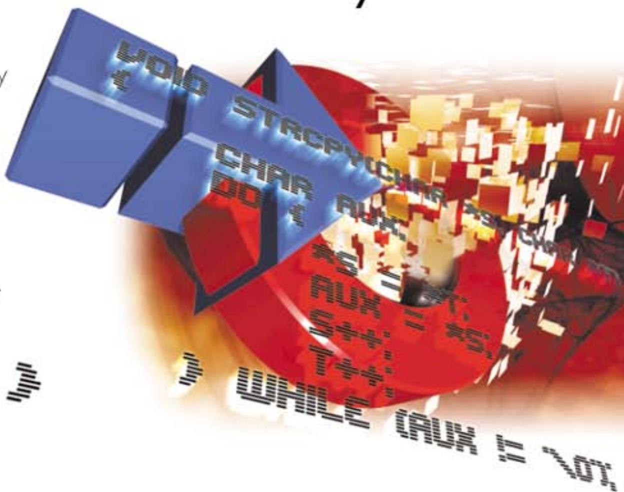
Poznaj w praktyce podstawowe narzędzie pracy profesjonalnych programistów!

Ćwiczenia praktyczne

Programowanie w języku C

Marek Tłuczek

Wydanie II



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

helion.pl

Idź do
- Spis treści
- Przykładowy rozdział

Katalog książek
- Katalog online
- Zamów drukowany katalog

Twój koszyk
- Dodaj do koszyka

Cennik i informacje
- Zamów informacje o nowościach
- Zamów cennik

Czytelnia
- Fragmenty książek online

Programowanie w języku C. Ćwiczenia praktyczne. Wydanie II

Autor: Marek Tłuczek
ISBN: 978-83-246-2834-6
Format: 140×208, stron: 120

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

Poznaj w praktyce podstawowe narzędzie pracy profesjonalnych programistów!

Opracowanie języka C było milowym krokiem w historii rozwoju informatyki i choć od czasu jego powstania minęło już niemal czterdzieści lat, nadal jest to jeden z najbardziej popularnych języków programowania na świecie. Zawdzięcza to swojej elastyczności, dużym możliwościom, wysokiej wydajności działania, łatwości tworzenia i konserwacji kodu oraz niezależności od platformy sprzętowej. Nie bez znaczenia jest też fakt, że na jego składni oparte są inne nowoczesne języki wysokiego poziomu, takie jak C++, C# czy Java – i że to właśnie jego poznanie jest często pierwszym krokiem na drodze do kariery profesjonalnego programisty.

Niezależnie od tego, z jakich powodów chcesz nauczyć się języka C, doskonałą pomocą okaże się książka „Programowanie w języku C. Ćwiczenia praktyczne. Wydanie II". Poprawiona i uzupełniona edycja ćwiczeń bezboleśnie wprowadzi Cię w świat programowania strukturalnego. Poznasz podstawowe pojęcia związane z językiem C i zasady tworzenia poprawnego kodu, nauczysz się prawidłowo korzystać z różnych typów danych i instrukcji, a także dowiesz się, jak przeprowadzać operacje wejścia-wyjścia. Zgłębisz również tajniki bardziej zaawansowanych technik, takich jak używanie wskaźników, tablic i struktur. Jeśli chcesz zacząć przygodę z programowaniem w C, trafiłeś na idealną książkę!

- Podstawy tworzenia kodu w C
- Definiowanie stałych i zmiennych oraz ich używanie
- Stosowanie prostych i złożonych typów danych
- Używanie instrukcji warunkowych i tworzenie pętli
- Korzystanie z funkcji standardowych
- Posługiwanie się łańcuchami znakowymi
- Operacje związane ze strumieniami wejścia-wyjścia
- Definiowanie i używanie wskaźników do danych i funkcji

Nauka języka C jeszcze nigdy nie była tak prosta!

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl
© Helion 1991–2011

Spis treści

	Wstęp	**5**
Rozdział 1.	**Podstawy języka C**	**7**
	Tworzenie programu w C	7
	printf() — funkcja wyjścia	9
	Zmienne w języku C	11
	Stałe w C	15
	scanf() — funkcja wejścia	17
	Instrukcja warunkowa if	19
	Co powinieneś zapamiętać z tego cyklu ćwiczeń?	25
	Ćwiczenia do samodzielnego wykonania	26
Rozdział 2.	**Programowanie strukturalne**	**27**
	Funkcje	28
	Pętle w języku C	35
	Wstęp do tablic	35
	Instrukcja switch	42
	Co powinieneś zapamiętać z tego cyklu ćwiczeń?	44
	Ćwiczenia do samodzielnego wykonania	45
Rozdział 3.	**Język C dla wtajemniczonych**	**47**
	Tablice wielowymiarowe	47
	Wskaźniki	51
	Wskaźniki i tablice	52
	Znaki oraz łańcuchy znaków	56
	Znaki	57
	Łańcuchy znaków	58

Zastosowanie wskaźników 65
Przekazywanie przez wskaźnik zmiennej jako argumentu funkcji 65
Dynamiczny przydział pamięci 67
Operacje arytmetyczne na wskaźnikach 68
Struktury w języku C 74
Co powinieneś zapamiętać z tego cyklu ćwiczeń? 78
Ćwiczenia do samodzielnego wykonania 80

