Wydział Elektroniki
i Technik Informacyjnych
POLITECHNIKA WARSZAWSKA

Programowanie obiektowe
Biblioteka standardowa: elementy podstawowe

Krzysztof Gracki
kgr@ii.pw.edu.pl

pok. 312

Politechnika
Warszawska

Biblioteka rodzajowa STL

Historia

1985	A. Stepanov: biblioteka rodzajowa dla języka Ada
1992- 94	A. Stepanov i M.Lee: biblioteka STL dla C++ z szablonami
1998, 2003	Przyjęty standard języka C++ z biblioteką STL (ISO/IEC 14882)
2005; 07	TR1: ISO/IEC TR 19768:2007, C++ Library Extensions
2011	Biblioteka standardowa C++11

Przestrzeń programowania

```
sort()
binary_search()
transform() ...
```

Typy danych (d)

```
int, char*,
Fraction, ...
```

Kontenery (k)

```
vector, list,
set, map ...
```

Algorytmy (a)

Złożoność oprogramowania

Wszystkie kombinacje: ==> a*d*k wariantów kodu
Funkcje szablonowe: ==> a*k wariantów kodu
Zastosowanie iteratorów: ==> a + k wariantów kodu

Programowanie generyczne – główne zalety

- Jednorodność koncepcyjna
  - Ogólne pojęcie **kontenera** ze standardowym interfejsem
  - Ogólne pojęcie **iteratora**
  - Ogólne pojęcie **algorytmu rodzajowego** (generycznego)
- Kod wielokrotnego użycia
  - Jeden szablon, wiele konkretyzacji
- Możliwość dostosowywania kodu do środowiska aplikacji
  - Zastosowanie specjalnych szablonów cech (traits)
- Efektywność kodu
  - Szablony są konkretyzowane w czasie kompilacji
  - Umożliwiają statyczną kontrolę typów
  - Mechanizm szablonów jest ortogonalny względem dziedziczenia
  - Dają się stosować do typów podstawowych

Biblioteka standardowa (przestrzeń std::)

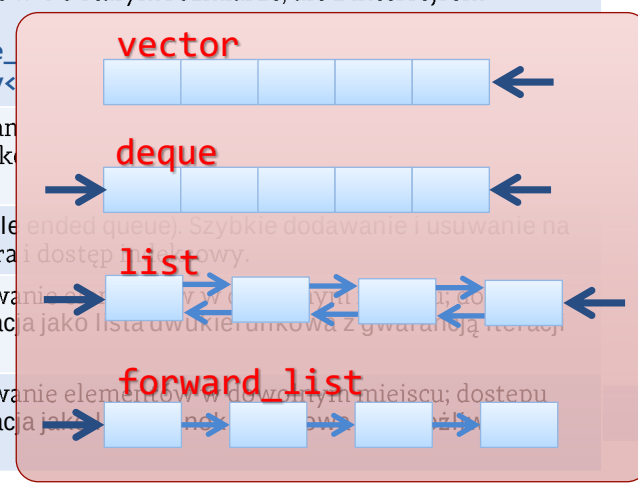
Komponenty biblioteki standardowej	niektóre pliki nagłówkowe
Biblioteka kontenerów	<array>,<list>,<queue>,<map>
Biblioteka iteratorów	<iterator>
Biblioteka algorytmów	<algorithm>, <cstdlib>
Biblioteka łańcuchów znakowych	<string>, <cctype>, <cwchar>
Biblioteka WE/WY	<iostream>, <ios>, <iomanip>
	<filesystem>(C++17), <print> (C++23)
Biblioteka diagnostyczna	<stdexcept>, <cassert>
Biblioteka usług podstawowych języka	<limits>, <cstdlib>, <new>
Biblioteka akcesoriów pomocniczych	<utility>, <tuple>, <memory>, <chrono>,
	<optional>, <variant> (C++17)
Biblioteka lokalizacji regionalnych	<locale>
Biblioteka numeryczna	<complex>, <valarray>
Biblioteka wyrażeń regularnych	<regex>
Biblioteka operacji niepodzielnych	<atomic>
Biblioteka obsługi współbieżności	<thread>, <mutex>,
	<semaphore> (C++20)

Kontenery sekwencyjne

Deklaracja generyczna: `template<class T, class A = std::allocator<T>> class kontener;`	
`<array>` [C++11]	Realizuje semantykę tablic w C o stałym rozmiarze, ale z interfejsem kontenerów: `template<class T, size_t N> struct array;` `array<int, 5> A; array<array<int, 5>, 7> T;`
`<vector>`	Wektor rozszerzalny dynamicznie. Szybki dostęp indeksowy oraz dodawanie i usuwanie elementów na końcu. Reprezentacja zwarta. Specjalizacja `vector<bool>.`
`<deque>`	Kolejka dwustronna (double ended queue). Szybkie dodawanie i usuwanie na obydwu końcach kontenera i dostęp indeksowy.
`<list>`	Szybkie wstawianie / usuwanie elementów w dowolnym miejscu; dostępu indeksowego brak. Realizacja jako lista dwukierunkowa z gwarancją iteracji dwukierunkowej.
`<forward_list>` [C++11]	Szybkie wstawianie / usuwanie elementów w dowolnym miejscu; dostępu indeksowego brak. Realizacja jako lista jednokierunkowa bez możliwości iteracji dwukierunkowej.

