Programowanie obiektowe

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

e-mail: pszwed@agh.edu.pl

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

Aktualizacja: 25.10.2019

6. 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*)

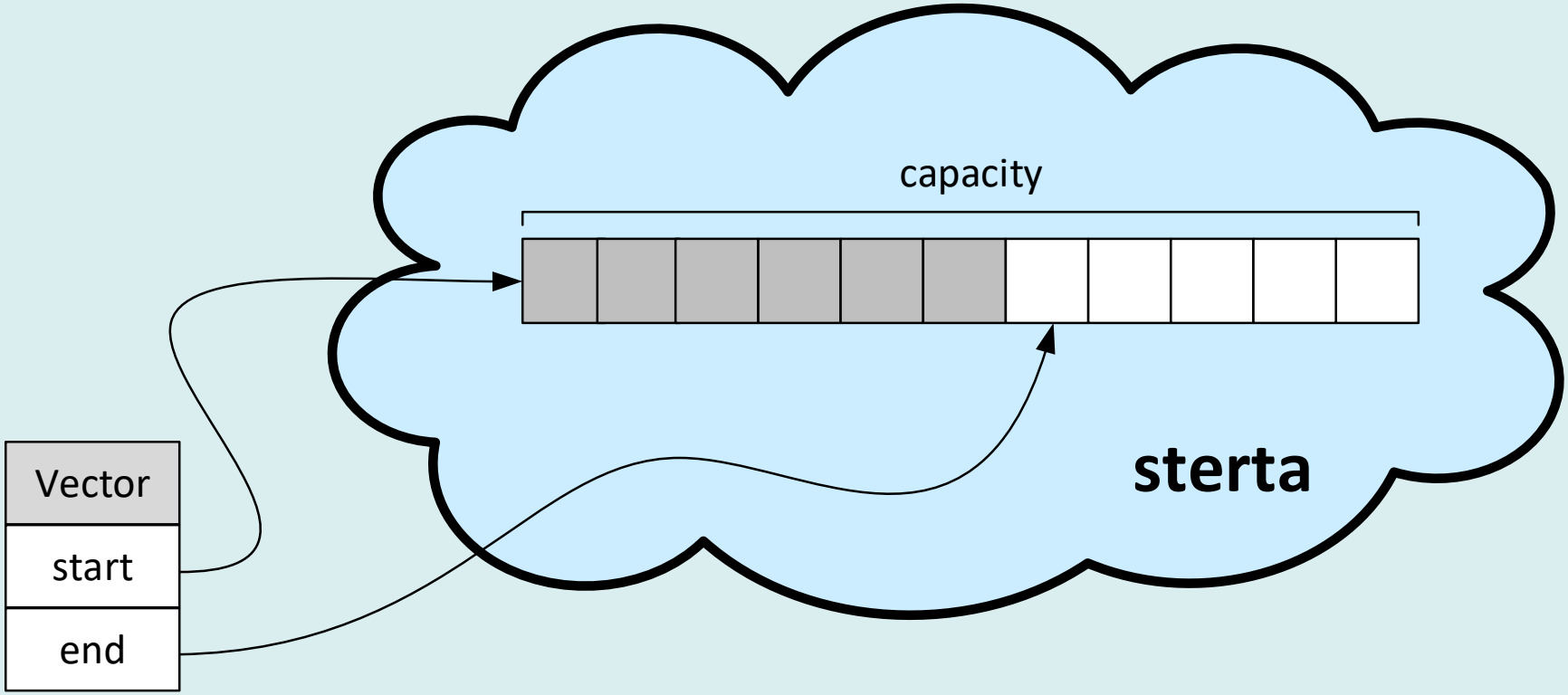
Kontenery

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

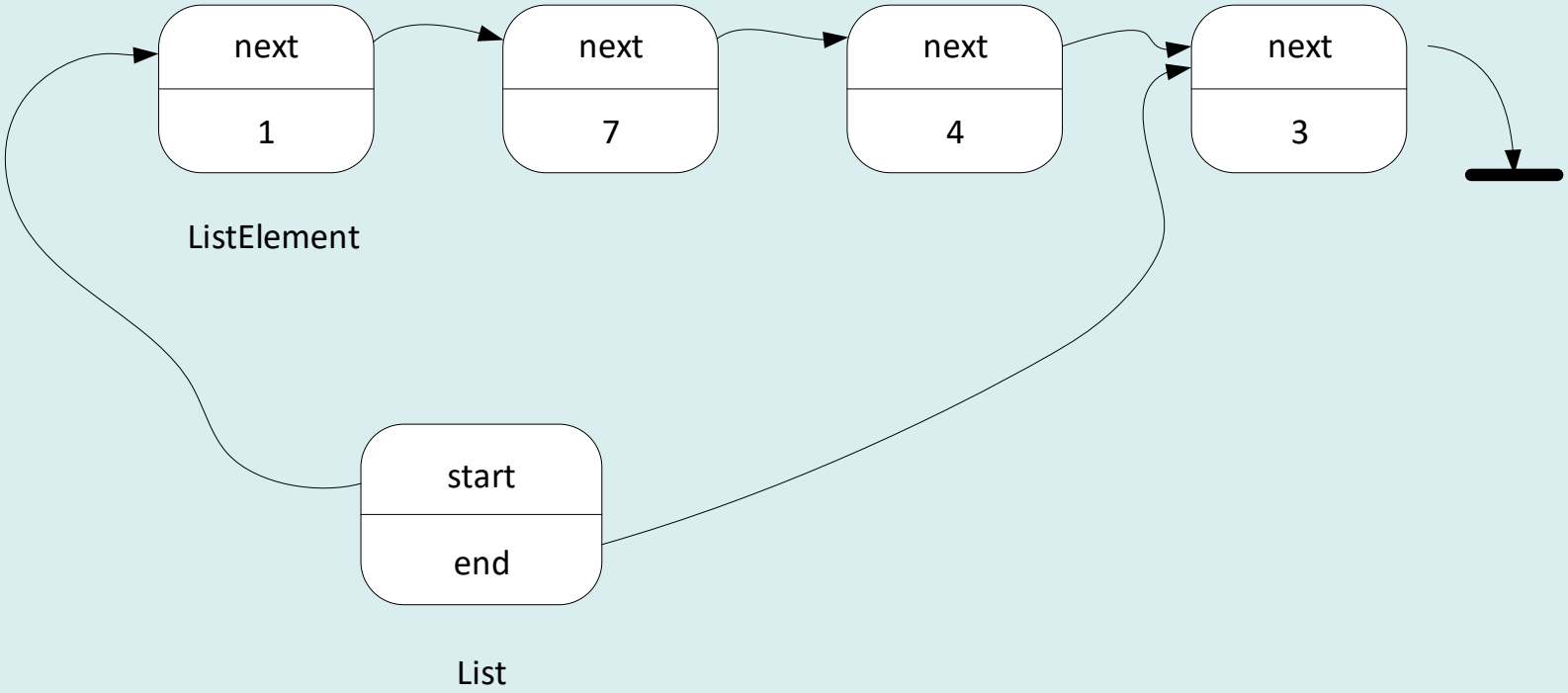
```
int main(){
    Vector vect;
    for(int i=0;i<5;i++){
        vect.pushFront(i);
        vect.pushBack(i+5);
