# Zaawansowane programowanie w C++ (PCP)

## Wykład 8 - biblioteka standardowa. Kolekcje i iteratory

dr inż. Robert Nowak

27 kwietnia 2007

# Powtórzenie - sprytne wskaźniki

**Zalety:** upraszczają zarządzanie obiektami na stercie

**Wady:** narzuty

Sprytne wskaźniki dostępne w bibliotekach standardowych:

- boost::scoped_ptr
- std::auto_ptr
- boost::shared_ptr
- boost::weak_ptr
- boost::intrusive_ptr

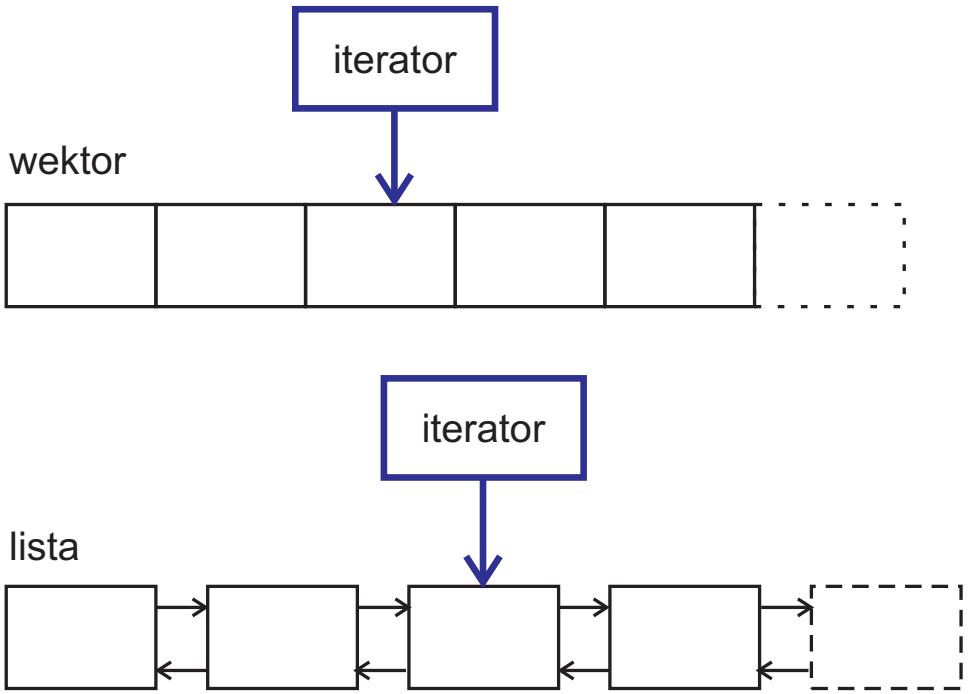
# Lista kolekcji standardowych

| | |
|---|---|
| **vector<T>** | jednowymiarowa tablica |
| **list<T>** | lista dwukierunkowa |
| **deque<T>** | kolejka o dwu końcach |
| **queue<T>** | kolejka |
| **priority_queue<T>** | kolejka priorytetowa |
| **stack<T>** | stos |
| **set<T>** | zbiór |
| **map<T>** | tablica asocjacyjna (słownik) |
| **multiset<T>** | zbiór, wartość może występować wielokrotnie |
| **multimap<T>** | słownik, klucz może występować wielorotnie |

# iteratory

Iterator (forward iterator):

- dereferencja (dostęp do elementu)
- inkrementacja (wskazuje następny element)



