

PROGRAMOWANIE W C

Sprytne podejście do trudnych zagadnień,
których wolałbyś unikać (takich jak język C)

Z E D A . S H A W

Tytuł oryginału: Learn C the Hard Way: Practical Exercises on the Computational Subjects You Keep Avoiding (Like C)

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-2545-6

Authorized translation from the English language edition, entitled: LEARN C THE HARD WAY: PRACTICAL EXERCISES ON THE COMPUTATIONAL SUBJECTS YOU KEEP AVOIDING (LIKE C); ISBN 0321884922; by Zed A. Shaw; published by Pearson Education, Inc, publishing as Addison-Wesley. Copyright © 2016 by Zed A. Shaw.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.
Polish language edition published by HELION S.A. Copyright © 2016.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/pcspry>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Podziękowania	12
Ta książka tak naprawdę nie jest o języku C	13
Niezdefiniowane zachowania	14
C to język zarazem świetny i paskudny	15
Czego się nauczysz?	16
Jak czytać tę książkę?	16
Wideo	17
Podstawowe umiejętności	18
Czytanie i pisanie	18
Zwracanie uwagi na szczegóły	18
Wychwytywanie różnic	19
Planowanie i debugowanie	19
Przygotowania	20
Linux	20
OS X	20
Windows	21
Edytor tekstu	21
Nie używaj IDE	22
Ćwiczenie 1. Odkurzenie kompilatora	24
Omówienie kodu w pliku	24
Co powinienś zobaczyć?	25
Jak to zepsuć?	26
Zadania dodatkowe	26
Ćwiczenie 2. Użycie pliku Makefile podczas kompilacji	28
Użycie narzędzia make	28
Co powinienś zobaczyć?	29
Jak to zepsuć?	30
Zadania dodatkowe	30
Ćwiczenie 3. Sformatowane dane wyjściowe	32
Co powinienś zobaczyć?	33
Zewnętrzne badania	33
Jak to zepsuć?	33
Zadania dodatkowe	34

Ćwiczenie 4. Użycie debugera	36
Sztuczki z GDB	36
Krótki przewodnik po GDB	36
Krótki przewodnik po LLDB	37
Ćwiczenie 5. Nauka na pamięć operatorów w C	40
Jak uczyć się na pamięć?	40
Listy operatorów	41
Ćwiczenie 6. Nauka na pamięć składni C	46
Słowa kluczowe	46
Składnia struktur	47
Słowo zachęty	50
Słowo ostrzeżenia	51
Ćwiczenie 7. Zmienne i typy	52
Co powinienś zobaczyć?	53
Jak to zepsuć?	54
Zadania dodatkowe	54
Ćwiczenie 8. Konstrukcje if, else-if i else	56
Co powinienś zobaczyć?	57
Jak to zepsuć?	57
Zadania dodatkowe	58
Ćwiczenie 9. Pętla while i wyrażenia boolowskie	60
Co powinienś zobaczyć?	60
Jak to zepsuć?	61
Zadania dodatkowe	61
Ćwiczenie 10. Konstrukcja switch	62
Co powinienś zobaczyć?	64
Jak to zepsuć?	65
Zadania dodatkowe	65
Ćwiczenie 11. Tablice i ciągi tekstowe	66
Co powinienś zobaczyć?	67
Jak to zepsuć?	68
Zadania dodatkowe	69

Ćwiczenie 12. Wielkość i tablice	70
Co powinienś zobaczyć?	71
Jak to zepsuć?	72
Zadania dodatkowe	73
Ćwiczenie 13. Pętla for i tablica ciągów tekstowych	74
Co powinienś zobaczyć?	75
Zrozumienie tablicy ciągów tekstowych	76
Jak to zepsuć?	76
Zadania dodatkowe	77
Ćwiczenie 14. Tworzenie i użycie funkcji	78
Co powinienś zobaczyć?	79
Jak to zepsuć?	80
Zadania dodatkowe	80
Ćwiczenie 15. Wskaźniki, przerażające wskaźniki	82
Co powinienś zobaczyć?	84
Poznajemy wskaźniki	85
Praktyczne użycie wskaźników	86
Leksykon wskaźnika	87
Wskaźniki nie są tablicami	87
Jak to zepsuć?	87
Zadania dodatkowe	88
Ćwiczenie 16. Struktury i prowadzące do nich wskaźniki	90
Co powinienś zobaczyć?	93
Poznajemy struktury	94
Jak to zepsuć?	94
Zadania dodatkowe	95
Ćwiczenie 17. Alokacja pamięci stosu i serty	96
Co powinienś zobaczyć?	102
Alokacja stosu kontra serty	102
Jak to zepsuć?	103
Zadania dodatkowe	104
Ćwiczenie 18. Wskaźniki do funkcji	106
Co powinienś zobaczyć?	110
Jak to zepsuć?	110
Zadania dodatkowe	111

Ćwiczenie 19. Opracowane przez Zeda wsłania makra debugowania	112
Problem obsługi błędów w C	112
Makra debugowania	113
Użycie dbg.h	115
Co powinienes zobaczyć?	118
W jaki sposób CPP obsługuje makra?	118
Zadania dodatkowe	120
Ćwiczenie 20. Zaawansowane techniki debugowania	122
Użycie makra debug() kontra GDB	122
Strategia debugowania	124
Zadania dodatkowe	125
Ćwiczenie 21. Zaawansowane typy danych i kontrola przepływu	126
Dostępne typy danych	126
Modyfikatory typu	126
Kwalifikatory typów	127
Konwersja typu	127
Wielkość typu	128
Dostępne operatory	129
Operatory matematyczne	130
Operatory danych	130
Operatory logiczne	131
Operatory bitowe	131
Operatory boolowskie	131
Operatory przypisania	131
Dostępne struktury kontroli	132
Zadania dodatkowe	132
Ćwiczenie 22. Stos, zakres i elementy globalne	134
Pliki ex22.h i ex22.c	134
Plik ex22_main.c	136
Co powinienes zobaczyć?	138
Zakres, stos i błędy	139
Jak to zepsuć?	140
Zadania dodatkowe	141
Ćwiczenie 23. Poznaj mechanizm Duffa	142
Co powinienes zobaczyć?	145
Rozwiązanie łamigłównki	145
Dlaczego w ogóle mam się tak męczyć?	146
Zadania dodatkowe	146

