

Część IV

Biblioteka standardowa

W tej części znajduje się opis biblioteki standardowej języka C++. Głównym celem jest pokazanie, jak posługiwać się tą biblioteką, przedstawienie ogólnie przydatnych technik projektowania i programowania oraz wskazanie sposobów rozszerzania biblioteki w prawidłowy sposób.

Rozdziały

- 30. Przegląd zawartości biblioteki standardowej
- 31. Kontenery STL
- 32. Algorytmy STL
- 33. Iteratory STL
- 34. Pamięć i zasoby
- 35. Narzędzia pomocnicze
- 36. Łańcuchy
- 37. Wyrażenia regularne
- 38. Strumienie wejścia i wyjścia
- 39. Lokalizacja
- 40. Liczby
- 41. Współbieżność
- 42. Wątki i zadania
- 43. Biblioteka standardowa C
- 44. Zgodność

„.... Właśnie odkrywam, jak trudno jest przelać myśli na papier. Samo opisywanie faktów nie stanowi zbyt wielkiego problemu; dopiero przedstawianie toku myślenia, tak aby był spójny i jasny, a zarazem umiarkowanie płynny, stanowi dla mnie trudność, z jaką, jak już wspomniałem, nie miałem nigdy wcześniej do czynienia...”

— Karol Darwin

Przegląd zawartości biblioteki standardowej

*Wiele sekretów sztuki i natury
niedouczonym wydaje się magią*
— Roger Bacon

- Wprowadzenie
 - Narzędzia biblioteki standardowej; Kryteria projektowe; Styl opisu
- Nagłówki
- Wsparcie dla języka
 - Wsparcie dla list inicjacyjnych; Wsparcie dla zakresowych pętli for
- Obsługa błędów
 - Wyjątki; Asercje; system_error
- Rady

30.1. Wprowadzenie

Biblioteka standardowa to zbiór opisanych w standardzie ISO języka C++ składników, które w każdej implementacji zachowują się tak samo (choć mogą się różnić pod względem wydajności). Aby program był jak najłatwiejszy do przenoszenia między różnymi platformami i jak najłatwiejszy w obsłudze programistycznej, zalecam używanie składników biblioteki standardowej zawsze, gdy jest to możliwe. Niewykłuczone, że w konkretnym programie lepiej pod jakimś względem sprawdzałyby się własnoręcznie napisane rozwiązania, ale:

- Czy programiście zajmującemu się programem w przyszłości łatwo będzie nauczyć się nowego projektu?
- Czy można oczekwać, że taki alternatywny projekt będzie działał także na nieznanych jeszcze platformach, które pojawią się dopiero za dziesięć lat?
- Czy ten alternatywny projekt będzie nadawał się do użytku w innych aplikacjach?
- Czy Twój projekt będzie dobrze współpracował z kodem napisanym przy użyciu narzędzi z biblioteki standardowej?
- Czy będziesz mógł poświęcić na testowanie swojego kodu tyle samo wysiłku i czasu, ile poświęcono bibliotece standardowej?

I oczywiście jeśli zdecydujesz się na zastosowanie własnego rozwiązania, to będziesz (lub Twoja firma będzie) odpowiedzialny za jego obsługę i rozwój już „na zawsze”. Najogólniej rzecz biorąc, staraj się nie wynajdywać koła na nowo.

Biblioteka standardowa jest dość duża. Jej specyfikacja w standardzie ISO języka C++ zajmuje 785 gęsto zapisanych stron. Do tego należałoby jeszcze doliczyć opis biblioteki standardowej języka C, która wchodzi w skład biblioteki C++ (kolejnych 139 stron). Dla porównania specyfikacja samego języka C++ zajmuje 398 stron. W tym rozdziale zamieściłem zestawienie w postaci tabel tego, co jest dostępne, oraz podałem kilka przykładów. Szczegółowe informacje można znaleźć w innych miejscach, np. dostępnych w internecie kopiach standardu, kompletnych dokumentacjach implementacji oraz (jeśli ktoś lubi czytać kod źródłowy) w otwartych implementacjach. Pełne informacje zawsze można znaleźć w standardzie, do którego zamieściłem wiele odwołań.

Rozdziałów zawierających opis biblioteki standardowej nie należy czytać po kolei. Każdy z nich i praktycznie każdy podrozdział można czytać niezależnie od reszy. W razie wątpliwości należy korzystać z odwołań i zamieszczonego na końcu indeksu.

30.1.1. Narzędzia biblioteki standardowej

Co powinna zawierać biblioteka standardowa języka C++? Programista chciałby w niej znaleźć każdą interesującą, znaczącą i sensownie ogólną klasę, funkcję, szablon itd. Ale pytanie nie brzmi „Co powinna zawierać *jakaś* biblioteka?”, tylko „Co powinna zawierać biblioteka *standardowa*?”. Na pierwsze pytanie można by odpowiedzieć „Wszystko!”, ale na drugie taka odpowiedź jest nie do przyjęcia. Biblioteka standardowa to coś, co musi być dostarczone w każdej implementacji, aby każdy programista mógł z tego korzystać.

Biblioteka standardowa języka C++ zapewnia:

- Wsparcie dla narzędzi językowych, np. do zarządzania pamięcią (11.2), zakresowych pętli `for` (9.5.1) oraz mechanizmów zdobywania informacji o typach w czasie wykonywania (22.2).
- Informacje o zależnych od implementacji aspektach języka, np. jaka jest największa wartość typu `float` (40.2).
- Podstawowe operacje, których nie da się łatwo lub wydajnie zaimplementować w samym języku, np. `is_polymorphic`, `is_scalar` i `is_nothrow_constructible` (35.4.1).
- Narzędzia do niskopoziomowego (bez blokad) programowania współbieżnego (41.3).
- Wsparcie dla programowania współbieżnego przy użyciu wątków (5.3, 42.2).
- Minimalne wsparcie dla programowania współbieżnego przy użyciu zadań, np. `future` i `async()` (42.4).
- Funkcje, których większość programistów nie może łatwo zaimplementować w sposób wydajny i przenośny, np. `uninitialized_fill()` (32.5) i `memmove()` (43.5).
- Minimalne wsparcie w zakresie (opcjonalnego) odzyskiwania nieużywanej pamięci (usuwanie nieużytków), np. `declare_reachable()` (34.5).
- Podstawowe narzędzia zapewniające przenośność programu, np. `list` (31.4), `map` (31.4.3), `sort()` (32.6) oraz strumienie wejścia i wyjścia (rozdział 38.).
- Szkielety umożliwiające rozszerzanie dostępnych narzędzi, np. konwencje i narzędzia pomocnicze umożliwiające posługiwanie się wejściem i wyjściem dla własnych typów przez użytkownika w taki sam sposób jak dla typów wbudowanych (rozdział 38.) i STL (rozdział 31.).

W bibliotece standardowej istnieje też grupa narzędzi, które zostały dodane tylko ze względu na konwencję i ogólną przydatność. Wśród przykładów można wymienić standardowe funkcje matematyczne, np. `sqrt()` (40.3), generatory liczb losowych (40.7), arytmetykę na liczbach zespolonych (40.4) oraz wyrażenia regularne (rozdział 37.).

Biblioteka standardowa stanowi bazę, na której opierają się inne biblioteki. Kombinacje dostępnych w niej narzędzi pozwalają na wykorzystanie jej jako pomocy do osiągnięcia trzech celów.

- Jako bazy w celu zapewnienia przenośności.
- Jako zbioru zwięzłych i wydajnych składników mogących pełnić rolę podstawy budowy bibliotek i aplikacji o wysokich wymaganiach dotyczących wydajności.
- Jako zbioru składników umożliwiających komunikację wewnętrz biblioteki.

Te trzy powiązane ze sobą role miały największy wpływ na kształt projektu biblioteki. Na przykład przenośność jest często jednym z ważnych kryteriów stawianych specjalnym bibliotekom, a powszechnie wykorzystywane kontenery, takie jak listy i słowniki, są niezbędne do wygodnej komunikacji między niezależnie tworzonymi bibliotekami.

Szczególne znaczenie pod względem projektowym ma ostatnia z wymienionych ról, ponieważ pomaga ograniczyć zakres biblioteki standardowej i ogranicza znajdujące się w niej narzędzia. Na przykład narzędzia do pracy z łańcuchami i listami są dostępne w bibliotece standardowej. Gdyby było inaczej, różne biblioteki mogłyby komunikować się tylko przy użyciu typów wbudowanych. Natomiast zaawansowanych narzędzi z zakresu algebry liniowej i przetwarzania grafiki w bibliotece standardowej nie ma. Oczywiście one też są przydatne, ale rzadko przydają się w komunikacji między bibliotekami.

Jeśli narzędzie nie pomaga w pełnieniu jednej z wymienionych ról, to może zostać przeniesione do jakiejś biblioteki poza standardem. To pozwala specjalistom od implementacji bibliotek konkurować na płaszczyźnie jak najlepszej realizacji pomysłów. Gdy biblioteka okaże się przydatna w szerokim zakresie środowisk i dziedzin, zyskuje status kandydatki do zostania biblioteką standardową. Przykładem tego jest biblioteka wyrażeń regularnych (rozdział 37.).

Istnieje też okrojona wersja biblioteki standardowej przeznaczona dla wolnych implementacji, czyli takich, które działają bez wsparcie lub z minimalnym wsparciem ze strony systemu operacyjnego (6.1.1).

30.1.2. Kryteria projektowe

Role biblioteki standardowej określają też pewne ograniczenia co do jej projektu. Dostępne w niej narzędzia spełniają następujące kryteria:

- Są cenne i mogą być używane zarówno przez studentów, jak i zawodowych programistów, wliczając twórców innych bibliotek.
- Są używane pośrednio lub bezpośrednio przez wszystkich programistów do wszystkiego, co mieści się w zakresie biblioteki.
- Są wystarczająco wydajne, aby z powodzeniem zastępować specjalnie napisane funkcje, klasy i szablony w implementacji kolejnych bibliotek.
- Są wolne od strategii lub umożliwiają dostarczanie strategii jako argumentów.
- Są proste w sensie matematycznym, tzn. składnik pełniący dwie słabo powiązane ze sobą role prawie na pewno będzie miał gorszą wydajność niż indywidualne składniki pełniące po jednej roli.
- Są wygodne, wydajne i bezpieczne do użycia w typowych zastosowaniach.

- Są kompletne. Niektórych ważnych funkcji może nie być w bibliotece standardowej, ale jeśli już coś się w niej znajdzie, to musi wykonywać swoje zadanie w całości, tak aby korzystający z biblioteki programiści nie musieli tego zastępować własnymi rozwiązaniami, aby wykonać podstawowe zadania.
- Są łatwe w użyciu z wbudowanymi typami i operacjami.
- Są domyślnie bezpieczne pod względem typów, a więc z zasady dadzą się sprawdzać w czasie wykonywania.
- Umożliwiają programiście posługiwanie się powszechnie stosowanymi stylami programowania.
- Są rozszerzalne, dzięki czemu mogą obsługiwać typy zdefiniowane przez użytkownika w podobny sposób, w jaki obsługiwane są typy wbudowane i należące do biblioteki standardowej.

Na przykład nie można wbudować kryteriów sortowania w funkcję sortującą, ponieważ jeden zbiór danych można posortować na wiele sposobów. Dlatego właśnie funkcja `qsort()` z biblioteki standardowej języka C przyjmuje jako argument funkcję sortującą, zamiast posługiwać się czymś stałym, np. operatorem `<` (12.5). Z drugiej strony, narzut powodowany przez konieczność wywoływanego dodatkowej funkcji w każdej operacji porównywania sprawia, że funkcja `qsort()` jest mało atrakcyjna jako składnik budowy kolejnych bibliotek. Dla prawie każdego typu danych można łatwo zrealizować porównywanie bez korzystania z dodatkowego wywołania funkcji.

Czy ten narzut jest duży? W większości przypadków raczej nie. Niemniej czasami może zdominować czas wykonywania całego algorytmu, co skłoni użytkowników do szukania bardziej wydajnych rozwiązań. Rozwiążanie tego problemu dla funkcji `sort()` i wielu innych funkcji należących do biblioteki standardowej, polegające na użyciu argumentu szablonowego, zostało opisane w podrozdziale 25.2.3. Przykład sortowania ilustruje napięcie występujące między dążeniem do optymalizacji wydajności a zwiększeniem ogólności oraz przedstawia możliwe rozwiązania mające na celu zmniejszenie tego napięcia. Zadaniem biblioteki standardowej jest nie tylko wykonywanie zadań. Dodatkowo musi ona te zadania wykonywać na tyle sprawnie, żeby użytkownicy nie czuli potrzeby szukania alternatywnych rozwiązań. Gdyby było inaczej, implementatorzy zaawansowanych składników pragnący być konkurencyjni musieliby szukać sposobów na obejście biblioteki standardowej. To znacznie utrudniłoby pracę programisty biblioteki i użytkowników chcących zachować niezależność od platformy lub korzystać z kilku różnych bibliotek.

Wymagania dotyczące prostoty i wygody w typowych zastosowaniach są sprzeczne, ponieważ pierwsze wyklucza możliwość optymalizacji biblioteki standardowej pod kątem typowych zastosowań. Ale składniki do zaspokajania typowych, lecz nie prostych, potrzeb mogą zostać dodane do biblioteki obok prostych narzędzi, a nie zamiast nich. Nie możemy pozwolić, aby kult ortogonalności uniemożliwił nam dostarczenie narzędzi wygodnych w użytku także dla początkujących lub okazjonalnych użytkowników. Nie powinien też zmuszać nas do projektowania niejasnych lub niebezpiecznych domyślnych zachowań składników.

30.1.3. Styl opisu

Pełny opis nawet prostej konstrukcji z biblioteki standardowej, np. konstruktora lub algorytmu, może zająć kilka stron. Dlatego zastosowałem bardzo zwięzły styl opisu, w którym zbiory powiązanych ze sobą operacji są przedstawiane w formie tabeli:

Jakieś operacje	
p=op(b,e,x)	op robi coś z przedziałem <b,e> i x oraz zwraca p
foo(x)	foo robi coś z x i nie zwraca wyniku
bar(b,e,x)	Czy x ma coś wspólnego z <b,e>?

Starałem się wybierać łatwe do zapamiętania identyfikatory. Dlatego b i e oznaczają iteratory wyznaczające przedział, p jest wskaźnikiem lub iteratorem, a x jakąś wartością zależną od kontekstu. W tym zapisie brak wyniku od wyniku logicznego można odróżnić tylko dzięki komentarzowi, a więc jeśli dobrze się postarasz, to możesz się pomylić. Objasnienia dotyczące operacji zwracających wartość logiczną zazwyczaj są zakończone znakiem zapytania. Jeśli algorytm zachowuje się w typowy sposób, zwracając koniec sekwencji wejściowej na znak „niepowodzenia”, „nieznalezienia czegoś” itd. (4.5.1, 33.1.1), to nie opisuję tego.

Takim skróconym opisom zazwyczaj towarzyszą odwołania do standardu ISO języka C++, dodatkowe objasnienia oraz przykłady.

30.2. Nagłówki

Wszystko, co znajduje się w bibliotece standardowej, należy do przestrzeni nazw std i jest po-dzielone na nagłówki określające najważniejsze części biblioteki. Dzięki temu patrząc na samą listę nagłówków, można się z grubsza zorientować, co ta biblioteka zawiera.

W dalszej części tego podrozdziału znajduje się wykaz nagłówków pogrupowanych wg przeznaczenia wraz z krótkim objasnieniem i odniesieniami do szczegółowych opisów. Przedstawione grupy odzwierciedlają strukturę organizacyjną standardu.

Standardowy nagłówek o nazwie zaczynającej się od litery c jest równoważny nagłówkowi o tej nazwie w bibliotece standardowej języka C. Dla każdego nagłówka <X.h> zawierającego definicję części biblioteki standardowej języka C w globalnej przestrzeni nazw i w przestrzeni std istnieje nagłówek <cX> zawierający definicje tych samych nazw. Najlepiej by było, gdyby nazwy z nagłówka <cX> nie przenikały do globalnej przestrzeni nazw (15.2.4), ale niestety (z powodu trudności związanych z utrzymaniem wielojęzykowych środowisk zawierających wiele systemów operacyjnych) większość z nich przenika.

Kontenery		
<vector>	Jednowymiarowa tablica o zmiennym rozmiarze	31.4.2
<deque>	Kolejka dwukierunkowa	31.4.2
<forward_list>	Lista jednokierunkowa	31.4.2
<list>	Lista dwukierunkowa	31.4.2
<map>	Tablica asocjacyjna	31.4.3
<set>	Zbiór	31.4.3
<unordered_map>	Mieszająca tablica asocjacyjna	31.4.3.2
<unordered_set>	Mieszający zbiór	31.4.3.2
<queue>	Kolejka	31.5.2
<stack>	Stos	31.5.1
<array>	Jednowymiarowa tablica o niezmiennym rozmiarze	34.2.1
<bitset>	Tablica wartości typu bool	34.2.2

Asocjacyjne kontenery `multimap` i `multiset` znajdują się w nagłówkach `<map>` i `<set>`. Kolejka `priority_queue` (31.5.3) jest zadeklarowana w `<queue>`.

Ogólne narzędzia		
<code><utility></code>	Operatory i pary	35.5, 34.2.4.1
<code><tuple></code>	Krotki	34.2.4.2
<code><type_traits></code>	Cechy typów	35.4.1
<code><typeindex></code>	Używa <code>type_info</code> jako klucza lub kodu mieszania	35.5.4
<code><functional></code>	Obiekty funkcyjne	33.4
<code><memory></code>	Wskaźniki do zarządzania pamięcią	34.3
<code><scoped_allocator></code>	Alokatory zakresowe	34.4.4
<code><ratio></code>	Arytmetyka liczb wymiernych czasu kompilacji	35.3
<code><chrono></code>	Narzędzia czasowe	35.2
<code><ctime></code>	Data i czas w stylu języka C	43.6
<code><iterator></code>	Iteratory i wspomaganie iteratorów	33.1

Iteratory stanowią mechanizm pozwalający na uogólnienie standardowych algorytmów (3.4.2, 33.1.4).

Algorytmy		
<code><algorithm></code>	Ogólne algorytmy	32.2
<code><cstdlib></code>	<code>bsearch()</code> , <code>qsort()</code>	43.7

Typowy ogólny algorytm można zastosować do dowolnej sekwencji (3.4.2, 32.2) elementów dowolnego typu. Funkcje `bsearch()` i `qsort()` z biblioteki standardowej języka C mają zastosowanie do wbudowanych tablic zawierających tylko elementy typów pozbawionych zdefiniowanych przez użytkownika konstruktorów i destruktatorów (12.5).

Diagnostyka		
<code><exception></code>	Klasa wyjątków	30.4.1.1
<code><stdexcept></code>	Standardowe wyjątki	30.4.1.1
<code><cassert></code>	Makro asercji	30.4.2
<code><cerrno></code>	Obsługa błędów w stylu C	13.1.2
<code><system_error></code>	Obsługa błędów systemowych	30.4.3

Asercje wykorzystujące wyjątki są opisane w podrozdziale 13.4.

Łańcuchy i znaki		
<string>	Łańcuch typu T	Rozdział 36.
<cctype>	Klasyfikacja znaków	36.2.1
<cwctype>	Klasyfikacja szerokich znaków	36.2.1
<cstring>	Funkcje łańcuchowe w stylu języka C	43.4
<cwchar>	Funkcje łańcuchowe dla szerokich znaków w stylu języka C	36.2.1
<cstdlib>	Funkcje alokacji w stylu języka C	43.5
<cuchar>	Znaki wielobajtowe w stylu języka C	
<regex>	Dopasowywanie wyrażeń regularnych	Rozdział 37.

Nagłówek <cstring> zawiera deklaracje funkcji z rodziny `strlen()`, `strcpy()` itd. Nagłówek <cstdlib> zawiera deklaracje funkcji `atof()` i `atoi()`, które konwertują łańcuchy w stylu języka C na wartości numeryczne.

Wejście i wyjście		
<iosfwd>	Deklaracje wyprzedzające narzędzi wejścia i wyjścia	38.1
<iostream>	Standardowe obiekty i operacje <code>iostream</code>	38.1
<ios>	Bazy <code>iostream</code>	38.4.4
<streambuf>	Bufory strumieniowe	38.6
<istream>	Szablon strumienia wejściowego	38.4.1
<ostream>	Szablon strumienia wyjściowego	38.4.2
<iomanip>	Manipulatory	38.4.5.2
<sstream>	Strumienie do/z łańcuchów	38.2.2
<cctype>	Funkcje klasyfikacji znaków	36.2.1
<fstream>	Strumienie do/z plików	38.2.1
<cstdio>	Rodzina <code>printf()</code> wejścia i wyjścia	43.3
<cwchar>	Wejście i wyjście szerokich znaków w stylu <code>printf()</code>	43.3

Manipulatory to obiekty służące do manipulowania stanem strumieni (38.4.5.2).

Lokalizacja		
<locale>	Reprezentacja różnic kulturowych	Rozdział 39.
<clocale>	Reprezentacja różnic kulturowych w stylu C	
<codecvt>	Aspekty konwersji kodu	39.4.6

Lokalizacja reprezentuje cechy właściwe różnym kulturom, np. format daty, symbol waluty czy kryteria porównywania łańcuchów, które są różne w różnych językach naturalnych i kulturach.

Wsparcie języka		
<limits>	Limity liczbowe	40.2
<climits>	Makra limitów liczbowych skalarów w stylu języka C	40.2
<cfloat>	Makra limitów liczbowych liczb zmiennoprzecinkowych w stylu języka C	40.2
<cstdint>	Standardowe nazwy typów całkowitoliczbowych	43.7
<new>	Dynamiczne zarządzanie pamięcią	11.2.3
<typeinfo>	Pomoc w uzyskiwaniu informacji o typach w czasie wykonywania	22.5
<exception>	Pomoc w obsłudze wyjątków	30.4.1.1
<initializer_list>	initializer_list	30.3.1
<cstddef>	Wsparcie językowe dla biblioteki C	10.3.1
<cstdarg>	Listy argumentów funkcji o zmiennej długości	12.2.4
<csetjmp>	Odwijanie stosu w stylu języka C	
<cstdlib>	Zamykanie programu	15.4.3
<ctime>	Zegar systemowy	43.6
<csignal>	Obsługa sygnałów w stylu języka C	

Nagłówek <cstddef> definiuje typ wartości zwracanych przez `sizeof()`, `size_t`, wyniku odejmowania wskaźników oraz indeksów tablicowych, `ptrdiff_t` (10.3.1) i niesławnego makra `NULL`.

Odwijanie stosu w stylu języka C (przy użyciu `setjmp` i `longjmp` z nagłówka <csetjmp>) jest niezgodne z używaniem destruktorów i obsługią wyjątków (rozdział 13., 30.4), a więc najlepiej go unikać. W tej książce nie ma opisu odwijania stosu oraz obsługi sygnałów w stylu języka C.

Obliczenia liczbowe		
<complex>	Liczby zespolone i operacje na nich	40.4
<valarray>	Wektory liczbowe i operacje na nich	40.5
<numeric>	Ogólne operacje na liczbach	40.6
<cmath>	Standardowe funkcje matematyczne	40.3
<cstdlib>	Liczby losowe w stylu języka C	40.7
<random>	Generatory liczb losowych	40.7

Z powodów historycznych funkcje `abs()` i `div()` znajdują się w nagłówku <cstdlib> zamiast w <cmath> razem z pozostałymi funkcjami matematycznymi (40.3).

Współbieżność		
<atomic>	Atomowe typy i operacje	41.3
<condition_variable>	Oczekiwanie na zdarzenie	42.3.4
<future>	Asynchroniczne zadanie	42.4.4
<mutex>	Klasy wzajemnego wykluczania	42.3.1
<thread>	Wątki	42.2

Biblioteka standardowa języka C zawiera narzędzia o różnym stopniu przydatności dla programistów języka C. Biblioteka standardowa C++ udostępnia wszystkie te narzędzia:

Zgodność z językiem C		
<code><cinttypes></code>	Aliases najczęściej używanych typów całkowitoliczbowych	43.7
<code><cstdlib></code>	Typ <code>bool</code> języka C	
<code><complex></code>	<code><complex></code>	
<code><cfenv></code>	Środowisko liczb zmiennoprzecinkowych	
<code><cstdalign></code>	Wyrównanie w stylu C	
<code><ctgmath></code>	Uogólnione funkcje matematyczne C: <code><complex></code> i <code><cmath></code>	

Nagłówek `<cstdlib>` nie zawiera definicji makr `bool`, `false` ani `true`. Nagłówek `<cstdalign>` nie zawiera definicji makra `alignas`. Ekwivalenty `.h` nagłówków `<cstdlib>`, `<complex>`, `<calign>` oraz `<ctgmath>` zawierają narzędzia C++ dla C. W razie możliwości należy ich unikać.

Nagłówek `<cfenv>` zawiera typy (np. `fenv_t` i `fexcept_t`), zmiennoprzecinkowe flagi stanu oraz tryby kontroli opisujące środowisko zmiennoprzecinkowe implementacji.

Użytkownik lub implementator biblioteki nie może niczego usuwać ze standardowych nagłówków ani dodawać do nich. Absolutnie nie powinno się też zmieniać zawartości nagłówków poprzez definiowanie makr zmieniających znajdujące się w nich deklaracje (15.2.3). Programy i implementacje zawierające takie sztuczki są niezgodne ze standardem, a poza tym programy takie są nieprzenośne. Nawet jeśli działają dzisiaj, kolejne wydanie nawet części implementacji może sprawić, że przestaną działać. Należy unikać stosowania takich sztuczek.

Aby można było użyć narzędzia z biblioteki standardowej, należy dołączyć do programu odpowiedni nagłówek. Samodzielne wypisanie odpowiednich deklaracji **nie** jest techniką zgodną ze standardem. Powodem tego jest fakt, że niektóre implementacje optymalizują komplikację na podstawie dołączeń standardowych nagłówków, a inne dostarczają zoptymalizowane implementacje narzędzi na podstawie nagłówków. Ogólnie rzecz biorąc, implementatorzy korzystają z nagłówków w nieprzewidywalny sposób i użytkownik nie powinien o tym wiedzieć.

Programista może natomiast specjalizować pomocnicze szablony, np. `swap()` (35.5.2), dla nie należących do biblioteki standardowej typów zdefiniowanych przez użytkownika.

30.3. Wsparcie dla języka

Niewielka, ale ważna część biblioteki standardowej wspiera sam język programowania, tzn. zawiera narzędzia, które są niezbędne do działania programów, bo używają ich konstrukcje językowe.

Wsparcie biblioteki dla konstrukcji języka		
<code><new></code>	<code>new</code> i <code>delete</code>	11.2
<code><typeinfo></code>	<code>typeid()</code> i <code>type_info</code>	22.5
<code><iterator></code>	Zakresowa pętla <code>for</code>	30.3.2
<code><initializer_list></code>	<code>initializer_list</code>	30.3.1

30.3.1. Wsparcie dla list inicjacyjnych

Lista {} jest konwertowana na obiekt typu `std::initializer_list<X>` zgodnie z zasadami opisanymi w podrozdziale 11.3. W nagłówku `<initializer_list>` znajduje się definicja `initializer_list`:

```
template<typename T>
class initializer_list { // iso.18.9
public:
    using value_type = T;
    using reference = const T&; // zwróć uwagę na const: elementy initializer_list są niezmienne
    using const_reference = const T&;
    using size_type = size_t;
    using iterator = const T*;
    using const_iterator = const T*;

    initializer_list() noexcept;

    size_t size() const noexcept; // liczba elementów
    const T* begin() const noexcept; // pierwszy element
    const T* end() const noexcept; // miejsce za ostatnim elementem
};

template<typename T>
const T* begin(initializer_list<T> lst) noexcept { return lst.begin(); }

template<typename T>
const T* end(initializer_list<T> lst) noexcept { return lst.end(); }
```

Niestety `initializer_list` nie ma operatora indeksowania. Jeśli ktoś chce używać operatora [] zamiast *, to może indeksować wskaźnik:

```
void f(initializer_list<int> lst)
{
    for(int i=0; i<lst.size(); ++i)
        cout << lst[i] << '\n'; // błęd

    const int* p = lst.begin();
    for(int i=0; i<lst.size(); ++i)
        cout << p[i] << '\n'; // OK
}
```

Oczywiście list `initializer_list` można używać także w zakresowych pętlach `for`. Na przykład:

```
void f2(initializer_list<int> lst)
{
    for (auto x : lst)
        cout << x << '\n';
}
```

30.3.2. Wsparcie dla zakresowej pętli `for`

Zakresowa instrukcji `for` jest przekształcana w zwykłą instrukcję `for` wykorzystującą iterator zgodnie z opisem przedstawionym w podrozdziale 9.5.1.

W nagłówku `<iterator>` znajdują się funkcje `std::begin()` i `std::end()` dla tablic wbudowanych i wszystkich innych typów udostępniających składowe `begin()` i `end()` (33.3).

Wszystkie kontenery z biblioteki standardowej (np. `vector` i `unordered_map`) i łańcuchy obsługują iterację przy użyciu zakresowej pętli `for`, natomiast adaptery kontenerowe (np. `stack` i `priority_queue`) nie. Nagłówki kontenerów, np. `<vector>`, dołączają nagłówek `<initializer_list>`, a więc rzadko zachodzi potrzeba dołączania go bezpośrednio.

30.4. Obsługa błędów

Biblioteka standardowa powstawała na przestrzeni prawie 40 lat, przez co stosowane w jej składnikach techniki obsługi błędów nie są spójne:

- Biblioteki w stylu języka C zawierają funkcje, z których wiele do oznaczania wystąpienia błędu używa zmiennej `errno` (13.1.2, 40.3).
- Wiele algorytmów operujących na sekwencjach elementów faktycznie nieznalezienia czegoś lub niepowodzenia operacji zgłasza poprzez zwrócenie iteratora wskazującego miejsce za ostatnim elementem (33.1.1).
- Biblioteka strumieni wejścia i wyjścia informuje o błędach, polegając na stanie każdego ze strumieni i może (na żądanie użytkownika) zgłaszać wyjątki (38.3).
- Niektóre składniki biblioteki standardowej, np. `vector`, `string` i `bitset`, informują o błędach poprzez zgłaszanie wyjątków.

Wszystkie znajdujące się w bibliotece standardowej narzędzia przestrzegają „gwarancji podstawowej” (13.2), tzn. nawet w przypadku wystąpienia wyjątku nie dopuszczają do wycieku zasobów (np. pamięci) oraz nie powodują złamania jakiegokolwiek niezmiennika klasy z biblioteki standardowej.

30.4.1. Wyjątki

Niektóre składniki biblioteki standardowej informują o błędach poprzez zgłoszenie wyjątku:

Wyjątki biblioteki standardowej	
<code>bitset</code>	Zgłasza <code>invalid_argument</code> , <code>out_of_range</code> , <code>overflow_error</code>
<code>iostream</code>	Zgłasza <code>ios_base::failure</code> , jeśli wyjątki są wyłączone
<code>regex</code>	Zgłasza <code>regex_error</code>
<code>string</code>	Zgłasza <code>length_error</code> , <code>out_of_range</code>
<code>vector</code>	Zgłasza <code>out_of_range</code>
<code>new T</code>	Zgłasza <code>bad_alloc</code> , jeśli nie może alokować pamięci dla <code>T</code>
<code>dynamic_cast<T>(r)</code>	Zgłasza <code>bad_cast</code> , jeśli nie może przekonwertować referencji na <code>T</code>
<code>typeid()</code>	Zgłasza <code>bad_typeid</code> , jeśli nie może dostarczyć <code>type_info</code>
<code>thread</code>	Zgłasza <code>system_error</code>
<code>call_once()</code>	Zgłasza <code>system_error</code>
<code>mutex</code>	Zgłasza <code>system_error</code>
<code>condition_variable</code>	Zgłasza <code>system_error</code>
<code>async()</code>	Zgłasza <code>system_error</code>
<code>packaged_task</code>	Zgłasza <code>system_error</code>
<code>future</code> i <code>promise</code>	Zgłasza <code>future_error</code>

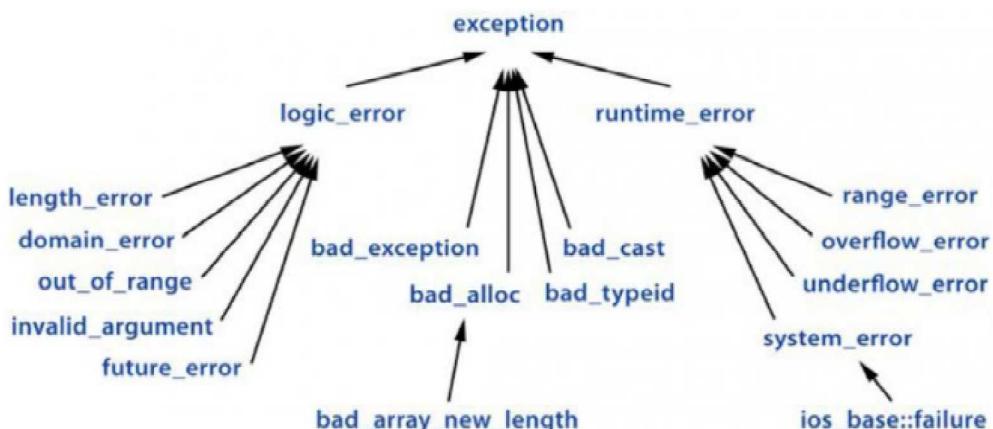
Wymienione rodzaje wyjątków można spotkać w każdym kodzie, który pośrednio lub bezpośrednio posługuje się tymi narzędziami. Ponadto dla każdej operacji mogącej zgłosić wyjątek należy przyjąć, że go zgłosi, chyba że podejmie się kroki temu zapobiegające. Na przykład `packaged_task` zgłosi wyjątek, jeśli zgłosi go funkcja, którą ma wykonać.

Jeśli nie ma się pewności, czy któryś z używanych narzędzi nie zgłosi jakiegoś wyjątku, dobrze jest dodać gdzieś (np. w funkcji `main()` — 13.5.2.3) mechanizm przechwytyujący wyjątki jednej z podstawowych klas hierarchii wyjątków biblioteki standardowej (np. `exception`) oraz wszystkie wyjątki (...).

30.4.1.1. Standardowa hierarchia wyjątków

Nie używaj do zgłoszania typów wbudowanych, np. `int` czy łańcuchów w stylu C, tylko obiektów typów specjalnie zaprojektowanych do reprezentowania wyjątków.

Poniżej przedstawiona jest hierarchia standardowych klas wyjątków:



Hierarchia ta pełni funkcję ogólnej struktury wyjątków, wykraczającej poza możliwości oferowane przez bibliotekę standardową. Błędy logiczne zasadniczo można przechwycić przed uruchomieniem programu lub podczas testowania argumentów funkcji i konstruktorów. Błędy wykonawcze z kolei to już coś całkiem innego. Opis klasy `system_error` znajduje się w podrozdziale 30.4.3.3.

Podstawę hierarchii klas wyjątków w bibliotece standardowej stanowi klasa `exception`:

```

class exception {
public:
    exception();
    exception(const exception&);
    exception& operator=(const exception&);
    virtual ~exception();
    virtual const char* what() const;
};
  
```

Za pomocą funkcji `what()` można uzyskać łańcuch zawierający informacje dotyczące błędu, który spowodował wyjątek.

Programista może zdefiniować wyjątek, wyprowadzając jego klasę z jednej ze standardowych klas:

```

struct My_error : runtime_error {
    My_error(int x) : runtime_error("My_error"), interesting_value{x} { }
    int interesting_value;
};
  
```

Nie wszystkie wyjątki należą do standardowej hierarchii zakorzenionej w klasie `exception`, ale należą do niej wszystkie wyjątki zgłoszane przez bibliotekę standardową.

Jeśli nie wiadomo, czy jakiś składnik programu nie zgłosi wyjątku, dobrym pomysłem jest wstawienie gdzieś mechanizmu przechwytyjącego wszystkie wyjątki. Na przykład:

```
int main()
try{
    //...
}

catch (My_error& me) {      // wystąpił błąd My_error
    // można używać me.interesting_value i me.what()
}

catch (runtime_error& re) { // wystąpił błąd runtime_error
    // można używać re.what()
}

catch (exception& e) {      // wystąpił jakiś wyjątek z biblioteki standardowej
    // można używać e.what()
}

catch (...) {               // wystąpił jakiś nie wymieniony wcześniej wyjątek
    // możemy zrobić porządek w najbliższym otoczeniu
}
```

Jako argumenty do funkcji przekazujemy referencje, aby uniknąć cięcia (17.5.1.4).

30.4.1.2. Propagacja wyjątków

W nagłówku `<exception>` znajdują się narzędzia umożliwiające programistom korzystanie z propagacji wyjątków:

Propagacja wyjątków	
<code>exception_ptr</code>	Nieokreślony typ służący do wskazywania wyjątków
<code>ep=current_exception()</code>	<code>ep</code> jest wskaźnikiem <code>exception_ptr</code> do bieżącego wyjątku lub nie wskazuje żadnego wyjątku, jeśli nie ma aktualnie aktywnego wyjątku; <code>noexcept</code>
<code>rethrow_exception(ep)</code>	Ponownie zgłasza wyjątek wskazywany przez <code>ep</code> ; wskaźnik znajdujący się w <code>ep</code> nie może być <code>nullptr</code> ; <code>noreturn</code> (12.1.7)
<code>ep=make_exception_ptr(e)</code>	<code>ep</code> jest wskaźnikiem <code>exception_ptr</code> do wyjątku <code>e</code> typu <code>exception</code> ; <code>noexcept</code>

Wskaźnik `exception_ptr` może wskazywać dowolny wyjątek, niekoniecznie należący do hierarchii klasy `exception`. Można go traktować jak inteligentny wskaźnik (podobny do `shared_ptr`) utrzymujący swój wyjątek „przy życiu”, dopóki na niego wskazuje. Dzięki temu wskaźnik do wyjątku można przekazać poza funkcję, w której wyjątek ten został przechwycony, i zgłosić go gdzie indziej. Za pomocą wskaźnika `exception_ptr` można ponowić zgłoszenie wyjątku w innym wątku, niż pierwotnie się pojawił. Na tym mechanizmie polegają `promise` i `future` (42.4). Użycie funkcji `rethrow_exception()` na wskaźniku `exception_ptr` (z różnych wątków) nie powoduje wyścigu do danych.

