

## IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

## 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

# Język C. Wskaźniki. Vademecum profesjonalisty

Autor: Kenneth A. Reek

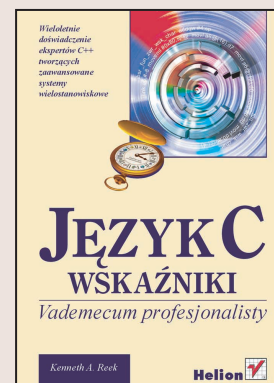
Tłumaczenie: Paweł Gonera

ISBN: 83-7361-198-3

Tytuł oryginału: [Pointers on C](#)

Format: B5, stron: 544

[Przykłady na ftp: 55 kB](#)



Książka „Język C. Wskaźniki. Vademecum profesjonalisty” przeznaczona jest dla zaawansowanych studentów i profesjonalistów, zapewniając obszerne źródło informacji dla tych, którzy potrzebują dogłębnego omówienia języka C. Dokładne wyjaśnienie podstaw oraz przegląd zaawansowanych funkcji pozwala programistom skorzystać z siły wskaźników w języku C. Dokładny opis idiomów programowych oraz gruntowna dyskusja zaawansowanych tematów powoduje, że książka jest nieocenionym podręcznikiem i informatorem dla studentów i zawodowych programistów.

- Zawiera wszystko, co jest niezbędne do dogłębnego poznania języka C
- Dokładnie opisuje wskaźniki, ich składnię, techniki efektywnego użycia oraz często stosowane idiomy programistyczne, w których występują wskaźniki
- Porównuje różne metody implementacji często stosowanych abstrakcyjnych typów danych
- Zawiera wskazówki na temat efektywności, przenośności i zagadnień inżynierii programowania, jak również ostrzeżenia o często popełnianych błędach
- Oferuje prosty, konwersacyjny styl, jasno opisujący trudne tematy, zawiera wiele ilustracji i diagramów pomagających z wizualizacji skomplikowanych zagadnień
- Opisuje wszystkie funkcje z biblioteki standardowej C.

O Autorze: **Kenneth A. Reek** jest Profesorem informatyki w Rochester Institute of Technology i doświadczonym programistą, który pracował w wielu firmach jako konsultant. Książka ta powstała po dziewięciu latach prowadzenia seminariów z programowania w C. Profesor Reek prowadził zajęcia na podstawowym i średnim poziomie z systemów operacyjnych, telekomunikacji, sieci komputerowych, analizy algorytmów i systemów przetwarzających.



# Spis treści

<b>Przedmowa</b> .....	<b>13</b>
<b>Rozdział 1. Szybki start</b> .....	<b>19</b>
1.1. Wstęp .....	19
1.1.1. Odstępy i komentarze .....	22
1.1.2. Dyrektywy preprocesora .....	23
1.1.3. Funkcja main .....	24
1.1.4. Funkcja czytaj_zakresy_kolumn .....	27
1.1.5. Funkcja przekształc .....	32
1.2. Inne możliwości .....	35
1.3. Kompilacja .....	35
1.4. Podsumowanie .....	35
1.5. Podsumowanie ostrzeżeń .....	36
1.6. Podsumowanie wskazówek .....	36
1.7. Pytania .....	37
1.8. Ćwiczenia .....	37
<b>Rozdział 2. Podstawowe pojęcia</b> .....	<b>39</b>
2.1. Środowiska .....	39
2.1.1. Translacja .....	39
2.1.2. Wykonanie .....	41
2.2. Zasady leksykalne .....	42
2.2.1. Znaki .....	42
2.2.2. Komentarze .....	44
2.2.3. Dowolna postać kodu źródłowego .....	44
2.2.4. Identyfikatory .....	45
2.2.5. Postać programu .....	45
2.3. Styl programowania .....	46
2.4. Podsumowanie .....	47
2.5. Podsumowanie ostrzeżeń .....	48
2.6. Podsumowanie wskazówek .....	48
2.7. Pytania .....	48
2.8. Ćwiczenia .....	50
<b>Rozdział 3. Dane</b> .....	<b>51</b>
3.1. Podstawowe typy danych .....	51
3.1.1. Rodzina liczb całkowitych .....	51
3.1.2. Typy zmiennoprzecinkowe .....	55
3.1.3. Wskaźniki .....	56
3.2. Podstawowe deklaracje .....	58
3.2.1. Inicjalizacja .....	59
3.2.2. Deklarowanie prostych tablic .....	59

3.2.3. Deklaracja wskaźników .....	60
3.2.4. Niejawne deklaracje .....	61
3.3. Typedef .....	61
3.4. Stałe .....	62
3.5. Zasięg .....	63
3.5.1. Zasięg ograniczony do bloku .....	64
3.5.2. Zasięg ograniczony do pliku .....	65
3.5.3. Zasięg ograniczony do prototypu .....	65
3.5.4. Zasięg ograniczony do funkcji .....	65
3.6. Sposób konsolidacji .....	66
3.7. Klasa zapisu .....	67
3.7.1. Inicjalizacja .....	69
3.8. Słowo kluczowe static .....	69
3.9. Przykład zasięgu, rodzaju łączenia i klas zapisu .....	70
3.10. Podsumowanie .....	72
3.11. Podsumowanie ostrzeżeń .....	72
3.12. Podsumowanie wskazówek .....	73
3.13. Pytania .....	73
<b>Rozdział 4. Instrukcje .....</b>	<b>77</b>
4.1. Instrukcja pusta .....	77
4.2. Instrukcja wyrażenia .....	77
4.3. Instrukcja bloku .....	78
4.4. Instrukcja if .....	79
4.5. Instrukcja while .....	80
4.5.1. Instrukcje break i continue .....	80
4.5.2. Działanie pętli while .....	81
4.6. Instrukcja for .....	82
4.6.1. Wykonanie pętli for .....	82
4.7. Instrukcja do .....	83
4.8. Instrukcja switch .....	84
4.8.1. Instrukcja break w switch .....	85
4.8.2. Wartości domyślne .....	86
4.8.3. Wykonanie instrukcji switch .....	86
4.9. Instrukcja goto .....	87
4.10. Podsumowanie .....	89
4.11. Podsumowanie ostrzeżeń .....	90
4.12. Podsumowanie wskazówek .....	90
4.13. Pytania .....	90
4.14. Ćwiczenia .....	92
<b>Rozdział 5. Operatory i wyrażenia .....</b>	<b>95</b>
5.1. Operatory .....	95
5.1.1. Arytmetyka .....	95
5.1.2. Przesunięcia .....	95
5.1.3. Operatory bitowe .....	97
5.1.4. Przypisania .....	98
5.1.5. Operatory jednoargumentowe .....	101
5.1.6. Operatory relacyjne .....	103
5.1.7. Operatory logiczne .....	104
5.1.8. Operator warunkowy .....	105
5.1.9. Operator przecinka .....	105
5.1.10. Indeks, wywołanie funkcji i element struktury .....	107
5.2. Wartości logiczne .....	108
5.3. L-wartości i R-wartości .....	109

5.4. Obliczanie wyrażeń.....	110
5.4.1. Niejawna konwersja typów .....	110
5.4.2. Konwersje arytmetyczne .....	111
5.4.3. Właściwości operatorów.....	112
5.4.4. Priorytety i kolejność wykonywania .....	114
5.5. Podsumowanie.....	116
5.6. Podsumowanie ostrzeżeń.....	118
5.7. Podsumowanie wskazówek .....	118
5.8. Pytania .....	118
5.9. Ćwiczenia.....	121

## **Rozdział 6. Wskaźniki ..... 123**

6.1. Pamięć i adresy .....	123
6.1.1. Adresy i zawartość.....	124
6.2. Wartości i ich typy.....	124
6.3. Zawartość zmiennej wskaźnikowej .....	126
6.4. Operator dereferencji .....	126
6.5. Niezainicjowane i nieprawidłowe wskaźniki .....	128
6.6. Wskaźnik NULL.....	129
6.7. Wskaźniki, dereferencja i L-wartości .....	130
6.8. Wskaźniki, dereferencja i zmienne .....	131
6.9. Stałe wskaźnikowe.....	131
6.10. Wskaźniki do wskaźników .....	132
6.11. Operacje na wskaźnikach.....	133
6.12. Przykłady .....	138
6.13. Arytmetyka wskaźników .....	141
6.13.1. Operacje arytmetyczne .....	142
6.13.2. Operacje relacyjne .....	145
6.14. Podsumowanie .....	146
6.15. Podsumowanie ostrzeżeń .....	147
6.16. Podsumowanie wskazówek .....	147
6.17. Pytania .....	148
6.18. Ćwiczenia.....	150

## **Rozdział 7. Funkcje..... 153**

7.1. Definicja funkcji .....	153
7.1.1. Instrukcja return.....	154
7.2. Deklaracje funkcji.....	155
7.2.1. Prototypy .....	155
7.2.2. Domyślne założenia dla funkcji .....	158
7.3. Argumenty funkcji.....	158
7.4. ATD i czarne skrzynki.....	162
7.5. Rekurencja .....	164
7.5.1. Śledzenie funkcji rekurencyjnych .....	166
7.5.2. Rekurencja a iteracja .....	169
7.6. Zmienna lista argumentów.....	172
7.6.1. Makra stdarg .....	173
7.6.2. Ograniczenia zmiennej listy argumentów .....	174
7.7. Podsumowanie.....	175
7.8. Podsumowanie ostrzeżeń.....	176
7.9. Podsumowanie wskazówek .....	176
7.10. Pytania .....	177
7.11. Ćwiczenia.....	178

<b>Rozdział 8. Tablice</b>	<b>179</b>
8.1. Tablice jednowymiarowe	179
8.1.1. Nazwy tablic	179
8.1.2. Indeksy	180
8.1.3. Wskaźniki kontra indeksy	183
8.1.4. Efektywność wskaźników	184
8.1.5. Tablice i wskaźniki	190
8.1.6. Nazwy tablic i argumenty funkcji	190
8.1.7. Deklarowanie parametrów tablicowych	192
8.1.8. Inicjalizacja	192
8.1.9. Niekompletne inicjalizacje	193
8.1.10. Automatyczne określanie wielkości tablicy	194
8.1.11. Inicjalizacja tablicy znaków	194
8.2. Tablice wielowymiarowe	194
8.2.1. Kolejność zapisu	195
8.2.2. Nazwy tablic	196
8.2.3. Indeksy	197
8.2.4. Wskaźniki na tablice	199
8.2.5. Tablice wielowymiarowe jako argumenty funkcji	200
8.2.6. Inicjalizacja	201
8.2.7. Automatyczne określanie wielkości tablic	204
8.3. Tablice wskaźników	204
8.4. Podsumowanie	207
8.5. Podsumowanie ostrzeżeń	208
8.6. Podsumowanie wskazówek	209
8.7. Pytania	209
8.8. Ćwiczenia	213
<b>Rozdział 9. Ciągi, znaki i bajty</b>	<b>219</b>
9.1. Ciągi znaków — podstawy	219
9.2. Długość ciągu	219
9.3. Nieograniczone funkcje operujące na ciągach	221
9.3.1. Kopiowanie ciągów	221
9.3.2. Łączenie ciągów	222
9.3.3. Wartość zwracana przez funkcję	222
9.3.4. Porównywanie ciągów	223
9.4. Funkcje operujące na ciągach o ograniczonej długości	223
9.5. Proste wyszukiwanie w ciągach	224
9.5.1. Wyszukiwanie znaków	225
9.5.2. Wyszukiwanie dowolnego z kilku znaków	225
9.5.3. Wyszukiwanie podciągu	225
9.6. Zaawansowane wyszukiwanie ciągów	227
9.6.1. Wyszukiwanie przedrostków w ciągu	227
9.6.2. Wyszukiwanie tokenów	227
9.7. Komunikaty dotyczące błędów	229
9.8. Operacje na znakach	229
9.8.1. Klasyfikacja znaków	229
9.8.2. Transformacje znaków	230
9.9. Operacje na pamięci	230
9.10. Podsumowanie	232
9.11. Podsumowanie ostrzeżeń	233
9.12. Podsumowanie wskazówek	233
9.13. Pytania	234
9.14. Ćwiczenia	234

