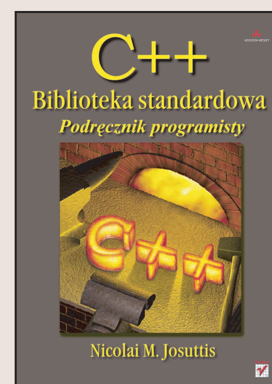
IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG
TWÓJ KOSZYK
DODAJ DO KOSZYKA
CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK
CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

C++. Biblioteka standardowa. Podręcznik programisty



Autor: Nicolai M. Josuttis
Tłumaczenie: Przemysław Steć (rozdz. 1 – 9), Rafał Szpoton (rozdz. 10 – 15)
ISBN: 83-7361-144-4
Tytuł oryginału: The C++ Standard Library: A Tutorial and Referencee
Format: B5, stron: 726

Biblioteka standardowa C++ to zestaw klas oraz interfejsów znacznie rozszerzających język C++. Nie jest ona jednak łatwa do przyswojenia. W celu pełnego wykorzystania udostępnianych przez nią komponentów oraz skorzystania z jej możliwości, konieczne jest odwołanie się do materiałów zawierających nieco więcej informacji niż tylko listę klas oraz zawartych w nich funkcji.

Książka „C++. Biblioteka standardowa. Podręcznik programisty" dostarcza wyczerpującej dokumentacji każdego z komponentów biblioteki, jak również przystępnych wyjaśnień złożonych zagadnień; prezentuje praktyczne szczegóły programowania, niezbędne do skutecznego zastosowania omawianej biblioteki w praktyce. Znajdziesz w niej również liczne przykłady działającego kodu źródłowego.

Książka „C++. Biblioteka standardowa. Podręcznik programisty" opisuje aktualną wersję biblioteki standardowej C++, w tym jej najnowsze elementy dołączone do pełnego standardu języka ANSI/ISO C++. Opis skoncentrowany jest na standardowej bibliotece wzorców STL (ang. Standard Template Library), kontenerach, iteratorach, obiektach funkcyjnych oraz algorytmach STL. W książce tej znajdziesz również szczegółowy opis kontenerów specjalnych, łańcuchów znakowych, klas numerycznych, zagadnienia lokalizacji programów oraz omówienie biblioteki IOStream.
Każdy z komponentów został dokładnie przedstawiony wraz z opisem jego przeznaczenia oraz założeń projektowych, przykładami, czyhającymi pułapkami, jak również definicją udostępnianych przez niego klas oraz funkcji.

Omówione w książce zagadnienia to między innymi:

- Krótkie wprowadzenie do C++ i biblioteki standardowej
- Standardowa biblioteka wzorców
- Kontenery STL
- Obiekty funkcyjne STL
- Algorytmy STL
- Kontenery specjalne: stosy, kolejki, klasa bitset
- Łańcuchy
- Kontenery numeryczne
- Operacje wejścia-wyjścia z wykorzystaniem klas strumieniowych
- Funkcje służące umiędzynarodowieniu aplikacji
- Alokatory

„C++. Biblioteka standardowa. Podręcznik programisty" stanowi wyczerpującą, szczegółową, przystępnie napisaną oraz praktyczną książkę. Tworzy ona materiał referencyjny C++, do którego będziesz stale powracać.

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

Spis treści

Podziękowania 13

Przedmowa 15

1
O książce 17

1.1. Dlaczego powstała ta książka? 17
1.2. Co należy wiedzieć przed przystąpieniem do lektury tej książki? 18
1.3. Styl i struktura książki 18
1.4. Jak czytać tę książkę? 21
1.5. Stan obecny 21
1.6. Przykładowy kod i dodatkowe informacje 22

2
Wprowadzenie do języka C++ i biblioteki standardowej 23

2.1. Historia 23
2.2. Nowe możliwości języka 25
2.2.1. Wzorce 25
2.2.2. Jawna inicjalizacja typów podstawowych 30
2.2.3. Obsługa wyjątków 30
2.2.4. Przestrzenie nazw 32
2.2.5. Typ bool 33
2.2.6. Słowo kluczowe explicit 34
2.2.7. Nowe operatory konwersji typu 35
2.2.8. Inicjalizacja stałych składowych statycznych 36
2.2.9. Definicja funkcji main() 36
2.3. Złożoność algorytmów a notacja O 37

3

Pojęcia ogólne 39

3.1. Przestrzeń nazw std 39
3.2. Pliki nagłówkowe 40
3.3. Obsługa błędów i wyjątków 42
3.3.1. Standardowe klasy wyjątków 42
3.3.2. Składowe klas wyjątków 45
3.3.3. Zgłaszanie wyjątków standardowych 46
3.3.4. Tworzenie klas pochodnych standardowych klas wyjątków 46
3.4. Alokatory 48

4

Narzędzia 49

4.1. Pary 49
4.1.1. Wygodna funkcja make_pair() 51
4.1.2. Przykłady użycia par 53
4.2. Klasa auto_ptr 53
4.2.1. Motywacja klasy auto_ptr 53
4.2.2. Przenoszenie własności w przypadku klasy auto_ptr 55
4.2.3. Wskaźniki auto_ptr jako składowe 59
4.2.4. Niewłaściwe użycie wskaźników auto_ptr 61
4.2.5. Przykłady zastosowania typu auto_ptr 62
4.2.6. Klasa auto_ptr w szczegółach 64
4.3. Ograniczenia liczbowe 71
4.4. Funkcje pomocnicze 77
4.4.1. Obliczanie wartości minimalnej oraz maksymalnej 77
4.4.2. Zamiana dwóch wartości 78
4.5. Dodatkowe operatory porównania 79
4.6. Pliki nagłówkowe <cstddef> oraz <cstdlib> 81
4.6.1. Definicje w pliku <cstddef> 81
4.6.2. Definicje w pliku <cstdlib> 82

5

Standardowa biblioteka wzorców (STL) 83

5.1. Składniki biblioteki STL 83
5.2. Kontenery 85
5.2.1. Kontenery sekwencyjne 86
5.2.2. Kontenery asocjacyjne 91
5.2.3. Adaptatory kontenerów 92
5.3. Iteratory 93
5.3.1. Przykłady użycia kontenerów asocjacyjnych 96
5.3.2. Kategorie iteratorów 102
5.4. Algorytmy 103
5.4.1. Zakresy 105
5.4.2. Obsługa wielu zakresów 110

5.5. Adaptatory iteratorów ... 112
5.5.1. Iteratory wstawiające ... 112
5.5.2. Iteratory strumieniowe ... 114
5.5.3. Iteratory odwrotne ... 116
5.6. Algorytmy modyfikujące ... 118
5.6.1. Usuwanie elementów ... 118
5.6.2. Algorytmy modyfikujące a kontenery asocjacyjne ... 121
5.6.3. Algorytmy a funkcje składowe ... 123
5.7. Funkcje ogólne definiowane przez użytkownika ... 124
5.8. Funkcje jako argumenty algorytmów ... 125
5.8.1. Przykłady użycia funkcji jako argumentów algorytmów ... 125
5.8.2. Predykaty ... 126
5.9. Obiekty funkcyjne ... 129
5.9.1. Czym są obiekty funkcyjne? ... 129
5.9.2. Predefiniowane obiekty funkcyjne ... 135
5.10. Elementy kontenerów ... 138
5.10.1. Wymagania wobec elementów kontenerów ... 138
5.10.2. Semantyka wartości a semantyka referencji ... 139
5.11. Obsługa błędów i wyjątków wewnątrz biblioteki STL ... 140
5.11.1. Obsługa błędów ... 141
5.11.2. Obsługa wyjątków ... 143
5.12. Rozbudowa biblioteki STL ... 145

6

Kontenery STL ... 147

6.1. Wspólne cechy i operacje kontenerów ... 148
6.1.1. Wspólne cechy kontenerów ... 148
6.1.2. Wspólne operacje kontenerów ... 148
6.2. Wektory ... 151
6.2.1. Możliwości wektorów ... 152
6.2.2. Operacje na wektorach ... 154
6.2.3. Używanie wektorów jako zwykłych tablic ... 158
6.2.4. Obsługa wyjątków ... 159
6.2.5. Przykłady użycia wektorów ... 160
6.2.6. Klasa vector<bool> ... 161
6.3. Kolejki o dwóch końcach ... 163
6.3.1. Możliwości kolejek deque ... 164
6.3.2. Operacje na kolejkach deque ... 165
6.3.3. Obsługa wyjątków ... 166
6.3.4. Przykłady użycia kolejek deque ... 167
6.4. Listy ... 168
6.4.1. Możliwości list ... 169
6.4.2. Operacje na listach ... 170
6.4.3. Obsługa wyjątków ... 174
6.4.4. Przykłady użycia list ... 175
6.5. Zbiory i wielozbiory ... 176
6.5.1. Możliwości zbiorów i wielozbiorów ... 178
6.5.2. Operacje na zbiorach i wielozbiorach ... 179
6.5.3. Obsługa wyjątków ... 187

6.5.4. Przykłady użycia zbiorów i wielozbiorów ... 187
6.5.5. Przykład określania kryterium sortowania podczas wykonywania ... 191
6.6. Mapy oraz multimapy ... 193
6.6.1. Możliwości map oraz multimap ... 194
6.6.2. Operacje na mapach oraz multimapach ... 195
6.6.3. Zastosowanie map jako tablic asocjacyjnych ... 204
6.6.4. Obsługa wyjątków ... 206
6.6.5. Przykłady użycia map i multimap ... 206
6.6.6. Przykład z mapami, łańcuchami oraz definiowaniem kryterium sortowania podczas wykonywania ... 210
6.7. Inne kontenery STL ... 212
6.7.1. Łańcuchy jako kontenery STL ... 213
6.7.2. Zwykłe tablice jako kontenery STL ... 214
6.7.3. Tablice mieszające ... 216
6.8. Implementacja semantyki referencji ... 217
6.9. Kiedy stosować poszczególne kontenery ... 220
6.10. Typy kontenerowe i ich składowe w szczegółach ... 223
6.10.1. Definicje typów ... 224
6.10.2. Operacje tworzenia, kopiowania i niszczenia ... 225
6.10.3. Operacje niemodyfikujące ... 227
6.10.4. Operacje przypisania ... 230
6.10.5. Bezpośredni dostęp do elementów ... 231
6.10.6. Operacje generujące iteratory ... 233
6.10.7. Wstawianie i usuwanie elementów ... 234
6.10.8. Specjalne funkcje składowe list ... 239
6.10.9. Obsługa alokatorów ... 241
6.10.10. Omówienie obsługi wyjątków w kontenerach STL ... 242

7

Iteratory STL ... 245

7.1. Pliki nagłówkowe iteratorów ... 245
7.2. Kategorie iteratorów ... 245
7.2.1. Iteratory wejściowe ... 246
7.2.2. Iteratory wyjściowe ... 247
7.2.3. Iteratory postępujące ... 248
7.2.4. Iteratory dwukierunkowe ... 249
7.2.5. Iteratory dostępu swobodnego ... 249
7.2.6. Problem z inkrementacją i dekrementacją iteratorów wektorów ... 252
7.3. Pomocnicze funkcje iteratorów ... 253
7.3.1. Przesuwanie iteratorów za pomocą funkcji advance() ... 253
7.3.2. Obliczanie odległości pomiędzy iteratorami za pomocą funkcji distance() ... 254
7.3.3. Zamiana wartości iteratorów za pomocą funkcji iter_swap() ... 256
7.4. Adaptatory iteratorów ... 257
7.4.1. Iteratory odwrotne ... 257
7.4.2. Iteratory wstawiające ... 262
7.4.3. Iteratory strumieniowe ... 268
7.5. Cechy iteratorów ... 273
7.5.1. Definiowanie funkcji ogólnych dla iteratorów ... 275
7.5.2. Iteratory definiowane przez użytkownika ... 277

