Funktory, zbiory, mapy, iteratory

Wykład 12

Predykaty

- Predykaty są specjalnym rodzajem funkcji (obiektu funkcyjnego)
  - Zwracają one wartości typu `true/false`
  - Służą one do określania kryterium sortowania, wyszukiwania, usuwania itp.
- Predykaty mogą być jedno- lub dwuargumentowe
  - Predykaty dla takiego samego wywołania zwracają zawsze tą samą wartość
  - Predykaty nie zmieniają swojego stanu wewnętrznego, np. wartość zwracania nie zależy od liczby wywołań
- Przykład cpp_12.1 i cpp_12.2

Obiekty funkcyjne

- Argumenty funkcyjne algorytmów nie muszą być funkcjami – mogą to być obiekty zachowujące się jak funkcje
  - Obiekty funkcyjne nazywamy inaczej funktorami
  - `fun(arg); //fun – klasa ze zdefiniowanym operatorem ()`
- Zalety funktorów
  - Obiekt funkcyjny posiada stan, możliwe są nawet różne stany inicjalizowane przed użyciem
  - Posiadają typ
  - Funktory często działają szybciej niż funkcje
- Przykładowe predefiniowane obiekty funkcyjne
  - Typowym obiektem funkcyjnym używającym operatora `<` jest predykat `less<T>` służący do określania kryterium sortowania
  - Podobnie `greater<T>` - operator `>`
- Przykład cpp_12.3

Obiekty funkcyjne

- Predefiniowane obiekty funkcyjne w nagłówku `<functional>`
  - `negate<T>()` oznacza `-param`
  - `plus<T>()` oznacza `param1 + param2`
  - `minus<T>()` oznacza `param1 - param2`
  - `multiplies<T>()` oznacza `param1 * param2`
  - `divides<T>()` oznacza `param1 / param2`
  - `modulus<T>()` oznacza `param1 % param2`
  - `equal_to<T>()` oznacza `param1 == param2`
  - `not_equal_to<T>()` oznacza `param1 != param2`
  - `less<T>()` oznacza `param1 < param2`
  - `greater<T>()` oznacza `param1 > param2`
  - `less_equal<T>()` oznacza `param1 <= param2`
  - `greater_equal<T>()` oznacza `param1 >= param2`
  - `logical_not<T>()` oznacza `!param`
  - `logical_and<T>()` oznacza `param1 && param2`
  - `logical_or<T>()` oznacza `param1 || param2`

Pary

- Klasa **`pair`** umożliwia potraktowanie dwóch wartości jako pojedynczego elementu
  - Jest wykorzystywana w kilku miejscach w bibliotece STL, a w szczególności w kontenerach **`map`** oraz **`multimap`**
- Klasa **`pair`** zdefiniowana jest w pliku nagłówkowym **`<utility>`**
  - **`std::pair<T1, T2> p;`**
  - Składowe **`p.first`** i **`p.second`**
  - Operatory **`==`** oraz **`<`**
    - Resztę można zdefiniować na ich podstawie
  - Szablon funkcji **`make_pair()`**

Zbiory i wielozbiory

- Kontenery **`set`** i **`multiset`** wykonują automatyczne sortowanie swoich elementów
  - Różnica polega na tym, iż wielozbiory dopuszczają powtórzenia, a zbiory nie
- W celu wykorzystywania zbiorów i wielozbiorów należy dołączyć odpowiedni nagłówek
  - **`#include <set>`**
- Domyślnym kryterium sortowanie jest operator **`<`** reprezentowany przez obiekt funkcyjny **`less`**