---

<b>Rozdział 10. Struktury i unie .....</b>	<b>241</b>
10.1. Podstawy struktur .....	241
10.1.1. Deklaracje struktur .....	242
10.1.2. Składniki struktury .....	243
10.1.3. Bezpośredni dostęp do składników .....	244
10.1.4. Pośredni dostęp do składników .....	244
10.1.5. Struktury odwołujące się do samych siebie .....	245
10.1.6. Niekompletne deklaracje .....	246
10.1.7. Inicjalizacja struktur .....	246
10.2. Struktury, wskaźniki i składniki .....	247
10.2.1. Odwołanie poprzez wskaźnik .....	248
10.2.2. Odwoływanie się do struktury .....	248
10.2.3. Odwoływanie się do składników struktury .....	249
10.2.4. Odwoływanie się do zagnieżdżonej struktury .....	251
10.2.5. Odwoływanie się do składnika wskaźnikowego .....	251
10.3. Sposób zapisu struktur .....	253
10.4. Struktury jako argumenty funkcji .....	254
10.5. Pola bitowe .....	257
10.6. Unie .....	260
10.6.1. Rekordy z wariantami .....	261
10.6.2. Inicjalizacja unii .....	262
10.7. Podsumowanie .....	263
10.8. Podsumowanie ostrzeżeń .....	264
10.9. Podsumowanie wskazówek .....	264
10.10. Pytania .....	264
10.11. Ćwiczenia .....	267
<b>Rozdział 11. Dynamiczne przydzielanie pamięci .....</b>	<b>271</b>
11.1. Dlaczego korzystamy z dynamicznego przydzielania pamięci .....	271
11.2. Funkcje malloc i free .....	271
11.3. Funkcje calloc i realloc .....	273
11.4. Wykorzystanie dynamicznie przydzielanej pamięci .....	273
11.5. Częste błędy pamięci dynamicznej .....	274
11.5.1. Wycieki pamięci .....	277
11.6. Przykłady przydzielania pamięci .....	277
11.7. Podsumowanie .....	283
11.8. Podsumowanie ostrzeżeń .....	283
11.9. Podsumowanie wskazówek .....	283
11.10. Pytania .....	284
11.11. Ćwiczenia .....	285
<b>Rozdział 12. Wykorzystanie struktur i wskaźników .....</b>	<b>287</b>
12.1. Listy .....	287
12.2. Lista jednokierunkowa .....	287
12.2.1. Wstawianie węzłów do listy jednokierunkowej .....	288
12.2.2. Inne operacje na listach .....	297
12.3. Lista dwukierunkowa .....	298
12.3.1. Wstawianie do listy dwukierunkowej .....	298
12.3.2. Inne operacje na listach .....	306
12.4. Podsumowanie .....	307
12.5. Podsumowanie ostrzeżeń .....	307
12.6. Podsumowanie wskazówek .....	308
12.7. Pytania .....	308
12.8. Ćwiczenia .....	308

<b>Rozdział 13. Zaawansowane zagadnienia dotyczące wskaźników</b>	<b>311</b>
13.1. Więcej o wskaźnikach do wskaźników	311
13.2. Deklaracje zaawansowane	313
13.3. Wskaźniki do funkcji	315
13.3.1. Funkcje wywołania zwrotnego	316
13.3.2. Tablice skoków	319
13.4. Argumenty wiersza poleceń	321
13.4.1. Przekazywanie argumentów wiersza poleceń	321
13.4.2. Przetwarzanie argumentów wiersza poleceń	323
13.5. Literały ciągów znaków	326
13.6. Podsumowanie	328
13.7. Podsumowanie ostrzeżeń	329
13.8. Podsumowanie wskazówek	329
13.9. Pytania	330
13.10. Ćwiczenia	333
<b>Rozdział 14. Preprocesor</b>	<b>337</b>
14.1. Symbole predefiniowane	337
14.2. #define	337
14.2.1. Makra	339
14.2.2. Podstawianie za pomocą #define	340
14.2.3. Makra kontra funkcje	341
14.2.4. Argumenty makr z efektami ubocznymi	342
14.2.5. Konwencje nazewnictwa	343
14.2.6. #undef	344
14.2.7. Definicje z wiersza poleceń	345
14.3. Kompilacja warunkowa	345
14.3.1. #if defined	347
14.3.2. Dyrektywy zagnieżdżone	347
14.4. Dołączanie plików	348
14.4.1. Dołączanie biblioteczne	349
14.4.2. Dołączanie lokalne	349
14.4.3. Zagnieżdżone dołączanie plików	350
14.5. Inne dyrektywy	351
14.6. Podsumowanie	352
14.7. Podsumowanie ostrzeżeń	353
14.8. Podsumowanie wskazówek	354
14.9. Pytania	354
14.10. Ćwiczenia	356
<b>Rozdział 15. Funkcje wejścia-wyjścia</b>	<b>357</b>
15.1. Raportowanie błędów	357
15.2. Przerwanie działania	358
15.3. Standardowa biblioteka wejścia-wyjścia	358
15.4. Założenia wejścia-wyjścia ANSI	359
15.4.1. Strumienie	359
15.4.2. Struktury FILE	361
15.4.3. Standardowe stałe wejścia-wyjścia	361
15.5. Przegląd strumieniowego wejścia-wyjścia	362
15.6. Otwieranie strumieni	363
15.7. Zamykanie strumieni	365
15.8. Znakowe wejście-wyjście	366
15.8.1. Makra znakowego wejścia-wyjścia	367
15.8.2. Wycyfywanie operacji znakowego wejścia-wyjścia	368
15.9. niesformatowane wierszowe wejście-wyjście	369

15.10. Formatowane wierszowe wejście-wyjście.....	371
15.10.1. Rodzina scanf .....	371
15.10.2. Kody formatujące funkcji scanf .....	371
15.10.3. Rodzina printf.....	376
15.10.4. Kody formatujące printf .....	376
15.11. Binarne wejście-wyjście .....	380
15.12. Funkcje wyszukujące i opróżniające bufor.....	381
15.13. Zmiana buforowania .....	384
15.14. Funkcje obsługi błędów strumienia .....	385
15.15. Pliki tymczasowe .....	385
15.16. Funkcje do manipulacji plikami .....	386
15.17. Podsumowanie .....	386
15.18. Podsumowanie ostrzeżeń .....	388
15.19. Podsumowanie wskazówek .....	389
15.20. Pytania .....	389
15.21. Ćwiczenia.....	390

## **Rozdział 16. Biblioteka standardowa ..... 395**

16.1. Funkcje typu całkowitego .....	395
16.1.1. Arytmetyka <stdlib.h>.....	395
16.1.2. Liczby losowe <stdlib.h> .....	396
16.1.3. Konwersja ciągów znaków <stdlib.h> .....	397
16.2. Funkcje zmiennoprzecinkowe .....	398
16.2.1. Trygonometria <math.h>.....	399
16.2.2. Funkcje hiperboliczne <math.h>.....	399
16.2.3. Funkcje logarytmiczne i wykładnicze <math.h>.....	399
16.2.4. Reprezentacja zmiennoprzecinkowa <math.h>.....	400
16.2.5. Potęgowanie <math.h> .....	400
16.2.6. Podłoga, sufit, wartość bezwzględna i reszta <math.h>.....	400
16.2.7. Konwersja ciągów znaków <stdlib.h> .....	401
16.3. Funkcje daty i czasu.....	401
16.3.1. Czas procesora <time.h> .....	401
16.3.2. Data i godzina <time.h> .....	402
16.4. Skoki nielokalne <setjmp.h> .....	405
16.4.1. Przykład .....	406
16.4.2. Kiedy używać nielokalnych skoków .....	408
16.5. Sygnały .....	408
16.5.1. Nazwy sygnałów <signal.h> .....	408
16.5.2. Przetwarzanie sygnałów <signal.h>.....	410
16.5.3. Obsługa sygnałów.....	411
16.6. Drukowanie list zmiennych argumentów <stdarg.h> .....	413
16.7. Środowisko wykonania.....	413
16.7.1. Przerwanie działania <stdlib.h> .....	413
16.7.2. Asercje <assert.h>.....	414
16.7.3. Środowisko <stdlib.h>.....	415
16.7.4. Wykonywanie poleceń systemowych <stdlib.h> .....	415
16.8. Sortowanie i wyszukiwanie <stdlib.h>.....	415
16.9. Ustawienia międzynarodowe.....	418
16.9.1. Formatowanie numeryczne i monetarne <locale.h> .....	419
16.9.2. Ciągi znaków i ustawienia regionalne <string.h> .....	421
16.9.3. Efekty zmiany ustawień regionalnych.....	422
16.10. Podsumowanie .....	422
16.11. Podsumowanie ostrzeżeń .....	424
16.12. Podsumowanie wskazówek .....	424
16.13. Pytania .....	425
16.14. Ćwiczenia.....	426