8

Obiekty funkcyjne STL 281

8.1. Pojęcie obiektów funkcyjnych 281
8.1.1. Obiekty funkcyjne jako kryteria sortowania 282
8.1.2. Obiekty funkcyjne ze stanem wewnętrznym 283
8.1.3. Wartość zwracana algorytmu for_each() 287
8.1.4. Predykaty a obiekty funkcyjne 288
8.2. Predefiniowane obiekty funkcyjne 291
8.2.1. Adaptatory funkcji 291
8.2.2. Adaptatory funkcji składowych 293
8.2.3. Adaptatory zwykłych funkcji 295
8.2.4. Obiekty funkcyjne definiowane przez użytkownika dostosowane do adaptatorów funkcji 296
8.3. Dodatkowe złożeniowe obiekty funkcyjne 298
8.3.1. Jednoargumentowe złożeniowe adaptatory obiektów funkcyjnych 299
8.3.2. Dwuargumentowe złożeniowe adaptatory obiektów funkcyjnych 302

9

Algorytmy STL 305

9.1. Pliki nagłówkowe algorytmów 305
9.2. Przegląd algorytmów 306
9.2.1. Krótkie wprowadzenie 306
9.2.2. Klasyfikacja algorytmów 307
9.3. Funkcje pomocnicze 316
9.4. Algorytm for_each() 318
9.5. Algorytmy niemodyfikujące 320
9.5.1. Zliczanie elementów 321
9.5.2. Wartość minimalna i maksymalna 322
9.5.3. Wyszukiwanie elementów 324
9.5.4. Porównywanie zakresów 335
9.6. Algorytmy modyfikujące 340
9.6.1. Kopiowanie elementów 341
9.6.2. Przekształcenia i kombinacje elementów 344
9.6.3. Wymienianie elementów 347
9.6.4. Przypisywanie nowych wartości 348
9.6.5. Zastępowanie elementów 350
9.7. Algorytmy usuwające 353
9.7.1. Usuwanie określonych wartości 353
9.7.2. Usuwanie powtórzeń 356
9.8. Algorytmy mutujące 360
9.8.1. Odwracanie kolejności elementów 360
9.8.2. Przesunięcia cykliczne elementów 361
9.8.3. Permutacje elementów 363
9.8.4. Tasowanie elementów 365
9.8.5. Przenoszenie elementów na początek 367
9.9. Algorytmy sortujące 368
9.9.1. Sortowanie wszystkich elementów 369
9.9.2. Sortowanie częściowe 371

9.9.3. Sortowanie według n-tego elementu ... 374
9.9.4. Algorytmy stogowe ... 375
9.10. Algorytmy przeznaczone dla zakresów posortowanych ... 378
9.10.1. Wyszukiwanie elementów ... 379
9.10.2. Scalanie elementów ... 385
9.11. Algorytmy numeryczne ... 393
9.11.1. Obliczanie wartości ... 393
9.11.2. Konwersje wartości względnych i bezwzględnych ... 397

10

Kontenery specjalne ... 401

10.1. Stosy ... 401
10.1.1. Interfejs ... 403
10.1.2. Przykład użycia stosów ... 403
10.1.3. Klasa stack<> w szczegółach ... 404
10.1.4. Klasa stosu definiowanego przez użytkownika ... 406
10.2. Kolejki ... 408
10.2.1. Interfejs ... 410
10.2.2. Przykład użycia kolejek ... 410
10.2.3. Klasa queue<> w szczegółach ... 411
10.2.4. Klasa kolejki definiowanej przez użytkownika ... 413
10.3. Kolejki priorytetowe ... 416
10.3.1. Interfejs ... 417
10.3.2. Przykład użycia kolejek priorytetowych ... 418
10.3.3. Klasa priority_queue<> w szczegółach ... 418
10.4. Kontener bitset ... 421
10.4.1. Przykłady użycia kontenerów bitset ... 422
10.4.2. Szczegółowy opis klasy bitset ... 424

11

Łańcuchy ... 431

11.1. Motywacja ... 431
11.1.1. Przykład pierwszy: Pobieranie tymczasowej nazwy pliku ... 432
11.1.2. Przykład drugi: Pobieranie słów i wypisywanie ich w odwrotnej kolejności ... 437
11.2. Opis klas reprezentujących łańcuchy znakowe ... 440
11.2.1. Typy łańcuchów znakowych ... 440
11.2.2. Przegląd funkcji składowych ... 441
11.2.3. Konstruktory oraz destruktory ... 443
11.2.4. Łańcuchy znakowe zwykłe oraz języka C ... 444
11.2.5. Rozmiar oraz pojemność ... 446
11.2.6. Dostęp do elementów ... 448
11.2.7. Porównania ... 449
11.2.8. Modyfikatory ... 450
11.2.9. Konkatenacja łańcuchów znakowych oraz ich fragmentów ... 453
11.2.10. Operatory wejścia-wyjścia ... 454
11.2.11. Poszukiwanie oraz odnajdywanie łańcuchów znakowych ... 455
11.2.12. Wartość npos ... 457
11.2.13. Obsługa iteratorów w przypadku łańcuchów znakowych ... 458

11.2.14. Obsługa standardów narodowych ... 464
11.2.15. Wydajność ... 466
11.2.16. Łańcuchy znakowe a wektory ... 466
11.3. Klasa string w szczegółach ... 467
11.3.1. Definicje typu oraz wartości statyczne ... 467
11.3.2. Funkcje składowe służące do tworzenia, kopiowania oraz usuwania łańcuchów znakowych ... 469
11.3.3. Funkcje dotyczące rozmiaru oraz pojemności ... 470
11.3.4. Porównania ... 471
11.3.5. Dostęp do znaków ... 473
11.3.6. Tworzenie łańcuchów znakowych języka C oraz tablic znaków ... 474
11.3.7. Funkcje do modyfikacji zawartości łańcuchów znakowych ... 475
11.3.8. Poszukiwanie oraz odnajdywanie ... 482
11.3.9. Łączenie łańcuchów znakowych ... 485
11.3.10. Funkcje wejścia-wyjścia ... 486
11.3.11. Funkcje tworzące iteratory ... 487
11.3.12. Obsługa alokatorów ... 488

12

Kontenery numeryczne ... 491

12.1. Liczby zespolone ... 491
12.1.1. Przykład wykorzystania klasy reprezentującej liczby zespolone ... 492
12.1.2. Funkcje operujące na liczbach zespolonych ... 494
12.1.3. Klasa complex<> w szczegółach ... 501
12.2. Klasa valarray ... 506
12.2.1. Poznawanie klas valarray ... 507
12.2.2. Podzbiory wartości umieszczonych w tablicy valarray ... 512
12.2.3. Klasa valarray w szczegółach ... 525
12.2.4. Klasy podzbiorów tablic typu valarray w szczegółach ... 530
12.3. Globalne funkcje numeryczne ... 535

13

Obsługa wejścia-wyjścia z wykorzystaniem klas strumieniowych ... 537

13.1. Podstawy strumieni wejścia-wyjścia ... 538
13.1.1. Obiekty strumieni ... 538
13.1.2. Klasy strumieni ... 539
13.1.3. Globalne obiekty strumieni ... 539
13.1.4. Operatory strumieniowe ... 540
13.1.5. Manipulatory ... 540
13.1.6. Prosty przykład ... 541
13.2. Podstawowe obiekty oraz klasy strumieniowe ... 542
13.2.1. Klasy oraz hierarchia klas ... 542
13.2.2. Globalne obiekty strumieni ... 545
13.2.3. Pliki nagłówkowe ... 546
13.3. Standardowe operatory strumieniowe << oraz >> ... 547
13.3.1. Operator wyjściowy << ... 547
13.3.2. Operator wejściowy >> ... 549
13.3.3. Operacje wejścia-wyjścia dla specjalnych typów ... 549

13.4. Stany strumieni ... 552
13.4.1. Stałe służące do określania stanów strumieni ... 552
13.4.2. Funkcje składowe operujące na stanie strumieni ... 553
13.4.3. Warunki wykorzystujące stan strumienia oraz wartości logiczne ... 555
13.4.4. Stan strumienia i wyjątki ... 557
13.5. Standardowe funkcje wejścia-wyjścia ... 561
13.5.1. Funkcje składowe służące do pobierania danych ... 562
13.5.2. Funkcje składowe służące do wysyłania danych ... 565
13.5.3. Przykład użycia ... 566
13.6. Manipulatory ... 567
13.6.1. Sposób działania manipulatorów ... 568
13.6.2. Manipulatory definiowane przez użytkownika ... 569
13.7. Formatowanie ... 570
13.7.1. Znaczniki formatu ... 570
13.7.2. Format wartości logicznych ... 572
13.7.3. Szerokość pola znak wypełnienia oraz wyrównanie ... 573
13.7.4. Znak wartości dodatnich oraz duże litery ... 576
13.7.5. Podstawa numeryczna ... 576
13.7.6. Notacja zapisu liczb zmiennoprzecinkowych ... 579
13.7.7. Ogólne definicje formatujące ... 581
13.8. Umiędzynarodawianie ... 582
13.9. Dostęp do plików ... 583
13.9.1. Znaczniki pliku ... 586
13.9.2. Dostęp swobodny ... 589
13.9.3. Deskryptory plików ... 592
13.10. Łączenie strumieni wejściowych oraz wyjściowych ... 592
13.10.1 Luźne powiązanie przy użyciu tie() ... 593
13.10.2. Ścisłe powiązanie przy użyciu buforów strumieni ... 594
13.10.3. Przekierowywanie strumieni standardowych ... 596
13.10.4. Strumienie służące do odczytu oraz zapisu ... 597
13.11. Klasy strumieni dla łańcuchów znakowych ... 599
13.11.1. Klasy strumieni dla łańcuchów znakowych ... 600
13.11.2. Klasy strumieni dla wartości typu char* ... 603
13.12. Operatory wejścia-wyjścia dla typów zdefiniowanych przez użytkownika ... 605
13.12.1. Implementacja operatorów wyjściowych ... 605
13.12.2. Implementacja operatorów wejściowych ... 607
13.12.3. Operacje wejścia-wyjścia przy użyciu funkcji pomocniczych ... 609
13.12.4. Operatory definiowane przez użytkownika używające funkcji nieformatujących ... 610
13.12.5. Znaczniki formatu definiowane przez użytkownika ... 612
13.12.6. Konwencje operatorów wejścia-wyjscia definiowanych przez użytkownika ... 615
13.13. Klasy bufora strumienia ... 616
13.13.1. Spojrzenie użytkownika na bufory strumieni ... 616
13.13.2. Iteratory wykorzystywane z buforem strumienia ... 618
13.13.3. Bufory strumienia definiowane przez użytkownika ... 621
13.14. Kwestie wydajności ... 632
13.14.1. Synchronizacja ze standardowymi strumieniami używanymi w języku C ... 633
13.14.2. Buforowanie w buforach strumieni ... 633
13.14.3. Bezpośrednie wykorzystanie buforów strumieni ... 634