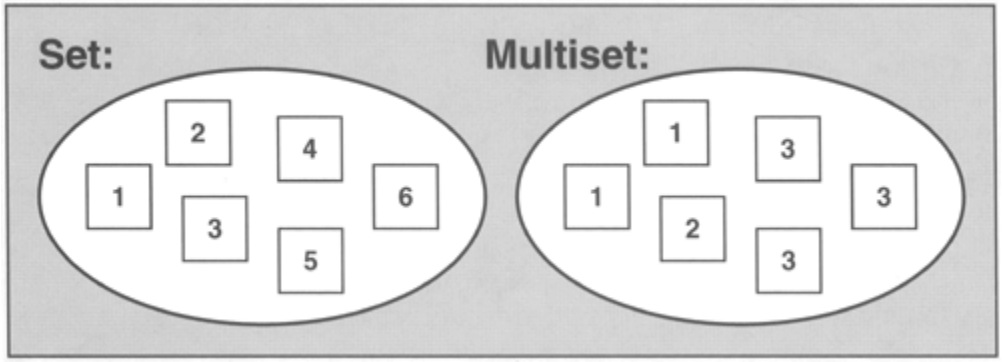
Kryterium sortowania

- Kryterium sortowania musi spełniać tzw. ścisłe uporządkowanie słabe
  - Musi być antysymetryczne
    - Jeżeli (x < y) jest prawdą, to (y < x) jest fałszywe
  - Musi być przechodnie
    - Jeżeli (x < y) jest prawdą oraz (y < z) też jest prawdą to (x < z) jest prawdziwe
  - Musi być niezwrotne
    - (x < x) zawsze jest fałszywe
- Kryterium sortowanie definiuje także równość
  - Dwa elementy są równe jeżeli żaden z nich nie jest mniejszy od drugiego

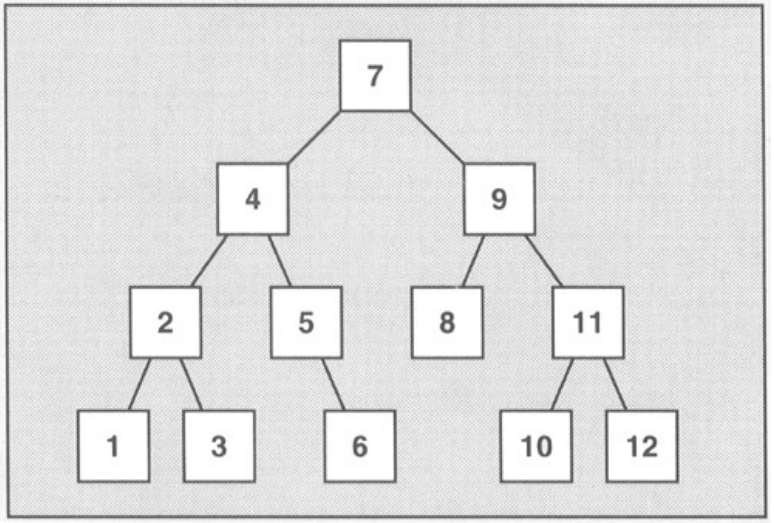
Możliwości

- Zbiory i wielozbiory zwykle implementowane są jako zrównoważone drzewa binarne (najczęściej drzewa czerwono-czarne)
- Największą zaletą automatycznego sortowania jest duża szybkość wyszukiwania
  - Wadą automatycznego sortowania jest niemożliwość bezpośredniej zmiany wartości elementu zbioru
  - Aby zmienić wartość trzeba najpierw element taki usunąć, a potem wstawić nowy
- Zbiory nie udostępniają operacji bezpośredniego dostępu do elementów
- Z punktu widzenia iteratorów elementy zbiory są stałe

Struktura



- Logiczna struktura zbiorów



- Wewnętrzna struktura zbiorów – drzewa binarne zrównoważone

Operacje tworzenia

- Operacje tworzenia
  - `set c` – tworzy pusty zbiór (wielozbiór)
  - `set c1 (c2)` – tworzy kopię zbioru
  - `set c(op)` – tworzy zbiór o kryterium sortowania `op`
  - `set c(beg, end)` – tworzy zbiór inicjalizowany elementami z zakresu `[beg, end)`
  - `set c(beg, end, op)` – tworzy zbiór inicjalizowany elementami z zakresu `[beg, end)` o kryterium sortowania `op`
- W powyższych przypadkach set może być
  - `set<T>` - zbiór sortowany kryterium `less<T>`
  - `set<T, op>` - zbiór sortowany kryterium `op`
  - `multiset<T>` - wielozbiór sortowany kryterium `less<T>`
  - `multiset<T, op>` - wielozbiór sortowany kryterium `op`
- Użycie kryterium sortowania jako parametru konstruktora umożliwia ustawienie tego kryterium w czasie wykonania programu
  - Ma znaczenie np. jeżeli potrzebne są różne kryteria sortowania dla danych tego samego typu

Operacje...