# iteratory przykład

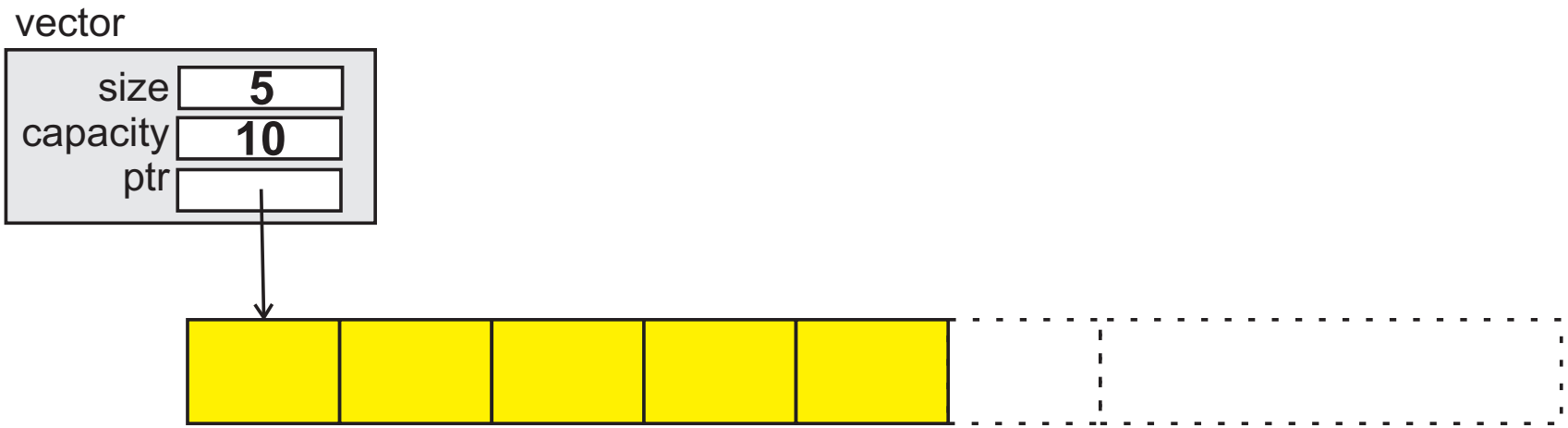
```
vector<int> v;
//operacje na wektorze
vector<int>::const_iterator ii = v.begin();
for(; ii != v.end(); ++ii )
    //*ii dostarcza element wskazywany przez iterator


list<int> l;
//operacje na liście
list<int>::const_iterator ii = l.begin();
for(; ii != l.end(); ++ii )
    //*ii dostarcza element wskazywany przez iterator
```

**Używaj iteratorów do dostępu do elementów kolekcji**

# std::vector

Nagłówek: `<vector>`

```
//Deklaracja
template<class T, class A=allocator<T> >
class std::vector;
```



# vector - definicje typów

| | |
|---|---|
| value_type | typ elementu |
| iterator | iterator do elementów wektora |
| const_iterator | |
| reverse_iterator | iterator odwrotny |
| const_reverse_iterator | |
| pointer, const_pointer | wskaźnik do elementu |
| reference, const_reference | referencja do elementu |

```
typedef vector<int> MyVector;//Definicja typu kolekcji
MyVector v;//Przykładowa kolekcja
//utworzenie obiektu iteratora
MyVector::const_iterator ii = v.begin();//ii jest iteratorem
MyVector::value_type val = v[0]; //val ma typ taki jak element
```

# vector - dostęp do elementów

Dostęp do elementu:

| | |
|---|---|
| reference operator[](size_type n);<br>const_reference operator[](size_type n) const; | bez kontroli |
| reference at(size_type n);<br>const_reference at(size_type n) const; | kontrola zakresu |
| reference front();<br>const reference front() const;<br>reference back();<br>const reference back() const; | pierwszy element<br><br>ostatni element |

Informacje pomocnicze:

| | |
|---|---|
| size_type size() const; | liczba elementów |
| size_type capacity() const; | liczba elementów dla których zaalokowano pamięć |
| bool empty() const; | bada czy size() == 0 |