Ćwiczenie 24. Dane wejściowe, dane wyjściowe i pliki	148
Co powinienś zobaczyć?	150
Jak to zepsuć?	151
Funkcje wejścia-wyjścia	151
Zadania dodatkowe	152
Ćwiczenie 25. Funkcje o zmiennej liczbie argumentów	154
Co powinienś zobaczyć?	158
Jak to zepsuć?	158
Zadania dodatkowe	158
Ćwiczenie 26. Projekt logfind	160
Specyfikacja logfind	160
Ćwiczenie 27. Programowanie kreatywne i defensywne	162
Nastawienie programowania kreatywnego	162
Nastawienie programowania defensywnego	163
8 strategii programisty defensywnego	164
Zastosowanie ośmiu strategii	164
Nigdy nie ufaj danym wejściowym	164
Unikanie błędów	168
Awaryjne powinny być wczesne i otwarte	169
Dokumentuj założenia	170
Preferuj prewencję zamiast dokumentacji	170
Automatyzuj wszystko	171
Upraszczaj i wyjaśniaj	171
Myśl logicznie	172
Kolejność nie ma znaczenia	172
Zadania dodatkowe	173
Ćwiczenie 28. Pośrednie pliki Makefile	174
Podstawowa struktura projektu	174
Makefile	175
Nagłówek	176
Docelowe wersje programu	177
Testy jednostkowe	178
Operacje porządkujące	180
Instalacja	180
Sprawdzenie	180
Co powinienś zobaczyć?	181
Zadania dodatkowe	181

Ćwiczenie 29. Biblioteki i linkowanie	182
Dynamiczne wczytywanie biblioteki współdzielonej	183
Co powinienś zobaczyć?	185
Jak to zepsuć?	187
Zadania dodatkowe	187
Ćwiczenie 30. Zautomatyzowane testowanie	188
Przygotowanie frameworka testów jednostkowych	189
Zadania dodatkowe	193
Ćwiczenie 31. Najczęściej spotykane niezdefiniowane zachowanie	194
20 najczęściej spotykanych przypadków niezdefiniowanego zachowania	196
Najczęściej spotykane niezdefiniowane zachowanie	196
Ćwiczenie 32. Lista dwukierunkowa	200
Czym są struktury danych?	200
Budowa biblioteki	200
Lista dwukierunkowa	202
Definicja	202
Implementacja	204
Testy	207
Co powinienś zobaczyć?	210
Jak można usprawnić kod?	210
Zadania dodatkowe	211
Ćwiczenie 33. Algorytmy listy dwukierunkowej	212
Sortowanie bąbelkowe i sortowanie przez scalanie	212
Test jednostkowy	213
Implementacja	215
Co powinienś zobaczyć?	217
Jak można usprawnić kod?	218
Zadania dodatkowe	219
Ćwiczenie 34. Tablica dynamiczna	220
Wady i zalety	227
Jak można usprawnić kod?	228
Zadania dodatkowe	228

Ćwiczenie 35. Sortowanie i wyszukiwanie	230
Sortowanie pozycyjne i wyszukiwanie binarne	233
Unie w języku C	234
Implementacja	235
Funkcja RadixMap_find() i wyszukiwanie binarne	241
RadixMap_sort() i radix_sort()	242
Jak można usprawnić kod?	243
Zadania dodatkowe	244
Ćwiczenie 36. Bezpieczniejsze ciągi tekstowe	246
Dlaczego stosowanie ciągów tekstowych C to niewiarygodnie kiepski pomysł?	246
Użycie bstrlib	248
Poznajemy bibliotekę	249
Ćwiczenie 37. Struktura Hashmap	250
Testy jednostkowe	257
Jak można usprawnić kod?	259
Zadania dodatkowe	260
Ćwiczenie 38. Algorytmy struktury Hashmap	262
Co powinienś zobaczyć?	267
Jak to zepsuć?	268
Zadania dodatkowe	269
Ćwiczenie 39. Algorytmy ciągu tekstowego	270
Co powinienś zobaczyć?	277
Analiza wyników	279
Zadania dodatkowe	280
Ćwiczenie 40. Binarne drzewo poszukiwań	282
Jak można usprawnić kod?	295
Zadania dodatkowe	295
Ćwiczenie 41. Projekt devpkg	296
Co to jest devpkg?	296
Co chcemy zbudować?	296
Projekt	297
Biblioteki Apache Portable Runtime	297
Przygotowanie projektu	299
Pozostałe zależności	299
Plik Makefile	299

Pliki kodu źródłowego	300
Funkcje bazy danych	302
Funkcje powłoki	305
Funkcje poleceń programu	309
Funkcja main() w devpkg	314
Ostatnie wyzwanie	316
Ćwiczenie 42. Stos i kolejka	318
Co powinienś zobaczyć?	321
Jak można usprawnić kod?	321
Zadania dodatkowe	322
Ćwiczenie 43. Prosty silnik dla danych statystycznych	324
Odchylenie standardowe i średnia	324
Implementacja	325
Jak można użyć tego rozwiązania?	330
Zadania dodatkowe	331
Ćwiczenie 44. Bufor cykliczny	334
Testy jednostkowe	337
Co powinienś zobaczyć?	337
Jak można usprawnić kod?	338
Zadania dodatkowe	338
Ćwiczenie 45. Prosty klient TCP/IP	340
Modyfikacja pliku Makefile	340
Kod netclient	340
Co powinienś zobaczyć?	344
Jak to zepsuć?	344
Zadania dodatkowe	344
Ćwiczenie 46. Drzewo trójkowe	346
Wady i zalety	354
Jak można usprawnić kod?	355
Zadania dodatkowe	355
Ćwiczenie 47. Szybszy router URL	356
Co powinienś zobaczyć?	358
Jak można usprawnić kod?	359
Zadania dodatkowe	360

Ćwiczenie 48. Prosty serwer sieciowy	362
Specyfikacja	362
Ćwiczenie 49. Serwer danych statystycznych	364
Specyfikacja	364
Ćwiczenie 50. Routing danych statystycznych	366
Ćwiczenie 51. Przechowywanie danych statystycznych	368
Specyfikacja	368
Ćwiczenie 52. Hacking i usprawnianie serwera	370
Zakończenie	372
Skorowidz	373

Lista dwukierunkowa

Zadaniem tej książki jest pokazanie, jak naprawdę działa komputer, co obejmuje również poznanie sposobu funkcjonowania różnych struktur danych i algorytmów. Komputery same z siebie nie przeprowadzają użytecznych operacji. Aby komputer mógł robić przydatne rzeczy, potrzebuje struktury danych, a następnie organizuje przetwarzanie tych struktur. W różnych językach programowania znajdują się dołączone biblioteki implementujące wszystkie tego rodzaju struktury lub dostępna jest bezpośrednia składnia dla nich. W języku C musisz samodzielnie zaimplementować wszystkie niezbędne struktury danych, co czyni z niego doskonały język, pozwalający poznać faktyczny sposób działania struktur danych.

