Podstawy programowania obiektowego

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

http://home.agh.edu.pl/~pszwed/

Aktualizacja: 14.05.2021

9. Kontenery

Kontenery

Kontenery są obiektami umożliwiającymi:

- grupowanie (agregację) innych obiektów
- dostęp sekwencyjny lub swobodny do zawartych w nich elementów.

Typowe kontenery to:

- wektor (*ang. vector*)
- lista (*ang. list*)
- kolejka (*ang. queue*)
- stos (*ang. stack*)
- zbiór i wielozbiór (*ang. set, multiset*)
- słownik (*ang. map, multimap*)

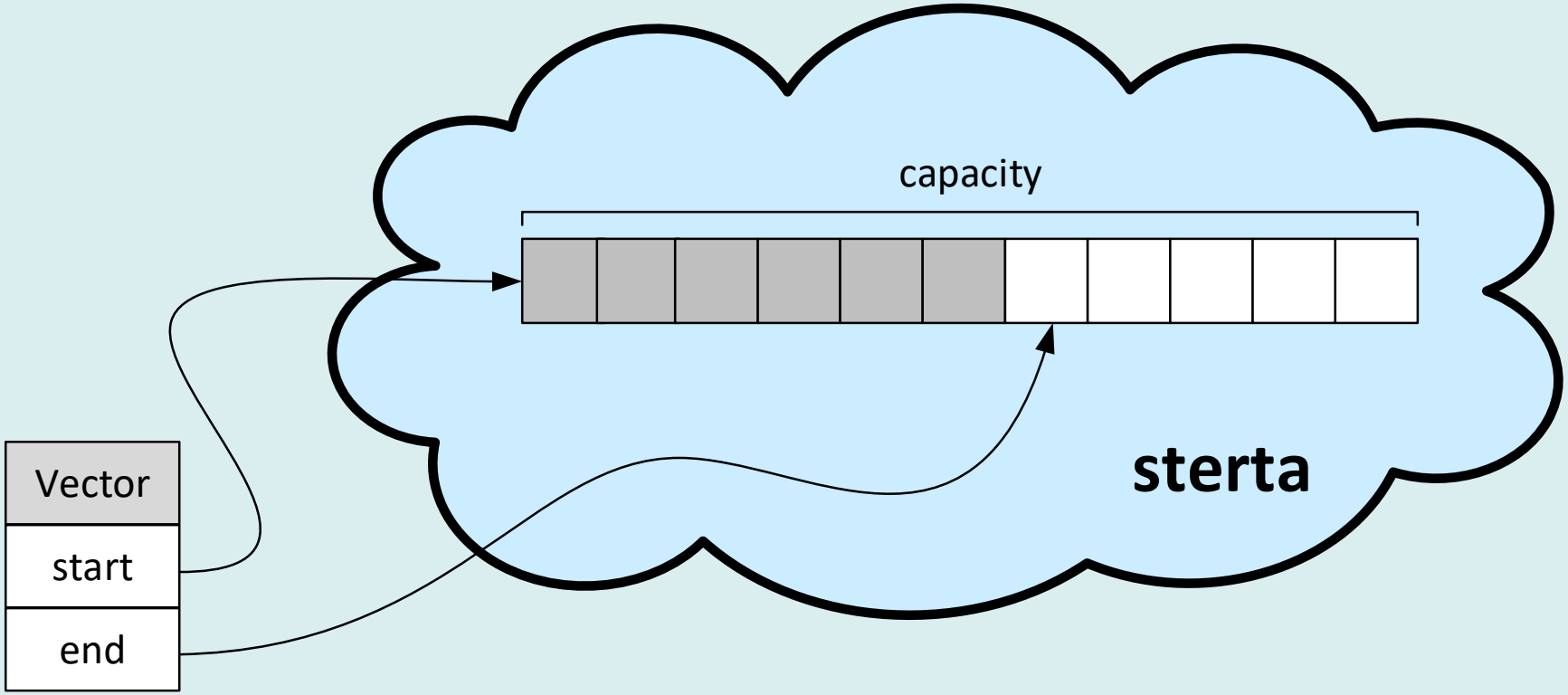
Kontenery

Wybór typu kontenera jest uzależniony od założonego sposobu dostępu do elementów i strategii umieszczania elementów w kontenerze:

- **wektor** jest najbardziej efektywny, jeżeli w momencie jego kreacji znana jest liczba elementów i alokujemy dla niego pamięć w jednym wywołaniu new/malloc
- **lista** jest efektywna, jeżeli często dodajemy i usuwamy pojedyncze elementy
- efektywna implementacja **zbioru** to zazwyczaj drzewo, konieczne są funkcje do porównywania elementów
- **słownik** zakłada, że do realizacji dostępu do elementów posługujemy się odwzorowaniem *klucz→wartość*

Wektor

Tablica przyrastająca podczas dodawania elementów



- `start` – wskaźnik na pierwszy element tablicy
- end – wskaźnik na pierwszy wolny element tablicy: elementy zajęte znajdują się pomiędzy `start` (włącznie) i end (wyłącznie)

Wektor

```
class Vector{
protected:
    double*start;
    double*end;
    int capacity;
    bool reserve(int newCapacity);
public:
    Vector(int size = 0);
    ~Vector();
    int getSize()const;
    bool pushFront(double v);
    bool pushBack(double v);
    bool insertAt(int where,double v);
    bool deleteFront();
    bool deleteBack();
    double&elementAt(int i)const;
    void dump()const;
};
```

Przedłuża tablicę

Zwraca liczbę elementów

- Dodawanie: pushFront, pushBack, insertAt
- Usuwanie deleteFront, deleteBack

Dostęp do i-tego elementu

Wektor – konstruktor, destruktor

```
Vector::Vector(int size):start(0),end(0),capacity(0){
    if(size>0){
        start = new double[size];
        capacity=size;
        end=start;
    }
}
Vector::~Vector(){
    if (start!=0)delete []start;
    start=end=0;
    capacity=0;
}

int Vector::getSize()const{
    return end-start;
}
```

Alokuje pamięć o początkowym rozmiarze

Zwalnia pamięć

Arytmetyka wskaźników

Wektor - rozszerzanie

Rozszerzanie tablicy do rozmiaru `newCapacity`

```
bool Vector::reserve(int newCapacity)
{
    if(newCapacity < capacity)return false;
    double*tmp=new double[newCapacity];
    if(capacity){
// Kopiowanie elementów w tablicy
//        memcpy(tmp,start,capacity*sizeof(double));
        for(int i=0;i<capacity;i++)tmp[i] = start[i];
        delete []start;
    }
    end=tmp+(end-start);
    capacity=newCapacity;
    start=tmp;
    return true;
}
```

Wektor - dodawanie

```
bool Vector::pushBack(double v)
{
    if(capacity==getSize() && !reserve(capacity+64))return false;

    start[getSize()]=v;    // czyli *end=v;
    end++;
    return true;
}

bool Vector::pushFront(double v)
{
    if(capacity==getSize() && !reserve(capacity+64))return false;

    for(int i=getSize()-1;i>=0;i--){
        start[i+1]=start[i];
    }
    start[0]=v;
    end++;
    return true;
}
```

Wektor - wstawianie

```
bool Vector::insertAt(int where,double v)
{
    if(capacity==getSize() &&
       !reserve(capacity+64)){
        return false;
    }

    if(where>getSize())where = getSize();

    for(int i=getSize()-1;i>=where;i--){
        start[i+1]=start[i];
    }
    start[where]=v;
    end++;
    return true;
}
```

Jeżeli indeks where wychodzi poza rozmiary wektora +1, ustawiany jest na końcu.

Można wyobrazić sobie inną strategię (automatyczne przedłużanie).

Wektor - usuwanie

```
bool Vector::deleteBack()
{
    if(getSize()==0)return false;
    end--;
    return true;
}

bool Vector::deleteFront()
{
    if(getSize()==0)return false;
    for(int i=1;i<getSize();i++){
        start[i-1]=start[i];
    }
    end--;
    return true;
}
```

Wektor – dostęp do elementów

```
class BadIndex{
public:
    int index;
    BadIndex(int i):index(i){}
};

double&Vector::elementAt(int i)const{
    if(i<0 || i>= getSize())throw BadIndex(i);
    return start[i];
}

void Vector::dump()const
{
    cout<<"[ ";
    for(int i=0;i<getSize();i++)
        cout<<start[i]<<" ";
    cout<<"]"<<endl;
}
```

- Zwraca referencję (możliwość modyfikacji)
- Generuje wyjątek, jeżeli brak takiego elementu.