Implementacja funkcji `make_exception_ptr()` może wyglądać następująco:

```
template<typename E>
exception_ptr make_exception_ptr(E e) noexcept;
try {
    throw e;
}
catch(...) {
    return current_exception();
}
```

`nested_exception` to klasa przechowująca wskaźnik `exception_ptr` uzyskany z wywołania funkcji `current_exception()`:

nested_exception (iso.18.8.6)	
<code>nested_exception ne {};</code>	Konstruktor domyślny: ne zawiera wskaźnik <code>exception_ptr</code> do <code>current_exception(); noexcept</code>
<code>nested_exception ne {ne2};</code>	Konstruktor kopiący: ne i ne2 zawierają wskaźnik <code>exception_ptr</code> do zapisanego wyjątku
<code>ne2=ne</code>	Przypisanie kopiące: ne i ne2 zawierają wskaźnik <code>exception_ptr</code> do zapisanego wyjątku
<code>ne.-nested_exception()</code>	Destruktor wirtualny
<code>ne.rethrow_nested()</code>	Ponownie zgłasza wyjątek zapisany w ne; jeśli w ne nie ma wyjątku, wywołuje funkcję <code>terminate(); noreturn</code>
<code>ep=ne.nested_ptr()</code>	ep jest wskaźnikiem <code>exception_ptr</code> wskazującym wyjątek zapisany w ne; noexcept
<code>throw_with_nested(e)</code>	Zgłasza wyjątek typu pochodnego od <code>nested_exception</code> i typu e; e nie może być pochodną <code>nested_exception; noreturn</code>
<code>rethrow_if_nested(e)</code>	Typ <code>dynamic_cast<const nested_exception&>(e).rethrow_nested();</code> musi być pochodną <code>nested_exception</code>

Klasa `nested_exception` ma służyć jako baza dla klasy używanej przez procedurę obsługi wyjątków do przekazywania informacji o lokalnym kontekście błędu wraz ze wskaźnikiem `exception_ptr` do wyjątku, który spowodował jej wywołanie. Na przykład:

```
struct My_error : runtime_error {
    My_error(const string&);
    //...
};

void my_code()
{
    try{
        //...
    }
    catch (...) {
        My_error err {"Coś się nie udało w funkcji my_code()"};
        //...
        throw_with_nested(err);
    }
}
```

Teraz informacje `My_error` są przekazywane (zgłasiane ponownie) wraz z obiektem `nested_exception` zawierającym wskaźnik `exception_ptr` do przechwyconego wyjątku.

W dalszej części łańcucha wywołań możemy obejrzeć zagnieżdżony wyjątek:

```
void user()
{
    try{
        my_code();
    }
    catch(My_error& err) {
        //... rozwiązywanie problemów związanych z My_error...

        try{
            rethrow_if_nested(err); // ponowne zgłoszenie zagnieżdzonego wyjątku, jeśli jest
        }
        catch (Some_error& err2) {
            //... rozwiązywanie problemów związanych z Some_error...
        }
    }
}
```

Założyłem, że wiemy, iż `some_error` może być zagnieżdżony w `My_error`.

Wyjątek nie może wyjść z funkcji `noexcept` (13.5.1.1).

30.4.1.3. `terminate()`

W nagłówku `<exception>` znajdują się narzędzia do radzenia sobie w przypadkach wystąpienia nieprzewidzianych wyjątków:

zamykanie programu (iso.18.8.3, iso.18.8.4)	
<code>h=get_terminate()</code>	<code>h</code> jest bieżącą procedurą obsługi zamknięcia programu; <code>noexcept</code>
<code>h2=set_terminate(h)</code>	<code>h</code> staje się bieżącą procedurą obsługi zamknięcia programu; <code>h2</code> jest poprzednią procedurą obsługi zamknięcia programu; <code>noexcept</code>
<code>terminate()</code>	Zamyka program; <code>noreturn</code> ; <code>noexcept</code>
<code>uncaught_exception()</code>	Czy w bieżącym wątku został zgłoszony wyjątek, którego jeszcze nie przechwycono? <code>noexcept</code>

Należy unikać używania tych funkcji, z wyjątkiem nielicznych przypadków, w których warto skorzystać z `set_terminate()` i `terminate()`. Wywołanie funkcji `terminate()` powoduje zamknięcie programu poprzez wywołanie procedury obsługi zamknięcia programu ustawionej za pomocą funkcji `set_terminate()`. Domyślnym działaniem — które prawie zawsze jest po-prawne — jest natychmiastowe zamknięcie programu. Kwestią wywoływania destruktorów dla lokalnych obiektów po wywołaniu funkcji `terminate()` jest z powodów związanych z działaniem systemów operacyjnych zależna od implementacji. Jeśli funkcja `terminate()` zostaje wywołana w wyniku złamania warunku `noexcept`, system może zastosować (ważne) optymalizacje implikujące, że stos może zostać częściowo rozwinięty (iso.15.5.1).

Niektórzy programiści twierdzą, że funkcja `uncaught_exception()` może służyć do pisania destruktorów, których działanie zależy od tego, czy funkcja zakończy działanie normalnie, czy z powodu wyjątku. Jednak wynikiem funkcji `uncaught_exception()` jest wartość `true` także podczas odwijania stosu (13.5.1) po przechwyceniu pierwotnego wyjątku. Moim zdaniem funkcja `uncaught_exception()` jest zbyt subtelna, aby nadawała się do realnego użytku.

30.4.2. Asercje

W standardzie znajdują się następujące funkcje:

Asercje (iso.7)	
<code>static_assert(e,s)</code>	Wyznacza wartość e w czasie komplikacji; zwraca s jako komunikat o błędzie kompilatora, jeśli !e
<code>assert(e)</code>	Jeśli makro NDEBUG nie jest zdefiniowane, wyznacza wartość e w czasie działania programu, a jeśli !e, to drukuje wiadomość w cerr i wywołuje funkcję <code>abort()</code> ; jeśli makro NDEBUG jest zdefiniowane, funkcja ta nie robi nic

Na przykład:

```
template<typename T>
void draw_all(vector<T*>& v)
{
    static_assert(Is_base_of<Shape,T>(),"Niewłaściwy typ dla draw_all()");

    for (auto p : v) {
        assert(p!=nullptr);
        //...
    }
}
```

Funkcja `assert()` jest makrem znajdującym się w nagłówku `<cassert>`. Zwracana przez nią wiadomość o błędzie jest zdefiniowana przez implementację, ale powinna zawierać nazwę pliku źródłowego (`_FILE_`) i numer wiersza, w którym się znajduje jej wywołanie.

Asercje częściej można spotkać w kodzie produkcyjnym (i słusznie) niż w niewielkich przykładach prezentowanych w podręcznikach do programowania.

Do wiadomości można też dołączyć nazwę funkcji (`_func_`). Założenie, że funkcja `assert()` jest wykonywana, gdy w rzeczywistości jest odwrotnie, może być poważnym błędem. Na przykład przy typowej komplikacji kompilatora `assert(p!=nullptr)` przechwyci błąd podczas diagnostyki, ale nie w finalnym produkcie.

Techniki posługiwania się asercjami są opisane w podrozdziale 13.4.

30.4.3. system_error

W nagłówku `<system_error>` znajdują się narzędzia do raportowania błędów z systemu operacyjnego i niskopoziomowych składników systemu. Można na przykład napisać taką funkcję sprawdzającą nazwę pliku i otwierającą ten plik:

```
ostream& open_file(const string& path)
{
    auto dn = split_into_directory_and_name(path); // dzieli na {ścieżka,nazwa}

    error_code err {does_directory_exist(dn.first)}; // pyta „system” o ścieżkę
    if (err) { //err!=0 means error

        //... sprawdzenie, czy da się coś zrobić...

        if (cannot_handle_err)
            throw system_error(err);
```

```

    }

    //...
    return ofstream(path);
}

```

Przy założeniu, że „system” nie zna wyjątków języka C++, nie mamy wyboru, czy posługiwać się kodami błędów, czy nie. Jedyne pytania to „gdzie?” i „jak?”. W nagłówku `<system_error>` znajdują się narzędzia do klasyfikacji kodów błędów, zamieniania kodów błędów specyficznych dla systemu na bardziej przenośne oraz do zamieniania kodów błędów na wyjątki:

Typy błędów systemowych	
<code>error_code</code>	Zawiera wartość określającą błąd i jego kategorię; specyficzne dla systemu (30.4.3.1)
<code>error_category</code>	Klasa bazowa typów służących do identyfikacji źródła i kodowania określonego rodzaju (kategorii) kodu błędu (30.4.3.2)
<code>system_error</code>	Wyjątek <code>runtime_error</code> zawierający kod błędu (<code>error_code</code>) — 30.4.3.3
<code>error_condition</code>	Zawiera wartość określającą błąd i jego kategorię; możliwa przenośność (30.4.3.4)
<code>errc</code>	Klasa wyliczeniowa (<code>enum class</code>) zawierająca enumeratory reprezentujące kody błędów z nagłówka <code><cerrno></code> (40.3); zasadniczo są to kody błędów POSIX
<code>future_errc</code>	Klasa wyliczeniowa (<code>enum class</code>) zawierająca enumeratory reprezentujące kody błędów z nagłówka <code><future></code> (42.4.4)
<code>io_errc</code>	Klasa wyliczeniowa (<code>enum class</code>) zawierająca enumeratory reprezentujące kody błędów z nagłówka <code><iostream></code> (38.4.4)

30.4.3.1. Kody błędów

Gdy błąd zostaje przekazany „do góry” z niższego poziomu jako kod błędu, to musimy go obsłużyć lub zamienić w wyjątek. Najpierw jednak musimy go jakoś zaklasyfikować: ten sam problem w różnych systemach może mieć inny kod błędu, a ponadto w różnych systemach występują po prostu różne rodzaje błędów.

<code>error_code (iso.19.5.2)</code>	
<code>error_code ec {};</code>	Konstruktor domyślny: <code>ec={0,&generic_category}; noexcept</code>
<code>error_code ec {n,cat};</code>	<code>ec={n,cat}; cat to error_category, a n to wartość typu int reprezentująca błąd znajdujący się w cat; noexcept</code>
<code>error_code ec {n};</code>	<code>ec={n,&generic_category}; n reprezentuje błąd; n jest wartością typu EE, dla której <code>is_error_code_enum<EE>::value==true</code>; noexcept</code>
<code>ec.assign(n,cat)</code>	<code>ec={n,cat}; cat to error_category; n reprezentuje błąd; n jest wartością typu EE, dla którego <code>is_error_code_enum<EE>::value==true</code>; noexcept</code>
<code>ec=n</code>	<code>ec={n,&generic_category}:ec=make_error_code(n); n reprezentuje błąd; n jest wartością typu EE, dla którego <code>is_error_code_enum<EE>::value==true</code>; noexcept</code>
<code>ec.clear()</code>	<code>ec={0,&generic_category()}; noexcept</code>
<code>n=ec.value()</code>	<code>n jest wartością zapisaną w ec; noexcept</code>

error_code (iso.19.5.2) — ciąg dalszy	
<code>cat=ec.category()</code>	cat jest referencją do kategorii zapisanej w ec; noexcept
<code>s=ec.message()</code>	s jest łańcuchem (<code>string</code>) reprezentującym ec potencjalnie użyty jako wiadomość o błędzie: <code>ec.category().message(ec.value())</code>
<code>bool b {ec};</code>	Konwertuje ec na <code>bool</code> ; b wynosi <code>true</code> , jeśli ec reprezentuje błąd, tzn. <code>b==false</code> oznacza „brak błędu”; noexcept
<code>ec==ec2</code>	Zarówno ec, jak i ec2 lub tylko jeden z tych dwóch może być kodem błędu <code>error_code</code> ; aby ec i ec2 zostały uznane za równe, muszą mieć taką samą kategorię (<code>category()</code>) i równoważne wartości (<code>value()</code>); jeśli ec i ec2 są tego samego typu, ekwiwalencję definiuje się poprzez <code>==</code> ; w przeciwnym przypadku ekwiwalencję definiuje się poprzez <code>category().equivalent()</code>
<code>ec!=ec2</code>	<code>!(ec==ec2)</code>
<code>ec<ec2</code>	Porządek <code>ec.category()<ec2.category() (ec.category()==ec2.category() && ec.value()<ec2.value())</code>
<code>e=ec.default_error_condition()</code>	e jest referencją do <code>error_condition</code> : <code>e=ec.category().default_error_condition(ec.value())</code>
<code>os<<ec</code>	Drukuje <code>ec.name()</code> w strumieniu wyjściowym (<code>ostream</code>) os
<code>ec=make_error_code(e)</code>	e to errc; <code>ec=error_code(static_cast<int>(e),&generic_category())</code>

Jak na typ reprezentujący coś tak prostego jak kod błędu `error_code` zawiera dużo składowych. W zasadzie jest to proste mapowanie z liczby całkowitej na wskaźnik do `error_category`.

```
class error_code {
public:
    // reprezentacja: {wartość,kategoria} typu {int,const error_category*}
};
```

Obiekt `error_category` jest interfejsem do obiektu klasy pochodnej klasy `error_category`. W związku z tym `error_category` jest przekazywany przez referencję i przechowywany jako wskaźnik. Każda kategoria błędów jest reprezentowana przez osobny obiekt.

Wróćmy jeszcze raz do przykładu funkcji `open_file()`:

```
ostream& open_file(const string& path)
{
    auto dn = split_into_directory_and_name(path);           // dzieli na {ścieżka,nazwa}

    if (error_code err {does_directory_exist(dn.first)}) { // pyta „system” o ścieżkę
        if (err==errc::permission_denied) {
            //...
        }
        else if (err==errc::not_a_directory) {
            //...
        }
        throw system_error(err); // nie da się nic zrobić lokalnie
    }

    //...
    return ofstream{path};
}
```

Kody błędów errc są opisane w podrozdziale 30.4.3.6. Zwróć uwagę na zastosowanie konstrukcji `if-then-else` zamiast bardziej oczywistej instrukcji `switch`. Powodem tego jest fakt, że operator `==` jest zdefiniowany w kategoriach ekwiwalencji, z uwzględnieniem zarówno kategorii (`category()`), jak i wartości (`value()`) błędu.

Operacje na kodach błędów `error_code` zależą od systemu. W niektórych przypadkach kody te można rzutować na `error_condition` (30.4.3.4) przy użyciu mechanizmów opisanych w podrozdziale 30.4.3.5. Warunek błędu (`error_condition`) wydobywa się z `error_code` przy użyciu funkcji `default_error_condition()`. Obiekt `error_condition` zazwyczaj zawiera mniej informacji niż `error_code`, a więc najlepszym rozwiązaniem jest zachowanie `error_code` w zasięgu i pobieranie z niego `error_condition` tylko w razie potrzeby.

Manipulowanie przy kodach błędów nie powoduje zmiany wartości `errno` (13.1.2, 40.3). Biblioteka standardowa nie zmienia stanów błędów dostarczanych przez inne biblioteki.

30.4.3.2. Kategorie błędów

Klasa `error_category` reprezentuje klasyfikację błędów. Konkretnie błędy są reprezentowane przez klasę wyprowadzoną z klasy `error_category`:

```
class error_category{
public:
    //... interfejs do konkretnych kategorii pochodnych od error_category...
};
```

error_category (iso.19.5.1.1)	
<code>cat.-error_category()</code>	Destruktor wirtualny; <code>noexcept</code>
<code>s=cat.name()</code>	<code>s</code> jest nazwą <code>cat</code> ; <code>s</code> to łańcuch w stylu C; funkcja wirtualna; <code>noexcept</code>
<code>ec=cat.default_error_condition(n)</code>	<code>ec</code> jest <code>error_condition</code> dla <code>n</code> w <code>cat</code> ; funkcja wirtualna; <code>noexcept</code>
<code>cat.equivalent(n,ec)</code>	Czy <code>ec.category() == cat</code> oraz <code>ec.value() == n</code> ? <code>ec</code> jest <code>error_condition</code> ; funkcja wirtualna; <code>noexcept</code>
<code>cat.equivalent(ec,n)</code>	Czy <code>ec.category() == cat</code> oraz <code>ec.value() == n</code> ? <code>ec</code> jest <code>error_code</code> ; funkcja wirtualna; <code>noexcept</code>
<code>s=cat.message(n)</code>	<code>s</code> jest łańcuchem string opisującym błąd <code>n</code> w <code>cat</code> ; funkcja wirtualna
<code>cat==cat2</code>	Czy kategoria <code>cat</code> jest taka sama jak <code>cat2</code> ? <code>noexcept</code>
<code>cat!=cat2</code>	<code>!(cat==cat2); noexcept</code>
<code>cat<cat2</code>	Czy <code>cat < cat2</code> w porządku opartym na adresach <code>error_category</code> : <code>std::less<const error_category*>()(cat, cat2)? noexcept</code>

Jako że klasa `error_category` jest przeznaczona do użytku jako klasa bazowa, nie ma w niej operacji kopiowania ani przenoszenia. Dostęp do egzemplarzy `error_category` można uzyskać poprzez wskaźniki i referencje.

W bibliotece standardowej znajdują się cztery nazwane kategorie:

Kategorie błędów biblioteki standardowej (iso.19.5.1.1)	
ec=generic_category()	ec.name() == "generic"; ec jest referencją do error_category
ec=system_category()	ec.name() == "system"; ec jest referencją do error_category; reprezentuje błędy systemowe; jeśli odpowiada błędowi POSIX, to ec.value() równa się numerowi errno tego błędu
ec=future_category()	ec.name() == "future"; ec jest referencją do error_category; reprezentuje błędy z nagłówka <future>
iostream_category()	ec.name() == "iostream"; ec jest referencją do error_category; reprezentuje błędy z biblioteki iostream

Kategorie te są niezbędne, ponieważ zwykły kod całkowitoliczbowy może w różnych kontekstach (kategoriach) znaczyć co innego. Na przykład 1 oznacza „operacja niedozwolona” (EPERM) w POSIX, jest ogólnym kodem (state) dla wszystkich błędów w iostream oraz oznacza „obiekt future już pobrany” (future_already_retrieved) w future.

30.4.3.3. Wyjątek system_error

Wyjątek system_error służy do zgłoszania błędów dotyczących tych części biblioteki standardej, które współpracują z systemem operacyjnym. Zawiera kod błędu error_code i może zawierać wiadomość o błędzie w formie łańcucha:

```
class system_error : public runtime_error {
public:
    ...
};
```

Klasa wyjątkowa system_error (iso.19.5.6)	
system_error se {ec,s};	se zawiera {ec,s}; ec to error_code; s to string lub łańcuch w stylu C mający być częścią wiadomości o błędzie
system_error se {ec};	se zawiera {ec}, ec jest error_code
system_error se {n,cat,s};	se zawiera {error_code{n,cat},s}; cat jest error_category, a n jest wartością typu int reprezentującą błąd w cat; s jest obiektem typu string lub łańcuchem w stylu C będącym częścią powiadomienia o błędzie
system_error se {n,cat};	se zawiera error_code{n,cat}; cat jest error_category, a n jest wartością typu int reprezentującą błąd w cat
ec=se.code()	ec jest referencją do error_code obiektu se; noexcept
p=se.what()	p jest wersją stylu C łańcucha błędu obiektu se; noexcept

Kod przechwytyujący błąd systemowy system_error ma dostęp do jego kodu error_code. Na przykład:

```
try{
    // coś
}
catch (system_error& err) {
    cout << "Przechwycono system_error " << err.what() << '\n'; // wiadomość o błędzie

    auto ec = err.code();
```

```

cout << "kategoria: " << ec.category().what() << '\n';
cout << "wartość: " << ec.value() << '\n';
cout << "wiadomość: " << ec.message() << '\n';
}

```

Oczywiście błędów systemowych może używać też kod nie należący do biblioteki standar-dowej. Przekazywany jest specyficzny dla systemu `error_code`, nie zaś potencjalnie prze-nośny `error_condition` (30.4.3.4). Aby pobrać `error_condition` z `error_code`, należy użyć funk-cji `default_error_condition()` (30.4.3.1).

30.4.3.4. Potencjalnie przenośne kody błędów

Potencjalnie przenośne kody błędów (`error_condition`) mają prawie identyczną reprezentację jak systemowe kody błędów (`error_code`):

```

class error_condition { // potencjalnie przenośne (iso.19.5.3)
public:
    // jak error_code, ale
    // brak operatora wyjściowego (<<) i
    // operacji default_error_condition()
};

```

Ogólnie chodzi o to, że każdy system ma własny zestaw kodów, które są zamieniane na po-tencjalnie przenośne kody dla wygody programistów piszących programy przeznaczone do użytku na wielu platformach.

30.4.3.5. Zamienianie kodów błędów

Tworzenie kategorii błędów `error_category` ze zbiorem kodów `error_code` należy rozpocząć od zdefiniowania wyliczenia zawierającego potrzebne wartości kodów błędów. Na przykład:

```

enum class future_errc {
    broken_promise = 1,
    future_already_retrieved,
    promise_already_satisfied,
    no_state
};

```

Znaczenie tych wartości całkowicie zależy od kategorii. Wartości całkowitoliczbowe tych enumeratorów są zdefiniowane przez implementację.

Kategoria błędów `future` wchodzi w skład standardu, więc znajduje się w bibliotece stan-dardowej, chociaż w szczegółach może się różnić od tego, co opisuję poniżej.

Następnie należy zdefiniować kategorię dla swoich kodów błędów:

```

class future_cat : error_category{ // zwarcane przez future_category()
public:
    const char* name() const noexcept override { return "future"; }

    string message(int ec) const override;
};

const error_category& future_category() noexcept
{
    static future_cat obj;
    return &obj;
}

```

Rzutowanie wartości całkowitoliczbowych na łańcuchy wiadomości błędów jest żmudne. Trzeba wymyślić zestaw powiadomień, które mogą być pomocne dla programisty. Poniżej przedstawiam niezbyt wyszukane przykłady:

```
string future_error::message(int ec) const
{
    switch (ec) {
    default: return "Kod poważnego błędu future_error";
    future_errc::broken_promise: return "future_error: złamana obietnica";
    future_errc::future_already_retrieved: return "future_error: future został już
        →pobrany";
    future_errc::promise_already_satisfied: return "future_error: obietnica została już
        →spełniona";
    future_errc::no_state: return "future_error: brak stanu";
    }
}
```

Teraz możemy utworzyć `error_code` z `future_errc`:

```
error_code make_error_code(future_errc e) noexcept
{
    return error_code{int(e), future_category()};
}
```

W przypadku konstruktora i przypisania `error_code` pobierających pojedynczą wartość oznaczającą błąd argument powinien być odpowiedniego typu dla `error_category`. Na przykład argument mający być wartością kodu błędu (`error_code`) `future_category()` musi być typu `future_errc`. W szczególności nie można po prostu użyć liczby całkowitej. Na przykład:

```
error_code ec1 {7};                                // błąd
error_code ec2 {future_errc::no_state};           // OK

ec1 = 9;                                         // błąd
ec2 = future_errc::promise_already_satisfied; // OK
ec2 = errc::broken_pipe;                         // błąd: niepoprawna kategoria błędu
```

Aby ułatwić pracę implementatorowi `error_code`, wyspecjalizujmy cechę `is_error_code_enum` dla naszego wyliczenia:

```
template<>
struct is_error_code_enum<future_errc> : public true_type { };
```

W standardzie istnieje już ogólny szablon:

```
template<typename>
struct is_error_code_enum : public false_type { };
```

Z tego wynika, że wszystko, co nie zostanie uznane za kod błędu, nie jest błędem. Aby `error_condition` działało dla naszej kategorii, musimy powtórzyć to, co zrobiliśmy dla `error_code`:

```
template<>
struct is_error_condition_enum<future_errc> : public true_type { };
```

Aby udoskonalić nasz projekt, moglibyśmy użyć osobnego wyliczenia dla `error_condition` i zaimplementować w funkcji `make_error_condition()` odpowiednie rzutowanie z `future_errc`.

30.4.3.6. Kody błędów errc

Standardowe kody błędów dla `system_category()` są zdefiniowane w klasie wyliczeniowej `errc` i mają wartości równoważne POSIX-owej zawartości nagłówka <cerrno>:

enum class errc (iso.19.5)	
address_family_not_supported	EAFNOSUPPORT
address_in_use	EADDRINUSE
address_not_available	EADDRNOTAVAIL
already_connected	EISCONN
argument_list_too_long	E2BIG
argument_out_of_domain	EDOM
bad_address	EFAULT
bad_file_descriptor	EBADF
bad_message	EBADMSG
broken_pipe	EPIPE
connection_aborted	ECONNABORTED
connection_already_in_progress	EALREADY
connection_refused	ECONNREFUSED
connection_reset	ECONNRESET
cross_device_link	EXDEV
destination_address_required	EDESTADDRREQ
device_or_resource_busy	EBUSY
directory_not_empty	ENOTEMPTY
executable_format_error	ENOEXEC
file_exists	EEXIST
file_too_large	EFBIG
filename_too_long	ENAMETOOLONG
function_not_supported	ENOSYS
host_unreachable	EHOSTUNREACH
identifier_removed	EIDRM
illegal_byte_sequence	EILSEQ
inappropriate_io_control_operation	ENOTTY
interrupted	EINTR
invalid_argument	EINVAL
invalid_seek	ESPIPE
io_error	EIO
is_a_directory	EISDIR
message_size	EMSGSIZE
network_down	ENETDOWN
network_reset	ENETRESET
network_unreachable	ENETUNREACH
no_buffer_space	ENOBUFS

enum class errc (iso.19.5) — ciąg dalszy	
no_child_process	ECHILD
no_link	ENOLINK
no_lock_available	ENOLCK
no_message	ENOMSG
no_message_available	ENODATA
no_protocol_option	ENOPROTOOPT
no_space_on_device	ENOSPC
no_stream_resources	ENOSR
no_such_device	ENODEV
no_such_device_or_address	ENXIO
no_such_file_or_directory	ENOENT
no_such_process	ESRCH
not_a_directory	ENOTDIR
not_a_socket	ENOTSOCK
not_a_stream	ENOSTR
not_connected	ENOTCONN
not_enough_memory	ENOMEM
not_supported	ENOTSUP
operation_canceled	ECANCELED
operation_in_progress	EINPROGRESS
operation_not_permitted	EPERM
operation_not_supported	EOPNOTSUPP
operation_would_block	EWOULDBLOCK
owner_dead	EOWNERDEAD
permission_denied	EACCES
protocol_error	EPROTO
protocol_not_supported	EPROTONOSUPPORT
read_only_file_system	EROFS
resource_deadlock_would_occur	EDEADLK
resource_unavailable_try_again	EAGAIN
result_out_of_range	ERANGE
state_not_recoverable	ENOTRECOVERABLE
stream_timeout	ETIME
text_file_busy	ETXTBSY
timed_out	ETIMEDOUT
too_many_files_open	EMFILE
too_many_files_open_in_system	ENFILE
too_many_links	EMLINK
too_many_symbolic_link_levels	ELOOP
value_too_large	EOVERFLOW
wrong_protocol_type	EPROTOTYPE

Kody te dotyczą kategorii systemowej — `system_category()`. W przypadku systemów obsługujących narzędzia POSIX-owe kody te dotyczą także kategorii ogólnej — `generic_category()`.

Makra POSIX są liczbami całkowitymi, natomiast enumeratory `errc` są typu `errc`. Na przykład:

```
void problem(errc e)
{
    if (e==EPIPE) {           // błąd: brak konwersji errc na int
        //...
    }

    if (e==broken_pipe) {     // błąd: broken_pipe poza zakresem
        //...
    }

    if (e==errc::broken_pipe) { // OK
        //...
    }
}
```

30.4.3.7. Kody błędów `future_errc`

Standardowe kody błędów dla kategorii `future_category()` są zdefiniowane w klasie wyliczeniowej `future_errc`:

Enumerator klasy wyliczeniowej <code>future_errc</code> (iso.30.6.1)	
<code>broken_promise</code>	1
<code>future_already_retrieved</code>	2
<code>promise_already_satisfied</code>	3
<code>no_state</code>	4

Kody te dotyczą kategorii `future — future_category()`.

30.4.3.8. Kody błędów `io_errc`

Standardowe kody błędów dla `iostream_category()` są zdefiniowane w klasie wyliczeniowej `io_errc`:

Enumerator klasy wyliczeniowej <code>io_errc</code> (iso.27.5.1)	
<code>string</code>	1

Kod ten dotyczy kategorii `iostream — iostream_category()`.

30.5. Rady

1. Używaj narzędzi z biblioteki standardowej, aby zapewnić przenośność programu — 30.1, 30.1.1.
2. Używaj narzędzi z biblioteki standardowej, aby zminimalizować koszty utrzymania — 30.1.
3. Używaj narzędzi z biblioteki standardowej jako bazy do budowy bardziej rozbudowanych i wyspecjalizowanych bibliotek — 30.1.1.
4. Używaj narzędzi z biblioteki standardowej jako modelowego przykładu elastycznego i powszechnie używanego oprogramowania — 30.1.1.
5. Narzędzia biblioteki standardowej są zdefiniowane w przestrzeni nazw `std` i znajdują się w nagłówkach — 30.2.
6. Standardowy nagłówek biblioteki języka C `X.h` jest w języku C++ reprezentowany jako `<cX>` — 30.2.
7. Nie próbuj używać narzędzi z biblioteki standardowej bez dołączenia odpowiedniego nagłówka za pomocą dyrektywy `#include` — 30.2.
8. Aby skorzystać z zakresowej pętli `for` w odniesieniu do tablicy wbudowanej, należy dołączyć nagłówek `<iterator>` — 30.3.2.
9. Przedkładaj obsługę błędów opartą na wyjątkach nad obsługę opartą na zwracaniu kodów błędów — 30.4.
10. Zawsze przechwytyj `exception&` (w przypadku wyjątków z biblioteki standardowej i dotyczących wsparcia języka) oraz ... (w przypadku nieprzewidzianych wyjątków) — 30.4.1.
11. Hierarchii wyjątków biblioteki standardowej można (ale nie jest to obowiązkowe) używać do tworzenia własnych typów wyjątków — 30.4.1.1.
12. W przypadku poważnych problemów wywołuj funkcję `terminate()` — 30.4.1.3.
13. Często używaj asercji `static_assert()` i `assert()` — 30.4.2.
14. Nie zakładaj, że funkcja `assert()` zawsze zostanie wykonana — 30.4.2.
15. Jeśli nie możesz używać wyjątków, rozważ możliwość użycia `<system_error>` — 30.4.3.

Kontenery STL

*To było nowe.
To było pojedyncze.
To było proste.
To musiało odnieść sukces!*
— H. Nelson

- Wprowadzenie
- Przegląd kontenerów
 - Reprezentacja kontenera; Wymagania dotyczące elementów
- Przegląd operacji
 - Typy składowe; Konstruktory; Destruktor i przypisania; Rozmiar i pojemność; Iteratory;
Dostęp do elementów; Operacje stosowe; Operacje listowe; Inne operacje
- Kontenery
 - Wektor; Listy; Kontenery asocjacyjne
- Adaptacje kontenerów
 - Stos; Kolejka; Kolejka priorytetowa
- Rady

31.1. Wprowadzenie

Biblioteka STL zawiera części biblioteki standardowej obejmujące iteratory, kontenery, algorytmy oraz obiekty funkcyjne. Pozostała część tej biblioteki jest przedstawiona w rozdziałach 32. i 33.

31.2. Przegląd kontenerów

Kontener służy do przechowywania sekwencji obiektów. W tym podrozdziale znajduje się zestawienie dostępnych typów kontenerów oraz krótki opis ich najważniejszych właściwości. Zestawienie operacji, jakie można wykonywać na kontenerach, znajduje się w podrozdziale 31.3.

Kontenery można podzielić na następujące kategorie:

- **Kontenery sekwencyjne** umożliwiające dostęp do (półotwartych) sekwencji elementów.
- **Kontenery asocjacyjne** umożliwiające wyszukiwanie elementów wg kluczy.

Dodatkowo biblioteka standardowa zawiera typy obiektów mogących przechowywać inne obiekty, a które nie mają wszystkich narzędzi dostępnych w kontenerach sekwencyjnych i asocjacyjnych:

- **Adaptery kontenerów** umożliwiają dostęp w specjalny sposób do podstawowych kontenerów.
- **Prawie kontenery** to sekwencje elementów, które udostępniają większość (ale nie wszystkie) operacji udostępnianych przez typowe kontenery.

Wszystkie kontenery dostępne w bibliotece STL (sekwencyjne i asocjacyjne) są uchwytnymi do zasobów udostępniającymi operacje kopiowania i przenoszenia (3.3.1). Wszystkie operacje na kontenerach zapewniają podstawową gwarancję (13.2), dzięki czemu poprawnie współpracują z mechanizmami obsługi błędów opartymi na wykorzystaniu wyjątków.

Kontenery sekwencyjne	
<code>vector<T,A></code>	Sekwencja przylegających elementów typu T; kontener pierwszego wyboru
<code>list<T,A></code>	Dwustronna lista elementów typu T; należy używać, gdy trzeba wstawać i usuwać elementy bez przesuwania pozostałych elementów
<code>forward_list<T,A></code>	Jednostronna lista elementów typu T; idealny kontener do przechowywania pustych i bardzo krótkich sekwencji elementów
<code>deque<T,A></code>	Dwukierunkowa kolejka elementów typu T; jest skrzyżowaniem wektora i listy; w większości przypadków od obu wolniejsza

Argument szablonowy A jest alokatorem wykorzystywanym przez kontener do zajmowania i zwalniania pamięci (13.6.1, 34.4). Na przykład:

```
template<typename T, typename A = allocator<T>>
class vector {
    ...
};
```

A to domyślnie alokator `std::allocator<T>` (34.4.1), który używa operatorów `new()` i `delete()` do zajmowania i zwalniania pamięci dla elementów.

Wymienione kontenery są zdefiniowane w nagłówkach `<vector>`, `<list>` oraz `<deque>`. Elementy w kontenerach sekwencyjnych (np. `vector`) są rozmieszczane w sposób ciągły albo są listami powiązanymi (np. `forward_list`) elementów ich typu wartości `value_type` (T w użytej powyżej notacji). Kontener `deque` (wym. dek) jest połączeniem listy powiązanej i ciągłej alokacji.

Jeśli nie ma poważnych argumentów przeciw, należy używać wektora. Kontener ten udostępnia operacje do wstawiania i usuwania elementów, dzięki którym można go zwiększać i zmniejszać. Jeśli trzeba przechować sekwencję niewielkich elementów w strukturze danych zapewniającej operacje listowe, to wektor jest do tego doskonałym wyborem.

Przy wstawianiu i usuwaniu elementów wektora pozostałe elementy mogą zostać przesunięte. Dla porównania elementy list i kontenerów asocjacyjnych pozostają w miejscu podczas wstawiania i usuwania innych elementów.

Kontener `forward_list` (lista jednokierunkowa) to w istocie lista specjalnie przystosowana do przechowywania pustych i bardzo krótkich sekwencji elementów. Pusta lista `forward_list` zajmuje tylko jedno słowo. Bardzo krótkie i puste listy znajdują zaskakującą dużo zastosowań.

Uporządkowane kontenery asocjacyjne (iso.23.4.2)

C jest typem porównywania. A jest typem alokatora

<code>map<K,V,C,A></code>	Uporządkowany słownik przyporządkowań K – V; sekwencja par (K,V)
<code>multimap<K,V,C,A></code>	Uporządkowany słownik przyporządkowań K – V; dozwolone są duplikaty kluczy
<code>set<K,C,A></code>	Uporządkowany zbiór elementów typu K
<code>multiset<K,C,A></code>	Uporządkowany zbiór elementów typu K; dozwolone są duplikaty kluczy

Te kontenery najczęściej implementuje się jako zrównoważone drzewa binarne (najczęściej czerwono-czarne).

Domyślnym kryterium porządkowania kluczy, K, jest `std::less<K>` (33.4).

Podobnie jak w przypadku kontenerów sekwencyjnych argument szablonowy A jest **alokatorem** używanym przez kontener do zajmowania i zwalniania pamięci (13.6.1, 34.4). Wartością domyślną argumentu szablonowego A jest `std::allocator<std::pair<const K,T>>` (31.4.3) dla słowników i `std::allocator<K>` dla zbiorów.

Nieuporządkowane kontenery asocjacyjne (iso.23.5.2)

H jest typem funkcji mieszającej; E jest testem równości; A jest typem alokatora

<code>unordered_map<K,V,H,E,A></code>	Nieuporządkowany słownik przyporządkowań K – V
<code>unordered_multimap<K,V,H,E,A></code>	Nieuporządkowany słownik przyporządkowań K – V; dozwolone są duplikaty kluczy
<code>unordered_set<K,H,E,A></code>	Nieuporządkowany zbiór elementów typu K
<code>unordered_multiset<K,H,E,A></code>	Nieuporządkowany zbiór elementów typu K; dozwolone są duplikaty kluczy

Kontenery te są zaimplementowane jako tablice mieszające z metodą rozwiązywania kolizji przy użyciu dołączanych list (ang. *linked overflow*). Domyślny typ funkcji mieszającej, H, dla typu K to `std::hash<K>` (31.4.3.2). Domyślny typ funkcji testu równości, E, dla typu K to `std::equal_to<K>` (33.4). Funkcja równości służy do sprawdzania, czy dwa obiekty o takim samym kodzie mieszania są równe.

Kontenery asocjacyjne są strukturami powiązanymi (drzewami) zawierającymi węzły typu określonego przez ich `value_type` (w użyciu wyżej notacji `pair<const K,V>` dla słowników i K dla zbiorów). Sekwencja elementów kontenerów `set`, `map` i `multimap` jest uporządkowana wg wartości kluczy (K). W kontenerach nieuporządkowanych relacja porządkowa elementów (np. <) jest niepotrzebna i zamiast niej używane są funkcje mieszające (31.2.2.1). Sekwencja nieuporządkowanego kontenera nie gwarantuje określonego porządku. Kontener `multimap` różni się od kontenera `map` tym, że każdy klucz może w nim występować wiele razy.

Adaptery kontenerów to kontenery udostępniające specjalne interfejsy do innych kontenerów:

Adaptery kontenerów

C jest typem kontenera

<code>priority_queue<T,C,Cmp></code>	Kolejka priorytetowa obiektów typu T. Cmp jest typem funkcji priorytetowej
<code>queue<T,C></code>	Kolejka elementów typu T udostępniająca operacje <code>push()</code> i <code>pop()</code>
<code>stack<T,C></code>	Stos elementów typu T udostępniający operacje <code>push()</code> i <code>pop()</code>

Domyślną wartością funkcji priorytetowej kolejki `priority_queue`, `Cmp`, jest `std::less<T>`. Domyślnym typem kontenera, `C`, jest `std::deque<T>` dla `queue` oraz `std::vector<T>` dla `stack` i `priority_queue`. Zobacz podrozdział 31.5.

Niektóre typy danych zawierają wiele narzędzi wymaganych od standardowych kontenerów, ale nie mają wszystkich. Typy takie czasami nazywane są „prawie kontenerami”. Oto lista najciekawszych z nich:

„Prawie kontenery”	
<code>T[N]</code>	Wbudowana tablica o stałym rozmiarze: <code>N</code> kolejnych elementów typu <code>T</code> ; brak funkcji <code>size()</code> i innych funkcji składowych
<code>array<T,N></code>	Tablica o stałym rozmiarze <code>N</code> kolejnych elementów typu <code>T</code> ; podobna do tablicy wbudowanej, ale wolna od większości jej wad
<code>basic_string<C,Tr,A></code>	Ciągła sekwencja znaków typu <code>C</code> z operacjami do operowania tekstem, np. konkatenacją (<code>+ i +=</code>); typ <code>basic_string</code> jest zazwyczaj zoptymalizowany w ten sposób, by nie używać wolnej pamięci do alokowania krótkich łańcuchów (19.3.3)
<code>string</code>	<code>basic_string<char></code>
<code>u16string</code>	<code>basic_string<char16_t></code>
<code>u32string</code>	<code>basic_string<char32_t></code>
<code>wstring</code>	<code>basic_string<wchar_t></code>
<code>valarray<T></code>	Numeryczny wektor zawierający operacje wektorowe, ale z ograniczeniami zachęcającymi do tworzenia wysokowydajnych implementacji; używać, tylko jeśli często wykonuje się działania arytmetyczne na wektorach
<code>bitset<N></code>	Zbiór <code>N</code> bitów zawierający operacje zbiorowe, jak <code>&</code> i <code> </code>
<code>vector<bool></code>	Specjalizacja typu <code>vector<T></code> kompaktowo przechowująca bity

W `basic_string` `A` jest alokatorem (34.4), a `Tr` to cechy znaków (36.2.2).