    }
    vect.insertAt(5,100);
    vect.insertAt(0,100);
    vect.insertAt(1000,100);
    vect.dump();
    for( int i=0;i<6;i++){
        vect.deleteBack();
        vect.deleteFront();
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

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

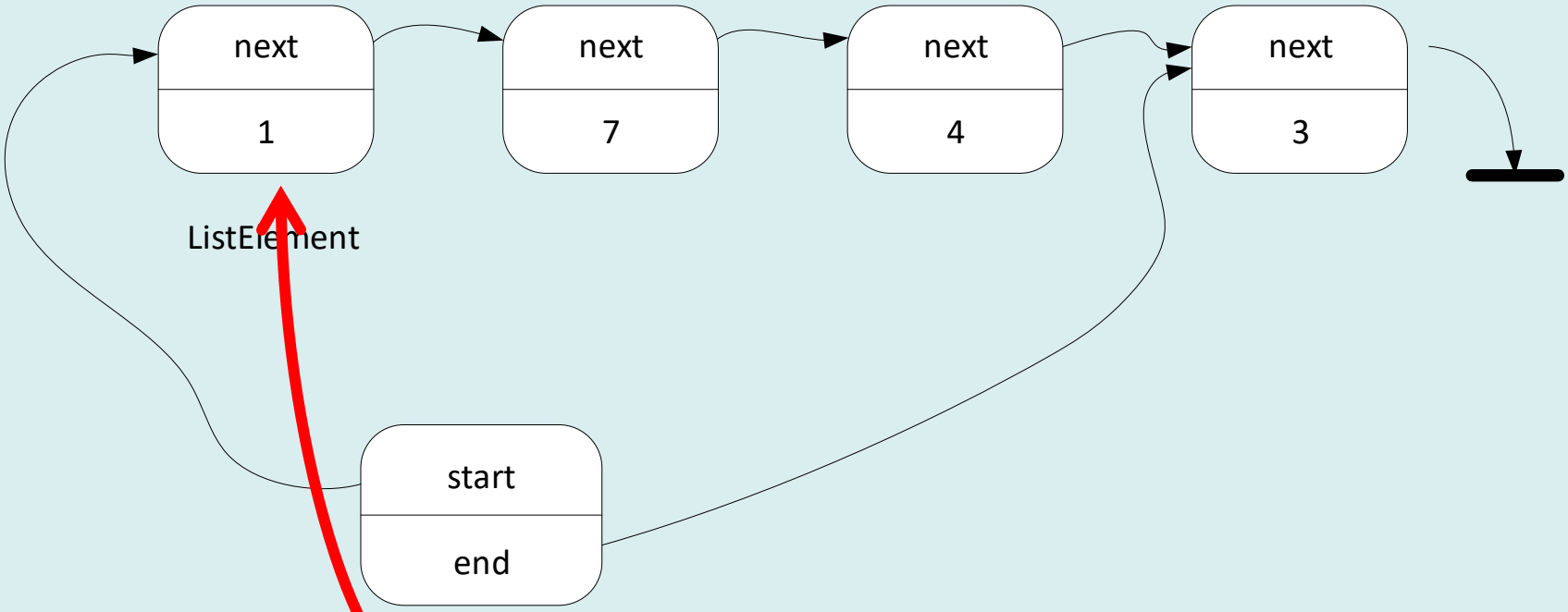
```
void List::deleteFront()
{
    if(!start)return;
    ListElement*todel=start;
    start=start->next;
    delete todel;
    if(start==0)end=0;
}