Test wektora

```
int main(){
    Vector vect;
    for(int i=0;i<5;i++){
        vect.pushFront(i);
        vect.pushBack(i+5);
    }
    vect.insertAt(5,100);
    vect.insertAt(0,100);
    vect.insertAt(1000,100);
    vect.dump();
    for( int i=0;i<6;i++){
        vect.deleteBack();
        vect.deleteFront();
        vect.dump();
    }
    vect.elementAt(0)=-1;
    vect.dump();
}
```

Wynik

[100 4 3 2 1 0 100 5 6 7 8 9 100]

[4 3 2 1 0 100 5 6 7 8 9]

[3 2 1 0 100 5 6 7 8]

[2 1 0 100 5 6 7]

[1 0 100 5 6]

[0 100 5]

[100]

[-1]

Przedłużanie wektora

Wielokrotne przełużanie wektora o kilkadziesiąt elementów (tutaj 64) może być nieefektywne.

Zmierzmy czas dodanie 1M elementów.

```
#include <time.h>

int main(){
    Vector v;
    clock_t s = clock();
    for(int i=0;i<1024*1204;i++)v.pushBack(i);
    clock_t e = clock();
    printf("Execution time: %fs",(double)(e-s)/CLOCKS_PER_SEC);

}
```

```
Execution time: 89.374000s
```

Analiza: nasze żądania przydziału pamięci to
64 + 128 + 192 +…1024*1024

Czyli $n = \frac{1024 \cdot 1024}{64} = 16\,384$,

Sumaryczna przydzielona pamięć

$S_n = n * \frac{64+1024*1024}{2} = 8\,590\,458\,880$

Zmiana sposobu przydziału

```
bool Vector::pushBack(double v)
{
    if(capacity==getSize() &&
       !reserve(capacity+64)){
        return false;
    }
    *end=v;
    end++;
    return true;
}
```

Execution time: 89.374000s

```
bool Vector::pushBack(double v)
{
    if(capacity==getSize() &&
       !reserve(capacity?2*capacity:64)){
        return false;
    }
    *end=v;
    end++;
    return true;
}
```

Execution time: 0.031000s

Zamiast dodawać stałą wartość, mnożymy `capacity` * 2

Żądania przydziału pamięci to $64,128,256,512, \ldots 1024 \cdot 1024$

$S_n = 64(2^0 + 2^1 + 2^2 + \cdots + 2^n)$

Obliczamy n:

$64 * 2^n = 1024 * 1024$, stąd

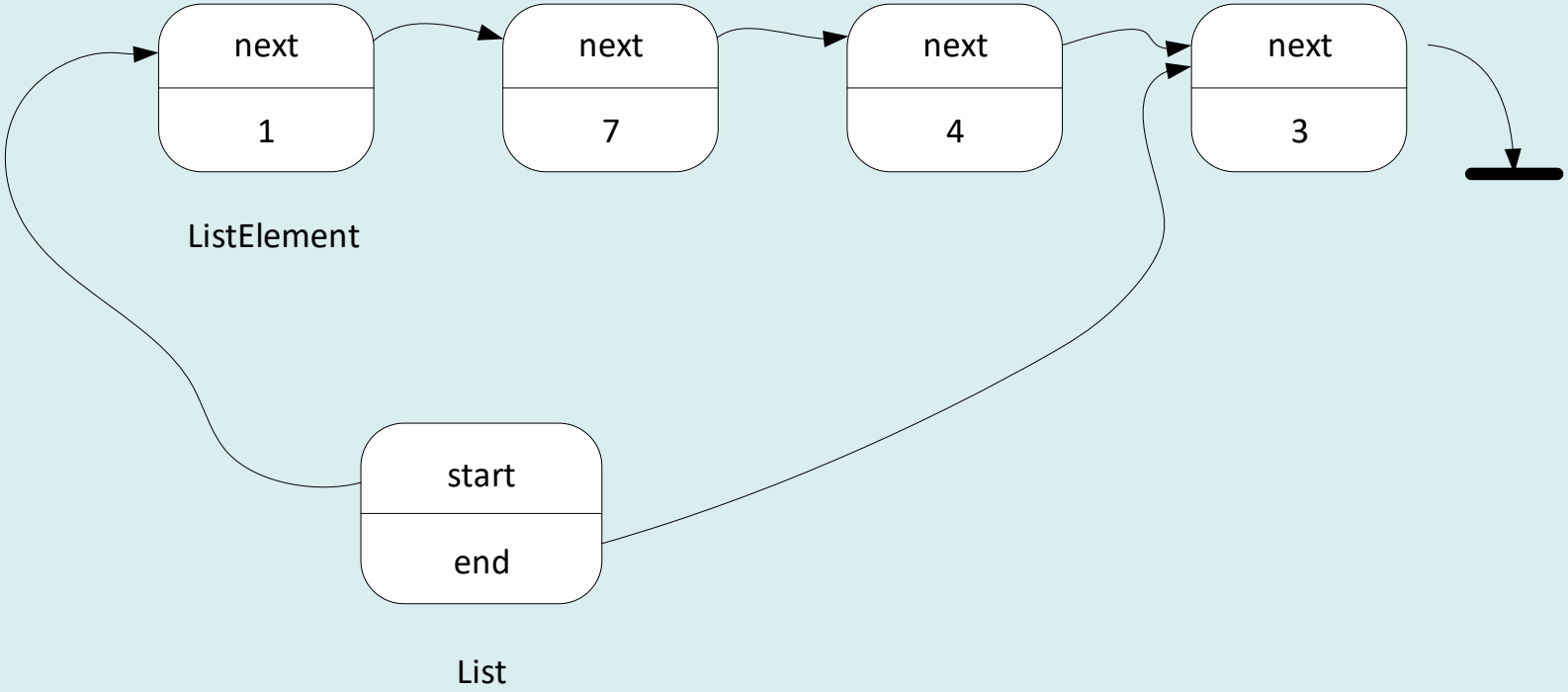
$n = 14$

Sumaryczna przydzielona pamięć to $\frac{64(1-2^{15})}{1-2} = 2097088$

Czas wykonania 2888 razy krótszy

Lista

Lista jednokierunkowa przechowująca wartości całkowite



- `ListElement` – zawiera dane
- `List` – klasa zapewnia interfejs do manipulacji listą (dodawania, usuwania elementów)

Lista - deklaracje

```
class ListElement
{
public:
    ListElement*next;
    int value;
};
```

Lista jednokierunkowa

```
class List
{
public:
    ListElement*start;
    ListElement*end;

    List();
    ~List();
    void pushFront(int v);
    void pushBack(int v);
    void insertAt(int where,int v);
    void deleteFront();
    void deleteBack();
    void dump()const;
};

List::List():start(0),end(0){}
```

Wskaźniki na pierwszy i ostatni element

Konstruktor

Lista – dodawanie elementów

```
void List::pushFront(int v)
{
    ListElement *le = new ListElement();
    le->value=v;
    le->next=start;
    start=le;
    if(end==0)end=start;
}
void List::pushBack(int v)
{
    ListElement *le = new ListElement();
    le->value=v;
    le->next=0;
    if(end)end->next=le;
    end=le;
    if(start==0)start=end;
}
```

Z przodu (przed pierwszym elementem)

Na końcu

Lista – wstawianie elementów

Wyznaczamy element (wskaźnik `ip`) po którym wstawimy nowy element `le`.

```
void List::insertAt(int where,int v)
{
    ListElement *le = new ListElement();
    le->value=v;

    ListElement*ip=0;
    int count=0;
    for(ListElement*i=start;   i!=0 && count<where;
        i=i->next,count++){
        ip=i;
    }
```

Lista – wstawianie elementów

- Jeżeli `ip!=0` wstawiamy element `le` po `ip`
- Jeżeli `ip==0` dodajemy `le` na początku

```
    // ...
    if(ip){
        le->next=ip->next;
        ip->next=le;
    }else{
        le->next=start;
        start=le;
    }
    if(end==0)end=start;
    if(le->next==0)end=le;
}
```