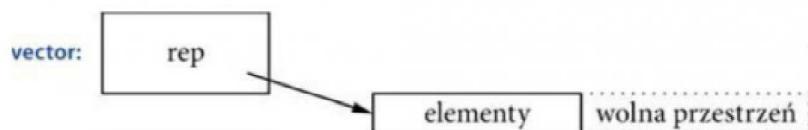
Jeśli ma się wybór, lepiej jest używać kontenerów w rodzaju `vector`, `string` czy `array` niż tablic wbudowanych, których niejawną konwersję na wskaźnik i konieczność zapamiętywania rozmiaru są źródłem wielu błędów (zobacz np. 27.2.1).

Lepiej jest używać standardowych łańcuchów zamiast innych, wliczając łańcuchy w stylu C. Semantyka wskaźników łańcuchów w stylu C zmusza programistę do posługiwanego się niezgrawną notacją i narzuca mu dodatkową pracę, a poza tym stanowi ona źródło wielu błędów (np. wycieków pamięci) — 36.3.1.

31.2.1. Reprezentacja kontenera

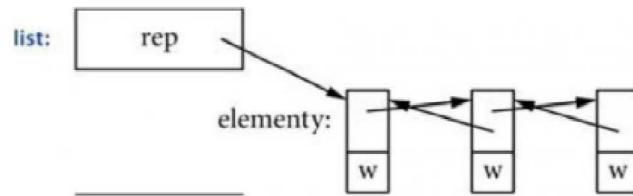
W standardzie nie są narzucone konkretne wymogi dotyczące reprezentacji standardowych kontenerów. Określono jedynie interfejsy kontenerów i pewne kryteria dotyczące poziomu złożoności. Implementatorzy spełniają stawiane wymagania, stosując odpowiednie i często sprytnie zoptymalizowane implementacje. Oprócz narzędzi do manipulowania elementami „uchwyty” takie zawierają także alokator (34.4).

W przypadku wektora jako struktura danych najczęściej stosowana jest tablica:

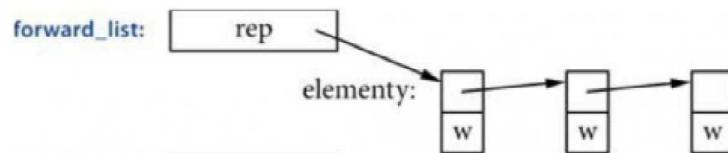


Wektor przechowuje wskaźnik do tablicy elementów, liczbę elementów oraz pojemność (liczbę alokowanych, ale aktualnie nieużywanych miejsc) — 13.6.

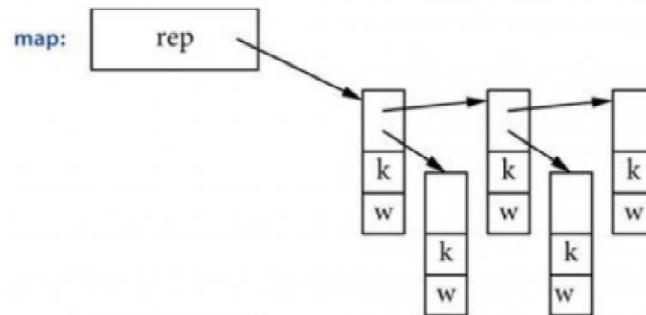
Lista `list` najczęściej jest reprezentowana jako sekwencja łączy wskazujących elementy i liczbę elementów:



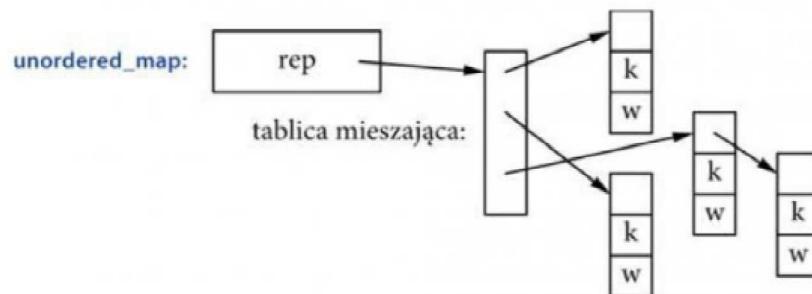
Lista `forward_list` jest zwykle reprezentowana przez sekwencję łączy wskazujących elementy:



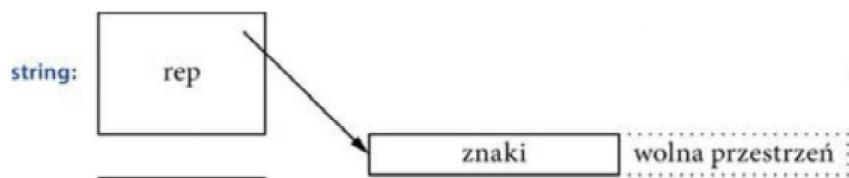
Słownik (`map`) jest najczęściej implementowany jako (zrównoważone) drzewo węzłów wskazujących pary (klucz, wartość):



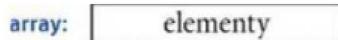
Słownik nieuporządkowany (`unordered_map`) najczęściej implementuje się jako tablicę mieszającą:



Łańcuch `string` może być zaimplementowany tak, jak opisano w podrozdziałach 19.3 i 23.2, tzn. jeśli łańcuch jest krótki, to jego znaki są przechowywane w samym uchwycie `string`, a jeśli jest długi, to zostają przeniesione do ciągłego obszaru w pamięci wolnej (podobnie jak elementy wektora). Obiekt typu `string`, podobnie jak `vector`, może „rosnąć” w wolnym obszarze, który jest alokowany w celu uniknięcia wielokrotnych realokacji:



Struktura array, podobnie jak tablica wbudowana (7.3), jest prostą sekwencją elementów bez uchwytu:



To oznacza, że lokalny obiekt array nie wykorzystuje pamięci wolnej (chyba że zostanie w niej alokowany) oraz że składowa klasy będąca typu array nie implikuje jakichkolwiek operacji na pamięci wolnej.

31.2.2. Wymagania dotyczące elementów

Aby obiekt mógł być elementem kontenera, musi być typu, który implementacja tego kontenera może kopiować i przenosić oraz zamieniać. Jeśli kontener do kopowania elementów używa konstruktora lub przypisania kopującego, wynikiem musi być równoważny obiekt. Z grubsza rzeczą biorąc, oznacza to, że wszelkie testy równości przeprowadzone na wartościach obiektów muszą uznawać kopię za równoznaczną z orginalem. Innymi słowy, kopowanie elementów musi działać podobnie do kopowania wartości typu `int`. Analogicznie konstruktor przenoszący i przypisanie przenoszące muszą mieć konwencjonalne definicje i semantykę przenoszenia (17.5.1). Ponadto musi dać się zamieniać (`swap()`) elementy w typowy sposób. Jeśli typ ma zdefiniowane kopowanie lub przenoszenie, to może być używany przez standardową operację `move()`.

Szczegółowe wymagania dotyczące elementów są rozsiane po całym standardzie i trudne do zrozumienia (iso.23.2.3, iso.23.2.1, iso.17.6.3.2), ale w uproszczeniu można powiedzieć, że kontener może przechowywać elementy typu mającego konwencjonalne operacje kopowania lub przenoszenia. Wiele podstawowych algorytmów, np. `copy()`, `find()` i `sort()`, działa na elementach spełniających warunki bycia elementami kontenera oraz ich specyficzne wymagania (np. elementy muszą być uporządkowane — 31.2.2.1).

Kompilator może wykryć tylko niektóre przypadki złamania warunków dotyczących używania standardowych kontenerów, przez co czasami mogą występować niespodziewane zdarzenia. Na przykład operacja przypisania zgłaszająca wyjątek może pozostawić częściowo skopiowany element. Tak działająca operacja jest źle zaprojektowana (13.6.1) i łamie reguły standardu, zgodnie z którymi powinna zapewniać podstawową gwarancję (13.2). Element pozostający w niepoprawnym stanie może powodować poważne problemy.

Jeśli kopowanie obiektów jest nierozsądne, zamiast obiektów w kontenerze można zapisywać tylko wskaźniki do tych obiektów. Najbardziej oczywistym przykładem są typy polimorficzne (3.2.2, 20.3.2). Można na przykład używać `vector<unique_ptr<Shape>>` lub `vector<Shape*>` zamiast `vector<Shape>`, aby zachować polimorficzność.

31.2.2.1. Porównywanie

Kontenery asocjacyjne, podobnie jak wiele stosowanych na nich operacji (np. `sort()` i `merge()`), wymagają, aby ich elementy dało się sortować. Domyślnie do definiowania porządku elementów używany jest operator `<`. Jeśli operator ten jest nieodpowiedni, programista musi dostarczyć inny sposób porządkowania (31.4.3, 33.4). Kryterium porządkowania musi definiować

ścisłe uporządkowanie słabe (ang. *strict weak ordering*). Nieformalnie można powiedzieć, że zarówno operacja mniejszości, jak i równości muszą być przemienne. To znaczy, że dla kryterium porządkowania `cmp` (traktuj je jak „operację mniejszości”) stawiane są następujące wymagania:

1. Przeciwwrotność: wynik `cmp(x,x)` jest fałszywy.
2. Przeciwsymetria: `cmp(x,y)` implikuje `!cmp(y,x)`.
3. Przechodniość: jeśli `cmp(x,y)` i `cmp(y,z)`, to `cmp(x,z)`.
4. Przechodniość ekwiwalencji: `equiv(x,y)` zdefiniowane jako `!(cmp(x,y) || cmp(y,x))`.
Jeśli `equiv(x,y)` i `equiv(y,z)`, to `equiv(x,z)`.

Ostatnia z wymienionych reguł umożliwia definiowanie `(x==y)` jako `!(cmp(x,y) || cmp(y,x))`, jeśli potrzebny jest operator `==`.

Operacje biblioteki standardowej wymagające porównywania występują w dwóch wersjach. Na przykład:

```
template<typename Ran>
void sort(Ran first, Ran last);           // używa < do porównywania
template<typename Ran, typename Cmp>
void sort(Ran first, Ran last, Cmp cmp); // używa cmp
```

W pierwszej wersji użyto operatora `<`, a w drugiej porównania `cmp` dostarczonego przez użytkownika. Programista może na przykład zechcieć posortować nazwy owoców bez rozróżniania wielkości liter. W tym celu zdefiniuje obiekt funkcyjny (3.4.3, 19.2.2) porównujący pary łańcuchów:

```
class Nocase { // porównywanie łańcuchów bez rozróżniania wielkości liter
public:
    bool operator()(const string&, const string&) const;
};

bool Nocase::operator()(const string& x, const string& y) const
{ // zwraca true, jeśli x jest leksykograficznie mniejszy niż y, nie uwzględniając wielkości liter
    auto p = x.begin();
    auto q = y.begin();

    while (p!=x.end() && q!=y.end() && toupper(*p)==toupper(*q)) {
        ++p;
        ++q;
    }
    if (p == x.end()) return q != y.end();
    if (q == y.end()) return false;
    return toupper(*p) < toupper(*q);
}
```

Przy użyciu tego kryterium porównawczego możemy wywołać funkcję `sort()`. Na przykład mamy dany następujący zbiór elementów:

woce:
jabłko gruszka Jabłko Gruszka cytryna

Sortowanie tego zbioru przy użyciu `sort(fruit.begin(), fruit.end(), Nocase())` da następujący wynik:

```
owoc:
cytryna Gruszka gruszka Jabłko jabłko
```

Dla zestawu znaków, w którym wielkie litery są przed małymi literami, zwykłe sortowanie `sort(fruit.begin(), fruit.end())` dałoby następujący wynik:

```
owoc:
Gruszka Jabłko cytryna gruszka jabłko
```

Nie zapomnij, że operator `<` dla łańcuchów w stylu języka C (tzn. ciągów `const char*`) wykonyuje porównywanie wartości wskaźników (7.4). Przez to jeśli kluczami kontenera są łańcuchy w stylu C, kontener ten działa inaczej, niż spodziewa się większość programistów. Aby zmusić kontenery do poprawnego działania, należy użyć operacji mniejszości wykonującej porównywanie na podstawie porządku leksykograficznego. Na przykład:

```
struct Cstring_less {
    bool operator()(const char* p, const char* q) const { return strcmp(p,q)<0; }
};

map<char*,int,Cstring_less> m; //słownik wykorzystujący strcmp() do porównywania kluczy const char*
```

31.2.2.2. Inne operatory relacyjne

Domyślnie kontenery i algorytmy do porównywania wykorzystują operator `<`. Jeśli operator ten jest nieodpowiedni, programista może dostarczyć własne kryterium porównywania. Nie ma natomiast sposobu na dostarczenie także testu równości. Za to gdy programista dostarcza porównanie `cmp`, równość jest sprawdzana przy użyciu dwóch porównań. Na przykład:

```
if (x == y)           //nie wykonywane, gdy użytkownik dostarczy porównywanie
if (!cmp(x,y) && !cmp(y,x)) //wykonywane, gdy użytkownik dostarczy porównanie cmp
```

Dzięki temu użytkownik nie musi dostarczać testu równości dla każdego typu wykorzystywanego jako typ wartości kontenera asocjacyjnego lub przez algorytm używający porównywania. Może się wydawać, że takie podwójne sprawdzanie jest kosztowne, ale biblioteka nieczęsto wykonuje testy równości. W około 50% przypadków wystarcza jedno wywołanie funkcji `cmp()`, a ponadto kompilator często usuwa podwójne sprawdzanie w procesie optymalizacji.

Użycie relacji równoważności zdefiniowanej przez mniejszość (domyślnie operator `<`) zamiast równości (domyślnie operator `==`) jest także praktyczne. Na przykład kontenery asocjacyjne (31.4.3) porównują klucze przy użyciu testu równoważności `!(cmp(x,y) || cmp(y,x))`. To oznacza, że równoważne klucze nie muszą być równe. Na przykład kontener `multimap` (31.4.3) wykorzystujący jako kryterium porównawcze porównywanie bez rozróżniania wielkości liter łańcuchy `Liśc`, `lisć`, `lIśc`, `lIść` oraz `lisć` uzna za równoważne, mimo że operator `==` dla łańcuchów uznałby je za różne. Dzięki temu można ignorować różnice, które uważamy za nieistotne podczas sortowania.

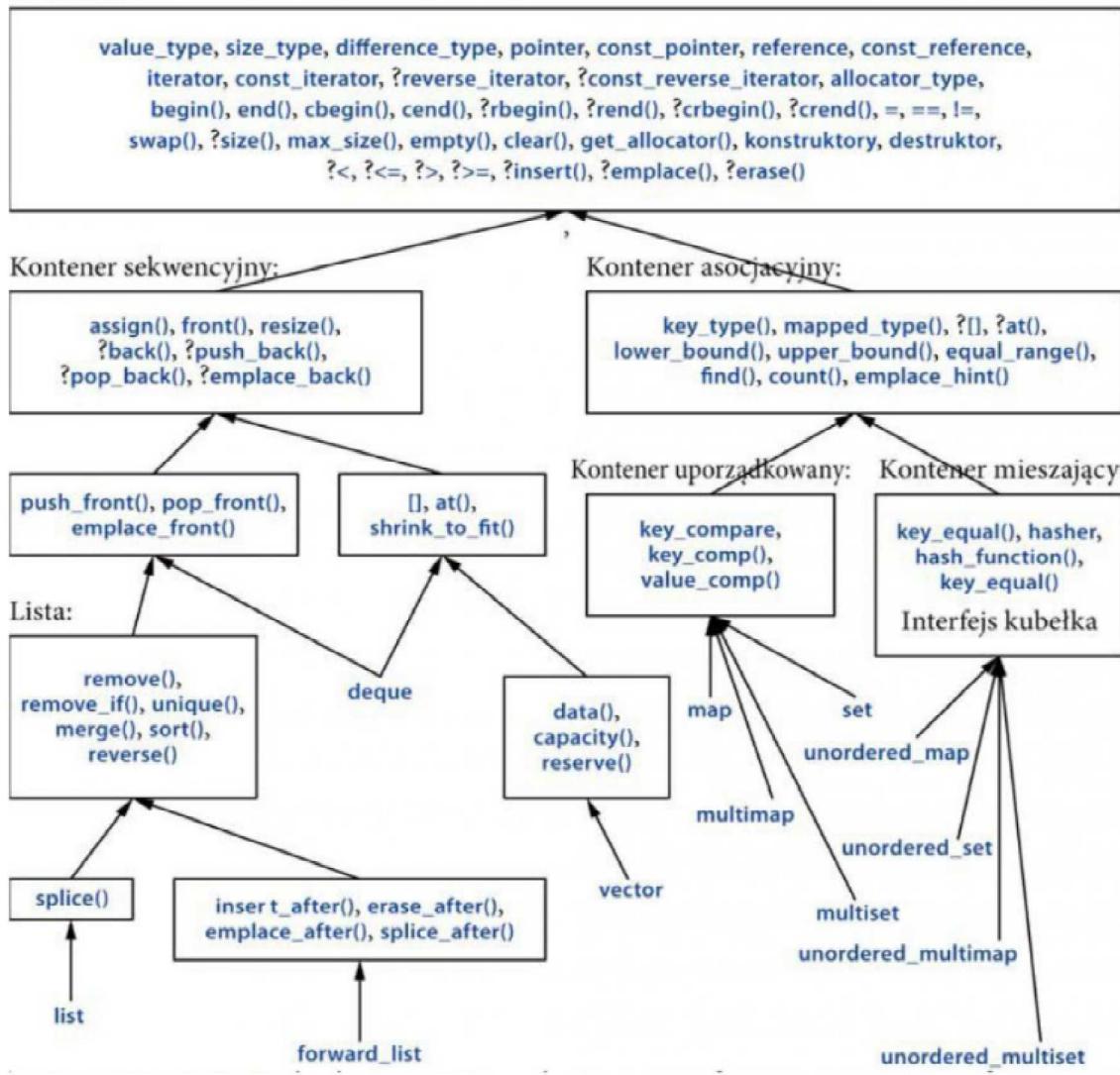
Jeśli wynik testu równości (domyślnie `==`) jest zawsze taki sam jak testu równoważności, `!(cmp(x,y) || cmp(y,x))` (domyślnie `cmp()` to `<`), to mówi się o **porządku totalnym** (ang. *total order*).

Przy użyciu operatorów `<` i `==` łatwo można zbudować pozostałe typowe porównania. W bibliotece standardowej ich definicje znajdują się w przestrzeni nazw `std::rel_ops` i nagłówku `<utility>` (35.5.3).

31.3. Przegląd operacji

Poniżej znajduje się schematyczne zestawienie operacji i typów udostępnianych przez standardowe kontenery:

Kontener:



Strzałka wskazuje, że dla danego kontenera dostarczony jest zbiór operacji, a nie dziedziczenie. Znak zapytania oznacza uproszczenie, tzn. na schemacie ukazane są tylko operacje dostępne jedynie dla niektórych kontenerów, a konkretnie:

- Kontenery asocjacyjne i zbiory multi* nie udostępniają operacji [] i at().
- Kontener forward_list nie udostępnia operacji insert(), erase() ani emplace(). Zamiast tego udostępnia operacje *_after.
- Kontener forward_list nie udostępnia operacji back(), push_back(), pop_back() ani emplace_back().
- Kontener forward_list nie udostępnia reverse_iterator, const_reverse_iterator, rbegin(), rend(), crbegin(), crend() ani size().
- Kontenery asocjacyjne unordered_* nie udostępniają operatorów <, <=, > oraz >=.

Operacje [] i at() zostały powtórzone tylko po to, by zmniejszyć liczbę strzałek.

Opis interfejsu kubelka znajduje się w podrozdziale 31.4.3.2.

W uzasadnionych przypadkach operacja dostępu występuje w dwóch wersjach: po jednej dla obiektów const i pozostałych.

Znajdujące się w bibliotece standardowej operacje zapewniają gwarancje złożoności:

Złożoność standardowych operacji kontenerowych					
	□ 31.2.2	Lista 31.3.7	Przód 31.4.2	Tyl 31.3.6	Iteratory 33.1.2
vector	const	$O(n) +$		const+	Swobodne
list		const	const	const	Dwukierunkowe
forward_list		const	const		Jednokierunkowe
deque	const	$O(n)$	const	const	Swobodne
stack				const	
queue			const	const	
priority_queue			$O(\log(n))$	$O(\log(n))$	
map		$O(\log(n))$	$O(\log(n)) +$		Dwukierunkowe
multimap			$O(\log(n)) +$		Dwukierunkowe
set			$O(\log(n)) +$		Dwukierunkowe
multiset			$O(\log(n)) +$		Dwukierunkowe
unordered_map	const+	const+			Jednokierunkowe
unordered_multimap		const+			Jednokierunkowe
unordered_set		const+			Jednokierunkowe
unordered_multiset		const+			Jednokierunkowe
string	const	$O(n) +$	$O(n) +$	const+	Swobodne
array	const				Swobodne
tablica wbudowana	const				Swobodne
valarray	const				Swobodne
bitset	const				

Operacje typu „Przód” dotyczą wstawiania i usuwania elementów przed pierwszym elementem, a operacje typu „Tyl” analogicznie dotyczą wstawiania i usuwania za ostatnim elementem. Operacje „Lista” dotyczą wstawiania i usuwania elementów niekoniecznie na końcach kontenera.

W kolumnie „Iteratory” zostały wymienione rodzaje iteratorów: o dostępie swobodnym, jednokierunkowe oraz dwukierunkowe (33.1.4).

Wpisy znajdujące się w pozostałych komórkach określają wydajność operacji. Wpis const oznacza, że czas wykonywania operacji nie zależy od liczby znajdującej się w kontenerze elementów. Standardowo **stały czas wykonywania** oznacza się $O(1)$. $O(n)$ oznacza, że czas wykonywania operacji jest proporcjonalny do liczby elementów. Znak + oznacza, że czasami

występuje znaczący dodatkowy koszt. Na przykład wstawianie elementu do listy to operacja o stałej złożoności (dlatego została oznaczona jako `const`), natomiast ta sama operacja w przypadku wektora wymaga przesunięcia wszystkich elementów znajdujących się za miejscem, w którym wstawiany jest nowy element (dlatego została oznaczona $O(n)$). Ponadto czasami konieczne jest realokowanie wszystkich elementów wektora i dlatego dodałem znak `+`. Notacja „z wielkim O” to standardowy sposób zapisu. Znak `+` dodałem dla wygody programistów, którzy oprócz informacji o średniej wydajności potrzebują też danych o przewidywalności. Konwencjonalna nazwa złożoności $O(n) +$ to **zamortyzowany koszt liniowy** (ang. *amortized linear time*).

Oczywiście jeśli stały koszt jest duży, to może znacznie przewyższać niewielki koszt proporcjonalny do liczby elementów. Ale generalnie w przypadku dużych struktur danych stały koszt oznacza „niewielko koszt”, $O(n)$ znaczy „duży koszt”, a $O(\log(n))$ znaczy „w miarę niewielki koszt”. Nawet dla umiarkowanych wartości n złożoność $O(\log(n))$, gdzie \log jest logarymem binarnym, jest znacznie bliższa stałej niż $O(n)$. Na przykład:

Przykłady złożoności logarytmicznej						
n	16	128	1024	16 384	1 048 576	
$\log(n)$	4	7	10	14	20	
$n \cdot n$	256	802 816	1 048 576	268 435 456	1,1e+12	

Programiści, którzy cenią wydajność, powinni dokładnie przyjrzeć się tej tabeli. W szczególności muszą wiedzieć, przez jakie elementy trzeba przejść, by dostać się do elementu n . Ale wiadomość jest jasna: nie paprać się z kwadratowymi algorytmami przy dużych wartościach n .

Pomiary złożoności i kosztów wykonywania wskazują górne granice i stanowią wskazówkę dla użytkownika, czego można się spodziewać po implementacjach. Oczywiście w ważnych przypadkach implementatorzy starają się zapewnić jak najlepsze wyniki.

Pomiary złożoności „dużego O” są asymptotyczne, tzn. aby różnica w wydajności była znacząca, konieczne mogłyby być użycie dużej liczby elementów. Dominować mogą inne czynniki, np. koszt operacji na pojedynczym elemencie. Na przykład złożoność obliczeniowa przeglądania wektora i listy wynosi $O(n)$. Mimo to we współczesnych architekturach sprzętu dostęp do następnego elementu poprzez dowiązanie (w liście) może być o wiele kosztowniejszy od dostępu do następnego elementu wektora (którego elementy są alokowane przylegając). Analogicznie algorytm liniowy może wymagać znacznie więcej lub znacznie mniej niż dziesięciokrotność czasu dla dziesięciokrotnie większej liczby elementów, ponieważ duże znaczenie mają też architektura pamięci i procesora. W kwestiach kosztów i złożoności nie należy polegać tylko na intuicji. Trzeba dodatkowo wykonywać pomiary. Na szczęście interfejsy kontenerów są tak do siebie podobne, że porównania łatwo jest zakodować.

Operacja `size()` jest stała czasowo dla wszystkich operacji. Zauważ, że kontener `forward_list` nie ma operacji `size()`, w związku z czym aby poznać liczbę jego elementów, należy je policzyć samodzielnie (koszt takiej operacji wynosi $O(n)$). Lista `forward_list` jest zoptymalizowana pod kątem zajmowania przestrzeni i nie przechowuje informacji o swoim rozmiarze ani wskaźnika do swojego ostatniego elementu.

Szacunkowe wartości dotyczące kontenera `string` dotyczą dłuższych łańcuchów. „Optymalizacja krótkich łańcuchów” (19.3.3) sprawia, że złożoność czasowa wszystkich operacji na krótkich łańcuchach (np. krótszych niż 14 znaków) jest stała.

Wartości podane dla kontenerów `stack` i `queue` odzwierciedlają koszt właściwy domyślnej implementacji, która jest zbudowana na bazie kontenera `deque` (31.5.1, 31.5.2).

31.3.1. Typy składowe

Każdy kontener zawiera zestaw definicji typów składowych:

Typy składowe (iso.23.2, iso.23.3.6.1)	
<code>value_type</code>	Typ elementu
<code>allocator_type</code>	Typ menedżera pamięci
<code>size_type</code>	Bezznakowy typ indeksów kontenera, liczb elementów itp.
<code>difference_type</code>	Typ ze znakiem różnic między iteratorami
<code>iterator</code>	Działa jak <code>value_type*</code>
<code>const_iterator</code>	Działa jak <code>const_value_type*</code>
<code>reverse_iterator</code>	Działa jak <code>value_type*</code>
<code>const_reverse_iterator</code>	Działa jak <code>const_value_type*</code>
<code>reference</code>	<code>value_type&</code>
<code>const_reference</code>	<code>const_value_type&</code>
<code>pointer</code>	Działa jak <code>value_type*</code>
<code>const_pointer</code>	Działa jak <code>const_value_type*</code>
<code>key_type</code>	Typ klucza; dotyczy tylko kontenerów asocjacyjnych
<code>mapped_type</code>	Typ mapowanej wartości; dotyczy tylko kontenerów asocjacyjnych
<code>key_compare</code>	Typ kryterium porównawczego; dotyczy tylko kontenerów uporządkowanych
<code>hasher</code>	Typ funkcji mieszającej; dotyczy tylko kontenerów nieuporządkowanych
<code>key_equal</code>	Typ funkcji sprawdzającej równoważność; dotyczy tylko kontenerów nieuporządkowanych
<code>local_iterator</code>	Typ iteratora kubelkowego; dotyczy tylko kontenerów nieuporządkowanych
<code>const_local_iterator</code>	Typ iteratora kubelkowego; dotyczy tylko kontenerów nieuporządkowanych

Każdy kontener i „prawie kontener” zawiera większość tych typów składowych. Brakuje tylko tych, które w danym przypadku nie mają sensu. Na przykład kontener `array` nie zawiera typu `allocator_type`, a `vector` nie zawiera `key_type`.

31.3.2. Konstruktory, destruktory i przypisania

Kontenery udostępniają różne konstruktory i operacje przypisania. Dla kontenera o nazwie `C` (np. `vector<double>` lub `map<string, int>`) mamy:

Konstruktory, destruktory i przypisania	
C jest kontenerem; domyślnie C używa domyślnego alokatora C::allocator_type{}	
C c {}	Konstruktor domyślny: c jest pustym kontenerem
C c {a}	Domyślna konstrukcja c; użycie alokatora a
C c(n)	Inicjacja c n elementami o wartości <code>value_type{}</code> ; nie dotyczy kontenerów asocjacyjnych
C c(n,x)	Inicjacja c n kopiami x; nie dotyczy kontenerów asocjacyjnych
C c(n,x,a)	Inicjacja c n kopiami x; użycie alokatora a; nie dotyczy kontenerów asocjacyjnych
C c {elem};	Inicjacja c przy użyciu elem; jeśli C ma konstruktor z listą inicjacyjną, to jest on preferowany; w przeciwnym przypadku zostaje użyty inny konstruktor
C c {c2};	Konstruktor kopiujący: kopiuje elementy c2 i alokator do c
C c {move(c2)};	Konstruktor przenoszący: przenosi elementy c2 i alokator do c
C c {{elem},a};	Inicjuje c przy użyciu <code>initializer_list{elem}</code> ; używa alokatora a
C c {b,e};	Inicjuje c elementami z przedziału <b,e>
C c {b,e,a};	Inicjuje c elementami z przedziału <b,e>; używa alokatora a
c.-c()	Destruktor: usuwa elementy c i zwalnia wszystkie zasoby
c2=c	Przypisanie kopiujące: kopiuje elementy c do c2
c2=move(c)	Przypisanie przenoszące: przenosi elementy c do c2
c={elem}	Przypisuje do c z <code>initializer_list{elem}</code>
c.assign(n,x)	Przypisuje n kopii x; nie dotyczy kontenerów asocjacyjnych
c.assign(b,e)	Przypisuje do c z <b,e>
c.assign({elem})	Przypisuje do c z <code>initializer_list{elem}</code>

Więcej konstruktorów kontenerów asocjacyjnych opisałem w podrozdziale 31.4.3.

Zauważ, że przypisanie nie kopiuje ani nie przenosi alokatorów. Kontener docelowy otrzymuje nowy zbiór elementów, ale zachowuje swój stary kontener, przy użyciu którego alokuje miejsca dla ewentualnych nowych elementów. Opis alokatorów znajduje się w podrozdziale 34.4.

Pamiętaj, że i konstruktor, i operacja kopiowania elementów może zgłosić wyjątek, aby poinformować o niemożliwości wykonania zadania.

Potencjalne niejednoznaczności związane z inicjatorami są opisane w podrozdziałach 11.3.3 i 17.3.4.1. Na przykład:

```
void use()
{
    vector<int> vi {1,3,5,7,9}; // wektor inicjowany pięcioma liczbami całkowitymi
    vector<string> vs(7);      // wektor inicjowany siedmioma pustymiłańcuchami

    vector<int> vi2;
    vi2 = {2,4,6,8};           // przypisanie sekwencji czterech liczb całkowitych do vi2
    vi2.assign(&vi[1],&vi[4]); // przypisanie sekwencji 3,5,7 do vi2

    vector<string> vs2;
    vs2 = {"Małpka", "Małpka i żabka"}; // przypisanie dwóchłańcuchów do vs2
    vs2.assign("Niedźwiedź", "Stary niedźwiedź mocno śpi"); // błąd wykonawczy
}
```

Błąd w przypisaniu do `vs2` polega na tym, że przekazywana jest para wskaźników (nie lista inicjalizacyjna), które nie wskazują do tej samej tablicy. Używaj operatora `()` dla inicjatorów rozmiarowych i `{}` dla pozostałych rodzajów iteratorów.

Kontenery często są bardzo duże, przez co prawie zawsze przekazuje się je przez referencję. Ponieważ jednak są one uchwytemi do zasobów (31.2.1), ich zwracanie (przy niejawnym użyciu przenoszenia) nie pochłania dużo zasobów. Analogicznie można je przenosić jako argumenty, gdy nie chce się aliasów. Na przykład:

```
void task(vector<int>&& v);

vector<int> user(vector<int>& large)
{
    vector<int> res;
    ...
    task(move(large)); // przekazuje prawa własności do danych do task()
    ...
    return res;
}
```

31.3.3. Rozmiar i pojemność

Rozmiar to liczba znajdujących się w kontenerze elementów. Pojemność to liczba elementów, jaką kontener może przechowywać bez alokowania dodatkowej pamięci:

Rozmiar i pojemność	
<code>x=c.size()</code>	<code>x</code> jest liczbą elementów <code>c</code>
<code>c.empty()</code>	Czy <code>c</code> jest pusty?
<code>x=c.max_size()</code>	<code>x</code> jest największą możliwą liczbą elementów <code>c</code>
<code>x=c.capacity()</code>	<code>x</code> jest przestrzenią alokowaną dla <code>c</code> ; dotyczy tylko kontenerów <code>string</code> i <code>vector</code>
<code>c.reserve(n)</code>	Rezerwuje przestrzeń dla <code>n</code> elementów kontenera <code>c</code> ; dotyczy tylko kontenerów <code>string</code> i <code>vector</code>
<code>c.resize(n)</code>	Zmienia rozmiar <code>c</code> na <code>n</code> ; dodawanym elementom nadaje wartość domyślną; dotyczy tylko kontenerów sekwencyjnych (i <code>string</code>)
<code>c.resize(n,v)</code>	Zmienia rozmiar <code>x</code> na <code>n</code> ; dodawanym elementom nadaje wartość <code>v</code> ; dotyczy tylko kontenerów sekwencyjnych (i <code>string</code>)
<code>c.shrink_to_fit()</code>	Sprawia, że wartość <code>c.capacity()</code> jest równa <code>c.size()</code> ; dotyczy tylko kontenerów <code>vector</code> , <code>deque</code> oraz <code>string</code>
<code>c.clear()</code>	Kasuje wszystkie elementy z kontenera <code>c</code>

Podczas zmiany rozmiaru lub pojemności elementy mogą zostać przeniesione w inne miejsce. To oznacza, że iteratory (i wskaźniki oraz referencje) do elementów mogą stać się bezwartościowe (tzn. wskazywać stare miejsca przechowywania elementów). Przykład przedstawiłem w podrozdziale 31.4.1.1.

Iterator do elementu kontenera asocjacyjnego (np. `map`) staje się nieważny, tylko gdy wskazywany przez niego element zostanie usunięty z kontenera (`erase()` — 31.3.7). Dla porównania: iterator do elementu kontenera sekwencyjnego (np. wektora) traci ważność tylko po realokacji

elementów (np. przez operacje `resize()`, `reserve()` lub `push_back()`) lub gdy wskazywany przez niego element zostanie przeniesiony w obrębie kontenera (np. przez operacje `erase()` lub `insert()` wykonane na elemencie o niższym indeksie).

Przyjęcie założenia, że `reserve()` poprawia wydajność, jest kuszące, ale standardowe strategie zwiększania rozmiaru wektora (31.4.1.1) są na tyle efektywne, że zysk wydajności rzadko jest wystarczająco dobrym powodem, aby użyć tej funkcji. Lepiej jest jej używać w celu zwiększenia przewidywalności wydajności oraz uniknięcia uszkodzenia iteratorów.

31.3.4. Iteratory

Kontener można traktować jak sekwencję elementów ułożonych w porządku zdefiniowanym przez iterator kontenera lub odwrotnym. W przypadku kontenerów asocjacyjnych porządek elementów jest ustalany na podstawie kryterium porównawczego kontenera (domyślnie `<`).

Iteratory	
<code>p=c.begin()</code>	p wskazuje pierwszy element c
<code>p=c.end()</code>	p wskazuje element za ostatnim elementem c
<code>cp=c.cbegin()</code>	p wskazuje stały pierwszy element c
<code>p=c.cend()</code>	p wskazuje stały pierwszy element za ostatnim elementem c
<code>p=c.rbegin()</code>	p wskazuje pierwszy element odwrotnej sekwencji c
<code>p=c.rend()</code>	p wskazuje element za ostatnim elementem odwrotnej sekwencji c
<code>p=c.crbegin()</code>	p wskazuje stały pierwszy element odwrotnej sekwencji c
<code>p=c.crend()</code>	p wskazuje stały element za ostatnim elementem odwrotnej sekwencji c

Najczęściej kontenery przegląda się od początku do końca. Najprostszym sposobem na zrobienie tego jest użycie zakresowej pętli `for` (9.5.1), która niejawnie wykorzystuje `begin()` i `end()`. Na przykład:

```
for (auto& x : v) // niejawne użycie v.begin() i v.end()
    cout << x << '\n';
```

Jeśli potrzebna jest informacja o pozycji elementu w kontenerze albo chcemy odwoływać się do więcej niż jednego elementu naraz, konieczne jest bezpośrednie użycie iteratora. W takim przypadkach przydaje się słowo kluczowe `auto`, które pozwala zredukować ilość kodu źródłowego i wyeliminować wiele okazji do popełnienia literówki. Poniżej znajduje się przykład z założeniem, że używany jest iterator o dostępie swobodnym:

```
for (auto p = v.begin(); p!=v.end(); ++p) {
    if (p!=v.begin() &&*(p-1)==*p)
        cout << "powielenie " << *p << '\n';
```

Jeśli nie trzeba modyfikować elementów, można używać iteratorów `cbegin()` i `cend()`. Innymi słowy, powiniensem był napisać:

```
for (auto p = v.cbegin(); p!=v.cend(); ++p) { // użycie iteratorów stałych
    if (p!=v.cbegin() &&*(p-1)==*p)
        cout << "powielenie " << *p << '\n';
}
```

W większości kontenerów i implementacji wielokrotne użycie iteratorów `begin()` i `end()` nie sprawia problemów wydajnościowych, w związku z czym dalem sobie spokój z komplikowaniem kodu w następujący sposób:

```
auto beg = v.cbegin();
auto end = v.cend();

for (auto p = beg; p!=end; ++p) {
    if (p!=beg &&*(p-1)==*p)
        cout << "powielenie " << *p << '\n';
}
```

31.3.5. Dostęp do elementów

Do niektórych elementów można uzyskać bezpośredni dostęp:

Dostęp do elementów	
<code>c.front()</code>	Odwołanie do pierwszego elementu <code>c</code> ; nie dotyczy kontenerów asocjacyjnych
<code>c.back()</code>	Odwołanie do ostatniego elementu; nie dotyczy kontenera <code>forward_list</code> i kontenerów asocjacyjnych
<code>c[i]</code>	Odwołanie do <code>i</code> -tego elementu <code>c</code> ; dostęp nie jest sprawdzany; nie dotyczy list i kontenerów asocjacyjnych
<code>c.at(i)</code>	Odwołanie do <code>i</code> -tego elementu <code>c</code> ; zgłasza wyjątek <code>out_of_range</code> , jeśli <code>i</code> jest poza zakresem; nie dotyczy list i kontenerów asocjacyjnych
<code>c[k]</code>	Odwołanie do elementu o kluczu <code>k</code> w <code>c</code> ; w przypadku nieznalezienia wstawia <code>(k,mapped_type{})</code> ; dotyczy tylko kontenerów <code>map</code> i <code>unordered_map</code>
<code>c.at(k)</code>	Odwołanie do elementu <code>i</code> klucza <code>k</code> w <code>c</code> ; w przypadku nieznalezienia <code>k</code> zgłasza wyjątek <code>out_of_range</code> ; dotyczy tylko kontenerów <code>map</code> i <code>unordered_map</code>

Niektóre implementacje — zwłaszcza wersje diagnostyczne — zawsze wykonują sprawdzanie zakresu, ale nie można zakładać, że test ten zawsze będzie wykonywany ani że dzięki jego brakowi zyska się na wydajności. Jeśli kwestie te są ważne, należy dokładnie sprawdzić implementację.