- Operacje niemodyfikujące
  - Logiczne (np.. `==`, `!=`)
  - `c.size()`, `c.empty()`, `c.max_size()` - wspólne z innymi kontenerami
- Operacje wyszukiwania
  - `c.count(elem)` - zwraca liczbę elementów o wartości `elem`
  - `c.find(elem)` - zwraca pozycję pierwszego elementu o wartości `elem` lub wartość `c.end()`
  - `c.lower_bound(elem)` - zwraca pierwszą pozycję, która jest większa bądź równa `elem`
  - `c.upper_bound(elem)` - zwraca pierwszą pozycję, która jest większa `elem`
  - `c.equal_range(elem)` - zwraca parę, która jest wynikiem wywołanie dwóch powyższych funkcji
  - `c.value_comp()` - zwraca obiekt, służący jako kryterium porównania (to samo co `c.key_comp()`)

Operacje…

- Operacje przypisania
  - `c1 = c2`
  - `c1.swap(c2)` – zamienia `c1` z `c2`
  - `swap(c1, c2)` – zamienia `c1` z `c2`
- Iteratory zbiorów są iteratorami dwukierunkowymi, nie można więc korzystać z algorytmów STL wymagających swobodnego dostępu
  - `c.begin()` – zwraca iterator dwukierunkowy dla pierwszego elementu
  - `c.end()` – zwraca iterator dwukierunkowy dla pozycji za ostatnim elementem
  - `c.rbegin()` – zwraca iterator odwrotny dla pierwszego elementu iteracji odwrotnej
  - `c.rend()` – zwraca iterator odwrotny dla pozycji za ostatnim elementem iteracji odwrotnej
  - Z punktu widzenia iteratorów wszystkie elementy zbirów są stałe, przez co nie można wywołać algorytmów modyfikujących

Wstawiane i usuwanie

- `c.insert(elem)` – wstawia kopię elementu, zwraca pozycję nowego elementu oraz dla zbioru czy operacja się powiodła
- `c.insert(pos, elem)` – wstawia kopię elementu, zwraca pozycję nowego elementu oraz dla zbioru czy operacja się powiodła, `pos` służy jak wskazówka gdzie zacząć poszukiwania
- `c.insert(beg, end)` – wstawia kopie elementów z zakresu `[beg, end)`, nic nie zwraca
- `c.erase(val)` – usuwa wszystkie elementy o wartości `val`, zwraca liczbę elementów usuniętych
- `c.erase(pos)` – usuwa element z pozycji iteratora `pos`, nic nie zwraca
- `c.erase(beg, end)` – usuwa elementy z zakresu `[beg, end)`, nic nie zwraca
- `c.clear()` – opróżnia zbiór

Obsługa wyjątków

- Zbiory są kontenerami opartymi na węzłach, czyli niepowodzenie we wstawieniu elementu nie zmienia kontenera
- Ponieważ z założenia destruktory nie zgłaszają wyjątków to usunięcie zawsze jest wykonywanie poprawnie
- W wypadku wieloelementowych operacji wstawienia powrót do stanu pierwotnego jest nierealny po zgłoszeniu wyjątku, gdyż kontener wymaga cały czas stanu wewnętrznego posortowania

Przykłady

- Użycie kontenera **`set`**
  - Przykład cpp_12.4
- Użycie kontenera **`multiset`**
  - Przykład cpp_12.5
- Określenie kryterium sortowania podczas wykonania programu
  - Przykład cpp_12.6

Mapy i multimapy

- Mapy i multimapy są kontenerami, w których elementy są parami klucz-wartość
- Ich elementy są automatycznie sortowane wg klucza
  - Mutlimapy dopuszczają powtórzenia wartości klucza, a mapy nie
- W celu wykorzystywania **`map`** i **`multimap`** należy dołączyć odpowiedni nagłówek
  - **`#include <map>`**
- Domyślnym kryterium sortowanie jest operator **`<`** reprezentowany przez obiekt funkcyjny **`less`**

