PROGRAMOWANIE

Teoria i praktyka w C++

WYDANIE IV

BJARNE STROUSTRUP

TWÓRCA JĘZYKA C++

Helion

Tytuł oryginału: Programming: Principles and Practice Using C++, 3rd Edition

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-289-2995-1

Authorized translation from the English language edition, entitled PROGRAMMING: PRINCIPLES & PRACTICE USING C++, 3rd Edition 9780138308681 by Bjarne Stroustrup published by Pearson Education, Inc, publishing as Addison-Wesley Professional.

Cover photo by Photowood Inc./Corbis.
Author photo courtesy of Bjarne Stroustrup.
Page 325: “Promenade a Skagen“ by Peder Severin Kroyer.
Page 337: Photo of NASA’s Ingenuity Mars Helicopter, The National Aeronautics and Space Administration (NASA).
Page 380: Photo of Hurricane Rita as seen from space, The National Oceanic and Atmospheric Administration (NOAA).

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/prtpc4
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

Wstęp 15

Poprzednie wydania 16
Podziękowania 17

Uwagi do czytelnika 19

0.1. Struktura książki 20
0.1.1. Informacje ogólne 20
0.1.2. Ćwiczenia, praca domowa itp. 21
0.1.3. Po przeczytaniu tej książki 23
0.2. Filozofia nauczania i uczenia się 23
0.2.1. Słowo do studentów 26
0.2.2. Słowo do nauczycieli 27
0.3. Standard ISO C++ 28
0.3.1. Przenośność 28
0.3.2. Gwarancje 29
0.3.3. Historia języka C++ w pigułce 29
0.4. Pomoc PPP 30
0.4.1. Zasoby sieciowe 31
0.5. Biografia autora 32
0.6. Bibliografia 33

Część I. Podstawy 35

1. Witaj, świecie! 37

1.1. Programy 38
1.2. Klasyczny pierwszy program 38
1.3. Kompilacja 41
1.4. Łączenie 43
1.5. Środowiska programistyczne 44

2. Obiekty, typy i wartości 51

2.1. Dane wejściowe 52
2.2. Zmienne 54
2.3. Typy danych wejściowych 55
2.4. Operacje i operatory 56
2.5. Przypisanie i inicjalizacja 59
2.5.1. Przykład wykrywania powtarzających się słów 60
2.5.2. Złożone operatory przypisania 62
2.5.3. Przykład: szukanie powtarzających się słów 62
2.6. Nazwy 63
2.7. Typy i obiekty 65
2.8. Bezpieczeństwo typów 67
2.9. Konwersje 67
2.10. Dedukcja typów: auto 70

3. Wykonywanie obliczeń 75

3.1. Wykonywanie obliczeń 76
3.2. Cele i narzędzia 77
3.3. Wyrażenia 79
3.3.1. Wyrażenia stałe 80
3.3.2. Operatory 81
3.4. Instrukcje 83
3.4.1. Selekcja 84
3.4.2. Iteracja 89
3.5. Funkcje 93
3.5.1. Po co zaprzątać sobie głowę funkcjami? 94
3.5.2. Deklarowanie funkcji 95
3.6. Wektor 96
3.6.1. Przeglądanie zawartości wektora 97
3.6.2. Powiększanie wektora 98
3.6.3. Przykład wczytywania liczb do programu 98
3.6.4. Przykład z użyciem tekstu 100
3.7. Właściwości języka 102

4. Błędy 109

4.1. Wstęp 110
4.2. Źródła błędów 111
4.3. Błędy kompilacji 112
4.3.1. Błędy składni 112
4.3.2. Błędy typów 113
4.4. Błędy konsolidacji 115

4.5. Błędy czasu wykonania 116
4.5.1. Rozwiązywanie problemu przez wywołującego 117
4.5.2. Rozwiązywanie problemu przez wywoływanego 118
4.5.3. Raportowanie błędów 120
4.6. Wyjątki 121
4.6.1. Nieprawidłowe argumenty 121
4.6.2. Błędy zakresu 122
4.6.3. Nieprawidłowe dane wejściowe 124
4.7. Unikanie i znajdowanie błędów 127
4.7.1. Szacowanie 127
4.7.2. Debugowanie 128
4.7.3. Asercje 132
4.7.4. Testowanie 136
4.7.5. Liczby losowe 136

5. Pisanie programu 145

5.1. Problem 146
5.2. Przemyślenie problemu 146
5.2.1. Etapy rozwoju oprogramowania 147
5.2.2. Strategia 147
5.3. Wracając do kalkulatora 149
5.3.1. Pierwsza próba 150
5.3.2. Tokeny 152
5.3.3. Implementowanie tokenów 153
5.3.4. Używanie tokenów 155
5.4. Powrót do tablicy 156
5.4.1. Gramatyki 158
5.4.2. Pisanie gramatyki 160
5.5. Zamiana gramatyki w kod 161
5.5.1. Implementowanie zasad gramatyki 161
5.5.2. Wyrażenia 162
5.5.3. Składniki 165
5.5.4. Podstawowe elementy wyrażeń 166
5.6. Wypróbowywanie pierwszej wersji 167
5.7. Wypróbowywanie drugiej wersji 170
5.8. Strumienie tokenów 171
5.8.1. Implementacja typu Token_stream 173
5.8.2. Wczytywanie tokenów 174
5.8.3. Wczytywanie liczb 175
5.9. Struktura programu 176

6. Kończenie programu 181

6.1. Wprowadzenie 182
6.2. Wejście i wyjście 182
6.3. Obsługa błędów 184
6.4. Liczby ujemne 186
6.5. Reszta z dzielenia 187
6.6. Oczyszczanie kodu 189
6.6.1. Stałe symboliczne 189
6.6.2. Użycie funkcji 191
6.6.3. Układ kodu 191
6.6.4. Komentarze 193
6.7. Odzyskiwanie sprawności po wystąpieniu błędu 194
6.8. Zmienne 197
6.8.1. Zmienne i definicje 197
6.8.2. Wprowadzanie nazw 201
6.8.3. Nazwy predefiniowane 203
6.8.4. Czy to już koniec? 204

7. Szczegóły techniczne — funkcje itp. 209

7.1. Szczegóły techniczne 210
7.2. Deklaracje i definicje 211
7.2.1. Rodzaje deklaracji 214
7.2.2. Deklaracje stałych i zmiennych 214
7.2.3. Domyślna inicjalizacja 216
7.3. Zakres 216
7.4. Wywoływanie i wartość zwrotna funkcji 220
7.4.1. Deklarowanie argumentów i typu zwrotnego 221
7.4.2. Zwracanie wartości 222
7.4.3. Przekazywanie przez wartość 223
7.4.4. Przekazywanie argumentów przez stałą referencję 223
7.4.5. Przekazywanie przez referencję 226
7.4.6. Przekazywanie przez wartość a przez referencję 227
7.4.7. Sprawdzanie argumentów i konwersja 229
7.4.8. Implementacja wywołań funkcji 230
7.4.9. Obliczenia w czasie kompilacji 234
7.4.10. Przyrostkowy typ zwrotny 235
7.5. Porządek wykonywania instrukcji 236
7.5.1. Obliczanie wartości wyrażeń 237
7.5.2. Globalna inicjalizacja 238
7.6. Przestrzenie nazw 239
7.6.1. Dyrektywy i deklaracje using 240
7.7. Moduły i nagłówki 241
7.7.1. Moduły 241
7.7.2. Pliki nagłówkowe 243

8. Szczegóły techniczne — klasy itp. 251

8.1. Typy zdefiniowane przez użytkownika 252
8.2. Klasy i składowe klas 253
8.3. Interfejs i implementacja 253
8.4. Tworzenie klasy — Date 255
8.4.1. Struktury i funkcje 255
8.4.2. Funkcje składowe i konstruktory 256
8.4.3. Ukrywanie szczegółów 258
8.4.4. Definiowanie funkcji składowych 259
8.4.5. Odwoływanie się do bieżącego obiektu 262
8.4.6. Raportowanie błędów 262
8.5. Wyliczenia 263
8.5.1. „Zwykłe" wyliczenia 265
8.6. Przeciążanie operatorów 266
8.7. Interfejsy klas 267
8.7.1. Typy argumentów 268
8.7.2. Kopiowanie 270
8.7.3. Konstruktory domyślne 270
8.7.4. Stałe funkcje składowe 272
8.7.5. Funkcje składowe i funkcje pomocnicze 274
8.7.6. Standard ISO 275

Część II. Wejście i wyjście 279

9. Strumienie wejścia i wyjścia 281

9.1. Wejście i wyjście 282
9.2. Model strumieni wejścia i wyjścia 283
9.3. Pliki 284
9.3.1. Otwieranie pliku 286
9.3.2. Odczytywanie i zapisywanie plików 287
9.4. Obsługa błędów wejścia i wyjścia 289
9.5. Wczytywanie pojedynczej wartości 291
9.5.1. Rozłożenie problemu na mniejsze części 293
9.5.2. Oddzielenie warstwy komunikacyjnej od funkcji 296
9.6. Definiowanie operatorów wyjściowych 297
9.7. Definiowanie operatorów wejściowych 298
9.8. Standardowa pętla wejściowa 298
9.9. Wczytywanie pliku strukturalnego 300
9.9.1. Reprezentacja danych w pamięci 301
9.9.2. Odczytywanie struktur wartości 302
9.9.3. Zmienianie reprezentacji 305
9.10. Formatowanie 306
9.10.1. Przyjmowanie na wejściu i wysyłanie na wyjście liczb całkowitych 307
9.10.2. Przyjmowanie na wejściu i wysyłanie na wyjście liczb zmiennoprzecinkowych 309

9.10.3. Przyjmowanie na wejściu i wysyłanie na wyjście łańcuchów 310
9.10.4. Przyjmowanie na wejściu i wysyłanie na wyjście znaków 310
9.10.5. Rozszerzanie wejścia i wyjścia 312
9.10.6. Funkcja format() 312
9.11. Strumienie łańcuchowe 314

10. Projektowanie klas graficznych 321

10.1. Czemu grafika? 322
10.2. Model graficzny 323
10.3. Pierwszy przykład 324
10.4. Biblioteka GUI 327
10.5. Współrzędne 328
10.6. Figury geometryczne 329
10.7. Używanie klas figur geometrycznych 329
10.7.1. Klasa Axis 330
10.7.2. Rysowanie wykresu funkcji 331
10.7.3. Wielokąty 332
10.7.4. Prostokąty 333
10.7.5. Wypełnianie kolorem 335
10.7.6. Tekst 336
10.7.7. Obrazy 337
10.7.8. Jeszcze więcej grafik 338
10.8. Uruchamianie pierwszego przykładu 339
10.8.1. Pliki źródłowe 340
10.8.2. Składanie wszystkiego w całość 341

11. Klasy graficzne 345

11.1. Przegląd klas graficznych 346
11.2. Klasy Point i Line 348
11.3. Klasa Lines 350
11.3.1. Inicjalizacja 352
11.4. Klasa Color 353
11.5. Typ Line_style 355
11.6. Linie łamane 358
11.6.1. Typ Open_polyline 358
11.6.2. Typ Closed_polyline 359
11.6.3. Typ Marked_polyline 360
11.6.4. Typ Marks 362
11.7. Figury zamknięte 363
11.7.1. Typ Polygon 364
11.7.2. Typ Rectangle 366
11.7.3. Wykorzystywanie obiektów bez nazw 369
11.7.4. Typ Circle 371
11.7.5. Typ Ellipse 373

11.8. Typ Text 374
11.9. Typ Mark 377
11.10. Typ Image 378

12. Projektowanie klas 385

12.1. Zasady projektowania 386
12.1.1. Typy 386
12.1.2. Operacje 388
12.1.3. Nazewnictwo 389
12.1.4. Zmienność 390
12.2. Klasa Shape 391
12.2.1. Klasa abstrakcyjna 392
12.2.2. Kontrola dostępu 393
12.2.3. Rysowanie figur 396
12.3. Klasy bazowe i pochodne 398
12.3.1. Układ obiektu 399
12.3.2. Tworzenie podklas i definiowanie funkcji wirtualnych 401
12.3.3. Przesłanianie 402
12.3.4. Dostęp 404
12.3.5. Czyste funkcje wirtualne 405
12.4. Inne funkcje klasy Shape 406
12.4.1. Kopiowanie 406
12.4.2. Przesuwanie figur 407
12.5. Zalety programowania obiektowego 407

13. Graficzne przedstawienie funkcji i danych 413

13.1. Wprowadzenie 414
13.2. Rysowanie wykresów prostych funkcji 414
13.3. Typ Function 417
13.3.1. Argumenty domyślne 418
13.3.2. Więcej przykładów 419
13.3.3. Wyrażenia lambda 420
13.4. Typ Axis 421
13.5. Wartość przybliżona funkcji wykładniczej 423
13.6. Przedstawianie danych na wykresach 428
13.6.1. Odczyt danych z pliku 429
13.6.2. Układ ogólny 431
13.6.3. Skalowanie danych 432
13.6.4. Budowanie wykresu 433