<b>Rozdział 17. Klasyczne przykłady abstrakcyjnych typów danych .....</b>	<b>429</b>
17.1. Przydział pamięci.....	429
17.2. Stosy.....	430
17.2.1. Interfejs stosu.....	430
17.2.2. Implementacja stosu .....	430
17.3. Kolejki .....	438
17.3.1. Interfejs kolejki.....	439
17.3.2. Implementacja kolejki .....	440
17.4. Drzewa .....	444
17.4.1. Wstawianie wartości do uporządkowanego drzewa binarnego.....	445
17.4.2. Usuwanie wartości z uporządkowanego drzewa binarnego.....	445
17.4.3. Wyszukiwanie wartości w uporządkowanym drzewie binarnym .....	446
17.4.4. Przeglądanie drzewa.....	446
17.4.5. Interfejs uporządkowanego drzewa binarnego .....	448
17.4.6. Implementacja uporządkowanego drzewa binarnego.....	448
17.5. Usprawnienia implementacji .....	455
17.5.1. Utworzenie więcej niż jednego stosu .....	455
17.5.2. Wykorzystywanie więcej niż jednego typu .....	456
17.5.3. Konflikty nazw .....	457
17.5.4. Standardowe biblioteki ATD.....	457
17.6. Podsumowanie .....	460
17.7. Podsumowanie ostrzeżeń .....	461
17.8. Podsumowanie wskazówek .....	461
17.9. Pytania .....	462
17.10. Ćwiczenia.....	463
<b>Rozdział 18. Środowisko wykonania .....</b>	<b>465</b>
18.1. Określanie cech środowiska wykonania .....	465
18.1.1. Program testowy .....	465
18.1.2. Zmienne statyczne i inicjalizacja.....	468
18.1.3. Ramka stosu.....	469
18.1.4. Zmienne register .....	470
18.1.5. Długość identyfikatorów zewnętrznych .....	471
18.1.6. Określanie układu ramki stosu .....	472
18.1.7. Efekty uboczne wyrażeń.....	477
18.2. Współpraca z kodem asemblera .....	478
18.3. Efektywność działania .....	479
18.3.1. Poprawianie wydajności .....	480
18.4. Podsumowanie .....	482
18.5. Podsumowanie ostrzeżeń .....	482
18.6. Podsumowanie wskazówek .....	483
18.7. Pytania .....	483
18.8. Ćwiczenia.....	483
<b>Dodatek A Wybrane rozwiązania problemów .....</b>	<b>485</b>
<b>Dodatek B Bibliografia .....</b>	<b>525</b>
<b>Skorowidz .....</b>	<b>527</b>

## Rozdział 12.

# Wykorzystanie struktur i wskaźników

Łącząc ze sobą wskaźniki i struktury, można tworzyć bardzo efektywne struktury danych. W rozdziale tym przyjrzymy się kilku technikom wykorzystania struktur i wskaźników. Wiele czasu poświęcimy strukturze nazywanej listą — nie tylko dlatego, że jest bardzo użyteczna, ale również dlatego, że wiele z technik wykorzystywanych do manipulowania listą można również stosować do innych struktur danych.

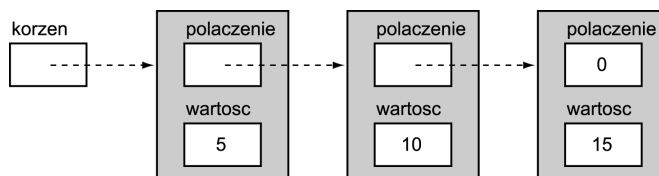
### 12.1. Listy

Jeżeli nie jesteś zaznajomiony z pojęciem listy, nakreślę krótkie wprowadzenie. *Lista* to zbiór niezależnych struktur (często nazywanych *węzłami*), zawierających dane. Poszczególne węzły na liście są połączone ze sobą za pomocą połączeń lub wskaźników. Program odwołuje się do węzłów listy podążając za wskaźnikami. Zwykle węzły są tworzone dynamicznie, choć czasami można spotkać się z listą skonstruowaną z elementów tablicy węzłów. Nawet w takim przypadku program przegląda listę podążając za wskaźnikami.

### 12.2. Lista jednokierunkowa

Na *liście jednokierunkowej* każdy węzeł zawiera wskaźnik do następnego węzła listy. Pole wskaźnikowe w ostatnim elemencie zawiera wartość NULL, wskazując, że na liście nie ma już więcej węzłów. Aby pamiętać, w którym miejscu lista się zaczyna, wykorzystywany jest wskaźnik nazywany *korzeniem* lub *głową* listy. Wskaźnik korzenia wskazuje na pierwszy element listy. Zwróć uwagę, że korzeń nie zawiera żadnych danych.

Poniżej przedstawiono diagram listy jednokierunkowej.



Węzły z tego przykładu są strukturami utworzonymi na podstawie następujących deklaracji.

```
typedef struct WEZEL {
    struct WEZEL *polaczenie;
    int          wartosc;
} Wezel;
```

W każdym z węzłów zapisywana jest jedna dana — liczba `int`. Przedstawiona lista zawiera trzy węzły. Jeżeli zaczniemy od korzenia i za wskaźnikiem podążymy do pierwszego węzła, możemy uzyskać dostęp do danych zapisanych w tym węźle. Podążając za wskaźnikiem, przechodzimy z pierwszego węzła do drugiego i uzyskujemy dostęp do jego danych. Na koniec następny wskaźnik przenosi nas do ostatniego węzła. Wartość zero została wykorzystana do pokazania wskaźnika `NULL`; oznacza on, że na liście nie ma już więcej węzłów.

Na diagramie węzły zostały umieszczone obok siebie w celu pokazania *logicznego* uporządkowania zapewnianego przez połączenia. W rzeczywistości węzły mogą być rozsiane po całej pamięci. Dla programu przetwarzającego taką listę nie ma żadnej różnicy, czy węzły sąsiadują ze sobą czy nie, ponieważ program zawsze korzysta z połączeń, aby przejść z jednego węzła do drugiego.

Lista jednokierunkowa może być przeglądana od początku do końca dzięki podążaniu za wskaźnikami, ale nie można jej przeglądać wstecz. Inaczej mówiąc, gdy program dojdzie do ostatniego węzła listy, jedyną możliwością cofnięcia się jest wykorzystanie wskaźnika korzenia. Oczywiście program może zapamiętać wskaźnik do bieżącego węzła przed przejściem do następnego węzła, może nawet zapamiętywać wskaźniki wskazujące na kilka kolejnych węzłów. Jednak lista jest strukturą dynamiczną i może zwiększyć się do kilkuset lub kilku tysięcy węzłów — nierealne jest więc zapamiętywanie wskaźników do wszystkich poprzednich węzłów w liście.

Węzły przedstawionej listy są połączone w taki sposób, że wartości w węzłach są uporządkowane w kolejności rosnącej. Takie uporządkowanie jest ważne w przypadku niektórych aplikacji, na przykład dla porządkowania spotkań względem czasu. Możliwe jest również tworzenie listy nieuporządkowanej, jeżeli aplikacja nie wymaga uporządkowania.

### 12.2.1. Wstawianie węzłów do listy jednokierunkowej

W jaki sposób należy wstawiać nowy węzeł do uporządkowanej listy jednokierunkowej? Załóżmy, że mamy nową wartość — 12 — i musimy wstawić ją do przedstawionej powyżej listy. Pod względem koncepcyjnym zadanie jest proste: zaczynamy od początku listy i podążamy za wskaźnikami, aż znajdziemy pierwszy węzeł o wartości większej niż 12. Wstawiamy wówczas nową wartość do listy przed znalezionym węzłem.

W praktyce algorytm jest bardziej interesujący. Przeglądamy listę i zatrzymujemy się, gdy osiągniemy węzeł z wartością 15 — pierwszy, który jest większy od 12. Wiemy, że nowa wartość musi być dodana do listy bezpośrednio przed tym węzłem, ale przed realizacją tej operacji zmienione musi zostać pole wskaźnikowe *poprzedniego* węzła; tymczasem my nie możemy się cofnąć. Rozwiązaniem jest zapamiętywanie wskaźnika do poprzedniego węzła.

Teraz możemy zaprojektować funkcję wstawiającą węzeł do uporządkowanej listy jednokierunkowej. W listingu 12.1 przedstawiamy pierwsze podejście do tego problemu.

**Listing 12.1.** Wstawianie węzła do uporządkowanej listy jednokierunkowej — pierwsza próba (wstaw1.c)

```
/*
** Wstawianie węzła do uporządkowanej listy jednokierunkowej. Argumentami
** są: wskaźnik do pierwszego węzła listy i wartość do wstawienia.
*/
#include <stdlib.h>
#include <stdio.h>
#include "wezel_poj.h"

#define FALSE 0
#define TRUE 1

int
wstaw_lista_poj( Wezel *biez, int nowa_wart )
{
    Wezel *poprz;
    Wezel *nowy;

    /*
    ** Szukanie właściwego miejsca poprzez przeglądanie listy
    ** do momentu znalezienia węzła, którego wartość jest większa
    ** lub równa nowej wartości.
    */
    while( biez->wartosc < nowa_wart ){
        poprz = biez;
        biez = biez->polaczenie;
    }

    /*
    ** Utworzenie nowego węzła i zapamiętanie w nim nowej wartości.
    ** W przypadku niepowodzenia zwracana jest wartość FALSE.
    */
    nowy = (Wezel *)malloc( sizeof( Wezel ) );
    if( nowy == NULL )
        return FALSE;
    nowy->wartosc = nowa_wart;

    /*
    ** Wstawienie nowego węzła do listy i zwrócenie wartości TRUE.
    */
    nowy->polaczenie = biez;
    poprz->polaczenie = nowy;
    return TRUE;
}
```

Funkcję tę wywołujemy w następujący sposób:

```
wynik = wstaw_lista_poj( korzen, 12 );
```

Prześledźmy kod, aby sprawdzić, czy wartość 12 jest prawidłowo wstawiana do listy. Na początek funkcja jest wywoływana z wartością zmiennej `korzen`, która jest wskaźnikiem na pierwszy węzeł listy. Poniżej przedstawiono stan listy na początku wykonywania funkcji:

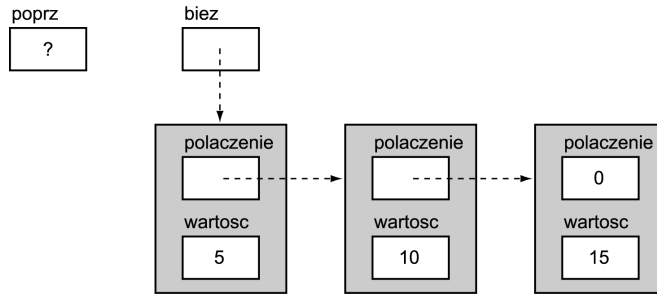
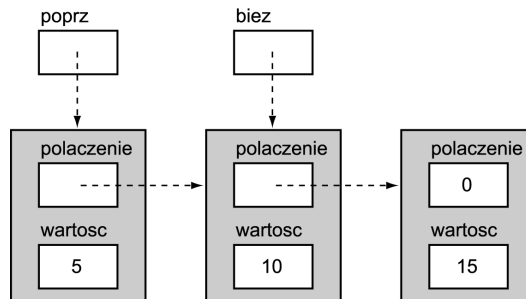
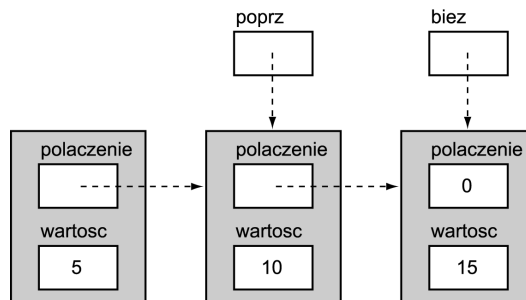


Diagram ten nie zawiera zmiennej korzen, ponieważ funkcja nie może się do niej odwoływać. Kopia jej wartości jest przekazana do funkcji jako parametr biez, ale funkcja nie może odwoływać się do korzen. Teraz biez->wartosc jest równe 5, czyli mniej niż 12 — wykonywane jest więc ciało pętli, w którym przesuwane są nasze wskaźniki.