Lista – usuwanie elementów

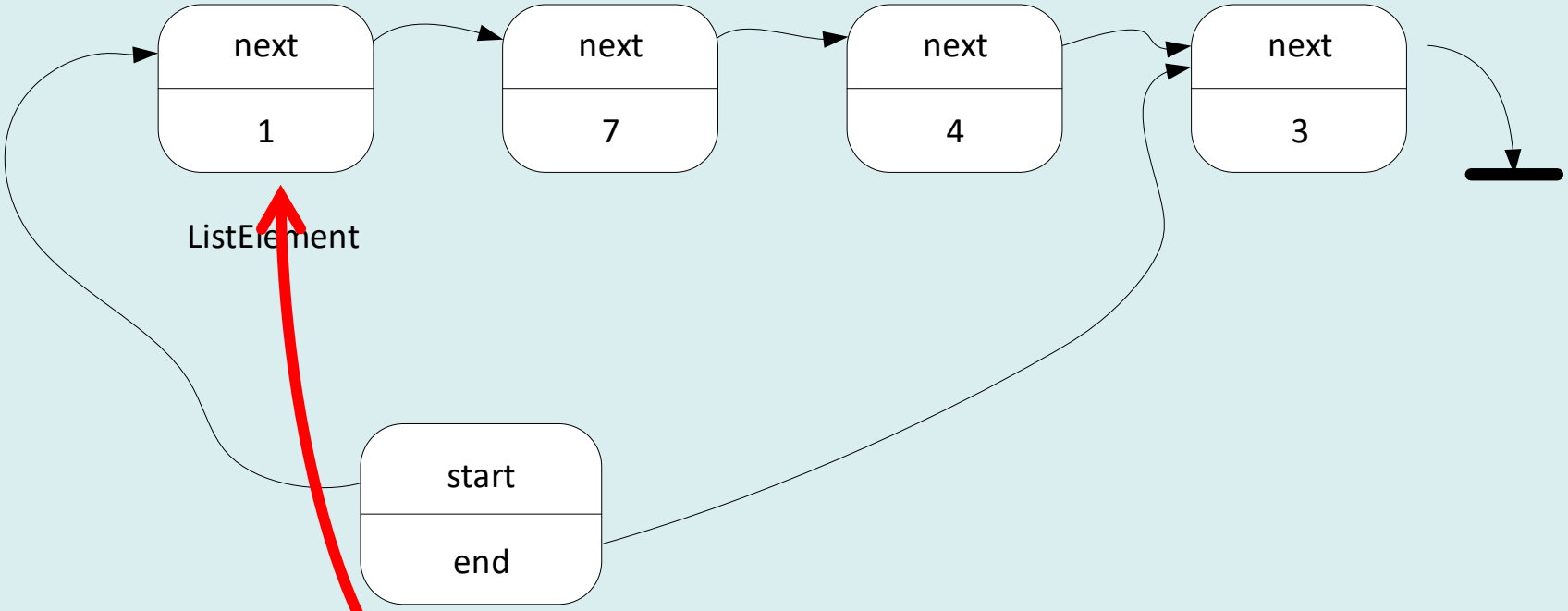
```
void List::deleteFront()
{
    if(!start)return;
    ListElement*todel=start;
    start=start->next;
    delete todel;
    if(start==0)end=0;
}

List::~List()
{
    while(start!=0)deleteFront();
}
```

Usuwa pierwszy element

Destruktor listy musi usunąć wszystkie elementy

Lista – iteracja




```
void List::dump()const
{
    cout<<"[ ";
    for(ListElement*i=start;i!=0;i=i->next)
        cout<<i->value<<" ";
    cout<<"]"<<endl;
}
```

Lista – iteracja

- Szukanie przedostatniego elementu: `newEnd`
- Usuwanie ostatniego elementu i uaktualnianie wskaźników

```
void List::deleteBack()
{
    ListElement*newEnd=0;
    for(ListElement*i=start;i!=0;i=i->next){
        if(i->next!=0 && i->next==end)newEnd=i;
    }
    if(end!=0)delete end;
    end = newEnd;
    if(end!=0)end->next=0;
    if(end==0)start=end;
}
```