14. Graficzne interfejsy użytkownika 439

14.1. Różne rodzaje interfejsów użytkownika 440
14.2. Przycisk Dalej 441
14.3. Proste okno 442
14.3.1. Pętla oczekująca 443
14.4. Typ Button i inne pochodne typu Widget 444
14.4.1. Widgety 444
14.4.2. Przyciski 446
14.4.3. Widgety In_box i Out_box 446
14.4.4. Menu 448
14.5. Przykład: rysowanie linii 448
14.5.1. Inwersja kontroli 451
14.5.2. Dodawanie menu 452
14.6. Prosta animacja 456
14.7. Debugowanie kodu GUI 457

Część III. Dane i algorytmy 463

15. Wektory i pamięć wolna 465

15.1. Wprowadzenie 466
15.2. Podstawowe wiadomości na temat typu vector 468
15.3. Pamięć, adresy i wskaźniki 469
15.3.1. Operator sizeof 471
15.4. Pamięć wolna a wskaźniki 472
15.4.1. Alokacja obiektów w pamięci wolnej 473
15.4.2. Dostęp poprzez wskaźniki 474
15.4.3. Inicjalizacja 475
15.4.4. Wskaźnik zerowy 476
15.4.5. Dealokacja pamięci wolnej 477
15.5. Destruktory 478
15.5.1. Generowanie destruktorów 479
15.5.2. Destruktory wirtualne 480
15.6. Dostęp do elementów 482
15.7. Przykład — listy 482
15.7.1. Operacje na listach 484
15.7.2. Zastosowania list 485
15.8. Wskaźnik this 487
15.8.1. Więcej przykładów użycia typu Link 488

16. Tablice, wskaźniki i referencje 495

16.1. Tablice 496
16.1.1. Arytmetyka wskaźników 498
16.2. Wskaźniki i referencje 500
16.2.1. Wskaźniki i referencje jako parametry 501
16.2.2. Wskaźniki jako parametry 502
16.3. Łańcuchy w stylu języka C 503
16.4. Alternatywy dla wskaźników 505
16.4.1. Typ span 505
16.4.2. Typ array 506
16.4.3. Typ not_null 506
16.5. Przykłady: palindromy 508
16.5.1. Wykorzystanie łańcuchów 508
16.5.2. Wykorzystanie tablic 509
16.5.3. Wykorzystanie wskaźników 510
16.5.4. Wykorzystanie typu span 510

17. Podstawowe operacje 517

17.1. Wprowadzenie 518
17.2. Dostęp do elementów 518
17.3. Inicjalizacja 520
17.4. Kopiowanie i przenoszenie 522
17.4.1. Konstruktory kopiujące 523
17.4.2. Przypisywanie z kopiowaniem 524
17.4.3. Terminologia związana z kopiowaniem 526
17.4.4. Przenoszenie 527
17.5. Podstawowe operacje 529
17.5.1. Konstruktory jawne 531
17.5.2. Debugowanie konstruktorów i destruktorów 532
17.6. Inne przydatne operacje 534
17.6.1. Operatory porównań 535
17.6.2. Pokrewne operatory 535
17.7. Pozostałe problemy z klasą Vector 536
17.8. Zmienianie rozmiaru 539
17.8.1. Reprezentacja 539
17.8.2. Rezerwacja pamięci i pojemność kontenera 541
17.8.3. Zmienianie rozmiaru 541
17.8.4. Funkcja push_back() 542
17.8.5. Przypisywanie 542
17.9. Podsumowanie dotychczasowej pracy nad typem vector 544

18. Szablony i wyjątki 549

18.1. Szablony 550
18.1.1. Typy jako parametry szablonów 550
18.1.2. Programowanie ogólne 553
18.1.3. Koncepcje 554
18.1.4. Kontenery a dziedziczenie 557
18.1.5. Wartościowe parametry szablonów 558
18.2. Uogólnianie wektora 558
18.2.1. Alokatory 559
18.3. Sprawdzanie zakresu i wyjątki 561
18.3.1. Dygresja — uwagi projektowe 563
18.3.2. Moduł PPP_support 564
18.4. Zasoby i wyjątki 565
18.4.1. Potencjalne problemy z zarządzaniem zasobami 566
18.4.2. Zajmowanie zasobów jest inicjalizacją 568
18.4.3. Gwarancje dotyczące zarządzania zasobami 569
18.4.4. Technika RAII dla wektora 570
18.5. Wskaźniki zarządzające zasobami 573
18.5.1. Zwrot przez przeniesienie 574
18.5.2. Obiekt unique_ptr 574
18.5.3. Obiekt shared_ptr 576

19. Kontenery i iteratory 581

19.1. Przechowywanie i przetwarzanie danych 582
19.1.1. Praca na danych 582
19.1.2. Uogólnianie kodu 583
19.1.3. Ideały twórcy biblioteki STL 585
19.2. Sekwencje i iteratory 589
19.2.1. Powrót do przykładu Jacka i Agatki 591
19.3. Listy powiązane 592
19.3.1. Operacje list 594
19.3.2. Iteracja 595
19.4. Jeszcze raz uogólnianie wektora 596
19.4.1. Przeglądanie kontenera 598
19.4.2. Funkcje insert() i erase() 599
19.4.3. Dostosowanie wektora do biblioteki STL 600
19.5. Przykład: prosty edytor tekstu 602
19.5.1. Wiersze 604
19.5.2. Iteracja 605
19.6. Typy vector, list oraz string 608

20. Słowniki i zbiory 615

20.1. Kontenery asocjacyjne 616
20.2. Słowniki 616
20.2.1. Wiązanie strukturalne 618
20.2.2. Opis ogólny słownika 618
20.2.3. Jeszcze jeden przykład zastosowania słownika 621
20.3. Kontener unordered_map 623
20.4. Czas 625
20.4.1. Daty 627
20.5. Zbiory 628
20.6. Przegląd kontenerów 629
20.6.1. „Prawie kontenery” 631
20.6.2. Kontener array 632
20.6.3. Dostosowywanie wbudowanych tablic do STL 632
20.6.4. Dostosowywanie strumieni wejścia/wyjścia do biblioteki STL 633
20.6.5. Utrzymywanie porządku przy użyciu kontenera set 635
20.7. Zakresy i iteratory 635
20.7.1. Kategorie iteratorów 636
20.7.2. Zakresy wyjściowe 637

21. Algorytmy 643

21.1. Algorytmy biblioteki standardowej 644
21.1.1. Najprostszy algorytm: find() 645
21.1.2. Kilka przykładów z programowania ogólnego 647
21.1.3. Ogólny algorytm wyszukiwania: find_if() 648
21.2. Obiekty funkcyjne 650
21.2.1. Abstrakcyjne spojrzenie na obiekty funkcyjne 651
21.2.2. Predykaty składowych klas 652
21.2.3. Wyrażenia lambda 653
21.3. Algorytmy numeryczne 654
21.3.1. Akumulacja 654
21.3.2. Uogólnianie funkcji accumulate() 656
21.3.3. Iloczyn skalarny 657
21.3.4. Uogólnianie funkcji inner_product() 658
21.4. Kopiowanie 659
21.4.1. Najprostsza wersja funkcji copy() 659
21.4.2. Uogólnianie funkcji copy_if() 660
21.5. Sortowanie i wyszukiwanie 660

Skorowidz 667

16
Tablice, wskaźniki i referencje

Caveat emptor!

— dobra rada

W tym rozdziale opisujemy pojęcia tablic i wskaźników na niższym poziomie. Przedstawiamy różne zastosowania wskaźników, na przykład do przeglądania tablic i arytmetyki adresów, oraz problemy wynikające z takiego użycia. Omawiamy również powszechnie używane łańcuchy w stylu języka C, czyli tablice znaków zakończone zerem. Wskaźniki i tablice są kluczowe dla implementacji typów, które chronią nas przed ryzykownym stosowaniem wskaźników, takich jak `vector`, `string`, `span`, `not_null`, `unique_ptr` i `shared_ptr`. Jako przykład pokazujemy kilka sposobów implementacji funkcji, która sprawdza, czy ciąg znaków reprezentuje palindrom.

16.1. Tablice
    16.1.1. Arytmetyka wskaźnikowa
16.2. Wskaźniki i referencje
    16.2.1. Wskaźniki i referencje jako parametry
    16.2.2. Wskaźniki jako parametry
16.3. Łańcuchy w stylu języka C
16.4. Alternatywy dla wskaźników
    16.4.1. Typ span
    16.4.2. Typ array
    16.4.3. Typ not_null
16.5. Przykłady: palindromy
    16.5.1. Wykorzystanie łańcuchów
    16.5.2. Wykorzystanie tablic
    16.5.3. Wykorzystanie wskaźników
    16.5.4. Wykorzystanie typu span

16.1. Tablice

Do tej pory wszystkie tablice alokowaliśmy w pamięci wolnej. Tego potrzebujemy do implementacji wektora. Tablice jednak można przechowywać także na stosie i w pamięci statycznej (podrozdział 15.4). Na przykład:

```
int ai[4]; // Statyczna tablica 4 liczb typu int

void fct()
{
    char ac[8]; // Tablica 8 znaków przechowywana na stosie
}
```

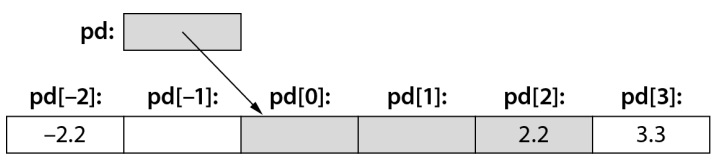
W języku C++ do tworzenia używana jest standardowa notacja []. W większości przypadków ciągłe sekwencje elementów typu T lepiej jest przechowywać w strukturze `vector<T>` niż w `T[]`. Słowo „ciągłe" oznacza, że między elementami nie ma żadnych przerw.

AA Nie trudno zauważyć, że jesteśmy zdecydowanie za stosowaniem wektorów zamiast tablic. Należy ich używać wszędzie, gdzie się da — a da się w większości sytuacji. Trzeba jednak zaznaczyć, że tablice powstały dużo wcześniej niż wektory i mniej więcej odpowiadają strukturom dostępnym w innych językach programowania (np. C). Dlatego należy je dobrze znać, aby rozumieć stary kod oraz kod napisany przez programistów, którzy nie korzystają z zalet wektorów.

XX Największym problemem ze wskaźnikami do tablic jest to, że wskaźniki „nie wiedzą", ile elementów wskazują. Wskaźnik do tablicy wskazuje jej pierwszy element, a nie jakiś „obiekt tablicowy". Na przykład:

```
void use(double* pd)
{
    pd[2] = 2.2;
    pd[3] = 3.3;
    pd[-2] = -2.2;
}

void test()
{
    double a[3];
    use(a); // Kiedy zmienna a zostanie użyta jako argument, to zamieni się we wskaźnik do a[0]
}
```

To, co „widzi" funkcja `use()`, można przedstawić graficznie w następujący sposób:



XX Czy `pd` zawiera trzeci element `pd[2]`? Czy zawiera czwarty element `pd[3]`? Jeśli spojrzysz na wywołanie funkcji `use(a)`, to dostrzeżesz, że odpowiedź na pierwsze pytanie brzmi „tak", a na drugie „nie". Jednak kompilator tego nie wie — nie śledzi on wartości wskaźników. Nasz kod po prostu uzyska dostęp do pamięci tak, jakbyśmy zaalokowali jej wystarczająco dużo. Nawet odwołanie do `pd[-2]` zostanie potraktowane tak, jakby miejsce w pamięci dwa elementy typu `double` przed tym, na co wskazuje `pd`, także zaliczało się do naszej alokacji.