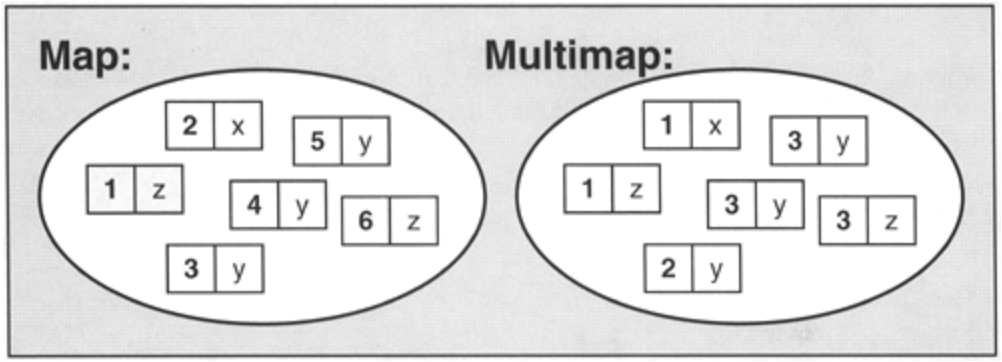
Wymagania w stosunku do map

- Pierwszym argumentem szablonu jest typ klucza, a drugim typ wartości
  - `map<string, double> c;`
  - Opcjonalny trzeci argument określa kryterium sortowania
- Elementy mapy mogą być dowolnych typów, które spełniają następujące wymagania
  - Para klucz-wartość musi umożliwiać operacje przypisania oraz kopiowania
  - Klucz musi być porównywalny za pomocą kryterium sortowania

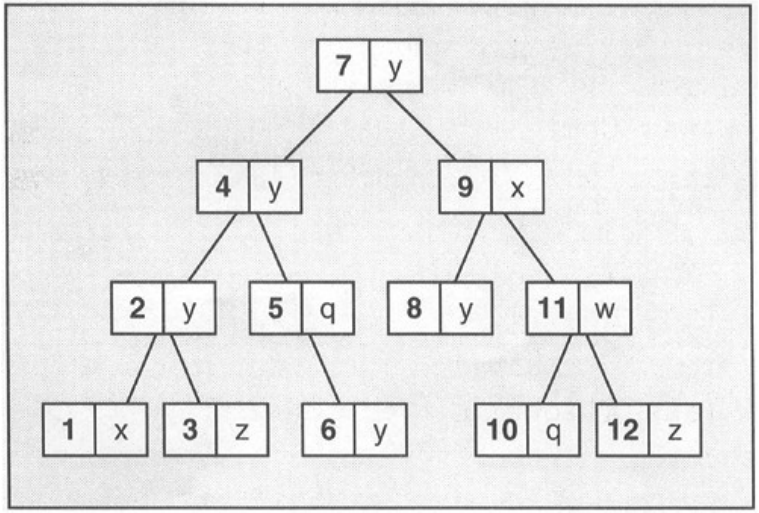
Możliwości

- Mapy i multimapy zwykle implementowane są tak jak zbiory
  - Zbiory i wielozbiory można traktować jak mapy i multimapy, w których klucz jest jednocześnie elementem
- Mapy i multimapy posiadają interfejs bardzo zbliżony do zbiorów i wielozbiorów
  - Elementami są pary, możliwość tworzenia tablic asocjacyjnych
  - Sortowanie jest automatyczne na podstawie kluczy
  - Szybkie wyszukiwanie wg klucza, natomiast wolne wg wartości
  - Nie udostępniają operacji bezpośredniego dostępu do elementów
  - Z punktu widzenia iteratorów klucze w mapach są stałe

Struktura



- Logiczna struktura map



- Wewnętrzna struktura map - drzewa zrównoważone

Operacje tworzenia

- Operacje tworzenia
  - `map c` – tworzy pustą mapę
  - `map c1 (c2)` – tworzy kopię mapy
  - `map c(op)` – tworzy mapę o kryterium sortowania `op`
  - `map c(beg, end)` – tworzy mapę inicjalizowaną elementami z zakresu `[beg, end)`
  - `map c(beg, end, op)` – tworzy mapę inicjalizowany elementami z zakresu `[beg, end)` o kryterium sortowania `op`
- W powyższych przypadkach `map` może być
  - `map<T, Elem>` - mapa sortowany kryterium `less<T>`
  - `map<T, Elem, op>` - mapa sortowany kryterium `op`
  - `multimap<T, Elem>` - wielozbiór sortowany kryterium `less<T>`
  - `multimap<T, Elem, op>` - wielozbiór sortowany kryterium `op`
- Użycie kryterium sortowania jako parametru konstruktora umożliwia ustawienie tego kryterium w czasie wykonania programu
  - Ma znaczenie np. jeżeli potrzebne są różne kryteria sortowania dla danych tego samego typu

Operacje...