Test listy

```
int main(){
    List l;
    for(int i=0;i<5;i++){l.pushFront(i);l.pushBack(i+5);}
    l.insertAt(5,100);
    l.insertAt(0,100);
    l.insertAt(1000,100);
    l.dump();
    for( int i=0;i<6;i++){
        l.deleteBack();
        l.deleteFront();
        l.dump();
    }
}
```

Wynik

[100 4 3 2 1 0 100 5 6 7 8 9 100]

[4 3 2 1 0 100 5 6 7 8 9]

[3 2 1 0 100 5 6 7 8]

[2 1 0 100 5 6 7]

[1 0 100 5 6]

[0 100 5]

[100]

Iteracja i iteratory

Iteracja

```
int*tab = new int[SIZE];

for(int i=0;i<SIZE;i++){
    cout<<tab[i]<<" ";
}
```

Iteracja po tablicy

```
for(ListElement*i=start;i!=0;i=i->next)
    cout<<i->value<<" ";
```

Iteracja po liście

Iteracja

```
int main(){
    Vector v;
    for(int i=0;i<10;i++)v.pushBack(i);
    for(int i=0;i<v.getSize();i++){
        cout<<v.elementAt(i)<<" ";
    }

}
```

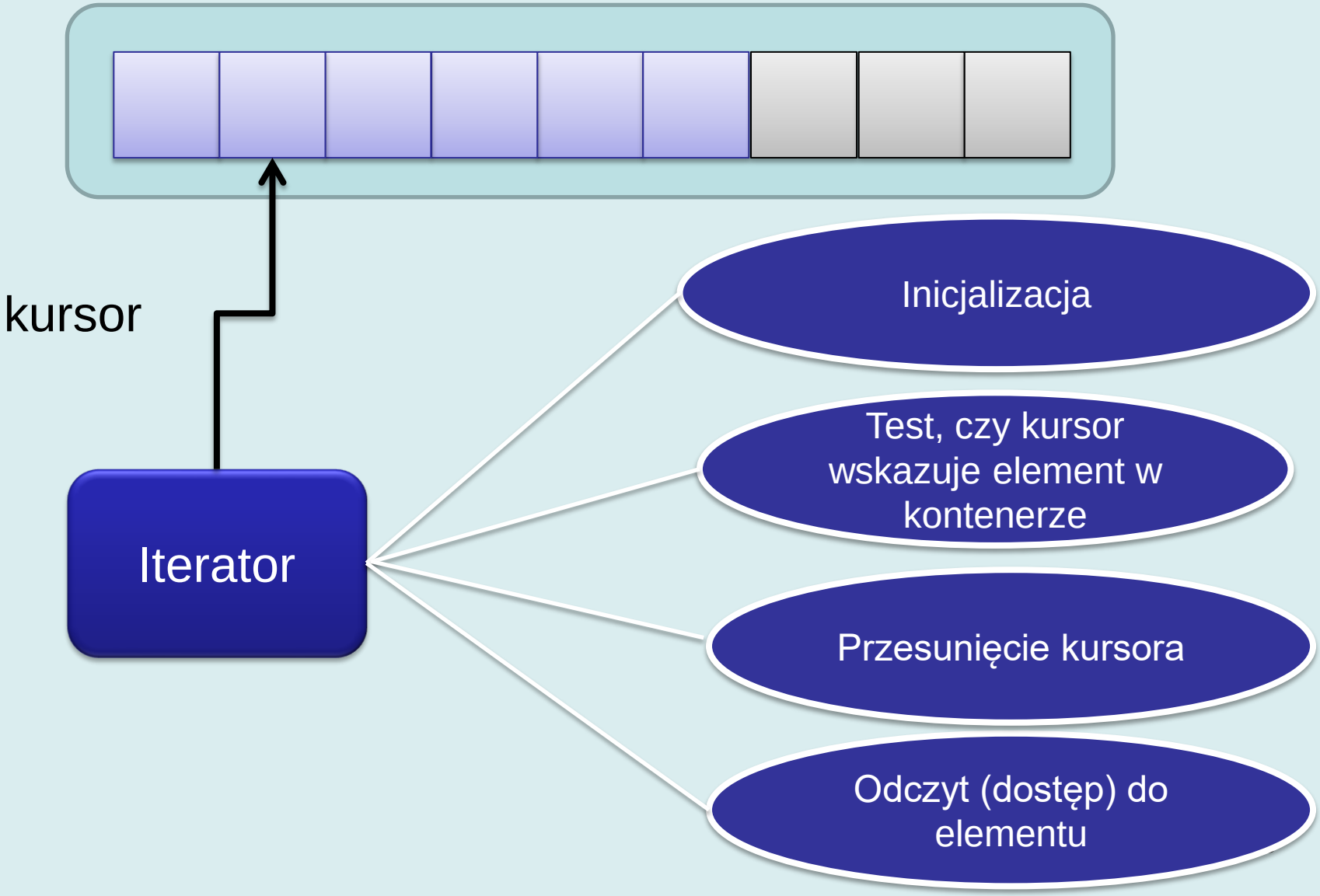
Funkcje do umożliwiające iterację mogą być zaimplementowane, jako metody kontenera. Często jednak taka implementacja jest niewydajna.