Nie mamy pojęcia, do czego są używane miejsca w pamięci o oznaczeniach `pd[-2]` i `pd[3]`. Wiemy jednak, że nie miały stanowić części naszej tablicy trzech liczb typu `double` wskazywanej przez `pd`. Najprawdopodobniej należą one do innych obiektów i właśnie je zamazaliśmy. To nie jest dobry pomysł. Jest to wręcz potwornie zły pomysł. Ta potworność wyraża się słowami „nie wiedzieć czemu mój program nagle przestaje działać" albo „mój program zwraca niepoprawne wyniki". Wymów te zdania na głos. Nie brzmią przyjemnie. Za wszelką cenę będziemy starać się tego uniknąć. CC

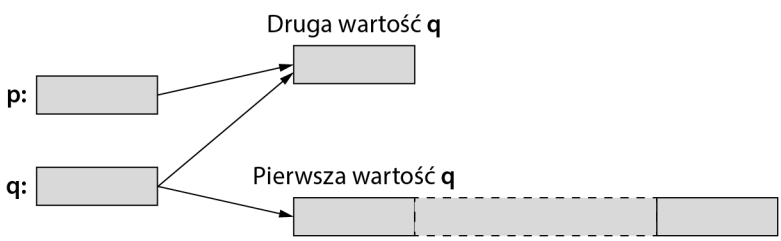
Dostęp poza zakresem nazywa się **błędem zakresu** (ang. *range error*) lub **przepełnieniem bufora** (ang. *buffer overflow*). Tego rodzaju błędy są szczególnie wredne, ponieważ dotykają pozornie niemających z nimi nic wspólnego części programu. Odczyt spoza zakresu daje nam „przypadkowe" wartości, które mogą być zależne od jakichś całkiem innych operacji. Zapis poza zakresem może przestawić jakiś obiekt w „niemożliwy" stan albo nadać mu absolutnie nieoczekiwaną i niepoprawną wartość. Takie nieprawidłowości zazwyczaj zostają zauważone dopiero długo po tym, jak wystąpiły, przez co wyjątkowo trudno je znaleźć. Co gorsza, jeśli uruchomisz program z błędem zakresu dwa razy ze zmienionymi danymi wejściowymi, to za każdym razem może zwrócić inny wynik. Błędy tego typu (błędy przejściowe) należą do najtrudniejszych do wykrycia. XX

Musimy dopilnować, aby taki dostęp poza zakresem nie miał nigdy miejsca. Jednym z powodów, dla których używamy wektora zamiast pamięci bezpośrednio alokowanej za pomocą operatora `new`, jest to, że wektor zna swój rozmiar, więc z łatwością można uniknąć błędów przekroczenia zakresu. AA

Jednym z czynników, które mogą utrudniać zapobieganie błędom przekroczenia zakresu, jest to, że możemy przypisać jedną wartość typu `double*` do innej wartości typu `double*` niezależnie od tego, ile obiektów każda z nich wskazuje. Wskaźnik nie wie, na ile obiektów wskazuje, np.:

```
double* p = new double; // Alokacja wartości typu double
double* q = new double[1000]; // Alokacja 1000 wartości typu doubles

q[700] = 7.7; // Dobrze: q wskazuje 1000 wartości typu double
q = p; // Niech q wskazuje ten sam obiekt, co p
double d = q[700]; // Źle: q wskazuje jedną wartość typu double: dostęp poza zakresem!
```

Tutaj, w zaledwie trzech wierszach kodu, `q[700]` odnosi się do dwóch różnych miejsc w pamięci i ostatni przypadek użycia jest dostępem poza zakresem, a więc zapowiada katastrofę.



Mamy nadzieję, że nasuwa Ci się już pytanie: ale dlaczego wskaźniki nie mogą pamiętać rozmiaru? Oczywiście można zaprojektować wskaźnik, który by to robił — wektor nim prawie jest, a jeśli przejrzysz literaturę na temat języka C++ i różne biblioteki, to znajdziesz wiele „inteligentnych wskaźników", które rekompensują niedoskonałości niskopoziomowych wskaźników wbudowanych (zobacz np. `span` w punkcie 16.4.1). Gdzieś jednak musimy

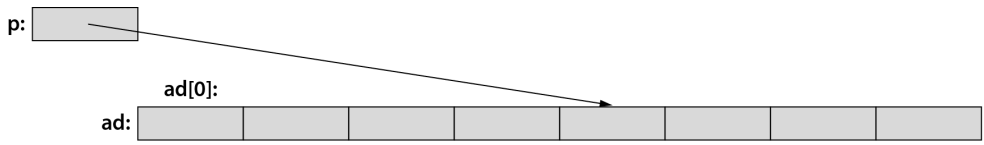
sięgnąć do poziomu sprzętu i dowiedzieć się, jak są adresowane obiekty — a adres sprzętowy „nie wie", co adresuje. Ponadto znajomość wskaźników jest niezbędna do zrozumienia wielu prawdziwych programów.

16.1.1. Arytmetyka wskaźników

Wskaźnik może wskazywać element tablicy. Na przykład:

```
double ad[8];
double* p = &ad[4]; // Wskazuje ad[4], czyli piąty element tablicy ad
```

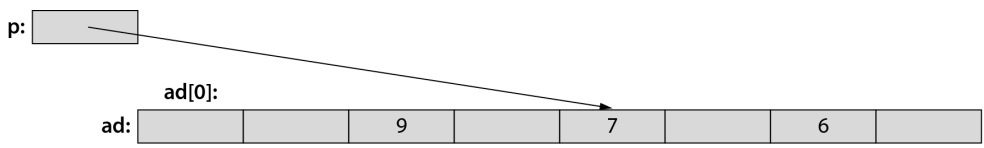
Utworzyliśmy wskaźnik o nazwie `p` do liczby typu `double` przechowywanej jako `ad[4]`:



Możemy ten wskaźnik indeksować i wyłuskiwać:

```
*p = 7;
p[2] = 6;
p[-2] = 9;
```

Otrzymujemy:

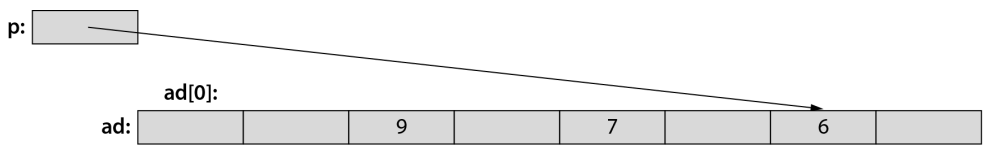


To znaczy, że jako indeksu można użyć zarówno dodatniej, jak i ujemnej wartości. Dopóki element mieści się w zakresie, wszystko jest w porządku. Niedozwolony jest natomiast dostęp do elementów leżących poza zakresem tablicy (jak w tablicach alokowanych w pamięci wolnej — podrozdział 16.1). Zazwyczaj takie próby dostępu poza zakresem nie są wykrywane przez kompilator i wcześniej lub później wywołują katastrofalne skutki.

Gdy wskaźnik wskazuje element w tablicy, można za pomocą dodawania i indeksowania przestawić go na inny element tej tablicy. Na przykład:

```
p += 2; // Przesuwa p o dwa elementy w prawo
```

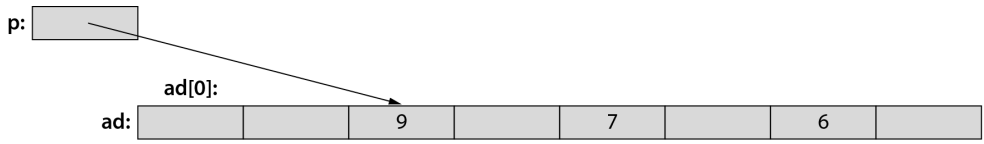
Otrzymujemy:



Możemy też działać w drugą stronę:

```
p -= 4; // Przesuwa p o 4 elementy w lewo
```

Otrzymujemy:



Przesuwanie wskaźników po elementach za pomocą operatorów `+`, `-`, `+=` oraz `-=` nazywa się **arytmetyką wskaźników** (ang. *pointer arithmetic*). Oczywiście stosując tę technikę, należy uważać, aby nie ustawić wskaźnika na element poza tablicą: CC

```
p += 1000;      // Szalony pomysł: p wskazuje element w tablicy zawierającej tylko 8 elementów
double d = *p; // Niedozwolone: prawdopodobnie niepoprawna wartość
*p = 12.34;     // Niedozwolone: prawdopodobnie uszkodzi jakieś nieznane dane
```

Niestety nie wszystkie błędy związane z arytmetyką wskaźników tak łatwo zauważyć. Najlepiej po prostu unikać tego rodzaju działań, z wyjątkiem sytuacji, gdy trzeba zaimplementować coś, dla czego nie ma sensownej alternatywy.

Najczęściej spotykaną formą arytmetyki wskaźników jest inkrementacja wskaźnika (za pomocą operatora `++`), aby ustawić go na następny element, oraz dekrementacja wskaźnika (za pomocą operatora `--`), aby ustawić go na poprzedni element. Na przykład wartości tablicy `ad` można wydrukować w następujący sposób:

```
const int max = sizeof(ad)/sizeof(*ad); // Jeden ze sposobów na sprawdzenie liczby
                                        // elementów w ad

for (double* p = &ad[0]; p<&ad[max]; ++p)
    cout << *p << '\n';
```

Lub w drugą stronę:

```
for (double* p = &ad[max-1]; p>=&ad[0]; --p)
    cout << *p << '\n';
```

Ten sposób zastosowania arytmetyki wskaźników jest dość często spotykany. Jednak naszym zdaniem w drugim z przedstawionych przykładów (tym drukującym wstecz) można łatwo popełnić błąd. Dlaczego `&ad[max-1]`, a nie `&ad[max]`? Czemu `>=`, a nie `>`? Kod tych przykładów równie dobrze można by było napisać przy użyciu indeksowania wektora lub obiektu `span`, którego zakres łatwiej się sprawdza. Ponadto wiele przykładów można wykonać przy użyciu zakresowej pętli `for` (punkt 3.6.1) zamiast indeksów.

Sposób sprawdzania liczby elementów w tablicy (tutaj `sizeof(ad)/sizeof(*ad)`) może wydawać się dziwny, ale zastanów się: operator `sizeof` zwraca rozmiar w bajtach wykorzystanych do przechowywania obiektu, a więc `sizeof(ad)` będzie wynosić `8*sizeof(double)`, ponieważ w tablicy `ad` umieściliśmy 8 elementów typu `double`. Aby uzyskać liczbę elementów (tutaj `8`), musimy wykonać dzielenie przez rozmiar elementu (tutaj `*ad`). Gdy obniżymy poziom abstrakcji, kod robi się bardziej zagmatwany i łatwiej w nim popełnić błąd, a operator `sizeof` pozwala nam zejść na sam dół.

Należy zauważyć, że w większości realnych przypadków zastosowania arytmetyki wskaźników jest obecny wskaźnik przekazywany jako argument do funkcji (jak przykład funkcji `use()` pokazany w podrozdziale 16.1). W takim przypadku kompilator nie ma pojęcia, ile

elementów zawiera tablica, której element jest wskazywany — programista jest pozostawiony sam sobie. Wolimy unikać takich sytuacji, kiedy się tylko da.

Dlaczego język C++ w ogóle dopuszcza arytmetykę wskaźników? Przecież technika ta może sprawiać duże problemy, a nie wnosi nic nowego, jeśli mamy indeksowanie. Na przykład:

```
double* p1 = &ad[0];
double* p2 = p1+7;
double* p3 = &p1[7];
if (p2 != p3) cout << "Niemożliwe!\n";
```

CC Najważniejszy powód ma historyczne podłoże. Reguły te opracowano wiele lat temu dla języka C i nie można się ich pozbyć, nie unieruchamiając dużych ilości kodu. Ponadto zdarzają się sytuacje w niskopoziomowym programowaniu, np. zarządzania pamięcią, w których ten rodzaj arytmetyki stanowi pewne udogodnienie.

16.2. Wskaźniki i referencje

CC Referencję można traktować jako automatycznie wyłuskiwany niezmienny wskaźnik albo jako alternatywną nazwę obiektu. Oto różnice między wskaźnikami i referencjami:

- Przypisanie do wskaźnika zmienia wartość wskaźnika (nie wskazywanej wartości).
  - Przypisanie do referencji zmienia wartość obiektu, do której się ona odnosi (nie referencji).
- Wskaźnik można zainicjalizować za pomocą operatora `new`, `&` lub nazwy tablicy.
  - Do inicjalizacji referencji używa się obiektu (może być wskaźnik wyłuskany przez * lub [ ]).
- Aby uzyskać dostęp do obiektu wskazywanego przez wskaźnik, używamy operatora * lub [ ] (podrozdział 16.1).
  - Aby uzyskać dostęp do obiektu, do którego odnosi się referencja, używamy tylko nazwy referencji.
- Uważaj na wskaźniki `null` (punkt 15.4.4).
  - Referencja musi być zainicjalizowana odniesieniem do obiektu i nie może zostać przestawiona na inny obiekt.

Na przykład:

Wskaźnik	Referencja	Komentarz
`int x = 10;`	`int x = 10;`	
`int* p = &x;`	`int& r = x;`	`&x`, aby uzyskać wskaźnik
`p = x;`	`r = &x;`	błędy typów
`*p = 7;`	`r = 7;`	zapis do wskazywanego obiektu
`p = 7;`	`*r = 7;`	błędy typów
`int x2 = *p;`	`int x2 = r;`	odczyt wartości wskazywanego obiektu

Wskaźnik	Referencja	Komentarz
`int* p2 = p;`	`int& r2 = r;`	`p2` wskazuje na obiekt wskazywany przez `p`
		`p2` odnosi się do obiektu wskazywanego przez `r`
`p = nullptr;`	`r = "nullref";`	nie ma `nullref`
		`p` wskazuje `x2`
`p = &x2;`	`r = x2;`	przypisuje `x2` do obiektu, do którego odnosi się `r`

Wyrażenie `r=x2` nie sprawi, że referencja będzie odnosić się do `x2`. Nie da się zmienić obiektu wskazywanego przez referencję po jej inicjalizacji i jest to cenna gwarancja. Wyrażenie `r=x2` przypisuje wartość `x2` do obiektu, do którego odnosi się referencja, czyli do `x`. Jeśli chcesz zmieniać wskazywany obiekt, użyj wskaźnika. AA

Opisy sposobów i miejsc użycia wskaźników znajdują się w podrozdziałach 15.7 i 16.5.

Referencje i wskaźniki są zaimplementowane przy użyciu adresów pamięci, tylko korzystają z nich na różne sposoby, aby dać nam — programistom — odrobinę różne narzędzia.