14

Umiędzynarodowienie

Umiędzynarodowienie ... 637

14.1. Różne standardy kodowania znaków ... 638
14.1.1. Reprezentacja za pomocą zestawu znaków szerokiego zakresu lub reprezentacja wielobajtowa ... 639
14.1.2. Cechy znaków ... 640
14.1.3. Umiędzynarodawianie specjalnych znaków ... 643
14.2. Pojęcie obiektów ustawień lokalnych ... 644
14.2.1. Wykorzystywanie ustawień lokalnych ... 646
14.2.2. Aspekty ustawień lokalnych ... 650
14.3. Klasa locale w szczegółach ... 653
14.4. Klasa facet w szczegółach ... 656
14.4.1. Formatowanie wartości numerycznych ... 657
14.4.2. Formatowanie czasu oraz daty ... 661
14.4.3. Formatowanie wartości walutowych ... 664
14.4.4. Klasyfikacja oraz konwersja znaków ... 669
14.4.5. Sortowanie łańcuchów znakowych ... 677
14.4.6. Lokalizacja komunikatów ... 679

15

Alokatory ... 681

15.1. Wykorzystywanie alokatorów przez programistów aplikacji ... 681
15.2. Wykorzystywanie alokatorów przez programistów bibliotek ... 682
15.3. Alokator domyślny ... 685
15.4. Alokator zdefiniowany przez użytkownika ... 687
15.5. Alokatory w szczegółach ... 689
15.5.1. Definicje typu ... 689
15.5.2. Funkcje składowe ... 691
15.6. Funkcje stosowane w przypadku niezainicjalizowanej pamięci ... 692

A

Bibliografia ... 695

Skorowidz ... 697

10
Kontenery specjalne

Standardowa biblioteka C++ zawiera nie tylko kontenery szkieletu STL, lecz również kontenery służące do specjalnych celów oraz dostarczające prostych interfejsów prawie niewymagających opisu. Kontenery te mogą zostać podzielone na:

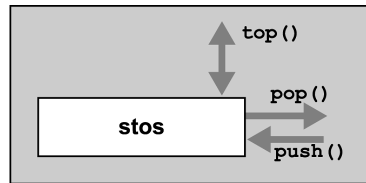
- Tak zwane *adaptatory kontenerów*.
  Tego rodzaju kontenery przystosowują standardowe kontenery STL do specjalnych celów. Wyróżniamy trzy rodzaje adaptatorów standardowych kontenerów:
  1. Stosy.
  2. Kolejki.
  3. Kolejki priorytetowe.

  Te ostatnie stanowią pewien rodzaj kolejki z elementami posortowanymi w sposób automatyczny według ustalonych kryteriów sortowania. Dlatego też „kolejnym" elementem w kolejce priorytetowej jest ten o „najwyższej" wartości.
- Specjalny kontener nazwany *bitset*.
  Kontener `bitset` jest polem bitowym, zawierającym dowolną, lecz ustaloną z góry liczbę bitów. Można traktować go jako kontener przechowujący wartości bitowe lub logiczne. Zwróć uwagę, że standardowa biblioteka C++ zawiera również specjalny kontener o zmiennym rozmiarze, przeznaczony dla wartości logicznych: `vector<bool>`. Został on opisany w podrozdziale 6.2.6., na stronie 161.

10.1. Stosy

Stos (zwany również kolejką LIFO) zaimplementowany został w klasie o nazwie `stack<>`. Funkcja składowa `push()` służy do wkładania na stos dowolnej liczby elementów (patrz rysunek 10.1). Pobranie elementów ze stosu możliwe jest natomiast przy użyciu funkcji składowej o nazwie `pop()`. Pobieranie następuje w odwrotnej kolejności do ich umieszczania[1].

[1] „last in, first out", co w dosłownym tłumaczeniu oznacza „ostatni wszedł, pierwszy wyszedł" — *przyp. tłum.*



RYSUNEK 10.1.
Interfejs stosu

W celu wykorzystania stosu konieczne jest dołączenie pliku nagłówkowego `<stack>`[2]:

```
#include <stack>
```

Deklaracja klasy `stack` w pliku nagłówkowym `<stack>` wygląda następująco:

```
namespace std {
  template <class T,
            class Container = deque<T> >
  class stack;
}
```

Pierwszy z parametrów wzorca określa rodzaj elementów. Drugi natomiast jest opcjonalny i definiuje kontener używany wewnętrznie w celu skolejkowania elementów umieszczonych we właściwym kontenerze stosu. Domyślnie przyjmowany jest kontener o nazwie `deque`[3]. Wybrany został właśnie ten kontener, ponieważ w przeciwieństwie do wektorów zwalnia on używaną przez siebie pamięć po usunięciu umieszczonych w nim elementów i nie musi przekopiowywać ich wszystkich w przypadku ponownego jej przydzielania (dokładne omówienie przypadków stosowania różnych kontenerów zostało umieszczone w podrozdziale 6.9., na stronie 220).

Na przykład poniższy wiersz zawiera deklarację, określającą stos liczb całkowitych[4]:

```
std::stack<int> st; //stos przechowujacy liczby całkowite
```

Implementacja stosu polega na prostym odwzorowaniu wykonywanych operacji na odpowiednie wywołania funkcji składowych używanego wewnętrznie kontenera (patrz rysunek 10.2). Możliwe jest użycie dowolnej klasy kontenera udostępniającej funkcje składowe o nazwie `back()`, `push_back()` oraz `pop_back()`. I tak na przykład w charakterze kontenera elementów możliwe byłoby wykorzystanie listy lub wektora:

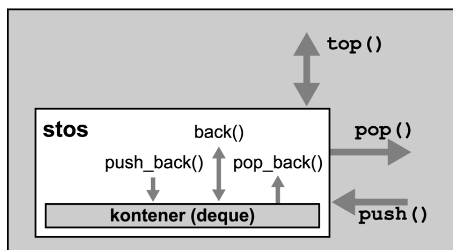
```
std::stack<int,std::vector<int> > st; //stos przechowujacy liczby calkowite wykorzystujacy wektor
```

[2] W oryginalnej bibliotece *STL* plik nagłówkowy zawierający deklarację stosu nosił nazwę: `<stack.h>`.

[3] Kolejka o dwóch końcach — *przyp. tłum.*

[4] W poprzednich wersjach biblioteki *STL* istniała konieczność przekazywania kontenera w charakterze obowiązkowego argumentu wzorca. Dlatego też stos liczb całkowitych musiał być wtedy deklarowany w następujący sposób:

```
stack<deque<int> > st;
```



RYSUNEK 10.2.
Wewnętrzny interfejs stosu

10.1.1. Interfejs

W przypadku stosów interfejs tworzą funkcje składowe o nazwach `push()`, `top()` oraz `pop()`:

- Funkcja składowa **`push()`** umieszcza element na stosie.
- Funkcja składowa **`top()`** zwraca kolejny element stosu.
- Funkcja składowa **`pop()`** usuwa element ze stosu.

Zwróć uwagę, że funkcja składowa `pop()` usuwa kolejny element, lecz go nie zwraca, podczas gdy `top()` zwraca kolejny element bez jego usuwania. Dlatego też w celu przetworzenia oraz usunięcia kolejnego elementu na stosie konieczne jest wywołanie obu tych funkcji składowych. Opisany interfejs jest nieco niewygodny, lecz działa znacznie lepiej w przypadku, gdy chcesz jedynie usunąć kolejny element bez jego analizy i przetwarzania. Zauważ, że działanie funkcji składowych `top()` oraz `pop()` jest niezdefiniowane w przypadku, gdy stos nie zawiera żadnych elementów. W celu umożliwienia sprawdzenia, czy na stosie umieszczone są jakiekolwiek elementy, dodane zostały funkcje składowe `size()` oraz `empty()`.

Jeśli standardowy interfejs kontenera `stack<>` nie przypadł ci do gustu, oczywiście możesz w łatwy sposób napisać swój własny i bardziej wygodny w użyciu. Odpowiedni przykład zostanie umieszczony w podrozdziale 10.1.4., na stronie 406.

10.1.2. Przykład użycia stosów

Poniższy przykład demonstruje wykorzystanie klasy `stack<>`:

```
//cont/stack1.cpp
#include <iostream>
#include <stack>
using namespace std;

int main()
{
    stack<int> st;

    // umiesc na stosie trzy elementy
    st.push(1);
    st.push(2);
    st.push(3);
```

```
        // pobierz ze stosu dwa elementy i je wypisz
        cout << st.top() << ' ';
        st.pop();
        cout << st.top() << ' ';
        st.pop();

        // zmien kolejny element
        st.top() = 77;

        // umiesc dwa dodatkowe elementy
        st.push(4);
        st.push(5);

        // pobierz jeden element bez jego przetwarzania
        st.pop();

        // pobierz i wypisz pozostale elementy
        while (!st.empty()) {
            cout << st.top() << ' ';
            st.pop();
        }
        cout << endl;
    }
```

Dane wyjściowe programu wyglądają następująco:

```
3 2 4 77
```

10.1.3. Klasa stack<> w szczegółach

Interfejs klasy `stack<>` jest tak niewielki, iż można go w prosty sposób zrozumieć, analizując jego typową implementację:

```
namespace std {
   template <class T, class Container = deque<T> >
   class stack {
     public:
       typedef typename Container::value_type value_type;
       typedef typename Container::size_type  size_type;
       typedef          Container             container_type;
     protected:
       Container c;   //kontener
     public:
       explicit stack(const Container& = Container());

       bool         empty() const              { return c.empty(); }
       size_type    size()  const              { return c.size(); }
       void         push(const value_type& x)  { c.push_back(x); }
       void         pop()                      { c.pop_back(); }
       value_type&  top()                      { return c.back(); }
       const value_type& top() const           { return c.back(); }
};

template <class T, class Container>
  bool operator==(const stack<T, Container>&,
                  const stack<T, Container>&);
template <class T, class Container>
```

```
  bool operator< (const stack<T, Container>&,
                  const stack<T, Container>&);
...// (inne operatory porownania)
}
```

Poniższa część tego podrozdziału zawiera szczegółowy opis pól oraz operacji.

Definicje typu

stack::**value_type**
- Rodzaj elementów umieszczonych w kontenerze.
- Składowa równoważna jest składowej *container*::value_type.

stack::**size_type**
- Typ całkowity bez znaku, określający rozmiar umieszczonych elementów.
- Składowa równoważna jest składowej *container*::size_type.

stack::**container_type**
- Rodzaj kontenera.

Funkcje składowe

stack::**stack** ()
- Konstruktor domyślny.
- Tworzy pusty stos.

explicit *stack*::**stack** (const Container& *cont*)
- Tworzy stos inicjalizowany elementami umieszczonymi w obiekcie *cont*.
- Kopiowane są wszystkie elementy umieszczone w kontenerze *cont*.

size_type *stack*::**size** () const
- Zwraca bieżącą liczbę elementów.
- W celu sprawdzenia, czy stos jest pusty, używaj funkcji składowej o nazwie empty(), ponieważ jej działanie może być szybsze.

bool *stack*::**empty** () const
- Zwraca wartość logiczną, określającą, czy stos jest pusty (nie zawiera elementów).
- Funkcja składowa równoważna konstrukcji postaci *stack*::**size()==0**, lecz może działać od niej szybciej.

void *stack*::**push** (const *value_type& elem*)
- Wstawia jako pierwszy element stosu kopię elementu, określonego przez wartość *elem*.

value_type& stack::**top** ()
const *value_type& stack*::**top** () const
- Obie postaci funkcji składowej zwracają kolejny element stosu. Będzie nim element wstawiony jako ostatni (po wszystkich innych elementach stosu).
- Przed wywołaniem funkcji składowej należy upewnić się, że stos zawiera jakiekolwiek elementy (size()>0). W innym przypadku jej działanie jest nieokreślone.

