przyjmuje ona jako swój parametr wartość całkowitą, a następnie sprawdza ją względem kilku wymienionych możliwości. Oto przykład:

```
switch (rok) {
  case 2005:
    /* Zrób coś. */
    break;
  case 2004:
    /* Zrób coś innego. */
    break;
  default:
    /* Zrób to. */
    break;
}
```

Instrukcja `break` jest niezmiernie ważna, jeśli chodzi o działanie instrukcji `switch`. Po dotarciu do `break` aplikacja opuści cały fragment `switch`. Jeśli pominie się `break`, wykonane zostaną także pozostałe instrukcje, nawet te należące do innych możliwości.

Możliwość `default` jest opcjonalna, ale gdy już się ją dodaje, to zazwyczaj na samym końcu instrukcji `switch`. Jeśli żadna z wcześniejszych możliwości nie będzie poprawna, wykonany zostanie kod możliwości `default` (działa ona mniej więcej tak samo jak klauzula `else`).

Kolejny przykład działa różnie w zależności od wartości właściwości `plec`. Zmienna typu `char`, która tak naprawdę jest okrojonym typem `int`, bez problemów funkcjonuje w instrukcji `switch`.

Aby użyć instrukcji switch:

1. Utwórz nowy, pusty dokument w edytorze tekstu lub IDE.

2. Dodaj początkowy komentarz i kod (listing 4.5).

```
/* plec.c - listing 4.5 */
#include <stdio.h>
int main(void) {
```

3. Zadeklaruj i zainicjalizuj zmienną znakową.

```
char plec = 'M';
```

   Zmienna przechowuje płeć osoby jako pojedynczy znak.

4. Rozpocznij instrukcję switch.

```
switch (plec) {
```

   Poprawna składnia instrukcji switch wymaga zastosowania słowa kluczowego switch i podania w nawiasach testowanej zmiennej. Otwierający nawias klamrowy oznacza początek zawartości instrukcji switch.

5. Dodaj pierwszą możliwość.

```
case 'M':
  printf("Witam pana!\n");
  break;
```

   Pierwszy element sprawdza, czy zmienna zawiera wartość M. Jeśli tak, wyświetlany jest tekst Witam pana!. Instrukcja break wymusza wyjście z konstrukcji switch, aby pozostałe instrukcje printf() nie zostały wykonane.

6. Dodaj drugą możliwość.

```
case 'K':
  printf("Witam panią!\n");
  break;
```

   Struktura tego elementu jest taka sama jak elementu z kroku poprzedniego. Po prostu testowana jest inna wartość.

Listing 4.5. *Instrukcja switch pozwala określić różne odpowiedzi dla różnych wartości całkowitych lub znaków*

Listing

```
1  /* plec.c - listing 4.5 */
2
3  #include <stdio.h>
4
5  int main(void) {
6
7    char plec = 'M'; // Płeć jako znak pojedynczy.
8
9    switch (plec) { // Zmień wyświetlane
   ➥pozdrowienie.
10
11     case 'M':
12       printf("Witam pana!\n");
13       break;
14
15     case 'K':
16       printf("Witam panią!\n");
17       break;
18
19     default:
20       printf("Witam wszystkich!\n");
21       break;
22
23   } // Koniec instrukcji switch.
24
25   getchar(); /* Zatrzymaj, aż użytkownik
   ➥naciśnie klawisz Enter lub Return. */
26
27   return 0;
28
29 }
```

Rysunek 4.11. *Wynik wykonania programu dla zmiennej ustawionej na wartość M*

Rysunek 4.12. *Jeśli zmienna nie jest równa M ani F, stosowany jest element domyślny*

- Nic nie stoi na przeszkodzie, by te same instrukcje dotyczyły kilku możliwości. Wtedy składnia konstrukcji `switch` jest następująca:

```
switch (plec) {
   case 'M':
   case 'm':
     /* Zrób coś. */
     break;
   case 'K':
   case 'k':
      /* I tak dalej... */
      break;
  }
```

7. Dodaj domyślny element i zakończ instrukcję `switch`.

```
   default:
     printf("Witam wszystkich!\n");
     break;
}
```

   Jeśli zmienna `plec` nie zawiera wartości `M` lub `K` (choć jest to mało prawdopodobne), pojawi się ogólny komunikat.