16.2.1. Wskaźniki i referencje jako parametry

Są trzy sposoby zmiany wartości zmiennej na wartość obliczoną przez funkcję. Na przykład:

```
int incr_v(int x) { return x+1; } // Oblicza nową wartość i ją zwraca
void incr_p(int* p) { ++*p; }     // Przekazanie wskaźnika
                                  // (wyłuskanie go i inkrementacja wyniku)
void incr_r(int& r) { ++r; }      // Przekazanie referencji
```

Którą opcję wybrać? Naszym zdaniem najkorzystniejsze dla czytelności kodu (a dzięki temu najmniej podatne na błędy) jest zwrócenie wartości:

```
int x = 2;
x = incr_v(x); // Kopiuje x do incr_v(); następnie kopiuje wynik na zewnątrz i go przypisuje
```

Ten styl stosujemy, gdy mamy do czynienia z niewielkimi obiektami, np. typu int. Ponadto jeśli „duży obiekt" ma konstruktor przenoszący (punkt 17.4.4), możemy go bardzo efektywnie przekazywać.

Jak dokonać wyboru między argumentem w postaci referencji a argumentem w postaci wskaźnika? Niestety każda z metod ma zalety i wady, przez co nie można podać jednoznacznej odpowiedzi. Decyzję należy podjąć, biorąc pod uwagę indywidualne cechy funkcji i jej możliwe zastosowania.

Obecność wskaźnika jako argumentu funkcji dla programisty jest oznaką, że ktoś może ją wywołać ze wskaźnikiem `null`, tzn. `nullptr`. Na przykład: XX

```
incr_p(nullptr); // Awaria: incr_p() będzie próbować wyłuskać wskaźnik null
int* p = nullptr;
incr_p(p); // Awaria: incr_p() będzie próbować wyłuskać wskaźnik null
```

Jest to bardzo nieprzyjemne. Osoba odpowiedzialna za implementację funkcji `incr_p()` może zapewnić ochronę przed taką sytuacją:

```
void incr_p(int* p)
{
    if (p==nullptr) error("Wskaźnik null podany jako argument funkcji incr_p().");
    ++*p; // Dereferencja wskaźnika i inkrementacja wskazywanego przez niego obiektu
}
```

Teraz jednak funkcja `incr_p()` nie jest już taka prosta i fajna jak wcześniej. Techniki radzenia sobie z niepoprawnymi argumentami opisaliśmy w rozdziale 4. Natomiast użytkownicy referencji (jak `incr_r()`) mają prawo zakładać, że referencja ta odwołuje się do jakiegoś obiektu.

Jeśli z punktu widzenia semantyki funkcji przekazywanie niczego („przekazywanie żadnego obiektu") jest akceptowalne, należy użyć wskaźnika jako argumentu. Uwaga: nie dotyczy to operacji inkrementacji — dlatego konieczne jest zgłoszenie wyjątku dla sytuacji `p==nullptr`.

AA Prawdziwa odpowiedź zatem brzmi: wybór zależy od właściwości funkcji:

- Dla małych obiektów lepiej stosować przekazywanie przez wartość. Pod pojęciem „małe" rozumiemy jedną lub dwie wartości typów wbudowanych (np. dwie liczby typu double) albo jeden lub dwa obiekty klas o takim samym rozmiarze.
- W funkcjach, w których „żaden obiekt" (reprezentowany przez `nullptr`) jest poprawnym argumentem, należy stosować parametry wskaźnikowe (i pamiętać o sprawdzeniu `nullptr`).
- W pozostałych przypadkach należy używać parametrów w postaci referencji.

Zobacz też punkt 7.4.6 i nie zapomnij o przekazywaniu przez stałą referencję.

16.2.2. Wskaźniki jako parametry

XX Wskaźniki są często używane jako parametry. Na przykład:

```
void print_n(int* p, int n)
{
    if (p==nullptr) // ochrona przed nullptr
        return;

    for (int i = 0; i<n; ++i)
        cout << v[i] << ' ';
}

void user()
{
    int a[12];
    int* p = new int [10];
    // ...wypełnienie a i *p...
    print_n(a,12);
    print_n(p,12);
}
```

W tym kodzie jest wiele problemów: rozwlekłość, stałe magiczne (punkt 3.3.1), wyciek pamięci (punkt 15.4.5) i błąd zakresu (punkt 4.6.2).

AA Przyczyną problemów jest to, że wskaźnik z definicji nie „wie", ile elementów, które wskazuje, jest używanych jako parametr. Zalecamy nie przekazywać sekwencji elementów jako par (wskaźnik,liczba). Należy zakładać, że argument będący wskaźnikiem wskazuje jeden obiekt lub jest `nullptr`. Zamiast tego należy przekazać obiekt reprezentujący zakres, np. `span` (punkt 16.4.1):

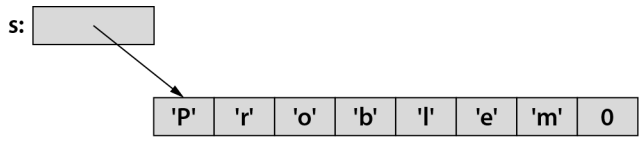
```
void print_n(span<int> s) // Span wskazuje tablicę i „pamięta” swoją liczbę elementów (punkt 16.4.1)
{
    for (int x : s)
        cout << x << ' ';
}

void user()
{
    int a[12];
    vector<int> v(10);
    // ...wypełnienie a i v...
    print_n(a); // Drukuje 12 liczb całkowitych
    print_n(v); // Drukuje 10 liczb całkowitych
}
```

16.3. Łańcuchy w stylu języka C

Od najwcześniejszych dni istnienia języka C łańcuch znaków w pamięci był reprezentowany przez zakończoną zerem tablicę znaków, do których dostęp można było uzyskać przez wskaźniki (PPP2.§27.5). Na przykład:

```
const char* s = "Problem";
```

Ten łańcuch możemy graficznie przedstawić następująco:



Nasze literały łańcuchowe są łańcuchami w stylu języka C. Znaki literału łańcuchowego są stałe, więc jeśli chcemy je zmienić, to musimy użyć tablicy:

```
char s[] = "Modyfikować";
cout << "modyfikować: " << s; // Drukuje „Modyfikować”

s[7] = 'u';
s[8] = 'j';
s[9] = 0; // Końcowe zero

cout << "modyfikować: " << s; // Drukuje „Modyfikować”
```

Liczbę znaków w łańcuchu w stylu języka C można sprawdzić za pomocą funkcji `strlen()`:

```
cout << strlen(s); // Drukuje 8
```

Funkcja `strlen()` nie liczy końcowego zera, czyli zwraca liczbę znaków do tego zera, ale bez jego uwzględnienia. To oznacza, że jeśli umieścimy `0` w środku łańcucha, jak zrobiliśmy w przypadku `s`, to nie będziemy już w stanie określić, ile pamięci zostało alokowanej dla tego łańcucha.

Nasze „łańcuchy” w stylu języka C są wskaźnikami i mają semantykę wskaźnikową. To znaczy, że operacje przypisania, kopiowania i `sizeof` odnoszą się do wskaźnika, a nie do tego, co on wskazuje. Na przykład:

```
string cat(const string& name, const string& addr)
{
    return id + '@' + addr;
}
```

Jeśli użyjemy łańcuchów w stylu języka C zamiast `std::string`, to otrzymamy:

```
char* cat2(const char* name , const char* addr)
{
    int nsz = strlen(name);
    int sz = nsz+strlen(addr)+2; // +2 dla @ i końcowego zera
    char* res = (char*) malloc(sz); // Użycie starej funkcji alokacji
    strcpy(res,name); // Kopiowanie z *name do *res do napotkania zera
    res[nsz+1] = '@'
    strcpy(res+2,addr);
    return res;
}
```

Funkcja biblioteki standardowej C i C++ `malloc()` przydziela pamięć na stercie. Pamięć zaalokowaną przez `malloc()` trzeba zwrócić menadżerowi sterty za pomocą funkcji `free()`. W kodzie w języku C często można spotkać właśnie takie konstrukcje zamiast operatorów `new/delete` — `string`, `new` i `delete` nie należą do języka C. Funkcja biblioteki standardowej C i C++ `strcpy()` kopiuje znaki wskazywane przez drugi argument do lokalizacji wskazywanej przez pierwszy argument, aż napotka znak końca łańcucha (zero końcowe). Ten znak końca również jest kopiowany.

XX Taki kod jest rozwlekły i podatny na błędy. Wymaga uciążliwej arytmetyki wskaźników, a poza tym — kto ma wywołać `free()` dla pamięci zwróconej przez `cat2()`? Zdecydowanie zalecamy korzystanie z typu `std::string` lub innych wysokopoziomowych typów łańcuchowych zamiast z łańcuchów w stylu języka C. Jako odbiorca lub opiekun kodu wolisz patrzeć na `cat()` czy `cat2()`?

Dlaczego w ogóle ktoś zaprojektował coś takiego jak łańcuchy w stylu języka C? W czasach gdy powstały łańcuchy w stylu C, rozmiar pamięci komputera mierzono w dziesiątkach, góra setkach kilobajtów. Pamiętam, jak byłem zachwycony, kiedy dostałem komputer z całym megabajtem pamięci! Łańcuchy w stylu języka C były wtedy bliskie doskonałości w kategoriach złożoności obliczeniowej i czasowej dla programów, jakie wówczas tworzono. Powód był prosty: nie wykonywały żadnych działań ani nie przechowywały żadnych dodatkowych informacji, które mogłyby okazać się zbędne. Co więcej, pierwsi użytkownicy łańcuchów w stylu języka C byli znacznie lepszymi programistami niż dzisiejszy przeciętny programista — po prostu nie popełniali większości oczywistych błędów programistycznych.

AA Dlaczego więc ktoś miałby pisać taki kod dziś? Jest wielu programistów C, którzy nie mają alternatywy i często przenoszą swoje nawyki do C++. Ponadto sporo programistów wierzy, że potrafi pisać optymalny kod bez popełniania błędów. Prawie zawsze się mylą.

W rzeczywistości łańcuch w stylu języka C ma wiele cech, które mogą zaskoczyć osobę nieobeznaną z tym podejściem do sprawy. Łańcuchy w stylu C są tak naprawdę wskaźnikami. Na przykład operator = przypisuje wskaźniki, a nie wartości, które one wskazują.

Ostrzegaliśmy! Używaj `std::string`, jeśli tylko masz taką możliwość. Jeśli nie — zapoznaj się z dodatkowymi materiałami i dokumentacją (np. [cppref])

16.4. Alternatywy dla wskaźników

Wskaźników można używać praktycznie do wszystkiego i dlatego właśnie staramy się ich unikać — patrząc na kod w dużym stopniu oparty na wskaźnikach, często nie jesteśmy w stanie jednoznacznie określić zamiaru programisty. To sprawia, że wiele zastosowań wskaźników jest podatnych na błędy. Warto rozważyć alternatywne narzędzia: AA

- Do przechowywania kolekcji wartości należy używać kontenerów z biblioteki standardowej, takich jak `vector`, `set` (podrozdział 20.5), `map` (podrozdział 20.2), `unordered_map` (20.3) czy `array` (16.4.2).
- Do przechowywanie łańcuchów znaków używaj klas `string` z biblioteki standardowej (podrozdział 2.4, punkt 9.10.3 PPP2, podrozdział 23.2).
- Do wskazywania obiektów, których jesteśmy właścicielami (czyli takich, które musimy usunąć), używaj `unique_ptr` (punkt 18.5.2) lub `shared_ptr` (punkt 18.5.3).
- Do wskazywania ciągłych sekwencji elementów, których nie jesteś właścicielem, używaj `span` (punkt 16.4.1).
- Aby systematycznie unikać dereferencji wskaźnika do `null`, używaj `not_null` (punkt 16.4.3).

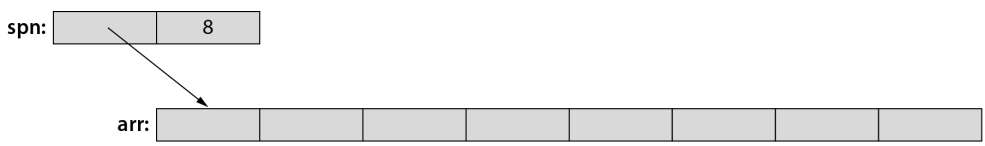
Często istnieją alternatywy dla elementów biblioteki standardowej, jednak traktuj bibliotekę standardową jako domyślny wybór.

16.4.1. Typ span

Większość przypadków użycia wskaźników obejmuje monitorowanie liczby elementów wskazywanych przez wskaźnik. Większość problemów, które nie mają związku z własnością, bierze się z błędnego liczenia elementów. Dlatego stosujemy oczywiste lekarstwo: „wskaźnik”, który monitoruje liczbę swoich elementów. Ten typ nazywa się `span`. Na przykład:

```
int arr[8];
span spn {arr}; // span<int> wskazujący 8 liczb całkowitych
```

Można to zobrazować schematycznie:



Tutaj `spn` zdefiniowano bez określania typu i liczby elementów — obie informacje zostały wydedukowane z definicji zmiennej `arr`. Nie zawsze mamy dostęp do tych informacji i nie zawsze chcemy całą zawartość tablicy. W takich przypadkach musimy wyrażać się bardziej precyzyjnie:

```
const int max = 1024;
int buf[max];
span<int>sp {buf,max/2}; // Pierwsza połowa buf
```

Używając typu span, możemy korzystać ze sprawdzania zakresu i zakresowej pętli for:

```
void test(span<int> s)
{
    cout << "rozmiar: " << s.size() << '\n';
    for (int x : s)
        cout << x << '\n';
    try {
        int y = s[size()];
    }
    catch (...) {
        cout << "Mamy sprawdzanie zakresu\n";
        return;
    }
    cout << "Brak sprawdzania zakresu! Buuu!\n";
    terminate(); // Koniec programu
}
```

Niestety std::span nie daje gwarancji sprawdzania zakresu, choć wersja z PPP_support tak.