Dla wektora dostęp do i-tego elementu jest wydajny, dla innych kontenerów niekoniecznie.

Iteratory są specjalnymi klasami przeznaczonymi do realizacji dostępu do elementów kontenera.

- Pozwalają ukryć szczegóły implementacji kontenera
- Zapewniają jednolity interfejs
- Równocześnie może istnieć wiele iteratorów różniących się stanem

Iteratory


kursor
Iterator
Inicjalizacja
Test, czy kursor wskazuje element w kontenerze
Przesunięcie kursora
Odczyt (dostęp) do elementu

Iterator wektora

```
class VectorIterator{
    const Vector&vector;
    double*cursor;
public:
    VectorIterator(const Vector&v)
       :vector(v),cursor(v.start){}
    VectorIterator&operator++(){
        cursor++;
        return *this;
    }
    VectorIterator operator++(int){
        VectorIterator tmp=*this;
        ++*this;
        return tmp;
    }
    bool good()const{return cursor<vector.end;}
    double&get()const{return *cursor;}
};
```

Przykład iteracji po wektorze

```
int main(){
    Vector vect;
    for(int i=1;i<16;i++)vect.pushBack(i*i);
    for(VectorIterator it(vect);it.good();++it){
        cout<<it.get()<<" ";
    }
    cout<<endl;

    // Zagnieżdżona iteracja O(n^3)
    for(VectorIterator ita(vect);ita.good();++ita){
        for(VectorIterator itb(vect);itb.good();++itb){
            for(VectorIterator itc(vect);itc.good();++itc) {
                if (ita.get() + itb.get() == itc.get()) {
                    cout << ita.get() << "+"<<itb.get()
                         << "=" << itc.get() << endl;
                }
            }
        }
    }
}
```

1 4 9 16 25 36 49 64 81 100 121 144 169 196 225

9+16=25 16+9=25 25+144=169 36+64=100
64+36=100 81+144=225 144+25=169 144+81=225

Iterator listy

```
class ListIterator{
    ListElement*cursor;
public:
    ListIterator(const List&l):cursor(l.start){}
    ListIterator&operator++(){
        if(cursor)cursor=cursor->next;
        return *this;
    }
    ListIterator operator++(int){
        ListIterator tmp=*this;
        ++*this;
        return tmp;
    }
    bool good()const{return cursor!=0;}
    int&get()const{return cursor->value;}
};
```

Przykład iteracji po liście

```
int main(){
    List l1;
    for(int i=0;i<10;i++)l1.pushBack(i);
    List l2;
    for(int i=0;i<10;i++)l2.pushBack(i+5);
    //O(n^2)
    for(ListIterator ita(l1);ita.good();++ita){
        cout<<ita.get()<<" > {";
        for(ListIterator itb(l2);itb.good();++itb){
            if (ita.get() > itb.get() ) {
                cout << itb.get()<<" ";
            }
        }
        cout<<"}"<<endl;
    }
}
```

```
0 > {}
1 > {}
2 > {}
3 > {}
4 > {}
5 > {}
6 > {5 }
7 > {5 6 }
8 > {5 6 7 }
9 > {5 6 7 8 }
```

Dostęp swobodny i sekwencyjny

- Iteratory zazwyczaj są stosowane do sekwencyjnej iteracji po zawartości kontenera (czyli przebiegają kolejne elementy).
- Mogą jednak zostać zaimplementowane w taki sposób, aby ustawić kursor na dowolnym elemencie.