- Operacje niemodyfikujące
  - Takie jak dla zbiorów
- Operacje wyszukiwania
  - `c.count(key)` – zwraca liczbę elementów o kluczu `key`
  - `c.find(key)` – zwraca pozycję pierwszego elementu o kluczu `key` lub wartość `c.end()`
  - `c.lower_bound(key)` – zwraca pierwszą pozycję, której klucz jest większy bądź równy `key`
  - `c.upper_bound(key)` – zwraca pierwszą pozycję, której klucz jest większy od `key`
  - `c.equal_range(key)` – zwraca parę, która jest wynikiem wywołanie dwóch powyższych funkcji
  - `c.value_comp()` – zwraca obiekt, służący jako kryterium porównania wartości
  - `c.key_comp()` – zwraca kryterium porównywania

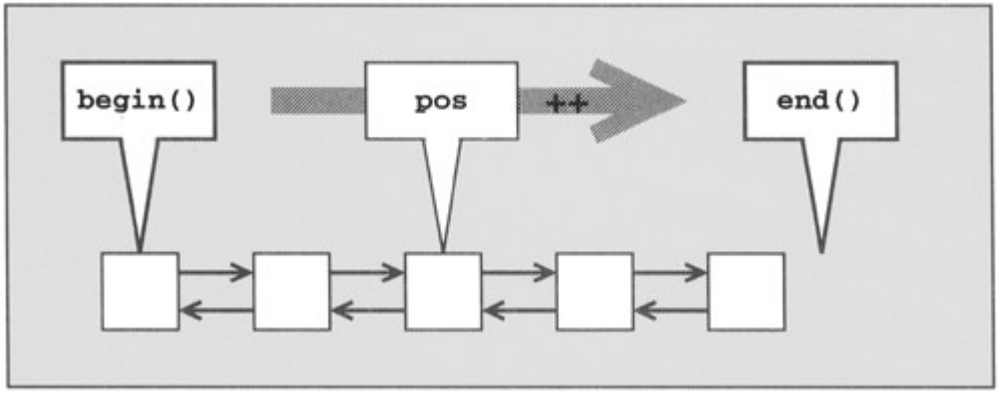
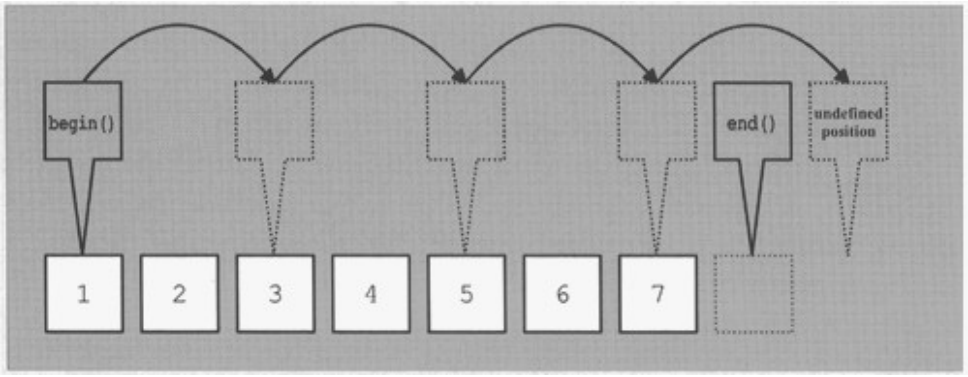
Operacje…

- Operacje przypisania
  - Takie jak dla zbiorów
- Funkcje iteratorowe
  - Takie jak dla zbiorów
- Funkcje wstawiające i usuwające
  - Takie jak dla zbiorów
  - Elementami są pary w miejsce wartości w zbiorach
  - Funkcja **`insert`** zwraca różne wartość dla **`map`** i **`multimap`**

Wstawianie par

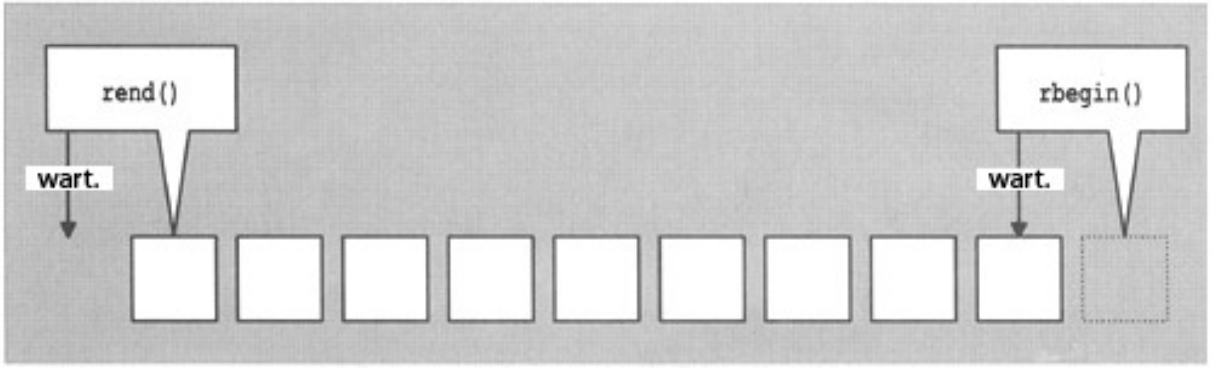
- Używają funkcji składowej **`insert`** należy pamiętać, że wstawiamy nie pojedynczą wartość, ale parę
- Dostępne możliwości
  - Użycie pary
    - `c.insert(pair<string, double>("data", 3.14159));`
  - Użycie funkcji **`make_pair()`**
    - `c.insert(make_pair("data", 3.14159));`
  - Użycie definicji typu **`value_type`**
    - `c.insert(map<string, double>::value_type("data", 3.14159));`
- Przykład cpp_12.7