16.4.2. Typ array

Wbudowana tablica zamienia się we wskaźnik przy najmniejszym pretekście. Wskaźnik ten nie zawiera informacji na temat rozmiaru i można go pomylić ze wskaźnikami, które powinny zostać usunięte. W bibliotece standardowej jest też typ tablicowy, który nie zamienia się automatycznie we wskaźnik. Na przykład:

```
std::array<int,8> arr { 0,1,2,3,4,5,6,7 };
int* p = arr; // błąd (i dobrze)
```

W odróżnieniu od wektora typ array nie jest uchwytem do elementów alokowanych w innym miejscu:

arr:	0	1	2	3	4	5	6	7

Tablica typu array, podobnie jak wbudowana tablica, jest typem prostszym i mniej elastycznym niż vector, co można traktować jako wadę i zaletę. Jej rozmiar nie jest przechowywany w pamięci, lecz zapamiętywany przez system typów. Tablica nie korzysta z pamięci dynamicznej, chyba że utworzysz ją za pomocą operatora new. Korzysta tylko z tego rodzaju pamięci (np. stosu lub pamięci statycznej), w którym została utworzona — i to może mieć znaczenie.

Niestety liczba elementów tablicy nigdy nie jest dedukowana automatycznie, więc trzeba ją podać jawnie. Z drugiej strony, jeśli elementów jest więcej niż wartości inicjalizujących, pozostałe elementy zostaną zainicjalizowane domyślnie. Na przykład:

```
array<string,4> as { "Witaj,", " ", "świecie!" }; // as[3] jest pustym łańcuchem
```

16.4.3. Typ not_null

Spójrz na możliwą implementację funkcji strlen():

```
int strlen(const char* p)
{
```

```
    if (p==nullptr)
        return 0;
    int n = 0;
    while (*p++)
        ++n;
    return n;
}
```

Czy test na `nullptr` jest tutaj konieczny? Gdyby go nie było, to wywołanie `strlen(p)`, gdzie `p` jest wskaźnikiem `null`, z wielkim prawdopodobieństwem spowodowałoby awarię programu. Jeśli tak się nie stanie, to zwróci błędny wynik. A jeśli mamy taki test, to czy powinien on zwrócić jakiś wynik? Innymi słowy, czy powinniśmy udawać, że (wyimaginowany) łańcuch wskazywany przez `nullptr` ma zero elementów? A może lepiej byłoby zgłosić błąd (np. wyrzucić wyjątek)?

WYPRÓBUJ

Przyjrzyj się definicji funkcji `std::strlen()`, aby zobaczyć, czego wymaga standard. Następnie wypróbuj `char* p = nullptr; size_t x = strlen(p);`, aby zobaczyć, co robi Twoja implementacja.

Za każdym razem gdy używamy wskaźników, natychmiast pojawia się pytanie, co zrobić z `nullptr`. Czy `nullptr` ma określone znaczenie? Czy jest to błąd? Jeśli to błąd, to kto powinien go przechwytywać — konstrukcja wywołująca czy wywoływana? Odpowiedzi mogą być obecne w dokumentacji, ale nie zawsze czytamy podręczniki. `PPP_support` zawiera proste rozwiązanie: typ, który sprawdza, czy jego argument jest `nullptr`, i zgłasza `not_null_error`, jeśli tak. Po tym teście obiekt typu `not_null` zachowuje się jak zwykły wskaźnik. Gdyby funkcja `strlen()` nie dopuszczała `nullptr`, moglibyśmy zdefiniować ją w taki sposób:

```
int strlen(not_null<const char*> p)
{
    int n = 0;
    while (*p++)
        ++n;
    return n;
}
```

Jeśli funkcja `strlen()` nie jest zdefiniowana w taki sposób — a jako funkcja języka C z lat 70. nie jest — to implementator musi zdecydować, czy zaufać wywołującemu, czy zabezpieczyć się przed możliwością pojawienia się argumentu `nullptr`. Dodatkowo każdy użytkownik musiałby sam zadbać o to, by żaden argument przekazywany do `strlen` nie był `nullptr`. Tak wygląda życie na niższych poziomach abstrakcji. Idealnie byłoby unikać takich dylematów, korzystając z typów wyższego poziomu, jak `vector` czy `string`. Gdy nie jest to możliwe, należy używać `not_null` dla argumentów wskaźnikowych pochodzących z kodu, nad którym nie mamy kontroli.

16.5. Przykłady: palindromy

Wystarczy tych technicznych przykładów! Czas na zagadkę. **Palindrom** to słowo, które brzmi tak samo czytane od przodu jak i wstecz. Przykładami mogą być wyrazy **ala**, **malajalam** czy **anna**. Istnieją dwie podstawowe techniki sprawdzania, czy dane słowo jest palindromem:

- Utworzenie kopii liter w odwróconej kolejności i porównanie jej z oryginałem.
- Sprawdzenie, czy pierwsza litera jest taka sama jak ostatnia, czy druga jest taka sama jak przedostatnia itd. aż do środka.

Tutaj przedstawimy zastosowanie drugiej z wymienionych metod. Można to wyrazić w kodzie na wiele sposobów w zależności od sposobu reprezentacji słowa oraz zapamiętywania, dokąd się doszło z porównywaniem znaków. Napiszemy niewielki program sprawdzający, czy słowa są palindromami, na kilka sposobów, aby pokazać, jak różne narzędzia językowe wpływają na wygląd i działanie kodu.

16.5.1. Wykorzystanie łańcuchów

Na pierwszy ogień pójdzie wersja programu wykorzystująca standardowy typ `string` i typ `int` do śledzenia, ile liter zostało już porównanych:

```
bool is_palindrome(const string& s)
{
    int first = 0;           // Indeks pierwszej litery
    int last = s.length()-1; // Indeks ostatniej litery
    while (first < last) {   // Nie doszliśmy jeszcze do środka
        if (s[first]!=s[last]) return false;
        ++first;             // Przesuwa w przód
        --last;              // Przesuwa wstecz
    }
    return true;
}
```

Jeśli dojdziemy do środka, nie znajdując żadnej różnicy, zwracamy wartość `true`. Zachęcamy do sprawdzenia, czy kod ten poprawnie zadziała, gdy nie będzie żadnych liter lub będzie tylko jedna litera, parzysta liczba liter oraz nieparzysta liczba liter w łańcuchu. Oczywiście nie należy oceniać poprawności kodu, opierając się tylko na logice. Należy także przeprowadzić testy. Funkcję `is_palindrome()` można przetestować następująco:

```
int main()
{
    for (string s; cin>>s; ) {
        cout << s;
        if (!is_palindrome(s)) cout << " nie";
        cout << " jest palindromem.\n";
    }
}
```

Głównym powodem, dla którego użyliśmy typu `string`, jest fakt, że typ ten dobrze sprawdza się w pracy ze słowami. Do łańcucha można łatwo wczytać słowo ograniczone spacją i typ `string` będzie znał jego rozmiar. Gdybyśmy chcieli przetestować funkcję `is_palindrome()` na łańcuchach zawierających spacje, moglibyśmy do wczytywania użyć funkcji `getline()`

(podrozdział 11.5). Wówczas łańcuchy takie jak **ah ha** i **as df fd sa** zostałyby zakwalifikowane jako palindromy.

16.5.2. Wykorzystanie tablic

Co zrobić, jeśli nie będzie można użyć łańcuchów (ani wektorów), przez co zostaną do dyspozycji tylko tablice do przechowywania znaków? Zobaczmy:

```
bool is_palindrome(const char s[], int n)
    // s wskazuje pierwszy znak tablicy n znaków
{
    int first = 0;          // Indeks pierwszej litery
    int last = n-1;         // Indeks ostatniej litery
    while (first < last) { // Nie doszliśmy jeszcze do środka
        if (s[first]!=s[last]) return false;
        ++first;            // Przesuwa w przód
        --last;             // Przesuwa wstecz
    }
    return true;
}
```

Aby przetestować tę funkcję `is_palindrome()`, musimy najpierw wczytać znaki do tablicy. Jeden z bezpiecznych sposobów na zrobienie tego (bez ryzyka przepełnienia tablicy) jest następujący:

```
istream& read_word(istream& is, char* buffer, int max)
    // Wczytuje najwyżej max–1 znaków z is do bufora
{
    is.width(max); // Wczytuje najwyżej max–1 znaków w znajdującym się dalej >>
    is >> buffer;  // Wczytuje zakończone spacją słowo
                   // i dodaje zero za ostatnim wczytanym znakiem
    return is;
}
```

Ustawienie odpowiedniej szerokości strumienia `istream` zapobiega przepełnieniu bufora przez następną operację >>. Niestety nie wiadomo też, czy odczyt zakończył się z powodu napotkania spacji czy napełnienia bufora (co oznacza, że trzeba wczytać jeszcze więcej znaków). Poza tym kto pamięta, jak dokładnie działa funkcja `width()` dla danych wejściowych? Jako bufory wejściowe lepiej sprawdzają się standardowe typy `string` i `vector`, ponieważ rozszerzają się w miarę napływu danych. Końcowe `0` jest potrzebne, ponieważ większość często wykorzystywanych operacji na tablicach znaków (łańcuchach w stylu C) przyjmuje, że znak ten oznacza koniec. Jeśli użyjemy funkcji `read_word()`, możemy napisać taki kod:

```
int main()
{
    constexpr int max = 128;
    for (char s[max]; read_word(cin,s,max); ) {
        cout << s;
        if (!is_palindrome(s,strlen(s))) cout << " nie";
        cout << " jest palindromem.\n";
    }
}
```

Wywołanie `strlen(s)` zwraca liczbę znaków w tablicy po wywołaniu funkcji `read_word()`, a instrukcja `cout<<s` wysyła na wyjście znaki znajdujące się w tablicy, aż napotka końcowe `0`.

AA To rozwiązanie z użyciem tablicy jest naszym zdaniem znacznie bardziej zagmatwane niż z użyciem typu `string`. Sytuacja jeszcze się pogarsza, gdy przychodzi do obsługi naprawdę długich łańcuchów. Zobacz 10. zadanie pracy domowej.

16.5.3. Wykorzystanie wskaźników

Do identyfikowania znaków można też użyć wskaźników:

```
bool is_palindrome(const char* first, const char* last)
    // Wskaźnik first wskazuje pierwszą literę, last — ostatnią
{
    while (first < last) { // Nie doszliśmy jeszcze do środka
        if (*first!=*last) return false;
        ++first;           // Przesuwa w przód
        --last;            // Przesuwa wstecz
    }
    return true;
}
```

To jest być może najbardziej przejrzysta funkcja `is_palindrome()`, jaka powstała do tej pory, ale ta prostota jest okupiona przerzuceniem obowiązku pobierania zakresu na użytkowników:

```
int main()
{
    const int max = 128;
    char s[max];
    for (char s[max]; read_word(cin,s,max); ) {
        cout << s;
        if (!is_palindrome(&s[0],&s[strlen(s)-1])) cout << " nie";
        cout << " jest palindromem.\n";
    }
}
```

Dla zabawy przerobimy funkcję `is_palindrome()`:

```
bool is_palindrome(const char* first, const char* last)
    // Wskaźnik first wskazuje pierwszą literę, last — ostatnią
{
    if (first<last) {
        return (*first==*last) ? is_palindrome(first+1,last-1) : false;
    return true;
}
```

Ten kod stanie się oczywisty, gdy sparafrazujemy definicję **palindromu**: słowo jest palindromem, jeśli jego pierwszy i ostatni znak są takie same oraz powstały po ich usunięciu ciąg znaków jest palindromem.

16.5.4. Wykorzystanie typu span

Zasadniczo typ span jest po prostu inną, i zazwyczaj lepszą, alternatywą dla tablic i wskaźników. Na przykład:

```
bool is_palindrome(span<char> s)
{
    return (s.size()) ? is_palindrome(s.data(),s.data()+s.size()) : true;
                                                    // Implementacja przy użyciu wskaźników
}
```

Typ `span`, jak wszystkie typy z biblioteki standardowej, ma tyle przydatnych funkcji składowych, że ich opis nie zmieściłby się w tej książce. Tutaj użyliśmy funkcji `data()`, która zwraca wskaźnik do pierwszego elementu obiektu `span`. Ponieważ pusty `span` nie ma pierwszego elementu, najpierw sprawdziliśmy jego rozmiar za pomocą wyrażenia `s.size()`.