Kontenery sekwencyjne

Deklaracja generyczna: `template<class T, class A = std::allocator<T>> class kontener;`	
`<array>` [C++11]	Realizuje semantykę tablic w C o stałym rozmiarze, ale z interfejsem kontenerów: `template<class T, size_` `array<int, 5> A; array<`
`<vector>`	Wektor rozszerzalny dynan i usuwanie elementów na k `vector<bool>.`
`<deque>`	Kolejka dwustronna (double obydwu końcach kontenera
`<list>`	Szybkie wstawianie / usuwa indeksowego brak. Realizac dwukierunkowej.
`<forward_list>` [C++11]	Szybkie wstawianie / usuwa indeksowego brak. Realizac iteracji dwukierunkowej.



vector czy array?


```
class Generator{
   std::uniform_int_distribution<int> dist{ 0, 9 };
   std::default_random_engine generator{ std::random_device()()};
public:
  auto operator()() {
    return dist(generator);
  }
};

int main() {
  std::array<int, 100> A{};
  std::vector<int> V(100);
  Generator gen;
  std::generate(V.begin(), V.end(), gen);


  std::cout << "\nA[" <<sizeof(A) << "]:";
  for (auto i : A)    std::cout << i;

  std::cout << "\nV[ " << sizeof(V) << "]:";
  for (auto i : V)   std::cout << i;

}
```

Co z przenoszeniem? (move semantics)

```
std::uniform_int_distribution<int> dist{ 0, 9 };
std::default_random_engine generator{
    std::random_device()() };
auto gen = [&]() { return dist(generator);  };
```

```
A[400]:00000000000000000000000000000000...
V[ 32]:72676825224295894042098417679510 52...
```

Politechnika Warszawska

Kontenery asocjacyjne

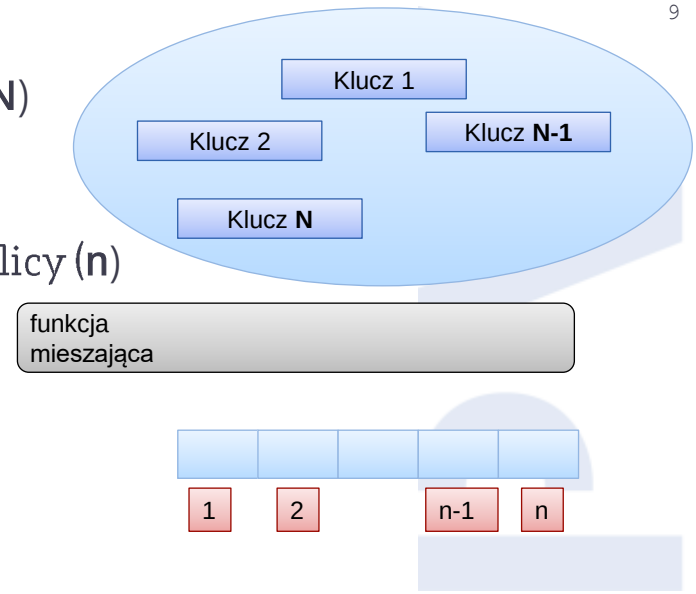

```
template<class K, class Cmp=less<K>, class A= allocator<K> > class set; // multiset
template<class K, class T, class Cmp = less<K>,
    class A=allocator<pair<const K,T>>> class map; // multimap
template<class K, class H=hash<K>, class KEq=equal_to<K>,
    class A=allocator<K>> class unordered_set;      // unordered_multiset
template<class K, class T, class H = hash<K>, class KEq = equal_to<K>,
    class A=allocator<pair<const K, T> > > class unordered_map; // unordered_multimap
```

<set>	Klasa set: kontener asocjacyjny uporządkowany. Dostęp do elementu wg **unikalnego** klucza. Reprezentacja: drzewo zrównoważone (np. czerwono-czarne) gwarantujące logarytmiczny czas dostępu. Klasa multiset: podobnie, ale bez unikalności klucza.
<map>	Klasa map: kontener asocjacyjny uporządkowany przechowujący pary <K, T> wg unikalnych kluczy. Reprezentacja i dostęp - jak dla klasy set. Klasa multimap: podobnie, ale bez unikalności klucza dla par <K, T>.
<unordered_set>	Kontener asocjacyjny z haszowaniem [C++11]
<unordered_map>	Kontener asocjacyjny z haszowaniem [C++11]