- Funkcja składowa w pierwszej postaci przeznaczona jest dla stosów, które nie są określone jako statyczne (`nonconstant`) i zwraca referencję. Dlatego też możliwe jest zmodyfikowanie kolejnego elementu jeszcze w chwili, gdy jest on umieszczony na stosie. Do ciebie należy podjęcie decyzji, czy jest to dobry styl programowania.

`void` *`stack`*`::`**`pop`**` ()`
- Usuwa kolejny element stosu. Będzie nim element wstawiony jako ostatni (po wszystkich innych elementach stosu).
- Funkcja nie zwraca wartości. W celu przetworzenia kolejnego elementu musisz wcześniej wywołać funkcję składową `top()`.
- Przed wywołaniem funkcji składowej należy upewnić się, że stos zawiera jakiekolwiek elementy (`size()>0`). W innym przypadku jej działanie jest nieokreślone.

`bool` ***`comparison`*** `(const` *`stack& stack1`*`, const` *`stack& stack2`*`)`
- Zwraca wynik porównania dwóch stosów tego samego rodzaju.
- ***`comparison`*** może być określony w jeden z poniższych sposobów:

```
operator ==
operator !=
operator <
operator >
operator <=
operator >=
```

- Dwa stosy są jednakowe w przypadku, gdy zawierają taką samą liczbę identycznych elementów umieszczonych w tej samej kolejności (porównanie każdych dwóch odpowiadających sobie elementów musi dawać w rezultacie wartość `true`).
- W celu sprawdzenia, czy jeden stos jest mniejszy od drugiego, oba porównywane są leksykograficznie. Więcej informacji zostało umieszczonych na stronie 338 wraz z opisem algorytmu o nazwie `lexicographical_compare()`.

10.1.4. Klasa stosu definiowanego przez użytkownika

Standardowa klasa `stack<>` preferuje szybkość działania nad wygodę oraz bezpieczeństwo użytkowania. Nie jest to tym, co zazwyczaj ja uważam za najlepsze podejście. Dlatego też napisałem własną klasę stosu. Ma ona dwie zalety:

1. `pop()` zwraca kolejny element.
2. `pop()` oraz `top()` zwraca wyjątek w przypadku, gdy stos jest pusty.

Dodatkowo pominąłem wszystkie funkcje składowe, które nie są niezbędne dla zwykłego użytkownika stosu, jak chociażby operacje porównania. Moja klasa stosu zdefiniowana została w następujący sposób:

```
//cont/Stack.hpp
/* ************************************************************
 * Stack.hpp
 *  - bezpieczniejsza oraz bardziej wygodna klasa stosu
 * ************************************************************/
#ifndef STACK_HPP
#define STACK_HPP
```

```
#include <deque>
#include <exception>

template <class T>
class Stack {
  protected:
    std::deque<T> c;         // kontener zawierajacy elementy

  public:
    /* klasa wyjatku dla funkcji składowych pop() oraz top() wywolanych w przypadku pustego stosu
     */
    class ReadEmptyStack : public std::exception {
      public:
        virtual const char* what() const throw() {
            return "proba odczytania pustego elementu";
        }
    };

    // liczba elementow
    typename std::deque<T>::size_type size() const {
        return c.size();
    }

    // czy stos jest pusty?
    bool empty() const {
        return c.empty();
    }

    // umiesc element na stosie
    void push (const T& elem) {
        c.push_back(elem);
    }

    // zdejmij element ze stosu i zwroc jego wartosc
    T pop () {
        if (c.empty()) {
            throw ReadEmptyStack();
        }
        T elem(c.back());
        c.pop_back();
        return elem;
    }

    // zwroc wartosc kolejnego elementu
    T& top () {
        if (c.empty()) {
            throw ReadEmptyStack();
        }
        return c.back();
    }
};

#endif /* STACK_HPP */
```

W przypadku zastosowania tej klasy stosu poprzedni przykład powinien zostać napisany następująco:

```
// cont/stack2.cpp

#include <iostream>
#include "Stack.hpp"       // wykorzystaj specjalna klase stosu
using namespace std;
```

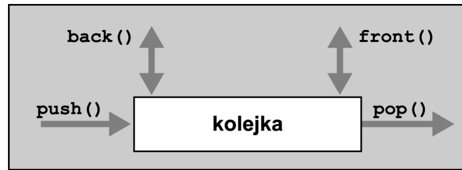
```
int main()
{
   try {
      Stack<int> st;

      // umiesc na stosie trzy elementy
      st.push(1);
      st.push(2);
      st.push(3);

      // zdejmij ze stosu dwa elementy i wypisz je
      cout << st.pop() << ' ';
      cout << st.pop() << ' ';

      // zmodyfikuj kolejny element
      st.top() = 77;

      // dodaj dwa nowe elementy
      st.push(4);
      st.push(5);

      // pobierz jeden element bez jego przetwarzania
      st.pop();

      /* pobierz trzy elementy i je wypisz
       * - Blad: o jeden element zbyt duzo
       */
      cout << st.pop() << ' ';
      cout << st.pop() << endl;
      cout << st.pop() << endl;
   }
   catch (const exception& e) {
      cerr << "WYJATEK: " << e.what() << endl;
   }
}
```

Dodatkowe wywołanie funkcji składowej `pop()` powoduje błąd. W przeciwieństwie do standardowej klasy stosu, ta zdefiniowana powyżej zwraca w takim przypadku wyjątek, zamiast zachowywać się w nieokreślony sposób. Wynik działania programu jest następujący:

```
3 2 4 77
WYJATEK: proba odczytania pustego elementu
```

10.2. Kolejki

Klasa `queue<>` implementuje kolejkę (znaną również pod nazwą kolejki FIFO). Funkcja składowa `push()` służy do wstawiania do niej dowolnej liczby elementów (patrz rysunek 10.3). Pobranie elementów z kolejki jest natomiast możliwe przy użyciu funkcji składowej o nazwie `pop()`. Następuje to w tej samej kolejności co ich umieszczanie („first in, first out", co w dosłownym tłumaczeniu oznacza „pierwszy wszedł, pierwszy wyszedł", przyp. tłum.).



RYSUNEK 10.3.
Interfejs kolejki

W celu wykorzystania kolejki konieczne jest dołączenie pliku nagłówkowego `<queue>`[5]:

```
#include <queue>
```

Deklaracja klasy `queue` w pliku nagłówkowym `<queue>` wygląda następująco:

```
namespace std {
  template <class T,
            class Container = deque<T> >
         class queue;
}
```

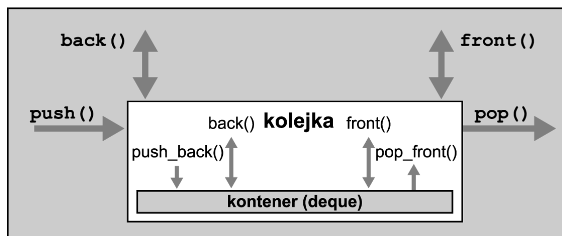
Pierwszy z parametrów wzorca określa rodzaj elementów. Drugi natomiast jest opcjonalny i definiuje kontener używany wewnętrznie w celu skolejkowania elementów umieszczonych we właściwym kontenerze kolejki. Domyślnie przyjmowany jest kontener o nazwie `deque`.

Na przykład poniższy wiersz zawiera deklarację określającą kolejkę, zawierającą łańcuchy znakowe[6]:

```
std::queue<string> buffer; //kolejka przechowujaca łańcuchy znakowe
```

Implementacja kolejki polega na prostym odwzorowaniu wykonywanych operacji na odpowiednie wywołania funkcji składowych używanego wewnętrznie kontenera (patrz rysunek 10.4). Możliwe jest użycie dowolnej klasy kontenera udostępniającej funkcje składowe o nazwie `front()`, `back()`, `push_back()` oraz `pop_front()`. I tak na przykład w charakterze kontenera elementów możliwe byłoby wykorzystanie listy lub wektora:

```
std::queue<std::string,std::list<std::string> > buffer;
```



RYSUNEK 10.4.
Wewnętrzny interfejs kolejki

---

[5] W oryginalnej bibliotece *STL* plik nagłówkowy zawierający deklarację kolejki nosił nazwę: `<stack.h>`.

[6] W poprzednich wersjach biblioteki *STL* istniała konieczność przekazywania kontenera w charakterze obowiązkowego argumentu wzorca. Dlatego też kolejka zawierająca łańcuchy znakowe musiała być wtedy deklarowana w następujący sposób:

```
queue<deque<string> > buffer;
```

10.2.1. Interfejs

W przypadku kolejek interfejs udostępniany jest poprzez funkcje składowe o nazwach `push()`, `front()`, `back()` oraz `pop()`:

- Funkcja składowa `push()` umieszcza element w kolejce.
- Funkcja składowa `front()` zwraca kolejny element z kolejki (element wstawiony do kolejki jako pierwszy).
- Funkcja składowa `back()` zwraca ostatni element z kolejki (element wstawiony do kolejki jako ostatni).
- Funkcja składowa `pop()` usuwa element z kolejki.

Zwróć uwagę, że funkcja składowa `pop()` usuwa kolejny element, lecz go nie zwraca, podczas gdy funkcje składowe `front()` oraz `back()` zwracają kolejny element bez jego usuwania. Dlatego też w celu przetworzenia oraz usunięcia kolejnego elementu z kolejki konieczne jest wywołanie funkcji składowych `front()`, a następnie `pop()`. Opisany interfejs jest nieco niewygodny, lecz działa znacznie lepiej w przypadku, gdy chcesz jedynie usunąć kolejny element bez jego analizy i przetwarzania. Zauważ, że działanie funkcji składowych `front()`, `back()` oraz `pop()` jest niezdefiniowane w przypadku, gdy stos nie zawiera żadnych elementów. W celu umożliwienia sprawdzenia, czy na stosie umieszczone są jakiekolwiek elementy, dodane zostały funkcje składowe `size()` oraz `empty()`.

Jeśli standardowy interfejs kontenera `queue<>` nie przypadł ci do gustu, możesz oczywiście w łatwy sposób napisać swój własny, bardziej wygodny w użyciu. Odpowiedni przykład zostanie umieszczony w podrozdziale 10.2.4. na stronie 413.

10.2.2. Przykład użycia kolejek

Poniższy przykład demonstruje wykorzystanie klasy `queue<>`:

```
// cont/queue1.cpp

#include <iostream>
#include <queue>
#include <string>
using namespace std;

int main()
{
    queue<string> q;

    // wstawia do kolejki trzy elementy
    q.push("To ");
    q.push("sa ");
    q.push("wiecej niz ");

    // odczytuje z kolejki dwa elementy i je wypisuje
    cout << q.front();
    q.pop();
    cout << q.front();
    q.pop();
```

```
    // wstawia nowe elementy
    q.push("cztery ");
    q.push("slowa!");

    // pomija jeden element
    q.pop();

    // odczytuje i wypisuje dwa elementy
    cout << q.front();
    q.pop();
    cout << q.front() << endl;
    q.pop();

    // wypisuje liczbe elementow w kolejce
    cout << "liczba elementow w kolejce: " << q.size()
         << endl;
}
```

Wynik działania programu wygląda następująco:

```
To sa wiecej niż cztery slowa!
liczba elementow w kolejce: 0
```

10.2.3. Klasa queue<> w szczegółach

Podobnie do klasy `stack<>` typowa implementacja klasy `queue<>` nie wymaga raczej szczegółowego opisu:

```
namespace std {
   template <class T, class Container = deque<T> >
   class queue {
     public:
       typedef typename Container::value_type value_type;
       typedef typename Container::size_type  size_type;
       typedef          Container             container_type;
     protected:
       Container c;  //kontener
     public:
       explicit queue(const Container& = Container());

       bool         empty() const              { return c.empty(); }
       size_type    size()  const              { return c.size(); }
       void         push(const value_type& x)  { c.push_back(x); }
       void         pop()                      { c.pop_front(); }
       value_type&  front()                    { return c.front(); }
       const value_type& front() const         { return c.front(); }
       value_type&  back()                     { return c.back(); }
       const value_type& back() const          { return c.back(); }
};

template <class T, class Container>
  bool operator==(const queue<T, Container>&,
                  const queue<T, Container>&);
template <class T, class Container>
  bool operator< (const queue<T, Container>&,
                  const queue<T, Container>&);
...// (inne operatory porownania)
}
```

Poniższa część tego podrozdziału zawiera szczegółowy opis pól oraz operacji.