Ta funkcja `is_palindrome()` jest przykładem tego, jak można przejść z jednego stylu do innego — tutaj z nowoczesnego podejścia z użyciem `span` do starszego, z wykorzystaniem wskaźników. W dużych, wieloletnich projektach nie jest niczym niezwykłym, że współistnieje kilka stylów, ponieważ narzędzia i mody ulegają zmianom. Możemy też dokonać odwrotnego przejścia:

```
bool is_palindrome(const char* first, const char* last)
{
    return is_palindrome(span<char>{first,last-first}); // Implementacja przy użyciu span
}
```

Niestety nie istnieje prosty i niezawodny test pozwalający stwierdzić, czy para wskaźników `first` i `last` jest możliwa, więc lepiej jest używać `span`:

```
bool is_palindrome(span<char> s)
{
    if (s.size()<2)
        return true;
    return (s.front()==s.back()) ? is_palindrome(span<int>{s.data()+1,s.size()-2}) :
    ↪false;
}
```

Ćwiczenia

W tym rozdziale będą dwa podobne zestawy ćwiczeń powtórzeniowych. Zadaniem pierwszego jest utrwalenie wiedzy o tablicach, a drugiego o wektorach. Rozwiąż ćwiczenia z obu zestawów i porównaj, które z nich wymagały więcej wysiłku.

Ćwiczenia dotyczące tablic:

1. Zdefiniuj globalną tablicę liczb typu `int` o nazwie `ga` i zainicjuj ją wartościami `1`, `2`, `4`, `8`, `16` itd.
2. Zdefiniuj funkcję o nazwie `f()` przyjmującą argument w postaci tablicy liczb typu `int` oraz argument typu `int` wskazujący liczbę elementów w tej tablicy.
3. W funkcji `f()`:
   - Zdefiniuj lokalną tablicę dziesięciu liczb typu `int` o nazwie `la`.
   - Skopiuj wartości z tablicy `ga` do `la`.
   - Wydrukuj elementy tablicy `la`.
   - Zdefiniuj wskaźnik `p` do typu `int` i zainicjuj go alokowaną w pamięci wolnej tablicą zawierającą taką samą liczbę elementów, jak tablica przekazana jako argument.
   - Skopiuj wartości z tablicy przekazanej jako argument do tablicy alokowanej w pamięci wolnej.

- Wydrukuj elementy z tablicy alokowanej w pamięci wolnej.
- Usuń tablicę alokowaną w pamięci wolnej.

4. W funkcji `main()`:
   - Wywołaj funkcję `f()` z tablicą `ga` jako argumentem.
   - Zdefiniuj tablicę `aa` z dziesięcioma elementami i zainicjuj ją dziesięcioma pierwszymi wartościami silni (`1`, `2*1`, `3*2*1`, `4*3*2*1` itd.).
   - Wywołaj funkcję `f()`, przekazując jej jako argument tablicę `aa`.

Ćwiczenia dotyczące wektora z biblioteki standardowej:

1. Zdefiniuj globalny wektor `vector<int>` o nazwie `gv` i zainicjuj go dziesięcioma liczbami typu `int` — `1`, `2`, `4`, `8`, `16` itd.
2. Zdefiniuj funkcję o nazwie `f()` przyjmującą argument typu `vector<int>`.
3. W funkcji `f()`:
   - Zdefiniuj lokalny wektor `vector<int>` o nazwie `lv` i o takiej samej liczbie elementów, co wektor przekazany w argumencie.
   - Skopiuj wartości z wektora `gv` do `lv`.
   - Wydrukuj elementy wektora `lv`.
   - Zdefiniuj lokalny wektor `vector<int>` o nazwie `lv2` i zainicjuj go tak, aby był kopią wektora przekazanego jako argument.
   - Wydrukuj elementy z wektora `lv2`.
4. W funkcji `main()`:
   - Wywołaj funkcję `f()` z wektorem `gv` jako argumentem.
   - Zdefiniuj wektor `vector<int>` o nazwie `vv` i zainicjuj go dziesięcioma pierwszymi wartościami silni (`1`, `2*1`, `3*2*1`, `4*3*2*1` itd.)
   - Wywołaj funkcję `f()` z wektorem `vv` jako argumentem.

Powtórzenie

1. Co znaczą słowa: „Caveat emptor!”?
2. Co to jest tablica?
3. Jak kopiuje się tablice?
4. Jak inicjalizuje się tablice?
5. Kiedy lepiej zastosować jako argument wskaźnik zamiast referencji? Dlaczego?
6. Kiedy lepiej jest użyć obiektu typu `span` zamiast wskaźnika? Dlaczego?
7. Czym `std::array` różni się od tablicy wbudowanej?

8. Co dobrego daje sprawdzanie zakresu?
9. Jakich informacji potrzeba do sprawdzania zakresu?
10. Co dobrego daje `not_null`?
11. Co to jest łańcuch w stylu języka C?
12. Co to jest palindrom?

Terminologia

tablica	`array`	`->`	arytmetyka wskaźników
wskaźnik	`not_null`	`[]`	łańcuch w stylu języka C
palindrom	`*`	`strlen()`	indeksowanie
`span`	`&`	błąd zakresu	dereferencja `nullptr`

Praca domowa

1. Napisz funkcje `void to_lower(char* s)`, która zamienia wszystkie wielkie litery w łańcuchu w stylu języka C `s` na małe. Na przykład, napis `Witaj, Świecie` powinien zmienić się w `witaj, świecie!`. Nie używaj żadnych funkcji z biblioteki standardowej. Łańcuch w stylu języka C jest tablicą znaków zakończoną zerem, więc jeśli znajdziesz znak o wartości `0`, to znaczy, że jesteś na końcu.
2. Napisz funkcję `char* strdup(const char*)` kopiującą łańcuchy w stylu języka C do pamięci alokowanej w pamięci wolnej. Nie używaj żadnej funkcji z biblioteki standardowej.
3. Napisz funkcję `char* find_x(const char* s, const char* x)`, znajdującą pierwsze wystąpienie łańcucha w stylu C `x` w `s`.
4. Napisz funkcję `int str_cmp(const char* s1, const char* s2)`, porównującą łańcuchy w stylu C. Niech zwraca liczbę ujemną, jeśli `s1` znajduje się w porządku leksykograficznym przed `s2`, zero, jeśli `s1` i `s2` są równe, oraz liczbę dodatnią, jeśli `s1` znajduje się w porządku leksykograficznym za `s2`. Nie używaj funkcji z biblioteki standardowej. Nie używaj operatora indeksowania. W zamian użyj operatora dereferencji (*).
5. Przemyśl, co się stanie, jeśli funkcjom `strdup()`, `findx()` oraz `str_cmp()` zostanie podany argument wskaźnikowy niebędący łańcuchem w stylu C. Sprawdź! Najpierw znajdź sposób na uzyskanie wskaźnika typu `char*`, który nie wskazuje tablicy zakończonej zerem (nie rób tego nigdy w realnych — tzn. nieeksperymentalnych — programach, ponieważ możesz spowodować kompletną demolkę). Spróbuj „fałszywych łańcuchów w stylu C” alokowanych w pamięci wolnej i na stosie. Jeśli wyniki wydają się sensowne, wyłącz tryb debugowania. Przeprojektuj i ponownie zaimplementuj te trzy funkcje, aby przyjmowały jeszcze jeden argument określający maksymalną dozwoloną liczbę elementów w argumentowych łańcuchach. Następnie przetestuj je na poprawnych i „złych” łańcuchach w stylu C.

6. Sprawdź, co się stanie, jeśli funkcji `strcmp()` z biblioteki standardowej przekażesz jako argument wskaźnik, który nie jest łańcuchem w stylu języka C.
7. Napisz funkcję `string cat_dot(const char* s1, const char* s2)`, łączącą dwa łańcuchy, wstawiając między nie kropkę. Na przykład wywołanie `cat_dot("Niels", "Bohr")` powinno zwrócić łańcuch `Niels.Bohr`.
8. Napisz wersję funkcji `cat_dot()`, która pobiera argumenty `const string&`.
9. Zmodyfikuj funkcję `cat_dot()` z dwóch poprzednich zadań, aby zamiast kropki wstawiała dowolny łańcuch podany przez użytkownika jako trzeci argument.
10. Napisz zmodyfikowane wersje funkcji `cat_dot()` z poprzednich zadań, które przyjmują jako argumenty łańcuchy w stylu C i zwracają łańcuchy w stylu C alokowane w pamięci wolnej. Nie używaj funkcji z biblioteki standardowej ani typów z implementacji. Przetestuj te funkcje na kilku łańcuchach. Nie zapomnij o zwolnieniu (`delete`) całej pamięci alokowanej w pamięci wolnej (za pomocą operatora `new`). Porównaj ilość wysiłku potrzebnego do wykonania tego zadania z zadaniami 5. i 6.
11. Przepisz wszystkie funkcje z podrozdziału 16.5, stosując technikę tworzenia odwróconej kopii łańcucha i porównywania go z oryginałem, np. funkcja pobiera łańcuch `"dom"`, tworzy łańcuch `"mod"` i porównuje te dwa łańcuchy, aby stwierdzić, że nie są identyczne, a więc **dom** nie jest palindromem.
12. Spójrz na „tablicowe" rozwiązanie problemu palindromu z punktu 16.5.2. Popraw je, aby obsługiwało też długie łańcuchy poprzez: a) raportowanie, że łańcuch był za długi i b) zezwolenie na wpisywanie dowolnie długich łańcuchów. Skomentuj stopień złożoności tych dwóch wersji.
13. Zaimplementuj własną wersję gry *Hunt the Wumpus*. Jest to prosta (bez interfejsu graficznego) gra komputerowa, której autorem jest Gregory Yob. W ciemnej jaskini składającej się z połączonych ze sobą pomieszczeń mieszka śmierdzący potwór Wumpus. Zadaniem gracza jest zabić Wumpusa za pomocą łuku i strzał. Poza potworem w jaskini czyhają jeszcze dwa niebezpieczeństwa: otchłanie i gigantyczne nietoperze. Jeśli gracz wejdzie do pomieszczenia z otchłanią, następuje koniec gry. Jeśli wejdzie do pomieszczenia z nietoperzem, ten przeniesie go do innego pomieszczenia. Wejście do pomieszczenia, w którym znajduje się Wumpus, kończy się pożarciem. Gdy gracz wchodzi do pomieszczenia, jest informowany, czy grozi mu niebezpieczeństwo:

    - „Czuję Wumpusa" — potwór jest w jednym z bezpośrednio sąsiadujących pomieszczeń.
    - „Czuję podmuchy wiatru" — jedno z bezpośrednio sąsiadujących pomieszczeń jest otchłanią.
    - „Słyszę nietoperza" — w jednym z bezpośrednio sąsiadujących pomieszczeń jest gigantyczny nietoperz.

    Dla wygody pomieszczenia są ponumerowane. Każde jest połączone tunelami z trzema innymi pomieszczeniami. Gdy gracz wchodzi do pomieszczenia, zostają wyświetlone następujące informacje: „Jesteś w pomieszczeniu nr 12. Tunele prowadzą do

pomieszczeń 1, 13 oraz 4. Idziesz dalej, czy strzelasz?". Możliwe odpowiedzi to `m13` (przejście do pomieszczenia nr 13) oraz `s13-4-3` (strzał przez pomieszczenia 13, 4 i 3). Strzała ma zasięg trzech pomieszczeń. Na początku gry gracz ma pięć strzał. Problem ze strzelaniem polega na tym, że budzi ono Wumpusa, który po przebudzeniu przechodzi do jednego sąsiedniego pokoju, w którym może znajdować się gracz.

Najtrudniejszą częścią implementacji tej gry jest prawdopodobnie stworzenie jaskini i określenie połączeń tunelowych między pomieszczeniami. Można użyć generatora liczb losowych (np. funkcji `randint()` z `PPP_support`) do tworzenia różnych układów pomieszczeń w różnych uruchomieniach gry oraz przemieszczania nietoperzy i Wumpusa. Wskazówka: zapewnij sobie sposób na wydrukowanie danych debugowania dotyczących stanu jaskini.

Podsumowanie

Wskaźniki i tablice są wszechobecne w kodzie w C oraz w starszym kodzie w C++. Na przykład literały łańcuchowe to łańcuchy w stylu języka C. Dlatego musimy rozumieć sposób ich działania i nauczyć się unikać związanych z nimi błędów. Typy `span`, `array` i `not_null` z biblioteki standardowej rozwiązują wiele problemów dotyczących błędów zakresu i mogą być używane tam, gdzie nie można konsekwentnie stosować typów wyższego poziomu, takich jak `vector`.

Skorowidz

Są dwa rodzaje wiedzy:
sami znamy temat albo wiemy,
gdzie szukać informacji.