Moim celem jest pomóc Ci w trzech zadaniach:

- Zrozumienie, co tak naprawdę dzieje się w kodzie języka Python, Ruby lub JavaScript w następującej postaci: `data = {"name": "Zed"}`.
- Utworzenie jeszcze lepszego kodu w języku C przez użycie struktur danych w odniesieniu do doskonale znanych problemów, dla których istnieją opracowane gotowe rozwiązania.
- Poznanie podstawowego zbioru struktur danych i algorytmów, aby jeszcze dokładniej wiedzieć, co najlepiej sprawdzi się w danych sytuacjach.

Czym są struktury danych?

Nazwa *struktura danych* jest samoobjaśniająca się. To po prostu metoda organizacji danych dopasowana do określonego modelu. Być może ten model jest przeznaczony do rozpoczęcia przetwarzania danych w zupełnie nowy sposób. A może jedynie jest zorganizowany tak, aby zapewnić możliwość efektywnego przechowywania danych na dysku. W tej książce będziemy stosować prosty wzorzec, zapewniający niezawodne działanie struktur danych:

- zdefiniowanie struktury dla głównej struktury zewnętrznej;
- zdefiniowanie struktury dla treści, zwykle węzłów wraz z łączami między nimi;
- utworzenie funkcji operujących na tych dwóch strukturach.

Istnieją jeszcze inne style struktur danych w języku C, ale przedstawiony powyżej wzorzec sprawdza się doskonale i zapewnia spójność podczas tworzenia większości struktur danych.

Budowa biblioteki

W pozostałej części książki będziemy tworzyć bibliotekę, z której później będziesz mógł korzystać. Biblioteka ta będzie miała następujące elementy:

- pliki nagłówkowe (.h) dla każdej struktury danych,
- pliki implementacji (.c) dla algorytmów,

- testy jednostkowe przeznaczone do przetestowania całej funkcjonalności i zagwarantowania jej prawidłowego działania,
- dokumentacja wygenerowana automatycznie na podstawie plików nagłówkowych.

Ponieważ mamy szkielet katalogu projektu (*c-skeleton*), więc użyjemy go do utworzenia projektu *liblcthw*.

Sesja dla ćwiczenia 32.:

```
$ cp -r c-skeleton liblcthw
$ cd liblcthw/
$ ls
LICENSE Makefile README.md bin build src tests
$ vim Makefile
$ ls src/
dbg.h libex29.c libex29.o
$ mkdir src/lcthw
$ mv src/dbg.h src/lcthw
$ vim tests/minunit.h
$ rm src/libex29.* tests/libex29*
$ make clean
rm -rf build tests/libex29_tests
rm -f tests/tests.log
find . -name "*.gc*" -exec rm {} \;
rm -rf `find . -name "*.dSYM" -print`
$ ls tests/
minunit.h runtests.sh
$
```

W powyższej sesji wykonałem następujące operacje:

- Utworzyłem kopię katalogu *c-skeleton*.
- Przeprowadziłem edycję pliku *Makefile*, aby zmienić nazwę *libNAZWA-BIBLIOTEKI.a* na *liblcthw* jako nowy cel (TARGET).
- Utworzyłem katalog *src/lcthw* przeznaczony na kod źródłowy.
- Przeniósłem plik *src/dbg.h* do nowego katalogu.
- Przeprowadziłem edycję pliku *tests/minunit.h* w celu dodania wiersza `#include <lcthw/dbh.h>`.
- Pozbyłem się niepotrzebnych plików kodu źródłowego i testów z projektu *libex29.**.
- Uprzątnąłem wszystko pozostałe.

Teraz jesteś gotowy do rozpoczęcia pracy nad budową biblioteki. Pierwsza tworzona tutaj struktura danych to lista dwukierunkowa.

Lista dwukierunkowa

Pierwszą strukturą danych, jaką dodamy do `liblcthw`, będzie lista dwukierunkowa (ang. *doubly linked list*). To jest najprostsza struktura danych, jaką można utworzyć i jaka oferuje użyteczne właściwości dla pewnych operacji. Działanie listy dwukierunkowej opiera się na węzłach zawierających wskaźniki do następnego i poprzedniego elementu. Lista dwukierunkowa zawiera wskaźniki do obu wymienionych elementów, natomiast lista jednokierunkowa (ang. *singly linked list*) tylko do następnego elementu.

Ponieważ każdy węzeł ma wskaźniki prowadzące do poprzedniego i następnego elementu, a także dlatego, że monitorowany jest pierwszy i ostatni element listy, za pomocą listy dwukierunkowej można bardzo szybko przeprowadzić pewne operacje. Każde zadanie obejmujące wstawienie lub usunięcie elementu będzie wykonywane bardzo szybko. Ponadto tego rodzaju listy są niezwykle łatwe do implementacji dla większości programistów.

Największą wadą listy jest to, że poruszanie się po niej wymaga przetworzenia każdego wskaźnika po drodze. Oznacza to, że operacje wyszukiwania, operacje sortowania i iteracji przez elementy będą w większości wykonywane wolno. Ponadto tak naprawdę nie można przechodzić do losowo wybranych elementów listy. Jeżeli masz tablicę elementów, za pomocą odpowiedniego indeksu możesz przejść do elementu w środku tablicy. Z kolei lista używa strumienia wskaźników. Dlatego też jeśli chcesz uzyskać dostęp do dziesiątego elementu, musisz najpierw przejść przez pierwszych dziewięć.

Definicja

Jak wspomniałem na początku ćwiczenia, zaczniemy od utworzenia pliku nagłówkowego zawierającego polecenia odpowiedniej struktury C.