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

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

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

Iteracja po tablicy

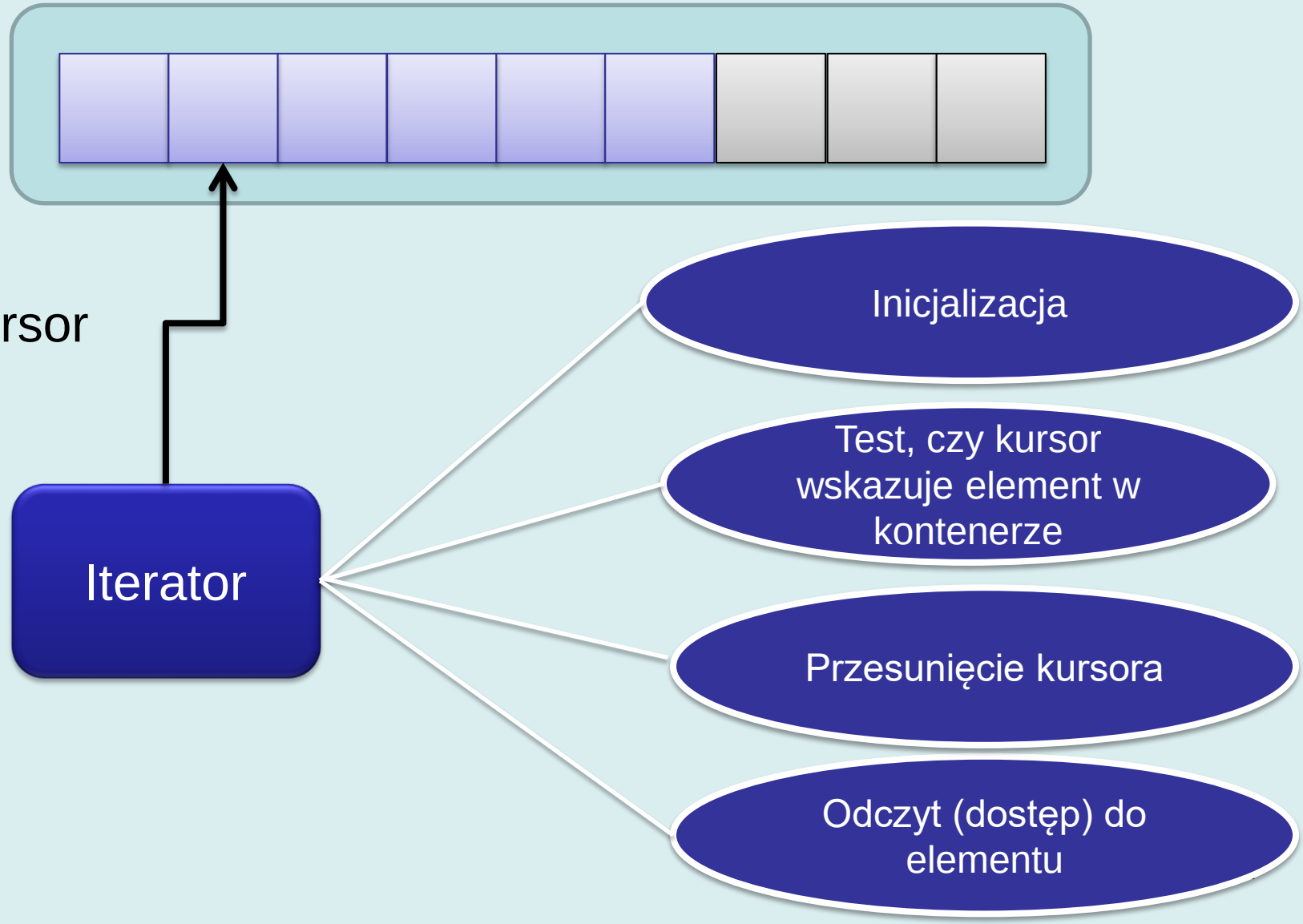
```
for(ListElement*i=start;i!=0;i=i->next)
    cout<<i->value<<" ";
```