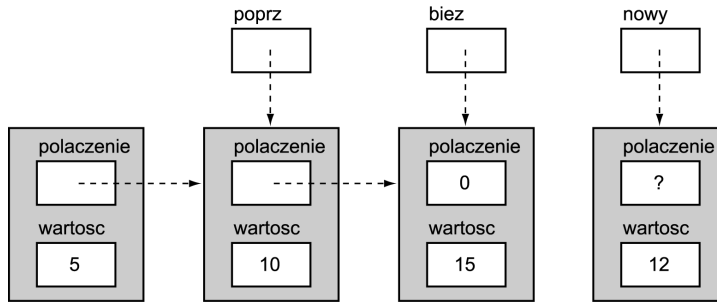


W chwili obecnej biez->wartosc jest równe 10, więc ciało pętli jest wykonywane kolejny raz, co daje następujący wynik:



Teraz biez->wartosc jest większe od 12, więc pętla jest przerywana.

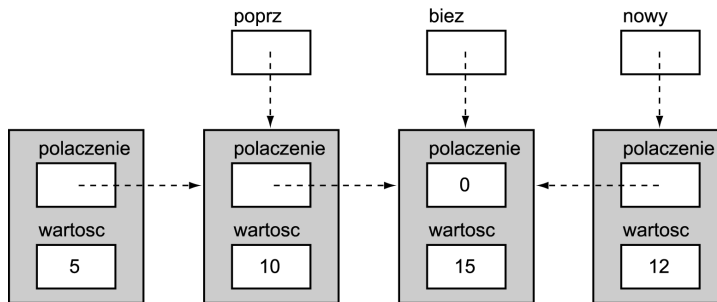
W tym momencie ważny staje się wskaźnik poprz, ponieważ wskazuje na węzeł, który musi zostać zmieniony, aby możliwe było dodanie nowej wartości. Na początek należy jednak utworzyć nowy węzeł i zapisać w nim wartość. Następny diagram pokazuje stan listy po skopiowaniu wartości do nowego węzła.



Włączenie nowego węzła do listy wymaga wykonania dwóch kroków. Pierwszym jest:

```
nowy->polaczenie = biez;
```

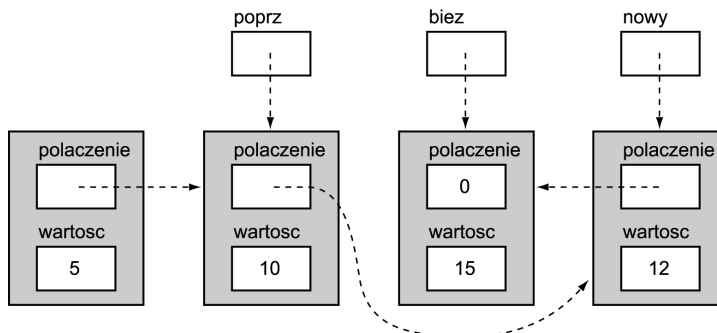
co powoduje, że nowy węzeł wskazuje na węzeł będący następnym węzłem na liście — pierwszym, którego wartość jest większa niż 12. Po tej operacji lista wygląda następująco:



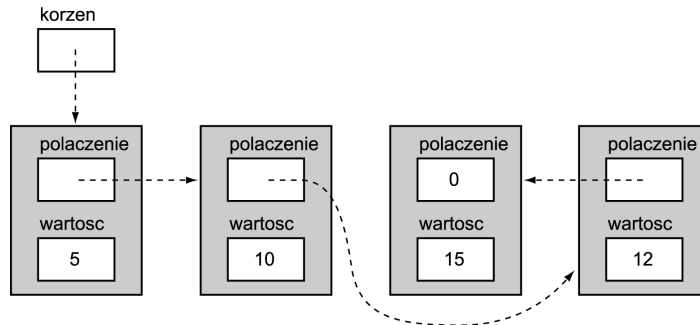
Drugim krokiem jest zmiana wskaźnika w poprzednim węźle — ostatnim o wartości mniejszej niż 12 — tak, aby wskazywał na nowy węzeł. Operacja ta jest wykonywana przez instrukcję:

```
poprz->polaczenie = nowy;
```

Wynikiem wykonania tego kroku jest:



Funkcja kończy się, pozostawiając listę w następującym stanie:



Rozpoczynając od wskaźnika `korzen` i podążając za wskaźnikami, możemy sprawdzić, że nowy węzeł został prawidłowo wstawiony.

## Usuwanie błędów z funkcji wstawiającej



Niestety funkcja wstawiająca jest nieprawidłowa. Spróbuj wstawić do listy wartość 20, a zobaczysz, na czym polega problem; pętla `while` przejdzie poza koniec listy i wykona dereferencję wskaźnika `NULL`. Aby rozwiązać ten problem, musimy przed obliczeniem `biez->wartosc` sprawdzać, czy wskaźnik `biez` jest różny od `NULL`.

```
while( biez != NULL && biez->wartosc < wartosc ) {
```

Następny problem jest poważniejszy. Prześledź zachowanie funkcji podczas wstawiania do listy wartości 3. Co się dzieje?

Aby wstawić element na początek listy, funkcja musi zmienić wskaźnik `korzen`. Funkcja jednak nie może odwoływać się do wartości `korzen`. Najprostszym sposobem naprawienia tego problemu jest zmiana zmiennej `korzen` na zmienną globalną, dzięki czemu funkcja wstawiająca mogłaby ją modyfikować. Niestety podejście to jest *najgorszym* sposobem rozwiązania tego problemu, ponieważ funkcja mogłaby być używana tylko dla jednej listy.

Lepszym rozwiązaniem jest przekazywanie wskaźnika do `korzen` jako argumentu. Funkcja mogłaby wykonywać dereferencję zarówno w celu pobrania wartości `korzen` (wskaźnika do pierwszego elementu listy), jak i zapisywania nowej wartości wskaźnika. Jaki powinien być typ takiego parametru? Zmienna `korzen` jest wskaźnikiem na `Wezel`, więc parametr powinien mieć typ `**Wezel`: wskaźnik do wskaźnika na `Wezel`. Funkcja zamieszczona w listingu 12.2 zawiera te modyfikacje. Funkcję tę należy wywoływać w następujący sposób:

```
wynik = wstaw_lista_poj( &korzen, 12 );
```

### Listing 12.2. Wstawianie węzła do uporządkowanej listy jednokierunkowej — druga próba (`wstaw2.c`)

```
/*
** Wstawianie do uporządkowanej listy jednokierunkowej. Argumentami
** są: wskaźnik do wskaźnika korzenia listy i wartość do wstawienia.
*/
```

```
#include <stdlib.h>
#include <stdio.h>
#include "wezel_poj.h"

#define FALSE 0
#define TRUE 1

int
wstaw_lista_poj( Wezel **korzen_wsk, int nowa_wart )
{
    Wezel *biez;
    Wezel *poprz;
    Wezel *nowy;

    /*
    ** Pobranie wskaźnika do pierwszego węzła.
    */
    biez = *korzen_wsk;
    poprz = NULL;

    /*
    ** Szukanie właściwego miejsca poprzez przeglądanie listy
    ** do momentu znalezienia węzła, którego wartość jest większa
    ** lub równa nowej wartości.
    */
    while( biez != NULL && biez->wartosc < nowa_wart ){
        poprz = biez;
        biez = biez->polaczenie;
    }

    /*
    ** Utworzenie nowego węzła i zapamiętanie w nim nowej wartości.
    ** W przypadku niepowodzenia zwracana jest wartość FALSE.
    */
    nowy = (Wezel *)malloc( sizeof( Wezel ) );
    if( nowy == NULL )
        return FALSE;
    nowy->wartosc = nowa_wart;

    /*
    ** Wstawienie nowego węzła do listy i zwrócenie wartości TRUE.
    */
    nowy->polaczenie = biez;
    if( poprz == NULL )
        *korzen_wsk = nowy;
    else
        poprz->polaczenie = nowy;
    return TRUE;
}
```

Ta druga wersja zawiera kilka dodatkowych instrukcji:

```
poprz = NULL;
```

Instrukcja ta jest niezbędna, ponieważ później możemy sprawdzić, czy nowa wartość powinna być pierwszym węzłem na liście.

```
biez = *korzen_wsk;
```

Wykorzystano tu dereferencję na argumencie będącym wskaźnikiem do korzenia, dzięki czemu pobraliśmy wartość wskaźnika `korzen` — wskaźnika do pierwszego węzła listy. Na koniec funkcji dodane zostały instrukcje:

```
if( poprz == NULL )
    *korzen_wsk = nowy;
else
    poprz->polaczenie = nowy;
```

Sprawdzają one, czy nowa wartość powinna być włączona na początku listy. Jeżeli tak, za pomocą dereferencji modyfikujemy wskaźnik `korzen` tak, aby pokazywał nowy węzeł.

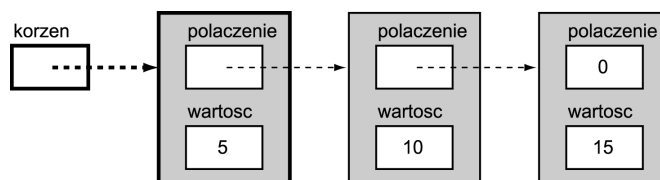
Funkcja działa prawidłowo i w wielu językach programowania jest najlepszym możliwym rozwiązaniem. Jednak w C możemy zapisać ją jeszcze lepiej, ponieważ język ten posiada możliwość odczytu adresu (wskaźnika) istniejącego obiektu.

## Optymalizacja funkcji wstawiającej

Może się wydawać, że wstawianie węzła na początku listy *musi być* specjalnym przypadkiem. Wskaźnik, który musi być zmieniony w celu wstawienia pierwszego węzła, jest wskaźnikiem korzenia. Dla pozostałych węzłów korygowanym wskaźnikiem jest pole łączące z poprzedniego węzła. Te pozornie różne operacje są w rzeczywistości tym samym.