Plik *list.h*:

```
#ifndef lcthw_List_h
#define lcthw_List_h

#include <stdlib.h>

struct ListNode;

typedef struct ListNode {
    struct ListNode *next;
    struct ListNode *prev;
    void *value;
} ListNode;

typedef struct List {
    int count;
    ListNode *first;
    ListNode *last;
} List;
```

```

List *List_create();
void List_destroy(List * list);
void List_clear(List * list);
void List_clear_destroy(List * list);

#define List_count(A) ((A)->count)
#define List_first(A) ((A)->first != NULL ? (A)->first->value : NULL)
#define List_last(A) ((A)->last != NULL ? (A)->last->value : NULL)

void List_push(List * list, void *value);
void *List_pop(List * list);
void List_unshift(List * list, void *value);
void *List_shift(List * list);

void *List_remove(List * list, ListNode * node);

#define LIST_FOREACH(L, S, M, V) ListNode *_node = NULL;\
    ListNode *V = NULL;\
    for(V = _node = L->S; _node != NULL; V = _node = _node->M)

#endif

```

Zaczynamy od utworzenia dwóch struktur dla `ListNode` i `List`, które będą zawierały wspomniane węzły. W ten sposób powstaje struktura danych przeznaczona do użycia w funkcjach i makrach, jakie zostaną później zdefiniowane. Jeżeli spojrzysz na te funkcje, to zobaczysz, że są całkiem proste. Wprowadzie do tych funkcji powrócimy przy okazji omawiania implementacji, ale mam nadzieję, że łatwo odgadniesz ich przeznaczenie.

Każdy element `ListNode` zawiera trzy komponenty wewnątrz struktury danych:

- Wartość będąca wskaźnikiem do czegoś i przechowująca to, co chcemy umieścić na liście.
- Wskaźnik `ListNode *next` prowadzący do następnej struktury `ListNode` przechowującej następny element na liście.
- Wskaźnik `ListNode *prev` prowadzący do poprzedniego elementu. To jest skomplikowane, prawda? Nazwanie poprzedniego elementu po prostu „poprzednim”. Mógłbym użyć słowa „wcześniejszy” lub „tylny”, ale robią tak tylko pajace.

Struktura `List` jest zaledwie kontenerem przeznaczonym do przechowywania struktur `ListNode` połączonych ze sobą w łańcuchu. Monitoruje wartości określające liczbę elementów na liście (`count`), a także pierwszy (`first`) i ostatni (`last`).

Na koniec spójrz na wiersz 37. pliku `src/lcthw/list.h`, w którym zdefiniowałem makro `LIST_FOREACH()`. Jest to często spotykane rozwiązanie w programowaniu, polegające na utworzeniu makra przeznaczonego do wygenerowania kodu iteracji, aby programiści nie mogli w tym kodzie namieszać. W przypadku struktur danych zapewnienie prawidłowego przetwarzania tego rodzaju może być trudne, więc wspomniane makro okazuje się dużą pomocą. Sposób jego użycia zobaczysz, gdy będziemy omawiać plik implementacji.

Implementacja

Na tym etapie powinieneś już znać ogólny sposób działania listy dwukierunkowej. Przypominam, że to po prostu węzły wraz ze wskaźnikami prowadzącymi do poprzedniego i następnego elementu na liście. Teraz możesz więc przystąpić do wprowadzenia kodu w pliku `src/lcthw/list.c` i poznać implementację poszczególnych operacji.

Plik *list.c*

```
1 #include <lcthw/list.h>
2 #include <lcthw/dbg.h>
3
4 List *List_create()
5 {
6     return calloc(1, sizeof(List));
7 }
8
9 void List_destroy(List * list)
10 {
11     LIST_FOREACH(list, first, next, cur) {
12         if (cur->prev) {
13             free(cur->prev);
14         }
15     }
16     free(list->last);
17     free(list);
18 }
19
20 void List_clear(List * list)
21 {
22     LIST_FOREACH(list, first, next, cur) {
23         free(cur->value);
24     }
25 }
26
27 void List_clear_destroy(List * list)
28 {
29     List_clear(list);
30     List_destroy(list);
31 }
32
33 void List_push(List * list, void *value)
34 {
35     ListNode *node = calloc(1, sizeof(ListNode));
36     check_mem(node);
37     node->value = value;
38
39     if (list->last == NULL) {
40         list->first = node;
41         list->last = node;
42     }
43 }
```

```
44     } else {
45         list->last->next = node;
46         node->prev = list->last;
47         list->last = node;
48     }
49
50     list->count++;
51
52 error:
53     return;
54 }
55
56 void *List_pop(List * list)
57 {
58     ListNode *node = list->last;
59     return node != NULL ? List_remove(list, node) : NULL;
60 }
61
62 void List_unshift(List * list, void *value)
63 {
64     ListNode *node = calloc(1, sizeof(ListNode));
65     check_mem(node);
66
67     node->value = value;
68
69     if (list->first == NULL) {
70         list->first = node;
71         list->last = node;
72     } else {
73         node->next = list->first;
74         list->first->prev = node;
75         list->first = node;
76     }
77
78     list->count++;
79
80 error:
81     return;
82 }
83
84 void *List_shift(List * list)
85 {
86     ListNode *node = list->first;
87     return node != NULL ? List_remove(list, node) : NULL;
88 }
89
90 void *List_remove(List * list, ListNode * node)
91 {
92     void *result = NULL;
93
94     check(list->first && list->last, "Lista jest pusta.");
95     check(node, "Wartością node nie może być NULL.");
96 }
```

```

97     if (node == list->first && node == list->last) {
98         list->first = NULL;
99         list->last = NULL;
100    } else if (node == list->first) {
101        list->first = node->next;
102        check(list->first != NULL,
103            "Nieprawidłowa lista, jakoś tak się stało, że pierwszy element
           ↳ ma wartość NULL.");
104        list->first->prev = NULL;
105    } else if (node == list->last) {
106        list->last = node->prev;
107        check(list->last != NULL,
108            "Nieprawidłowa lista, jakoś tak się stało, że następny element
           ↳ ma wartość NULL.");
109    list->last->next = NULL;
110    } else {
111        ListNode *after = node->next;
112        ListNode *before = node->prev;
113        after->prev = before;
114        before->next = after;
115    }
116
117    list->count--;
118    result = node->value;
119    free(node);
120
121 error:
122     return result;
123 }