Definicje typu

queue::**value_type**
- Rodzaj elementów umieszczonych w kontenerze.
- Składowa równoważna jest składowej *container*::value_type.

queue::**size_type**
- Typ całkowity bez znaku, określający rozmiar umieszczonych elementów.
- Składowa równoważna jest składowej *container*::size_type.

queue::**container_type**
- Rodzaj kontenera.

Funkcje składowe

queue::**queue** ()
- Konstruktor domyślny.
- Tworzy pustą kolejkę.

explicit *queue*::**queue** (const Container& *cont*)
- Tworzy kolejkę inicjalizowaną elementami umieszczonymi w obiekcie *cont*.
- Kopiowane są wszystkie elementy umieszczone w kontenerze *cont*.

size_type *queue*::**size** () const
- Zwraca bieżącą liczbę elementów.
- W celu sprawdzenia, czy kolejka jest pusta, używaj funkcji składowej o nazwie empty(), ponieważ może ona działać szybciej.

bool *queue*::**empty** () const
- Zwraca wartość logiczną, określającą, czy kolejka jest pusta (nie zawiera elementów).
- Funkcja składowa równoważna konstrukcji postaci *queue*::size()==0, lecz może od niej działać szybciej.

void *queue*::**push** (const *value_type*& *elem*)
- Wstawia jako nowy element kolejki kopię elementu określonego przez wartość *elem*.

value_type& *queue*::**front** ()
const *value_type*& *queue*::**front** () const
- Obie postaci funkcji składowej zwracają kolejny element kolejki. Będzie nim element wstawiony jako pierwszy (przed wszystkimi innymi elementami kolejki).
- Przed wywołaniem funkcji składowej należy upewnić się, że kolejka zawiera jakiekolwiek elementy (size()>0). W innym przypadku jej działanie jest nieokreślone.
- Pierwsza postać funkcji składowej przeznaczona jest dla stosów nieokreślonych jako stałe (*nonconstant*) i zwraca referencję. Dlatego też możliwe jest zmodyfikowanie

kolejnego elementu jeszcze w chwili, gdy jest on umieszczony w kolejce. Do ciebie należy decyzja, czy jest to dobry styl programowania.

`value_type& queue::`**`back`**` ()`
`const value_type& queue::`**`back`**` () const`

- Obie postaci funkcji składowej zwracają ostatni element kolejki. Będzie nim element wstawiony jako ostatni (po wszystkich innych elementach kolejki).
- Przed wywołaniem funkcji składowej należy upewnić się, że kolejka zawiera jakiekolwiek elementy (`size()>0`). W innym przypadku jej działanie jest nieokreślone.
- Pierwsza postać funkcji składowej przeznaczona dla stosów nieokreślonych jako statyczne (*nonconstant*) i zwraca referencję. Dlatego też możliwe jest zmodyfikowanie kolejnego elementu jeszcze w chwili, gdy jest on umieszczony w kolejce. Do ciebie należy podjęcie decyzji, czy jest to dobry styl programowania.

`void queue::`**`pop`**` ()`

- Usuwa kolejny element kolejki. Elementem tym będzie element wstawiony jako pierwszy (przed wszystkimi innymi elementami kolejki).
- Funkcja nie zwraca wartości. W celu przetworzenia kolejnego elementu musisz wcześniej wywołać funkcję składową `front()`.
- Przed wywołaniem funkcji składowej należy upewnić się, że stos zawiera jakiekolwiek elementy (`size()>0`). W innym przypadku jej działanie jest nieokreślone.

`bool `***`comparison`***` (const queue& queue1, const queue& queue2)`

- Zwraca wynik porównania dwóch kolejek tego samego rodzaju.
- ***`comparison`*** może być określony na jeden z poniższych sposobów:

```
operator ==
operator !=
operator <
operator >
operator <=
operator >=
```

- Dwie kolejki są jednakowe, w przypadku gdy zawierają taką samą liczbę elementów, z których wszystkie są jednakowe oraz umieszczone w tej samej kolejności (porównanie każdych dwóch odpowiadających sobie elementów musi dawać w rezultacie wartość `true`).
- W celu sprawdzenia, czy jedna kolejka jest mniejsza od drugiej, obie porównywane są leksykograficznie. Więcej informacji umieszczonych jest wraz z opisem algorytmu o nazwie `lexicographical_compare()` na stronie 338.

10.2.4. Klasa kolejki definiowanej przez użytkownika

Standardowa klasa `queue<>` preferuje szybkość działania nad wygodę oraz bezpieczeństwo użytkowania. Nie jest to tym, co zazwyczaj ja sam uważam za odpowiednie. Dlatego też napisałem własną klasę kolejki. Posiada ona dwie zalety:

1. Funkcja składowa `pop()` zwraca kolejny element.
2. Funkcje składowe `pop()` oraz `front()` zwracają wyjątek, w przypadku gdy kolejka jest pusta.

Dodatkowo pominąłem wszystkie funkcje składowe, które nie są niezbędne dla zwykłego użytkownika kolejki, jak chociażby operacje porównania oraz funkcję składową `back()`. Moja klasa kolejki zdefiniowana jest w nastepujący sposób:

```
// cont/Queue.hpp
/* ************************************************************
 * Queue.hpp
 *  - bezpieczniejsza i bardziej wygodna klasa kolejki
 * ************************************************************/
#ifndef QUEUE_HPP
#define QUEUE_HPP

#include <deque>
#include <exception>

template <class T>
class Queue {
  protected:
    std::deque<T> c;          // kontener przechowujacy elementy

  public:
    /* klasa wyjatku w przypadku wywolania funkcji składowych pop() oraz top() z pusta kolejka
     */
    class ReadEmptyQueue : public std::exception {
      public:
        virtual const char* what() const throw() {
            return "proba odczytania pustego elementu";
        }
    };

    // liczba elementow
    typename std::deque<T>::size_type size() const {
        return c.size();
    }

    // czy kolejka jest pusta?
    bool empty() const {
        return c.empty();
    }

    // wstawia element do kolejki
    void push (const T& elem) {
        c.push_back(elem);
    }

    // odczytuje element z kolejki i pobiera jego wartosc
    T pop () {
        if (c.empty()) {
            throw ReadEmptyQueue();
        }
        T elem(c.front());
        c.pop_front();
        return elem;
    }
```

```
    // pobiera wartosc kolejnego elementu
    T& front () {
        if (c.empty()) {
            throw ReadEmptyQueue();
        }
        return c.front();
    }
};

#endif /* QUEUE_HPP */
```

W przypadku zastosowania powyższej klasy kolejki poprzedni przykład powinien mieć następującą postać:

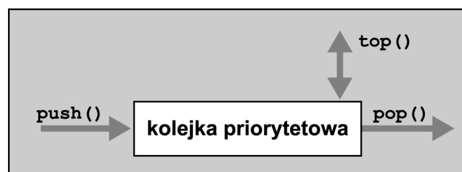
```
// cont/queue2.cpp

#include <iostream>
#include <string>
#include "Queue.hpp"      // uzywa specjalnej kolejki using namespace std;

int main()
{
   try {
      Queue<string> q;

      // wstawia do kolejki trzy elementy
      q.push("To ");
      q.push("sa ");
      q.push("wiecej niz ");

      // odczytuje z kolejki dwa elementy i je wyswietla
      cout << q.pop();
      cout << q.pop();

      // umieszcza w kolejce dwa nowe elementy
      q.push("cztery ");
      q.push("slowa!");

      // pomija jeden element
      q.pop();

      // odczytuje z kolejki dwa elementy i je wyswietla
      cout << q.pop();
      cout << q.pop() << endl;

      // wyswietla liczbe pozostalych elementow
      cout << "liczba elementow w kolejce: " << q.size()
           << endl;

      // odczytuje i wyswietla jeden element
      cout << q.pop() << endl;
   }
   catch (const exception& e) {
      cerr << "WYJATEK: " << e.what() << endl;
   }
}
```

Dodatkowe wywołanie funkcji składowej `pop()` powoduje wystąpienie błędu. W przeciwieństwie do standardowej klasy kolejki, ta zdefiniowana powyżej zwraca w takim przypadku wyjątek, zamiast zachowywania się w bliżej nieokreślony sposób. Wynik działania programu jest następujący:

```
To sa cztery slowa!
liczba elementow w kolejce: 0
WYJATEK: proba odczytania pustego elementu
```

10.3. Kolejki priorytetowe

Klasa o nazwie `priority_queue<>` implementuje kolejkę, z której elementy odczytywane są zgodnie z ich priorytetem. Interfejs jest bardzo podobny do zwykłych kolejek. To znaczy funkcja składowa `push()` wstawia elementy do kolejki, podczas gdy funkcje składowe `top()` oraz `pop()` służą do pobrania oraz usunięcia kolejnego elementu (rysunek 10.5). Tym niemniej kolejny element nie jest elementem, który został wstawiony jako pierwszy. Jest on za to elementem o najwyższym priorytecie. Dzięki temu wszystkie elementy są częściowo posortowane ze względu na swoją wartość. Podobnie jak zazwyczaj kryterium sortowania może zostać podane w charakterze parametru wzorca. Domyślnie wszystkie elementy posortowane są przy użyciu operatora < w kolejności malejącej. Dlatego też kolejny element w kolejce ma zawsze większą wartość. Jeśli istnieje więcej niż jeden taki element, kolejność ich pobierania jest niezdefiniowana.



RYSUNEK 10.5. Interfejs kolejki priorytetowej

Kolejki priorytetowe zdefiniowane są w tym samym pliku nagłówkowym co zwykłe kolejki: `<queue>`[7]

```
#include <queue>
```

Deklaracja klasy `priority_queue` w pliku nagłówkowym `<queue>` wygląda następująco:

```
namespace std {
  template <class T,
            class Container = vector<T>,
            class Compare = less<typename Container::value_type> >
         class priority_queue;
}
```

Pierwszy z parametrów wzorca określa rodzaj elementów. Drugi natomiast jest opcjonalny i definiuje kontener używany wewnętrznie w celu skolejkowania elementów umieszczonych we właściwym kontenerze kolejki priorytetowej. Domyślnie przyjmowany jest kontener o nazwie `vector`. Ostatni opcjonalny trzeci parametr definiuje kryterium sortowania używane w celu odszukania kolejnego elementu o najwyższym priorytecie. Domyślnie elementy prównywane są przy użyciu operatora <.

[7] W oryginalnej bibliotece *STL* kolejki priorytetowe zdefiniowane były w pliku o nazwie `<stack.h>`.