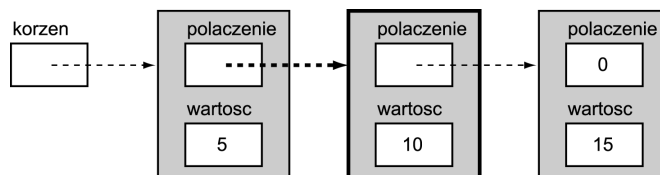
Kluczem do wyeliminowania tego specjalnego przypadku jest fakt, że każdy węzeł listy posiada wskaźnik, który na niego wskazuje. Dla pierwszego węzła jest to wskaźnik `korzen`, a dla pozostałych węzłów — pole łączące z poprzedniego węzła. Najważniejsze jest, że istnieje wskaźnik wskazujący na węzeł. To, czy wskaźnik jest zapisany w węźle czy nie, jest nieistotne.

Spójrzmy jeszcze raz na listę, aby wyjaśnić ten wniosek. Mamy tutaj pierwszy węzeł i odpowiadający mu wskaźnik.



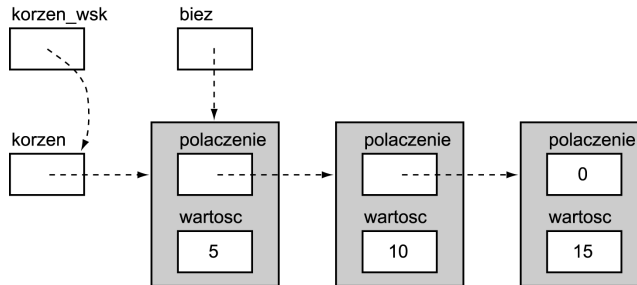
Jeżeli nowa wartość jest wstawiana przed pierwszym węzłem, musi być zmieniony odpowiedni wskaźnik.

Tutaj mamy drugi węzeł i jego wskaźnik.



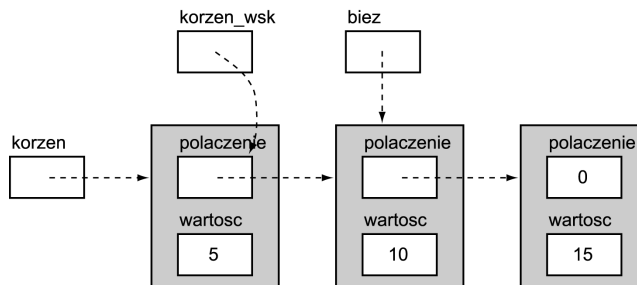
Jeżeli nowa wartość jest wstawiana przed drugim węzłem, to *ten* wskaźnik musi być zmieniony. Zwróć uwagę, że zajmujemy się tylko wskaźnikami; zawartość węzła jest w tym momencie nieistotna. Ten sam wzór powtarza się dla każdego węzła listy.

Teraz spójrzmy na zmienioną funkcję, gdy rozpoczyna ona działanie. Poniżej przedstawiono wszystkie zmienne bezpośrednio po pierwszym przypisaniu.



Mamy wskaźnik do bieżącego węzła i wskaźnik do połączenia wskazującego na bieżący węzeł. Nie potrzebujemy niczego więcej! Jeżeli wartość w bieżącym węźle jest większa niż nowa wartość, wskaźnik *korzen\_wsk* wskazuje nam na pole połączeniowe, które musi zostać zmienione, aby możliwe było włączenie nowego węzła do listy. Jeżeli wstawienie w dowolne inne miejsce listy może być zrealizowane tak samo, specjalny przypadek znika. Kluczem jest pokazana wcześniej relacja wskaźnik-węzeł.

Przesuwając się do następnego węzła, zapamiętujemy — zamiast wskaźnika na poprzedni węzeł — wskaźnik na *połączenie*, które wskazuje na następny węzeł. Najłatwiej narysować to na diagramie:



Zwróć uwagę, że *korzen\_wsk* nie wskazuje na węzeł, lecz na pole łączące w węźle. Ma to kluczowe znaczenie dla uproszczenia funkcji wstawiającej, ale zależy od możliwości uzyskania adresu pola łączącego z bieżącego węzła. W języku C operacja taka jest bardzo prosta — realizuje ją wyrażenie `&biez->polaczenie`. W listingu 12.3 zamieszczona została ostateczna wersja naszej funkcji wstawiającej. Parametr *korzen\_wsk* jest teraz nazwany *polacz\_wsk*, ponieważ wskazuje teraz na wiele różnych połączeń, nie tylko na korzeń. Nie potrzebujemy już zmiennej *poprzedni*, ponieważ nasz wskaźnik na połączenie pozwala na zlokalizowanie pola łączącego, wymagającego zmodyfikowania. Specjalny przypadek na końcu funkcji już nie występuje, ponieważ zawsze mamy wskaźnik do pola połączeniowego, które należy zmodyfikować — zmieniamy wskaźnik korzenia w identyczny sposób, co pole łączące w węźle. Na koniec wreszcie dodana została deklaracja rejestru dla zmiennej wskaźnikowej, co powinno poprawić efektywność wynikowego kodu.

**Listing 12.3.** Wstawianie węzła do uporządkowanej listy jednokierunkowej — wersja ostateczna (wstaw3.c)

```

/*
** Wstawianie węzła do uporządkowanej listy jednokierunkowej. Argumentami
** są: wskaźnik do wskaźnika korzenia listy i wartość do wstawienia.
*/
#include <stdlib.h>
#include <stdio.h>
#include "wezel_poj.h"

#define FALSE 0
#define TRUE 1

int
wstaw_lista_poj( register Wezel **polacz_wsk, int nowa_wart )
{
    register Wezel *biez;
    register Wezel *nowy;

    /*
    ** Szukanie właściwego miejsca poprzez przeglądanie listy
    ** do momentu znalezienia węzła, którego wartość jest
    ** większa lub równa nowej wartości.
    */
    while( ( biez = *polacz_wsk ) != NULL &&
           biez->wartosc < nowa_wart )
        polacz_wsk = &biez->polaczenie;

    /*
    ** Utworzenie nowego węzła i zapamiętanie w nim nowej wartości.
    ** W przypadku niepowodzenia zwracana jest wartość FALSE.
    */
    nowy = (Wezel *)malloc( sizeof( Wezel ) );
    if( nowy == NULL )
        return FALSE;
    nowy->wartosc = nowa_wart;

    /*
    ** Wstawienie nowego węzła do listy i zwrócenie wartości TRUE.
    */
    nowy->polaczenie = biez;
    *polacz_wsk = nowy;
    return TRUE;
}

```

W pętli while w tej końcowej wersji zastosowana została sztuczka z wbudowanym przypisaniem do zmiennej `biez`. Poniżej przedstawiamy realizującą to samo zadanie, choć nieco dłuższą pętlę.

```

/*
** Wyszukiwanie właściwego miejsca.
*/
biez = *polacz_wsk;
while( biez != NULL && biez->wartosc < nowa_wart ) {
    polacz_wsk = &biez->polaczenie;
    biez = *polacz_wsk;
}

```

Na początku wskaźnik `biez` ustawiany jest na pierwszy węzeł na liście. Instrukcja `while` sprawdza, czy osiągnęliśmy koniec pętli. Jeżeli nie, sprawdza, czy jesteśmy w odpowiednim miejscu do wstawienia węzła. Jeżeli nie, wykonywane jest ciało pętli, w którym wskaźnik `polacz_wsk` jest tak zmieniany, aby wskazywał na pole łączące w bieżącym węźle, a wskaźnik `biez` jest przesuwany do następnego węzła.

Fakt, że ostatnia instrukcja w ciele pętli jest identyczna z instrukcją umieszczoną przed pętlą, pozwala „uproszczyć” program poprzez wbudowanie w wyrażenie `while` przypisania do wskaźnika `biez`. W wyniku usunięcia nadmiarowego przypisania otrzymujemy bardziej złożoną, ale i bardziej zwartą pętlę.



Wyeliminowanie specjalnego przypadku uprościło tę funkcję. Wystąpiły tu dwa czynniki umożliwiające tę modyfikację. Pierwszym jest nasza zdolność do prawidłowej interpretacji problemu. Jeżeli nie zidentyfikujemy wspólnych cech w pozornie różnych operacjach, będziemy zmuszeni tworzyć dodatkowy kod dla specjalnych przypadków. Często wiedza ta jest zdobywana po dłuższym czasie pracy z takimi strukturami i po ich dogłębnym poznaniu. Drugim czynnikiem jest to, że język C zapewnia właściwe narzędzia, pozwalające wykorzystać te wspólne cechy.

Ulepszona funkcja korzysta z zawartych w języku C możliwości odczytywania adresu istniejącego obiektu. Podobnie jak wiele funkcji C, możliwość ta jest jednocześnie potężna i niebezpieczna. W językach Modula i Pascal nie ma operatora „adres czegoś”, więc jedyne wskaźniki, jakie w nich występują, to wskaźniki uzyskane poprzez dynamiczny przydział pamięci. Niemożliwe jest uzyskanie wskaźnika do zwykłej zmiennej lub nawet do pola dynamicznie utworzonej struktury. Niedozwolona jest arytmetyka wskaźników, nie ma również sensu rzutowanie wskaźników jednego typu na inne. Ograniczenia te są wprowadzone w celu zabezpieczenia przed popełnianiem przez programistów takich błędów, jak indeksowanie poza obszarem tablicy czy generowanie wskaźników niewłaściwego typu.



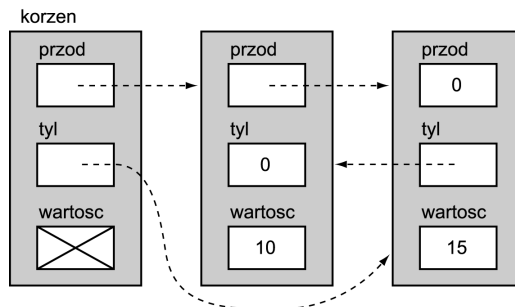
W języku C występuje o wiele mniej ograniczeń dotyczących operacji na wskaźnikach — dzięki temu mogliśmy usprawnić naszą funkcję. Z drugiej strony programista C musi o wiele bardziej uważać, aby uniknąć pomyłek przy stosowaniu wskaźników. Filozofię języka Pascal dotyczącą stosowania wskaźników można podsumować zdaniem: „Możesz się skaleczyć siekierą, więc jej nie dostaniesz”. Filozofia C to raczej: „Oto siekiera. Masz tu też inne rodzaje siekier. Powodzenia!”. Dysponując takimi narzędziami, programista C może łatwiej popaść w kłopoty niż programista Pascala. Dobry programista C może jednak tworzyć mniejsze, efektywniejsze i łatwiejsze do utrzymania programy niż jego koledzy korzystający z Modula czy Pascala. Z tego powodu język C jest tak popularny w branży i dlatego doświadczeni programiści C są tak poszukiwani.

## 12.2.2. Inne operacje na listach

Aby lista jednokierunkowa była naprawdę użyteczna, niezbędne jest wykorzystywanie większej liczby operacji, na przykład wyszukiwania i usuwania. Jednak algorytmy tych operacji są proste i łatwe do zaimplementowania z wykorzystaniem technik pokazanych w funkcji wstawiającej. Napisanie tych funkcji pozostawiam Czytelnikowi jako ćwiczenie.