```

Następnie implementujemy wszystkie operacje na liście dwukierunkowej, czego nie można zrobić za pomocą prostych makr. Zamiast omawiać każdy, najmniejszy wiersz kodu przedstawionego pliku, poniżej prezentuję jedynie ogólne omówienie każdej operacji zdefiniowanej w plikach *list.h* i *list.c*, a później zostawię Cię, abyś mógł przeanalizować kod.

list.h:List_count(). Zwraca liczbę elementów na liście. Ta wartość jest modyfikowana wraz z dodawaniem i usuwaniem elementów.

list.h:List_first(). Zwraca pierwszy element listy, ale nie usuwa go.

list.h:List_last(). Zwraca ostatni element listy, ale nie usuwa go.

list.h:LIST_FOREACH(). Przeprowadza iterację przez elementy listy.

list.h:List_create(). Po prostu tworzy główną strukturę List.

list.h:List_destroy(). Usuwa strukturę List wraz z wszystkimi elementami, jakie mogły się w niej znajdować.

list.h:List_clear(). Wygodna funkcja pozwalająca na zwolnienie *wartości* w poszczególnych węzłach, a nie same węzły.

list.h:List_clear_destroy(). Usuwa zawartość listy oraz samą listę. Działanie nie należy do zbyt efektywnych, ponieważ iteracja przez listę odbywa się dwukrotnie.

list.h:List_push(). To jest pierwsza operacja pokazująca zalety listy dwukierunkowej. Umieszcza nowy element na końcu listy. Ponieważ to po prostu kilka wskaźników, operacja jest niezwykle szybka.

list.h:List_pop(). Działanie odwrotne do List_pop() powoduje usunięcie ostatniego elementu listy i jego zwrot.

list.h:List_unshift(). Kolejna operacja łatwa do przeprowadzenia na liście dwukierunkowej, czyli bardzo szybkie dodanie elementów na *początku* listy. W omawianym przypadku nadałem funkcji nazwę List_unshift(), ponieważ nic lepszego nie przyszło mi na myśl.

list.h:List_shift(). Podobnie jak w przypadku List_pop(), ale ta funkcja powoduje usunięcie i zwrot pierwszego elementu listy.

list.h:List_remove(). To jest operacja faktycznie odpowiedzialna za usunięcie elementu w przypadku użycia List_pop() lub List_shift(). Usuwanie elementów wydaje się trudne w przypadku struktur danych i ta funkcja to potwierdza. Obsługuje ona niemałą liczbę warunków w zależności od tego, gdzie znajduje się element przeznaczony do usunięcia: na początku, na końcu, na początku i na końcu, w środku.

Większość funkcji znajdujących się w omówionym pliku niczym specjalnym się nie wyróżnia i dlatego bardzo łatwo można odgadnąć przeznaczenie kodu. Zdecydowanie powinieneś skoncentrować się na makrze LIST_FOREACH() użytym w funkcji List_destroy(), aby przekonać się, jak bardzo może ono uprościć tę często wykonywaną operację.

Testy

Po skompilowaniu kodu źródłowego listy możemy przystąpić do przygotowania testów, aby mieć pewność, że utworzone wcześniej operacje listy funkcjonują prawidłowo.

Plik *list_tests.c*:

```
1 #include "minunit.h"
2 #include <lcthw/list.h>
3 #include <assert.h>
4
5 static List *list = NULL;
6 char *test1 = "dane testowe 1";
7 char *test2 = "dane testowe 2";
8 char *test3 = "dane testowe 3";
9
10 char *test_create()
11 {
12     list = List_create();
13     mu_assert(list != NULL, "Nie udało się utworzyć listy.");
14
15     return NULL;
16 }
17
```

```
18 char *test_destroy()
19 {
20     List_clear_destroy(list);
21
22     return NULL;
23 }
24
25
26 char *test_push_pop()
27 {
28     List_push(list, test1);
29     mu_assert(List_last(list) == test1, "Nieprawidłowa ostatnia wartość.");
30
31     List_push(list, test2);
32     mu_assert(List_last(list) == test2, "Nieprawidłowa ostatnia wartość.");
33
34     List_push(list, test3);
35     mu_assert(List_last(list) == test3, "Nieprawidłowa ostatnia wartość.");
36     mu_assert(List_count(list) == 3, "Nieprawidłowa liczba elementów podczas
    ↳ tworzenia nowego na końcu listy.");
37
38     char *val = List_pop(list);
39     mu_assert(val == test3, "Nieprawidłowa wartość podczas usuwania elementu
    ↳ na końcu listy.");
40
41     val = List_pop(list);
42     mu_assert(val == test2, "Nieprawidłowa wartość podczas usuwania elementu
    ↳ na końcu listy.");
43
44     val = List_pop(list);
45     mu_assert(val == test1, "Nieprawidłowa wartość podczas usuwania elementu
    ↳ na końcu listy.");
46     mu_assert(List_count(list) == 0, "Nieprawidłowa liczba elementów
    ↳ po usunięciu elementu na końcu listy.");
47
48     return NULL;
49 }
50
51 char *test_unshift()
52 {
53     List_unshift(list, test1);
54     mu_assert(List_first(list) == test1, "Nieprawidłowa pierwsza wartość.");
55
56     List_unshift(list, test2);
57     mu_assert(List_first(list) == test2, "Nieprawidłowa pierwsza wartość.");
58
59     List_unshift(list, test3);
60     mu_assert(List_first(list) == test3, "Nieprawidłowa pierwsza wartość.");
61     mu_assert(List_count(list) == 3, "Nieprawidłowa liczba elementów po dodaniu
    ↳ nowego na początku listy.");
62
63     return NULL;
64 }
```

```
65
66 char *test_remove()
67 {
68     // Musimy przetestować jedynie usunięcie elementu w środku listy,
69     // ponieważ usunięcie na początku i końcu listy jest testowane w innych miejscach.
70
71     char *val = List_remove(list, list->first->next);
72     mu_assert(val == test2, "Nieprawidłowy usunięty element.");
73     mu_assert(List_count(list) == 2, "Nieprawidłowa liczba elementów
    ↳po usunięciu elementu.");
74     mu_assert(List_first(list) == test3, "Nieprawidłowy pierwszy element
    ↳po usunięciu.");
75     mu_assert(List_last(list) == test1, "Nieprawidłowy ostatni element
    ↳po usunięciu.");
76
77     return NULL;
78 }
79
80 char *test_shift()
81 {
82     mu_assert(List_count(list) != 0, "Nieprawidłowa liczba elementów przed
    ↳usunięciem pierwszego na liście.");
83
84     char *val = List_shift(list);
85     mu_assert(val == test3, "Nieprawidłowa wartość podczas usuwania pierwszego
    ↳elementu na liście.");
86
87     val = List_shift(list);
88     mu_assert(val == test1, "Nieprawidłowa wartość podczas usuwania pierwszego
    ↳elementu na liście.");
89     mu_assert(List_count(list) == 0, "Nieprawidłowa liczba elementów
    ↳po usunięciu pierwszego na liście.");
90
91     return NULL;
92 }
93
94 char *all_tests()
95 {
96     mu_suite_start();
97
98     mu_run_test(test_create);
99     mu_run_test(test_push_pop);
100    mu_run_test(test_unshift);
101    mu_run_test(test_remove);
102    mu_run_test(test_shift);
103    mu_run_test(test_destroy);
104
105    return NULL;
106 }
107
108 RUN_TESTS(all_tests);
```