— Samuel Johnson

A

abstrakcja, 78
adres, 469
akumulacja, 654
akumulator, accumulator, 655
algorytm
  accumulate(), 654
  binary_search(), 661
  equal_range(), 661
  find(), 645
  find_if(), 648
  kopiowania, 659
  sort(), 660
  unique_copy(), 660
algorytmy
  biblioteki standardowej, 644
  numeryczne, 654
alias typu, 597
alokacja obiektów, 473
alokatory, 559
alternacja, 160
analiza, 147
analizator składni, 159
animacja, 456
argumenty funkcji, 132
  domyślne, 418
  formalne, 94, 221
  konwersja, 229
  przekazywane
    przez referencję, 226, 227
    przez stałą referencję, 223
    przez wartość, 223, 227
  sprawdzanie, 229
arytmetyka wskaźników, pointer arithmetic, 499
asercja, assertion, 132

B

bezpieczeństwo typów, 67
biblioteka, 44
  FLTK, 375
  GUI, 327
  interfejsowa, 339
  okien, 324, 325
  Qt, 325, 327, 339, 443
  STL, 465, 585
bit, 66
blok
  catch, 194, 195
  try, 194, 195
bloki, 91
  zagnieżdżanie, 219
błędny argument, 118
błędy
  czasu wykonania, 44, 116
  debugowanie, 128

błędy
  działania, 110
  kompilacji, 44, 110, 112
  konsolidacji, 44, 110, 115
  logiczne, 44, 110
  obsługa, 117, 118, 121, 184
  odzyskiwanie sprawności, 194
  pomyłki o jeden, 123
  raportowanie, 120, 262
  składni, 112
  typów, 113
  wartości brzegowych, 123
  wejścia i wyjścia, 289
  zakresu, range error, 122, 123, 497
  znajdowanie, 127
  źródła, 111
bug, 44, 128

C

Ctr+D, 61, 290
Ctr+Z, 61, 290
czas, 625
  kompilacji, 234
czyste funkcje wirtualne, 392, 405

D

dane
  organizacja, 587
  przechowywanie, 582, 587
  przetwarzanie, 582, 586
  wejściowe, 52, 76
    nieprawidłowe, 117, 124
data, 627
dealokacja pamięci wolnej, 477
debugowanie, debugging, 44, 129, 128
  kodu GUI, 457
  konstruktorów i destruktorów, 532
dedukcja typów, 70
definiowanie, 53, 54, 65, 212, 213
  funkcji, 94
    składowych, 259
    wirtualnych, 401
  kontenera, 630
  lambdy, lambda introducer, 421
  operatorów
    wejściowych, 298
    wyjściowych, 297
  przycisku, 444
deklaracja, 65, 211, 213
  funkcji, 95
  stałej, 214
  using, 240, 597
  wskaźnika, 475
  zmiennej, 214
deklarowanie argumentów, 221
derywacja, 398
destruktor, destructor, 478, 529, 545
  domyślny, 479
  wirtualny, 480
dokumentacja, 193
domyślna inicjalizacja składowej, 173
dostęp
  do referencji zwrotnej, 545
  do składowej ->, 475, 491
  do wektorów, 517
dostosowywanie strumieni
  wejścia/wyjścia, 633
drzewo
  czerwono-czarne, red-black tree, 616
  zrównoważone, 619
dyrektywa, 240
  #include, 46, 52, 243, 341
dziedziczenie, 398, 557
  implementacji, 407
  interfejsu, 407

E

edytor tekstu, 602
  iteracja, 605
  operacje, 603
  wiersze, 604
elipsa, 373
elizja kopiowania, copy elision, 528
etykieta
  private:, 253
  public:, 253

F

figury
  geometryczne, 329
    funkcja sinus, 331
    prostokąt, 333
    tekst, 336
    wielokąt, 332
    wypełnianie kolorem, 335
  zamknięte, 363
format
  GIF, 380
  JPEG, 380
formatowanie, 306
  danych wyjściowych, 307–310
formaty liczb zmiennoprzecinkowych, 309
frameworki testowe, 136
funkcja, 40
  accumulate(), 656
  add(), 351, 359, 484
  advance(), 484
  app.exec(), 451
  at(), 562
  attach(), 350
  begin(), 534, 596, 632
  binary_search(), 661, 662
  cin.clear(), 289
  Circle
    draw_lines(), 400
  color(), 376
  connect(), 446
  constexpr(), 235
  copy(), 633, 634, 659
  copy_if(), 660
  destroy_at(), 601
  draw(), 396, 397
  draw_specifics(), 352, 359, 369, 376, 396–400
  duration_cast, 626
  end(), 534, 596, 632
  equal_range(), 662
  erase(), 484, 599, 601
  expect(), 133
  fill(), 528
  find(), 484, 623, 645, 647
  find_if(), 648, 650
  format(), 312
  free(), 504
  get(), 174, 202, 482
  get_int(), 447
  get_string(), 447
  hide(), 445
  high(), 584, 585, 591
  inner_product(), 657, 658
  insert(), 483, 484, 599, 601, 621
  intersect(), 365
  isdigit(), 311
  Link::insert(), 487
  lower_bound(), 662
  main(), 40, 191
  make_unique(), 370
  make_vec(), 574
  malloc(), 504
  move(), 407, 601
  move_backward(), 602
  next(), 450
  now(), 625
  number_of_points(), 352
  operator[](), 563
  operator<<(), 297
  operator=(), 539, 572, 573
  parametryzowana, parametrized function, 553
  point(), 352
  precision(), 309
  push_back(), 98, 100, 200, 539, 542
  push_front(), 596
  putback(), 201
  random_int(), 137, 138
  reserve(), 541, 560, 602
  resize(), 539, 541
  seed(), 138
  set(), 482
  set_color(), 355, 376
  set_fill_color(), 366
  set_label(), 350
  set_major(), 374
  set_mask(), 381
  set_minor(), 374
  set_point(), 394
  set_style(), 355
  setprecision(), 309
  setw(), 310

funkcja
Shape
move(), 401
show(), 445
size(), 98, 122, 562, 598
skrótu, hash function, 623
sort(), 100, 660
strlen(), 503, 506
suspicious(), 566
swap(), 227, 534, 572
tolower(), 311
toupper(), 311
unget(), 290, 293, 298
uninitialized_fill(), 561
uninitialized_move(), 572
unique_copy(), 634
upper_bound(), 662
v.size(), 97
vector::reserve(), 541
vector::capacity(), 541
wait_for_button(), 398
Window::attach(), 445
x_max(), 351
funkcje, 93
argumenty, 221
domyślne, 418
nieprawidłowe, 132
przekazywanie, 223, 226, 227, 229
definiowanie, 94
deklarowanie, 95
implementacja wywołań, 230
klasy Shape, 406
klasyfikujące znaki, 311
lokalne, 219
pomocnicze, 256, 274
rekurencyjne, 233
rysowanie wykresów, 414, 423
składowe, member function, 98, 218, 257–261, 274
stałe, 272
wektora, 482
typ zwrotny, 213, 221
typy argumentów, 213
używanie, 191
wirtualne czyste, 392, 405
wirtualne, 398, 401
wykładnicze, 423
wywoływanie, 220
zwracanie wartości, 222

G

generator typów, type generator, 552
generowanie
danych, 625
destruktorów, 479
globalna inicjalizacja, 238
graficzny interfejs użytkownika, GUI, 149, 322, 439
grafika, 322
gramatyka wyrażeń, 158, 160
gwarancje dotyczące zarządzania zasobami, 569

H

haszowanie, 624
hermetyzacja, encapsulation, 398

I

IDE, interactive development environment, 44
iloczyn skalarny, 657
implementacja, 147, 253
okna, 442
operatora [], 561
tokenów, 153
typu Token_stream, 173
wywołań funkcji, 230
zasad gramatyki, 161
inicjalizacja, 59
domyślna, 216
referencji, 500
w klasie, 173
wektora, 520
wskaźnika, 475, 500
inicjalizator, 54, 214
składowych, 272
w klasie, 272
inkrementacja, 62, 83

instrukcja, statement, 40
  for, 91
  if, 84
  import, 41, 52
  return, 222
  switch, 86, 192
  while, 61, 89, 90
instrukcje, 83
  porządek wykonywania, 236
  złożone, 91
interfejs, 253
  klas, 267
  przeglądarki internetowej, 440
  użytkownika, 440
inwersja kontroli, 451
iteracja, 75, 89
iterator, 589, 602
  ciągły, 636
  dostępu swobodnego, random-access iterator, 630, 636
  dwukierunkowy, bidirectional iterator, 630, 636
  listy, 595
  operacje, 589
  postępujący, 636
  wejściowy, 636
  wyjściowy, 636

J

języki programowania, 38

K

kalkulator, 149
  druga wersja, 170
  gramatyki, 158, 160
  liczby ujemne, 186
  obsługa zmiennych, 197
  pierwsza wersja, 167
  reszta z dzielenia, 187
  struktura programu, 176
  tablica, 156
  tokeny, 152–155
  wczytywanie liczb, 175
  wczytywanie tokenów, 174
  zamiana gramatyki w kod, 161
kanwa rysunku, canvas, 329
klasa, 253, *Patrz także* typ
  allocator, 559
  Axis, 330
  Color, 353
  Date, 262
  Font, 376
  Line, 348
  Lines, 350
  Link, 484
  List, 484
  Point, 348
  Shape, 391, 396, 406
  Simple_window, 324, 343
  Vector, 98, 518
klasy, 253
  abstrakcyjne, 392, 405
  bazowe, 398
  derywowane, 398
  dostęp do składowych, 404
  figur geometrycznych, 329
  interfejsu graficznego, 346
  interfejsu GUI, 347
  interfejsy, 267
  kontrola dostępu, 393
  lokalne, 219
  nazewnictwo, 389
  parametryzowane, 553
  pochodne, 392, 398
  podrzędne, 394
  składowe, 219
  tworzenie, 255
  zasady projektowania, 386
  zestaw operacji, 388
  zmienność, 390
klasyfikacja znaków, 311
kod
  maszynowy, 41
  obiektowy, 41
  źródłowy, 41
kolejność
  obliczania wartości, 237
  wykonywania instrukcji, 236
kolizja, 217
kolor, 335
komentarze, 39, 193
kompilator, 41

koncepcje, 554–557
konkatenacja, 58
konkretyzacja szablonu,
  template instantiation, 552
konsola, 440
konsolidator, 43
konstrukcja if-else, 84
konstruktor, constructor, 257
  domyślny, 202, 270, 529
  jawny, 531
  kopiujący, 406, 523, 529, 545
  przenoszący, 529, 545, 574
  z parametrami, 529
kontenery, 122, 557, 581, 587
  asocjacyjne, associative container, 616
  figur, 347
  definicja, 630
  standardowe, 629, 630
  szybkość działania, 625
kontrakt, contract, 132
kontrolka, control, 444
konwersja
  rozszerzająca, 68
  zawężająca, 68
kopiowanie, 659
  głębokie, 526
  obiektów, 270
  płytkie, 526

L

lambda, 219, 420, 653
liczby
  całkowite, 307
  losowe, 136
  ujemne, 186
  zmiennoprzecinkowe, 309
lifting, 592
linie łamane, 358
linker, 43
lista, 482
  dwukierunkowa, 483
  inicjalizatora, 260
  jednokierunkowa, 483
  powiązana, linked list, 592
    dwustronnie, doubly-linked list, 593
    jednostronnie, singly-linked list, 593
listy
  iteracja, 595
  operacje, 594
literały łańcuchowe, 39, 503
l-wartość, 79

Ł

łańcuch, 310, 503, 508
  formatu, 313

M

manipulator
  dec, 308
  oct, 308
mechanizm default_random_engine, 137
menu, 448, 452
model graficzny, 323
moduł, 241, 243
  PPP_support, 564
  std, 40

N

nadklasa, superclass, 398
nagłówek, 241, 243, 340
namespace, 239
narzędzia
  do tworzenia GUI, 448
  graficzne, 324, 325
nawias
  klamrowy, 520
  trójkątny, 97
nazwy, 54, 63
  predefiniowane, 203
  zarezerwowane, 64
niezmiennik, invariant, 259
notacja
  naukowa, 309
  stałoprzecinkowa, 309
  wykorzystująca mantysę i wykładnik,
    309

O

obiekt, 52, 54, 65
  shared_ptr, 576
  unique_ptr, 574
obiekty
  bez nazw, 369
  funkcyjne, 650, 651
  tymczasowe, 228
obraz, 337, 378
obsługa
  błędów, 121, 184, 293–296, 568, 569
  błędów wejścia i wyjścia, 289
  czasu, 625
  wyjątków, 121
odczytywanie
  danych z pliku, 429
  struktur wartości, 302
odzyskiwanie sprawności, 194
okno
  dialogowe, dialog box, 447
  programu, 442, 448
okrąg, 371
operacje, 56
  kopiowania, 659
  na iteratorach, 589
  na listach, 484, 594
operator, 56, 57, 81, 82, 535
  !=, 102
  [], 474, 561
  ||, 102
  +=, 100
  <, 102
  <=, 102
  ==, 102
  ->, 475, 491
  adresu &, 469
  delete, 566
  dereferencji *, 470, 474
  dopełnienia ~, 478
  kropki, 475
  narrow, 597
  new, 472, 473, 477, 566
  reszty z dzielenia %, 187
  sizeof, 471
  wejścia >>, 53, 298, 303–305, 310
  wyjścia <<, 39, 42, 256, 297, 312
  zawartości, 470
operatory
  porównań, 535, 536
  przeciążanie, 266
  przypisania, 62
osie, axis, 421