Kontenery asocjacyjne `map` i `unordered_map` mają operacje `[]` i `at()`, które przyjmują jako argumenty klucze, nie zaś pozycje (31.4.3).

31.3.6. Operacje stosowe

Standardowe kontenery `vector`, `deque` oraz `list` (ale nie `forward_list` ani kontenery asocjacyjne) udostępniają wydajne operacje wykonywane na końcu swoich sekwencji elementów:

Operacje stosowe	
<code>c.push_back(x)</code>	Dodaje <code>x</code> do <code>c</code> (przy użyciu kopowania lub przenoszenia), za ostatnim elementem
<code>c.pop_back()</code>	Usuwa ostatni element z <code>c</code>
<code>c.emplace_back(args)</code>	Dodaje obiekt utworzony z <code>args</code> do <code>c</code> , za ostatnim elementem

Operacja `c.push_back(x)` przenosi lub kopiuje `x` do `c` oraz zwiększa rozmiar `c` o jeden. Jeśli zabraknie pamięci lub konstruktor kopiący obiektu `x` zgłosi wyjątek, wykonanie operacji `c.push_back(x)` nie powiedzie się. Awaria następująca w czasie działania funkcji `push_back()` nie ma wpływu na kontener dzięki zapewnionej silnej gwarancji (13.2).

Zauważ, że operacja `pop_back()` nie zwraca wartości. Gdyby to robiła, konstruktor kopiący zgłaszający wyjątki mógłby bardzo skomplikować implementację.

Ponadto kontenery `list` i `deque` udostępniają równoważne operacje dla początku swoich sekwencji elementów (31.4.2). Dotyczy to też kontenera `forward_list`.

Operacja `push_back` jest od zawsze ulubioną funkcją programistów używaną w przypadkach, gdy trzeba zwiększyć kontener bez preallokacji lub ryzyka wystąpienia przepelnienia, ale w podobny sposób można używać też `emplace_back()`. Na przykład:

```
vector<complex<double>> vc;
for (double re,im; cin>>re>>im; ) // wczytuje dwie liczby typu double
    vc.emplace_back(re,im);           // dodaje complex<double> {re,im} na końcu
```

31.3.7. Operacje listowe

Kontenery udostępniają następujące operacje listowe:

Operacje listowe	
<code>q=c.insert(p,x)</code>	Dodaje <code>x</code> przed <code>p</code> ; stosuje kopiowanie lub przenoszenie
<code>q=c.insert(p,n,x)</code>	Dodaje <code>n</code> kopii <code>x</code> przed <code>p</code> ; jeśli <code>c</code> jest kontenerem asocjacyjnym, <code>p</code> wskazuje, w którym miejscu zacząć szukanie
<code>q=c.insert(p,first,last)</code>	Dodaje elementy z przedziału <code><first,last></code> przed <code>p</code> ; nie dotyczy kontenerów asocjacyjnych
<code>q=c.insert(p,{elem})</code>	Dodaje elementy z <code>initializer_list{elem}</code> przed <code>p</code> ; <code>p</code> wskazuje, gdzie rozpocząć szukanie miejsca do wstawienia nowego elementu; dotyczy tylko kontenerów asocjacyjnych
<code>q=c.emplace(p,args)</code>	Dodaje element utworzony z <code>args</code> przed <code>p</code> ; nie dotyczy kontenerów asocjacyjnych
<code>q=c.erase(p)</code>	Usuwa z <code>c</code> element znajdujący się na pozycji <code>p</code>
<code>q=c.erase(first,last)</code>	Kasuje <code><first,last></code> z <code>c</code>
<code>c.clear()</code>	Kasuje wszystkie elementy z <code>c</code>

Wynik funkcji wstawiających (`insert()`) `q` wskazuje ostatni wstawiony element. W przypadku funkcji usuwających (`erase()`) `q` wskazuje element, który znajdował się za ostatnim usuniętym elementem.

W przypadku kontenerów o ciągłej alokacji, np. `vector` i `deque`, wstawianie i usuwanie elementu może spowodować przesunięcie pozostałych elementów. Wówczas iterator wskazujący jeden z przesuniętych elementów staje się bezużyteczny. Przeniesieniu ulegają elementy znajdujące się za miejscem wstawiania lub usuwania albo wszystkie elementy, jeśli dojdzie do re-alokacji z powodu przekroczenia pojemności kontenera. Na przykład:

```
vector<int> v {4,3,5,1};
auto p = v.begin() + 2; // wskazuje element v[2], czyli 5
v.push_back(6);         // p staje się bezużyteczny; v == {4,3,5,1,6}
p = v.begin() + 2;     // wskazuje element v[2], czyli 5
auto p2 = v.begin() + 4; // p2 wskazuje element v[4], czyli 6
v.erase(v.begin() + 3); // v == {4,3,5,6}; p jest nadal przydatny; p2 jest nieprzydatny
```

Każda operacja dodająca element do wektora może spowodować realokację wszystkich elementów (13.6.4).

Operacji `emplace()` używa się, gdy nie ma eleganckiej notacji lub możliwości efektywnego utworzenia obiektu, a następnie jego skopiowania (lub przeniesienia) do kontenera. Na przykład:

```
void user(list<pair<string,double>>& lst)
{
    auto p = lst.begin();
    while (p!=lst.end() && p->first!="Dania")           // znajduje miejsce wstawiania
        /* nic nie robi */;
    p=lst.emplace(p,"Anglia",7.5);                      // zwięzle i eleganckie
    p=lst.insert(p,make_pair("Francja",9.8));            // funkcja pomocnicza
    p=lst.insert(p,pair<string,double>{"Grecja",3.14}); // rozwlekłe
}
```

Kontener `forward_list` nie udostępnia operacji, jak np. `insert()`, działających przed elemencem wskazywanym przez iterator. Operacji takiej nie dałoby się zaimplementować, ponieważ nie ma ogólnego sposobu na znalezienie poprzedniego elementu w liście `forward_list`, mając do dyspozycji tylko iterator. Lista ta udostępnia natomiast operacje działające na elemencie znajdującym się za wskazywanym przez iterator, np. `insert_after()`. Analogicznie nieuporządkowane kontenery wykorzystują funkcję `emplace_hint()` w celu dostarczenia wskazówki, nie zaś zwykłej funkcji `emplace()`.

31.3.8. Inne operacje

Kontenery można porównywać i zamieniać:

Porównywania i zamianianie	
<code>c1==c2</code>	Czy wszystkie elementy na odpowiednich pozycjach w <code>c1</code> i <code>c2</code> są równe?
<code>c1!=c2</code>	<code>!(c1==c2)</code>
<code>c1<c2</code>	Czy <code>c1</code> jest w porządku leksykograficznym przed <code>c2</code> ?
<code>c1<=c2</code>	<code>!(c2<c1)</code>
<code>c1>c2</code>	<code>c2<c1</code>
<code>c1>=c2</code>	<code>!(c1<c2)</code>
<code>c1.swap(c2)</code>	Zamienia wartościami <code>c1</code> i <code>c2</code>
<code>swap(c1,c2)</code>	<code>c1.swap(c2)</code>

Przy porównywaniu kontenerów przy użyciu operatora (np. `<=`) elementy są porównywane za pomocą równoważnego operatora porównywania elementów wygenerowanego z `==` lub `<` (np. `a>b` jest wykonywane jako `!(b<a)`).

Operacje `swap()` zamieniają zarówno elementy, jak i alokatory.

31.4. Kontenery

W tym podrozdziale znajduje się bardziej szczegółowy opis:

- domyślnego kontenera `vector` (31.4.1);
- list powiązanych: `list` i `forward_list` (31.4.2);
- kontenerów asocjacyjnych, np. `map` i `unordered_map` (31.4.3).

31.4.1. vector

Znajdujący się w bibliotece STL typ `vector` to domyślny kontener w języku C++, tzn. należy go używać zawsze, jeśli nie ma się bardzo dobrego powodu, aby użyć czegoś innego. Jeżeli jako alternatywę bierzesz pod uwagę użycie słownika albo tablicę wbudowaną, to zastanów się dwa razy, zanim dokonasz wyboru.

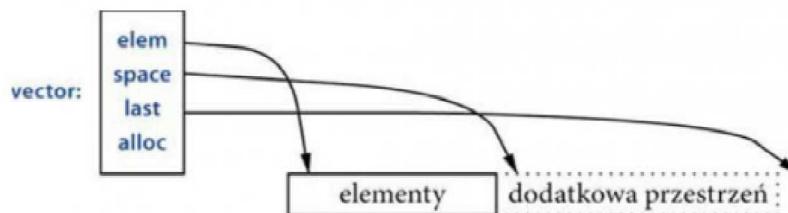
W podrozdziale 31.3 znajduje się opis operacji wektora, które można porównać z operacjami dostępnymi dla innych kontenerów. Jednak wektor jest tak ważnym kontenerem, że warto przyjrzeć mu się dokładniej, ze szczególnym uwzględnieniem jego operacji.

Argumenty szablonowe i typy składowe wektora są zdefiniowane następująco:

```
template<typename T, typename Allocator = allocator<T>>
class vector {
public:
    using reference = value_type&;
    using const_reference = const value_type&;
    using iterator = /*zależne od implementacji*/;
    using const_iterator = /*zależne od implementacji*/;
    using size_type = /*zależne od implementacji*/;
    using difference_type = /*zależne od implementacji*/;
    using value_type = T;
    using allocator_type = Allocator;
    using pointer = typename allocator_traits<Allocator>::pointer;
    using const_pointer = typename allocator_traits<Allocator>::const_pointer;
    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
    //...
};
```

31.4.1.1. Powiększanie wektora

Spójrz na schemat układu obiektu typu `vector` w pamięci (opis znajduje się w podrozdziale 13.6):



Użycie zarówno rozmiaru (liczby elementów), jak i pojemności (liczby dostępnych miejsc na elementy bez przeprowadzania realokacji) sprawia, że zwiększenie wektora przez funkcję `push_back()` jest względnie efektywne: nie trzeba realokować wszystkich elementów za każdym razem, gdy doda się nowy elementy, tylko wystarczy to zrobić raz na jakiś czas po przekroczeniu pojemności (13.6). Standard nie określa, o ile ma być zwiększana pojemność przy realokacji, ale powszechnie stosuje się zwiększenie o połowę dotychczasowej pojemności. Kiedyś bardzo ostrożnie podchodziłem do wczytywania danych do wektora przy użyciu funkcji `reserve()`. Jakież było moje zdziwienie, kiedy odkryłem, że praktycznie we wszystkich przypadkach, z jakimi miałem do czynienia, użycie tej funkcji nie miało zauważalnego wpływu na wydajność. Domyślona strategia zwiększenia była tak samo dobra jak moje szacunki, więc zaprzestałem optymalizowania wydajności kodu zawierającego wywołania `reserve()`. W zamian funkcji tej używam w celu zwiększenia przewidywalności czasu realokacji oraz zapobiegania uszkodzeniu wskaźników i iteratorów.

Dzięki dodatkowemu wolnemu miejscu w wektorze iteratory pozostają poprawne do czasu następienia relokacji. Spójrz na przykład wczytywania liter do bufora i zapamiętywania granic słów:

```

vector<char> chars; // bufor wejściowy na znaki
constexpr int max = 20000;
chars.reserve(max);
vector<char*> words; // wskaźniki do początków słów

bool in_word = false;
for (char c; cin.get(c); ) {
    if (isalpha(c)) {
        if (!in_word) { // znaleziono początek słowa
            in_word = true;
            chars.push_back(0); // koniec poprzedniego słowa
            chars.push_back(c);
            words.push_back(&chars.back());
        }
        else
            chars.push_back(c);
    }
    else
        in_word = false;
}
if (in_word)
    chars.push_back(0); // koniec ostatniego słowa

if (max<chars.size()) { // ups: rozmiar wektora chars przekroczył pojemność; słowa są niepoprawne
    ...
}
chars.shrink_to_fit(); // zwolnienie nadmiarowej pojemności

```

Gdybym nie użył tu funkcji `reserve()`, wskaźniki w wektorze `words` zostałyby uszkodzone, gdyby wywołanie `chars.push_back()` spowodowało relokację. Słowo „uszkodzone” w tym przypadku oznacza, że wskaźniki te miałyby niezdefiniowane zachowanie. Mogłyby — ale nie na pewno — wskazywać jakieś elementy, ale prawie na pewno nie byłyby to te same elementy, które wskazywały przed relokacją.

Możliwość zwiększania wektora przy użyciu funkcji `push_back()` i podobnych operacji sprawia, że używanie niskopoziomowych funkcji w stylu języka C, np. `malloc()` i `realloc()` (43.5), jest tak samo niepotrzebne, jak żmudne i ryzykowne, jeśli chodzi o możliwość popełnienia błędu.

31.4.1.2. Wektor i zagnieżdżanie

Wektor (i inne podobne struktury o ciąglej alokacji) ma trzy zalety w porównaniu z innymi strukturami danych:

- Elementy wektora są efektywnie przechowywane, tzn. żaden element nie powoduje dodatkowego narzutu. W przybliżeniu wektor typu `vector<X>` zajmuje `sizeof(vector<X>)+vec.size()*sizeof(X)` pamięci. Wielkość `sizeof(vector<X>)` wynosi około 12 bajtów, co w przypadku dużych wektorów jest bez znaczenia.

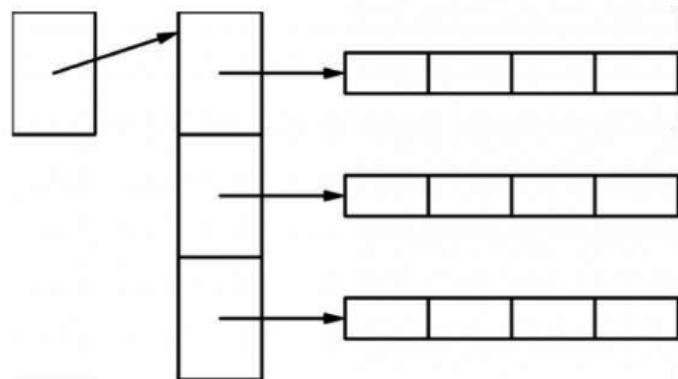
- Przeglądanie wektora jest bardzo szybkie. Aby przejść do następnego elementu, kod nie musi posługiwać się pośrednikiem w postaci wskaźnika, a nowoczesne architektury są zoptymalizowane pod kątem przeglądania kolejnych elementów struktur podobnych do wektora. Dzięki temu operacje liniowo przeglądające wektory, np. `find()` i `copy()`, są prawie optymalne.
- Wektor umożliwia łatwy i efektywny swobodny dostęp do elementów, co jest podstawą wydajności wielu działających na wektorach algorytmów, np. `sort()` i `binary_search()`.

Wielu programistów nie docenia tych zalet. Ale na przykład lista dwukierunkowa, choćby `list`, zazwyczaj dodaje narzut pamięciowy w postaci czterech słów na element (dwa łącza plus nagłówek alokacji pamięci wolnej), a jej przeglądanie może być o rząd wielkości mniej wydajne od przeglądania wektora zawierającego równoważne dane. Efekt porównania tych dwóch struktur może być tak spektakularny, że warto je wykonać samemu [Stroustrup 2012a].

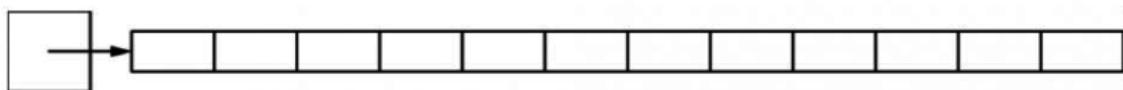
Korzyści wynikające ze zwięzości i szybkiego dostępu można nieumyślnie zniweczyć. Zastanówmy się nad tym, jakie są możliwości reprezentacji dwuwymiarowej macierzy. Od razu nasuwają się dwa oczywiste potencjalne rozwiązania:

- Wektor wektorów: struktura `vector<vector<double>>` o dostępie przy użyciu podwójnych indeksów w stylu języka C: `m[i][j]`.
- Specjalny typ macierzowy, `Matrix<2,double>` (rozdział 29.) przechowujący elementy w sposób ciągły (np. w wektorze `vector<double>`) i obliczający lokalizacje w tej sekwencji z par wskaźników `m(i,j)`.

Rozmieszczenie w pamięci elementów wektora `vector<vector<double>>` o wymiarach 3×4 byłoby następujące:



Natomiast struktura `Matrix<2,double>` wyglądałaby tak:



Do zbudowania obiektu `vector<vector<double>>` potrzebne są cztery wywołania konstruktora z czterema operacjami alokacji elementów w pamięci wolnej. Aby uzyskać dostęp do elementu, należy skorzystać z podwójnej pośredniości.

Do zbudowania obiektu `Matrix<2,double>` potrzebne jest jedno wywołanie konstruktora z jedną alokacją w pamięci wolnej. Aby uzyskać dostęp do elementu, wystarczy pojedynczą pośredniość.

Gdy już dotrze się do jednego elementu w wierszu, przejście do następnego nie wymaga dalszego posługiwania się pośredniością, a więc dostęp do struktury `vector<vector<double>>` nie zawsze jest dwa razy kosztowniejszy od dostępu do `Matrix<2,double>`. Jednak dla algorytmów o wysokich wymaganiach dotyczących wydajności koszty alokacji, dezalokacji oraz dostępu implikowane przez powiązaną strukturę, taką jak `vector<vector<double>>`, mogą stanowić duży problem.

W strukturze `vector<vector<double>>` istnieje ryzyko, że poszczególne wiersze będą miały różne rozmiary. Czasami może to być korzystne, ale najczęściej stwarza tylko dodatkowe okazje do popełnienia błędu i jest uciążliwe dla testera.

Wraz ze zwiększaniem liczb wymiarów sytuacja komplikuje się jeszcze bardziej: porównaj liczbę pośredniości i alokacji dla struktur `vector<vector<double>>` i `Matrix<3,double>`.

Podsumowując: czasami natrafiam na przypadki niedoceniania lub nieumyślnego niwelowania zalet zwięzości struktur danych. A przecież korzyści te są logiczne i ściśle wiążą się z wydajnością. Jeśli połączy się to dodatkowo z tendencją do nadużywania wskaźników i operatora `new`, można stwierdzić, że istnieje powszechny problem. Zastanów się na przykład nad trudnością utworzenia, kosztami wykonawczymi i pamięciowymi oraz ryzykiem popełnienia błędu przy pisaniu implementacji dwuwymiarowej struktury, której wiersze są zaimplementowane jako niezależne obiekty alokowane w pamięci wolnej: `vector<vector<double>>`.

31.4.1.3. Wektor a tablice

Wektor jest uchwytem do zasobów. To właśnie ta cecha sprawia, że można zmieniać jego rozmiar i że ma on efektywną semantykę przenoszenia. Jednak czasami sposób przechowywania struktury danych oddziennie od uchwytu stanowi wadę w porównaniu ze strukturami o innej budowie (np. tablicami wbudowanymi i array). Przechowywanie sekwencji elementów na stosie lub w innym obiekcie może być zarówno zaletą, jak i wadą, jeśli chodzi o wydajność.

Wektor przechowuje poprawnie zainicjowane obiekty. Dzięki temu używanie ich jest łatwe i można polegać na prawidłowej destrukcji elementów. Jednak czasami cecha ta stanowi wadę w porównaniu ze strukturami danych (np. tablicami wbudowanymi i array), które pozwalają na zapisywanie niezainicjowanych elementów.

Na przykład elementów tablicy nie trzeba inicjować przed wczytaniem do nich wartości:

```
void read()
{
    array<int,MAX> a;
    for (auto& x : a)
        cin.get(&x);
}
```

W celu uzyskania podobnego efektu (bez konieczności określania `MAX`) w wektorze użylibyśmy funkcji `emplace_back()`.

31.4.1.4. vector a string

Obiekt `vector<char>`, podobnie jak `string`, jest zmienną, ciągłą sekwencją elementów typu `char`. Jak w takim razie wybrać jeden z tych typów?

Wektor jest ogólnym mechanizmem przechowywania wartości. Nie ma w nim żadnych założeń dotyczących relacji zachodzących między przechowywanymi w nim wartościami.

Dla wektora `vector<char>` łańcuch `Witaj, świecie!` to tylko sekwencja 14 elementów typu `char`. Posortowanie ich w celu uzyskania ciągu `! , Waceeijśtw` (ze spacją na początku) jest jak najbardziej sensowne. Dla porównania typ `string` służy do przechowywania sekwencji znaków. Zakłada się, że znaki te łączy ważna relacja. Dlatego właśnie rzadko sortuje się znaki przechowywane w obiektach typu `string`, ponieważ to niszczy ich znaczenie. Znajduje to odzwierciedlenie w sposobie działania niektórych operacji (np. `c_str()`, `>>` oraz `find()`, „wiedzą”, że łańcuchy w stylu języka C są zakończone zerem). Implementacja typu `string` odzwierciedla założenia dotyczące typowego sposobu używania łańcuchów. Na przykład optymalizacja używania krótkich łańcuchów (19.3.3) byłaby bezsensowna, gdyby nie fakt, że krótkie łańcuchy są bardzo często używane, a w takim przypadku eliminacja pamięci wolnej jest warta zachodu.

Czy powinno się też stosować optymalizację używania krótkich wektorów? Myślę, że nie, ale żeby mieć pewność, trzeba by było przeprowadzić szeroko zakrojone badania empiryczne.

31.4.2. Listy

W bibliotece STL znajdują się dwa rodzaje list:

- `list` — lista dwukierunkowa,
- `forward_list` — lista jednokierunkowa.

Lista `list` jest sekwencją zoptymalizowaną pod kątem wstawiania i usuwania elementów. Wstawianie i usuwanie elementów w tej strukturze nie ma wpływu na położenie pozostałych przechowywanych w niej elementów. W szczególności nie powoduje to uszkodzenia iteratörów wskazujących elementy.

W porównaniu z wektorem indeksowanie listy byłoby strasznie powolne i dlatego struktury te w ogóle nie obsługują indeksowania. Do poruszania się po elementach listy należy używać operacji `advance()` i podobnych (33.1.4). Listę można przejrzeć za pomocą iteratorów. Struktura `list` udostępnia iteratory dwukierunkowe (33.1.2), a `forward_list` — jednokierunkowe (odzwierciedla to jej nazwa).

Domyślnie elementy listy `list` są alokowane indywidualnie w pamięci oraz zawierają wskaźniki na poprzedni i następny element (11.2.2). W porównaniu z wektorem lista `list` zużywa więcej pamięci na element (najczęściej przynajmniej cztery słowa), a jej przeglądanie (iteracja) jest znacznie wolniejsze, ponieważ polega na wykorzystaniu pośredniości zamiast bezpośredniego dostępu.

Struktura `forward_list` to lista jednokierunkowa. Należy ją traktować jako strukturę danych zoptymalizowaną pod kątem przechowywania pustych lub bardzo krótkich list, które zwykle przegląda się od początku. Z oszczędności w strukturze tej brak nawet operacji `size()`. Pusta lista `forward_list` zajmuje tylko jedno słowo. Aby dowiedzieć się, ile elementów zawiera taka lista, należy je po prostu policzyć. Jeśli liczba elementów jest na tyle duża, że ich liczenie jest kosztowne, należy rozważyć możliwość użycia innego kontenera.

Z wyjątkiem indeksowania, zarządzania pojemnością oraz operacji `size()`, których brak w `forward_list`, listy biblioteki STL zawierają te same typy i operacje składowe co wektor (31.4). Dodatkowo `list` i `forward_list` udostępniają także specyficzne funkcje, dostępne tylko w nich:

Operacje <code>list<T></code> i <code>forward_list<T></code> (iso.23.3.4.5, iso.23.3.5.4)	
<code>lst.push_front(x)</code>	Dodaje x (przy użyciu kopiowania lub przenoszenia) przed pierwszym elementem <code>lst</code>
<code>lst.pop_front()</code>	Usuwa pierwszy element <code>lst</code>
<code>lst.emplace_front(args)</code>	Dodaje <code>T{args}</code> przed pierwszym elementem <code>lst</code>
<code>lst.remove(v)</code>	Usuwa z <code>lst</code> wszystkie elementy o wartości v
<code>lst.remove_if(f)</code>	Usuwa z <code>lst</code> wszystkie elementy, dla których <code>f(x)==true</code>
<code>lst.unique()</code>	Usuwa przylegające powielone elementy z <code>lst</code>
<code>lst.unique(f)</code>	Usuwa przylegające powielone elementy z <code>lst</code> , do porównywania wykorzystując f
<code>lst.merge(lst2)</code>	Scala uporządkowane listy <code>lst</code> i <code>lst2</code> , używając < do porządkowania; <code>lst2</code> jest wcielana do <code>lst</code> i opróżniana w tym procesie
<code>lst.merge(lst2,f)</code>	Scala uporządkowane listy <code>lst</code> i <code>lst2</code> przy użyciu f do porządkowania; <code>lst2</code> jest wcielana do <code>lst</code> i opróżniana w tym procesie
<code>lst.sort()</code>	Sortuje <code>lst</code> przy użyciu < do porównywania
<code>lst.sort(f)</code>	Sortuje <code>lst</code> przy użyciu f do porównywania
<code>lst.reverse()</code>	Odwraca kolejność elementów <code>lst</code> ; noexcept

W odróżnieniu od ogólnych algorytmów `remove()` i `unique()` (32.5) algorytmy składowe mają realny wpływ na rozmiar listy. Na przykład:

```
void use()
{
    list<int> lst {2,3,2,3,5};
    lst.remove(3); // lst zawiera teraz {2,2,5}
    lst.unique(); // lst zawiera teraz {2,5}
    cout << lst.size() << '\n'; // drukuje 2
}
```

Algorytm `merge()` jest stabilny, tzn. równoważne elementy zachowują pierwotną kolejność.

Operacje kontenera <code>list<T></code> (iso.23.3.5.5)	
p wskazuje element <code>lst</code> lub <code>lst.end()</code>	
<code>lst.splice(p,lst2)</code>	Wstawia elementy z listy <code>lst2</code> przed p; lista <code>lst2</code> zostaje opróżniona
<code>lst.splice(p,lst2,p2)</code>	Wstawia element wskazywany przez p2 w liście <code>lst2</code> , przed p; element wskazywany przez p2 zostaje usunięty z <code>lst2</code>
<code>lst.splice(p,lst2,b,e)</code>	Wstawia elementy z przedziału <b,e> listy <code>lst2</code> przed p; elementy te zostają usunięte z <code>lst2</code>

Operacja `splice()` nie kopiuje wartości elementów i nie uszkadza wskazujących ich iteratorów. Na przykład:

```
list<int> lst1 {1,2,3};
list<int> lst2 {5,6,7};

auto p = lst1.begin();
```

```

++p; // p wskazuje 2

auto q = lst2.begin();
++q; // q wskazuje 6

lst1.splice(p,lst2); // lst1 to teraz {1,5,6,7,2,3}; lst2 to {}
// p nadal wskazuje 2, a q wskazuje 6

```

Lista `forward_list` nie daje dostępu do elementu znajdującego się przed wskazywanym przez iterator (nie ma w niej łączy do poprzednich elementów) i dlatego operacje `emplace()`, `erase()` i `splice()` działają na pozycjach za iteratorem:

Operacje kontenera <code>forward_list<T></code> (iso.23.3.4.6)	
<code>p2=lst.emplace_after(p,args)</code>	Umieszcza element utworzony z args za p; p2 wskazuje ten nowy element
<code>p2=lst.insert_after(p,x)</code>	Wstawia x za p; p2 wskazuje nowy element
<code>p2=lst.insert_after(p,n,x)</code>	Wstawia n kopii x za p; p2 wskazuje ostatni z nowych elementów
<code>p2=lst.insert_after(p,b,e)</code>	Wstawia <b,e> za p; p2 wskazuje ostatni z nowych elementów
<code>p2=lst.insert_after(p,{elem})</code>	Wstawia {elem} za p; p2 wskazuje ostatni z nowych elementów; elem jest listą inicjalizacyjną <code>initializer_list</code>
<code>p2=lst.erase_after(p)</code>	Usuwa element znajdujący się za p; p2 wskazuje element znajdujący się za p lub <code>lst.end()</code>
<code>p2=lst.erase_after(b,e)</code>	Usuwa <b,e>; p2=e
<code>lst.splice_after(p,lst2)</code>	Wstawia elementy z lst2 za p
<code>lst.splice_after(p,b,e)</code>	Wstawia elementy z <b,e> za p
<code>lst.splice_after(p,lst2,p2)</code>	Wstawia elementy z p2 za p; usuwa p2 z lst2
<code>lst.splice_after(p,lst2,b,e)</code>	Wstawia elementy z <b,e> za p; usuwa <b,e> z lst2

Wszystkie operacje listy `list` są **stabilne**, tzn. zachowują relatywną kolejność równoważnych elementów.

31.4.3. Kontenery asocjacyjne

Kontenery asocjacyjne umożliwiają dostęp do elementów za pomocą kluczy. Wyróżnia się dwa rodzaje tych kontenerów:

- **Uporządkowane kontenery asocjacyjne:** wyszukują elementy, posługując się jakimś kryterium porównawczym, domyślnie `<` (mniejszość). Są implementowane jako zrównoważone drzewa binarne, najczęściej czerwono-czarne.
- **Nieuporządkowane kontenery asocjacyjne:** wyszukują elementy, posługując się funkcją mieszającą (ang. *hash function*). Są implementowane jako tablice mieszające z rozwiązywaniem kolizji poprzez dowiązywanie (ang. *linked overflow*).

Oba rodzaje występują jako:

- **słowniki** (`map`) — sekwencje par {klucz,wartość},
- **zbiory** (`set`) — słowniki bez wartości (można też powiedzieć, że klucze są wartościami).

Słowniki i zbiory, nieważne czy uporządkowane, czy nie, występują w dwóch wariantach:

- „Zwykłe” zbiory lub słowniki, w których dla każdego klucza istnieje unikatowy wpis.
- „Wielokrotne” zbiory i słowniki, w których dla każdego klucza może istnieć wiele wpisów.

Nazwa kontenera asocjacyjnego określa jego miejsce w tej trójwymiarowej przestrzeni: {zbiór|słownik, zwykły|nieuporządkowany, zwykły|wielokrotny}. Słowo oznaczające „zwykły” nigdy się w nazwie nie pojawia, dlatego wyróżnia się następujące nazwy kontenerów asocjacyjnych:

Kontenery asocjacyjne (iso.23.4.1, iso.23.5.1)
set multiset unordered_set unordered_multiset
map multimap unordered_map unordered_multimap

Argumenty szablonowe tych kontenerów są opisane w podrozdziale 31.4.

Pod względem budowy wewnętrznej kontenery `map` i `unordered_map` bardzo się różnią (ich graficzne reprezentacje są przedstawione w podrozdziale 31.2.1). Kontener `map` wyszukuje klucze w drzewie zrównoważonym (operacja o złożoności $O(\log(n))$) przy użyciu kryterium porównawczego (najczęściej `<`), natomiast `unordered_map` stosuje funkcję mieszającą do klucza w celu znalezienia miejsca w tablicy mieszania (przy dobrej funkcji mieszającej operacja ta ma złożoność $O(1)$).

31.4.3.1. Uporządkowane kontenery asocjacyjne

Oto lista argumentów szablonowych i typów składowych kontenera `map`:

```
template<typename Key,
         typename T,
         typename Compare = less<Key>,
         typename Allocator = allocator<pair<const Key, T>>>
class map {
public:
    using key_type = Key;
    using mapped_type = T;
    using value_type = pair<const Key, T>;
    using key_compare = Compare;
    using allocator_type = Allocator;
    using reference = value_type&;
    using const_reference = const value_type&;
    using iterator = /* zależy od implementacji */;
    using const_iterator = /* zależy od implementacji */;
    using size_type = /* zależy od implementacji */;
    using difference_type = /* zależy od implementacji */;
    using pointer = typename Allocator_traits<Allocator>::pointer;
    using const_pointer = typename Allocator_traits<Allocator>::const_pointer;
    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
    class value_compare { /* operator()(k1,k2) wykonuje key_compare()(k1,k2)* */};
    ...
};
```

Oprócz konstruktorów wymienionych w podrozdziale 31.3.2 kontenery asocjacyjne zawierają konstruktory umożliwiające przekazanie porównywacza:

Konstruktory <code>map<K,T,C,A></code> (iso.23.4.4.2)	
<code>map m {cmp,a};</code>	Tworzy obiekt <code>m</code> wykorzystujący porównywacz <code>cmp</code> i alokator <code>a</code> ; <code>explicit</code>
<code>map m {cmp};</code>	<code>map m {cmp, A{}}; explicit</code>
<code>map m {};</code>	<code>map m {C{}}; explicit</code>
<code>map m {b,e,cmp,a};</code>	Tworzy obiekt <code>m</code> wykorzystujący porównywacz <code>cmp</code> i alokator <code>a</code> ; do inicjacji używa elementów z <code><b,e></code>
<code>map m {b,e,cmp};</code>	<code>map m {b,e,cmp,A{}};</code>
<code>map m {b,e};</code>	<code>map m {b,e,C{}};</code>
<code>map m {m2};</code>	Konstruktory kopiujące i przenoszące
<code>map m {a};</code>	Tworzy domyślny słownik; używa alokatora <code>a</code> ; <code>explicit</code>
<code>map m {m2,a};</code>	Tworzy obiekt <code>m</code> przy użyciu kopiowania lub przenoszenia z <code>m2</code> ; używa alokatora <code>a</code>
<code>map m {{elem},cmp,a};</code>	Tworzy obiekt <code>m</code> używający porównywacza <code>cmp</code> i alokatora <code>a</code> ; do inicjacji używa elementów z <code>initializer_list{elem}</code>
<code>map m {{elem},cmp};</code>	<code>map m {{elem},cmp,A{}}</code>
<code>map m {{elem}};</code>	<code>map m {{elem},C{}}</code>

Na przykład:

```
map<string,pair<Coordinate,Coordinate>> locations
{
    {"Kopenhaga", {"55:40N", "12:34E"}},
    {"Rzym", {"41:54N", "12:30E"}},
    {"Nowy Jork", {"40:40N", "73:56W"}}
};
```

Kontenery asocjacyjne zawierają szeroki wachlarz operacji wstawiania i wyszukiwania:

Operacje kontenerów asocjacyjnych (iso.23.4.4.1)	
<code>v=c[k]</code>	<code>v</code> jest referencją do elementu o kluczu <code>k</code> ; jeśli klucz <code>k</code> nie zostanie znaleziony, następuje wstawienie <code>{k,mapped_type{}}</code> do <code>c</code> ; dotyczy tylko struktur <code>map</code> i <code>unordered_map</code>
<code>v=c.at(k)</code>	<code>v</code> jest referencją do elementu o kluczu <code>k</code> ; jeśli klucz <code>k</code> nie zostanie znaleziony, następuje zgłoszenie wyjątku <code>out_of_range</code> ; dotyczy tylko struktur <code>map</code> i <code>unordered_map</code>
<code>p=c.find(k)</code>	<code>p</code> wskazuje pierwszy element o kluczu <code>k</code> lub <code>c.end()</code>
<code>p=c.lower_bound(k)</code>	<code>p</code> wskazuje pierwszy element o kluczu <code>>=k</code> lub <code>c.end()</code> ; dotyczy tylko kontenerów uporządkowanych
<code>p=c.upper_bound(k)</code>	<code>p</code> wskazuje pierwszy element o kluczu <code>>k</code> lub <code>c.end()</code> ; dotyczy tylko kontenerów uporządkowanych
<code>pair(p1,p2)=c.equal_range(k)</code>	<code>p1=c.lower_bound(k); p2=c.upper_bound(k)</code>

Operacje kontenerów asocjacyjnych (iso.23.4.4.1)	
<code>pair(p,b)=c.insert(x)</code>	x to value_type lub coś, co można skopiować do value_type (np. dwuelementowa krotka); b ma wartość true, jeżeli x zostanie wstawiony, a false, jeśli w strukturze już był wpis o kluczu z x; p wskazuje (potencjalnie nowy) element o kluczu z x
<code>p2=c.insert(p,x)</code>	x to value_type lub coś, co można skopiować do value_type (np. dwuelementowa krotka); p wskazuje, gdzie należy zacząć szukanie elementu z kluczem z x; p wskazuje (potencjalnie nowy) element o kluczu z x; p2 wskazuje (potencjalnie nowy) element z kluczem x
<code>c.insert(b,e)</code>	<code>c.insert(*p)</code> dla każdego p w <b,e>
<code>c.insert({args})</code>	Wstawia wszystkie elementy z initializer_list args; element jest typu pair<key_type,mapped_type>
<code>p=c.emplace(args)</code>	p wskazuje obiekt typu value_type struktury c utworzony z args i wstawiony do c
<code>p=c.emplace_hint(h,args)</code>	p wskazuje obiekt typu value_type struktury c utworzony z args i wstawiony do c; h jest iteratorem w c, potencjalnie używanym jako wskazówka, gdzie należy rozpoczęć szukanie miejsca dla nowego wpisu
<code>r=c.key_comp()</code>	r jest kopią obiektu porównywania kluczy; dotyczy tylko kontenerów uporządkowanych
<code>r=c.value_comp()</code>	r jest kopią obiektu porównywania wartości; dotyczy tylko kontenerów uporządkowanych
<code>n=c.count(k)</code>	n jest liczbą elementów o kluczu k

Operacje specyficzne kontenerów nieuporządkowanych są opisane w podrozdziale 31.4.3.5.

Jeśli klucz k nie zostanie znaleziony przez operację indeksowania `m[k]`, to wstawiana jest wartość domyślna. Na przykład:

```
map<string,string> dictionary;
dictionary["morze"]="dużo wody"; // wstawienie lub przypisanie do elementu
cout << dictionary["foka"]; // odczytanie wartości
```

Jeśli foka nie znajduje się w słowniku, nic nie zostanie wydrukowane, tylko zostanie wstawiony pusty łańcuch jako wartość klucza foka, który następnie zostanie zwrócony jako wynik szukania.

Jeśli nie o to chodzi programiście, może on bezpośrednio użyć funkcji `find()` i `insert()`:

```
auto q = dictionary.find("foka"); // szuka, nie wstawia
if (q==dictionary.end()) {
    cout << "nie znaleziono pozycji";
    dictionary.insert(make_pair("foka","lubi ryby"));
}
else
    cout q->second;
```

W rzeczywistości operator [] jest czymś więcej niż wygodną formą zapisu zastępującą `insert()`. Wynik operacji `m[k]` jest równoważny wynikowi operacji `(*m.insert(make_pair(k,V{}))).second`, gdzie V jest mapowaną wartością.

Notacja `insert(make_pair())` jest przydługa i dlatego można by było użyć `emplace()`:

```
dictionary.emplace("krowa morska", "wyginęła");
```

Jeśli optymalizator jest dobry, to ta operacja również może być wydajna.

Jeśli spróbujesz wstawić do słownika `map` element, którego klucz już istnieje, słownik pozostanie niezmieniony. Jeśli chcesz przechowywać po kilka wartości pod jednym kluczem, użyj wielosłownika — `multimap`.