VectorIterator	Przesuwanie w przód	Może zostać wydajnie zaimplementowane
	Przesuwanie w tył	Może zostać wydajnie zaimplementowane
	Ruch w obie strony	Możliwy
	Cechy (potencjalne)	Dostęp swobodny, dwukierunkowy
ListIterator	Przesuwanie w przód	Możliwe, ale mniej wydajne
	Przesuwanie w tył	Algorytmicznie możliwe, ale bardzo niewydajne (lista jednokierunkowa)
	Ruch w obie strony	Nie
	Cechy	Dostęp sekwencyjny, jednokierunkowy

Rozszerzenia VectorIterator

```
class VectorIterator{
…
    VectorIterator&operator+=(int d);
    VectorIterator operator+(int d)const;
    VectorIterator&operator-=(int d){
        return *this+=-d;
    };
    VectorIterator operator-(int d)const{
        return *this+(-d);
    }
};
```

```
VectorIterator&VectorIterator::operator+=(int d) {
    cursor+=d;
    if(cursor<vector.start)
        cursor=vector.start;
    return *this;
}
```

Kursor jest wskaźnikiem do elementów tablicy. Implementacja ruchu kursora jest wydajna (arytmetyka wskaźników). Należy jedynie zadbać, aby nie przesunąć go przed adres start.

Rozszerzenia VectorIterator

```
VectorIterator VectorIterator::operator+(int d)const{
    VectorIterator r = *this;
    r.cursor +=d;
    if(r.cursor<r.vector.start)
        r.cursor=r.vector.start;
    return r;
}

int main(){
    Vector v;
    for(int i=0;i<10;i++)v.pushBack(i);
    for(VectorIterator it(v);it.good();it+=3){
        cout<<it.get()<<" ";
        VectorIterator it2=it-1;
        cout<<"(it2:"<< it2.get()<<") ";
    }

}
```

```
0 (it2:0) 3 (it2:2) 6 (it2:5) 9 (it2:8)
```

Rozszerzenia ListIterator

```
ListIterator&ListIterator::operator+=(int d) {
    if(d<0) return *this; // albo throw ForbiddenMovement();
    for(int i=0;i<d && cursor;i++){
        cursor=cursor->next;
    }
    return *this;
}

ListIterator ListIterator::operator+(int d)const{
    ListIterator r = *this;
    r +=d;
    return r;
}

int main(){
    List lst;
    for(int i=0;i<20;i++)lst.pushBack(i);
    for(ListIterator it(lst);it.good();it+=5){
        cout<<it.get()<<" ";
    }
}
```

Wynik
0 5 10 15

Rozszerzenia ListIterator

Możliwa jest implementacja dostępu swobodnego dla listy, ale często nie jest ona realizowana, aby zapobiec przypadkowej niewydajnej implementacji .

```
class NoSuchElementException{};

int get(const List&lst,int i){
    ListIterator it(lst);
    it+=i;
    if(it.good())return it.get();
    else throw NoSuchElementException();
}
```

Funkcja przesuwa iterator o i elementów do przodu, czyli i razy wykonuje `cursor=cursor->next.`

Aby przeiterować n elementową listę, kursor przesuwany jest $\frac{n(n+1)}{2}$ razy.

Porównanie

```
int main(){
    int n = 10000;
    List lst;
    for(int i=0;i<n;i++){
        lst.pushBack(i);
    }
    clock_t t0 = clock();
    for(int i=0;i<lst.getSize();i++){
        int k = get(lst,i);
    }
    clock_t t1 = clock();
    for(ListIterator it(lst);it.good();++it){
        int k=it.get();
    }
    clock_t t2 = clock();
    cout<<"n="<<n
    <<" random access:"<<(double)(t1-t0)/CLOCKS_PER_SEC
    <<" sequential:"<<(double)(t2-t1)/CLOCKS_PER_SEC;

}
```

```
n=10000 random access:0.125 sequential:0
n=50000 random access:3.921 sequential:0
n=100000 random access:14.719 sequential:0
n=200000 random access:69.047 sequential:0
```