Zastosowania map i multimap

- Wykorzystanie mapy jako tablicy asocjacyjnej
  - Kontenery asocjacyjne na ogół nie udostępniają bezpośredniego dostępu do elementów
    - Wyjątkiem są mapy, które nie są stałe – udostępniają operator `[]`
  - Tablica asocjacyjna to tak, w której kluczem może być dowolny typ, nie tylko liczba naturalna
    - ```
      tab["dana1"] = 15;
      tab["dana2"] = 56;
      ```
  - Indeks nie może być w takich sytuacjach nieprawidłowy (oczywiście jeżeli jest odpowiedniego typu)
- Przykład cpp_12.8 i cpp_12.9
      ```

Przegląd możliwości

	Vector	Deque	List	Set	MultiSet	Map	Multimap
Struktura wewnętrzna	Tablica dynamiczna	Tablica tablic	Lista dwukier.	Drzewo binarne	Drzewo binarne	Drzewo binarne	Drzewo binarne
Elementy	Wartość	Wartość	Wartość	Wartość	Wartość	Para	Para
Duplikaty	Tak	Tak	Tak	Nie	Tak	Nie - klucz	Tak
Dostęp swobodny	Tak	Tak	Nie	Nie	Nie	Tak - klucz	Nie
Kat. iteratorów	Swobodny	Swobodny	Dwukier-unkowy	Dwukier. (stałe elem.)	Dwukier. (stałe elem.)	Dwukier. (stałe klucz.)	Dwukier. (stałe klucz.)
Wyszukiwanie	Wolne	Wolne	Bardzo wolne	Szybkie	Szybkie	Szybkie – klucz	Szybkie - klucz
Szybkie wstawianie	Koniec	Początek i koniec	Wszędzie	-	-	-	-
Wstawianie unieważ. iter.	Realokacja	Zawsze	Nigdy	Nigdy	Nigdy	Nigdy	Nigdy
Zwalnia. pam.	Nigdy	Czasami	Zawsze	Zawsze	Zawsze	Zawsze	Zawsze
Bezpiecze-ństwo transakcyjne	Wstaw. i usuwan. na końcu	Wstaw. i usuwan. na początku i końcu	Wszystko bez sort() i operacji =	Wszystko bez wieloelem, wstawiania	Wszystko bez wieloelem, wstawiania	Wszystko bez wieloelem, wstawiania	Wszystko bez wieloelem, wstawiania

Kiedy stosować poszczególne kontenery?

- Domyślnie należy stosować wektor
  - Jest prosty, wygodny i elastyczny
  - Zapewnia bezpośredni dostęp do danych
  - Szybkość jest często wystarczająca
  - Może symulować zwykłą tablice
- Jeżeli planowane jest wstawianie i usuwanie elementów z początku należy użyć kolejki dwukierunkowej
  - Umożliwia zwalnianie nie używanych bloków pamięci
- Listę należy rozważyć jeżeli dokonujemy częstych zmian w środku kontenera
  - Jeżeli łączymy lub dzielimy dane
  - Wadą list jest brak bezpośredniego dostępu do elementów
  - Jest jednym z bezpieczniejszych kontenerów ze względu na obsługę wyjątków

Kiedy stosować poszczególne kontenery?