Politechnika Warszawska

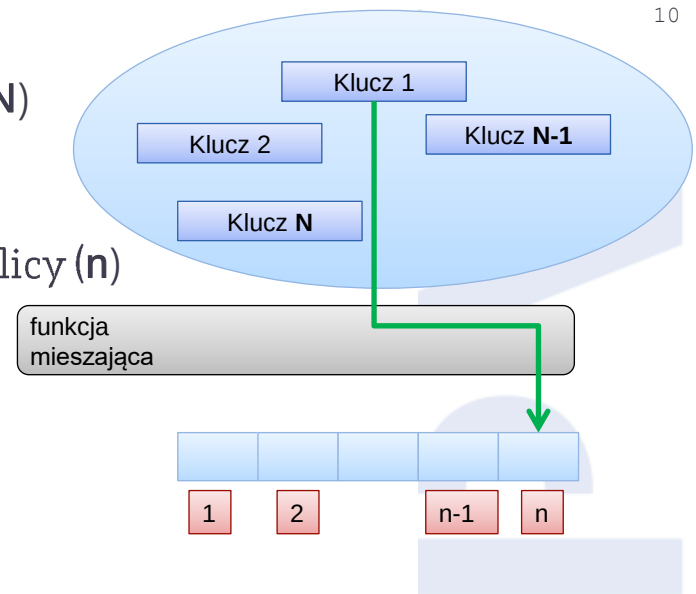
Kontenery z haszowaniem

- duża przestrzeń kluczy (**N**) np. *Imię i Nazwisko*
- funkcja haszująca
- przestrzeń indeksów tablicy (**n**)
- **N** >> **n**



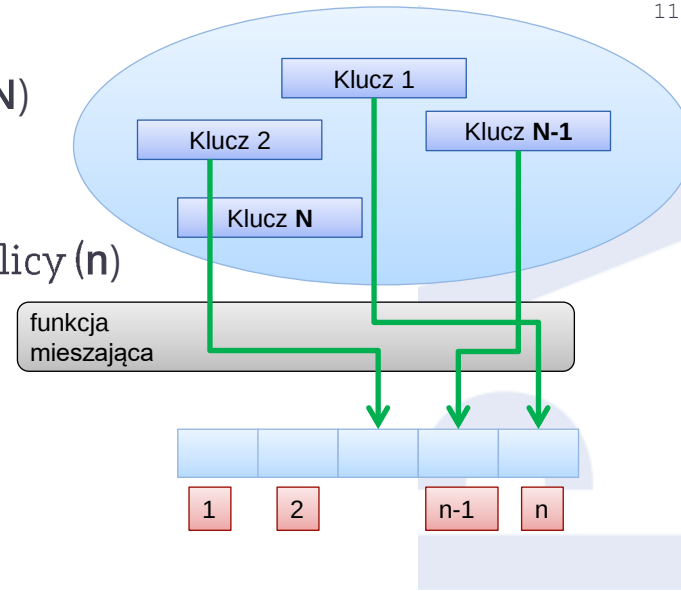
Kontenery z haszowaniem

- duża przestrzeń kluczy (**N**) np. *Imię i Nazwisko*
- funkcja haszująca
- przestrzeń indeksów tablicy (**n**)
- **N** >> **n**



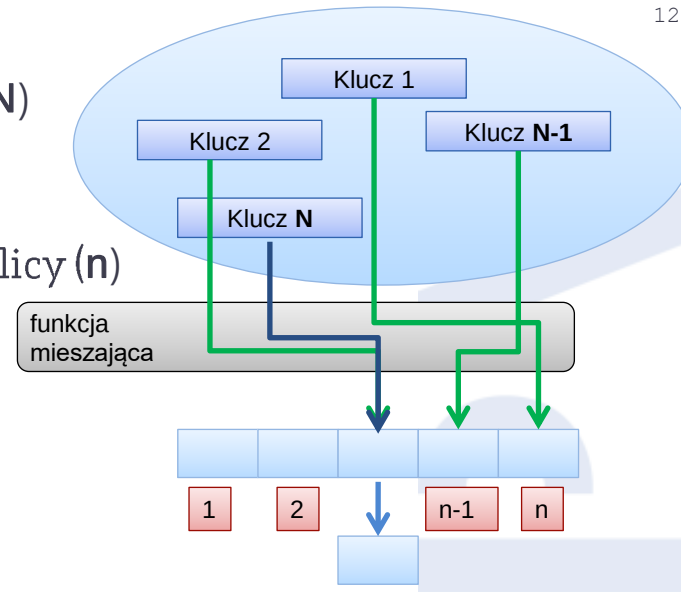
Kontenery z haszowaniem

- duża przestrzeń kluczy (**N**) np. *Imię i Nazwisko*
- funkcja haszująca
- przestrzeń indeksów tablicy (**n**)
- **N** >> **n**



Kontenery z haszowaniem

- duża przestrzeń kluczy (**N**) np. *Imię i Nazwisko*
- funkcja haszująca
- przestrzeń indeksów tablicy (**n**)
- **N** >> **n**



Kontenery adaptowane


```
template<class T, class Cont = deque<T> > class stack; // class queue

template<class T, class Cont = vector<T>,
    class Cmp = less<typename Cont::value_type> > class priority_queue;
```