# vector - konstruktory

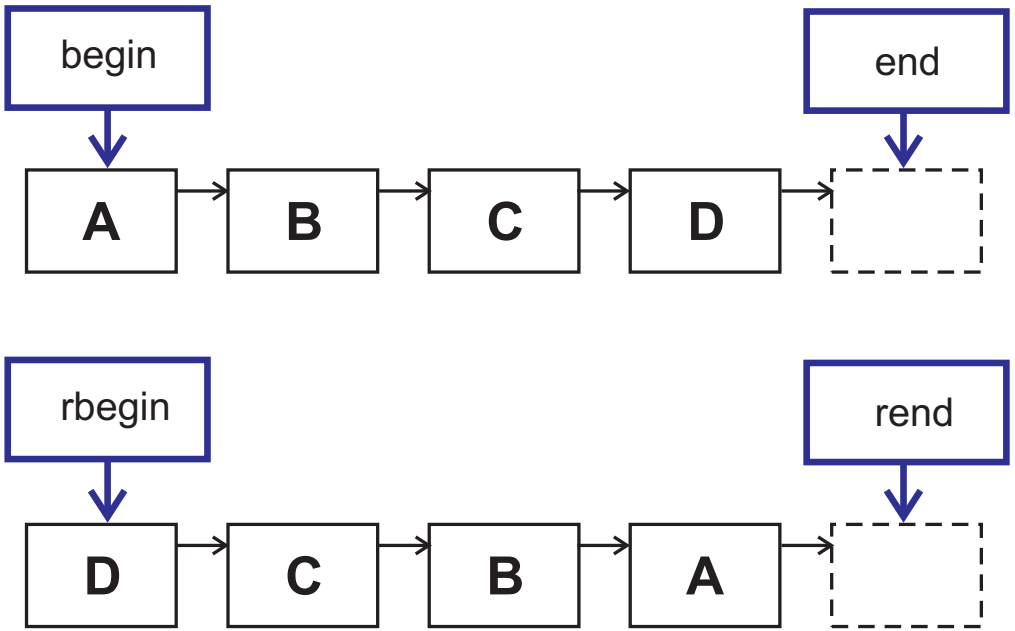
| | |
|---|---|
| vector(); | pusty kontener |
| vector(size_type n, const T& val = T()); | n kopii val |
| template<class In> vector(In first, In last); | kopiuje zakres |
| vector(const vector& v); | konstruktor kopiujący |
| vector& operator=(const vector& v); | |

Przykłady:

```
const int TAB[] = { 3, 2, 0, 8 };
const int TAB_SIZE = sizeof(TAB)/sizeof(TAB[0]);
//kopiowanie zakresu
vector<int> v(TAB, TAB + TAB_SIZE);
//Wypełnia stałą
vector<int> v2(TAB_SIZE, 0);
vector<int> v3 = v1;//Konstruktor kopiujący
```

# vector - przeglądanie kolejnych elementów

| | |
|---|---|
| iterator begin(); | iterator do pierwszego elementu |
| iterator end(); | iter. do pierwszego za ostatnim |
| reverse_iterator rbegin();<br>reverse_iterator rend(); | iteratory dla odwrotnej kolejności |



# vector - dodawanie i usuwanie elementów

| | |
|---|---|
| void push_back(const T&);<br>void pop_back(); | wstawia na koniec<br>usuwa ostatni |
| iterator insert(iterator pos, const T& t);<br>iterator erase(iterator pos); | wstawia przed<br>usuwa |
| template<class In><br>void insert(iterator pos, In first, In last);<br>iterator erase(iterator first, iterator last); | wstawia zakres<br><br><br>usuwa |
| void clear(); | usuwa wszystkie |

Inne funkcje:

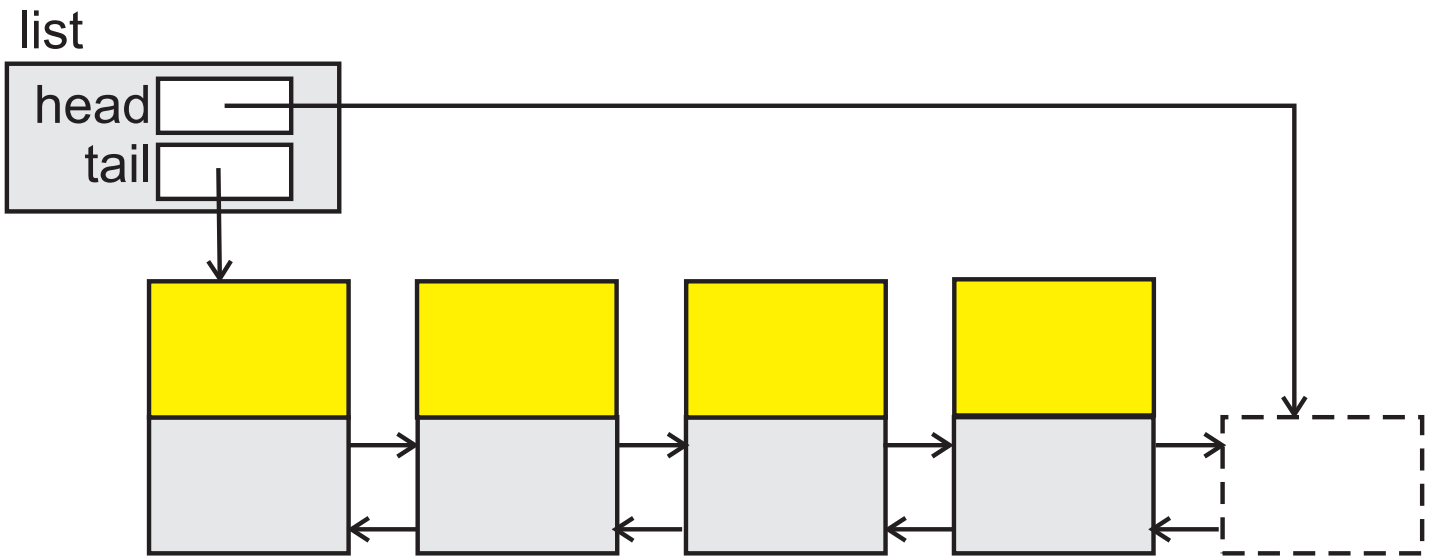
- void vector::swap(vector& v);
- bool operator==(const vector&, const vector&);
- bool operator<(const vector&, const vector&);

Specjalizacje: vector<bool>

# std::list

Nagłówek: `<list>`

```
//Deklaracja
template<class T, class A=allocator<T> >
class std::vector;
```



# operacje dla listy dwukierunkowej

dostarcza operacji takich jak vector za wyjątkiem:

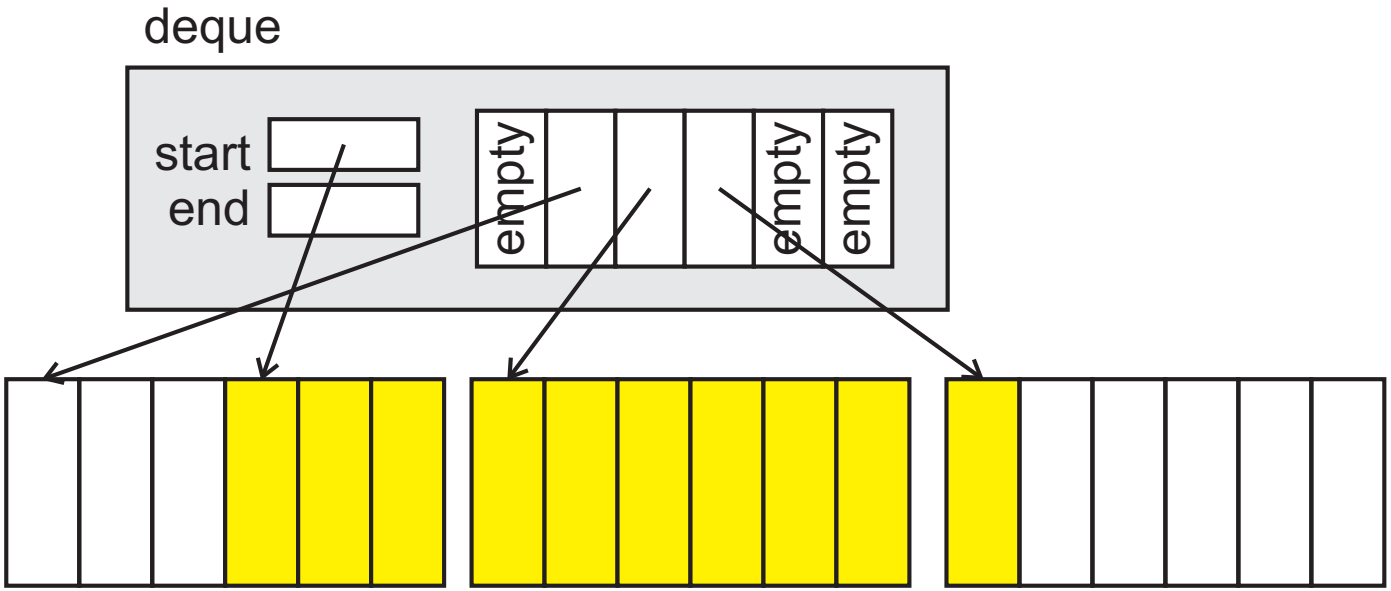
- dostępu swobodnego do elementów
  - operatora indeksowania (operator[])
  - metody at()
- rezerwacji pamięci ( capacity(), reserve() )