- Jeżeli potrzebujemy szybkiego wyszukiwania elementów to należy zastosować zbiór (wielozbiór) sortujący zgodnie z kryterium wyszukiwania
  - Jeszcze lepiej jest zastosować tablice mieszające (hash tables), które zapewniają wyszukiwanie jeszcze o rząd wielkości szybsze, ale nie ma ich w STL
- Do przechowywania par należy stosować mapy
  - Ewentualnie znowu tablice mieszające
- Tablica asocjacyjna – mapa
- Słownik – multimapa
- Jeżeli faktycznie zależy nam na efektywności najlepiej zaimplementować odpowiednią procedurę testową i dokonać sprawdzenia, jaki kontener nam najbardziej odpowiada
  - Przykład cpp_12.10 i cpp_12.11

Iteratory

- Iteratory są obiektami, które potrafią poruszać się po elementach kontenerów
  - Iteratory posiadają interfejs taki jak wskaźniki
  - Iteratory są zgodne z koncepcją czystej abstrakcji tzn. iteratorem jest wszystko co zachowuje się tak jak iterator
  - Iteratory dzieli się na kategorie
- Każdy kontener definiuje swoje własne iteratory
  - Nie trzeba dołączać żadnego nagłówka
- Jednak istnieją kilka specjalnych definicji iteratorów, np. iteratory odwrotne
  - Do ich używania potrzebny jest plik nagłówkowy `<iterator>`
  - Normalnie jest dołączany przez sam kontener

Kategorie iteratorów

Kategoria	Możliwości	Typy udostępniające
Iter. wejściowy ***Input iterator***	Odczyt w przód	Strumień wejściowy (`istream`)
Iter. wyjściowy ***Output iterator***	Zapis w przód	Strumień wyjściowy (`ostream`)
Iter. postępujący ***Forward iterator***	Odczyt i zapis w przód	Kombinacja dwóch powyższych
Iter. dwukierunkowy ***Bidirectional iterator***	Odczyt i zapis w przód i wstecz	Lista, zbiór, wielozbiór, mapa i multimapa
Iter. dostępu swob. ***Random access iterator***	Odczyt i zapis z dostępem swobodnym	Wektor, kolejka dwustronna, łańcuch

Iteratory dwukierunkowe

- Są uboższą wersją iteratorów dostępu swobodnego
- Udostępniają następujące operacje
  - `*iter, iter->skl, ++iter, iter++, iter1==iter2, iter1!=iter 2, iter1=iter2, --iter, iter--`



Iteratory dostępu swobodnego

- Iteratory te udostępniają tzw. arytmetykę iteratorów
  - Dodawanie, odejmowanie przesunięć, obliczanie różnic oraz porównywanie iteratorów za pomocą < lub >
- Operacje takie jak iteratorów dwukierunkowych oraz
  - `iter[n], iter+=n, iter-=n, iter+n, iter-n, iter1-iter2, iter1<iter2, iter1>iter2, iter1<=iter2, iter1>=iter2`
  - Należy pamiętać o zakresach!!!
- Przykład cpp_12.12



Funkcje pomocnicze dla iteratorów

- STL zapewnia trzy pomocnicze funkcje dla iteratorów
  - `void advance(pos, n)` – przesuwa pozycje iteratora o `n`
    - Dzięki zastosowaniu odpowiednich własności poszczególnych iteratorów wykonuje się zawsze w najszybszy możliwy sposób
      - Czyli dla wektora wykona `pos += n;`
      - Ale dla listy `n` razy `++n`
    - Dzięki temu w łatwy sposób możemy zmienić typ kontenera, jednocześnie zapewniając dużą wydajność
  - `long distance(pos1, pos2)` – oblicza odległość między iteratorami
    - Uwagi j.w.
  - `void iter_swap(pos1, pos2)` – służy do zmiany wartości do których odnoszą się dwa iteratory
- Przykład cpp_12.13 (uwaga dotycząca wektorów)

Adaptatory iteratorów

- Adaptator iteratorów to specjalne iteratory umożliwiające działanie algorytmom w trybie odwrotnym, trybie wstawiania oraz ze strumieniami
  - Iteratory odwrotne to adaptatory umożliwiające działanie odwrotne inkrementacji i dekrementacji
  - Iteratory wstawiające (wstawiacze) pozwalają algorytmom na zamianę operacji przypisania w operację wstawienia
  - Iteratory strumieniowe umożliwiają wykorzystanie algorytmów do pracy ze strumieniami