Na przykład poniższy wiersz zawiera deklarację, określającą kolejkę priorytetową zawierającą liczby rzeczywiste[8]:

```
std::priority_queue<float> pbuffer; //kolejka przechowujaca liczby rzeczywiste
```

Implementacja kolejki polega na prostym odwzorowaniu wykonywanych operacji na odpowiednie wywołania funkcji składowych używanego wewnętrznie kontenera. Możliwe jest użycie dowolnej klasy kontenera udostępniającej iteratory o swobodnym dostępie oraz funkcje składowe o nazwach `front()`, `push_back()` oraz `pop_front()`. Swobodny dostęp jest konieczny w celu sortowania elementów przy użyciu algorytmów stogowych (zostały one opisane w podrozdziale 9.9.4., na stronie 375). I tak na przykład możliwe byłoby wykorzystanie w charakterze kontenera elementów obiektu typu `deque`:

```
std::priority_queue<float,std::deque<float> > pbuffer;
```

W celu zdefiniowania własnego kryterium sortowania konieczne jest przekazanie funkcji lub obiektu funkcyjnego w charakterze predykatu binarnego, używanego przez algorytmy sortowania do porównania dwóch elementów (więcej informacji na temat kryteriów sortowania znajduje się w podrozdziale 6.5.2., na stronie 179 lub też w podrozdziale 8.1.1., na stronie 282). I tak na przykład poniższa deklaracja określa kolejkę priorytetową z sortowaniem odwrotnym:

```
std::priority_queue<float,std::vector<float>,
std::greater<float> > pbuffer;
```

W tego rodzaju kolejce kolejny element posiada zawsze mniejszą wartość.

10.3.1. Interfejs

W przypadku kolejek priorytetowych interfejs udostępniany jest poprzez funkcje składowe o nazwach `push()`, `top()` oraz `pop()`:

- Funkcja składowa `push()` umieszcza element w kolejce priorytetowej.
- Funkcja składowa `top()` zwraca kolejny element z kolejki priorytetowej.
- Funkcja składowa `pop()` usuwa element z kolejki priorytetowej.

Podobnie jak w przypadku innych adaptatorów kontenerów, funkcja składowa `pop()` usuwa kolejny element bez jego zwracania, podczas gdy funkcja składowa `top()` zwraca kolejny element bez jego usuwania. Dlatego też w celu przetworzenia oraz usunięcia kolejnego elementu kolejki priorytetowej konieczne jest zawsze wywołanie obu tych funkcji. Podobnie również jak zazwyczaj, działanie obu wspomnianych funkcji nie jest określone w przypadku, gdy kolejka priorytetowa nie zawiera żadnych elementów. W przypadku wątpliwości możesz zawsze w celu określenia ich liczby użyć funkcji składowych `size()` oraz `empty()`.

---

[8] W poprzednich wersjach biblioteki *STL* istniała konieczność przekazywania kontenera oraz kryterium sortowania w charakterze obowiązkowych argumentów wzorca. Dlatego też kolejka priorytetowa zawierająca liczby rzeczywiste musiała być wtedy deklarowana w następujący sposób:

```
priority_queue<vector<float>, less<float> > buffer;
```

10.3.2. Przykład użycia kolejek priorytetowych

Poniższy program demonstruje zastosowanie klasy `priority_queue<>`:

```
//cont/pqueue1.cpp
#include <iostream>
#include <queue>
using namespace std;

int main()
{
    priority_queue<float> q;

    // wstaw do kolejki priorytetowej trzy elementy
    q.push(66.6);
    q.push(22.2);
    q.push(44.4);

    // odczytaj i wypisz dwa elementy
    cout << q.top() << ' ';
    q.pop();
    cout << q.top() << endl;
    q.pop();

    // wstaw kolejne trzy elementy
    q.push(11.1);
    q.push(55.5);
    q.push(33.3);

    // pomin jeden element
    q.pop();

    // pobierz i wypisz pozostale elementy
    while (!q.empty()) {
        cout << q.top() << ' ';
        q.pop();
    }
    cout << endl;
}
```

A oto wynik działania powyższego programu:

```
66.6 44.4
33.3 22.2 11.1
```

Jak widać, po wstawieniu do kolejki wartości `66.6 22.2` oraz `44.4` jako elementy o najwyższych wartościach program wyświetla `66.6` oraz `44.4`. Po wstawieniu trzech kolejnych elementów kolejka zawiera liczby `22.2`, `11.1`, `55.5` oraz `33.3` (podane w kolejności ich wstawiania). Kolejny element pomijany jest poprzez wywołanie funkcji składowej `pop()`, dlatego też końcowa pętla powoduje wyświetlenie liczb w kolejności: `33.3`, `22.2` oraz `11.1`.

10.3.3. Klasa priority_queue<> w szczegółach

Większość operacji wykonywanych przez klasę `priority_queue<>` nie wymaga opisu, podobnie jak w przypadku klasy `stack<>` oraz `queue<>`:

```
namespace std {
   template <class T, class Container = vector<T>,
             class Compare = less<typename Container::value_type> >
   class priority_queue {
     public:
       typedef typename Container::value_type value_type;
       typedef typename Container::size_type  size_type;
       typedef          Container             container_type;
     protected:
       Compare comp;  //kryterium sortowania
       Container c;   //kontener
     public:
       //konstruktory
       explicit priority_queue(const Compare& cmp = Compare(),
                               const Container& cont = Container())
         : comp(cmp), c(cont) {
            make_heap(c.begin(),c.end(),comp);
       }

       template <class InputIterator>
       priority_queue(InputIterator first, InputIterator last,
                      const Compare& cmp = Compare(),
                      const Container& cont = Container())
         : comp(cmp), c(cont) {
           c.insert(c.end(),first,last);
           make_heap(c.begin(),c.end(),comp);
       }

       void push(const value_type& x); {
           c.push_back(x);
           push_heap(c.begin(),c.end(),comp);
       }
       void pop() {
           pop_heap(c.begin(),c.end(),comp);
           c.pop_back();
       }

       bool              empty() const          { return c.empty(); }
       size_type         size()  const          { return c.size(); }
       const value_type& top() const            { return c.front(); }
    };
}
```

Jak można zauważyć, kolejka priorytetowa używa algorytmów stogowych, zdefiniowanych w bibliotece *STL*. Algorytmy te opisane są w podrozdziale 9.9.4. na stronie 375. Zwróć uwagę jednak, że w odróżnieniu od innych kontenerów nie zostały w tym miejscu zdefiniowane żadne operatory porównania.

Poniższa część tego podrozdziału zawiera szczegółowy opis pól oraz operacji.

Definicje typu

`priority_queue``::`**`value_type`**

- Rodzaj elementów umieszczonych w kontenerze.
- Składowa równoważna jest składowej *`container`*`::value_type`.

`priority_queue``::`**`size_type`**

- Typ całkowity bez znaku, określający rozmiar umieszczonych elementów.
- Składowa równoważna jest składowej *`container`*`::size_type`.

priority_queue::**container_type**
- Rodzaj kontenera.

Konstruktory

priority_queue::**priority_queue** ()
- Konstruktor domyślny.
- Tworzy pustą kolejkę priorytetową.

explicit *priority_queue*::**priority_queue** (const CompFunc& *op*)
- Tworzy pustą kolejkę priorytetową wykorzystującą kryterium sortowania określone przez argument *op*.
- Przykłady przedstawiające sposób przekazywania kryterium sortowania w postaci argumentu konstruktora przedstawione zostały na stronie 191 oraz 210.

```
priority_queue::priority_queue (const CompFunc& op,
                                const Container& cont)
```
- Tworzy kolejkę priorytetową inicjalizowaną elementami zawartymi w kontenerze `cont` oraz używającą kryterium sortowania przekazane w argumencie *op*.
- Kopiowane są wszystkie elementy umieszczone w kontenerze `cont`.

```
priority_queue::priority_queue (InputIterator beg,
                                InputIterator end)
```
- Tworzy kolejkę priorytetową inicjalizowaną elementami należącymi do zakresu *[beg,end)*.
- Funkcja ta jest wzorcem (patrz strona 27), dlatego też elementy zakresu źródłowego mogą być dowolnego typu, który możliwy jest do przekształcenia na typ elementu umieszczonego w kontenerze.

```
priority_queue::priority_queue (InputIterator beg,
                                InputIterator end,
                                const CompFunc& op)
```
- Tworzy kolejkę priorytetową inicjalizowaną elementami należącymi do zakresu *[beg,end)*, używającą kryterium sortowania przekazane w argumencie *op*.
- Funkcja ta jest wzorcem (patrz strona 27), dlatego też elementy zakresu źródłowego mogą być dowolnego typu, który możliwy jest do przekształcenia na typ elementu umieszczonego w kontenerze.
- Przykłady przedstawiające sposób przekazywania kryterium sortowania w postaci arugumentu konstruktora przedstawione zostały na stronie 191 oraz 210.

```
priority_queue::priority_queue (InputIterator beg,
                                InputIterator end,
                                const CompFunc& op,
                                const Container& cont)
```
- Tworzy kolejkę priorytetową inicjalizowaną elementami zawartymi w kontenerze `cont` oraz należącymi do zakresu *[beg, end)*, używającą kryterium sortowania przekazane w argumencie *op*.

- Funkcja ta jest wzorcem (patrz strona 27), dlatego też elementy zakresu źródłowego mogą być dowolnego typu, który możliwy jest do przekształcenia na typ elementu umieszczonego w kontenerze.

Inne funkcje składowe

`size_type` *`priority_queue`*`::`**`size`**` () const`
- Zwraca bieżącą liczbę elementów.
- W celu sprawdzenia, czy kolejka jest pusta, używaj funkcji składowej o nazwie *`empty()`*, ponieważ może ona działać szybciej.

`bool` *`priority_queue`*`::`**`empty`**` () const`
- Zwraca wartość logiczną, określającą, czy kolejka jest pusta (nie zawiera elementów).
- Funkcja składowa równoważna konstrukcji postaci *`priority_queue`*`::size() ==0`, lecz może działać od niej szybciej.

`void` *`priority_queue`*`::`**`push`**` (const` *`value_type& elem`*`)`
- Wstawia do kolejki kopię elementu określonego przez wartość *`elem`*.

`const` *`value_type& priority_queue`*`::`**`top`**` () const`
- Zwraca kolejny element kolejki priorytetowej. Będzie nim element posiadający największą wartość z wszystkich elementów kolejki. Jeśli istnieje więcej niż jeden taki element, nie jest zdefiniowane, który z nich zostanie zwrócony.
- Przed wywołaniem funkcji składowej należy upewnić się, czy kolejka zawiera jakiekolwiek elementy (*`size()>0`*). W innym przypadku jej działanie jest nieokreślone.

`void` *`priority_queue`*`::`**`pop`**` ()`
- Usuwa kolejny element kolejki priorytetowej. Będzie nim element posiadający maksymalną wartość z wszystkich elementów kolejki. Jeśli istnieje więcej niż jeden taki element, nie jest zdefiniowane, który z nich zostanie usunięty.
- Funkcja nie zwraca wartości. W celu przetworzenia kolejnego elementu musisz wcześniej wywołać funkcję składową `top()`.
- Przed wywołaniem funkcji składowej należy upewnić się, czy stos zawiera jakiekolwiek elementy (`size()>0`). W innym przypadku jej działanie jest nieokreślone.

10.4. Kontener bitset

Kontenery typu `bitset` są tablicami o ustalonym rozmiarze, zawierającymi bity lub wartości logiczne. Są one przydatne do zarządzania zestawami znaczników, w przypadku których odpowiednie zmienne mogą reprezentować dowolną kombinację znaczników. Programy utworzone w języku C oraz starszych standardach języka C++ wykorzystują zazwyczaj w charakterze tablic bitów zmienne typu `long`, manipulując nimi przy użyciu operatorów bitowych w rodzaju `&`, `|` oraz `~`. Klasa `bitset` posiada nad takim podejściem

tę przewagę, że może przechowywać dowolną liczbę bitów, jak również zawiera dodatkowe operatory, służące do zmiany stanu bitów. Na przykład możliwe jest przyporządkowywanie pojedynczych bitów i odczytywanie oraz zapisywanie ich całych zestawów poprzez użycie sekwencji zer i jedynek.