Pierwszy iterator pary `pair` (34.2.4.1) zwrócony przez `equal_range()` to `lower_bound()`, a drugi to `upper_bound()`. Aby wydrukować wartości wszystkich elementów z kluczem "apple" znajdujących się w strukturze `multimap<string,int>`, można napisać:

```
multimap<string,int> mm {{ "apple",2}, { "pear",2}, {"apple",7}, {"orange",2}, {"apple",9}};
const string k {"apple"};
auto pp = mm.equal_range(k);
if (pp.first==pp.second)
    cout << "brak elementu o wartości " << k << "\n";
else {
    cout << "elementy o wartości " << k << ":\n";
    for (auto p=pp.first; p!=pp.second; ++p)
        cout << p->second << ' ';
}
```

Program ten wydrukuje napis 2 7 9.

Równie dobrze można by było napisać:

```
auto pp = make_pair(m.lower_bound(),m.upper_bound());
//...
```

To jednak wymagałoby dodatkowego przeglądania słownika. Operacje `equal_range()`, `lower_bound()` oraz `upper_bound()` są dostępne także dla sekwencji sortowanych (32.6).

Zbiór (`set`) można sobie wyobrazić jako słownik bez osobnego `value_type`, ponieważ w zbiorze `value_type` jest jednocześnie `key_type`. Na przykład:

```
struct Record {
    string label;
    int value;
};
```

Aby móc używać obiektów `set<Record>`, należy dostarczyć funkcję porównującą. Na przykład:

```
bool operator<(const Record& a, const Record& b)
{
    return a.label<b.label;
}
```

Teraz możemy pisać:

```
set<Record> mr {{ "duck",10}, {"pork",12}};

void read_test()
{
    for (auto& r : mr) {
```

```

        cout << '{' << r.label << ':' << r.value << '}';
    }
    cout << endl;
}

```

Klucz elementu w asocjacyjnym kontenerze jest niezmienny (iso.23.2.4), co oznacza, że wartości zbiorów nie można zmieniać. Nie można nawet zmienić składowej elementu, która nie jest w żaden sposób brana pod uwagę przy porównywaniu. Na przykład:

```

void modify_test()
{
    for (auto& r : mr)
        ++r.value; // błąd: elementy zbioru są niezmienne
}

```

Jeśli potrzebna jest możliwość modyfikowania elementów, należy użyć słownika. Nie próbuj modyfikować klucza: gdyby Ci się to udało, mechanizm wyszukiwania elementów uległby awarii.

31.4.3.2. Nieuporządkowane kontenery asocjacyjne

Nieuporządkowane kontenery asocjacyjne (`unordered_map`, `unordered_set`, `unordered_multimap`, `unordered_multiset`) są tablicami mieszającymi. Jeśli chodzi o najprostsze zastosowania, to różnice między nimi a kontenerami uporządkowanymi są niewielkie, ponieważ większość operacji jest taka sama (31.4.3.1). Na przykład:

```

unordered_map<string,int> score1 {
    {"andrzej", 7}, {"robert",9}, {"wojtek",-3}, {"barbara",12}
};

map<string,int> score2 {
    {"andrzej", 7}, {"robert",9}, {"wojtek",-3}, {"barbara",12}
};

template<typename X, typename Y>
ostream& operator<<(ostream& os, const pair<X,Y>& p)
{
    return os << '{' << p.first << ',' << p.second << '}';
}

void user()
{
    cout <<"nieuporządkowany: ";
    for (const auto& x : score1)
        cout << x << ", ";

    cout << "\nuporządkowany: ";
    for (const auto& x : score2)
        cout << x << ", ";
}

```

Widoczna różnica polega na tym, że iteracja przez słownik `map` odbywa się w sposób uporządkowany, a przez słownik `unordered_map` nie:

```

nieuporządkowany: {andrzej,7}, {robert,9}, {wojtek,-3}, {barbara,12},
uporządkowany: {andrzej,9}, {robert, 7}, {wojtek,-3}, {barbara,12},

```

Iteracja przez `unordered_map` jest zależna od kolejności wstawiania elementów, funkcji mieszającej oraz współczynnika zapełnienia. Nie ma gwarancji, że elementy zostaną wydrukowane w takiej samej kolejności, w jakiej zostały wstawione.

31.4.3.3. Konstruowanie słowników nieuporządkowanych

Słownik `unordered_map` zawiera wiele argumentów szablonowych i składowych aliasów typów:

```
template<typename Key,
         typename T,
         typename Hash = hash<Key>,
         typename Pred = std::equal_to<Key>,
         typename Allocator = std::allocator<std::pair<const Key, T>>>
class unordered_map {
public:
    using key_type = Key;
    using value_type = std::pair<const Key, T>;
    using mapped_type = T;
    using hasher = Hash;
    using key_equal = Pred;
    using allocator_type = Allocator;
    using pointer = typename allocator_traits<Allocator>::pointer;
    using const_pointer = typename allocator_traits<Allocator>::const_pointer;
    using reference = value_type&;
    using const_reference = const value_type&;
    using size_type = /*zależne od implementacji */;
    using difference_type = /*zależne od implementacji */;
    using iterator = /*zależne od implementacji */;
    using const_iterator = /*zależne od implementacji */;
    using local_iterator = /*zależne od implementacji */;
    using const_local_iterator = /*zależne od implementacji */;
    //...
};
```

Domyślnie `unordered_map<X>` do mieszania wykorzystuje `hash<X>`, a do porównywania kluczy — `equal_to<X>`.

Domyślna implementacja `equal_to<X>` (33.4) porównuje wartości typu `X` przy użyciu operatora `==`.

W ogólnym (podstawowym) szablonie typ `hash` nie ma domyślnej definicji. Zdefiniowanie typu `hash<X>` w razie potrzeby jest zadaniem użytkownika. Dla najczęściej używanych typów, np. `string`, dostępne są standardowe specjalizacje typu `hash`:

Typy z <code>hash<T></code> (iso.20.8.12) dostępne w bibliotece standardowej			
<code>string</code>	<code>u16string</code>	<code>u32string</code>	<code>wstring</code>
<code>bool</code>	<code>znaki</code>	<code>liczby całkowite</code>	<code>typy zmiennoprzecinkowe</code>
<code>wskaźniki</code>	<code>type_index</code>	<code>thread::id</code>	<code>error_code</code>
<code>bitset<N></code>	<code>unique_ptr<T,D></code>	<code>shared_ptr<T></code>	

Funkcja mieszająca (np. specjalizacja `hash` dla `T` lub wskaźnik do funkcji) musi dać się wywoływać z argumentem typu `T` i zwracać `size_t` (iso.17.6.3.4). Dwa wywołania funkcji mieszającej dla tej samej wartości muszą dawać taki sam wynik i najlepiej, by wyniki tej funkcji były

równomiernie rozłożone w zbiorze wartości `size_t`, aby ryzyko, że $h(x) == h(y)$, gdy $x \neq y$, było minimalne.

Liczba możliwych kombinacji typów argumentów szablonowych, konstruktorów i wartości domyślnych kontenera nieuporządkowanego jest bardzo duża. Na szczęście można wyróżnić pewne prawidłowości:

Konstruktory <code>unordered_map<K,T,H,E,A></code> (iso.23.5.4)	
<code>unordered_map m {n,hf,eql,a}</code>	Tworzy <code>m</code> zawierający <code>n</code> kubelków, funkcję mieszającą <code>hf</code> , funkcję porównującą <code>eql</code> oraz alokator <code>a</code> ; <code>explicit</code>
<code>unordered_map m {n,hf,eql}</code>	<code>unordered_map m{n,hf,eql,allocator_type{}}</code> ; <code>explicit</code>
<code>unordered_map m {n,hf}</code>	<code>unordered_map m {n,hf,key_equal{}}</code> ; <code>explicit</code>
<code>unordered_map m {n}</code>	<code>unordered_map m {n,hasher{}}</code> ; <code>explicit</code>
<code>unordered_map m {}</code>	<code>unordered_map m {N}</code> ; liczba kubelków <code>N</code> zależy od implementacji; <code>explicit</code>

Wartość `n` jest liczbą elementów w przeciwnym razie pustego słownika.

Konstruktory <code>unordered_map<K,T,H,E,A></code> (iso.23.5.4)	
<code>unordered_map m {b,e,n,hf,eql,a};</code>	Tworzy słownik <code>m</code> zawierający <code>n</code> kubelków z elementami z przedziału <code><b,e></code> , używający funkcji mieszającej <code>hf</code> , funkcji porównującej <code>eql</code> oraz alokatora <code>a</code>
<code>unordered_map m {b,e,n,hf,eql};</code>	<code>unordered_map m {b,e,n,hf,eql,allocator_type{}}</code>
<code>unordered_map m {b,e,n,hf};</code>	<code>unordered_map m {b,e,n,hf,key_equal{}}</code>
<code>unordered_map m {b,e,n};</code>	<code>unordered_map m {b,e,n,hasher{}}</code>
<code>unordered_map m {b,e};</code>	<code>unordered_map m {b,e,N}</code> ; liczba kubelków <code>N</code> zależy od implementacji

Tutaj elementy początkowe są pobierane z sekwencji `<b,e>`. Liczba elementów będzie równa liczbie elementów w przedziale `<b,e>`, `distance(b,e)`.

Konstruktory <code>unordered_map<K,T,H,E,A></code> (iso.23.5.4)	
<code>unordered_map m {{elem},n,hf,eql,a}</code>	Tworzy słownik <code>m</code> z elementów listy inicjalizacyjnej, zawierający <code>n</code> kubelków, funkcję mieszającą <code>hf</code> , funkcję porównującą <code>eql</code> oraz alokator <code>a</code>
<code>unordered_map m {{elem},n,hf,eql}</code>	<code>unordered_map m {{elem},n,hf,eql,allocator_type{}}</code>
<code>unordered_map m {{elem},n,hf}</code>	<code>unordered_map m {{elem},n,hf,key_equal{}}</code>
<code>unordered_map m {{elem},n}</code>	<code>unordered_map m {{elem},n,hasher{}}</code>
<code>unordered_map m {{elem}}</code>	<code>unordered_map m {{elem},N}</code> ; liczba kubelków <code>N</code> zależy od implementacji

W tym przypadku początkowe elementy są pobierane z listy inicjalizacyjnej `{}`. Liczba elementów w słowniku będzie równa liczbie elementów na tej liście.

W końcu słownik `unordered_map` zawiera konstruktory kopujące i przenoszące oraz równoważne konstruktory dostarczające alokatory:

Konstruktory <code>unordered_map<K,T,H,E,A></code> (iso.23.5.4)	
<code>unordered_map m {m2}</code>	Konstruktory kopiujące i przenoszące: tworzą <code>m</code> z <code>m2</code>
<code>unordered_map m {a}</code>	Domyślnie tworzy <code>m</code> i dodaje mu alokator <code>a</code> ; <code>explicit</code>
<code>unordered_map m {m2,a}</code>	Tworzy <code>m</code> z <code>m2</code> i dodaje mu alokator <code>a</code>

Należy uważać, gdy tworzy się słownik `unordered_map` przy użyciu jednego lub dwóch argumentów. Duża liczba możliwych kombinacji typów sprawia, że mogą powstawać dziwne błędy. Na przykład:

```
map<string,int> m {My_comparator};           // OK
unordered_map<string,int> um {My_hasher}; // błąd
```

Jedyny argument konstruktora musi być innym słownikiem `unordered_map` (dotyczy zarówno konstruktora kopiującego, jak i przenoszącego), liczbą kubeliów lub alokatorem. Powyższy błąd można poprawić tak:

```
unordered_map<string,int> um {100,My_hasher}; // OK
```

31.4.3.4. Funkcje mieszająca i porównująca

Oczywiście użytkownik może zdefiniować funkcję mieszającą i może to zrobić na kilka sposobów, przy czym każda technika służy do czego innego. Poniżej przedstawiam kilka wersji. Zacznę od najbardziej bezpośredniej, a skończę na najprostszej. Mamy dany prosty typ o nazwie `Record`:

```
struct Record {
    string name;
    int val;
};
```

Operacje mieszania i porównywania dla `Record` można zdefiniować w następujący sposób:

```
struct Nocase_hash {
    int d = 1; // przesywa kod d o pewną liczbę bitów w każdej iteracji
    size_t operator()(const Record& r) const
    {
        size_t h=0;
        for (auto x : r.name) {
            h <= d;
            h ^= toupper(x);
        }
        return h;
    }
};

struct Nocase_equal {
    bool operator()(const Record& r,const Record& r2) const
    {
        if (r.name.size()!=r2.name.size()) return false;
        for (int i = 0; i<r.name.size(); ++i)
            if (toupper(r.name[i])!=toupper(r2.name[i]))
                return false;
        return true;
    }
};
```

Teraz mogę definiować nieuporządkowane zbiory rekordów i używać ich w programie:

```
unordered_set<Record,Nocase_hash,Nocase_equal> m {
    { {"andrzej", 7}, {"robert",9}, {"wojtek",-3}, {"barbara",12} },
    20, /* liczba kubelków */
    Nocase_hash{2},
    Nocase_equal{}
};

for (auto r : m)
    cout << "{" << r.name << ',' << r.val << "}\n";
```

Gdybym chciał użyć domyślnych funkcji mieszającej i porównującej, co zdarza się bardzo często, to po prostu nie podałbym odpowiadających im argumentów wywołania konstruktora. Standardowo zbiór `unordered_set` używa domyślnych wersji:

```
unordered_set<Record,Nocase_hash,Nocase_equal> m {
    { {"andrzej", 7}, {"robert",9}, {"wojtek",-3}, {"barbara",12} }
    // użycie 4 kubelków, Nocase_hash{} oraz Nocase_equal{}
};
```

Najłatwiejszym sposobem na napisanie funkcji mieszającej często jest użycie standardowych funkcji mieszających dostarczonych jako specjalizacje hash (31.4.3.2). Na przykład:

```
size_t hf(const Record& r) { return hash<string>()(r.name)^hash<int>()(r.val); }

bool eq (const Record& r, const Record& r2) { return r.name==r2.name && r.val==r2.val; }
```

Łączenie wartości mieszania za pomocą lub wykluczającego (^) pozwala zachować ich rozkład w zbiorze wartości typu `size_t` (3.4.5, 10.3.1).

Mając te funkcje mieszania i porównywania, możemy zdefiniować zbiór `unordered_set`:

```
unordered_set<Record,decltype(&hf),decltype(&eq)> m {
    { {"andrzej", 7}, {"robert",9}, {"wojtek",-3}, {"barbara",12} },
    20, /* liczba kubelków */
    hf,
    eq
};

for (auto r : m)
    cout << "{" << r.name << ',' << r.val << "}\n";
```

Użyłem `decltype`, aby uniknąć konieczności powtarzania typów `hf` i `eq`.

Jeśli nie ma dostępnej listy inicjalizacyjnej, można zamiast niej podać rozmiar początkowy:

```
unordered_set<Record,decltype(&hf),decltype(&eq)> m {10,hf,eq};
```

To ułatwia skupienie się na operacjach mieszania i porównywania.

Gdybyśmy nie chcieli oddzielać definicji `hf` i `eq` od miejsca ich użycia, moglibyśmy użyć lambd:

```
unordered_set<Record,                                     // typ wartości
             function<size_t(const Record&)>,           // typ funkcji mieszającej
             function<bool(const Record&,const Record&)> // typ funkcji porównującej
```

```
>m{10,
[](const Record& r) { return hash<string>{}(r.name)^hash<int>{}(r.val); },
[](const Record& r, const Record& r2) { return r.name==r2.name && r.val==r2.val; }
};
```

Lambda (zarówno nazwanych, jak i bez nazwy) używa się zamiast funkcji dlatego, że można je definiować lokalnie w funkcjach — w pobliżu miejsca, w którym są używane.

Ale w tym przypadku `function` może spowodować narzut, którego woałbym uniknąć, gdyby zbiór był intensywnie wykorzystywany. Ponadto przedstawiona wersja wydaje mi się nieporządną i dlatego woałbym nadać lambdom nazwy:

```
auto hf = [](const Record& r) { return hash<string>{}(r.name)^hash<int>{}(r.val); };
auto eq = [](const Record& r, const Record& r2) { return r.name==r2.name && r.val==r2.val; };

unordered_set<Record, decltype(hf), decltype(eq)> m {10,hf,eq};
```

Operacje mieszania i porównywania można też zdefiniować dla wszystkich nieuporządkowanych kontenerów rekordów. W tym celu należy utworzyć specjalizacje standardowych szablonów `hash` i `equal_to` wykorzystywanych przez `unordered_map`:

```
namespace std {
    template<>
    struct hash<Record>{
        size_t operator()(const Record &r) const
        {
            return hash<string>{}(r.name)^hash<int>{}(r.val);
        }
    };

    template<>
    struct equal_to<Record> {
        bool operator()(const Record& r, const Record& r2) const
        {
            return r.name==r2.name && r.val==r2.val;
        }
    };
}

unordered_set<Record> m1;
unordered_set<Record> m2;
```

Domyślny szablon `hash` i otrzymywane przy jego użyciu wartości mieszania tworzone za pomocą operacji „lub wykluczającego” są często bardzo dobre. Zanim ruszysz do pisania własnej funkcji mieszającej, sprawdź, czy gotowe domyślne rozwiązanie nie jest wystarczająco dobre.

31.4.3.5. Poziom zapełnienia i kubełki

Znaczne części implementacji kontenerów nieuporządkowanych są widoczne dla programisty. Jeśli dwa klucze mają tę samą wartość mieszania, to mówi się, że trafiły do „tego samego kubełka” (31.2.1). Programista może badać i ustawać rozmiar tablicy mieszania (zwany „liczbą kubełków”):

Zasady mieszania (iso.23.2.5)	
<code>h=c.hash_function()</code>	<code>h</code> jest funkcją mieszącą kontenera <code>c</code>
<code>eq=c.key_eq()</code>	<code>eq</code> jest funkcją porównującą kontenera <code>c</code>
<code>d=c.load_factor()</code>	<code>d</code> jest liczbą elementów podzieloną przez liczbę kubeliów: <code>double(c.size())/c.bucket_count(); noexcept</code>
<code>d=c.max_load_factor()</code>	<code>d</code> jest maksymalnym współczynnikiem wypełnienia <code>c</code> ; <code>noexcept</code>
<code>c.max_load_factor(d)</code>	Ustawia maksymalny współczynnik zapełnienia kontenera <code>c</code> na <code>d</code> ; jeżeli współczynnik zapełniania <code>c</code> jest bliski maksimum, <code>c</code> zwiększa rozmiar tablicy mieszania (zwiększa liczbę kubeliów)
<code>c.rehash(n)</code>	Ustawia liczbę kubeliów <code>c</code> na <code>>=n</code>
<code>c.reserve(n)</code>	Robi miejsce dla <code>n</code> wpisów (uwzględniając współczynnik zapełnienia): <code>c.rehash(ceil(n/c.max_load_factor()))</code>

Współczynnik zapełnienia nieuporządkowanego kontenera asocjacyjnego to po prostu stopień, w jakim zajęta jest dostępna pojemność. Jeśli na przykład pojemność wynosi 100 elementów, a rozmiar wynosi 30, to współczynnik zapełnienia wynosi 0,3.

Pamiętaj, że ustawianie współczynnika `max_load_factor` przy użyciu funkcji `rehash()` lub `reserve()` może być bardzo kosztowne (w najbardziej pesymistycznym przypadku $O(n^2)$), ponieważ mogą — i zazwyczaj tak jest — powodować ponowne obliczanie wartości mieszania dla wszystkich elementów. Funkcji tych używa się w celu przeprowadzenia ponownego mieszania w dogodnym momencie w czasie wykonywania programu. Na przykład:

```
unordered_set<Record, [](const Record& r) { return hash(r.name); }> people;
//...
constexpr int expected = 1000000; // oczekiwana maksymalna liczba elementów
people.max_load_factor(0.7);    // stopień zapełnienia przynajmniej 70%
people.reserve(expected);       // około 1 430 000 kubeliów
```

Najlepszy współczynnik zapełnienia dla określonego zbioru elementów i konkretnej funkcji mieszącej można odkryć drogą eksperymentów, chociaż często dobrym wyborem jest 70% (0.7).

Interfejs kubeliów (iso.23.2.5)	
<code>n=c.bucket_count()</code>	<code>n</code> jest liczbą kubeliów w <code>c</code> (rozmiar tablicy mieszania); <code>noexcept</code>
<code>n=c.max_bucket_count()</code>	<code>n</code> jest największą możliwą liczbą elementów w kubelku; <code>noexcept</code>
<code>m=c.bucket_size(n)</code>	<code>m</code> jest liczbą elementów w <code>n</code> -tym kubelku
<code>i=c.bucket(k)</code>	Element o kluczu <code>k</code> znajdowałby się w <code>i</code> -tym kubelku
<code>p=c.begin(n)</code>	<code>p</code> wskazuje pierwszy element w kubelku <code>n</code>
<code>p=c.end(n)</code>	<code>p</code> wskazuje za ostatni element w kubelku <code>n</code>
<code>p=c.cbegin(n)</code>	<code>p</code> wskazuje pierwszy element w kubelku <code>n</code> ; <code>p</code> jest iteratorem <code>const</code>
<code>p=c.cend(n)</code>	<code>p</code> wskazuje pierwszy element w kubelku <code>n</code> ; <code>p</code> jest iteratorem <code>const</code>

Użycie wartości `n`, dla której `c.max_bucket_count()<=n`, jako indeksu do kubelka jest niezdefiniowane (i może być fatalne w skutkach).

Jednym ze sposobów wykorzystania interfejsu kubelków jest eksperymentowanie z funkcjami mieszającymi: słaba funkcja mieszająca spowoduje powstawanie dużych kubelków dla niektórych kluczy, czyli sprawi, że jednej wartości mieszającej będzie przyporządkowanych wiele kluczy.

31.5. Adaptacje kontenerów

Adaptacja kontenera to kontener udostępniający inny (zwykle ograniczony) interfejs do kontenera. Adaptacji kontenerów można używać tylko poprzez ich specjalne interfejsy. Znajdujące się w bibliotece STL adaptacje kontenerów nie dają bezpośredniego dostępu do kontenerów, na których bazują. Nie udostępniają iteratorów ani indeksów.

Techniki tworzenia adaptacji kontenerów z kontenerów są ogólnie przydatne do nieinwazyjnego dostosowywania interfejsu klasy do potrzeb użytkowników.

31.5.1. Stos

Adaptacja kontenera `stack` (`stos`) jest zdefiniowana w nagłówku `<stack>`. W celu jej opisania wystarczy przedstawić część jej implementacji:

```
template<typename T, typename C = deque<T>>
class stack { // iso.23.6.5.2
public:
    using value_type = typename C::value_type;
    using reference = typename C::reference;
    using const_reference = typename C::const_reference;
    using size_type = typename C::size_type;
    using container_type = C;
public:
    explicit stack(const C&); // kopiuje z kontenera
    explicit stack(C&& = C{}); // przenosi z kontenera

    // domyślne kopiowanie, przeniesienie, przypisanie i destruktor

    template<typename A>
    explicit stack(const A& a); // domyślny kontener, alokator a
    template<typename A>
    stack(const C& c, const A& a); // elementy z c, alokator a
    template<typename A>
    stack(C&&, const A&);
    template<typename A>
    stack(const stack&, const A&);
    template<typename A>
    stack(stack&&, const A&);

    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    reference top() { return c.back(); }
    const_reference top() const { return c.back(); }
    void push(const value_type& x) { c.push_back(x); }
    void push(value_type&& x) { c.push_back(std::move(x)); }
    void pop() { c.pop_back(); } // usuwa ostatni element

    template<typename... Args>
```

```

void emplace(Args&&... args)
{
    c.emplace_back(std::forward<Args>(args)...);
}

void swap(stack& s) noexcept(noexcept(swap(c, s.c)))
{
    using std::swap; // pamiętaj, by użyć standardowej funkcji swap()
    swap(c,s.c);
}
protected:
    Cc;
};

```

Jak widać, stos jest interfejsem do kontenera typu przekazanego jako argument szablonu. Usunięto niestosowe operacje podstawowego kontenera oraz zdefiniowano konwencjonalne nazwy: `top()`, `push()` oraz `pop()`.

Ponadto stos udostępnia typowe operatory porównywania (`==`, `<` itd.) oraz niebędącą składową funkcję `swap()`.

Domyślnie stos przechowuje elementy w kolejce `deque`, ale do tego celu można użyć do wolnej sekwencji udostępniającej operacje `back()`, `push_back()` oraz `pop_back()`. Na przykład:

```

stack<char> s1;           // przechowuje elementy w deque<char>
stack<int,vector<int>> s2; // przechowuje elementy w vector<int>

```

Kontener `vector` jest często szybszy od `deque` i zużywa mniej pamięci.

Elementy do stosu dodaje się za pomocą operacji `push_back()` działającej na podstawowym kontenerze. W konsekwencji stos nie może ulec „przepełnieniu”, dopóki w komputerze jest dostępna do zajęcia pamięć. Stos może natomiast być niedopełniony:

```

void f()
{
    stack<int> s;
    s.push(2);
    if (s.empty()) { // niedopełnieniu można zapobiec
        // nie usuwać
    }
    else {          // ale jest ono możliwe
        s.pop();    // w porządku: s.size() wynosi 0
        s.pop();    // nie wiadomo, jaki będzie tego skutek, ale pewnie nieprzyjemny
    }
}

```

Nie wywołuje się funkcji `pop()` na używanym elemencie. Zamiast tego pobiera się pierwszy element za pomocą funkcji `top()`, a następnie usuwa się go przy użyciu `pop()`, gdy jest już niepotrzebny. Jest to w miarę wygodne, może być bardziej efektywne, gdy nie trzeba używać funkcji `pop()`, oraz znacznie upraszcza implementację gwarancji wyjątków. Na przykład:

```

void f(stack<char>& s)
{
    if (s.top()=='c') s.pop(); // opcjonalnie usuwa opcjonalne początkowe 'c'
    //...
}

```

Domyślnie stos wykorzystuje alokator swojego podstawowego kontenera. Jeśli jednak ten jest niewystarczający, dostępny jest kilka konstruktorów umożliwiających przekazanie innego.

31.5.2. Kolejka

Kolejka (`queue`) jest zdefiniowana w nagłówku `<queue>`. Adaptacja ta jest interfejsem do kontenera umożliwiającego wstawianie elementów na końcu i pobieranie elementów z początku:

```
template<typename T, typename C = deque<T>>
class queue { //iso.23.6.3.1
    //...jak stos ...
    void pop() { c.pop_front(); } //pobiera pierwszy element
};
```

Kolejki są używane chyba w każdym systemie. Na przykład serwer prostego systemu obsługi komunikatów można zdefiniować tak:

```
void server(queue<Message>& q, mutex& m)
{
    while (!q.empty()) {
        Message mess;
        { lock_guard<mutex> lck(m); //blokada na czas pobierania wiadomości
            if (q.empty()) return; //ktos inny otrzymał wiadomość
            mess = q.front();
            q.pop();
        }
        //obsługa żądania
    }
}
```

31.5.3. Kolejka priorytetowa

Kolejka priorytetowa (`priority_queue`) to kolejka, w której każdy element ma określony priorytet decydujący o jego miejscu na drodze do początku kolejki. Deklaracja kolejki priorytetowej jest bardzo podobna do deklaracji zwykłej kolejki, tylko zawiera dodatkowe narzędzia do obsługi obiektu porównującego oraz konstruktory inicjujące z sekwencji:

```
template<typename T, typename C = vector<T>, typename Cmp = less<typename C::value_type>>
class priority_queue { //iso.23.6.4
protected:
    Cc;
    Cmp comp;
public:
    priority_queue(const Cmp& x, const C&);
    explicit priority_queue(const Cmp& x = Cmp{}, C& = C{});
    template<typename In>
    priority_queue(In b, In e, const Cmp& x, const C& c); //ustawia <b,e> do c
    //...
};
```

Deklaracja kolejki `priority_queue` znajduje się w nagłówku `<queue>`.

Domyślnie kolejka `priority_queue` po prostu porównuje elementy przy użyciu operatora `<`, a jej funkcja `top()` zwraca największy element:

```
struct Message {
    int priority;
    bool operator<(const Message& x) const { return priority < x.priority; }
    //...
};
```

```

void server(priority_queue<Message>& q, mutex& m)
{
    while (!q.empty()) {
        Message mess;
        { lock_guard<mutex> lck(m); // blokada na czas pobierania komunikatu
          if (q.empty()) return;      // ktoś inny otrzymał wiadomość
          mess = q.top();
          q.pop();
        }
        // obsługa żądania o najwyższym priorytecie
    }
}

```

W tym przypadku różnica między kolejką priorytetową a zwykłą kolejką (31.5.2) jest taka, że najpierw zostaną obsłużone komunikaty o najwyższym priorytecie. Kolejność dochodzenia do początku kolejki elementów o takim samym priorytecie jest niezdefiniowana. Dwa elementy uznawane są za równoważne pod względem priorytetu, gdy priorytet żadnego z nich nie jest wyższy od drugiego (31.2.2.1).

Utrzymywanie porządku wśród elementów nie jest darmowe, ale też nie musi być bardzo kosztowne. Jednym ze sposobów implementacji kolejki priorytetowej jest wykorzystanie struktury drzewiastej przechowującej informacje o względnym położeniu elementów. To daje złożoność obliczeniową $O(\log(n))$ operacjom `push()` i `pop()`. Kolejki priorytetowe prawie zawsze są implementowane przy użyciu sterty (32.6.4).

31.6. Rady

1. Kontenery STL definiują sekwencje — 31.2.
2. Używaj wektora jako domyślnego kontenera — 31.2, 31.4.
3. Operatory wstawiania, np. `insert()` i `push_back()`, często wydajniej działają na wektorach niż na listach — 31.2, 31.4.1.1.
4. Do reprezentacji sekwencji, które przez większość czasu są puste, używaj struktury `forward_list` — 31.2, 31.4.2.
5. W kwestiach wydajności nie ufaj intuicji, tylko wykonuj testy — 31.3.
6. Nie wierz ślepo asymptotycznym miarom złożoności. Niektóre sekwencje są krótkie, przez co koszty poszczególnych operacji mogą być dramatycznie różne — 31.3.
7. Kontenery biblioteki STL są uchwytemi do zasobów — 31.2.1.
8. Słownik `map` jest zazwyczaj implementowany jako drzewo czerwono-czarne — 31.2.1, 31.4.3.
9. Słownik `unordered_map` jest tablicą mieszającą — 31.2.1, 31.4.3.2.
10. Typy elementów kontenerów STL muszą udostępniać operacje kopiowania lub przenoszenia — 31.2.2.
11. Jeśli chcesz zachować polimorficzność, używaj kontenerów wskaźników lub wskaźników inteligentnych — 31.2.2.
12. Operacje porównywania powinny implementować ścisły porządek słaby — 31.2.2.1.
13. Kontenery przekazuj przez referencję, a zwracaj przez wartość — 31.3.2.
14. Używaj notacji `()` dla rozmiarów oraz `{}` dla list elementów — 31.3.2.
15. Do prostego przeglądania kontenerów używaj zakresowej pętli `for` lub `par` iteratorów — 31.3.4.
16. Jeśli nie musisz modyfikować elementów kontenera, przeglądaj go przy użyciu iteratorów stałych — 31.3.4.

17. Używaj słowa kluczowego `auto`, aby uniknąć rozwlekłości i literówek podczas używania iteratorów — 31.3.4.
18. Używaj funkcji `reserve()`, aby zapobiec uszkodzeniu wskaźników i iteratorów do elementów — 31.3.3, 31.4.1.
19. Nie zakładaj, że użycie funkcji `reserve()` będzie korzystne pod względem wydajnościowym, tylko wykonaj testy, aby to sprawdzić — 31.3.3.
20. Używaj funkcji `push_back()` i `resize()` na kontenerach zamiast `realloc()` na tablicach — 31.3.3, 31.4.1.1.
21. Nie używaj iteratorów wskazujących elementy w wektorach i kolejkach `deque` o zmienionym rozmiarze — 31.3.3.
22. W razie konieczności użyj funkcji `reserve()`, aby uczynić wydajność przewidywalną — 31.3.3.
23. Nie zakładaj, że operator `[]` wykonuje sprawdzanie zakresu — 31.2.2.
24. Jeśli potrzebujesz gwarancji sprawdzania zakresu, użyj funkcji `at()` — 31.2.2.
25. Używaj funkcji `emplace()` jako wygodnej formy zapisu — 31.3.7.
26. Preferuj kompaktowe i ciągłe struktury danych — 31.4.1.2.
27. Używaj funkcji `emplace()` w celu uniknięcia wstępnej inicjacji elementów — 31.4.1.3.
28. Przeglądanie list `list` jest względnie powolne — 31.4.2.
29. Lista `list` zazwyczaj dokłada dodatkowe cztery słowa narzutu na element — 31.4.2.
30. Porządek elementów w kontenerze uporządkowanym jest zdefiniowany przez jego obiekt porównawczy (domyślnie `<`) — 31.4.3.1.
31. Porządku elementów w kontenerze nieuporządkowanym (mieszającym) nie da się przewidzieć — 31.4.3.2.
32. Jeśli chcesz moc szybko wyszukiwać dane w dużych zbiorach danych, użyj kontenera nieuporządkowanego — 31.3.
33. Używaj nieuporządkowanych kontenerów do przechowywania elementów nie mających naturalnego porządku (np. dla których nie da się sensownie zdefiniować operacji `<`) — 31.4.3.
34. Jeśli chcesz przeglądać elementy w określonym porządku, użyj uporządkowanego kontenera asocjacyjnego (np. `map` lub `set`) — 31.4.3.2.
35. Sprawdź eksperymentalnie, czy Twoja funkcja mieszająca działa prawidłowo — 31.4.3.4.
36. Często dobrze sprawdzają się funkcje mieszające tworzone z połączenia standardowych funkcji mieszających dla elementów przy użyciu lub wykluczającego — 31.4.3.4.
37. Często dobrym współczynnikiem zapełnienia jest 0.7 — 31.4.3.5.
38. Interfejsom można definiować alternatywne interfejsy — 31.5.
39. Adaptacje kontenerów STL nie umożliwiają bezpośredniego dostępu do kontenerów, na podstawie których są zbudowane — 31.5.

Algorytmy STL

*Forma cię wyzwoli
— powiedzenie inżynierskie*

- Wprowadzenie
- Algorytmy
 - Sekwencje; Argumenty zasad; Złożoność
- Algorytmy nie modyfikujące sekwencji
 - for_each(); Predykaty sekwencji; count(); find(); equal() i mismatch(); search()
- Algorytmy modyfikujące sekwencje
 - copy(); unique(); remove() i replace(); rotate(), random_shuffle() i partition(); Permutacje; fill(); swap()
- Sortowanie i wyszukiwanie
 - Wyszukiwanie binarne; merge(); Algorytmy działające na zbiorach; Sterty; lexicographical_compare()
- Element minimalny i maksymalny
- Rady

32.1. Wprowadzenie

W tym rozdziale znajduje się opis algorytmów biblioteki STL. Biblioteka ta obejmuje część biblioteki standardowej zawierającą iteratory, kontenery, algorytmy oraz obiekty funkcyjne. Pozostała część biblioteki STL jest opisana w rozdziałach 31. i 33.

32.2. Algorytmy

W nagłówku `<algorithm>` znajdują się definicje około 80 standardowych algorytmów działających na **sekwencjach** definiowanych przez pary iteratorów (dla wejścia) lub pojedyncze operatory (dla wyjścia). Przy kopiowaniu, porównywaniu itd. dwóch sekwencji, pierwszą reprezentuje para iteratorów, $\langle b, e \rangle$, a drugą tylko jeden iterator, b_2 , oznaczający początek sekwencji o odpowiedniej długości, np. zawierającej tyle elementów co pierwsza sekwencja: $\langle b_2, b_2 + (e - b) \rangle$. Niektóre algorytmy, np. `sort()`, wymagają iteratorów o dostępie swobodnym, a inne, np. `find()`, przeglądają sekwencje po kolej, więc wystarcza im iterator jednokierunkowy. Wiele algorytmów fakt nieznalezienia elementu standardowo oznacza zwróceniem końca sekwencji (4.5). Nie przypominam o tym w opisie każdego algorytmu.

Algorytmy, zarówno standardowe, jak i definiowane przez użytkownika, są niezmiernie ważne:

- Każdy algorytm nadaje nazwę jakiejś konkretnej operacji, dokumentuje jakiś interfejs oraz określa jakąś semantykę.
- Każdy algorytm może być powszechnie wykorzystywany i znany przez wielu programistów.

Te cechy sprawiają, że pod względem wydajności, poprawności i łatwości obsługi kodu algorytmy znacznie przewyższają zwykły kod zawierający mniej konkretnie określone funkcje i zależności. Jeśli piszesz kod zawierający kilka pętli, lokalne zmienne nie mające ze sobą nic wspólnego albo skomplikowane konstrukcje sterujące, zastanów się, czy nie dałoby się go uprościć poprzez umieszczenie w funkcji/algorytmie o dobrze dobranej nazwie, zdefiniowanym przeznaczeniu i interfejsie oraz dobrze zdefiniowanych zależnościach.

Algorytmy numeryczne napisane w stylu algorytmów biblioteki STL są przedstawione w podrozdziale 40.6.

32.2.1. Sekwencje

Idealny algorytm biblioteki standardowej powinien udostępniać jak najbardziej ogólny i elastyczny interfejs do czegoś, co można optymalnie zaimplementować. Interfejsy oparte na iteratorach (33.1.1) są bliskie temu ideałowi, ale nie osiągają go w pełni. Na przykład interfejs oparty na iteratorach nie reprezentuje bezpośrednio pojęcia sekwencji, co może wywoływać nieporozumienia i utrudniać wykrywanie niektórych błędów dotyczących zakresu:

```
void user(vector<int>& v1, vector<int>& v2)
{
    copy(v1.begin(),v1.end(),v2.begin()); // ryzyko przepelenienia v2
    sort(v1.begin(),v2.end());           // ups!
}
```

Wiele tego typu problemów można zmniejszyć poprzez dostarczenie kontenerowych wersji algorytmów z biblioteki standardowej. Na przykład:

```
template<typename Cont>
void sort(Cont& c)
{
    static_assert(Range<Cont>(), "sort(): argument Cont nie jest zakresem");
    static_assert(Sortable<Iterator<Cont>>(), "sort(): argument Cont nie jest sortowalny");

    std::sort(begin(c),end(c));
}

template<typename Cont1, typename Cont2>
void copy(const Cont1& source, Cont2& target)
{
    static_assert(Range<Cont1>(), "copy(): argument Cont1 nie jest zakresem");
    static_assert(Range<Cont2>(), "copy(): argument Cont2 nie jest zakresem");
    if (target.size()<source.size()) throw out_of_range("obiekt docelowy jest za mały");

    std::copy(source.begin(),source.end(),target.begin());
}
```

To uprościłoby definicję funkcji `user()`, uniemożliwiłoby wyrażenie drugiego błędu oraz pozwoliłoby przechwycić pierwszy błąd w czasie wykonywania:

```
void user(vector<int>& v1, vector<int>& v2)
{
    copy(v1,v2); // przepelnienia zostaną wykryte
    sort(v1);
}
```

Ale wersje kontenerowe są mniej ogólne od wersji korzystających bezpośrednio z iteratorów. Nie można na przykład użyć kontenerowego algorytmu `sort()` do posortowania połowy kontenera i nie można użyć kontenerowej wersji algorytmu `copy()` do zapisu w strumieniu wyjściowym.