## 12.3. Lista dwukierunkowa

Alternatywą dla listy jednokierunkowej jest *lista dwukierunkowa*. W przypadku tej listy każdy węzeł ma dwa wskaźniki — jeden wskazuje na następny węzeł listy, a drugi na poprzedni węzeł. Wskaźnik do poprzedniego węzła pozwala na przeglądanie listy w dowolnym kierunku. Na poniższym diagramie przedstawiono listę dwukierunkową.



Deklaracja typu węzła jest następująca:

```
typedef struct WEZEL {
    struct WEZEL *przod;
    struct WEZEL *tyl;
    int wartosc;
} Wezel;
```

Korzeń ma teraz dwa wskaźniki: jeden wskazuje na pierwszy węzeł listy, drugi natomiast na ostatni. Te dwa wskaźniki pozwalają nam na przeglądanie listy z dowolnego jej końca.

Możemy zadeklarować dwa wskaźniki korzenia jako osobne zmienne, ale musielibyśmy przekazywać oba te wskaźniki do funkcji wstawiającej. Wygodniej jest zadeklarować cały węzeł jako korzeń i nie wykorzystywać pół wartości. W naszym przykładzie technika ta powoduje niepotrzebne zajęcie pamięci tylko na jedną liczbę `int`. Osobne wskaźniki mogą być lepsze w przypadku list zawierających duże pola wartości. Można również wykorzystać pole wartości węzła korzenia do przechowywania danych o liście, na przykład informacji o ilości węzłów w liście.

Pole `przod` węzła korzenia wskazuje na pierwszy węzeł na liście, natomiast pole `tyl` wskazuje na ostatni węzeł. Jeżeli lista jest pusta, oba te wskaźniki przyjmują wartość `NULL`. Pole `tyl` pierwszego węzła listy oraz pole `przod` ostatniego węzła również mają wartość `NULL`. Na liście uporządkowanej węzły są łączone w rosnącym porządku pól `wartosc`.

### 12.3.1. Wstawianie do listy dwukierunkowej

Tym razem zaprojektujemy funkcję wstawiającą wartość do uporządkowanej listy dwukierunkowej. Funkcja `wstaw_lista_podw` wymaga dwóch argumentów: wskaźnika do węzła korzenia oraz wartości całkowitej.

Zaprojektowana wcześniej funkcja wstawiająca węzeł do listy pojedynczej pozwala na wstawianie duplikatów do listy. W niektórych aplikacjach prawidłowym działaniem jest niedodawanie duplikatów. Funkcja `wstaw_lista_podw` wstawi nową wartość tylko wtedy, gdy nie ma jej jeszcze na liście.

Funkcję tę zaprojektujemy w sposób bardziej zdyscyplinowany. W czasie wstawiania wartości do listy mogą wystąpić cztery przypadki:

1. Wartość będzie musiała być wstawiona w środek listy.
2. Wartość będzie musiała być wstawiona na początek listy.
3. Wartość będzie musiała być wstawiona na koniec listy.
4. Wartość będzie musiała być wstawiona zarówno na początek, jak i na koniec listy (czyli wstawienie do pustej listy).

W każdym z przypadków zmodyfikowane będą musiały być cztery wskaźniki.

- W przypadkach (1) i (2) pole `przed` nowego węzła musi wskazywać na następny węzeł listy, natomiast pole `tyl` następnego węzła musi wskazywać na nowy węzeł. W przypadkach (3) i (4) pole `przed` nowego węzła musi mieć wartość `NULL`, a pole `tyl` węzła korzenia musi wskazywać na nowy węzeł.
- W przypadkach (1) i (3) pole `tyl` nowego węzła musi wskazywać na poprzedni węzeł listy, a pole `przed` poprzedniego węzła musi wskazywać na nowy węzeł. W przypadkach (2) i (4) pole `tyl` nowego węzła musi mieć wartość `NULL`, a pole `przed` węzła korzenia musi wskazywać na nowy węzeł.

Jeżeli ten opis wydaje się niejasny, pomoc powinna prosta implementacja przedstawiona w listingu 12.4.

**Listing 12.4.** Prosta funkcja wstawiająca węzeł do listy dwukierunkowej (`dwu_wstaw1.c`)

```
/*
** Wstawienie węzła do listy dwukierunkowej. korzen_wsk jest wskaźnikiem
** na węzeł korzenia, a wartosc jest nową, wstawianą wartością.
** Zwraca: 0, jeżeli wartość jest już na liście, -1, jeżeli nie ma miejsca w pamięci
** na utworzenie nowego węzła lub 1 -- po udanym wstawieniu węzła.
*/
#include <stdlib.h>
#include <stdio.h>
#include "wezel_podw.h"

int
wstaw_lista_podw( Wezel *korzen_wsk, int wartosc )
{
    Wezel    *biez;
    Wezel    *nast;
    Wezel    *nowyWezel;

    /*
    ** Sprawdzenie, czy wartosc znajduje się na liście; jeżeli tak,
    ** następuje koniec funkcji. W przeciwnym przypadku następuje
    ** przydzielenie nowego węzła wskazującego na wartosc
    */
}
```

```
** ("nowyWezel" będzie na niego wskazywał).
** "biez" wskazuje na węzeł, przed który ma być wstawiona nowa
** wartosc. "nast" wskazuje na węzeł umieszczony po nim.
*/
for( biez = korzen_wsk; (nast = biez->przod) != NULL; biez = nast ){
    if( nast->wartosc == wartosc )
        return 0;
    if( nast->wartosc > wartosc )
        break;
}
nowyWezel = (Wezel *)malloc( sizeof( Wezel ) );
if( nowyWezel == NULL )
    return -1;
nowyWezel->wartosc = wartosc;

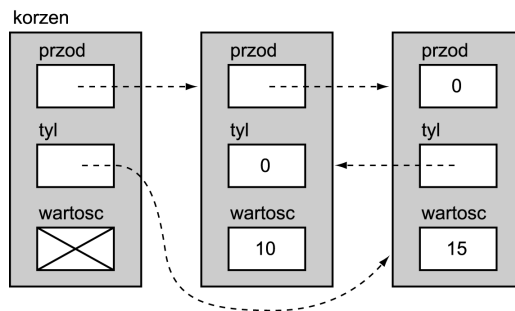
/*
** Dodanie nowego węzła do listy.
*/
if( nast != NULL ){
    /*
    ** Przypadek 1 lub 2: nie jest to koniec listy.
    */
    if( biez != korzen_wsk ){ /* Przypadek 1: nie na początku */
        nowyWezel->przod = nast;
        biez->przod = nowyWezel;
        nowyWezel->tyl = biez;
        nast->tyl = nowyWezel;
    }
    else { /* Przypadek 2: na początku. */
        nowyWezel->przod = nast;
        korzen_wsk->przod = nowyWezel;
        nowyWezel->tyl = NULL;
        nast->tyl = nowyWezel;
    }
}
else {
    /*
    ** Przypadek 3 lub 4: na końcu listy.
    */
    if( biez != korzen_wsk ){ /* Przypadek 3: nie na początku. */
        nowyWezel->przod = NULL;
        biez->przod = nowyWezel;
        nowyWezel->tyl = biez;
        korzen_wsk->tyl = nowyWezel;
    }
    else { /* Przypadek 4: na początku. */
        nowyWezel->przod = NULL;
        korzen_wsk->przod = nowyWezel;
        nowyWezel->tyl = NULL;
        korzen_wsk->tyl = nowyWezel;
    }
}
return 1;
}
```

Funkcja rozpoczyna się od przypisania wskaźnikowi `biez` wskaźnika na węzeł korzenia. Wskaźnik `nast` zawsze wskazuje na węzeł po `biez`; wskaźniki te będą przesuwane razem aż do znalezienia miejsca, w którym pomiędzy nie będzie wstawiony nowy węzeł. Pętla `for` sprawdza wartości w węźle `nast`, określając, czy została osiągnięta odpowiednia pozycja.

Jeżeli na liście zostanie znaleziona nowa wartość, funkcja po prostu kończy swoje działanie. W przeciwnym przypadku pętla kończy się na końcu listy lub gdy osiągnięta zostanie właściwa pozycja do wstawiania. W obu tych przypadkach nowy węzeł powinien być wstawiony po węźle wskazywanym przez `biez`. Zwróć uwagę, że pamięć przeznaczona na nowy węzeł nie jest przydzielana do czasu, gdy zostanie sprawdzone, że wartość powinna być dodana do listy.

Przydzielenie nowego węzła na początku może powodować potencjalny wyciek pamięci w przypadku znalezienia podwójnych wartości.

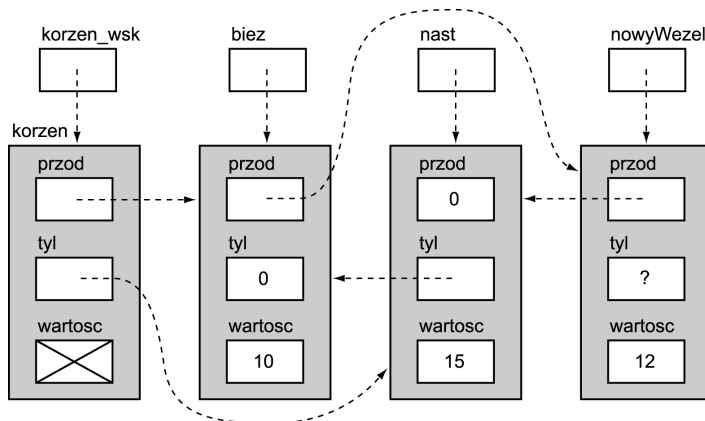
Przedstawione cztery przypadki są implementowane osobno. Prześledźmy pierwszy, dodając do listy wartość 12. Poniższy diagram pokazuje stan naszych zmiennych bezpośrednio po przerwaniu pętli `for`.



Teraz przydzielany jest pierwszy węzeł. Po wykonaniu instrukcji:

```
nowyWzegl->przod = nast;
biez->przod = nowyWzegl;
```

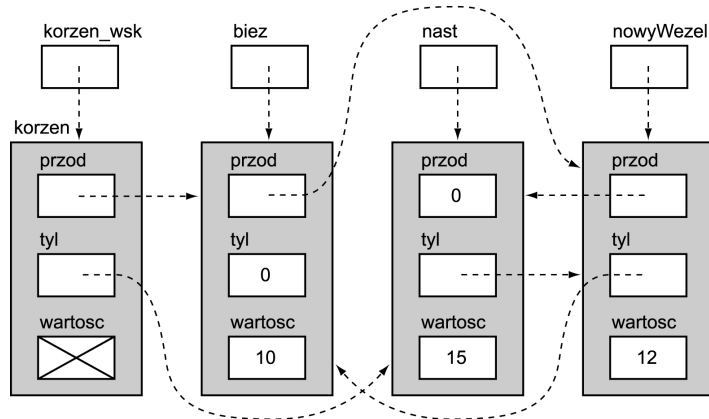
lista będzie wyglądać następująco:



### Kolejne instrukcje

```
nowyWezel->tyl = biez;
nast->tyl = nowyWezel;
```

pozwalają zakończyć dodawanie nowej wartości do listy:



Przeanalizuj kod, aby sprawdzić, czy pozostałe przypadki działają prawidłowo.