8. Zakończ funkcję `main()`.

```
   getchar();
   return 0;
}
```

9. Zapisz dokument w pliku *plec.c*.
10. Skompiluj przykład.
11. Uruchom aplikację (patrz rysunek 4.11).
12. Dla porównania zmień wartość zmiennej na inną literę lub spację, a następnie ponownie skompiluj i uruchom aplikację (patrz rysunek 4.12).

Wskazówki

- Główną wadą instrukcji `switch` jest to, że można jej używać tylko dla liczb całkowitych i pojedynczych znaków. Znacznie ogranicza to jej przydatność. Gdy jednak można ją zastosować, byłaby konstrukcją szybszą i bardziej przejrzystą niż `if`.
- W języku C istnieje jeszcze jedna instrukcja sterująca nieomawiana w tej książce — instrukcja `goto`. Osoby zaznajomione z innymi językami, takimi jak na przykład Basic lub Pascal, powinny ją znać, ale tak naprawdę w języku C nie jest ona do niczego potrzebna.

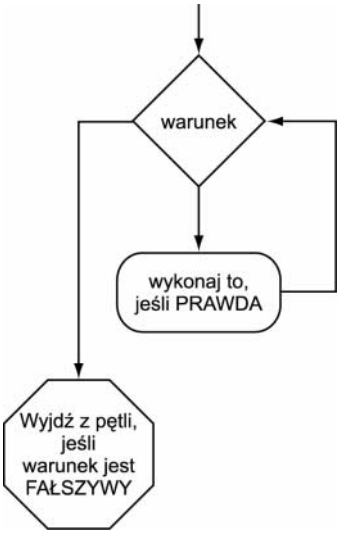
Pętla while

Instrukcje warunkowe to tylko jeden rodzaj sterowania działaniem programu — drugim są pętle. Język C obsługuje dwie postacie pętli: `while` (i jego siostra `do...while`) oraz `for`. Każdy z typów pętli wykonuje to samo zadanie — powtarza określony ciąg instrukcji aż do uzyskania fałszywości pewnego warunku — ale w nieco inny sposób.

Pętla `while` wygląda bardzo podobnie do instrukcji `if`, gdyż wykonuje pewne instrukcje, gdy określony warunek jest prawdziwy.

```
while (warunek) {
  /* Zrób coś. */
}
```

Gdy wykona się jedna iteracja pętli, warunek sprawdzany jest ponownie. Jeżeli nadal jest prawdziwy, zachodzi kolejna iteracja pętli. Jeśli jest fałszywy, program przechodzi do następnej instrukcji po pętli (patrz rysunek 4.13).



Rysunek 4.13. *Diagram prezentujący działanie pętli w języku C. Instrukcje wykonywane są tak długo, jak długo warunek jest prawdziwy*

Typowym błędem początkujących programistów jest tworzenie pętli, w których warunek nigdy nie stanie się fałszywy. Powstaje wtedy **pętla nieskończona** (aplikacja działa i działa). Z tego powodu trzeba zapewnić takie określenie zawartości pętli, by w pewnym momencie warunek przestał być prawdziwy.