Przedstawiony powyżej test po prostu wykonuje każdą operację i sprawdza, czy przebiega ona zgodnie z oczekiwaniami. Zastosowałem tutaj pewne uproszczenie i utworzyłem zaledwie jedną listę (`List *list`) dla całego programu, a następnie przeprowadziłem na niej testy. Dzięki temu uniknąłem konieczności tworzenia listy dla każdego testu. To jednak może oznaczać, że niektóre testy zostały zaliczone ze względu na sposób wykonania poprzedniego. W takim przypadku staram się, aby każdy test zachował listę bez zmian lub wykorzystywał wyniki otrzymane w poprzednim teście.

Co powinieneś zobaczyć?

Jeżeli wszystko zrobisz prawidłowo, po kompilacji i uruchomieniu testów jednostkowych powinieneś otrzymać następujące dane wyjściowe.

Sesja dla ćwiczenia 32.:

```
$ make
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG -fPIC -c -o\
    src/lcthw/list.o src/lcthw/list.c
ar rcs build/liblcthw.a src/lcthw/list.o
ranlib build/liblcthw.a
cc -shared -o build/liblcthw.so src/lcthw/list.o
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG build/liblcthw.a
    tests/list_tests.c -o tests/list_tests
sh ./tests/runtests.sh
Wykonywanie testów jednostkowych:
----
WYKONYWANIE: ./tests/list_tests
WSZYSTKIE TESTY ZOSTAŁY ŻALICZONE
Liczba wykonanych testów: 6
tests/list_tests PASS
$
```

Upewnij się o wykonaniu sześciu testów, o przeprowadzeniu kompilacji bez żadnych ostrzeżeń i błędów, a także o utworzeniu plików `build/liblcthw.a` i `build/liblcthw.so`.

Jak można usprawnić kod?

Zamiast pokazywać, jak zepsuć kod, tym razem zaproponuję różne możliwości jego usprawnienia.

- Działanie funkcji `List_clear_destroy()` może być znacznie efektywniejsze dzięki wykorzystaniu makro `LIST_FOREACH()` i wykonaniu obu wywołań `free()` wewnątrz jednej pętli.
- Możesz dodać asercje dla warunków początkowych, aby program nie otrzymywał wartości `NULL` dla parametrów `List *list`.

- Możesz dodać inwarianty sprawdzające, czy zawartość listy zawsze jest prawidłowa. Na przykład wartość `count` nigdy nie jest mniejsza niż 0, zatem możesz sprawdzić, czy wartość `count` jest większa niż 0, a następnie czy `first` ma wartość inną niż `NULL`.
- Możesz dodać dokumentację do pliku nagłówkowego w postaci komentarzy — przed każdą strukturą, funkcją i każdym makrem — opisujących przeznaczenie danego fragmentu kodu.

Powyższe usprawnienia są przykładem omówionych wcześniej w książce praktyk programowania defensywnego, które zwiększają niezawodność kodu i poprawiają jego użyteczność. Powinieneś wprowadzić wymienione usprawnienia, a następnie poszukać jak najwięcej kolejnych, aby w maksymalnym stopniu usprawnić ten kod.

Zadania dodatkowe

- Poszukaj informacji dodatkowych o listach jedno- i dwukierunkowej, a także o tym, w jakich sytuacjach są preferowane poszczególne listy.
- Poszukaj informacji o ograniczeniach listy dwukierunkowej. Na przykład wprowadzenie tego rodzaju lista jest efektywna podczas wstawiania i usuwania elementów, ale jednocześnie jest bardzo wolna w trakcie iteracji przez wszystkie elementy.
- Jakie operacje zostały pominięte w ćwiczeniu, które według Ciebie mogą być użyteczne? Wybrane przykłady to kopiowanie, łączenie i dzielenie. Zaimplementuj te operacje oraz utwórz dla nich testy jednostkowe.

Skorowidz

A

algorytmy
 ciągu tekstowego, 270
 listy dwukierunkowej,
 212
 struktury Hashmap, 262
 alokacja
 danych, 101
 pamięci, 96
 stosu, 102
 analiza wyników, 279
 ANSI C, 14
 APR, Apache Portable
 Runtime, 296
 ASCII, 79
 automatyczne testowanie,
 188
 automatyzowanie, 171
 awarie, 169

B

baza danych, 302
 biblioteka, 182, 200
 Better String Library,
 249
 bstrlib, 248, 302
 biblioteki
 Apache Portable
 Runtime, 297
 współdzielone, 183
 binarne drzewo
 poszukiwań, 282
 blok case, 145
 błędy, 139, 168
 BMH, Boyer-Moore-
 Horspool, 270
 bstrlib, 248
 budowa biblioteki, 200
 bufor cykliczny, 334

C

ciągi tekstowe, 66, 70,
 246, 270
 CRUD, create, read, update,
 delete, 364

D

dane
 statystyczne, 324, 364
 dane
 wejściowe, 148, 164
 wyjściowe, 32, 53, 148
 debugger, 36
 GDB, 36
 LLDD, 37
 debugowanie, 112, 122,
 124
 deklaracja wyprzedzająca,
 79
 devpkg, 296
 docelowe wersje programu,
 177
 dokumentacja, 170
 dokumentowanie założeń,
 170
 drzewo trójkowe, 346
 dynamiczne
 tablice, 227
 wczytywanie biblioteki,
 183