### Upraszczanie funkcji wstawiającej



Spostrzegawczy programista zauważy wiele podobieństw w grupach instrukcji umieszczonych w zagnieżdżonych instrukcjach `if`, a *dobry* programista będzie zaniepokojony tymi powtórzeniami. Spróbujmy więc wyeliminować te powtórzenia, korzystając z dwóch technik. Pierwsza jest nazywana *przebudową instrukcji* (ang. *factoring*) i została pokazana na następującym przykładzie:

```
if( x == 3 ) {
    i = 1;
    dalsze instrukcje;
    j = 2;
}
else {
    i = 1;
    inne instrukcje;
    j = 2;
}
```

Zwróć uwagę, że instrukcje `i = 1;` oraz `j = 2;` są wykonywane niezależnie od tego, czy wyrażenie `x==3` jest spełnione czy nie. Wykonanie `i = 1;` przed `if` nie wpłynie na test `x==3`, więc oba przypisania mogą zostać wyjęte z instrukcji `if`, tworząc prostsze, ale identycznie działające instrukcje:

```
i = 1;
if( x == 3 )
    dalsze instrukcje;
else
    inne instrukcje;
j = 2;
```



Należy uważać, aby nie wyciągać przed `if` instrukcji, które zmieniają wynik testu. Na przykład we fragmencie:

```
if( x == 3 ) {
    x = 0;
    dalsze instrukcje;
}
else {
    x = 0;
    inne instrukcje;
}
```

instrukcja `x = 0;` nie może być wyjęta przed test, ponieważ zmieniłaby wynik porównania.

Przebudowa najbardziej zagnieżdżonej instrukcji `if` z listingu 12.4 daje w wyniku kod zamieszczony w listingu 12.5. Porównaj ten kod z poprzednią funkcją i przekonaj się, że działa on identycznie.

**Listing 12.5.** Przebudowa instrukcji funkcji wstawiającej węzeł do listy dwukierunkowej (`dwu_wstaw2.c`)

```
/*
** Dodanie nowego węzła do listy.
*/
if( nast != NULL ){
    /*
    ** Przypadek 1 lub 2: nie jest to koniec listy.
    */
    nowyWezel->przod = nast;
    if( biez != korzen_wsk ){ /* Przypadek 1: nie na początku. */
        biez->przod = nowyWezel;
        nowyWezel->tyl = biez;
    }
    else { /* Przypadek 2: na początku. */
        korzen_wsk->przod = nowyWezel;
        nowyWezel->tyl = NULL;
    }
    nast->tyl = nowyWezel;
}
else {
    /*
    ** Przypadek 3 lub 4: na końcu listy.
    */
    nowyWezel->przod = NULL;
    if( biez != korzen_wsk ){ /* Przypadek 3: nie na początku. */
        biez->przod = nowyWezel;
        nowyWezel->tyl = biez;
    }
    else { /* Przypadek 4: na początku. */
        korzen_wsk->przod = nowyWezel;
        nowyWezel->tyl = NULL;
    }
    korzen_wsk->tyl = nowyWezel;
}
```

Drugą technikę upraszczania najłatwiej pokazać na przykładzie:

```
if( wskaznik != NULL )
    pole = wskaznik;
else
    pole = NULL;
```

Chcemy tutaj nadać zmiennej wartość wskaznik lub wartość NULL, jeżeli wskaznik na nic nie wskazuje. Ale spójrz na instrukcję:

```
pole = wskaznik;
```

Jeżeli wskaznik ma wartość różną od NULL, pole — tak jak poprzednio — otrzymuje kopię tej wartości. Ale jeżeli wskaznik ma wartość NULL, pole otrzymuje kopię wartości NULL ze zmiennej wskaznik, co daje ten sam efekt, co przypisanie stałej NULL. Wyrażenie to wykonuje to samo zadanie i oczywiście jest prostsze.

Kluczem do zastosowania tej techniki w kodzie z listingu 12.5 jest zidentyfikowanie instrukcji wykonujących te same zadania, chociaż wyglądają one inaczej, a następnie ich odpowiednie przepisanie. Jako pierwsze możemy zmienić pierwsze instrukcje przypadków (3) i (4):

```
nowyWezel->przod = nast;
```

ponieważ instrukcja `if` właśnie sprawdziła, że `nast == NULL`. Zmiana ta powoduje, że obie strony warunku `if` w pierwszej instrukcji są identyczne, możemy więc przebudować instrukcję. Wprowadź te zmiany i sprawdź, co pozostało.

Czy już to zauważyłeś? Obie zagnieżdżone instrukcje `if` są teraz identyczne, więc również mogą zostać przebudowane. Wynik wprowadzenia tych zmian został przedstawiony w listingu 12.6.

**Listing 12.6.** Dalsza przebudowa instrukcji funkcji wstawiającej węzeł do listy dwukierunkowej (`dwu_wstaw3.c`)

```
/*
** Dodanie nowego węzła do listy.
*/
nowyWezel->przod = nast;

if( biez != korzen_wsk ){
    biez->przod = nowyWezel;
    nowyWezel->tyl = biez;
}
else {
    korzen_wsk->przod = nowyWezel;
    nowyWezel->tyl = NULL;
}
if( nast != NULL )
    nast->tyl = nowyWezel;
else
    korzen_wsk->tyl = nowyWezel;
```

Możemy jeszcze bardziej ulepszyć nasz kod. Pierwsza instrukcja w klauzuli `else` dla pierwszego warunku `if` może być zapisana jako:

```
biez->przod = nowyWezel;
```

ponieważ w instrukcji `if` sprawdziliśmy już, że `biez == korzen_wsk`. Zmieniona instrukcja i identyczna z nią instrukcja z drugiej gałęzi `if` może być również umieszczona przed `if`.

W listingu 12.7 zamieszczona została cała funkcja po wprowadzeniu wszystkich przedstawionych tu zmian. Wykonuje to samo zadanie, co początkowa wersja, ale jest znacznie mniejsza. Lokalne wskaźniki zostały zadeklarowane jako `register`, aby jeszcze bardziej poprawić wielkość i szybkość kodu.

**Listing 12.7.** *W pełni uproszczona funkcja wstawiająca węzeł do listy dwukierunkowej (dwu\_wstaw4.c)*

```
/*
** Wstawienie węzła do listy dwukierunkowej. korzen_wsk jest wskaźnikiem
** na węzeł korzenia, a wartosc jest nową, wstawianą wartością.
** Zwraca: 0, jeżeli wartość jest już na liście, -1, jeżeli nie ma pamięci
** na utworzenie nowego węzła lub 1 -- po udanym wstawieniu węzła.
*/
#include <stdlib.h>
#include <stdio.h>
#include "wezel_podw.h"

int
wstaw_lista_podw( register Wezel *korzen_wsk, int wartosc )
{
    register Wezel    *biez;
    register Wezel    *nast;
    register Wezel    *nowyWezel;

    /*
    ** Sprawdzenie, czy wartosc znajduje się na liście; jeżeli tak,
    ** następuje koniec funkcji. W przeciwnym przypadku następuje
    ** przydzielenie nowego węzła wskazującego na wartosc
    ** ("nowyWezel" będzie na niego wskazywał).
    ** "biez" wskazuje na węzeł, przed którym ma być wstawiona
    ** nowa wartosc, "nast" wskazuje na węzeł po nim.
    */
    for( biez = korzen_wsk; (nast = biez->przod) != NULL; biez = nast ){
        if( nast->wartosc == wartosc )
            return 0;
        if( nast->wartosc > wartosc )
            break;
    }
    nowyWezel = (Wezel *)malloc( sizeof( Wezel ) );
    if( nowyWezel == NULL )
        return -1;
    nowyWezel->wartosc = wartosc;

    /*
    ** Dodanie nowego węzła do listy.
    */
    nowyWezel->przod = nast;
    biez->przod = nowyWezel;
```

```

if( biez != korzen_wsk )
    nowyWezel->tyl = biez;
else
    nowyWezel->tyl = NULL;

if( nast != NULL )
    nast->tyl = nowyWezel;
else
    korzen_wsk->tyl = nowyWezel;

return 1;
}

```

Funkcji tej nie da się już znacznie ulepszyć, można jednak zmniejszyć ilość kodu źródłowego. Zadaniem pierwszej instrukcji `if` jest określenie prawej strony przypisania. Możemy więc zastąpić instrukcję `if` wyrażeniem warunkowym. Możemy również zamienić drugą instrukcję `if` na wyrażenie warunkowe, ale zmiana ta jest mniej oczywista.



**Wskazówka**

Ilość kodu z listingu 12.8 jest znacznie mniejsza, ale czy faktycznie jest on lepszy? Choć zawiera mniej instrukcji, ilość porównań i przypisań jest taka sama, jak poprzednio, więc nie jest on szybszy niż poprzednia wersja. Występują tu drobne różnice: `nowyWezel->tyl` i `->tyl = nowyWezel` są zapisane jednokrotnie, a nie dwukrotnie. Czy te różnice pozwolą na wygenerowanie mniejszej ilości kodu binarnego? Być może tak, ale zależy to od jakości optymalizatora w kompilatorze. Różnica ta będzie jednak niewielka, a kod będzie mniej czytelny niż poprzednio, szczególnie dla mało doświadczonych programistów. Dlatego kod z listingu 12.8 będzie sprawiał więcej kłopotu przy konserwacji.

**Listing 12.8.** Funkcja wstawiająca węzeł, wykorzystująca wyrażenie warunkowe (*dwu\_wstaw5.c*)

```

/*
** Dodanie nowego węzła do listy.
*/

nowyWezel->przod = nast;
biez->przod = nowyWezel;
nowyWezel->tyl = biez != korzen_wsk ? biez : NULL;
( nast != NULL ? nast : korzen_wsk )->tyl = nowyWezel;

```

Jeżeli wielkość programu lub szybkość wykonywania są naprawdę istotne, jedynym rozwiązaniem, jakie pozostało, jest próba zapisania tej funkcji w asemblerze. Nawet tak drastyczna decyzja nie gwarantuje znacznej poprawy, a stopień skomplikowania tworzenia, czytania i konserwacji kodu asemblera powoduje, że możliwość ta powinna być brana pod uwagę tylko jako ostateczność.

### 12.3.2. Inne operacje na listach

Podobnie, jak w przypadku list jednokierunkowych, w przypadku list dwukierunkowych niezbędne są dodatkowe operacje. Praktyki w ich tworzeniu nabierzesz podczas rozwiązywania ćwiczeń.