P

palindrom, 508
  sprawdzanie z wykorzystaniem
    łańcuchów, 508
    tablic, 509
    typu span, 510
    wskaźników, 510
pamięć, 469
  automatyczna, 472
  dynamiczna, 472
  kodu, 472
  statyczna, 472
parametry, 94, 221
  szablonowe T, 551
  wartościowe szablonów, 558
parsowanie, 159
pętla
  for, 91
  for zakresowa, 98
  oczekująca, 443
  wejściowa, 298
  while, 61, 90
pierwszy program, 38
pisanie programu, 145, 181
  kalkulatora, 149
  postawienie problemu, 146
  rozwiązanie problemu, 146
  strategia, 147
pliki, 284
  nagłówkowe, 46, 243, 340
  odczytywanie, 287, 429
  otwieranie, 286
  strukturalne
    wczytywanie, 300
  wczytywanie pojedynczej wartości, 291
  zapisywanie, 287
  źródłowe, 340
podklasa, subclass, 398
polimorficzny zasób pamięciowy, 560

polimorfizm, 553
  ad hoc, 553
  czasu działania programu, 398
  parametryczny, 553
prawie kontenery, 631
precyzja liczb zmiennoprzecinkowych, 309
predykaty składowych klas, 652
problem z zarządzaniem zasobami, 566
produkcje, 160
program, 38
  GUI, 441, 452, 457
programowanie, 38
  obiektowe, OOP, 386, 399, 407, 553, 554
  ogólne, generic programming, 553, 554, 647
  różnicowe, programming by difference, 360
projekt, 156
projektowanie, 147
  klas, 385
prostokąt, 333, 366
prototyp, 149
przechowywanie
  danych, 582
  sekwencji znaków, 608
przeciążanie operatorów, operator overloading, 266, 535
przepełnienie, 427
  bufora, buffer overflow, 497
przesłanianie, overriding, 397, 402
przestrzeń nazw, namespace, 239
przetwarzanie
  danych, 582
  wstępne, preprocessing, 244
przezroczystość, 355
przycisk, button, 441, 446
przypadek testowy, test case, 136
przypadki użycia, 150
przypisanie, 59, 62
  kopiujące, 529, 545
  przenoszące, 529, 545
pseudokod, 150
pusta sekwencja, 596

R

RAII, Resource Acquisition is Initialization, 549, 570
referencje, 224–227, 500
  do r-wartości, 528
  inicjalizacja, 500
  jako parametry, 501
rekord aktywacji funkcji, 231, 234
rekurencja
  nieskończona, infinite recursion, 162
  zstępująca, recursive descent, 170
repetycja, 160
reprezentacja danych w pamięci, 301
rezerwacja pamięci, 541
RGB, red, green, blue, 354
rozkład, 137
rozmiar
  czcionki, 375
  okna, 351
rozszerzenie
  .cpp, 244
  .h, 244
r-wartość, 79
rysowanie
  figur, 324, 396
  linii, 448
  prostokątu, 333
  wielokątu, 332
  wykresów funkcji, 331, 414, 419, 423

S

sekwencja, 160, 589, 592
selekcja, 75, 84
semantyka
  referencyjna, reference semantics, 527
  wartościowa, value semantics, 527
  wskaźnikowa, pointer semantics, 527
skalowanie danych, 432
składniki, 165
składowa klasy, 154, 253
  chroniona, protected, 398, 404
  prywatna, private, 398, 404
  publiczna, public, 404

słownik, map, 306, 616
  nieuporządkowany, unordered_map, 623, 625
  opis ogólny, 618
  wiązanie strukturalne, 618
  wyszukiwanie, 624
  zastosowanie, 621
słowo kluczowe
  auto, 70, 236
  case, 192
  catch, 121
  class, 154, 253
  const, 215, 224
  constexpr, 215, 234
  else, 86
  enum, 263
  extern, 212, 213
  int, 42
  let, 198, 202
  override, 350
  private, 172, 173
  public, 154, 172
  requires, 555
  struct, 255
  throw, 121
  try, 121
  vector, 97
  virtual, 402
  void, 94, 173, 221
sortowanie, 100, 660
specjalizacja, specialization, 552
sprawdzanie zakresu, 561
stałe, 80
  funkcje składowe, 272
  magiczne, 81, 189, 302
  symboliczne, 189
stan, 252, 259
  strumienia, stream state
    bad(), 289
    eof(), 289
    fail(), 289
    good(), 289
standard ISO, 275
statyczna kontrola typów, 67
sterownik urządzenia, 282
sterta, 472
stos, 233, 472
  wywołań, 234
struktura, 255, 300, *Patrz także* typ
  danych, 76
  programu, 176, 191
strumienie
  łańcuchowe, 314
  tokenów, 171
  wejścia i wyjścia, 281
strumień
  fstream, 286
  ifstream, 286
  istream, 284, 288, 298
  istringstream, 314, 451
  ofstream, 286
  ostream, 283
  ostringstream, 314, 339
styl linii, 355
szablon, 550
  funkcji, function template, 553
  klasy, class template, 553
szablony
  koncepcje, 554
  programowanie ogólne, 553
  typy jako parametry, 550
  wartościowe parametry, 558
szacowanie, estimation, 127

T

tablica, array, 156, 496, 509, 608, 632
  asocjacyjna, associative array, 616
  funkcji wirtualnych, 400
  skrótów, hash table, 616, 623, 624
  symboli, 201
technika „dziel i zwyciężaj”, 78
tekst, 336, 374
testowanie, 136
tokeny, tokenize, 152
  deklaracji, 201
  implementowanie, 153
  literały zmiennoprzecinkowe, 152, 158
  nawiasy, 153, 158
  nazwy, 201
  operatory, 153, 158
  strumienie, 171
  używanie, 155
  wczytywanie, 174

tworzenie
GUI, 440
klasy, 255
listy, 485
menu, 452
typ, 52, 54, 65
array, 506
Axis, 421
bool, 54
Button, 444
char, 54, 66
Circle, 371
Closed_polyline, 359
Date, 258
double, 54
Ellipse, 373
enum, 263
exception, 125
Function, 417
Image, 378
initializer_list, 353
int, 54, 65
istream, 283
Line_style, 355
Link, 487, 488
list, 608
Mark, 377
Marked_polyline, 360
Marks, 362
not_null, 506
Open_polyline, 358
ostream, 283
parametryzowany, parametrized type, 553
pmr, 560
Polygon, 364
Rectangle, 366
span, 505, 510
string, 54, 66, 608
T, 550
Text, 374
Token_stream, 173
Vector, 468, 608
Vector_ref, 371
Widget, 444
typy
argumentów, 268
bezpieczeństwo, 67
danych, 55
dedukcja, 70
konwersje, 67
wartościowe, value type, 557
wbudowane, 252
własne, 153
zdefiniowane przez użytkownika, 252
zwrotne, 213, 221
końcowe, 235
przyrostkowe, 235

U

układ
kodu, 191
obiektu, 399
ogólny, 431
uogólnianie
funkcji accumulate(), 656
funkcji copy_if(), 660
funkcji inner_product(), 658
kodu, 583
wektora, 596
uszkodzenie pamięci, memory corruption, 471

W

warstwy oprogramowania, 443
wartość, 54, 65
mieszająca, hash value, 623
zwrotna funkcji, 220
warunek
końcowy, postcondition, 135
wstępny, pre-condition, 132
wczytywanie
liczb, 175
tokenów, 174
wejście/wyjście, input/output, 77, 182, 282, 633
obsługa błędów, 289
wektor, vector, 96, 466
dostęp do elementów, 518
dostosowanie, 600
inicjalizacja, 520
konstruktory kopiujące, 523
kopiowanie, 522

operacje inne, 534
podstawowe operacje, 517
pojemność, 541
powiększanie, 98
problemy, 536
przeglądanie, 97, 598
przenoszenie, 522, 527, 572
przypisywanie, 542
przypisywanie z kopiowaniem, 524
reprezentacja, 539
technika RAII, 570
uogólnianie, 558, 596
wyjątki, 565
wyszukiwanie, 624
z interfejsem, 347
zasoby, 565
zmienianie rozmiaru, 539, 541
wiązanie strukturalne, structured binding, 618
widget, 444
In_box, 446
Out_box, 446
wielkie O, 625
wielkość litery, 311
wielokąt, polygon, 332, 364
wskaźnik, pointer, 469, 500
inteligentny, smart pointer, 481
nullptr, 483
this, 487, 488
wskaźniki
alternatywne narzędzia, 505
do tablic, 496
indeksowanie, 498
inicjalizacja, 500
jako parametry, 501, 502
przesuwanie, 499
sprawdzanie palindromu, 510
wyłuskiwanie, 498
zarządzające zasobami, 481, 573
zerowe, 476
współrzędne ekranu, 328
wstawianie, inlining, 261, 276
wyciek
pamięci, memory leak, 477, 568
zasobów, 481
wyczerpanie pamięci, memory exhaustion, 492
wyjątek typu out_of_range, 123
wyjątki, exceptions, 121, 561, 565
obsługa, 121
ponowne zgłoszenie, 567
przechwytywanie, 567
wykonywanie obliczeń, 77
wykres funkcji, 414, 419
cosinus, 419
osie, 421
pisanie kodu, 433
przedstawianie danych, 428
skalowanie danych, 432
wykładniczej, 423
wyliczenia, 263
zwykłe, 265
wypełnianie kolorem, 335
wyrażenia, 79, 162–164
lambda, 420, 653
obliczanie wartości, 237
podstawowe elementy, 166
stałe, 80
warunkowe, 217
wyszukiwanie binarne, 661
wyświetlenie obiektów, 326
wywołanie zwrotne, callback, 443

Z

zachęta, 52
zagnieżdżanie
bloków, 220
funkcji, 219
zakres
definiowanie, 636
globalny, 216
instrukcji, 217
klasowy, 217
lokalny, 217
modułowy, 216
przestrzeni nazw, 217
wyjściowy, 637
zarządzanie zasobami
gwarancje, 569
problemy, 566

zasada
wszystkich, 531
zera, 531
zasady gramatyki wyrażeń, 160, 161
zasoby, 565, 566
zbiór, set, 628, 635
ziarno, seed, 138
zintegrowane środowisko programistyczne, IDE, 44
zmienianie reprezentacji, 305
zmienne, 52, 54, 65, 197
globalne, 236, 238
iteracyjne, 90, 92
kontrolne, 90
nazwy, 63
znak
&, 198, 224
*, 469
@, 195
>, 183
średnika, 42
znaki
&&, 528
::, 240
\t, 308
->, 235, 475
białe, 55
klasyfikowanie, 311
nieterminalne, 160
nowego wiersza \n, 39, 53
terminalne, 160

Ź

źródła błędów, 111

Chłoń wiedzę od samego twórcy języka C++!

Chcesz naprawdę dobrze opanować C++? Ucz się od najlepszego — Bjarne Stroustrup, twórca tego języka, jak nikt inny potrafi wyjaśnić zarówno podstawy, jak i najbardziej zaawansowane metody programowania. To on zaprojektował i zaimplementował C++, a w tej książce dzieli się swoim bogatym doświadczeniem i ekspercką wiedzą. Teraz dostępną także dla Ciebie!

Programowanie. Teoria i praktyka w C++ to kompletny przewodnik, który krok po kroku odkrywa tajniki jednego z najważniejszych i najpotężniejszych języków programowania. Najnowsze wydanie zostało zaktualizowane i uwzględnia standardy C++20 i C++23, dzięki czemu będziesz się uczyć na przykładach zgodnych z aktualnym kierunkiem rozwoju języka.

Dzięki tej książce dowiesz się, jak pisać kod nie tylko wydajny, ale i elegancki. Poznasz zasady, które z powodzeniem zastosujesz również w innych językach programowania. Autor wprowadzi Cię w świat kluczowych paradygmatów programowania — od proceduralnego, przez obiektowe, aż po generyczne — i pokaże, jak tworzyć bezpieczne, praktyczne i łatwe w utrzymaniu programy. Oprócz technik programowania wysokopoziomowego opanujesz techniki niższego poziomu, niezbędne do efektywnego korzystania z możliwości sprzętu.

Znajdziesz tu zarówno solidne podstawy, jak i bardziej zaawansowane zagadnienia, takie jak:
- opis pojęć i technik programistycznych
- instrukcje sterujące, obsługa błędów, funkcje i system typów
- obsługa danych liczbowych i tekstu
- praca z graficznym interfejsem użytkownika
- kontenery i algorytmy w bibliotece STL
- parametryzacja klas i funkcji

Niezależnie od tego, czy dopiero zaczynasz przygodę z C++, czy chcesz rozwinąć swoje umiejętności — tu znajdziesz wszystko, czego potrzebujesz, aby pisać doskonały kod.

DR BJARNE STROUSTRUP zaprojektował i zaimplementował język C++. Jest autorem książek, licznych publikacji popularnonaukowych i akademickich. Piastuje stanowisko profesora informatyki na Columbia University w Nowym Jorku. Jest członkiem amerykańskiej National Academy of Engineering, stowarzyszeń IEEE, ACM i CHM, a także laureatem prestiżowych wyróżnień.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-2995-1
9 788328 929951
Cena: 149,00 zł
Pearson
Addison-Wesley