Rozdział 4. Operacje wejścia-wyjścia 81
Strumienie wejścia-wyjścia 81
Funkcje wejścia 82
Funkcje wyjścia 86
Operacje na łańcuchach znaków 87
Kopiowanie łańcuchów znaków 88
Łączenie łańcuchów znaków 90
Operacje na plikach 92
Otwieranie, tworzenie i zamykanie plików tekstowych 92
Odczytywanie pliku tekstowego 93
Zapisywanie pliku tekstowego 97
Co powinieneś zapamiętać z tego cyklu ćwiczeń? 101
Ćwiczenia do samodzielnego wykonania 102

Rozdział 5. Język C dla guru 103
Struktury ze wskaźnikami 103
Wskaźniki do funkcji 108
Tablice wskaźników do funkcji 112
Preprocesor 113
Sparametryzowane makrodefinicje (makra) 115
Kompilacja warunkowa 116
Co powinieneś zapamiętać z tego cyklu ćwiczeń? 118
Ćwiczenia do samodzielnego wykonania 119

Język C dla guru

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

Struktury ze wskaźnikami

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

ĆWICZENIE

5.1

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ĆWICZENIE

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

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

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

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

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

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

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

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

Wskaźniki do funkcji

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

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

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

Nawiasy są konieczne, ponieważ w przypadku deklaracji:

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

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

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

```
int funkcja(int)
```

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

```
wsk_do_funkcji = funkcja;
```

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

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

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

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

ĆWICZENIE

5.3

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

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

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

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

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

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

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

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

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

Wiersze 23 – 40: definiowane są funkcje obsługujące zdarzenie.

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

Tablice wskaźników do funkcji

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

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

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

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

ĆWICZENIE

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

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

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

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

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

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

Preprocesor

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

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

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

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

Po przetworzeniu przez preprocesor program będzie wyglądał następująco:

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

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

Sparametryzowane makrodefinicje (makra)

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

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

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

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

odczytujemy jako:

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

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

Wyobraź sobie następujący przykład:

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

Po przetworzeniu instrukcja warunkowa wyglądałaby tak:

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

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

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

ĆWICZENIE

5.5

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

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

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

Kompilacja warunkowa

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

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

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

Najlepiej zilustruje to poniższy fragment kodu:

```
#define DEBUG

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

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

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

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

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

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

```
#ifndef MOJ_PLIK_NAGLOWKOWY
#define MOJ_PLIK_NAGLOWKOWY

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

#endif
```

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

Co powinieneś zapamiętać z tego cyklu ćwiczeń?

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

Ćwiczenia do samodzielnego wykonania

ĆWICZENIE 1.

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

ĆWICZENIE 2.

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

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

ĆWICZENIE 3.

Dodaj obsługę nowego zdarzenia do programu z ćwiczenia 5.3.

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

ĆWICZENIE 4.

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

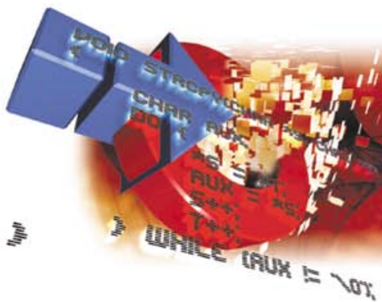
Pamiętaj o zastosowaniu dyrektyw preprocesora: `#ifndef`, `#define` i `#endif`.

ĆWICZENIE 5.

Napisz sparametryzowaną makrodefinicję obliczającą pierwiastek kwadratowy podanego parametru.

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

Programowanie w języku C



Opracowanie języka C było milowym krokiem w historii rozwoju informatyki i choć od czasu jego powstania minęło już niemal czterdzieści lat, nadal jest to jeden z najbardziej popularnych języków programowania na świecie. Zawdzięcza to swojej elastyczności, dużym możliwościom, wysokiej wydajności działania, łatwości tworzenia i konserwacji kodu oraz niezależności od platformy sprzętowej. Nie bez znaczenia jest też fakt, że na jego składni oparte są inne nowoczesne języki wysokiego poziomu, takie jak C++, C# czy Java – i że to właśnie jego poznanie jest często pierwszym krokiem na drodze do kariery profesjonalnego programisty.

Niezależnie od tego, z jakich powodów chcesz nauczyć się języka C, doskonałą pomocą okaże się książka „Programowanie w języku C. Ćwiczenia praktyczne. Wydanie II". Poprawiona i uzupełniona edycja ćwiczeń bezboleśnie wprowadzi Cię w świat programowania strukturalnego. Poznasz podstawowe pojęcia związane z językiem C i zasady tworzenia poprawnego kodu, nauczysz się prawidłowo korzystać z różnych typów danych i instrukcji, a także dowiesz się, jak przeprowadzać operacje wejścia-wyjścia. Zgłębisz również tajniki bardziej zaawansowanych technik, takich jak używanie wskaźników, tablic i struktur. Jeśli chcesz zacząć przygodę z programowaniem w C, trafiłeś na idealną książkę!

- Podstawy tworzenia kodu w C
- Definiowanie stałych i zmiennych oraz ich używanie
- Stosowanie prostych i złożonych typów danych
- Używanie instrukcji warunkowych i tworzenie pętli
- Korzystanie z funkcji standardowych
- Posługiwanie się łańcuchami znakowymi
- Operacje związane ze strumieniami wejścia-wyjścia
- Definiowanie i używanie wskaźników do danych i funkcji

Nauka języka C jeszcze nigdy nie była tak prosta!

helion.pl
księgarnia internetowa

Helion

Nr katalogowy: 5650

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900

0 601 339900

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

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

Informatyka w najlepszym wydaniu

Cena 19,90 zł
ISBN 978-83-246-2834-6