Dodatkowe operacje:

- operacje na początku listy:
  - reference front();
  - const_reference front() const;
  - void push_front(const T&);
  - void pop_front();
- inne operacje:
  - void remove(const T& val); //usuwa el. równe val
  - void reverse(); //odwraca porządek elementów
  - splice, merge, sort

# std::deque - kolejka o dwu końcach

Nagłówek <deque>




Operacje:

- takie jak dla wektora (za wyjątkiem capacity(), reserve() )
- operacje z przodu ciągu (tak jak dla listy)

# Porównanie kolekcji sekwencyjnych

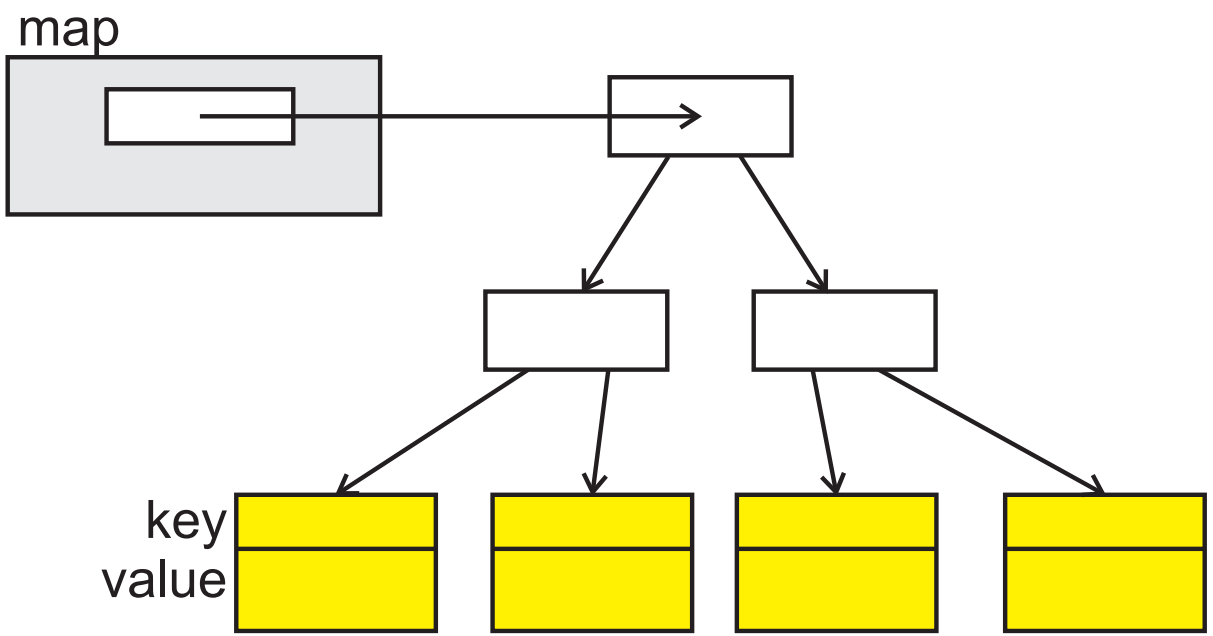
| | indeksowanie | insert, erase | początek | koniec |
|---|---|---|---|---|
| tablica | O(1) | - | - | - |
| vector | O(1) | O(n)+ | - | O(1)+ |
| list | - | O(1) | O(1) | O(1) |
| deque | O(1) | O(n) | O(1) | O(1) |