Zwróć uwagę na fakt, iż niemożliwa jest zmiana liczby bitów w danym kontenerze `bitset`. Ich liczba określona jest jako parametr wzorca. Jeśli zachodzi potrzeba wykorzystania zmiennej liczby bitów lub wartości logicznych, możliwe jest wykorzystanie zmiennej typu `vector<bool>` (opisanej w podrozdziale 6.2.6., na stronie 161).

Klasa `bitset` zdefiniowana jest w pliku nagłówkowym `<bitset>`:

```
#include <bitset>
```

W pliku nagłówkowym `<bitset>` klasa `bitset` zdefiniowana jest w postaci klasy wzorca, posiadającego parametr, określający liczbę bitów:

```
namespace std {
     template <size_t Bits>
     class bitset;
}
```

W tym przypadku parametr nie jest określonym typem, lecz wartością całkowitą bez znaku (ta cecha języka została przedstawiona na stronie 26).

Wzorce posiadające różne argumenty stanowią różne typy danych. Możliwe jest porównywanie oraz łączenie kontenerów typu `bitset` zawierających tę samą liczbę bitów.

10.4.1. Przykłady użycia kontenerów bitset

Wykorzystanie kontenerów bitset do przechowywania zestawu znaczników

Pierwszy przykład zastosowania klas `bitset` przedstawia sposób ich wykorzystania do zarządzania zestawem znaczników. Każdy znacznik posiada wartość określoną za pomocą typu wyliczeniowego. Wartość ta została wykorzystana do określenia pozycji bitu. W tym przykładzie bity reprezentują kolory. Dlatego też również każda wartość typu wyliczeniowego `Color` określa jeden kolor. Dzięki zastosowaniu klasy `bitset` możliwe jest wykorzystanie zmiennych, które mogą zawierać dowolną kombinację kolorów:

```
//cont/bitset1.cpp
#include <bitset>
#include <iostream>
using namespace std;

int main()
{
     /* typ wyliczeniowy uzywany w klasie bitset
      * - kazdy bit reprezentuje kolor
      */
     enum Color { red, yellow, green, blue, white, black, ...,
                  numColors };
```

```
    // tworzy kontener bitset dla wszystkich bitow (kolorow)
    bitset<numColors> usedColors;

    // ustawia bity dla dwoch kolorow
    usedColors.set(red);
    usedColors.set(blue);

    // wypisuje informacje o niektorych danych przechowywanych
    // w klasie bitset
    cout << "wartosci bitowe uzytych kolorow:   " << usedColors
         << endl;
    cout << "liczba uzytych kolorow:   " << usedColors.count()
         << endl;
    cout << "wartosci bitowe niewykorzystanych kolorow: " << ~usedColors
         << endl;

    // jesli zostal wykorzystany jakikolwiek kolor
    if (usedColors.any()) {
        // przejrzyj wszystkie kolory
        for (int c = 0; c < numColors; ++c) {
            // jesli wykorzystany zostal rzeczywisty kolor
            if (usedColors[(Color)c]) {
                //...
            }
        }
    }
}
```

Wykorzystanie kontenerów bitset do operacji wejścia-wyjścia korzystających z reprezentacji bitowych

Jedną z przydatnych właściwości klasy `bitset` jest możliwość przekonwertowania wartości całkowitych do postaci sekwencji bitów i na odwrót. Konieczne jest w tym celu wykorzystanie tymczasowego obiektu klasy `bitset`:

```
// cont/bitset2.cpp
#include <bitset>
#include <iostream>
#include <string>
#include <limits>
using namespace std;

int main()
{
    /* wyswietla reprzentacje bitowa kilku liczb
     */
    cout << "267 w postaci binarnej liczby typu short:     "
         << bitset<numeric_limits<unsigned short>::digits>(267)
         << endl;

    cout << "267 w postaci binarnej liczby typu long:      "
         << bitset<numeric_limits<unsigned long>::digits>(267)
         << endl;

    cout << "10,000,000 w postaci binarnej liczby 24 bitowej: "
         << bitset<24>(1e7) << endl;

    /* przeksztalc reprezentacje binarna do postaci liczby calkowitej
     */
```

```
        cout << "\"1000101011\" w postaci liczbowej:  "
             << bitset<100>(string("1000101011")).to_ulong() << endl;
    }
```

W zależności od liczby bitów przeznaczonych do reprezentowania liczby typu `short` oraz `long`, program mógłby dać następujące wyniki:

```
267 w postaci binarnej liczby typu short:        0000000100001011
267 w postaci binarnej liczby typu long:         00000000000000000000000100001011
10,000,000 w postaci binarnej liczby 24 bitowej: 100110001001011010000000
"1000101011" w postaci liczbowej:                555
```

W tym przykładzie zastosowanie klasy:

```
bitset<numeric_limits<unsigned short>::digits>(267)
```

powoduje przekształcenie liczby 267 do postaci kontenera `bitset`, zawierającego bity reprezentujące wartości typu `unsigned short` (omówienie największych możliwych do reprezentacji wartości liczbowych zostało umieszczone w podrozdziale 4.3., na stronie 72). Odpowiedni operator zdefiniowany w klasie `bitset` powoduje wyświetlenie sekwencji bitów w postaci znaków 0 oraz 1.

Analogicznie użycie

```
bitset<100>(string("1000101011"))
```

przekształca sekwencję znaków reprezentujących bity do postaci obiektu klasy `bitset`, którego funkcja składowa `to_ulong()` powoduje zwrócenie wartości całkowitej. Zwróć uwagę, że liczba bitów powinna być mniejsza od wartości wyrażenia `sizeof(unsigned long)`. W przypadku gdy dana wartość nie będzie mogła zostać reprezentowana w postaci zmiennej typu `unsigned long`, wygenerowany zostanie wyjątek[9].

10.4.2. Szczegółowy opis klasy bitset

Klasa `bitset` zawiera następujące operacje.

Funkcje składowe służące do tworzenia, kopiowania oraz usuwania obiektów klasy bitset

W przypadku klasy `bitset` zostały zdefiniowane pewne szczególne konstruktory. Tym niemniej nie istnieje zdefiniowany żaden specjalny konstruktor kopiujący, operator

---

[9] Zwróć uwagę, iż wartość używana do inicjalizacji musi zostać najpierw jawnie przekształcona do postaci `string`. Stanowi to najprawdopodobniej błąd standardu, ponieważ w jego wcześniejszych wersjach możliwe było użycie konstrukcji postaci:

```
bitset<100>("1000101011")
```

Tego rodzaju niejawna konwersja została przez przypadek pominięta podczas definiowania różnych konstruktorów tego wzorca. W programie został zaprezentowany proponowany sposób rozwiązania tego problemu.

przypisania ani destruktor. Dlatego też obiekty klasy `bitset` są przypisywane i kopiowane przy użyciu domyślnych operacji na poziomie bitowym.

bitset<*bits*>::bitset ()

- Konstruktor domyślny.
- Tworzy zestaw bitów zainicjalizowany zerami.
- Na przykład:

```
bitset<50> flags:   //zmienna flags: 0000...000000
                    //dlatego tez zawiera 50 niezainicjalizowanych bitow
```

bitset<*bits*>::bitset (unsigned long *value***)**

- Tworzy zestaw bitów zainicjalizowany zgodnie z bitami wartości całkowitej `value`.
- Jeśli liczba bitów wartości *`value`* jest zbyt mała, początkowe bity inicjalizowane są zerami.
- Na przykład:

```
bitset<50> flags(7):   //zmienna flags: 0000...000111
```

explicit **bitset<*bits*>::bitset** (const *string& str*)
bitset<*bits*>::bitset (const *string& str*, *string::*size_type *str_idx*)
bitset<*bits*>::bitset (const *string& str*, *string::*size_type *str_idx*, *string::*size_type *str_num*)

- Wszystkie formy funkcji tworzą kontener typu `bitset` inicjalizowany przy użyciu łańcucha znakowego *`str`* lub jego fragmentu.
- Łańcuch znakowy lub jego część mogą zawierać jedynie znaki 0 oraz 1.
- Wartość *`str_idx`* zawiera indeks pierwszego znaku łańcucha *`str`*, używanego do inicjalizacji.
- W przypadku pominięcia wartości *`str_num`* używane są wszystkie znaki umieszczone w łańcuchu znakowym *`str`* począwszy od pozycji określonej wartością argumentu *`str_idx`*, aż do jego końca.
- Jeśli łańcuch znakowy lub jego część zawierają mniejszą liczbę znaków, niż jest to wymagane, początkowe bity inicjalizowane są zerami.
- Jeśli łańcuch znakowy lub jego część zawierają więcej znaków, niż jest to konieczne, pozostałe znaki są ignorowane.
- W przypadku gdy *`str_idx`* `>` *`str`*`.size()`, generowany jest wyjątek `out_of_range`.
- W przypadku gdy jeden ze znaków jest inny niż dozwolony 0 lub 1, generowany jest wyjątek `invalid_argument`.
- Zwróć uwagę, że ten konstruktor jest tylko wzorcem funkcji składowej (patrz strona 11), dlatego też w przypadku pierwszego argumentu nie została zdefiniowana niejawna kowersja typu `const char*` na `string`[10].

---

[10] Jest to najprawdopodobniej pomyłka podczas definiowania standardu, ponieważ w przypadku wcześniejszych jego wersji możliwe było użycie konstrukcji postaci:

```
bitset<50>("1010101")
```

Tego rodzaju niejawna konwersja została przez przypadek pominięta podczas definiowania różnych konstruktorów tego wzorca. W poniższym kodzie został zaprezentowany proponowany sposób rozwiązania tego problemu.

- Na przykład:

```
bitset<50>flags(string("1010101"));          //zmienna flags: 0000...0001010101
bitset<50>flags(string("1111000"), 2, 3);    //zmienna flags: 0000...0000000110
```

Funkcje składowe niezmieniające wartości kontenera bitset

`size_t` **`bitset<`**_`bits`_**`>::size`** `() const`
- Zwraca liczbę używanych bitów.

`size_t` **`bitset<`**_`bits`_**`>::count`** `() const`
- Zwraca liczbę ustawionych bitów (bitów o wartości 1).

`bool` **`bitset<`**_`bits`_**`>::any`** `() const`
- Zwraca wartość logiczną, określającą, czy został ustawiony jakikolwiek bit.

`bool` **`bitset<`**_`bits`_**`>::none`** `() const`
- Zwraca wartość logiczną, określającą, czy nie został ustawiony żaden bit.

`bool` **`bitset<`**_`bits`_**`>::test`** `(size_t` _`idx`_`) const`
- Zwraca wartość logiczną, określającą, czy został ustawiony bit na pozycji _`idx`_.
- W przypadku gdy _`idx`_ `>= size()` zwraca wyjątek `out_of_range`.

`bool` **`bitset<`**_`bits`_**`>::operator==`** `(const bitset<`_`bits`_`>&` _`bits`_`) const`
- Zwraca wartość logiczną, określającą, czy bity umieszczone w kontenerze wskazywanym przez `*this` oraz `bits` posiadają tę samą wartość.