E

edytor tekstu, 21
 elementy globalne, 134

F

format wskaźnika funkcji,
 106
 formatowanie danych
 wyjściowych, 32

funkcja, 78

 bchare(), 263
 BSTree_delete(), 289
 BSTree_get(), 289
 BSTree_getnode(), 289
 BSTree_node_delete(),
 290
 BSTree_set(), 289
 bubble_sort(), 110
 calloc(), 199
 copy(), 168
 die(), 100, 109
 duffs_device(), 145
 free(), 199
 fscanf(), 149
 fuzz(), 294
 get_age(), 136
 HashMap_create(), 256
 HashMap_find_bucket(),
 256
 insert(), 347
 List_destroy(), 207
 main(), 79, 92, 314
 malloc(), 90, 92
 normal_copy(), 145
 perror(), 100
 Person_create(), 92
 Person_destroy(), 92
 Person_print(), 92
 print_size(), 136
 printf(), 32
 qsort(), 232
 radix_sort(), 241, 242
 RadixMap_add(), 241
 RadixMap_create(), 241
 RadixMap_delete(), 241
 RadixMap_destroy(), 241
 RadixMap_find(), 241
 RadixMap_sort(), 241,
 242
 read_scan(), 157
 realloc(), 199
 safecopy(), 166, 247
 Scan_find(), 280

funkcja

scanf(), 149, 157
 search(), 347
 search_index(), 347
 set_age(), 136
 sort(), 260
 Stats_create(), 327
 Stats_dump(), 328
 Stats_mean(), 328
 Stats_recreate(), 327
 Stats_sample(), 328
 Stats_stddev(), 328
 strcpy(), 104
 strdup(), 92
 String_find(), 271
 StringScanner_scan(), 271
 test_sorting(), 110
 traverse(), 347
 TSTree_insert(), 350
 update_ratio(), 136
 zeds_device(), 145

funkcje

bazy danych, 302
 biblioteki Better String Library, 249
 o zmiennej liczbie argumentów, 154
 poleceń programu, 309
 powłoki, 305
 wejścia-wyjścia, 151

fuzzing, 151

G

GDB, 36, 122, 123
 gniazdo serwera, 363

H

hash, 256
 Hashmap, 250

I

IDE, 22
 implementacja algorytmów sortowania, 215

listy dwukierunkowej, 204
 odchylenia standardowego, 325
 RadixMap, 235
 inicjalizacja pętli for, 84
 struktury, 95
 inkrementacja, 84
 instalacja, 180

K

klient TCP/IP, 340
 kod netclient, 340
 źródłowy, 24
 kolejka, 318
 kompilacja, 28 biblioteki, 185
 kompilator, 24
 konstrukcja else-if, 56
 if, 56
 switch, 48, 62
 kontrola przepływu, 126
 konwersja ciągów tekstowych, 101
 typu, 127
 kopiowanie prototypów struktur, 101
 kubelek, 268
 kwalifikatory typów, 127

L

linkowanie, 182
 lista dwukierunkowa, 200, 202, 212
 operatorów, 41
 typów, 128
 LLDB, 37
 logfind, 160

M

magazyn danych, 135
 Makefile

docelowe wersje programu, 177
 instalacja, 180
 nagłówkek, 176
 sprawdzenie, 180
 testy jednostkowe, 178
 makra, 118 debugowania, 112, 113
 makro check(), 119, 120
 debug(), 122, 123
 LIST_FOREACH(), 210
 log_err():, 119
 mechanizm Duffa, 142
 minifunkcja, 137
 modyfikacja pliku Makefile, 340
 modyfikatory typu, 126
 myślenie logiczne, 172

N

narzędzie make, 28
 Valgrind, 123
 nauka na pamięć, 40, 46
 nawiasy klamrowe, 57
 niezdefiniowane zachowanie, 194, 196

O

obsługa błędów, 112
 makr, 118
 odchylenie standardowe, 324
 operacja delete(), 283
 get(), 283
 operacje CRUD, 364
 porządkujące, 180
 operatory, 40, 129
 arytmetyczne, 41
 bitowe, 42, 131
 boolowskie, 131
 danych, 43, 130
 logiczne, 42, 131

matematyczne, 130
 przypisania, 43, 131
 relacji, 42
 różne, 43

P

pętla

for, 74
 while, 60

plik, 148

bstree.c, 284
 bstree.h, 282
 bstree_tests.c, 291
 bstrlib.c, 248
 commands.c, 310
 commands.h, 309
 darray.c, 224
 darray.h, 220
 darray_algos.c, 230
 darray_algos_tests.c,
 231
 darray_tests.c, 222
 db.c, 302
 dbg.h, 113, 115
 devpkg.c, 314
 ex1.c, 24
 ex10.c, 62
 ex11.c, 66
 ex12.c, 70
 ex13.c, 74
 ex14.c, 78
 ex15.c, 82
 ex16.c, 90
 ex17.c, 96
 ex18.c, 107
 ex19.c, 115
 ex2.1.mak, 29
 ex22.c, 134, 135
 ex22.h, 134
 ex22_main.c, 136
 ex23.c, 142
 ex24.c, 148
 ex25.c, 154
 ex27.c, 165
 ex29.c, 184
 ex3.c, 32
 ex30.Makefile.diff, 191
 ex35.c, 234

ex36.c, 246
 ex36.diff, 248
 ex7.c, 52
 ex8.c, 56
 ex9.c, 60
 hashmap.c, 252, 269
 hashmap.h, 250
 hashmap_algos.c, 262
 hashmap_algos_tests.c,
 264
 hashmap_tests.c, 257
 libex29.c, 183
 libex29.so, 187
 libex29_tests.c, 189
 limits.h, 128
 list.c, 204
 list.h, 202
 list_algos.c, 215
 list_algos.h, 215
 list_algos_tests.c, 213
 list_tests.c, 207
 Makefile, 28, 138, 174,
 299
 minunit.h, 188
 netclient.c, 340
 queue_tests.c, 319
 radixmap.c, 235
 radixmap.h, 233
 radixmap_tests.c, 238
 ringbuffer.c, 335
 ringbuffer.h, 334
 runtests.sh, 179, 193
 shell.c, 306
 shell.h, 305
 stack_tests.c, 318
 stats.c, 326
 stats.h, 326
 stats_tests.c, 328
 stdint.h, 128
 stdio.h, 32
 string_algos.c, 271
 string_algos.h, 270
 string_algos_tests.c, 274
 tstree.c, 347
 tstree.h, 346
 tstree_tests.c, 352
 urlot.c, 356
 pliki kodu źródłowego, 300
 podręcznik systemowy, 34