Uzupełniającym podejściem jest zdefiniowanie abstrakcji „zakresu” lub „sekwencji”, aby móc definiować sekwencje, gdy są potrzebne. Do oznaczania wszystkiego, co ma iteratory `begin()` i `end()`, używam koncepcji Range (24.4.4). To znaczy, że w bibliotece STL nie ma klasy `Range` zawierającej dane, podobnie jak nie ma klas `Iterator` ani `Container`. W przykładach „kontenerowego algorytmu `sort()`” i „kontenerowego algorytmu `copy()`” argument szablonowy nazwałem `Cont` (co oznacza kontener), ale tak naprawdę można przekazać dowolną sekwencję z iteratorami `begin()` i `end()` spełniającą wszystkie pozostałe wymogi algorytmu.

Algorytmy z biblioteki standardowej najczęściej zwracają iteratory, nie zaś kontenery wyników (z wyjątkiem paru rzadkich przypadków, takich jak `pair`). Jednym z powodów takiego sposobu ich działania jest to, że gdy projektowano bibliotekę STL, nie było jeszcze bezpośredniej obsługi semantyki przenoszenia, a więc nie istniał oczywisty i wydajny sposób na zwracanie dużych ilości danych przez algorytm. Niektórzy programiści wprost posługiwali się pośredniością (np. wskaźnikami, referencjami lub iteratorami) albo stosowali sprytne sztuczki. Dziś możemy to zrobić lepiej:

```
template<typename Cont, typename Pred>
vector<Value_type<Cont>*>
find_all(Cont& c, Pred p)
{
    static_assert(Range<Cont>(), "find_all(): argument Cont nie jest zakresem");
    static_assert(Predicate<Pred>(), "find_all(): argument Pred nie jest predykatem");

    vector<Value_type<Cont>*> res;
    for (auto& x : c)
        if (p(x)) res.push_back(&x);
    return res;
}
```

W C++98 tak zdefiniowana funkcja `find_all()` radykalnie obniżałyby wydajność za każdym razem, gdy liczba dopasowań byłaby duża. Nawet jeśli algorytmy z biblioteki standardowej są zbyt restrykcyjne lub niewystarczające, lepiej jest tworzyć nowe wersje algorytmów STL albo pisać całkiem nowe algorytmy, niż stosować „luźny kod”, aby tylko obejść problem.

Zwróć uwagę, że bez względu na to, co zwraca algorytm STL, nie może to być podany jako argument kontener. Argumentami algorytmów STL są iteratory (rozdział 33.) i algorytmy te nic nie wiedzą o wskazywanych przez te iteratory strukturach danych. Najważniejszą funkcją iteratorów jest oddzielanie algorytmów od struktur danych i odwrotnie.

32.3. Argumenty zasad

Większość algorytmów znajdujących się w bibliotece standardowej występuje w dwóch wersjach:

- Wersja zwykła wykorzystująca do działania zwykłe operacje, np. `< i ==`.
- Wersja pobierająca kluczowe operacje jako argumenty.

Na przykład:

```
template<typename Iter>
void sort(Iter first, Iter last)
{
    //... sortowanie przy użyciu e1 < e2...
}

template<typename Iter, typename Pred>
void sort(Iter first, Iter last, Pred pred)
{
    //... sortowanie przy użyciu pred(e1,e2)...
}
```

To znacznie zwiększa elastyczność biblioteki standardowej i poszerza wachlarz jej zastosowań.

Dwie typowe wersje algorytmu mogą być zaimplementowane jako dwa (przeciążone) szablony funkcji lub jako jeden szablon funkcji z domyślnym argumentem. Na przykład:

```
template<typename Ran, typename Pred = less<Value_type<Ran>>> // używa domyślnego
                                                    // argumentu szablonowego
sort(Ran first, Ran last, Pred pred ={})
{
    //... używa pred(x,y)...
}
```

Różnica między dwiema funkcjami a jedną z domyślnym argumentem może zostać zauważona przez programistę używającego wskaźników do funkcji. Jednak dzięki potraktowaniu wielu wariantów standardowych algorytmów jako „wersji z domyślnym predykatem” sprawia, że mamy do zapamiętania o połowę mniej szablonów funkcji.

W niektórych przypadkach argument można zinterpretować jako predykat lub wartość.
Na przykład:

```
bool pred(int);

auto p = find(b,e,pred); // znajduje element pred czy stosuje pred()?(to drugie)
```

Ogólnie rzecz biorąc, kompilator nie może rozwiklać znaczenia takich przykładów, a nawet gdyby mógł, to myliłyby one programistów.

Aby ułatwić pracę programistom, często używa się przyrostka `_if` oznaczającego, że algorytm przyjmuje predykat. Powodem stosowania dwóch nazw jest chęć minimalizacji niejednoznaczności i nieporozumień. Na przykład:

```
using Predicate = bool(*)(int);

void f(vector<Predicate>& v1, vector<int>& v2)
{
    auto p1 = find(v1.begin(),v1.end(),pred); // znajduje element o wartości pred
    auto p2 = find_if(v2.begin(),v2.end(),pred); // liczy, dla ilu elementów pred() zwraca true
}
```

Niektóre operacje przekazywane jako argumenty do algorytmu mają za zadanie modyfikować elementy [np. niektóre operacje przekazywane do `for_each()` (32.4.1)], ale większość to predykaty (np. porównywanie obiektów dla algorytmu `sort()`). Jeśli nie zaznaczono inaczej, należy zakładać, że przekazany do algorytmu argument zasady nie powinien modyfikować elementów. W szczególności nie należy próbować modyfikować elementów poprzez predykaty:

```
int n_even(vector<int>& v) // nie rób tego
    // liczy parzyste wartości w v
{
    return find_if(v.begin(),v.end(),[](int& x) {++x; return x&1; });
}
```

Modyfikacja elementu poprzez predykat zaciemnia rzeczywisty zamiar programisty. Jeśli jesteś podstępny, możesz nawet zmodyfikować sekwencję (np. wstawiając lub usuwając element przy użyciu nazwy przeglądanego kontenera), aby iteracja nie udała się (zapewne z niewiadomych powodów). Aby uniknąć wypadków, argumenty do predykatów możesz przekazywać przez referencję `const`.

Predykat nie powinien też przenosić stanu zmieniającego znaczenie jego działania. Implementacja algorytmu może kopować predykat i rzadko zdarza się, abyśmy chcieli, żeby predykat użyty wielokrotnie na tej samej wartości dawał różne wyniki. Niektóre obiekty funkcyjne przekazywane do algorytmów, np. generatorów liczb losowych, przenoszą zmienny stan. Jeśli nie masz absolutnej pewności, że algorytm nie kopiuje, przechowuj zmienny stan argumentu obiektu funkcyjnego w osobnym obiekcie i sięgaj do niego za pomocą wskaźnika lub referencji.

Operacje `==` i `<` na wskaźnikach są rzadko potrzebne w algorytmach STL, ponieważ porównują adresy maszynowe, a nie przekazywane wartości. W szczególności nie należy sortować ani przeszukiwać kontenerów łańcuchów w stylu języka C przy użyciu domyślnych operatorów `==` i `<` (32.6).

32.3.1. Złożoność

Złożoność algorytmów, podobnie jak kontenerów (31.3), jest określona w standardzie. Większość algorytmów ma złożoność liniową, $O(n)$, dla n , które najczęściej jest długością sekwencji wejściowej.

Złożoność algorytmów (iso.25)	
$O(1)$	<code>swap()</code> , <code>iter_swap()</code>
$O(\log(n))$	<code>lower_bound()</code> , <code>upper_bound()</code> , <code>equal_range()</code> , <code>binary_search()</code> , <code>push_heap()</code> , <code>pop_heap()</code>
$O(n \cdot \log(n))$	<code>inplace_merge()</code> (najbardziej pesymistyczny przypadek), <code>stable_partition()</code> (najbardziej pesymistyczny przypadek), <code>sort()</code> , <code>stable_sort()</code> , <code>partial_sort()</code> , <code>partial_sort_copy()</code> , <code>sort_heap()</code>
$O(n^2)$	<code>find_end()</code> , <code>find_first_of()</code> , <code>search()</code> , <code>search_n()</code>
$O(n)$	Wszystkie pozostałe

Jak zwykle podane są wartości asymptotyczne, a więc aby z podanych danych wyciągnąć wartościowe wnioski, trzeba znać rozmiar n . Jeśli na przykład $n < 3$, najlepszym wyborem może być algorytm o złożoności kwadratowej. Koszt każdej iteracji może być radykalnie różny. Na przykład przeglądanie listy może być znacznie wolniejsze od przeglądania wektora, mimo że złożoność obliczeniowa obu tych struktur jest liniowa ($O(n)$). Obliczenia złożoności nie następują zdrowego rozsądku i realnych pomiarów czasu wykonywania. Są jedynie jednym z wielu narzędzi pozwalających zapewnić wysoką jakość kodu.

32.4. Algorytmy nie modyfikujące sekwencji

Algorytm nie modyfikujący odczytuje wartości elementów podanej mu sekwencji. Nie zmienia kolejności ani wartości tych elementów. Zazwyczaj operacje przekazywane do algorytmu przez użytkownika również nie zmieniają wartości elementów. Najczęściej są to predykaty (które nie mogą zmieniać swoich argumentów).

32.4.1. `for_each()`

Najprostszym algorytmem jest `for_each()`, który wykonuje określoną operację na każdym elemencie sekwencji:

<code>for_each()</code> (iso.25.2.4)
<code>f=for_each(b,e,f)</code> Wykonuje <code>f(x)</code> na każdym <code>x</code> z przedziału <code><b,e></code> ; zwraca <code>f</code>

W razie możliwości lepiej jest używać bardziej wyspecjalizowanych algorytmów.

Przekazana do `for_each()` operacja może modyfikować elementy. Na przykład:

```
void increment_all(vector<int>& v) // zwiększa każdy element kontenera v
{
    for_each(v.begin(),v.end(), [](int& x) {++x;});
}
```

32.4.2. Predykaty sekwencji

Predykaty sekwencji (iso.25.2.1)	
<code>all_of(b,e,f)</code>	Czy <code>f(x)</code> wynosi true dla wszystkich <code>x</code> w <code><b,e></code> ?
<code>any_of(b,e,f)</code>	Czy <code>f(x)</code> wynosi true dla wszystkich <code>x</code> w <code><b,e></code> ?
<code>none_of(b,e,f)</code>	Czy <code>f(x)</code> wynosi false dla wszystkich <code>x</code> w <code><b,e></code> ?

Na przykład:

```
vector<double> scale(const vector<double>& val, const vector<double>& div)
{
    assert(val.size()<div.size());
    assert(all_of(div.begin(),div.end(),[] (double x){ return 0<x; }));

    vector res(val.size());
    for (int i = 0; i<val.size(); ++i)
        res[i] = val[i]/div[i];
    return res;
}
```

W razie niepowodzenia predykaty te nie informują, który element spowodował awarię.

32.4.3. `count()`

<code>count()</code> (iso.25.2.9)
<code>x=count(b,e,v)</code> x jest liczbą elementów *p w <code><b,e></code> , tak że <code>v==*p</code>
<code>x=count_if(b,e,f)</code> x jest liczbą elementów *p w <code><b,e></code> , tak że <code>f(*p)</code>

Na przykład:

```
void f(const string& s)
{
    auto n_space = count(s.begin(), s.end(), ' ');
    auto n_whitespace = count_if(s.begin(), s.end(), isspace);
    //...
}
```

Predykat `isspace()` (36.2) umożliwia policzenie wszystkich białych znaków, nie tylko spacji.

32.4.4. `find()`

Algorytmy z rodziny `find()` wykonują liniowe wyszukiwanie elementów lub wartości spełniających warunek predykatu:

Rodzina <code>find</code> (iso.25.2.5)	
<code>p=find(b,e,v)</code>	<code>p</code> wskazuje pierwszy element sekwencji $\langle b, e \rangle$, tak że $*p == v$
<code>p=find_if(b,e,f)</code>	<code>p</code> wskazuje pierwszy element sekwencji $\langle b, e \rangle$, tak że $f(*p)$
<code>p=find_if_not(b,e,f)</code>	<code>p</code> wskazuje pierwszy element sekwencji $\langle b, e \rangle$, tak że $\neg f(*p)$
<code>p=find_first_of(b,e,b2,e2)</code>	<code>p</code> wskazuje pierwszy element sekwencji $\langle b, e \rangle$, tak że $*p == *q$ dla pewnego <code>q</code> w $\langle b2, e2 \rangle$
<code>p=find_first_of(b,e,b2,e2,f)</code>	<code>p</code> wskazuje pierwszy element sekwencji $\langle b, e \rangle$, tak że $f(*p, *q)$ dla pewnego <code>q</code> w $\langle b2, e2 \rangle$
<code>p=adjacent_find(b,e)</code>	<code>p</code> wskazuje pierwszy element sekwencji $\langle b, e \rangle$, tak że $*p == *(p+1)$
<code>p=adjacent_find(b,e,f)</code>	<code>p</code> wskazuje pierwszy element sekwencji $\langle b, e \rangle$, tak że $f(*p, *(p+1))$
<code>p=find_end(b,e,b2,e2)</code>	<code>p</code> wskazuje ostatni <code>*p</code> w sekwencji $\langle b, e \rangle$, tak że $*p == *q$ dla elementu <code>*q</code> w $\langle b2, e2 \rangle$
<code>p=find_end(b,e,b2,e2,f)</code>	<code>p</code> wskazuje ostatni <code>*p</code> w sekwencji $\langle b, e \rangle$, tak że $f(*p, *q)$ dla elementu <code>*q</code> w $\langle b2, e2 \rangle$

Algorytmy `find()` i `find_if()` zwracają iterator do pierwszego elementu pasującego odpowiednio do wartości i predykatu.

```
void f(const string& s)
{
    auto p_space = find(s.begin(), s.end(), ' ');
    auto p_whitespace = find_if(s.begin(), s.end(), isspace);
    //...
}
```

Algorytmy `find_first_of()` znajdują pierwsze wystąpienie w sekwencji elementu z innej sekwencji. Na przykład:

```
array<int> x = {1,3,4 };
array<int> y = {0,2,3,4,5};
void f()
{
    auto p = find_first_of(x.begin(), x.end(), y.begin(), y.end()); // p = &x[1]
    auto q = find_first_of(p+1, x.end(), y.begin(), y.end()); // q = &x[2]
}
```

Iterator p będzie wskazywał $x[1]$, ponieważ 3 jest pierwszym elementem x mającym odpowiednik w y . Analogicznie q będzie wskazywać $x[2]$.

32.4.5. equal() i mismatch()

Algorytmy `equal()` i `mismatch()` porównują pary sekwencji:

<code>equal()</code> i <code>mismatch()</code> (iso.25.2.11, 25.2.10)	
<code>equal(b,e,b2)</code>	Czy $v==v2$ dla wszystkich odpowiadających sobie par elementów sekwencji $\langle b,e \rangle$ i $\langle b2,b2+(e-b) \rangle$?
<code>equal(b,e,b2,f)</code>	Czy $f(v,v2)$ dla wszystkich odpowiadających sobie par elementów sekwencji $\langle b,e \rangle$ i $\langle b2,b2+(e-b) \rangle$?
<code>pair(p1,p2)=mismatch(b,e,b2)</code>	$p1$ wskazuje pierwszy element w $\langle b,e \rangle$, a $p2$ pierwszy element w $\langle b2,b2+(e-b) \rangle$, tak że $!(p1==p2)$ lub $p1==e$
<code>pair(p1,p2)=mismatch(b,e,b2,f)</code>	$p1$ wskazuje pierwszy element w $\langle b,e \rangle$, a $p2$ pierwszy element w $\langle b2,b2+(e-b) \rangle$, tak że $!f(p1,p2)$ lub $p1==e$

Algorytm `mismatch()` szuka pierwszej pary elementów dwóch sekwencji, które są różne, i zwraca iteratory do nich. Dla drugiej sekwencji nie podaje się końca, tzn. nie ma `last2`. Przyjęte jest założenie, że druga sekwencja zawiera przynajmniej tyle samo elementów co pierwsza oraz że $first2+(last-first)$ jest używane jako `last2`. Technika ta jest używana w bibliotece standarodowej wszędzie tam, gdzie używane są pary sekwencji do wykonywania operacji na parach elementów. Implementacja `mismatch()` może wyglądać tak:

```
template<typename In, typename In2, typename Pred = equal_to<Value_type<In>>>
pair<In, In2> mismatch(In first, In last, In2 first2, Pred p ={})
{
    while (first != last && p(*first,*first2)) {
        ++first;
        ++first2;
    }
    return {first,first2};
}
```

Użyłem standardowego obiektu funkcyjnego `equal_to` (33.4) i funkcji typowej `Value_type` (28.2.1).

32.4.6. search()

Algorytmy `search()` i `search_n()` znajdują sekwencje w innych sekwencjach:

Szukanie sekwencji (iso.25.2.13)	
<code>p=search(b,e,b2,e2)</code>	p wskazuje pierwszy $*p$ w $\langle b,e \rangle$, tak że $\langle p,p+(e2-b2) \rangle$ równa się $\langle b2,e2 \rangle$
<code>p=search(b,e,b2,e2,f)</code>	p wskazuje pierwszy $*p$ w $\langle b,e \rangle$, tak że $\langle p,p+(e2-b2) \rangle$ równa się $\langle b2,e2 \rangle$, używa f do porównywania elementów
<code>p=search_n(b,e,n,v)</code>	p wskazuje pierwszy element $\langle b,e \rangle$, taki że każdy element $\langle p,p+n \rangle$ ma wartość v
<code>p=search_n(b,e,n,v,f)</code>	p wskazuje pierwszy element $\langle b,e \rangle$, tak że dla każdego elementu $*q$ w $\langle p,p+n \rangle$ mamy $f(*p,v)$

Algorytm `search()` szuka drugiej podanej sekwencji w pierwszej. Jeśli ją znajdzie, zwraca iterator do pierwszego znalezionego elementu w pierwszej sekwencji. Nieznalezienie niczego jest jak zwykle oznaczane zwrotem końca sekwencji. Na przykład:

```
string quote {"Po co się męczyć z nauką, skoro niewiedzę ma się od razu?"};

bool in_quote(const string& s)
{
    auto p = search(quote.begin(), quote.end(), s.begin(), s.end()); // znajduje s w cytacie
    return p!=quote.end();
}

void g()
{
    bool b1 = in_quote("skoro"); // b1 = true
    bool b2 = in_quote("sporo"); // b2 = false
}
```

Zatem `search()` jest ogólnym algorytmem sekwencyjnym do znajdowania podłańcuchów.

Do wyszukiwania pojedynczych elementów służą algorytmy `find()` i `binary_search()` (32.6).

32.5. Algorytmy modyfikujące sekwencje

Algorytmy modyfikujące mogą modyfikować elementy sekwencji przekazanych im jako argumenty.

transform() (iso.25.3.4)	
<code>p=transform(b,e,out,f)</code>	Wykonuje <code>*q=f(*p1)</code> na każdym <code>*p1 w <b,e></code> , zapisując wynik w odpowiednim <code>*q w <out,out+(e-b)></code> ; <code>p=out+(e-b)</code>
<code>p=transform(b,e,b2,out,f)</code>	Wykonuje <code>*q=f(*p1,p2)</code> na każdym elemencie w <code>*p1 w <b,e></code> i odpowiednim <code>*p2 w <b2,b2+(e-b)></code> , zapisując wynik w odpowiednim <code>*q w <out,out+(e-b)></code> ; <code>p=out+(e-b)</code>

To trochę mylące, ale algorytm `transform()` niekoniecznie zmienia sekwencję wejściową. Zamiast tego tworzy sekwencję wyjściową będącą przekształconą wersją wejściowej przy użyciu operacji dostarczonej przez użytkownika. Wersja przyjmująca jedną sekwencję wejściową może mieć następującą definicję:

```
template<typename In, typename Out, typename Op>
Out transform(In first, In last, Out res, Op op)
{
    while (first!=last)
        *res++ = op(*first++);
    return res;
}
```

Sekwencja wyjściowa może być ta sama co wejściowa:

```
void toupper(string& s) // ujednolica wielkość liter
{
    transform(s.begin(),s.end(),s.begin(),toupper);
}
```

Ta funkcja przekształca wejściową sekwencję s.

32.5.1. copy()

Algorytmy z rodziny `copy()` kopiują elementy z jednej sekwencji do innej. W dalszych podrozdziałach przedstawione są wersje `copy()` połączone z innymi algorytmami, np. `replace_copy()` (32.5.3).

Rodzina <code>copy()</code> (iso.25.3.1)	
<code>p=copy(b,e,out)</code>	Kopiuje wszystkie elementy z $b < e$ do $<out,p)$; $p=out+(e-b)$
<code>p=copy_if(b,e,out,f)</code>	Kopiuje elementy x z $b < e$, dla których $f(x)$, do $<out,p)$
<code>p=copy_n(b,n,out)</code>	Kopiuje n pierwszych elementów z $b < b+n$ do $<out,p)$; $p=out+n$
<code>p=copy_backward(b,e,out)</code>	Kopiuje wszystkie elementy z $b < e$ do $<out,p)$, zaczyna od ostatniego elementu; $p=out+(e-b)$
<code>p=move(b,e,out)</code>	Przenosi wszystkie elementy z $b < e$ do $<out,p)$; $p=out+(e-b)$
<code>p=move_backward(b,e,out)</code>	Przenosi wszystkie elementy z $b < e$ do $<out,p)$, zaczynając od ostatniego elementu; $p=out+(e-b)$

Punktem docelowym algorytmu kopiowania nie musi być kontener. Może być nim wszystko, co można opisać za pomocą iteradora wyjściowego (38.5). Na przykład:

```
void f(list<Club>& lc, ostream& os)
{
    copy(lc.begin(),lc.end(),ostream_iterator<Club>(os));
}
```

Do odczytania sekwencji potrzebne są dwa iteratory określające jej początek i koniec. Aby coś zapisać, wystarczy tylko jeden iterotor wskazujący miejsce zapisu. Trzeba tylko uważać, żeby nie wyjść poza przeznaczoną do zapisu przestrzeń. Jednym ze sposobów na dopilnowanie tego jest użycie wstawiacza (33.2.2) zwiększającego dostępny obszar w razie potrzeby. Na przykład:

```
void f(const vector<char>& vs, vector<char>& v)
{
    copy(vs.begin(),vs.end(),v.begin());           // może nadpisać koniec v
    copy(vs.begin(),vs.end(),back_inserter(v)); // dodaje elementy z vs na końcu v
}
```

Sekwencje wejściowa i wyjściowa mogą się pokrywać. Algorytmu `copy()` używamy, gdy sekwencje się nie pokrywają lub jeśli koniec sekwencji wyjściowej znajduje się w sekwencji wejściowej.

Algorytm `copy_if()` służy do kopирования tylko tych elementów, które spełniają określony warunek. Na przykład:

```
void f(list<int>& ld, int n, ostream& os)
{
    copy_if(ld.begin(),ld.end(),
            ostream_iterator<int>(os),
            [](int x) { return x>n; });
}
```

Zobacz również algorytm `remove_copy_if()`.

32.5.2. unique()

Algorytm `unique()` usuwa przystające duplikaty elementów z sekwencji:

Rodzina algorytmów <code>unique</code> (iso.25.3.9)	
<code>p=unique(b,e)</code>	Przenosi elementy sekwencji $\langle b, e \rangle$, tak że $\langle b, p \rangle$ nie zawiera żadnych przystających duplikatów
<code>p=unique(b,e,f)</code>	Przenosi elementy sekwencji $\langle b, e \rangle$, tak że $\langle b, p \rangle$ nie zawiera żadnych przystających duplikatów; definicję duplikatu określa $f(*p, *(p+1))$
<code>p=unique_copy(b,e,out)</code>	Kopiuje $\langle b, e \rangle$ do $\langle out, p \rangle$; nie kopiuje przystających duplikatów
<code>p=unique_copy(b,e,out,f)</code>	Kopiuje $\langle b, e \rangle$ do $\langle out, p \rangle$; nie kopiuje przystających duplikatów; definicję duplikatu określa $f(*p, *(p+1))$

Algorytmy `unique()` i `unique_copy()` eliminują przystające powielone wartości. Na przykład:

```
void f(list<string>& ls, vector<string>& vs)
{
    ls.sort(); // sortowanie listy (31.4.2)
    unique_copy(ls.begin(), ls.end(), back_inserter(vs));
}
```

Kod ten kopiuje `ls` do `vs`, eliminując po drodze duplikaty wartości, które znalazły się obok siebie dzięki uprzedniemu użyciu funkcji `sort()`.

Podobnie jak inne standardowe algorytmy `unique()` działa na iteratorach. Nie wie, jaki kontener te iteratory wskazują, a więc nie może go zmodyfikować. Może jedynie modyfikować wartości znajdujących się w nim elementów. To oznacza, że algorytm `unique()` nie eliminuje duplikatów w taki sposób, jaki można by było naiwnie zakładać. Zatem poniższy kod nie usunie duplikatów z wektora:

```
void bad(vector<string>& vs)      // ostrzeżenie: nie robi tego, na co wygląda!
{
    sort(vs.begin(), vs.end());    // sortuje wektor
    unique(vs.begin(), vs.end()); // eliminuje duplikaty (wcześnie nie!)
}
```

Algorytm `unique()` przenosi niepowtarzające się elementy na początek sekwencji i zwraca iterator do końca części zawierającej niepowtarzające się elementy. Na przykład:

```
int main()
{
    string s ="abbcccde";
    auto p = unique(s.begin(), s.end());
    cout << s << ' ' << p-s.begin() << '\n';
}
```

Wynik działania tego kodu będzie następujący:

```
abcdecde 5
```

To znaczy, że `p` wskazuje drugie `c` (pierwszy element z duplikatów).

Algorytmy, które teoretycznie mogłyby usuwać elementy (ale praktycznie nie mogą) występują w dwóch postaciach: „zwykłej”, która zmienia kolejność elementów w taki sposób jak `unique()`, oraz wersji `_copy`, która tworzy nową sekwencję w sposób podobny jak `unique_copy()`.

Aby pozbyć się duplikatów z kontenera, należy go zmniejszyć:

```
template<typename C>
void eliminate_duplicates(C& c)
{
    sort(c.begin(),c.end());           // sortuje
    auto p = unique(c.begin(),c.end()); // kompaktuje
    c.erase(p,c.end());               // zmniejsza
}
```

Moglibyśmy też powyższy kod zapisać tak: `c.erase(unique(c.begin(),c.end()),c.end())`, ale to raczej nie poprawiłoby czytelności ani nie ułatwiłoby obsługi programu.

32.5.3. remove() i replace()

Algorytm `remove()` „usuwa” elementy na koniec sekwencji:

remove (iso.25.3.8)	
<code>p=remove(b,e,v)</code>	Usuwa elementy o wartości <code>v</code> z <code><b,e></code> , tak że <code><b,p></code> obejmuje elementy, dla których <code>(*q==v)</code>
<code>p=remove_if(b,e,f)</code>	Usuwa elementy <code>*q</code> z <code><b,e></code> , tak że <code><b,p></code> obejmuje elementy, dla których <code>(!(*q))</code>
<code>p=remove_copy(b,e,out,v)</code>	Kopiuje elementy z <code><b,e></code> , dla których <code>(*q==v)</code> , do <code><out,p></code>
<code>p=remove_copy_if(b,e,out,f)</code>	Kopiuje elementy z <code><b,e></code> , dla których <code>f(*q)</code> , do <code><out,p></code>
<code>reverse(b,e)</code>	Odwija kolejność elementów w <code><b,e></code>
<code>p=reverse_copy(b,e,out)</code>	Kopiuje <code><b,e></code> do <code><out,p></code> w odwrotnej kolejności

Algorytm `replace()` przypisuje nowe wartości do wybranych elementów:

replace (iso.25.3.5)	
<code>replace(b,e,v,v2)</code>	Zamienia elementy <code>*p</code> w <code><b,e></code> , dla których <code>*p==v</code> , na <code>v2</code>
<code>replace_if(b,e,f,v2)</code>	Zamienia elementy <code>*p</code> w <code><b,e></code> , dla których <code>f(*p)</code> , na <code>v2</code>
<code>p=replace_copy(b,e,out,v,v2)</code>	Kopiuje <code><b,e></code> do <code><out,p></code> i zamienia elementy, dla których <code>*p==v</code> , na <code>v2</code>
<code>p=replace_copy_if(b,e,out,f,v2)</code>	Kopiuje <code><b,e></code> do <code><out,p></code> i zamienia elementy, dla których <code>f(*p)</code> , na <code>v2</code>

Te algorytmy nie mogą zmieniać rozmiaru sekwencji wejściowej, a więc nawet `remove()` pozostawia go bez zmian. Podobnie jak `unique()` „usuwa” on elementy poprzez ich przesunięcie. Na przykład:

```
string s {"*CamelCase*IsUgly*"};
cout << s << '\n';                                // *CamelCase*IsUgly*
auto p = remove(s.begin(),s.end(),'*');
copy(s.begin(),p,ostream_iterator<char>(cout)); // CamelCaseIsUgly
cout << s << '\n';                                // CamelCaseIsUgly*
```

32.5.4. `rotate()`, `random_shuffle()` oraz `partition()`

Algorytmy `rotate()`, `random_shuffle()` oraz `partition()` umożliwiają przenoszenie elementów w sekwencjach:

<code>rotate()</code> (iso.25.3.11)	
<code>p=rotate(b,m,e)</code>	Dokonuje rotacji elementów w lewo: traktuje $\langle b, e \rangle$ jak koło, w którym pierwszy element znajduje się za ostatnim; przesuwa $\ast(b+i)$ do $\ast(b+(i+(e-m))\%(e-b))$;
<code>p=rotate_copy(b,m,e,out)</code>	Kopiuje $\langle b, e \rangle$ do obróconej sekwencji $\langle out, p \rangle$

Algorytm `rotate()` (podobnie jak algorytmy przestawiania i dzielenia) do przesuwania elementów używa funkcji `swap()`.

<code>random_shuffle()</code> (iso.25.3.12)	
<code>random_shuffle(b,e)</code>	Przestawia elementy $\langle b, e \rangle$ przy użyciu domyślnego generatora liczb losowych
<code>random_shuffle(b,e,f)</code>	Przestawia elementy $\langle b, e \rangle$ przy użyciu generatora liczb losowych f
<code>shuffle(b,e,f)</code>	Przestawia elementy $\langle b, e \rangle$ przy użyciu generatora liczb losowych o rozkładzie równomiernym f

Algorytm `shuffle()` przestawia elementy w sekwencji w podobny sposób, jak tasuje się talię kart, tzn. po zakończeniu operacji kolejność elementów jest losowa, przy czym „losowość” definiuje rozkład określony przez generator liczb losowych.

Domyślnie algorytm `random_shuffle()` przestawia elementy sekwencji przy użyciu generatora liczb losowych o rozkładzie równomiernym, tzn. tak wybiera permutację elementów sekwencji, że każda permutacja ma takie same szanse na to, by zostać wybraną. Jeśli potrzebny jest inny rozkład albo lepszy generator liczb losowych, można dostarczyć własne rozwiązanie. Dla wywołania `random_shuffle(b,e,r)` generator jest wywoływany z liczbą elementów sekwencji (lub podsekwencji) podaną jako argument. Na przykład dla wywołania `r(e-b)` generator musi zwrócić wartość mieszącą się w przedziale $\langle 0, e-b \rangle$. Jeżeli `My_rand` jest takim generatorem, talię kart można potasować następująco:

```
void f(deque<Card>& dc, My_rand& r)
{
    random_shuffle(dc.begin(),dc.end(),r);
    //...
}
```

Algorytm `partition()` dzieli sekwencję na dwie części wg podanego kryterium:

<code>partition()</code> (iso.25.3.13)	
<code>p=partition(b,e,f)</code>	Elementy, dla których $f(\ast p_1)$ wstawia do $\langle b, p \rangle$, a pozostałe do $\langle p, e \rangle$
<code>p=stable_partition(b,e,f)</code>	Elementy, dla których $f(\ast p_1)$ wstawia do $\langle b, p \rangle$, a pozostałe do $\langle p, e \rangle$; zachowuje względną kolejność
<code>pair(p1,p2)=partition copy(b,e,out1,out2,f)</code>	Kopiuje elementy z $\langle b, e \rangle$, dla których $f(\ast p)$, do $\langle out_1, p_1 \rangle$, a elementy $\langle b, e \rangle$, dla których $\neg f(\ast p)$, do $\langle out_2, p_2 \rangle$
<code>p=partition_point(b,e,f)</code>	Dla $\langle b, e \rangle$ p jest punktem takim, że <code>all_of(b,p,f)</code> i <code>none_of(p,e,f)</code>
<code>is_partitioned(b,e,f)</code>	Czy każdy element $\langle b, e \rangle$, dla którego $f(\ast p)$, poprzedza każdy element, dla którego $\neg f(\ast p)$

32.5.5. Permutacje

Algorytmy permutacyjne umożliwiają wygenerowanie wszystkich permutacji sekwencji.

Permutacje (iso.25.4.9, iso.25.2.12)	
x ma wartość true, jeśli operacja next_* zakończy się powodzeniem, w przeciwnym razie ma wartość false	
x=next_permutation(b,e)	Sekwencja <b,e> jest następną permutacją w porządku leksykograficznym
x=next_permutation(b,e,f)	Sekwencja <b,e> jest następną permutacją, przy użyciu f do porównywania
x=prev_permutation(b,e)	Sekwencja <b,e> jest poprzednią permutacją w porządku leksykograficznym
x=prev_permutation(b,e,f)	Sekwencja <b,e> jest poprzednią permutacją, przy użyciu f do porównywania
is_permutation(b,e,b2)	Czy istnieje permutacja sekwencji <b2,b2+(e-b)>, która jest równa <b,e>?
is_permutation(b,e,b2,f)	Czy istnieje permutacja sekwencji <b2,b2+(e-b)>, która jest równa <b,e>, przy zastosowaniu f(*p,*q) do porównywania elementów?

Permutacji używa się do generowania kombinacji elementów sekwencji. Na przykład permutacjami sekwencji abc są acb, bac, bca, cab i cba.

Algorytm `next_permutation()` pobiera sekwencję <b,e> i przekształca ją w następną permutację. Następna permutacja jest znajdowana poprzez przyjęcie założenia, że zbiór wszystkich permutacji jest uporządkowany leksykograficznie. Jeśli kolejna permutacja istnieje, funkcja `next_permutation()` zwraca wartość `true`. W przeciwnym razie przekształca sekwencję w najmniejszą permutację, tzn. uporządkowaną rosnąco (tu: abc) i zwraca `false`. Permutacje sekwencji abc można wygenerować w następujący sposób:

```
vector<char> v {'a','b','c'};
while(next_permutation(v.begin(),v.end()))
    cout << v[0] << v[1] << v[2] << ' ';
```

Podobnie wartością zwrotną funkcji `prev_permutation()` jest `false`, jeśli <b,e> już zawiera pierwszą permutację (tu: abc). W takim przypadku zwracana jest ostatnia permutacja (tu: cba).

32.5.6. fill()

Algorytmy z rodziny `fill()` umożliwiają przypisywanie do elementów i inicjowanie elementów sekwencji:

Rodzina algorytmów fill (iso.25.3.6, iso.25.3.7, iso.20.6.12)	
<code>fill(b,e,v)</code>	Przypisuje v do każdego elementu sekwencji <b,e>
<code>p=fill_n(b,n,v)</code>	Przypisuje v do każdego elementu sekwencji <b,b+n>; p=b+n
<code>generate(b,e,f)</code>	Przypisuje f() do każdego elementu sekwencji <b,e>
<code>p=generate_n(b,n,f)</code>	Przypisuje f() do każdego elementu sekwencji <b,b+n>; p=b+n
<code>uninitialized_fill(b,e,v)</code>	Inicjuje każdy element sekwencji <b,e> wartością v
<code>p=uninitialized_fill_n(b,n,v)</code>	Inicjuje każdy element sekwencji <b,e> wartością v; p=b+n
<code>p=uninitialized_copy(b,e,out)</code>	Inicjuje każdy element sekwencji <out,out+(e-b)> odpowiednim elementem z <b,e>; p=out+n
<code>p=uninitialized_copy_n(b,n,out)</code>	Inicjuje każdy element sekwencji <out,out+n> odpowiednim elementem z <b,b+n>; p=out+n

Algorytm `fill()` wielokrotnie przypisuje podaną wartość, podczas gdy `generate()` przypisuje wartości uzyskane poprzez wielokrotne wywołanie funkcji podanej jako argument. Można na przykład użyć generatorów liczb losowych `Randint` i `Urand` z podrozdziału 40.7:

```

int v1[900];
array<int,900> v2;
vector v3;

void f()
{
    fill(begin(v1),end(v1),99);           // ustawia wszystkie elementy v1 na 99
    generate(begin(v2),end(v2),Randint{}); // ustawia na losowe wartości (40.7)

    // zwraca 200 losowych liczb całkowitych z przedziału <0,100>
    generate_n(ostream_iterator<int>(cout),200,Urand{100}); // zobacz 40.7

    fill_n(back_inserter{v3},20,99);        // dodaje do v3 20 elementów o wartości 99
}

```

Funkcje `generate()` i `fill()` przypisują, nie inicjują. Aby pracować na surowej pamięci, np. w celu zamiany obszaru pamięci w obiekty o określonym typie i stanie, należy użyć jednej z wersji z przedrostkiem `uninitialized_` (dostępnych w nagłówku `<memory>`).

Niezainicjowane sekwencje powinny występować na najniższym poziomie programowania, najlepiej wewnątrz implementacji kontenerów. Elementy będące celem funkcji `uninitialized_fill()` i `uninitialized_copy()` muszą być typu wbudowanego lub niezainicjowane. Na przykład:

```

vector<string> vs {"Breugel","El Greco","Delacroix","Constable"};
vector<string> vs2 {"Hals","Goya","Renoir","Turner"};
copy(vs.begin(),vs.end(),vs2.begin());           // OK
uninitialized_copy(vs.begin(),vs.end(),vs2.begin()); // wyciek!

```

W podrozdziale 34.6 znajduje się opis jeszcze kilku innych narzędzi do pracy z niezainicjowaną pamięcią.

32.5.7. swap()

Algorytm `swap()` zamienia wartości dwóch obiektów:

Rodzina swap (iso.25.3.3)	
<code>swap(x,y)</code>	Zamienia wartościami x i y
<code>p=swap_ranges(b,e,b2)</code>	Zamienia odpowiadające sobie elementy z $b, e)$ i $b2, b2 + (e - b))$
<code>iter_swap(p,q)</code>	<code>swap(*p,*q)</code>

Na przykład:

```

void use(vector<int>& v, int*p)
{
    swap_ranges(v.begin(),v.end(),p); // zamienia wartości
}

```

Wskaźnik `p` powinien wskazywać tablicę zawierającą przynajmniej `v.size()` elementów.

Algorytm `swap()` jest jednym z najprostszych i chyba najważniejszych algorytmów w całej bibliotece standardowej. Używa się go do implementacji wielu powszechnie używanych innych algorytmów. Przykład jego implementacji przedstawiłem w podrozdziale 7.7.2, a w podrozdziale 35.5.2 przedstawiona jest jego biblioteczna wersja.

32.6. Sortowanie i wyszukiwanie

Operacje sortowania i wyszukiwania w posortowanych sekwencjach są bardzo ważne, a potrzeby programistów w tym zakresie są bardzo zróżnicowane. Standardowo do porównywania wykorzystuje się operator `<`, a równoważność wartości `a` i `b` sprawdza się za pomocą operacji `!(a<b)&&!(b<a)`, nie zaś `==`.