Iteratory odwrotne

- Deklaracje
  - `kontener::reverse_iterator`
  - `kontener::const_reverse_iterator`
- Metody
  - `rbegin()` – odpowiednik `begin()`, zwraca pozycje ostatniego elementu
  - `rend()` – odpowiednik `end()`, zwraca pozycję za pierwszym elementem
- Możliwa jest konwersja iteratorów na iteratory odwrotne
  - ```
    coll::iterator pos;
    coll::reverse_iterator rpos(pos);
    ```
  - Stąd
    - Wartością `rbegin()` jest `kontener::reverse_iterator(end())`
    - Wartością `rend()` jest `kontener::reverse_iterator(begin())`
- Możliwa jest konwersja iteratorów odwrotnych na iteratory
  - Funkcja składowa `base()`
    ```

Wartość, a iteratory odwrotne

- Jeśli dokonamy konwersji iteratora na iterator odwrotny to wartość pokazywana przez iterator ulegnie zmianie
  - Wynika to z zastosowania zakresów półotwartych
  - Zaleta jest taka, że nie trzeba zmieniać zakresów, które po konwersji dalej są poprawne
- Przykład cpp_12.14



Iteratory wstawiające

Nazwa	Klasa	Wywoływana funkcja	Tworzenie
Wstawiacz końcowy	`back_insert_iterator`	`push_back (val)`	`back_inserter (cont)`
Wstawiacz początkowy	`front_insert_iterator`	`push_front (val)`	`front_inserter (cont)`
Wstawiacz ogólny	`insert_iterator`	`insert (pos, val)`	`inserter (cont, pos)`

- Dany kontener musi udostępniać odpowiednią funkcję składową, aby móc używać wstawiacza
- Operacje
  - `iter = val; //wstawia wartość`
  - `*iter; ++iter; iter++; //rozkazy puste zwracające iter`
- Przykład cpp_12.15

Iteratory strumienia wyjściowego

- Iteratory strumienia wyjściowego
  - Zapisują przypisywane wartości do strumienia wyjściowego
  - Implementacja tych iteratorów jest bardzo zbliżona do implementacji iteratorów wstawiających
- Operacje
  - `osteram_iterator<T>(ostream)` – tworzy iterator strumienia wyjściowy dla `ostream`
  - `ostream_iterator<T>(ostream, del)` – tworzy iterator strumienia wyjściowy dla `ostream` wraz z separatorem `del` typu `const char*`
  - `iter = val; //zapisuje wartość do strumienia wyjściowego`
  - `*iter; ++iter; iter++; //rozkazy puste zwracające iter`
- Przykład cpp_12.16

Iteratory strumienia wejściowego

- Iteratory strumienia wejściowego
  - Odczytują wartości ze strumienia wyjściowego tak, że algorytmy mogą pracować na tych danych
  - Trochę bardziej skomplikowane w stosunku do iteratorów wyjściowych
- Operacje
  - `osteram_iterator<T>()` – tworzy iterator końca strumienia
  - `osteram_iterator<T>(istream)` – tworzy iterator strumienia wejściowego dla `istream` i odczytuje pierwszą wartość
  - `*iter` – zwraca aktualną wartość odczytaną wcześniej
  - `++iter` – odczytuje kolejną wartość i zwraca jej pozycję
  - `iter++` – odczytuje kolejną wartość i zwraca pozycję poprzednią
  - `iter1 == iter2` – wykonuje test równości iteratorów
  - `iter1 != iter2` – wykonuje test różności iteratorów
  - Dwa iteratory są równe jeżeli obydwa są iteratorami końca strumienia lub oba mogą prowadzić odczyt z tego samego strumienia
- Przykład cpp_12.17

Adaptatory funkcji

- Adaptator funkcji to obiekt funkcyjny pozwalający na łączenie obiektów funkcyjnych ze sobą nawzajem, z wartościami
- Adaptatory funkcji
  - `bind1st(op, value)` oznacza `op(value, param)`
  - `bind2nd(op, value)` oznacza `op(param, value)`
  - `not1(op)` oznacza `!op(param)`
  - `not2(op)` oznacza `!op(param1, param2)`
- Przykład cpp_12.18

Adaptatory funkcji...

- Adaptatory funkcji składowych
  - `mem_fun_ref(op)` – wywołuje `op()` jako funkcje składową obiektu
  - `mem_fun(op)` – wywołuje `op()` jako funkcje składową wobec wskaźnika do obiektu
  - Przykład cpp_12.19
- Adaptatory zwykłych funkcji
  - `ptr_fun(op)` oznacza `op(par)` lub `op(par1, par2)`
  - Umożliwiają wykorzystanie zwykłych funkcji z poziomu innych adaptatorów
  - Przykład cpp_12.20