Iteracja po liście

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 }
```

Operacje kopiowania i przypisania

```
List l1;
l1.pushBack(123);

// kopiowanie
List l2=l1;

// przypisanie
List l3;
l3.pushBack(234);
l3=l2;
```

Obiekt `l2` jest tworzony. Podczas wywołania konstruktora kopiującego do pustego obiektu powinna zostać skopiowana zawartość `l1`

Obiekt `l3` przechowuje dane. Podczas wywołania operatora przypisania jego zawartość powinna zostać usunięta i zastąpiona skopiowaną zawartością `l2`

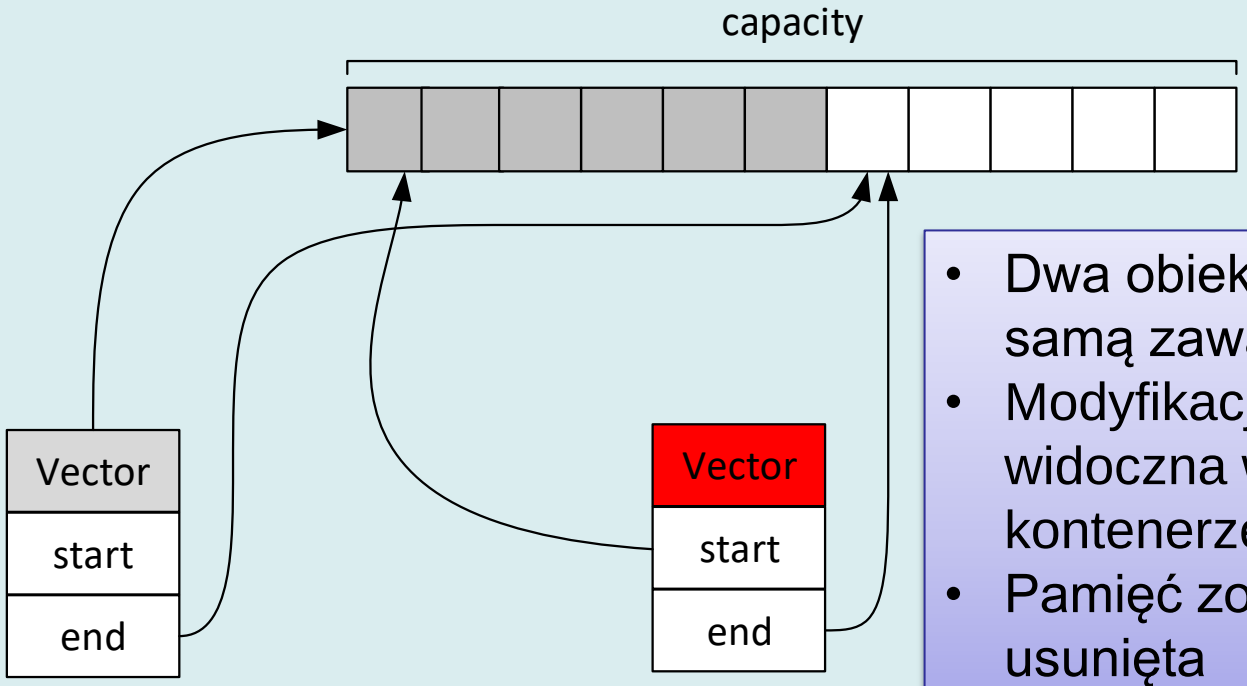
Dla klas `Vector` i `List` kompilator zapewni standardową implementację konstruktora kopiującego i operatora przypisania.

Zadziała ona błędnie.

Operacje kopiowania i przypisania