W następnym przykładzie wykorzystamy pętlę `while` do napisania programu, który obliczy silnię z wybranej liczby. Silna (reprezentowana przez znak wykrzyknika) to wartość uzyskana z mnożenia wszystkich liczb całkowitych od 1 aż do podanej wartości. Oto przykład:

```
5! = 1 * 2 * 3 * 4 * 5 // 120
3! = 1 * 2 * 3 // 6
```

Listing 4.6. *Pętla while pomaga policzyć silnię z wybranej liczby. Pętla wykonuje się tak długo, jak długo zmienna mnoznik jest mniejsza lub równa zmiennej liczba*

Listing

```
1  /* silnia.c - listing 4.6 */
2
3  #include <stdio.h>
4
5  int main(void) {
6
7    // W tym przykładzie stosujemy tylko liczby
     ➥dodatnie.
8
9    unsigned int liczba = 5; // Wartość, z której
     ➥wyliczamy silnię.
10   unsigned int wynik = 1; // Zmienna
     ➥przechowująca przyszły wynik - silnię.
11   unsigned int mnoznik = 1; // Mnożnik
     ➥używany do obliczenia silni.
12
13   // Przejdź przez wszystkie mnożniki od 1 do liczba.
14   while (mnoznik <= liczba) {
15
16     wynik *= mnoznik; // Mnożenie
       ➥wcześniejszego wyniku przez mnożnik.
17     ++mnoznik; // Inkrementacja mnożnika.
18
19   } // Koniec pętli.
20
21   // Wyświetlenie wyniku.
22   printf("Silnia z liczby %u to %u.\n",
     liczba, wynik);
23
24   getchar(); /* Zatrzymaj, aż użytkownik naciśnie
     ➥klawisz Enter lub Return. */
25
26   return 0;
27
28 }
```

Aby użyć pętli while:

1. Utwórz nowy dokument w edytorze tekstu lub IDE.
2. Dodaj początkowy komentarz i kod (listing 4.6).

```
/* silnia.c - listing 4.6 */
#include <stdio.h>
int main(void) {
```

3. Zadeklaruj i ustaw wartości zmiennych.

```
unsigned int liczba = 5;
unsigned int wynik = 1;
unsigned int mnoznik = 1;
```

Aplikacja korzysta z trzech wartości całkowitych bez znaku. Przypisujemy zmiennym wartości początkowe. Zmienna `liczba` określa wartość, dla której liczymy silnię. Zmienna `wynik` przechowuje wyniki obliczeń. Zmienna `mnoznik` jest wykorzystywana w pętli do obliczenia silni.

4. Rozpoczęcie pętli `while`.

```
while (mnoznik <= liczba) {
```

Silnię liczymy, mnożąc wynik przez kolejne liczby całkowite od 1 do `liczba`. Obliczenia wykonujemy w pętli, więc warunek pętli musi pozwalać wyjść z pętli po wykonaniu odpowiedniej liczby mnożeń. Gdy `mnoznik` jest mniejszy lub równy zmiennej `liczba`, oznacza to, że obliczenia jeszcze się nie zakończyły i trzeba wykonać zawartość pętli. W przeciwnym razie wszystkie obliczenia zostały przeprowadzone.

5. Wykonaj obliczenia.

```
wynik *= mnoznik;
```

Przy wykorzystaniu operatora przypisania z mnożeniem wartość zmiennej `wynik` jest ustawiana na wartość `wynik` pomnożoną przez wartość zmiennej `mnoznik`. Przy pierwszej iteracji pętli zmienna `wynik` będzie równa 1 (1·1). W drugiej iteracji będzie równa 2 (1·2), w trzeciej 6 (2·3), w czwartej 24 (6·4), a w piątej i ostatniej 120 (24·5).

6. Zwiększ wartość zmiennej `mnoznik` o 1.

```
++mnoznik;
```

W pewien sposób jest to najważniejszy wiersz pętli. Jeśli wartość zmiennej mnożnik nie byłaby zwiększana, warunek trwania pętli byłby zawsze prawdziwy, więc powstałaby pętla nieskończona.

7. Dokończ pętlę `while`.

```
} // Koniec pętli.
```

Gdy kod staje się coraz to bardziej złożony, warto oznaczać komentarzem, której pętli dotyczy dane zakończenie.