`<stack>`	Klasa **`stack`**: dynamiczny kontener realizujący protokół stosu. Podległy kontener sekwencyjny (domyślnie **`deque`**) ma interfejs dostosowany do potrzeb manipulacji stosowych (operacje: **`push`**, **`top`**, **`pop`**, **`empty`**).
`<queue>`	Klasa **`queue`**: dynamiczny kontener realizujący protokół kolejki. Jest to proste dostosowanie kontenera sekwencyjnego (domyślnie **`deque`**) do standardowego interfejsu kolejki (operacje: **`push`**, **`pop`**, **`front`**, **`back`**, **`empty`**). Klasa **`priority_queue`**: wykorzystuje strukturę kopca (inna nazwa: sterta, heap) nałożoną na kontener sekwencyjny (domyślnie **`vector`**) do efektywnej realizacji kolejki priorytetowej. Interfejs: **`push`**, **`pop`**, **`front`**, **`back`**, **`empty`**. Wewnętrznie wykorzystuje operacje na stercie z biblioteki **`<algorithm>`** (**`push_heap`**, **`pop_heap`**, **`make_heap`**, ...).

Kontenery sekwencyjne (cd1)

- Na pewnym poziomie abstrakcji łańc
  traktowane także jako specyficzne k
  ostream, ...).
- Różne charakterystyki wynikają z ro
  iteratora.

- jest miejsce (przypadek optymistyczny):
  - dodanie elementu to proste wstawienie O(1)
- brak miejsca O(N)
  - zaalokowanie pamięci (można na zapas)
  - skopiowanie elementów
  - dodanie nowego elementu

Efektywność kontenerów sekwencyjnych

Kontener	*Czas wstawiania / usuwania*		
	na początku	***w środku***	***na końcu***
`vector`	liniowy	liniowy	stały amortyzowany
`list`	stały	stały	stały
`deque`	stały amortyzowany	liniowy	stały amortyzowany
`forward_list`	stały	stały	stały

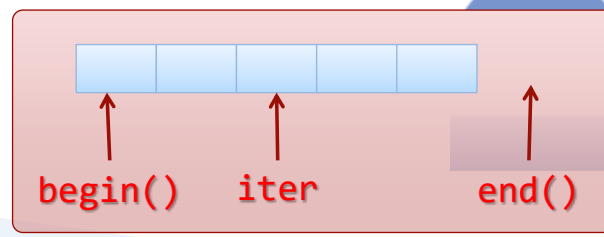
Iteratory

Iterator to obiekt, służący do uniwersalnego nawigowania w kontenerach

Obsługuje wybrane operatory, np:

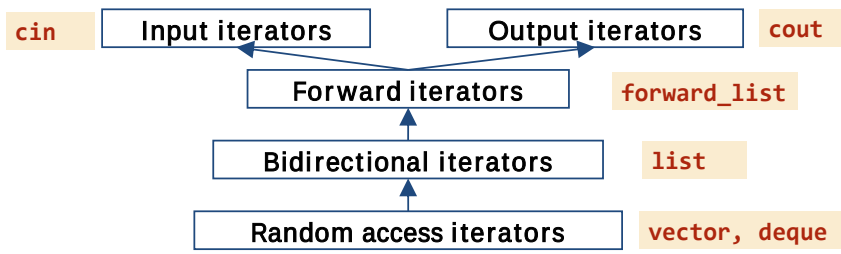
`*` `->` `++` `--` `==` `!=` `=`

Podstawowe funkcje kontenerów związane z iteratorami, np.:
`begin( ), end( )`



Iteratory

- 5 kategorii podstawowych



Specjalizacje

- `const iterators` – pozwalają odwiedzać kontenery ustalone
- `reverse iterators` – odwracają porządek odwiedzin
- iteratory programisty, w tym dziedziczące względem klasy `std::iterator`:

```
template<class Category, class T, class Dist = ptrdiff_t,
         class Ptr = T*, class Ref = T& > struct iterator;
```

Wymagania dla iteratorów

Input iterators
- konstruktor
- operator przypisania
- operatory równości / nierówności
- operator dereferencji
- pre/post inkrementacja

Output iterator
- konstruktor
- operator przypisania
- operator dereferencji
- pre/post inkrementacja

Forward iterator
- konstruktor
- operator przypisania
- operatory równości / nierówności
- operator dereferencji
- pre/post inkrementacja

Bidirectional iterator (dodatkowo)
- pre/post dekrementacja

Random Access iterator
- operator+ (int)
- operator+= (int)
- operator- (int)
- operator-= (int)
- operator- (random access iterator)
- operator[] (int)
- operator < (random access iterator)
- operator > (random access iterator)
- operator >= (random access iterator)
- operator <= (random access iterator)

Interfejs kontenerów sekwencyjnych (podzbiór)

Metoda	vector	deque	list	array	forward_list
begin, cbegin	√	√	√	√	√ (c)before_begin
end, cend	√	√	√	√	√
rbegin, crbegin	√	√	√	√	
rend, crend	√	√	√	√	
size	√	√	√	√	
max_size	√	√	√	√	√
empty	√	√	√	√	√
resize	√	√	√		√
front	√	√	√	√	√
back	√	√	√	√	
operator[], at	√	√		√	
assign	√	√	√	√	√
insert	√	√	√	√	insert_after
erase	√	√	√	√	erase_after
push_, pop_back	√	√	√	√	
push_, pop_front		√	√	√	√
clear	√	√	√	√	√
swap	√	√	√	√	√