Rodzina <code>sort</code> (iso.25.4.1)	
<code>sort(b,e)</code>	Sortuje <code><b,e)</code>
<code>sort(b,e,f)</code>	Sortuje <code><b,e)</code> przy użyciu <code>f(*p,*q)</code> jako kryterium sortowania

Oprócz „zwykłej” funkcji sortowania istnieją też różne jej odmiany:

Rodzina <code>sort</code> (iso.25.4.1)	
<code>stable_sort(b,e)</code>	Sortuje <code><b,e)</code> , zachowując kolejność równoważnych elementów
<code>stable_sort(b,e,f)</code>	Sortuje <code><b,e)</code> przy użyciu <code>f(*p,*q)</code> jako kryterium sortowania, zachowując kolejność równoważnych elementów
<code>partial_sort(b,m,e)</code>	Sortuje taką część <code><b,e)</code> , aby uporządkować <code><b,m)</code> ; przedział <code><m,e)</code> nie musi być posortowany
<code>partial_sort(b,m,e,f)</code>	Sortuje taką część <code><b,e)</code> , aby uporządkować <code><b,m)</code> , przy użyciu <code>f(*p,*q)</code> jako kryterium sortowania; przedział <code><m,e)</code> nie musi być posortowany
<code>p=partial_sort_copy ~(b,e,b2,e2)</code>	Sortuje taką część <code><b,e)</code> , aby skopiować <code>e2–b2</code> pierwszych elementów do <code><b2,e2)</code> ; <code>p</code> jest mniejszą wartością z <code>e2</code> i <code>b2+(e-b)</code>
<code>p=partial_sort_copy ~(b,e,b2,e2,f)</code>	Sortuje taką część <code><b,e)</code> , aby skopiować <code>e2–b2</code> pierwszych elementów do <code><b2,e2)</code> , przy użyciu <code>f</code> do porównywania; <code>p</code> jest mniejszą wartością z <code>e2</code> i <code>b2+(e-b)</code>
<code>is_sorted(b,e)</code>	Czy przedział <code><b,e)</code> jest posortowany?
<code>is_sorted(b,e,f)</code>	Czy przedział <code><b,e)</code> jest posortowany? Używa <code>f</code> do porównywania
<code>p=is_sorted_until(b,e)</code>	<code>p</code> wskazuje pierwszy element <code><b,e)</code> , który nie jest uporządkowany
<code>p=is_sorted_until ~(b,e,f)</code>	<code>p</code> wskazuje pierwszy element <code><b,e)</code> , który nie jest uporządkowany, używa <code>f</code> do porównywania
<code>nth_element(b,n,e)</code>	<code>*n</code> znajduje się na tej pozycji, na której znajdowałby się, gdyby przedział <code><b,e)</code> był posortowany; elementy w <code><b,n)</code> są $\leq *n$ i $*n \leq$ elementom w <code><n,e)</code>
<code>nth_element(b,n,e,f)</code>	<code>*n</code> znajduje się na tej pozycji, na której znajdowałby się, gdyby przedział <code><b,e)</code> był posortowany; elementy w <code><b,n)</code> są $\leq *n$ i $*n \leq$ elementom w <code><n,e)</code> , używa <code>f</code> do porównywania

Algorytmy `sort()` wymagają do działania iteratorów o dostępie swobodnym (33.1.2).

Algorytm `is_sorted_until()` mimo swojej nazwy zwraca iterator, nie wartość logiczną.

Standardowy kontener `list` (31.3) nie udostępnia iteratorów o dostępie swobodnym i dlatego do sortowania list należy używać specjalnych operacji listowych (31.4.2) albo sortować je poprzez skopiowanie elementów do wektora, posortowanie ich w wektorze i następnie skopiowanie z powrotem do listy:

```
template<typename List>
void sort_list(List& lst)
{
    vector v {lst.begin(), lst.end()}; // inicjacja z lst
    sort(v);                         // sortowania kontenera (32.2)
    copy(v, lst);
}
```

Podstawowa wersja algorytmu `sort()` jest wydajna (średnia złożoność obliczeniowa wynosi $N \log(N)$). Jeśli potrzebne jest sortowanie stabilne, należy używać algorytmu `stable_sort()` o złożoności $N \log(N) \log(N)$, która poprawia się w kierunku $N \log(N)$, jeśli w systemie jest wystarczająco dużo dodatkowej pamięci. Do uzyskania takiej dodatkowej pamięci można wykorzystać funkcję `get_temporary_buffer()` (34.6). Względna kolejność równoważnych elementów jest zachowywana tylko przez algorytm `stable_sort()`.

Czasami potrzebne są tylko początkowe elementy posortowanej sekwencji. W takim przypadku lepiej jest posortować ją tylko do wybranego miejsca, czyli wykonać sortowanie częściowe. Zwykłe algorytmy `partial_sort(b, m, e)` ustawiają w porządku elementy z przedziału $[b, m]$. Algorytmy `partial_sort_copy()` tworzą N elementów, przy czym N jest mniejszą z liczbą elementów w sekwencji wyjściowej i liczbą elementów w sekwencji wejściowej. Należy podać zarówno początek, jak i koniec sekwencji wynikowej, bo w ten sposób określa się liczbę elementów do posortowania. Na przykład:

```
void f(const vector<Book>& sales) // znajduje dziesięć najpopularniejszych książek
{
    vector<Book> bestsellers(10);
    partial_sort_copy(sales.begin(), sales.end(),
                      bestsellers.begin(), bestsellers.end(),
                      [] (const Book& b1, const Book& b2) { return b1.copies_sold() > b2.copies_sold(); });
    copy(bestsellers.begin(), bestsellers.end(), ostream_iterator<Book>(cout, "\n"));
}
```

Nie można wysłać wyniku sortowania bezpośrednio do `cout`, bo celem algorytmu `partial_sort_copy()` musi być iterator o dostępie swobodnym.

Jeśli liczba elementów do posortowania przez algorytm `partial_sort()` jest tylko niewielką częścią liczby wszystkich elementów, to zysk wydajności w porównaniu z pełnym algorytmem `sort()` może być duży. Złożoność obliczeniowa w takim przypadku zbliża się do $O(N)$, podczas gdy złożoność `sort()` wynosi $O(N \log(N))$.

Algorytm `nth_element()` sortuje tylko do momentu przemieszczenia N -tego elementu na właściwe miejsce i dopóki żaden element mniejszy od N nie znajduje się za nim w sekwencji. Na przykład:

```
vector<int> v;
for (int i=0; i<1000; ++i)
    v.push_back(randint(1000)); // 40.7
constexpr int n = 30;
nth_element(v.begin(), v.begin() + n, v.end());
cout << "n-ty: " << v[n] << '\n';
for (int i=0; i<n; ++i)
    cout << v[i] << ' ';
```

Wynik:

```
n-ty: 24
10 8 15 19 21 15 8 7 6 17 21 2 18 8 1 9 3 21 20 18 10 7 3 3 8 11 11 22 22 23
```

Algorytm `nth_element()` różni się od `partial_sort()` tym, że elementy znajdujące się przed `n` nie muszą zostać posortowane — wystarczy, że żaden z nich nie będzie mniejszy od `n`-tego. Gdyby w powyższym przykładzie zamieniono `nth_element` na `partial_sort` (i użyto tego samego ziarna do uruchomienia generatora liczb losowych, aby otrzymać taką samą sekwencję liczb), to wynik byłby następujący:

```
n-ty: 24
1 2 3 3 3 6 7 7 8 8 8 9 10 10 11 11 15 15 17 18 18 19 20 21 21 21 22 22 23
```

Algorytm `nth_element()` jest przydatny dla wszystkich — np. ekonomistów, socjologów i nauczycieli — którzy wyliczają mediany, percentyle itp.

W celu posortowania łańcucha w stylu C konieczne jest bezpośrednie podanie kryterium sortowania. Jest to spowodowane tym, że łańcuchy w stylu C są wskaźnikami, przez co użycie na nich operatora `<` powoduje porównanie adresów maszynowych, nie sekwencji znaków. Na przykład:

```
vector<string> vs = {"Helsinki", "Copenhagen", "Oslo", "Stockholm"};
vector<char*> vcs = {"Helsinki", "Copenhagen", "Oslo", "Stockholm"};

void use()
{
    sort(vs); // Zdefiniowałem zakresową wersję funkcji sort()
    sort(vcs);

    for (auto& x : vs)
        cout << x << ' '
    cout << '\n';
    for (auto& x : vcs)
        cout << x << ' ';
```

Wynik działania tego kodu:

```
Copenhagen Helsinki Stockholm Oslo
Helsinki Copenhagen Oslo Stockholm
```

Ktoś naiwny mógłby oczekiwąć, że z obu wektorów otrzyma ten sam wynik. Ale żeby posortować łańcuchy w stylu C wg wartości, nie zaś adresów, potrzebny jest predykat sortowania. Na przykład:

```
sort(vcs, [](const char*p, const char*q){ return strcmp(p,q)<0; });
```

Opis standardowej funkcji `strcmp()` znajduje się w podrozdziale 43.4.

Zwróć uwagę, że w celu posortowania łańcuchów w stylu C nie musiałem dostarczać operatora `==`. Aby uprościć interfejs użytkownika, w bibliotece standardowej do porównywania używa się operacji `!(x<y>|y<x)` zamiast `==` (31.2.2.2).

32.6.1. Wyszukiwanie binarne

Algorytmy z rodziny `binary_search()` służą do binarnego przeszukiwania uporządkowanych (posortowanych) sekwencji:

Wyszukiwanie binarne (iso.25.4.3)	
p=lower_bound(b,e,v)	p wskazuje pierwsze wystąpienie v w <b,e>
p=lower_bound(b,e,v,f)	p wskazuje pierwsze wystąpienie v w <b,e>, używa f do porównywania
p=upper_bound(b,e,v)	p wskazuje pierwszą wartość większą od v w <b,e>
p=upper_bound(b,e,v,f)	p wskazuje pierwszą wartość większą od v w <b,e>, używa f do porównywania
binary_search(b,e,v)	Czy v znajduje się w posortowanej sekwencji <b,e>?
binary_search(b,e,v,f)	Czy v znajduje się w posortowanej sekwencji <b,e>? Używa f do porównywania
pair(p1,p2)=equal_range(b,e,v)	<p1,p2> jest podsekwencją <b,e> zawierającą wartość v; zasadniczo jest to binarne wyszukiwanie v
pair(p1,p2)=equal_range(b,e,v,f)	<p1,p2> jest podsekwencją <b,e> zawierającą wartość v; używa f do porównywania; zasadniczo jest to binarne wyszukiwanie v

Sekwencyjne przeszukiwanie, np. za pomocą funkcji `find()` (32.4), długich sekwencji jest strasznie nieefektywne, ale niewiele da się z tym zrobić bez sortowania lub mieszania (31.4.3.2). Ale jeśli sekwencja jest posortowana, to do znajdowania w niej wartości można używać wyszukiwania binarnego. Na przykład:

```
void f(vector<int>& c)
{
    if (binary_search(c.begin(),c.end(),7)) { // czy w c znajduje się 7?
        ...
    }
    ...
}
```

Funkcja `binary_search()` zwraca wartość logiczną oznaczającą, czy szukana wartość jest obecna. Podobnie jak w przypadku używania funkcji `find()` często chcemy też wiedzieć, w którym miejscu dana wartość się znajduje. Ale elementów o określonej wartości w sekwencji może być wiele i najczęściej chodzi o znalezienie pierwszego z nich lub wszystkich. Dlatego właśnie utworzono również algorytmy do znajdowania zakresów takich samych elementów (`equal_range()`) oraz do znajdowania dolnej i górnej granicy tych zakresów — odpowiednio `lower_bound()` i `upper_bound()`. Algorytmy te odpowiadają operacjom na wielosłownikach (31.4.3). Funkcję `lower_bound()` można traktować jak szybką wersję `find()` i `find_if()` dla posortowanych sekwencji. Na przykład:

```
void g(vector<int>& c)
{
    auto p = find(c.begin(),c.end(),7);           // zapewne wolne: O(N); c nie musi być posortowany
    auto q = lower_bound(c.begin(),c.end(),7); // zapewne szybkie: O(log(N)); c musi być posortowany
    ...
}
```

Jeśli operacja `lower_bound(first,last,k)` nie znajdzie k, zwraca iterator do pierwszego elementu o kluczu większym od k lub ostatni, jeśli taki element nie istnieje. Ten sposób informowania o niepowodzeniu jest również stosowany przez funkcje `upper_bound()` i `equal_range()`. To oznacza, że przy użyciu tych algorytmów można znajdować miejsca do wstawiania nowych elementów do posortowanych sekwencji, tak żeby sekwencje te pozostały posortowane: wystarczy wstawić element przed drugim elementem zwróconej pary.

Co ciekawe, algorytmy wyszukiwania binarnego nie wymagają iteratorów o dostępie swobodnym: wystarczy im iterator jednokierunkowy.

32.6.2. merge()

Algorytmy `merge` służą do łączenia dwóch uporządkowanych (posortowanych) sekwencji w jedną:

Rodzina <code>merge</code> (iso.25.4.4)	
<code>p=merge(b,e,b2,e2,out)</code>	Łączy dwie posortowane sekwencje $\langle b2, e2 \rangle$ i $\langle b, e \rangle$ w $\langle out, p \rangle$
<code>p=merge(b,e,b2,e2,out,f)</code>	Łączy dwie posortowane sekwencje $\langle b2, e2 \rangle$ i $\langle b, e \rangle$ w $\langle out, out+p \rangle$, używa <code>f</code> do porównywania
<code>inplace_merge(b,m,e)</code>	Łączy dwie posortowane podsekwencje $\langle b, m \rangle$ i $\langle m, e \rangle$ w posortowaną sekwencję $\langle b, e \rangle$
<code>inplace_merge(b,m,e,f)</code>	Łączy dwie posortowane podsekwencje $\langle b, m \rangle$ i $\langle m, e \rangle$ w posortowaną sekwencję $\langle b, e \rangle$, używa <code>f</code> do porównywania

Algorytm `merge()` przyjmuje różne rodzaje sekwencji i elementy różnych typów. Na przykład:

```
vector<int> v {3,1,4,2};
list<double> lst {0.5,1.5,2,2.5}; // lista lst jest uporządkowana

sort(v.begin(),v.end());           // porządkuje v

vector<double> v2;
merge(v.begin(),v.end(),lst.begin(),lst.end(),back_inserter(v2)); // łączy v i lst w v2
for (double x : v2)
    cout << x << ", ";
```

Opis wstawiaczy znajduje się w podrozdziale 33.2.2. Wynik działania przedstawionego programu jest następujący:

0.5, 1, 1.5, 2, 2, 2.5, 3, 4,

32.6.3. Algorytmy działające na zbiorach

Algorytmy z tej kategorii traktują sekwencje jako zbiory elementów i wykonują podstawowe operacje na zbiorach. Sekwencje wejściowe powinny być posortowane. Zwracane sekwencje również są posortowane.

Algorytmy działające na zbiorach (iso.25.4.5)	
<code>includes(b,e,b2,e2)</code>	Czy wszystkie elementy $\langle b, e \rangle$ znajdują się także w $\langle b2, e2 \rangle$?
<code>includes(b,e,b2,e2,f)</code>	Czy wszystkie elementy $\langle b, e \rangle$ znajdują się także w $\langle b2, e2 \rangle$? Używa <code>f</code> do porównywania
<code>p=set_union(b,e,b2,e2,out)</code>	Tworzy posortowaną sekwencję $\langle out, p \rangle$ z elementów, które znajdują się w $\langle b, e \rangle$ lub $\langle b2, e2 \rangle$
<code>p=set_union(b,e,b2,e2,out,f)</code>	Tworzy posortowaną sekwencję $\langle out, p \rangle$ z elementów, które znajdują się w $\langle b, e \rangle$ lub $\langle b2, e2 \rangle$, używa <code>f</code> do porównywania

Algorytmy działające na zbiorach (iso.25.4.5) — ciąg dalszy	
p=set_intersection(b,e,b2,e2,out)	Tworzy posortowaną sekwencję <out,p> z elementów, które znajdują się zarówno w <b,e>, jak i <b2,e2>
p=set_intersection(b,e,b2,e2,out,f)	Tworzy posortowaną sekwencję <out,p> z elementów, które znajdują się zarówno w <b,e>, jak i <b2,e2>, używa f do porównywania
p=set_difference(b,e,b2,e2,out)	Tworzy posortowaną sekwencję <out,p> z elementów, które znajdują się w <b,e>, ale nie w <b2,e2>
p=set_difference(b,e,b2,e2,out,f)	Tworzy posortowaną sekwencję <out,p> z elementów, które znajdują się w <b,e>, ale nie w <b2,e2>, używa f do porównywania
p=set_symmetric_difference(b,e,b2,e2,out)	Tworzy posortowaną sekwencję <out,p> z elementów, które znajdują się w <b,e> lub <b2,e2>, ale nie w obu naraz
p=set_symmetric_difference(b,e,b2,e2,out,f)	Tworzy posortowaną sekwencję <out,p> z elementów, które znajdują się w <b,e> lub <b2,e2>, ale nie w obu naraz, używa f do porównywania

```

string s1 = "qwertyasdfgzxcvb";
string s2 = "poiuyasdfg./,mnb";
sort(s1.begin(),s1.end());           // algorytmy działające na zbiorach wymagają sekwencji posortowanych
sort(s2.begin(),s2.end());

string s3(s1.size()+s2.size(),'*'); // odłożenie pamięci na największy możliwy wynik
cout << s3 << '\n';
auto up = set_union(s1.begin(),s1.end(),s2.begin(),s2.end(),s3.begin());
cout << s3 << '\n';
for (auto p = s3.begin(); p!=up; ++p) cout << *p;
cout << '\n';

s3.assign(s1.size()+s2.size(),'+');
up = set_difference(s1.begin(),s1.end(),s2.begin(),s2.end(),s3.begin());
cout << s3 << '\n';
for (auto p = s3.begin(); p!=up; ++p) cout << *p;
cout << '\n';

```

Wynik:

```

*****
.,/abcdefghijklmnopqrstuvwxyz*****
.,/abcdefghijklmnopqrstuvwxyz
ceqr twxz+++++
ceqr twxz

```

32.6.4. Sterty

Sterta to kompaktowa struktura danych, w której największy element jest umieszczany na początku. Można ją traktować jak reprezentację drzewa binarnego. Algorytmy stertowe umożliwiają programiście traktowanie sekwencji o dostępie swobodnym jak sterły:

Operacje stertowe (iso.25.4.6)	
<code>make_heap(b,e)</code>	Przygotowuje $\langle b,e \rangle$ do użycia jako sterty
<code>make_heap(b,e,f)</code>	Przygotowuje $\langle b,e \rangle$ do użycia jako sterty, używa f do porównywania
<code>push_heap(b,e)</code>	Dodaje $\ast(e-1)$ do sterty $\langle b,e-1 \rangle$; po tym $\langle b,e \rangle$ jest stertą
<code>push_heap(b,e,f)</code>	Dodaje element do sterty $\langle b,e-1 \rangle$, używa f do porównywania
<code>pop_heap(b,e)</code>	Usuwa $\ast(e-1)$ ze sterty $\langle b,e \rangle$, po tym $\langle b,e-1 \rangle$ jest stertą
<code>pop_heap(b,e,f)</code>	Usuwa element ze sterty $\langle b,e \rangle$, używa f do porównywania
<code>sort_heap(b,e)</code>	Sortuje stertę $\langle b,e \rangle$
<code>sort_heap(b,e,f)</code>	Sortuje stertę $\langle b,e \rangle$, używa f do porównywania
<code>is_heap(b,e)</code>	Czy $\langle b,e \rangle$ jest stertą?
<code>is_heap(b,e,f)</code>	Czy $\langle b,e \rangle$ jest stertą? Używa f do porównywania
<code>p=is_heap_until(b,e)</code>	p jest największym p , takim że $\langle b,p \rangle$ jest stertą
<code>p=is_heap_until(b,e,f)</code>	p jest największym p , takim że $\langle b,p \rangle$ jest stertą, używa f do porównywania

Koniec sterty $\langle b,e \rangle$, e , należy traktować jak wskaźnik, którego wartość zmniejsza się za pomocą funkcji `pop_heap()`, a zwiększa przy użyciu `push_heap()`. Największy element pobiera się poprzez b (np. $x=\ast b$) i wykonanie `pop_heap()`. Nowy element wstawia się poprzez e (np. $*e=x$) i wykonanie `push_heap()`. Na przykład:

```
string s = "herewego";           // herewego
make_heap(s.begin(),s.end());    // worheege
pop_heap(s.begin(),s.end());     // roghheew
pop_heap(s.begin(),s.end()-1);   // ohgeeerw
pop_heap(s.begin(),s.end()-2);   // hegeeorw

*(s.end()-3)='f';
push_heap(s.begin(),s.end()-2); // hegeefrw
*(s.end()-2)='x';
push_heap(s.begin(),s.end()-1); // xeheefgw
*(s.end()-1)='y';
push_heap(s.begin(),s.end());   // yxheefge
sort_heap(s.begin(),s.end());   // eeefghxy
reverse(s.begin(),s.end());    // yxhgfeee
```

Aby zrozumieć zmiany w s , należy sobie uświadomić, że użytkownik odczytuje tylko $s[0]$ i zapisuje tylko $s[x]$, gdzie x jest indeksem aktualnego końca sterty. Sterta usuwa element (zawsze $s[0]$) poprzez zamianę go z $s[x]$.

Sterty używa się, gdy potrzebna jest możliwość szybkiego dodawania elementów i znajdowania elementu o najwyższej wartości. Najczęściej sterty używa się do implementacji kolejek priorytetowych.

32.6.5. `lexicographical_compare()`

Porównywanie leksykograficzne (ang. *lexicographical compare*) to zbiór zasad, wg których sortuje się słowa w słowniku.

Porównywanie leksykograficzne (iso.25.4.8)	
<code>lexicographical_compare(b,e,b2,e2)</code>	Czy $\langle b,e \rangle < \langle b2,e2 \rangle$?
<code>lexicographical_compare(b,e,b2,e2,f)</code>	Czy $\langle b,e \rangle < \langle b2,e2 \rangle$? Używa f do porównywania elementów

Implementacja funkcji `lexicographical_compare()` może wyglądać następująco:

```
template<typename In, typename In2>
bool lexicographical_compare(In first, In last, In2 first2, In2 last2)
{
    for (; first!=last && first2!=last2; ++first,++first2) {
        if (*first<*first2)
            return true; // <first,last) < <first2,last2)
        if (*first2<*first)
            return false; // <first2,last2) < <first,last)
    }
    return first==last && first2==last2; // <first,last) < <first2,last2), jeśli <first,last) jest krótszy
}
```

Łańcuch jest traktowany jak sekwencja znaków. Na przykład:

```
string n1 {"10000"};
string n2 {"999"};

bool b1 = lexicographical_compare(n1.begin(),n1.end(),n2.begin(),n2.end()); // b1 == true
n1 = "Zebra";
n2 = "Aardvark";
bool b2 = lexicographical_compare(n1.begin(),n1.end(),n2.begin(),n2.end()); // b2 == false
```

32.7. Element minimalny i maksymalny

Operacje porównywania wartości są przydatne w wielu miejscach:

Rodzina <code>min</code> i <code>max</code> (iso.25.4.7)	
<code>x=min(a,b)</code>	<code>x</code> jest mniejszą z wartości <code>a</code> i <code>b</code>
<code>x=min(a,b,f)</code>	<code>x</code> jest mniejszą z wartości <code>a</code> i <code>b</code> , używa <code>f</code> do porównywania
<code>x=min({elem})</code>	<code>x</code> jest najmniejszym elementem w <code>{elem}</code>
<code>x=min({elem},f)</code>	<code>x</code> jest najmniejszym elementem w <code>{elem}</code> , używa <code>f</code> do porównywania
<code>x=max(a,b)</code>	<code>x</code> jest większą z wartości <code>a</code> i <code>b</code>
<code>x=max(a,b,f)</code>	<code>x</code> jest większą z wartości <code>a</code> i <code>b</code> , używa <code>f</code> do porównywania
<code>x=max({elem})</code>	<code>x</code> jest największym elementem w <code>{elem}</code>
<code>x=max({elem},f)</code>	<code>x</code> jest największym elementem w <code>{elem}</code> , używa <code>f</code> do porównywania
<code>pair(x,y)=minmax(a,b)</code>	<code>x</code> jest <code>min(a,b)</code> , <code>y</code> jest <code>max(a,b)</code>
<code>pair(x,y)=minmax(a,b,f)</code>	<code>x</code> jest <code>min(a,b,f)</code> , <code>y</code> jest <code>max(a,b,f)</code>
<code>pair(x,y)=minmax({elem})</code>	<code>x</code> jest <code>min({elem})</code> , <code>y</code> jest <code>max({elem})</code>
<code>pair(x,y)=minmax({elem},f)</code>	<code>x</code> jest <code>min({elem},f)</code> , <code>y</code> jest <code>max({elem},f)</code>
<code>p=min_element(b,e)</code>	<code>p</code> wskazuje najmniejszy element w <code><b,e)</code> lub <code>e</code>
<code>p=min_element(b,e,f)</code>	<code>p</code> wskazuje najmniejszy element w <code><b,e)</code> lub <code>e</code> , używa <code>f</code> do porównywania
<code>p=max_element(b,e)</code>	<code>p</code> wskazuje największy element w <code><b,e)</code> lub <code>e</code>
<code>p=max_element(b,e,f)</code>	<code>p</code> wskazuje największy element w <code><b,e)</code> lub <code>e</code> , używa <code>f</code> do porównywania
<code>pair(x,y)=minmax_element(b,e)</code>	<code>x</code> jest <code>min_element(b,e)</code> , a <code>y</code> jest <code>max_element(b,e)</code>
<code>pair(x,y)=minmax_element(b,e,f)</code>	<code>x</code> jest <code>min_element(b,e,f)</code> , a <code>y</code> jest <code>max_element(b,e,f)</code>

Jeśli porównywane są dwie wartości lewostronne, wynikiem jest referencja do wyniku. W pozostałych przypadkach zwracana jest wartość prawostronna. Niestety wersje pobierające wartości lewostronne pobierają tylko wartości `const`, przez co nie można modyfikować wyników ich działania. Na przykład:

```
int x = 7;
int y = 9;
++min(x,y); // błąd: wynikiem min(x,y) jest const int&
++min({x,y}); // błąd: wynikiem min({x,y}) jest wartość prawostronna (lista initializer_list jest niezmienna)
```

Funkcje z przyrostkiem `_element` zwracają iteratory, a funkcja `minmax` zwraca pary:

```
string s = "Large_Hadron_Collider";
auto p = minmax_element(s.begin(),s.end(),
    [](char c1,char c2) { return toupper(c1)<toupper(c2); });
cout << "min==" <<*(p.first) << ' ' << "max==" <<*(p.second) << '\n';
```

Przy użyciu zestawu znaków ASCII na moim komputerze wynik tego testu był następujący:

```
min==a max==_
```

32.8. Rady

1. Algorytmy STL działają na jednej lub większej liczbie sekwencji — 32.2.
2. Sekwencja wejściowa jest przedziałem jednostronnie otwartym, a definiuje ją para iteratorów — 32.2.
3. Na znak nieznalezienia szukanego elementu algorytmy wyszukiwania najczęściej zwracają koniec sekwencji wejściowej — 32.2.
4. Lepiej jest używać dopracowanych algorytmów niż „chaotycznego kodu” — 32.2.
5. Pisząc pętlę, zastanów się, czy dałoby się jej zdefiniować jako ogólny algorytm — 32.2.
6. Upewnij się, czy para argumentów iteratorowych rzeczywiście określa sekwencję — 32.2.
7. Jeśli posługiwanie się iteratorami jest żmudne, użyj zakresowego algorytmu kontenerowego — 32.2.
8. Używaj predykatów i innych obiektów funkcyjnych, aby nadać standardowym algorytmom szerszy zakres znaczeń — 32.3.
9. Predykat nie może modyfikować swojego argumentu — 32.3.
10. Domyślne wersje operatorów `==` i `<` są rzadko przydatne do użytku na wskaźnikach w standardowych algorytmach — 32.3.
11. Sprawdzaj, jaką złożoność obliczeniową mają używane przez Ciebie algorytmy, ale pamiętaj, że jest to tylko przybliżona informacja dotycząca wydajności — 32.3.1.
12. Funkcji `for_each()` i `transform()` używaj tylko wtedy, gdy nie możesz użyć żadnego bardziej specjalistycznego algorytmu — 32.4.1.
13. Algorytmy bezpośrednio nie dodają ani nie usuwają elementów sekwencji — 32.5.2, 32.5.3.
14. Do pracy z niezainicjowanymi obiektami możesz używać algorytmów `unitIALIZED_*` — 32.5.6.
15. Algorytm z biblioteki STL używa testu równości wygenerowanego z jego operacji porządkowania, nie operatora `==` — 32.6.
16. W celu sortowania i przeszukiwania łańcuchów w stylu C użytkownik musi dostarczyć operację porównywania łańcuchów — 32.6.

Iteratory STL

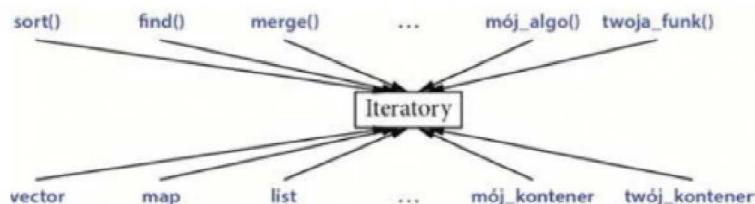
Powodem, dla którego kontenery i algorytmy STL tak dobrze ze sobą współpracują, jest to, że jedne nic nie wiedzą o drugich
 — Alex Stepanov

- Wprowadzenie
- Model iteratorów
 - Kategorie iteratorów; Cechy iteratorów; Operacje iteratorów
- Adaptacje iteratorów
 - Iteratory odwrotne; Iteratory wstawiające; Iteratory przenoszące
- Zakresowe funkcje dostępowe
- Obiekty funkcyjne
- Adaptacje funkcji
 - bind(); mem_fn(); function
- Rady

33.1. Wprowadzenie

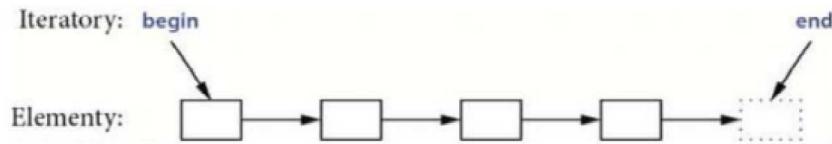
W tym rozdziale znajduje się opis iteratorów i dodatkowych narzędzi (obiektów funkcyjnych) dostępnych w bibliotece STL. Biblioteka STL obejmuje iteratory, kontenery, algorytmy i obiekty funkcyjne biblioteki standardowej. Pozostałe części biblioteki STL są opisane w rozdziałach 31. i 32.

Iteratory są spojwem łączącym algorytm biblioteki standardowej z danymi, na których one operują. Można też powiedzieć, że iteratory to mechanizm umożliwiający zminimalizowanie zależności algorytmów od struktur danych, na których te algorytmy działają:



33.1.1. Model iteratorów

Iterator jest podobny do wskaźnika, bo podobnie jak wskaźnik udostępnia operacje do pośredniego dostępu (np. `*` do dereferencji) i można go przesuwać, aby wskazywał nowe elementy (np. za pomocą operatora `++` iterator można przesunąć na następny element). **Sekwencję** określa para iteratorów definiujących jednostronnie otwarty przedział `<begin, end>`:



Iterator begin wskazuje pierwszy element sekwencji, a end — miejsce za ostatnim elementem sekwencji. Nigdy nie należy odczytywać ani zapisywać *end. Dla pustej sekwencji spełniony jest warunek `begin==end`, tzn. `[p,p)` jest pustą sekwencją dla każdego iteratatora p.

Aby odczytać sekwencję, algorytm najczęściej pobiera parę iteratorów `(b,e)` reprezentującą przedział jednostronnie otwarty `<b,e)` i iteruje za pomocą operatora `++` aż do napotkania końca sekwencji:

```
while (b!=e) { // użycie !=, nie zas <
    // jakieś działania
    ++b; // przejście do następnego elementu
}
```

Test `!=` został użyty do sprawdzenia, czy osiągnięto koniec sekwencji, z dwóch powodów. Po pierwsze: lepiej wyraża zamiar programisty niż `<`, a po drugie: operatory `<` obsługują tylko iteratory o dostępie swobodnym.

Algorytmy wyszukujące elementów w sekwencjach fakt nieznalezienia szukanego elementu najczęściej zgłaszają poprzez zwrócenie końca sekwencji, np.:

```
auto p = find(v.begin(),v.end(),x); // szuka x w v

if (p!=v.end()) {
    // x znaleziono na pozycji p
}
else {
    // nie znaleziono x w <v.begin(),v.end())
}
```

Algorytmem zapisującym coś w sekwencjach zazwyczaj przekazuje się tylko iterator do pierwszego elementu. Wówczas programista sam musi zadbać, by nie wyjść poza sekwencję. Na przykład:

```
template<typename Iter>
void forward(Iter p, int n)
{
    while (n>0)
        *p++ = --n;
}

void user()
{
    vector<int> v(10);
    forward(v.begin(),v.size()); // OK
    forward(v.begin(),1000);    // poważne problemy
}
```

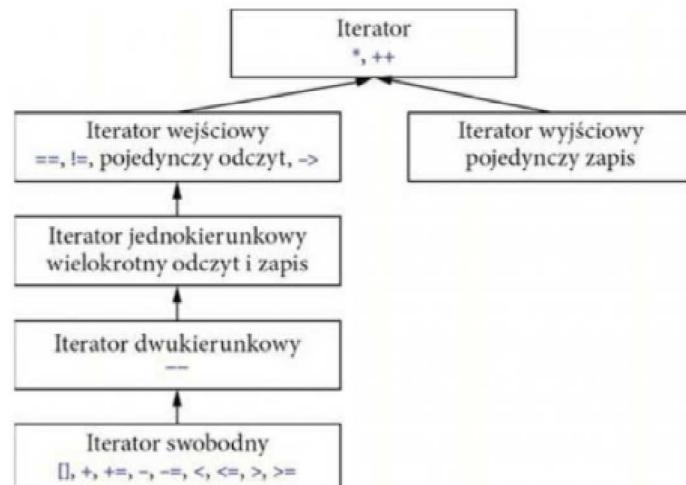
Niektóre implementacje biblioteki standardowej wykonują sprawdzanie zakresu, tzn. zgłosząby wyjątek dla ostatniego z wywołań funkcji `forward()` w powyższym kodzie. Ale nie można na tym polegać, jeśli kod ma być przenośny, ponieważ wiele implementacji tego nie robi. Jeśli potrzebujesz prostego i bezpiecznego alternatywnego rozwiązania, użyj iteratora wstawiającego (33.2.2).

33.1.2. Kategorie iteratorów

W bibliotece standardowej znajduje się **pięć rodzajów (kategorii) iteratorów**:

- **Iterator wejściowy** — umożliwia przeglądanie sekwencji do przodu przy użyciu operatora `++` oraz odczytywanie elementów za pomocą operatora `*`. Iteratory wejściowe można porównywać za pomocą operatorów `==` i `!=`. Tego rodzaju iterator jest dostępny w strumieniu `istream` (38.5).
- **Iterator wyjściowy** — umożliwia przeglądanie sekwencji do przodu przy użyciu operatora `++` oraz odczytywanie elementów za pomocą operatora `*`. Tego rodzaju iterator jest dostępny w strumieniu `ostream` (38.5).
- **Iterator jednokierunkowy** — umożliwia wielokrotne przeglądanie sekwencji do przodu przy użyciu operatora `++` oraz wielokrotne odczytywanie i zapisywanie elementów (pod warunkiem że nie są `const`) za pomocą operatora `*`. Jeśli iterator jednokierunkowy wskazuje obiekt klasy, można odwoływać się do jego składowych przy użyciu operatora `->`. Iteratory jednokierunkowe można porównywać za pomocą operatorów `==` i `!=`. Tego rodzaju iterator udostępnia struktura `forward_list` (31.4).
- **Iterator dwukierunkowy** — umożliwia przeglądanie sekwencji do przodu (przy użyciu operatora `++`) i wstecz (przy użyciu operatora `--`) oraz wielokrotne odczytywanie i zapisywanie elementów (pod warunkiem że nie są `const`) za pomocą operatora `*`. Jeśli iterator dwukierunkowy wskazuje obiekt klasy, można odwoływać się do jego składowych przy użyciu operatora `->`. Iteratory dwukierunkowe można porównywać za pomocą operatorów `==` i `!=`. Tego rodzaju iterator udostępniają struktury `list`, `map` i `set` (31.4).
- **Iterator o dostępie swobodnym** — umożliwia przeglądanie sekwencji do przodu (przy użyciu operatora `++` lub `+=`) i wstecz (przy użyciu operatora `--` lub `-=`) oraz wielokrotne odczytywanie i zapisywanie elementów (pod warunkiem że nie są `const`) za pomocą operatora `*` lub `[]`. Jeśli iterator swobodny wskazuje obiekt klasy, można odwoływać się do jego składowych przy użyciu operatora `->`. Iterator swobodny można indeksować przy użyciu operatora `[]`, zwiększać o liczbę całkowitą za pomocą operatora `+` oraz zmniejszać o liczbę całkowitą za pomocą operatora `-`. Różnicę między dwoma iteratorami swobodnymi wskazującymi miejsca w tej samej sekwencji można obliczyć poprzez odjęcie jednego od drugiego. Do porównywania iteratorów swobodnych można używać operatorów `==`, `!=`, `<`, `<=` oraz `>`. Tego rodzaju iterator udostępnia strukturę `vector` (31.4).

Rodzaje iteratorów tworzą hierarchię (iso.24.2):



Kategorie iteratorów są koncepcjami (24.3), nie klasami, a więc powyższy schemat nie przedstawia hierarchii klas zaimplementowanej przy użyciu derywacji. Do wykonywania bardziej zaawansowanych czynności z kategoriami iteratorów należy używać `iterator_traits` (pośrednio lub bezpośrednio).