8. Wyświetlenie wyników obliczeń.

```
printf("Silnia z liczby %u to %u.\n",
➥liczba, wynik);
```

Instrukcja `printf()` wyświetla zarówno parametr silni, jak i sam wynik obliczeń.

9. Dokończ funkcję `main()`.

```
  getchar();
  return 0;
}
```

10. Zapisz dokument w pliku *silnia.c*.

11. Skompiluj kod źródłowy.

Instrukcje break, continue i exit

Instrukcja `break` (która występuje między innymi we wnętrzu instrukcji `switch`) jest po prostu jedną z konstrukcji języka C stosowaną wewnątrz struktur sterujących. Przypomnijmy, że instrukcja `break` pozwala opuścić aktualną pętlę lub instrukcję `switch`. Oto przykładowy sposób jej użycia:

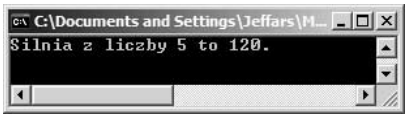
```
while (warunek1) {
  /* Wykonaj cokolwiek. */
  if (warunek2) {
    break; /* Opuszczenie pętli. */
  }
}
```

Instrukcja `continue` pozwala opuścić aktualną iterację pętli i powrócić do sprawdzania warunku. Po sprawdzeniu warunku pętla może, ale nie musi, być wykonywana po raz kolejny.

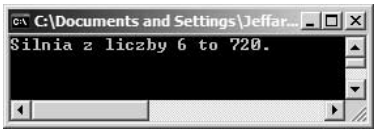
```
while (warunek1) {
  /* Wykonaj cokolwiek. */
  if (warunek2) {
    continue; /* Powrót do początku pętli. */
  }
}
```

Warto podkreślić, że przedstawione powyżej instrukcje działają tylko w pętlach i instrukcjach `switch`. Nie są aktywne w instrukcjach `if`.

Inną konstrukcją języka C jest instrukcja `exit` powodująca wyłączenie całej aplikacji. Pod koniec książki pokażemy, w jaki sposób skorzystać z tej instrukcji, jeśli w trakcie działania programu wystąpi poważny problem.



Rysunek 4.14. *Uruchomienie aplikacji pozwala poznać wartość silni dla liczby 5*



Rysunek 4.15. *Wystarczy tylko zmienić wartość zmiennej liczba, aby program przedstawił nową wartość silni*

12. Uruchom aplikację (patrz rysunek 4.14).

13. Dla porównania zmodyfikuj wartość zmiennej `liczba`, a następnie ponownie skompiluj i uruchom program (patrz rysunek 4.15).

Wskazówki

- Istnieje odmiana pętli `while` o nazwie `do...while`. Główna różnica między nią a przedstawioną wcześniej wersją polega na tym, że warunek sprawdzany jest na końcu pętli (co oznacza, że pętla wykona się co najmniej jeden raz). Oto składnia tej pętli:

```
do {
  /* instrukcje */
} while (warunek);
```

- W rozdziale 5., „Standardowe wejście i wyjście", wykorzystamy pętlę `while` do ciągłego pobierania znaków z klawiatury.
- Przedstawiona aplikacja z silnią może być doskonałym przykładem problemu z przepełnieniem (patrz rozdział 3., „Liczby"). Gdy oblicza się silnię z dużych liczb, bardzo łatwo można przekroczyć dopuszczalny zakres wartości liczb całkowitych.