Interfejs kontenerów asocjacyjnych (podzbiór)

Metoda	set	multiset	map	multimap	unordered _set	unordered _multiset	unordered _map	unordered _multimap
(c,r)begin	√	√	√	√	(c)begin	(c)begin	(c)begin	(c)begin
(c,r)end	√	√	√	√	(c)end	(c)end	(c)end	(c)end
size, max_size	√	√	√	√	√	√	√	√
empty	√	√	√	√	√	√	√	√
operator[], at			√				√	
emplace	√	√	√	√	√	√	√	√
insert	√	√	√	√	√	√	√	√
erase	√	√	√	√	√	√	√	√
clear	√	√	√	√	√	√	√	√
swap	√	√	√	√	√	√	√	√
count	√	√	√	√	√	√	√	√
find	√	√	√	√	√	√	√	√
equal_range	√	√	√	√	√	√	√	√
lower_bound	√	√	√	√				
upper_bound	√	√	√	√				

Politechnika Warszawska

Interfejsy publiczne 3 kontenerów różnych kategorii

Typy skojarzone (przez typedef)	Funkcje interfejsu publicznego vector<>	list<>	set<>
allocator_type pointer const_pointer reference const_reference reverse_iterator const_reverse_iterator difference_type iterator const_iterator size_type value_type --- **(Dla set<T>)** key_type value_compare key_compare	begin end rbegin rend clear empty erase get_allocator insert max_size size swap --- assign back front pop_back push_back resize --- capacity reserve at operator[]	begin end rbegin rend clear empty erase get_allocator insert max_size size swap --- assign back front pop_back, push_back resize --- pop_front, push_front remove, remove_if merge, sort, splice unique, reverse	begin end rbegin rend clear empty erase get_allocator insert max_size size swap --- count equal_range find lower_bound upper_bound value_comp key_comp

Standaryzacja nazw typów pozwala jednorodnie pracować z różnymi kategoriami kontenerów.

Kontener vector<T>


```
template <class T, class Alloc = std::allocator<T> >
class vector
{
public:
  typedef T value_type;
  typedef value_type* pointer;
  typedef const value_type* const_pointer;
  typedef value_type* iterator;
  typedef const value_type* const_iterator;
  typedef value_type& reference;
  typedef const value_type& const_reference;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;
  typedef reverse_iterator<const_iterator>
    const_reverse_iterator;
  typedef reverse_iterator<iterator> reverse_iterator;
```

Sekcja ustanawia standardowe nazwy typów skojarzonych

Kontener vector<T> (cd1)


```
protected:
  typedef simple_alloc<value_type, Alloc> data_allocator;
  iterator start;
  iterator finish;
  iterator end_of_storage;
  void insert_aux(iterator position, const T& x);
  void deallocate();
  void fill_initialize(size_type n, const T& value);
  // ... kilka dalszych składowych chronionych

public:
  iterator       begin()       { return start; }
  const_iterator begin() const { return start; }
  iterator       end()         { return finish; }
  const_iterator end() const   { return finish; }
```

Kontener vector<T> (cd2)


```
reverse_iterator rbegin()
  { return reverse_iterator(end()); }
const_reverse_iterator rbegin() const  //cbegin() [C++11]
  { return const_reverse_iterator(end()); }
reverse_iterator rend()
  { return reverse_iterator(begin()); }
const_reverse_iterator rend() const   //crend() [C++11]
  { return const_reverse_iterator(begin()); }
size_type size() const
  { return size_type(end() - begin()); }
size_type max_size() const
  { return size_type(-1) / sizeof(T); }
size_type capacity() const
  { return size_type(end_of_storage - begin()); }
bool empty() const { return begin() == end(); }
reference operator[](size_type n)
  { return *(begin() + n); }
const_reference operator[](size_type n) const
  { return *(begin() + n); }
```


Kontener vector<T> (cd3)


```
// Konstruktory, destruktor
  vector():start(0), finish(0), end_of_storage(0) {}
  vector(size_type n, const T& value)
    { fill_initialize(n, value); }
  vector(int n, const T& value)
    { fill_initialize(n, value); }
  vector(long n, const T& value)
    { fill_initialize(n, value); }
  explicit vector(size_type n)
    { fill_initialize(n, T()); }
  vector(const vector<T, Alloc>& x)
  { start=allocate_and_copy(x.end() - x.begin(),
          x.begin(), x.end());
    finish = start + (x.end() - x.begin());
    end_of_storage = finish;
  }
```