- tablica - struktura odziedziczona po 'C'
- wektor - optymalizacja przy dostępie przez indeks, rezerwowanie pamięci.
- lista - optymalne wstawianie i usuwanie elementów.
- deque - optymalne operacje na końcach, indeksowanie prawie tak optymalne jak w wektorze.

# słownik

## Nagłówek: <map>

```
//Deklaracja
template<class K, class T, class Cmp = less<Key>,
         class A = allocator<Pair<const K, T> > >
class std::map;
```



# map - definicje typów

```
template<class K, class T, class Cmp = less<Key>,
         class A = allocator<Pair<const K, T> > >
class std::map {
public:
    typedef K key_type;
    typedef T data_type;
    typedef std::pair<const K, T> value_type;
    /* ...  */reference;
    /* ...  */const_reference;
    /* ...  */iterator;
    /* ...  */const_iterator;
    /* ...  */reverse_iterator;
    /* ...  */const_reverse_iterator;
    /* itd */
```

# iteracja po słowniku

| |
|---|
| iterator begin() |
| iterator end() |
| reverse_iterator rbegin() |
| reverse_iterator rend() |

**Uwaga**

Iterator dostarcza `value_type`, czyli `std::pair<const K, T>`.

std::pair

| | |
|---|---|
| first | pierwszy element (tutaj klucz) |
| second | drugi element (tutaj wartość) |

# map - konstruktory

| | |
|---|---|
| map();<br>map(const key_compare& cmp); | pusty słownik<br>j.w. + funkcja por. |
| template<class In><br>map(In first, In last); | kopiuje zakres |
| map(const map& m);<br>map& operator=(const map& m); | konstruktor kopiujący |

```
//Przykłady
```

# map - modyfikacja elementów

| | |
|---|---|
| pair<iterator,bool><br>insert(const value_type& x); | wstawia |
| iterator insert(it p, value_type& x); | j.w. p to sugestia |
| void erase(iterator pos); | usuwa |
| size_type erase(key_type& k); | usuwa dla klucza |
| void clear(); | wszystkie elementy |

```
//Przykłady
map<string,int> ks;
typedef pair<string,int> P;//mozna uzyc map::value_type
ks.insert( P(¨Abacki¨, 123456) );
ks.insert( P(¨Babacki¨, 234567) );
ks.insert( P(¨Cabacki¨, 346778) );
```

# map - operacje słownikowe

- iterator find(const key_type& k);
- const_iterator find(const key_type& k);
- size_type count(const key_type& k) const;
- pair<iterator,iterator> equal_range(const key_type& k);
- pair<const_iterator, const_iterator>
  equal_range(const key_type& k) const;

```
//Przykłady
map<string,int> ks;
/* ...  */ //dodaje elementy do słownika
map<string,int>::const_iterator ii = ks.find(¨Abacki¨);
```

# inne kontenery asocjacyjne

- set (zbiór)
- multimap
- multiset

# Porównanie kolekcji

| | indeksowanie | insert, erase | początek | koniec |
|---|---|---|---|---|
| tablica | O(1) | - | - | - |
| vector | O(1) | O(n)+ | - | O(1)+ |
| list | - | O(1) | O(1) | O(1) |
| deque | O(1) | O(n) | O(1) | O(1) |
| map | O(log(n)) | O(log(n)) | - | - |
| set | - | O(log(n)) | - | - |