Pętla for

Ostatnią strukturą sterującą omawianą w tym rozdziale (jak i w całej książce) jest pętla `for`. Podobnie jak pętla `while` wykonuje ona pewną liczbę iteracji, a każda iteracja składa się z wcześniej określonych poleceń. Choć działanie obu pętli jest podobne, ich składnia jest zdecydowanie inna.

```
for (wyrażenie inicjalizujące; warunek; wyrażenie
➥iteracyjne) {
  /* Zrób cokolwiek. */
}
```

Gdy aplikacja po raz pierwszy dotrze do pętli, wykonuje wyrażenie inicjalizujące. Następnie sprawdzony zostanie warunek działania pętli. Jeśli będzie prawdziwy, wykona się kod pętli (reprezentowany tutaj przez komentarz `Zrób cokolwiek.`). Przed przejściem do następnej iteracji zostanie wykonane wyrażenie iteracyjne. Następnie dojdzie do ponownego sprawdzenia warunku. Cały proces będzie się powtarzał (poza wyrażeniem inicjalizującym, patrz rysunek 4.16) aż do uzyskania fałszywego warunku działania pętli. W pętli `for` **wyrażenie iteracyjne** jest odpowiedzialne za doprowadzenie w pewnym momencie do zmiany warunku pętli na fałsz.

Zmodyfikujmy przykład z silnią w taki sposób, aby wykorzystać pętlę `for`. Przekonamy się, że pętli `for` i `while` używa się w bardzo podobny sposób.

Aby użyć pętli for:

1. Otwórz plik *silnia.c* (listing 4.6) w edytorze tekstu lub środowisku IDE.
2. Zmodyfikuj nazwę zmiennej `mnoznik` na `i` oraz nie ustawiaj jej wartości początkowej (patrz listing 4.7).

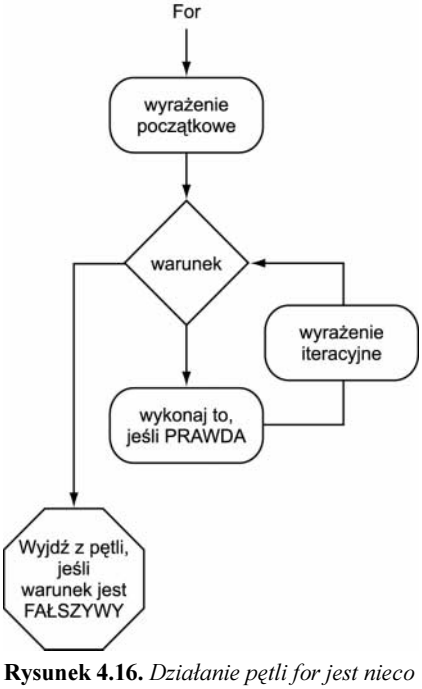
   ```
   unsigned int i;
   ```

   Przyjęło się oznaczać w języku C zmienną związaną z pętlą `for` literą `i`. Choć nie jest to konieczne, w tym rozdziale zastosujemy się do tego zalecenia. Nie trzeba przy deklaracji ustawiać wartości początkowej zmiennej, gdyż zostanie to wykonane w pętli `for`.

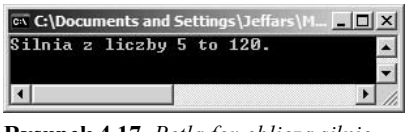
Listing 4.7. *Powrót do przykładu z silnią i zamiana pętli while na pętlę for*

Listing

```
1  /* silnia2.c - listing 4.7 - modyfikacja listingu 4.6
   ➥(silnia.c) */
2
3  #include <stdio.h>
4
5  int main(void) {
6
7    // W tym przykładzie stosujemy tylko liczby
     ➥dodatnie.
8
9    unsigned int liczba = 5; // Wartość,
     ➥z której wyliczamy silnię.
10   unsigned int wynik = 1; // Zmienna
     ➥przechowująca przyszły wynik - silnię.
11   unsigned int i; // Mnożnik używany
     ➥do obliczenia silni.
12
13   // Przejdź przez wszystkie mnożniki od 1 do liczba.
14   for (i = 1; i <= liczba; ++i) {
15
16     wynik *= i; // Mnożenie wcześniejszego
       ➥wyniku przez mnożnik.
17
18   } // Koniec pętli.
19
20   // Wyświetlenie wyniku.
21   printf("Silnia z liczby %u to %u.\n",
     ➥liczba, wynik);
22
23   getchar(); /* Zatrzymaj, aż użytkownik naciśnie
     ➥klawisz Enter lub Return. */
24
25   return 0;
26
27 }
```