Kontener vector<T> (cd4)


```
// Konstruktory, destruktor (cd)
 vector(const iterator first, const_iterator last)
  { size type n = 0;
    distance(first, last, n);
    start = allocate and_copy(n, first, last);
    finish = start + n;
    end_of_storage = finish;
  }
 ~vector()
  { destroy(start, finish);
    deallocate();
  }
  vector<T, Alloc>& operator=(const vector<T, Alloc>& x);
  void reserve(size type n);
  reference front()           { return *begin(); }
  const reference front()const{ return *begin(); }
  reference back()            { return *(end() - 1); }
  const reference back()const { return *(end() - 1); }
  void push_back(const T& x);
```


Kontener vector<T> (cd5)


```
  void swap(vector<T, Alloc>& x);
  iterator insert(iterator position, const T& x);
  iterator insert(iterator position)
    { return insert(position, T()); }
  void insert(iterator position,
              const_iterator first,
              const_iterator last);
  void insert(iterator pos, size_type n, const T& x);
  void insert(iterator pos, int n, const T& x)
    { insert(pos, (size_type) n, x); }
  void pop_back() { --finish; destroy(finish); }
  iterator erase(iterator position);
  iterator erase(iterator first, iterator last);
  void resize(size_type new_size, const T& x);
  void resize(size_type new_size);
  void clear() { erase(begin(), end()); }
```

Przykład 1: kontenery sekwencyjne


```
#include <iostream>
#include <vector>
#include <deque>
#include <algorithm>
using namespace std;
#define P(x) cout<<#x ": " <<(x)<<endl

template <typename T>
ostream& operator<<(ostream &os, const vector<T>& v)
{ typedef /*typename*/ vector<T>::const_iterator vciter;
  for(vciter i=v.begin(); i!=v.end(); ++i)  os<<*i;
  return os;
}
template <typename T>
ostream& operator<<(ostream &os, const deque<T>& v)
{ typedef deque<T>::const_iterator dciter;
  for(dciter i = v.begin(); i!= v.end(); ++i) os<<*i;
  return os;
}
```


Przykład 1: kontenery sekwencyjne (cd1)


```
int main()
{ int A[] = { 3,1,4,1,5,9,2,6,5,3 };
  vector<int> v;
  P(sizeof(v));
  P(v.size());
  P(v.capacity());
  P(v.empty());

  v.insert(v.begin(), 3);
  //v[v.size()] = 1; // Błąd: poza zakresem
  v.push_back(1);
  P(v);

  vector<int> w(A, A + sizeof A / sizeof A[0]);
  P(w);
  P(sizeof(w));
  P(w.size());
  P(w.capacity());
```

```
sizeof(v): 20
v.size(): 0
v.capacity(): 0
v.empty(): 1
```

```
v: 31
```

```
w: 3141592653
sizeof(w): 20
w.size(): 10
w.capacity(): 10
```

Przykład 1: kontenery sekwencyjne (cd2)


```
vector<int> y(w.rbegin(), w.rend());
P(y);
sort(y.begin(), y.end());
P(y);

vector<int> x;
for(int i=0; i<1000; ++i)
  x.push_back(i+1);
P(x.size());
P(x.capacity());

x.assign(v.begin(), v.end());
x.insert(x.end(), 3, 9); // Na końcu 3 dziewiątki
x.pop_back(); // Usuń ostatni element
P(x);

deque<int> deq(x.begin(), x.end());
deq.push_front(0);  // zero na początku
P(deq.size());
P(deq);
```

```
y: 3562951413
y: 1123345569
```

```
x.size(): 1000
x.capacity(): 1066
```

```
x: 3199
```

```
deq.size(): 5
deq: 03199
```


Przykład 1: kontenery sekwencyjne (cd3)


```
// Pojemność i rozmiar przy kopiowaniu
// Można zarezerwować pojemność kontenera
// Zmniejszenie pojemności nadmiarowej przez kopiowanie
vector<int> vi1;
vi1.reserve(100);
vi1.push_back(1);
P(vi1.capacity());
P(vi1.size());
vi1.clear();
P(vi1.capacity());
P(vi1.size());

vi1.reserve(100);
vi1.push_back(1);
vector<int> vi2=vi1;
P(vi2.capacity());
P(vi2.size());
vector<int>(vi1).swap(vi1);  //self-swap, obcina pojemność
P(vi1.capacity());
P(vi1.size());
}
```

```
vi1.capacity(): 100
vi1.size(): 1
```

```
vi1.capacity(): 100
vi1.size(): 0
```

```
vi2.capacity(): 1
vi2.size(): 1
```

```
vi1.capacity(): 1
vi1.size(): 1
```

Przykład 2: kontenery sekwencyjne i set<T>


```
#define P(x) cout<<#x ": "<<(x)<<endl
// Odpowiednie pliki nagłówkowe dla kontenerów sekwencyjnych
#include <set>
#include <unordered_set>
using namespace std;
// Generyczny operator<< dla kontenerów sekwencyjnych
template<class T,
  template<class E, class A=allocator<E> > class C> // C: kontener
ostream& operator<<(ostream& os, const C<T>& cont)
{ os<<"( "; // Opakowanie
  for(const auto& e: cont) os<<e<<' ';
  return os<<')';
}
template<class C> // C: kontener definiujący zakres [begin ... end)
void show(const C& cont, const string& s="")
{
  cout<<s<<": < ";
  for(const auto& elem: cont)
    cout<<elem<<' ';
  cout<<">"<<endl;
}
```