33.1.3. Cechy iteratorów

W nagłówku `<iterator>` biblioteki standardowej znajduje się zbiór funkcji typowych umożliwiających pisanie kodu wyspecjalizowanego pod kątem wybranych właściwości iteratora:

Cechy iteratorów (iso.24.4.1)	
<code>iterator_traits<Iter></code>	Typ cechujący dla Iter nie będącego wskaźnikiem
<code>iterator_traits<T*></code>	Typ cechujący dla wskaźnika T*
<code>iterator<Cat,T,Dist,Ptr,Re></code>	Prosta klasa definiująca podstawowe typy składowe iteratorów
<code>input_iterator_tag</code>	Kategoria iteratorów wejściowych
<code>output_iterator_tag</code>	Kategoria iteratorów wyjściowych
<code>forward_iterator_tag</code>	Kategoria iteratorów jednokierunkowych; pochodna <code>input_iterator_tag</code> ; dostarczona dla <code>forward_list</code> , <code>unordered_set</code> , <code>unordered_multiset</code> , <code>unordered_map</code> oraz <code>unordered_multimap</code>
<code>bidirectional_iterator_tag</code>	Kategoria iteratorów dwukierunkowych; pochodna <code>forward_iterator_tag</code> ; dostarczona dla <code>list</code> , <code>set</code> , <code>multiset</code> , <code>map</code> oraz <code>multimap</code>
<code>random_access_iterator_tag</code>	Kategoria iteratorów swobodnych; pochodna <code>bidirectional_iterator_tag</code> ; dostarczona dla <code>vector</code> , <code>deque</code> , <code>array</code> , tablic wbudowanych oraz <code>string</code>

Znaczniki iteratorów to typy służące do wybierania algorytmów na podstawie typu iteratora. Na przykład iterator swobodny może bezpośrednio odwołać się do elementu:

```
template<typename Iter>
void advance_helper(Iter p, int n, random_access_iterator_tag)
{
    p+=n;
}
```

Z drugiej strony, iterator jednokierunkowy, aby dojść do wybranego elementu, musi przejść pojedynczo przez wszystkie wcześniejsze elementy (np. podążając po łączach w liście):

```
template<typename Iter>
void advance_helper(Iter p, int n, bidirectional_iterator_tag)
{
    if (0<n)
        while (n--) ++p;
    else if (n<0)
        while (n++) --p;
}
```

Dzięki tym pomocom funkcja `advance()` może zawsze używać optymalnego algorytmu:

```
template<typename Iter>
void advance(Iter p, int n) // używa optymalnego algorytmu
{
    advance_helper(p,n,typename iterator_traits<Iter>::iterator_category{});
```

Funkcje `advance()` i `advance_helper()` w większości przypadków zostaną poddane rozwijaniu w linii, aby uniknąć narzutu związanego ze stosowaniem tej **techniki wykorzystania znaczników** (ang. *tag dispatch*). Różne odmiany tej techniki są powszechnie stosowane w bibliotece STL.

Kluczowe właściwości iteratora są opisane przez aliasy znajdujące się w klasie `iterator_traits`:

```
template<typename Iter>
struct iterator_traits {
    using value_type = typename Iter::value_type;
    using difference_type = typename Iter::difference_type;
    using pointer = typename Iter::pointer; // typ wskaźnikowy
    using reference = typename Iter::reference; // typ referencyjny
    using iterator_category = typename Iter::iterator_category; // (tag)
};
```

Jeśli iterator nie ma tych typów składowych (np. `int*`), można zdefiniować specjalizację szablonu `iterator_traits`:

```
template<typename T>
struct iterator_traits<T*> { // specjalizacja dla wskaźników
    using difference_type = ptrdiff_t;
    using value_type = T;
    using pointer = T*;
    using reference = T&;
    using iterator_category = random_access_iterator_tag;
};
```

Ogólnie rzecz biorąc, nie można napisać:

```
template<typename Iter>
typename Iter::value_type read(Iter p, int n) // nieogólne
{
    //... sprawdzanie...
    return p[n];
}
```

Ten kod zawiera błąd, który wcześniej czy później się ujawni. Wywołanie tej funkcji `read()` ze wskaźnikiem jako argumentem byłoby błędem. Kompilator go znalazł, ale zwrócone informacje o błędzie mogłyby być bardzo obszerne i niejasne. Zamiast tego można napisać:

```
template<typename Iter>
typename iterator_traits<Iter>::value_type read(Iter p, int n) // bardziej ogólne
{
    //... sprawdzanie...
    return p[n];
}
```

Chodzi o to, że aby znaleźć właściwość iteratora, należy zatrzymać się do jego `iterator_traits` (28.2.4), a nie analizować sam iterator. Aby uniknąć bezpośredniego sięgania do `iterator_traits`, co w istocie jest tylko szczegółem implementacyjnym, można zdefiniować alias. Na przykład:

```
template<typename Iter>
using Category<Iter> = typename std::iterator_traits<Iter>::iterator_category;

template<typename Iter>
using Difference_type<Iter> = typename std::iterator_traits<Iter>::difference_type;
```

Gdy więc chcemy poznać typ różnicy między dwoma iteratorami (wskazującymi tę samą sekwencję), mamy kilka możliwości do wyboru:

```
template<typename Iter>
void f(Iter p, Iter q)
{
    Iter::difference_type d1 = distance(p,q);           // błęd skladni: brak typename

    typename Iter::difference_type d2 = distance(p,q); // nie działa dla wskaźników itd.
    typename iterator_traits<Iter>::difference_type d3 = distance(p,q); // OK, ale brzydkie
    Difference_type<Iter> d4 = distance(p,q);          // OK, znacznie lepsze

    auto d5 = distance(p,q); // OK, jeśli nie trzeba podawać typu bezpośrednio
    //...
}
```

Zalecam dwa ostatnie rozwiązania.

Szablon `iterator` jest po prostu zbiorem najważniejszych właściwości iteratorów w postaci struktury. Ułatwia pracę implementatorom iteratorów oraz zawiera kilka ustawień domyślnych:

```
template<typename Cat, typename T, typename Dist = ptrdiff_t, typename Ptr = T*,
typename Ref = T&>
struct iterator {
    using value_type = T;
    using difference_type = Dist; // typ używany przez distance()
    using pointer = Ptr;         // typ wskaźnikowy
    using reference = Ref;       // typ referencyjny
    using iterator_category = Cat; // kategoria (tag)
};
```

33.1.4. Operacje iteratorów

W zależności od kategorii, do której przynależy (33.1.2), iterator może udostępniać niektóre lub wszystkie z poniższych operacji:

Operacje iteratorów (iso.24.2.2)	
<code>++p</code>	Preinkrementacja (przesunięcie o jeden element do przodu): przestawia <code>p</code> na następny element lub na miejsce za ostatnim elementem; wynikiem jest zwiększoną wartość
<code>p++</code>	Postinkrementacja (przesunięcie o jeden element do przodu): przestawia <code>p</code> na następny element lub na miejsce za ostatnim elementem; wynikiem jest wartość <code>p</code> przed zwiększeniem
<code>*p</code>	Dostęp (dereferencja): <code>*p</code> odnosi się do elementu wskazywanego przez <code>p</code>
<code>--p</code>	Predekrementacja (przesunięcie o jeden element wstecz): przestawia <code>p</code> na poprzedni element; wynikiem jest zmniejszona wartość
<code>p--</code>	Postdekrementacja (przesunięcie o jeden element wstecz): przestawia <code>p</code> na poprzedni element; wynikiem jest wartość <code>p</code> przed zmniejszenia
<code>p[n]</code>	Dostęp (indeksowanie): <code>p[n]</code> odnosi się do elementu wskazywanego przez <code>p+n</code> ; równoważne z <code>(*p).n</code>
<code>p->m</code>	Dostęp (do składowej): równoważne z <code>(*p).m</code>
<code>p==q</code>	Równość: czy <code>p</code> i <code>q</code> wskazują ten sam element albo czy wskazują miejsce za ostatnim elementem?

Operacje iteratorów (iso.24.2.2) — ciąg dalszy	
p!=q	Nierówność: !(p==q)
p<q	Czy p wskazuje element znajdujący się przed elementem wskazywanym przez q?
p<=q	p<q p==q
p>q	Czy p wskazuje element znajdujący się za elementem wskazywanym przez q?
p>=q	p>q p==q
p+=n	Przesuwa n do przodu: ustawia p na n-ty element za aktualnie wskazywanym elementem
p-=n	Przesuwa n wstecz: ustawia p na n-ty element przed aktualnie wskazywanym elementem
q=p+n	q wskazuje n-ty element za elementem wskazywanym przez p
q=p-n	q wskazuje n-ty element przed elementem wskazywanym przez p

Operacja $++p$ zwraca referencję do p, podczas gdy $p++$ musi zwracać kopię p zawierającą starą wartość. Dlatego w skomplikowanych iteratorach operacja $++p$ jest zazwyczaj bardziej wydajna niż $p++$.

Poniższe operacje działają dla wszystkich iteratorów, dla których można je zaimplementować, ale największą wydajność można uzyskać dla iteratorów swobodnych (zobacz 33.1.2):

Operacje iteratorów (iso.24.4.4)	
advance(p,n)	Jak $p+=n$; p musi być przynajmniej iteratorem wejściowym
x=distance(p,q)	Jak $x=q-p$; p musi być przynajmniej iteratorem wejściowym
q=next(p,n)	Jak $q=p+1$; p musi być przynajmniej iteratorem jednokierunkowym
q=next(p)	$q=next(p,1)$
q=prev(p,n)	Jak $q=p-n$; p musi być przynajmniej iteratorem dwukierunkowym
q=prev(p)	$q=prev(p,1)$

W każdym przypadku, jeśli p nie jest iteratorem swobodnym, algorytm wykona n kroków.

33.2. Adaptacje iteratorów

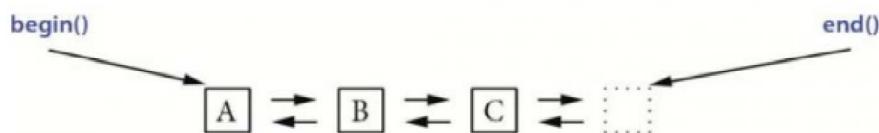
W nagłówku <iterator> znajdują się adaptacje służące do generowania przydatnych typów iteratorów z danego typu iteratora:

Adaptacje iteratorów		
reverse_iterator	Iteracja wstecz	33.2.1
back_insert_iterator	Wstawianie na końcu	33.2.2
front_insert_iterator	Wstawianie na początku	33.2.2
insert_iterator	Wstawianie w dowolnym miejscu	33.2.2
move_iterator	Przenoszenie zamiast kopowania	33.2.3
raw_storage_iterator	Zapis w niezainicjowanej pamięci	34.6.2.

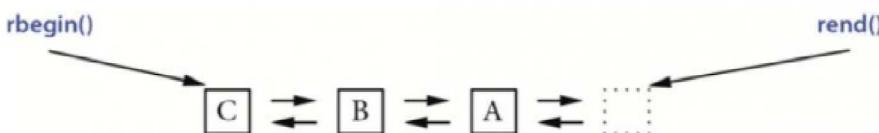
Iteratory strumieni wejścia i wyjścia są opisane w podrozdziale 38.5.

33.2.1. Iterator odwrotny

Przy użyciu iteratorka można przejrzeć sekwencję $\langle b, e \rangle$ od b do e . Jeśli sekwencja umożliwia dostęp dwukierunkowy, można ją przeglądać również w odwrotnym porządku, tzn. od e do b . Iteratorka służący do takiego przeglądania nazywa się iteratorem odwrotnym (`reverse_iterator`). Iteratorka odwrotna przegląda sekwencję od końca zdefiniowanego przez jego iteratorka podstawowy do początku. Aby otrzymać przedział jednostronnie otwarty, $b-1$ należy traktować jako miejsce za końcem, a $e-1$ jako początek sekwencji: $\langle e-1, b-1 \rangle$. Zatem podstawową relację między iteratorem odwrotnym i jego odpowiednikiem wyraża $\&*(\text{reverse_iterator}(p)) == \&*(p-1)$. Na przykład jeśli v jest wektorem, $v.rbegin()$ wskazuje jego ostatni element $v[v.size()-1]$. Spójrz na poniższy schemat:



W przypadku użycia iteratorka `reverse_iterator` sekwencję tę można postrzegać tak:



Definicja `reverse_iterator` wygląda mniej więcej tak:

```
template<typename Iter>
class reverse_iterator
    : public iterator<Iterator_category<Iter>,
        Value_type<Iter>,
        Difference_type<Iter>,
        Pointer<Iter>,
        Reference<Iter>> {
public:
    using iterator_type = Iter;

    reverse_iterator(): current{} { }
    explicit reverse_iterator(Iter p): current{p} { }
    template<typename Iter2>
    reverse_iterator(const reverse_iterator<Iter2>& p) :current(p.base()) { }

    Iter base() const { return current; } // bieżąca wartość iteratorka

    reference operator*() const { tmp = current; return *--tmp; }
    pointer operator->() const;
    reference operator[](difference_type n) const;

    reverse_iterator& operator++() { --current; return *this; } // uwaga: nie ++
    reverse_iterator operator++(int) { reverse_iterator t = current; --current; return t; }
    reverse_iterator& operator--() { ++current; return *this; } // uwaga: nie --
    reverse_iterator operator--(int) { reverse_iterator t = current; ++current; return t; }

    reverse_iterator operator+(difference_type n) const;
    reverse_iterator& operator+=(difference_type n);
```

```

reverse_iterator operator-(difference_type n) const;
reverse_iterator& operator-=(difference_type n);
//...
protected:
    Iterator current; //current wskazuje element za wskazywanym przez *this
private:
    //...
    iterator tmp;      // dla obiektów tymczasowych, które muszą istnieć poza zakresem funkcji
};

```

Iterator `reverse_iterator<Iter>` ma takie same typy składowe i operacje jak `Iter`. W szczególności jeśli `Iter` jest iteratorem o dostępie swobodnym, jego `reverse_iterator<Iter>` ma operatory `[]`, `+` oraz `<`. Na przykład:

```

void f(vector<int>& v, list<char>& lst)
{
    v.rbegin()[3] = 7;                      // OK: iterator swobodny
    lst.rbegin()[3] = '4';                   // błąd: operator dwukierunkowy nie obsługuje []
    *(next(lst.rbegin(),3)) = '4';          // OK!
}

```

Użyłem funkcji `next()` do przesunięcia iteratora, ponieważ (podobnie jak `[]`) + nie działa dla iteratorów dwukierunkowych, takich jak `list<char>::iterator`.

Iteratory odwrotne umożliwiają przeglądanie przez algorytmy sekwencji od końca. Aby na przykład znaleźć w sekwencji ostatnie wystąpienie elementu, można zastosować funkcję `find()` do jej odwrotności:

```
auto ri = find(v.rbegin(),v.rend(),val); // ostatnie wystąpienie
```

Zwróć uwagę, że `C::reverse_iterator` to nie ten sam typ co `C::iterator`. Gdybym więc chciał napisać algorytm `find_last()` przy użyciu odwrotnej sekwencji, musiałbym zdecydować, który typ iteratora zwracać:

```

template<typename C, typename Val>
auto find_last(C& c, Val v) -> decltype(c.begin()) // użycie iteratora C w interfejsie
{
    auto ri = find(c.rbegin(),c.rend(),v);
    if (ri == c.rend()) return c.end();                // użycie c.end() jako znaku „nie znaleziono”
    return prev(ri.base());
}

```

Dla `reverse_iterator` funkcja `ri.base()` zwraca iterator wskazujący miejsce za pozycją wskazywaną przez `ri`. Aby więc otrzymać iterator wskazujący ten sam element, który wskazuje iterator odwrotny `ri`, należy zwrócić `ri.base()-1`. Ale ponieważ kontenerem może być lista, której operatory nie obsługują operatora `-`, użyłem funkcji `prev()`.

Iterator odwrotny to całkiem zwyczajny iterator, a więc mogę napisać pętlę jawnie:

```

template<typename C, Val v>
auto find_last(C& c, Val v) -> decltype(c.begin())
{
    for (auto p = c.rbegin(); p!=c.rend(); ++p) // widzi sekwencję w odwrotnej kolejności
        if (*p==v) return --p.base();           // użycie c.end() jako znaku „nie znaleziono”
    return c.end();
}

```

Równoważny kod przeszukujący wstecz przy użyciu iteratatora (dwukierunkowego) wygląda tak:

```
template<typename C>
auto find_last(C& c, Val v) -> decltype(c.begin())
{
    for (auto p = c.end(); p!=c.begin(); ) // przeszukuje wstecz, od end
        if (*--p==v) return p;
    return c.end();                      // użycie c.end() jako znaku „nie znaleziono”
}
```

Podobnie jak poprzednie definicje `find_last()` ta również wymaga przynajmniej iteratora dwukierunkowego.

33.2.2. Iteratory wstawiające

Konsekwencją wstawienia danych przez iterator do kontenera może być nadpisanie elementów znajdujących się za miejscem wstawienia. To z kolei może pociągać za sobą przepełnienie i uszkodzenie pamięci. Na przykład:

```
void f(vector<int>& vi)
{
    fill_n(vi.begin(),200,7); // przypisuje 7 do vi[0]..[199]
}
```

Jeśli kontener `vi` zawiera mniej niż 200 elementów, to mamy problem.

Rozwiązanie tego problemu znajduje się w nagłówku `<iterator>` pod postacią tzw. **wstawiaczy** (ang. *inserter*). Wstawiacz wstawia nowy element do sekwencji, zamiast nadpisywać istniejący element. Na przykład:

```
void g(vector<int>& vi)
{
    fill_n(back_inserter(vi),200,7); // dodaje 200 siódemek na końcu vi
}
```

Gdy do zapisu elementu używa się iteratatora wstawiającego, iterator ten wstawi swoją wartość do sekwencji, zamiast nadpisać wskazywany przez siebie element. W efekcie kontener zwiększa się o jeden element. Wstawiacze są równie proste i wydajne, jak i przydatne.

Dostępne są trzy iteratory wstawiające:

- **insert_iterator**: wstawia element przed elementem wskazywanym za pomocą funkcji `insert()`;
- **front_insert_iterator**: wstawia element przed pierwszym elementem sekwencji za pomocą funkcji `push_front()`;
- **back_insert_iterator**: wstawia element za ostatnim elementem sekwencji za pomocą funkcji `push_back()`.

Iterator wstawiający zazwyczaj konstruuje się przy użyciu wywołania funkcji pomocniczej:

Funkcje do tworzenia wstawiaczy (iso.24.5.2)	
<code>ii= inserter(c,p)</code>	ii to <code>insert_iterator</code> wskazujący p w kontenerze c
<code>ii= back_inserter(c)</code>	ii to <code>back_insert_iterator</code> wskazujący <code>back()</code> w kontenerze c
<code>ii= front_inserter(c)</code>	ii to <code>front_insert_iterator</code> wskazujący <code>front()</code> w kontenerze c

Iterator przekazany do funkcji `inserter()` musi wskazywać element w kontenerze. W przypadku kontenerów sekwencyjnych oznacza to, że musi to być iterator dwukierunkowy (aby można było wstawiać elementy przed nim). Nie można na przykład użyć funkcji `inserter()` w celu utworzenia iteratora wstawiającego elementy do `forward_list`. W przypadku kontenerów asocjacyjnych, w których iterator służy tylko jako wskazówka do wstawienia elementu, może wystarczyć iterator jednokierunkowy (np. dostarczany przez `unordered_set`).

Wstawiacze są iteratorami wyjściowymi:

Operacje <code>insert_iterator<C></code> (iso.24.5.2)	
<code>insert_iterator p {c,q};</code>	Wstawiacz dla kontenera c wskazujący *q; q musi wskazywać element w c
<code>insert_iterator p {q};</code>	Konstruktor kopiący: p jest kopią q
<code>p=q</code>	Przypisanie kopiące: p jest kopią q
<code>p=move(q)</code>	Przypisanie przenoszące: p wskazuje to, co wskazywał q
<code>++p</code>	Przestawia p na następny element; wartością jest nowa wartość p
<code>p++</code>	Przestawia p na następny element; wartością jest stara wartość p
<code>*p=x</code>	Wstawia x przed p
<code>*p++=x</code>	Wstawia x przed p, a następnie zwiększa p

Iteratory wstawiające `front_insert_iterator` i `back_insert_iterator` różnią się tym, że ich konstruktory nie wymagają iteratora. Na przykład:

```
vector<string> v;
back_insert_iterator<vector<string>> p {v};
```

Przy użyciu wstawiacza nie można odczytywać wartości.

33.2.3. Iteratory przenoszące

Iterator przenoszący to taki, który przy odczycie wskazywanego elementu przenosi go, zamiast kopować. Iterator tego rodzaju najczęściej robi się z innego iteratora przy użyciu funkcji pomocniczej:

Funkcja do tworzenia iteratorów przenoszących	
<code>mp=make_move_iterator(p)</code>	mp jest iteratorem przenoszącym <code>move_iterator</code> wskazującym ten sam element co p; p musi być iteratorem wejściowym

Iterator przenoszący udostępnia ten sam zestaw operacji co iterator, z którego został utworzony. Jeśli na przykład iterator przenoszący zostanie utworzony z iteratora dwukierunkowego, to można na nim wykonywać operacje `--p`. Funkcja `operator*()` iteratora przenoszącego zwraca referencję prawostronną (7.7.2) do wskazywanego elementu: `std::move(q)`. Na przykład:

```
vector<string> read_strings(istream&);
auto vs = read_strings(cin); //pobierałańcuchy

vector<string> vs2;
copy(vs.begin(),vs.end(),back_inserter(vs2)); //kopiuje z vs do vs2

vector<string> vs3;
copy(make_move_iterator(vs2.begin()),make_move_iterator(vs2.end()),back_inserter(vs3)); //przenosi
```

Funkcja `make_move_iterator()` jest tym, co zasadniczo wykorzystuje wewnętrznie algorytm `move()` (32.5.1).

33.3. Zakresowe funkcje dostępowe

W nagłówku `<iterator>` znajdują się nie będące składowymi funkcje `begin()` i `end()` służące do pracy z kontenerami:

Funkcje <code>begin()</code> i <code>end()</code> (iso.24.6.5)	
<code>p=begin(c)</code>	<code>p</code> jest iteratorem do pierwszego elementu <code>c</code> ; <code>c</code> jest tablicą wbudowaną lub ma funkcję <code>c.begin()</code>
<code>p=end(c)</code>	<code>p</code> jest iteratorem do miejsca za ostatnim elementem <code>c</code> ; <code>c</code> jest tablicą wbudowaną lub ma funkcję <code>c.end()</code>

Budowa tych funkcji jest bardzo prosta:

```
template<typename C>
    auto begin(C& c) -> decltype(c.begin());
template<typename C>
    auto begin(const C& c) -> decltype(c.begin());
template<typename C>
    auto end(C& c) -> decltype(c.end());
template<typename C>
    auto end(const C& c) -> decltype(c.end());

template<typename T, size_t N> // dla tablic wbudowanych
    auto begin(T (&array)[N]) -> T*;
template<typename T, size_t N>
    auto end(T (&array)[N]) -> T*;
```

Funkcji tych używa zakresowa instrukcja `for` (9.5.1), ale oczywiście użytkownik może też używać ich bezpośrednio. Na przykład:

```
template<typename Cont>
void print(Cont& c)
{
    for(auto p=begin(c); p!=end(c); ++p)
        cout << *p << '\n';
}

void f()
{
    vector<int> v {1,2,3,4,5};
    print(v);

    int a[] {1,2,3,4,5};
    print(a);
}
```

Gdybym napisał `c.begin()` i `c.end()`, wywołanie `print(a)` zakończyłoby się niepowodzeniem.

Gdy zostanie dołączony nagłówek `<iterator>`, zdefiniowany przez użytkownika kontener zawierający składowe `begin()` i `end()` automatycznie bierze nieskładowe wersje tych funkcji. Aby nieskładowe funkcje `begin()` i `end()` miał niestandardowy kontener `My_container`, należałoby napisać coś takiego:

```

template<typename T>
Iterator<My_container<T>> begin(My_container<T>& c)
{
    return Iterator<My_container<T>>{&c[0]};           // iterator do pierwszego elementu
}

template<typename T>
Iterator<My_container<T>> end(My_container<T>& c)
{
    return Iterator<My_container<T>>{&c[0]+c.size()}; // iterator do ostatniego elementu
}

```

Przyjęłem założenie, że iterator do pierwszego elementu kontenera `My_container` tworzy się poprzez przekazanie adresu tego elementu oraz że `My_container` ma funkcję `size()`.

33.4. Obiekty funkcyjne

Wiele standardowych algorytmów przyjmuje jako argumenty obiekty funkcyjne (lub funkcje), które sterują ich działaniem. Wśród typowych przykładów można wymienić kryteria porównywania, predykaty (funkcje zwracające wartości logiczne) oraz operacje arytmetyczne. W nagłówku `<functional>` znajduje się kilka często używanych obiektów funkcyjnych:

Predykaty (iso.20.8.5, iso.20.8.6, iso.20.8.7)	
p=equal_to<T>(x,y)	p(x,y) oznacza $x==y$, gdy x i y są typu T
p=not_equal_to<T>(x,y)	p(x,y) oznacza $x!=y$, gdy x i y są typu T
p=greater<T>(x,y)	p(x,y) oznacza $x>y$, gdy x i y są typu T
p=less<T>(x,y)	p(x,y) oznacza $x<y$, gdy x i y są typu T
p=greater_equal<T>(x,y)	p(x,y) oznacza $x\geq y$, gdy x i y są typu T
p=less_equal<T>(x,y)	p(x,y) oznacza $x\leq y$, gdy x i y są typu T
p=logical_and<T>(x,y)	p(x,y) oznacza $x\&\&y$, gdy x i y są typu T
p=logical_or<T>(x,y)	p(x,y) oznacza $x\mid\mid y$, gdy x i y są typu T
p=logical_not<T>(x)	p(x) oznacza $\neg x$, gdy x jest typu T
p=bit_and<T>(x,y)	p(x,y) oznacza $x\&y$, gdy x i y są typu T
p=bit_or<T>(x,y)	p(x,y) oznacza $x\mid y$, gdy x i y są typu T
p=bit_xor<T>(x,y)	p(x,y) oznacza $x\wedge y$, gdy x i y są typu T

Na przykład:

```

vector<int> v;
//...
sort(v.begin(),v.end(),greater<int>()); // sortuje v w porządku malejącym

```

Takie predykaty są z grubsza równoważne prostym lambdami. Na przykład:

```

vector<int> v;
//...
sort(v.begin(),v.end(),[](int a, int b) { return a>b; }); // sortuje v w porządku malejącym

```

Pamiętaj, że `logical_and` i `logical_or` zawsze obliczają wartość obu swoich argumentów (`&&` i `||` tego nie robią).

Operacje arytmetyczne (iso.20.8.4)	
<code>f=plus<T>(x,y)</code>	<code>f(x,y)</code> oznacza $x+y$, gdy x i y są typu T
<code>f=minus<T>(x,y)</code>	<code>f(x,y)</code> oznacza $x-y$, gdy x i y są typu T
<code>f=multiplies<T>(x,y)</code>	<code>f(x,y)</code> oznacza $x*y$, gdy x i y są typu T
<code>f=divides<T>(x,y)</code>	<code>f(x,y)</code> oznacza x/y , gdy x i y są typu T
<code>f=modulus<T>(x,y)</code>	<code>f(x,y)</code> oznacza $x \% y$, gdy x i y są typu T
<code>f=negate<T>(x)</code>	<code>f(x)</code> oznacza $-x$, gdy x jest typu T

33.5. Adaptacje funkcji

Adaptacja funkcji pobiera funkcję jako argument i zwraca obiekt funkcyjny, przy użyciu którego można wywołać oryginalną funkcję.

Adaptacje (iso.20.8.9, iso.20.8.10, iso.20.8.8)	
<code>g=bind(f,args)</code>	<code>g(args2)</code> jest równoważne z <code>f(args3)</code> , przy czym <code>args3</code> otrzymuje się poprzez zamianę symboli zastępczych w <code>args</code> na argumenty z <code>args2</code> przeznaczone do użytku przez symbole zastępcze, takie jak <code>_1</code> , <code>_2</code> i <code>_3</code>
<code>g=mem_fn(f)</code>	<code>g(p,args)</code> oznacza <code>p->f(args)</code> , jeżeli <code>p</code> jest wskaźnikiem, a <code>p.f(args)</code> , jeśli <code>p</code> nie jest wskaźnikiem; <code>args</code> jest (potencjalnie pustą) listą argumentów
<code>g=not1(f)</code>	<code>g(x)</code> oznacza <code>!f(x)</code>
<code>g=not2(f)</code>	<code>g(x,y)</code> oznacza <code>!f(x,y)</code>

Adaptacje `bind()` i `mem_fn()` wykonują wiązanie argumentów, czyli czynność nazywaną także **curryingiem** lub **częściową ewaluacją**. Funkcje te, jak również wycofywane z użytku ich poprzedniczki (`bind1st()`, `mem_fun()` oraz `mem_fun_ref()`), kiedyś były powszechnie wykorzystywane, ale w większości przypadków łatwiej jest użyć lambd zamiast nich (11.4).

33.5.1. bind()

Mając daną funkcję i zbiór argumentów, funkcja `bind()` tworzy obiekt funkcyjny, który można wywoływać z „pozostałymi” argumentami, jeśli takie są, funkcji. Na przykład:

```
double cube(double);

auto cube2 = bind(cube,2);
```

Wywołanie `cube2()` wywoła `cube` z argumentem 2, tzn. `cube(2)`. Nie ma obowiązku wiązania wszystkich argumentów funkcji. Na przykład:

```
using namespace placeholders;

void f(int,const string&);
auto g = bind(f,2,_1); // wiąże pierwszy argument funkcji f() z 2
f(2,"witaj");
g("witaj");           // także wywołuje f(2,"hello");
```

Argument `_1` jest symbolem zastępczym informującym funkcję `bind()` o tym, gdzie powinny trafić argumenty w powstałym obiekcie funkcyjnym. W tym przypadku (pierwszy) argument `g()` został użyty jako drugi argument `f()`.

Symbole zastępcze znajdują się w (pod)przestrzeni nazw `std::placeholders` należącej do nagłówka `<functional>`. Mechanizm symboli zastępczych jest bardzo elastyczny. Na przykład:

```
f(2,"witaj");
bind(f)(2,"witaj");           // także wywołuje f(2,"witaj");
bind(f,_1,_2)(2,"witaj");   // także wywołuje f(2,"witaj");
bind(f,_2,_1)("witaj",2);    // odwrotna kolejność argumentów: także wywołuje f(2,"witaj");

auto g = [](const string& s, int i) { f(i,s); } // odwrotna kolejność argumentów
g("witaj",2);                                // także wywołuje f(2,"witaj");
```

Aby związać argumenty przeciążonej funkcji, należy konkretnie określić, która wersja funkcji nas interesuje:

```
int pow(int,int);
double pow(double,double); //funkcja pow() jest przeciążona

auto pow2 = bind(pow,_1,2);           // błąd: która funkcja pow()?;
auto pow2 = bind((double(*)(double,double))pow,_1,2); // OK (ale brzydkie)
```

Zwróć uwagę, że funkcja `bind()` przyjmuje jako argumenty zwykłe wyrażenia. To oznacza, że referencje są poddawane dereferencji, zanim funkcja `bind()` ich użyje. Na przykład:

```
void incr(int& i)
{
    ++i;
}

void user()
{
    int i = 1;
    incr(i); // i ma wartość 2
    auto inc = bind(incr,_1);
    inc(i); // i pozostaje z wartością 2; inc(i) zwiększyło lokalną kopię i
}
```

W celu rozwiązania tego problemu w bibliotece standardowej dostarczono jeszcze jedną parę adaptacji:

reference_wrapper<T> (iso.20.8.3)	
r=ref(t)	r to reference_wrapper dla T& t, noexcept
r=cref(t)	r to reference_wrapper dla const T& t, noexcept

To rozwiązuje „problem z referencjami” w funkcji `bind()`:

```
void user2()
{
    int i = 1;
    incr(i); // i ma wartość 2
    auto inc = bind(incr,_1);
    inc(ref(i)); // i ma wartość 3
}
```

Przedstawiona funkcja `ref()` jest potrzebna do przekazywania referencji jako argumentów do wątków, ponieważ konstruktory wątków są szablonami zmiennymi (42.2.2).

Do tej pory wyniku funkcji `bind()` używałem bezpośrednio albo przypisywałem go do zmiennej `auto`. Dzięki temu nie musiałem określać typu zwrotnego wywołania funkcji `bind()`. Jest to przydatne, ponieważ typ zwrotny `bind()` różni się w zależności od typu funkcji do wywołania i zapisanych wartości argumentów. Zwrócony obiekt funkcyjny jest większy, gdy zawiera wartości wiązanych parametrów. Jednak czasami trzeba konkretnie określić typy wymaganych argumentów i zwracanego wyniku. Jeśli tak, można je określić dla `function` (33.5.3).

33.5.2. `mem_fn()`

Adaptacja funkcji `mem_fn(mf)` tworzy obiekt funkcyjny, który można wywoływać jako funkcję nieskładową. Na przykład:

```
void user(Shape* p)
{
    p->draw();
    auto draw = mem_fn(&Shape::draw);
    draw(p);
}
```

Adaptacji `mem_fn()` najczęściej używa się wtedy, gdy algorytm wymaga, aby operacja była wywoływana jako funkcja nieskładowa. Na przykład:

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(),v.end(),mem_fn(&Shape::draw));
}
```

Adaptację `mem_fn()` można traktować jak przejściówkę z obiektowego stylu wywoływania na funkcyjny.

Często prostym i ogólnym zastępstwem mogą być lambdy. Na przykład:

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(),v.end(),[] (Shape* p) { p->draw(); });
}
```

33.5.3. `function`

Funkcji `bind()` można używać bezpośrednio albo do inicjacji zmiennej `auto`. Pod tym względem funkcja `bind()` jest podobna do lambdy.

Aby przypisać wynik funkcji `bind()` do zmiennej określonego typu, można użyć standardego typu `function`. Typ ten określa się przy użyciu określonego typu zwrotnego i typu argumentu.

<code>function<R(Argtypes...)></code> (iso.20.8.11.2)	
<code>function f {};</code>	<code>f</code> jest pustym obiektem <code>function</code> ; <code>noexcept</code>
<code>function f {nullptr};</code>	<code>f</code> jest pustym obiektem <code>function</code> ; <code>noexcept</code>
<code>function f {g};</code>	<code>f</code> jest funkcją zawierającą <code>g</code> ; <code>g</code> może być czymkolwiek, co można wywołać z typami argumentów <code>f</code>
<code>function f {allocator_arg_t,a};</code>	<code>f</code> jest pustym obiektem <code>function</code> ; używa alokatora <code>a</code> ; <code>noexcept</code>
<code>function f {allocator_arg_t,a,nullptr_t};</code>	<code>f</code> jest pustym obiektem <code>function</code> ; używa alokatora <code>a</code> ; <code>noexcept</code>

function<R(Argtypes...)> (iso.20.8.11.2) — ciąg dalszy	
<code>function f {allocator_arg_t,a,g};</code>	f jest funkcją zawierającą g; używa alokatora a; noexcept
<code>f2=f</code>	f2 jest kopią f
<code>f=nullptr</code>	f staje się pusty
<code>f.swap(f2)</code>	Zamienia zawartość f i f2; f i f2 muszą być tego samego typu function; noexcept
<code>f.assign(f2,a)</code>	f otrzymuje kopię f2 i alokator a
<code>bool b {f};</code>	Konwertuje f na bool; b jest true, gdy f nie jest pusty; noexcept
<code>r=f(args)</code>	Wywołuje zawartą funkcję z args; typy argumentów muszą odpowiadać tym w f
<code>ti=f.target_type()</code>	ti to type_info dla f; jeśli f nie zawiera czegoś, co da się wywołać, to ti==typeid(void); noexcept
<code>p=f.target<F>()</code>	Jeśli f.target_type()==typeid(F), p wskazuje zawarty obiekt, w przeciwnym razie p==nullptr; noexcept
<code>f==nullptr</code>	Czy f jest pusty? noexcept
<code>nullptr==f</code>	f==nullptr
<code>f!=nullptr</code>	!(f==nullptr)
<code>nullptr!=f</code>	!(f==nullptr)
<code>swap(f,f2)</code>	f.swap(f2)

Na przykład:

```
int f1(double);
function<int(double)> fct {f1}; // inicjacja do f1
int f2(int);

void user()
{
    fct = [](double d) { return round(d); }; // przypisuje lambda do fct
    fct = f1;                                // przypisuje funkcję do fct
    fct = f2;                                // błąd: niepoprawny typ argumentu
}
```

Funkcje docelowe są dostarczone dla rzadkich przypadków, gdy ktoś chce zbadać function, nie zaś po to, by umożliwić wywołanie w normalny sposób.

Standardowa konstrukcja function jest typem mogącym zawierać dowolny obiekt, który można wywołać za pomocą operatora wywoływania () (2.2.1, 3.4.3, 11.4, 19.2.2). Innymi słowy, obiekt typu function jest obiektem funkcyjnym. Na przykład:

```
int round(double x) { return static_cast<int>(floor(x+0.5)); } // konwencjonalne zaokrąglanie 4/5
function<int(double)> f; // f może zawierać wszystko, co można wywołać z typem double i zwraca typ int
enum class Round_style { truncate, round };

struct Round { // obiekt funkcyjny przenoszący stan
    Round_style s;
    Round(Round_style ss) :s(ss) { }
    int operator()(double x) const { return static_cast<int>((s==Round_style::round) ?
        (x+0.5) : x); }
};
```

```

void t1()
{
    f = round;
    cout << f(7.6) << '\n';                                // wywołanie przez funkcji round

    f = Round(Round_style::truncate);
    cout << f(7.6) << '\n';                                // wywołanie obiektu funkcyjnego

    Round_style style = Round_style::round;
    f = [style] (double x){ return static_cast<int>((style==Round_style::round) ?
        x+0.5 : x); };

    cout << f(7.6) << '\n';                                // wywołanie lambdy

    vector<double> v {7.6};
    f = Round(Round_style::round);
    std::transform(v.begin(),v.end(),v.begin(),f); // przekazanie do algorytmu

    cout << v[0] << '\n';                                // przekształcone przez lambdę
}

```

W wyniku otrzymamy 8, 7, 8 i 8.

Oczywiście obiekty function są przydatne do tworzenia wywołań zwrotnych, przekazywania operacji jako argumentów itd.

33.6. Rady

1. Sekwencję wejściową definiuje para iteratorów — 33.1.1.
2. Sekwencję wyjściową definiuje jeden iterator; uważaj na dopełnienia — 33.1.1.
3. Dla każdego iteratora p przedział $\langle p, p \rangle$ jest pustą sekwencją — 33.1.1.
4. Zwracaj koniec sekwencji w celu oznaczania faktu nieznalezienia elementu — 33.1.1.
5. Iteratory traktuj jak bardziej ogólne i łagodniejsze wskaźniki — 33.1.1.
6. Używaj typów iteratorowych, np. `list<char>::iterator`, zamiast wskaźników w celu odnoszenia się do elementów kontenera — 33.1.1.
7. Używaj `iterator_traits` w celu zdobycia informacji o iteratorach — 33.1.3.
8. Przy użyciu `iterator_traits` można dokonywać dyspozycji w czasie komilacji — 33.1.3.
9. Używaj `iterator_traits` w celu wybrania optymalnego algorytmu na podstawie kategorii iteratora — 33.1.3.
10. `iterator_traits` to szczegół implementacyjny, którego najlepiej używać niejawnie — 33.1.3.
11. W celu wydobycia iteratora z iteratora odwrotnego należy użyć funkcji `base()` — 33.2.1.
12. W celu dodania elementów do kontenera można użyć iteratora wstawiającego — 33.2.2.
13. Za pomocą iteratora przenoszącego `move_iterator` operację kopowania można zamienić w operację przenoszenia — 33.2.3.
14. Upewnij się, że Twoje kontenery można przeglądać za pomocą zakresowej pętli `for` — 33.3.
15. Używaj funkcji `bind()` do tworzenia wariantów funkcji i obiektów funkcyjnych — 33.5.1.
16. Pamiętaj, że funkcja `bind()` wcześnie wyłuskuje referencje; jeśli chcesz to opóźnić, użyj funkcji `ref()` — 33.5.1.
17. Za pomocą funkcji `mem_fn()` lub lambdy można przekonwertować styl wywoływania `p->f(a)` na `f(p,a)` — 33.5.2.
18. Do tworzenia zmiennych mogących przechowywać różne wywoływalne obiekty używaj `function` — 33.5.3.