`bool` **`bitset<`**_`bits`_**`>::operator!=`** `(const bitset<`_`bits`_`>&` _`bits`_`) const`
- Zwraca wartość logiczną, określającą, czy bity umieszczone w kontenerze wskazywanym przez `*this` oraz `bits` posiadają różną wartość.

Funkcje składowe zmieniające wartości kontenera bitset

`bitset<`_`bits`_`>&` **`bitset<`**_`bits`_**`>::set`** `()`
- Ustawia wszystkie bity, nadając im wartość `true`.
- Zwraca zmodyfikowany zestaw bitów.

`bitset<`_`bits`_`>&` **`bitset<`**_`bits`_**`>::set`** `(size_t` _`idx`_`)`
- Ustawia bit na pozycji _`idx`_, nadając mu wartość `true`.
- Zwraca zmodyfikowany zestaw bitów.
- W przypadku gdy _`idx`_ `>= size()`, generuje wyjątek `out_of_range`.

`bitset<`_`bits`_`>&` **`bitset<`**_`bits`_**`>::set`** `(size_t` _`idx`_`, int` _`value`_`)`
- Ustawia bit na pozycji _`idx`_, nadając mu wartość _`value`_.
- Zwraca zmodyfikowany zestaw bitów.

- Wartość określona przez *value* przetwarzana jest jak wartość logiczna (Boolean). W przypadku gdy jest ona równa 0, bit ustawiany jest jako false. Każda inna wartość powoduje ustawienie bitu jako true.
- W przypadku gdy *idx* >= size(), generuje wyjątek out_of_range.

bitset<*bits*>& **bitset<*bits*>::reset** ()
- Zeruje wszystkie bity, nadając im wartość false.
- Zwraca zmodyfikowany zestaw bitów.

bitset<*bits*>& **bitset<*bits*>::reset** (size_t *idx*)
- Zeruje bit na pozycji *idx*, nadając mu wartość *false*.
- Zwraca zmodyfikowany zestaw bitów.
- W przypadku gdy *idx* >= size(), generuje wyjątek out_of_range.

bitset<*bits*>& **bitset<*bits*>::flip** ()
- Zamienia wartości wszystkich bitów (ustawia bity nieustawione oraz na odwrót).
- Zwraca zmodyfikowany zestaw bitów.

bitset<*bits*>& **bitset<*bits*>::flip** (size_t idx)
- Zamienia wartość bitu na pozycji *idx* (ustawia bit nieustawiony oraz na odwrót).
- Zwraca zmodyfikowany zestaw bitów.
- W przypadku gdy *idx* >= size(), generuje wyjątek out_of_range.

bitset<*bits*>& **bitset<*bits*>::operator^=** (const bitset<*bits*>& *bits*)
- Operator dokonujący bitowej operacji xor (exclusive or, czyli różnica symetryczna, przyp. tłum.).
- Zamienia wartości wszystkich bitów ustawionych również w zmiennej *bits* i pozostawia niezmienione wszystkie inne wartości.
- Zwraca zmodyfikowany zestaw bitów.

bitset<*bits*>& **bitset<*bits*>::operator|=** (const bitset<*bits*>& *bits*)
- Operator dokonujący bitowej operacji or (lub).
- Ustawia wartości wszystkich bitów ustawionych również w zmiennej *bits* i pozostawia niezmienione wszystkie inne wartości.
- Zwraca zmodyfikowany zestaw bitów.

bitset<*bits*>& **bitset<*bits*>::operator&=** (const bitset<*bits*>& *bits*)
- Operator dokonujący bitowej operacji and (i).
- Zeruje wartości wszystkich bitów nieustawionych w zmiennej *bits* i pozostawia niezmienione wszystkie inne wartości.
- Zwraca zmodyfikowany zestaw bitów.

bitset<*bits*>& **bitset<*bits*>::operator<<=** (size_t *num*)
- Przesuwa wszystkie bity w lewo o liczbę pozycji określoną za pomocą wartości *num*.
- Zwraca zmodyfikowany zestaw bitów.
- Pierwszych *num* bitów ustawianych zostaje jako false.

`bitset<bits>& bitset<bits>::operator>>= (size_t num)`
- Przesuwa wszystkie bity w prawo o liczbę pozycji określoną za pomocą wartości *num*.
- Zwraca zmodyfikowany zestaw bitów.
- Ostatnich *num* bitów ustawianych zostaje jako `false`.

Dostęp do bitów przy użyciu operatora []

`bitset<bits>::reference bitset<bits>::operator[] (size_t idx)`
`bool bitset<bits>::operator[] (size_t idx) const`
- Obie wersje funkcji zwracają bit umieszczony na pozycji wskazywanej przez wartość *idx*.
- Pierwsza funkcja używa typu pośredniczącego (`proxy`) w celu umozliwienia zwrócenia wartości modyfikowalnej (`lvalue`). Szczegółowe informacje znajdziesz w dalszej części podrozdziału.
- Przed wywołaniem funkcji konieczne jest sprawdzenie, czy wartość *idx* jest poprawnym indeksem. W innym przypadku jej zachowanie jest niezdefiniowane.

W przypadku wywołania dla niestatycznych (`nonconstant`) komponentów klasy `bitset`, operator `[]` zwraca specjalny tymczasowy obiekt typu `bitset<>::reference`. Obiekt ten używany jest w charakterze obiektu pośredniczącego[11] (`proxy`) pozwalającego na wykonywanie określonych modyfikacji z bitem wskazywanym przez operator `[]`. W szczególności dla typu `reference` zdefiniowanych zostało pięć różnych operacji:

1. `reference& operator= (bool)`
   Ustawia określony bit zgodnie z przekazaną wartością.
2. `reference& operator= (const reference&)`
   Ustawia określony bit zgodnie z wartością wskazywaną przez podaną referencję.
3. `reference& flip ()`
   Zamienia wartość bitu na przeciwną.
4. `operator bool () const`
   Konwertuje w sposób automatyczny podaną wartość do wartości typu `Boolean` (wartości logicznej).
5. `bool operator~ () const`
   Zwraca wartość dopełniającą (inaczej komplementarna, czyli wartość przeciwną) danego bitu.

Możliwe jest na przykład utworzenie kodu zawierającego poniższe instrukcje:

```
bitset<50> flags;
...
flags[42] = true;          //nadaje wartosc bitowi o indeksie 42
flags[13] = flags[42];     //przyporzadkowuje wartosc bitu o indeksie 42 bitowi o indeksie 13
flags[42].flip();          //zamienia wartosc bitu o indeksie 42
if (flags[13]) {           //jesli ustawiony jest bit o indeksie 13
    flags[10] = ~flags[42];    //wtedy przyporzadkowuje bitowi o indeksie 10 wartosc
                               //komplementarna bitu o indeksie 42
}
```

[11] Obiekt tego rodzaju pozwala na większą kontrolę nad wykonywanymi operacjami. Jest często stosowany w celu zwiększenia bezpieczeństwa. W takim przypadku zazwyczaj zezwala on na wykonywanie konkretnych operacji, chociaż wartość zwracana zachowuje się w zasadzie dokładnie w ten sam sposób co wartość `bool`.

Tworzenie nowych kontenerów typu bitset

```
bitset<bits> bitset<bits>::operator~ () const
```

- Zwraca nowy zestaw bitów, posiadający wszystkie bity ustawione przeciwnie do bitów wskazywanych, przez wskaźnik *this.

```
bitset<bits> bitset<bits>::operator<< (size_t num) const
```

- Zwraca nowy zestaw bitów, posiadający wszystkie bity przesunięte w lewo, o liczbę miejsc wskazywaną przez argument *num*.

```
bitset<bits> bitset<bits>::operator>> (size_t num) const
```

- Zwraca nowy zestaw bitów, posiadający wszystkie bity przesunięte w prawo, o liczbę miejsc wskazywaną przez argument *num*.

```
bitset<bits> bitset<bits>::operator& (const bitset<bits>& bits1,
                                      const bitset<bits>& bits2)
```

- Zwraca kontener bitset, którego zawartość jest wynikiem wykonania logicznej operacji and na zestawach bitów *bits1* oraz *bits2*.
- Zwraca nowy zestaw bitów z ustawionymi tymi bitami, które są również ustawione w *bits1* oraz *bits2*.

```
bitset<bits> bitset<bits>::operator| (const bitset<bits>& bits1,
                                      const bitset<bits>& bits2)
```

- Zwraca kontener bitset, którego zawartość jest wynikiem wykonania logicznej operacji or na zestawach bitów *bits1* oraz *bits2*.
- Zwraca nowy zestaw bitów z ustawionymi tymi bitami, które są również ustawione w *bits1* lub *bits2*.

```
bitset<bits> bitset<bits>::operator^ (const bitset<bits>& bits1,
                                      const bitset<bits>& bits2)
```

- Zwraca kontener bitset, którego zawartość jest wynikiem wykonania logicznej operacji xor na zestawach bitów *bits1* oraz *bits2*.
- Zwraca nowy zestaw bitów z ustawionymi tymi bitami, które są również ustawione w *bits1*, lecz nie w *bits2* i na odwrót.

Funkcje składowe służące do konwersji typów

```
unsigned long bitset<bits>::to_ulong () const
```

- Zwraca wartość całkowitą reprezentowaną poprzez bity umieszczone w przekazanym zbiorze bitów.
- W przypadku gdy wartość ta nie może zostać zaprezentowana przy użyciu typu unsigned long, generuje wyjątek overflow_error.

```
string bitset<bits>::to_string () const
```

- Zwraca łańcuch znakowy, zawierający wartość zbioru bitów zapisaną w reprezentacji binarnej (0 dla bitów nieustawionych oraz 1 dla ustawionych).
- Kolejność znaków jest równoważna kolejności uporządkowania bitów wraz ze zmniejszaniem indeksu.

- Funkcja ta jest jedynie wzorcem funkcji parametryzowanym przez typ zwracanej wartości. Zgodnie z regułami języka musisz użyć poniższej konwencji:

```
bitset<50> b;
...
b.template to_string<char,char_traits<char>,allocator<char> >()
```

Operacje wejścia-wyjścia

`istream& operator>> (istream& strm, bitset<bits>& bits)`

- Wczytuje do zmiennej *bits* zestaw bitów określony w postaci sekwencji znaków `0` oraz `1`.
- Kontynuuje odczyt do czasu wystąpienia jednej z poniższych sytuacji:
  - Odczytane zostały wszystkie znaki, które mogą zostać umieszczone w kontenerze określonym przez zmienną *bits*.
  - W strumieniu *strm* wystąpił znak końca pliku.
  - Kolejny znak nie jest ani `0`, ani `1`.
- Zwraca strumień wejściowy *strm*.
- Jeśli liczba odczytanych bitów jest mniejsza niż liczba bitów w kontenerze `bitset`, jest on wypełniany zerami umieszczonymi w charakterze poczatkowych bitów dopełniających.
- Jeśli operator ten nie może odczytać żadnego znaku w strumieniu *strm*, ustawia wartość `ios::failbit`, co może spowodować wygenerowanie odpowiadającego wyjątku (patrz podrozdział 13.4.4., strona 557).

`ostream& operator<< (ostream& strm, const bitset<bits>& bits)`

- Zapisuje bity umieszczone w zmiennej *bits*, przekonwertowane do postaci łańcucha znakowego zawierającego ich reprezentację binarną (i dlatego też będzie on zawierać sekwencję znaków `0` oraz `1`).
- W celu utworzenia znaków umieszczonych w strumieniu wyjściowym używa funkcji składowej `to_string()` (patrz strona 430).
- Zwraca strumień wyjściowy *strm*.
- Przykład użycia tego operatora został zaprezentowany na stronie 423.