polecenia

GDB, 36
 LLDD, 37

polecenie

@echo, 181
 #define, 100
 include, 91
 make, 33

POSIX, 338

pośrednie pliki Makefile, 174

powłoka, 305

prewencja, 170

program install, 180

programowanie

defensywne, 16, 162
 kreatywne, 162

projekt

devpkg, 296
 logfind, 160

przechowywanie

danych statystycznych,
 368

przetwarzanie

skomplikowanych
 argumentów, 101

przypadki

niezdefiniowanego
 zachowania, 196

psucie kodu, 26

R

rodzaje magazynu danych,
 135

router URL, 356

routing danych

statystycznych, 366

S

sekcja

#define debug(), 114
 #ifdef, 114

serwer

danych statystycznych,
 364

sieciowy, 362

silnik dla danych

statystycznych, 324

składnia

- C, 46
- pętli for, 74
- struktur, 47
- słowo kluczowe, 46
 - const, 137
 - extern, 135
 - static, 135
 - typedef, 106
- sortowanie, 230
 - bąbelkowe, 212
 - pozycyjne, 233
 - przez scalanie, 212
- specyfikacja logfind, 160
- sprawdzenie, 180
 - warunku, 84
- stała, 137
- standard ANSI C, 14
- statystyka, 324
- sterta, 96, 103
- stos, 96, 103, 134, 139, 318
- stosowanie
 - ciągów tekstowych, 246
 - tablicy dynamicznej, 227
- strategia debugowania, 124
- strategie programisty
 - defensywnego, 164
- struktura, 47, 90, 94
 - FILE, 101
 - Hashmap, 250, 262, 351
 - HashmapNode, 251
 - Person, 91
 - projektu, 174
 - TSTree, 354
- struktury
 - danych, 200
 - kontroli, 132
 - o wielkości ustalonej, 100

Ś

- średnia, 324
 - odchyień
 - standardowych, 331
 - średnich, 331

T

- tabela przeskoków, 62
- tablica, 66
 - bajtów, 70
 - ciągów tekstowych, 74, 76
 - dynamiczna, 220, 251
- techniki debugowania, 122
- testowanie
 - zautomatyzowane, 188
- testy, 207
 - jednostkowe, 178, 189, 213, 257, 337
- tworzenie
 - funkcji, 78
 - wskaźnika, 83
- typy, 52
 - bibliotek, 182
 - danych, 126

U

- unia RMElement, 235
- unie, 234
- unikanie błędów, 168
- upraszczanie, 171
- URL, 356
- usprawnianie serwera, 370
- usunięcie programu, 34
- użycie
 - debugera, 36
 - funkcji, 78
 - narzędzia make, 28
 - wskaźników, 86

W

- wartość NULL, 101
- węzeł, 290
- wielkość
 - typu, 128
 - wskaźnika, 87
- Windows, 21
- wskaźnik, 82, 85
 - ptr, 87
- wskaźniki
 - do funkcji, 106
 - do struktury, 90
- wrażenia boolowskie, 60
- wyszukiwanie, 230
 - binarne, 233, 241

Z

- zagnieżdżone struktury
 - wskaźników, 101
- zakres, 134, 139
- zgłaszanie błędów, 100
- zmienne, 52
- znaki ASCII, 79

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA



Helion SA

ZROZUM C, PROGRAMUJ STARANNIE — DOBRZE DZIAŁAĆ MOŻE TYLKO DOBRY KOD!

Istnieje wiele nowoczesnych języków programowania, które pozwalają na szybkie wdrożenie i pracę. Takim językiem na pewno nie jest C. Niektóre jego cechy bardzo utrudniają tworzenie bezpiecznego i bezawaryjnego kodu. Warto więc dogłębnie poznać C — przy niezwykle prostej składni i niewielkich wymaganiach sprzętowych ma potężne możliwości!

Niniejsza książka jest świetnym podręcznikiem dla początkujących programistów. Nauczysz się C, wykonując 52 sprytnie skonstruowane zadania zilustrowane kodem i specjalnie opracowanymi klipami wideo. Autor kładzie duży nacisk na dogłębną analizę tworzonego kodu — zmusza do zrozumienia znaczenia każdej linii programu, do koncentracji i dokładności. Zachęca też do praktykowania tzw. programowania defensywnego, dzięki któremu możliwe jest podniesienie jakości i bezpieczeństwa tworzonego oprogramowania. Wartościowym elementem książki są wskazówki, jak zepsuć napisany kod, a następnie go zabezpieczyć. To znacznie ułatwia unikanie wielu poważnych, często spotykanych błędów.

Najistotniejsze zagadnienia poruszone w książce:

- Podstawowa składnia C
- Konfiguracja środowiska programistycznego, kompilacja kodu, pliki Makefile i linkery
- Kontrola przebiegu działania programu, alokacja pamięci
- Operacje wejścia-wyjścia i pliki, stosy oraz kolejki
- Usuwanie błędów, programowanie defensywne i automatyczne testowanie
- Eliminacja błędu przepełnienia stosu, niedozwolonego dostępu do pamięci itd.
- Hakowanie własnego kodu utworzonego w C



ZED A. SHAW — ceniony programista, którego najbardziej znanym projektem jest serwer WWW Mongrel dla aplikacji Ruby. Jest również autorem wielu artykułów i książek dotyczących technik programowania, jak *Learn Python the Hard Way* i *Learn Ruby the Hard Way*, które cieszą się ogromną popularnością — są czytane i dyskutowane przez miliony czytelników na całym świecie. Shaw posiada niezwykłą umiejętność pisanie o trudnych tematach w sposób przystępny, żywy i interesujący. Jak nikt inny potrafi objaśniać najtrudniejsze zagadnienia informatyki.

 **Addison-Wesley**

Helion

44778 numer katalogowy

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>

ISBN 978-83-283-2545-6



9 788328 325456

cena: 67,00 zł

Informatyka w najlepszym wydaniu

siegnij po **WIĘCEJ**



KOD KORZYŚCI