## 12.4. Podsumowanie

Lista jednokierunkowa jest strukturą danych, która zapisuje dane przy wykorzystaniu wskaźników. Każdy węzeł na liście zawiera pole, które wskazuje na następny węzeł. Na pierwszy węzeł wskazuje osobny węzeł, nazywany korzeniem. Ponieważ węzły są tworzone dynamicznie, mogą być rozsiane po całej pamięci. Jednak lista jest przeglądana za pomocą wskaźników, więc fizyczne rozmieszczenie węzłów nie ma znaczenia. Lista jednokierunkowa może być przeglądana tylko w jedną stronę.

Aby wstawić nową wartość do uporządkowanej listy jednokierunkowej, należy na początek znaleźć właściwe miejsce na liście. W przypadku listy bez uporządkowania nowe wartości mogą być wstawiane w dowolne miejsce. Aby dołączyć nowy węzeł do listy, należy wykonać dwie operacje. Po pierwsze, pole łączące nowego węzła musi być ustawione tak, aby wskazywało na następny węzeł. Po drugie, poprzednie pole łączące musi być zmienione, aby wskazywało na nowy węzeł. W wielu językach programowania funkcja wstawiająca zapamiętuje wskaźnik do poprzedniego węzła, dzięki czemu może wykonać kolejny krok. Technika ta sprawia, że wstawianie na początek listy jest specjalnym przypadkiem. W języku C można wyeliminować ten specjalny przypadek, zapamiętując wskaźnik do pola łączącego zamiast wskaźnika wskazującego na poprzedni węzeł.

Każdy węzeł listy dwukierunkowej zawiera dwa pola łączące: jedno wskazuje na następny węzeł na liście, drugie na węzeł poprzedni. Zastosowane są również dwa wskaźniki korzenia, które wskazują na pierwszy i na ostatni węzeł listy. Dlatego przeglądanie listy dwukierunkowej może zacząć się od dowolnego końca i może być wykonywane w dowolnym kierunku. Wstawienie nowego węzła do listy dwukierunkowej wymaga zmiany czterech połączeń. Ustawione muszą być pola łączące w przód i wstecz w nowym węźle, natomiast pole łączące wstecz następnego węzła i pole łączące w przód poprzedniego węzła muszą wskazywać na nowy węzeł.

Przebudowa instrukcji to technika upraszczania programu poprzez usuwanie z niego nadmiarowych instrukcji. Jeżeli klauzule „then” i „else” instrukcji `if` zawierają identyczne sekwencje instrukcji, mogą być one zastąpione pojedynczą sekwencją tych instrukcji, umieszczoną po `if`. Identyczne sekwencje instrukcji mogą być również przeniesione przed instrukcję `if`, o ile nie zmieniają wyniku warunku instrukcji `if`. Jeżeli różne instrukcje wykonują te same operacje, być może będziesz w stanie zmodyfikować je w identyczny sposób. Następnie można zastosować przebudowanie instrukcji, co pozwoli na uproszczenie programu.




## 12.5. Podsumowanie ostrzeżeń

1. Wyjście poza koniec listy jednokierunkowej (strona 292).
2. Należy zachować szczególną uwagę przy operacjach wykonywanych na wskaźnikach, ponieważ w języku C nie istnieją zabezpieczenia działające przy ich wykorzystywaniu (strona 297).
3. Przebudowywanie instrukcji, które może wpływać na warunek instrukcji `if` (strona 303).

## 12.6. Podsumowanie wskazówek

1. Eliminowanie przypadków specjalnych ułatwia utrzymanie kodu (strona 297).
2. Możliwe jest eliminowanie powtarzających się instrukcji z instrukcji `if` poprzez ich przebudowywanie (strona 302).
3. Nie oceniaj jakości kodu jedynie po jego wielkości (strona 306).


## 12.7. Pytania

1. Czy program z listingu 12.3 może być napisany bez zmiennej `biez`?  
Jeżeli tak, porównaj wynik z oryginałem.
-  2. Niektóre podręczniki sugerują zastosowanie w przypadku listy jednokierunkowej „węzła czołowego”. Ten nadmiarowy węzeł jest zawsze pierwszym elementem listy i eliminuje specjalny przypadek przy wstawianiu na początku listy. Omów zalety i wady tej techniki.
3. Gdzie funkcja wstawiająca z listingu 12.3 wstawiłaby duplikującą się wartość? Jaki byłby efekt zmiany porównania `z < na <=`?
-  4. Omów techniki pomijania pola wartości z węzła korzenia w przypadku listy dwukierunkowej.
5. Jaki byłby wynik, jeżeli w kodzie z listingu 12.7 funkcja `malloc` byłaby wywoływana na początku funkcji?
6. Czy możliwe jest sortowanie węzłów w nieuporządkowanej liście jednokierunkowej?
-  7. *Lista konkordancji* jest alfabetyczną listą słów, znajdującą się w książce lub artykule. Można ją utworzyć, korzystając z uporządkowanej listy jednokierunkowej ciągów oraz funkcji wstawiającej, eliminującej duplikaty. Problemem takiej implementacji jest wzrastający (wraz z wielkością listy) czas wyszukiwania.

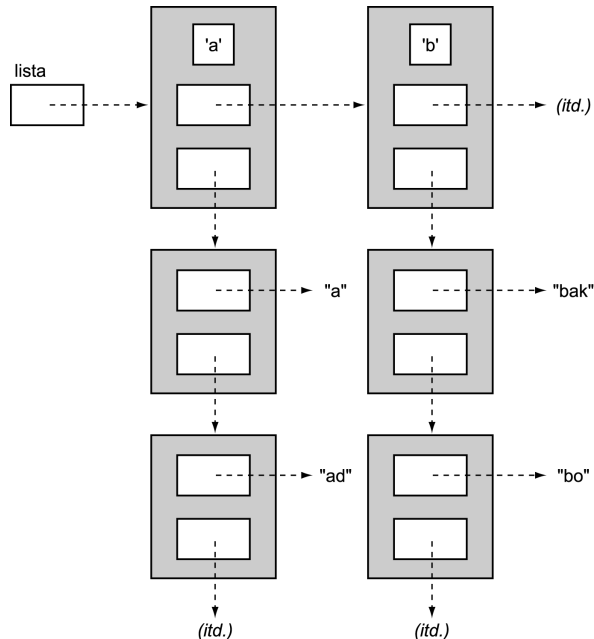
Na rysunku 12.1 pokazana została alternatywna struktura danych, służąca do zapamiętywania listy konkordancji. Założeniem jest rozbicie długiej listy na 26 mniejszych — po jednej dla każdej litery alfabetu. Każdy z węzłów listy głównej zawiera literę i wskazuje na jednokierunkową listę słów (zapamiętanych jako ciągi) zaczynających się na tę literę.

Jak ma się czas wyszukiwania w takiej strukturze w porównaniu do czasu wyszukiwania w liście jednokierunkowej zawierającej wszystkie słowa?

## 12.8. Ćwiczenia

-  ★ 1. Zaprojektuj funkcję zliczającą liczbę węzłów na liście pojedynczej. Jako jedyne argumentu powinna ona wymagać wskaźnika do pierwszego węzła. Co trzeba znać, aby napisać taką funkcję? Jakie inne zadania może wykonywać taka funkcja?

**Rysunek 12.1.**  
Lista konkordancji



- ★ 2. Zaprojektuj funkcję wyszukującą określoną wartość na nieuporządkowanej liście jednokierunkowej i zwracającą wskaźnik do tego węzła. Możesz założyć, że struktura węzła jest zdefiniowana w pliku *wezel\_poj.h*.

Czy potrzebne są jakiegokolwiek zmiany, aby funkcja korzystała z uporządkowanej listy jednokierunkowej?

- ★★★ 3. Zmień funkcję z listingu 12.7, aby wskaźniki głowy i ogona listy były przekazywane jako osobne wskaźniki, a nie elementy węzła korzenia. Jakie modyfikacje pociągnie za sobą ta zmiana w logice funkcji?
- ★★★★ 4. Zaprojektuj funkcję odwracającą kolejność węzłów na liście pojedynczej. Funkcja powinna mieć następujący prototyp:

```
struct WEZEL * lpoj_odwroc( struct WEZEL *pierwszy );
```

Skorzystaj z deklaracji węzłów z pliku *wezel\_poj.h*.

Argumentem jest pierwszy węzeł listy. Po zmianie kolejności elementów funkcja powinna zwrócić wskaźnik do nowej głowy listy. Pole łączące ostatniego węzła listy powinno zawierać wartość NULL, a w przypadku przekazania pustej listy (pierwszy == NULL) funkcja powinna zwracać wartość NULL.

- ✍️ ★★★ 5. Zaprojektuj program usuwający węzeł z listy jednokierunkowej. Funkcja powinna mieć prototyp:

```
int lpoj_usun( struct WEZEL **wsk_korzen, struct WEZEL *wezel );
```

Możesz założyć, że struktura węzła jest zdefiniowana w pliku *wezel\_poj.h*.

Pierwszy argument wskazuje na korzeń listy, a drugi wskazuje na węzeł do usunięcia. Funkcja zwraca false, jeżeli lista nie zawiera wskazanego węzła; w przeciwnym przypadku usuwa węzeł i zwraca true.

Czy istnieje jakaś zaleta przekazywania wskaźnika do węzła do usunięcia zamiast jego wartości?

- ★★★ 6. Zaprojektuj program usuwający węzeł z listy dwukierunkowej. Funkcja powinna mieć prototyp:

```
int lpodw_usun( struct WEZEL *wsk_korzen, struct WEZEL *wezel );
```

Możesz założyć, że struktura węzła jest zdefiniowana w pliku *wezel\_podw.h*. Pierwszy argument wskazuje na węzeł zawierający (tak samo jak w listingu 12.7) korzeń listy, a drugi wskazuje na węzeł przeznaczony do usunięcia. Funkcja zwraca `false`, jeżeli lista nie zawiera wskazanego węzła; w przeciwnym przypadku usuwa węzeł i zwraca `true`.

- ★★★★★ 7. Zaprojektuj funkcję wstawiającą nowe słowo do listy konkordancji, opisanej w pytaniu 7. Funkcja powinna pobierać wskaźnik do wskaźnika `lista` oraz ciąg do wstawienia. Zakładamy, że ciąg zawiera jedno słowo. Jeżeli słowo nie istnieje na liście, powinno być skopiowane do dynamicznie przydzielonej pamięci i dopiero wtedy wstawione. Funkcja powinna zwrócić `true`, jeżeli udało się wstawienie ciągu, lub `false`, jeżeli ciąg istnieje na liście, nie zaczyna się od litery lub cokolwiek innego pójdzie niepomyślnie.

Funkcja powinna utrzymywać porządek liter na liście głównej i porządek słów na listach podrzędnych.