Przykład 2: kontenery sekwencyjne i set<T> (cd1)


```
int main()
{ string s("abcdefg");
  vector<int> vi;
  list<int>   li;
  deque<int>  di;
  set<int>    si;
  multiset<int> mi;
  unordered_set<int> ui;
  queue<int>  qi;
  stack<int>  st;

  for(int i=0; i<10; ++i)
  { int e = rand()%9;
    vi.push_back(e);
    li.push_front(e);
    di.push_back(e);
    si.insert(e);
    mi.insert(e);
    ui.insert(e);
    qi.push(e);
    st.push(e);
  }
  P(s);
  P(vi);
  P(li);
  P(di);
  show(s, "s");
  show(vi, "vi");
  show(si, "si");
  show(mi, "mi");
  show(ui, "ui");

  //show(qi, "qi"); // Brak iteratora
  //show(st, "st"); // Brak iteratora
}
```

```
s: abcdefg
vi: ( 5 8 7 4 8 1 3 0 7 2 )
li: ( 2 7 0 3 1 8 4 7 8 5 )
di: ( 5 8 7 4 8 1 3 0 7 2 )
s: < a b c d e f g >
vi: < 5 8 7 4 8 1 3 0 7 2 >
si: < 0 1 2 3 4 5 7 8 >
mi: < 0 1 2 3 4 5 7 7 8 8 >
ui: < 5 0 8 7 4 1 3 2 >
```

Algorytmy w <algorithm>

- Algorytmy są szablonami funkcji ==> mogą być konkretyzowane na nieograniczoną liczbę sposobów
- Większość algorytmów operuje na zakresach iteratorów ==> działają pośrednio na zawartości kontenerów
- Stosowane intuicyjne nazewnictwo ==> określa ogólną rolę algorytmu
  - Przyrostek `_if` oznacza operację sterowaną predykatem (np. `find()` ==> znajdź zadany obiekt; `find_if()` ==> znajdź wg kryterium).
  - Przyrostek `_copy` oznacza dodatkowe kopiowanie obiektów
- Klasyfikacja ogólna algorytmów (ze względu na stan kontenera):
  - Niemodyfikujące – pozostawiają elementy bez zmiany (15)
  - Modyfikujące – mogą zmieniać elementy
  - Usuwające – podkategoria modyfikujących
  - Reorganizujące (mutating) – mogą zmieniać ustawienie w kontenerze
  - Porządkujące – podkategoria poprzedniej
  - Manipulacje zakresami uporządkowanymi -
  - Numeric algorithms

1. Algorytmy niemodyfikujące

`for_each()`	Operacja na każdym elemencie (może być modyfikująca)
`count()`	Liczba elementów
`count()_if()`	Liczba elementów spełniających predykat
`min_element()`	Najmniejszy w zakresie
`max_element()`	Największy w zakresie
`find()`	Znajdź element wg wartości
`find_if()`	Znajdź element wg predykatu
`search_n()`	Znajdź n kolejnych elementów wg kryterium
`search()`	Znajdź pierwsze wystąpienie elementu lub podzakresu
`find_end()`	Znajdź ostatnie wystąpienie elementu lub podzakresu
`find_first_of()`	Znajdź pierwsze wystąpienie jednego z elementów
`adjacent_find()`	Znajdź sąsiadującą parę wg zadanego kryterium
`equal()`	Bada, czy 2 zakresy są równe
`mismatch()`	Zwraca parę iteratorów desygnujących 2 różne elementy w 2 zakresach
`lexicographical_compare()`	Porównanie leksykograficzne 2 zakresów

1. Algorytmy niemodyfikujące - przykład


```
void collatz(int& n){ n=(n&1)? 3*n+1:n/2; }// Modyfikuje
void show(int n){ cout<<n<<' '; }
bool even(int n){ return !(n&1); }
vector<int> v;
// ....
show(v, "v");// v: 3 1 4 1 5 9 2 6 5 3

for_each(v.begin(), v.end(), collatz);

show(v, "v");// v: 10 4 2 4 16 28 1 3 16 10

int n10   = count   (v.begin(), v.end(), 10);
int neven = count_if(v.begin(), v.end(), even);
cout<<"num10: "<< n10 << endl;  // num10: 2
cout<<"even: "<< neven << endl; // even: 8

// ...
```

p. przykład kontenery sekwencyjne

2. Algorytmy modyfikujące i usuwające

copy()	Kopiowanie zakresu (od pierwszego elementu)
copy _backward()	Kopiowanie zakresu (od ostatniego elementu)
transform()	Transformacje zakresu (funkcje jedno i dwuargumentowe)
merge()	Połącz 2 zakresy uporządkowane
swap_ranges()	Wymiana elementów 2 zakresów
fill()	Wypełnij zakres daną wartością
fill_n()	Wypełnij n elementów daną wartością
generate()	Wypełnij zakres wynikiem operacji
generate_n()	Wypełnij n elementów wynikiem operacji
replace()	Zamień zadane elementy na nowe
replace()_if()	Zamień elementy spełniające predykat na nowe
replace_copy()	Zamień zadane elementy i skopiuj zakres
replace_copy_if()	Zamień elementy spełniające predykat i skopiuj zakres
remove()	Usuń elementy o zadanej wartości
remove_if()	Usuń elementy wg predykatu
remove_copy()	Kopiuj elementy różne od zadanego
remove_copy()_if()	Kopiuj elementy niespełniające predykatu
unique()	Usuń duplikaty sąsiadujące (elementy == poprzednikowi)
unique_copy()	Kopiuj pozostałość po usunięciu duplikatów