Rysunek 4.16. *Działanie pętli for jest nieco inne od działania pętli while, gdyż dochodzi inicjalizacja zmiennej i wyrażenie inkrementacyjne*



Rysunek 4.17. *Pętla for oblicza silnię równie wydajnie jak pętla while*

3. Usuń całą pętlę `while`.

4. Rozpocznij definicję pętli `for`.

```
for (i = 1; i <= liczba; ++i) {
```

Przy pierwszym napotkaniu pętli aplikacja ustawi wartość zmiennej `i` na 1. Następnie, jeśli liczba jest większa lub równa `i`, pętla będzie wykonywać instrukcję z kroku 5. Po wykonaniu instrukcji z pętli (po każdej iteracji) nastąpi inkrementacja zmiennej `i` o 1.

5. Dodaj instrukcję wewnątrz pętli.

```
wynik *= i;
```

Ponieważ mnożnik nosi teraz nazwę `i`, trzeba odpowiednio zmodyfikować wnętrze pętli. Zauważ, że inkrementacja mnożnika występuje teraz w definicji pętli i nie stanowi części instrukcji wykonywanych w samej pętli.

6. Zamknij pętlę.

```
} // Koniec pętli.
```

7. Zapisz dokument w pliku *silnia2.c*.

8. Skompiluj i w razie konieczności popraw kod programu. Uruchom aplikację wynikową (patrz rysunek 4.17).

9. Jeśli chcesz, zmodyfikuj wartość zmiennej `liczba`, a następnie skompiluj i ponownie uruchom aplikację.

Wskazówki

- Choć na ogół w pętli `for` korzysta się z trzech osobnych wyrażeń, można utworzyć bardziej rozbudowane rozwiązanie. Pierwsza i ostatnia część pętli `for` (wyrażenie początkowe i iteracyjne) może posiadać wiele wyrażeń, jeśli zostaną one oddzielone przecinkami.

```
for (wynik = 1, i = 1; i <= liczba; ++i) {...
```

- Co więcej, wszystkie z trzech wyrażeń są opcjonalne.

```
for (;;) {...
```

  Powyższy kod jest w pełni poprawny i powoduje utworzenie pętli nieskończonej.

- Pętlę `for` bardzo często stosuje się w połączeniu z tablicami, aby wykonać operacje na ich elementach. To zagadnienie przedstawimy dokładniej w rozdziale 6., „Tablice".

- W trakcie zagnieżdżania pętli `for` (patrz ramka) często najbardziej zewnętrzna pętla stosuje zmienną `i`, bardziej wewnętrzna zmienną `j`, a najbardziej wewnętrzna zmienną `k`.

Zagnieżdżanie instrukcji warunkowych i pętli

Język C umożliwia zagnieżdżanie różnych struktur sterujących, na przykład umieszczenie jednej instrukcji warunkowej wewnątrz innej, jednej pętli wewnątrz drugiej, pętli wewnątrz instrukcji warunkowej itp. Gdy jednak korzysta się z tego rozwiązania, niezmiernie ważne jest zachowanie odpowiedniej składni. Oto kilka sugestii związanych z zagnieżdżaniem struktur sterujących:

- Zawsze używaj otwierających i zamykających nawiasów klamrowych do oznaczenia początku i końca struktury sterującej.
- Stosuj coraz większe wcięcia dla kolejnych zagnieżdżeń.
- Korzystaj z komentarzy w celu opisania działania struktury sterującej.

W trakcie stosowania zagnieżdżeń często popełnianym błędem jest brak zrównoważenia liczby nawiasów otwierających i zamykających. W dalszej części książki pojawi się wiele zagnieżdżeń. Wszystkie stosują się do powyższych sugestii w celu zminimalizowania liczby błędów.