3. Algorytmy reorganizujące i sortujące

`reverse()`	Odwróć porządek elementów
`reverse_copy()`	Kopiuj odwracając porządek
`rotate()`	rotate(i, j, k): element wskazany przez j będzie pierwszy
`rotate_copy()`	Kopiowanie efektu rotacji
`next_permutation()`	Generuje następną permutację zakresu
`prev_permutation()`	Generuje poprzednią permutację
`random_shuffle()`	Generuje losową permutację
`partition()`	Podział: elementy spełniające kryterium na początek
`stable_partition()`	Jak wyżej, ale z zachowaniem względnych pozycji
`sort()`	Sortuj zakres
`stable_sort()`	Sortuj zachowując względne pozycje elementów równych
`partial_sort()`	Sortuj n elementów zakresu
`partial_sort_copy()`	Kopiuj elementy wg uporządkowania powyżej
`nth_element()`	Podział zakresu wg n-tej pozycji
`make_heap()`	Utwórz kopiec wg zakresu
`push_heap()`	Dodaj element do kopca
`pop_heap()`	Pobierz element z kopca
`sort_heap()`	Sortuj kopiec (zakres przestaje być kopcem)

Memento w/s sortowania


```
#include ...
struct Punkt
{ int x, y;
  Punkt(int xx=0, int yy=0): x(xx), y(yy){}
  friend ostream& operator<<(ostream& os, const Punkt&p)
  { return os<<'('<<p.x<<','<<p.y<<')'; }
};

bool x up(const Punkt& p, const Punkt& q)
{ return p.x<q.x || (p.x==q.x && p.y<q.y); }

bool x down(const Punkt& p, const Punkt& q)
{ return !x up(p, q);
  // return (p.x>=q.x || (p.x==q.x && p.y>=q.y));
  // return (p.x>q.x || (p.x==q.x && p.y>q.y));
}
void show(const vector<Punkt>& P)
{ for(size t i=0; i<P.size(); ++i)
    cout<<P[i]<< (((i+1)%5 != 0)?' ':'\n');
  cout<<((P.size()%5 == 0)?"":"\n");
  cout<<endl;
}
```

Memento w/s sortowania (cd1)


```
int main()
{ int n =10;
int z = 10 ;
  vector<Punkt> vp;
  while(true)
  {
    // Pobierz n, z
    // ...
    vp.clear();
    for(int i=0; i<n; ++i)  // Generuj punkty
      vp.push_back(Punkt(rand()%z+1, rand()%z+1));

    cout<<"Zbiór oryginalny\n";
    show(vp);

    cout<<"Po sortowaniu 'up'\n";
    sort(vp.begin(), vp.end(), x_up);
    show(vp);

    cout<<"Po sortowaniu 'down'\n";
    sort(vp.begin(), vp.end(), x_down);
    show(vp);
  }
}
```

```
Liczba punktów n = 10
Zakres współrzędnych, z = 10
Zbiór oryginalny
(8,2) (1,5) (5,10) (9,9) (5,3)
(6,6) (8,2) (2,2) (3,6) (7,8)

Po sortowaniu up
(1,5) (2,2) (3,6) (5,3) (5,10)
(6,6) (7,8) (8,2) (8,2) (9,9)

Po sortowaniu down
AWARIA w sort() (VS 2012)
```


Memento w/s sortowania (cd2)

Biblioteka standardowa dla wszystkich akcesoriów korzystających z porządkowania elementów wymaga użycia relacji ostrego słabego porządku (strict weak order)

Relacja '<' jest ostrym słabym porządkiem w zbiorze A jeżeli jest

- przeciwzwrotna: dla dowolnego $a \in A$ jest $!(a<a)$
- asymetryczna: dla $a,b \in A$, $a<b \Rightarrow !(b<a)$
- przechodnia: dla $a,b,c \in A$, $a<b \;\&\&\; b<c \Rightarrow a<c$
- przechodnia relacja nieporównywalności:
  Elementy $a,b \in A$ są nieporównywalne jeśli $!(a<b) \;\&\&\; !(b<a)$
  dla $a,b,c \in A$, $!(a<b) \;\&\&\; !(b<c) \Rightarrow !(a<c)$

Definiując operatory relacji wykorzystywane w sortowaniu lub tworzeniu kontenerów asocjacyjnych trzeba sprawdzić powyższe warunki (jeśli nie są spełnione – zachowanie nieokreślone).