```
Vector v;
for(int i =0;i<1000000;i++)v.pushBack(i);

Vector v2=v;
```



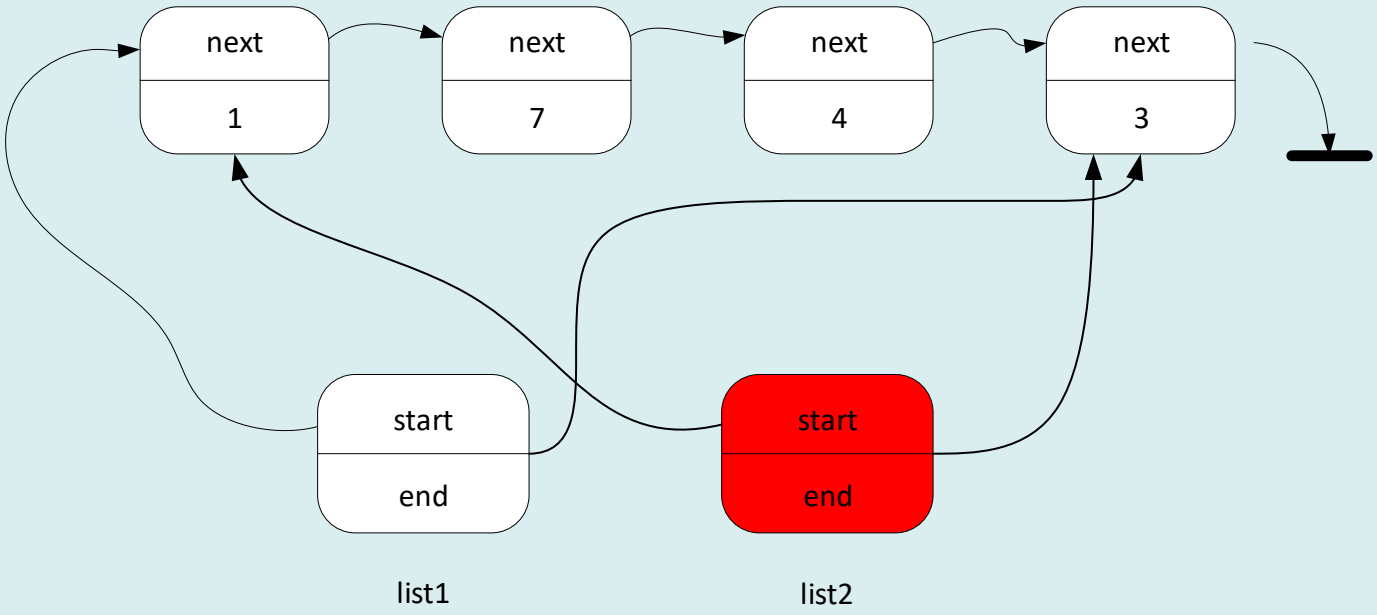
- Dwa obiekty wskazują tę samą zawartość.
- Modyfikacja elementu jest widoczna w drugim kontenerze
- Pamięć zostanie dwukrotnie usunięta

Operacje kopiowania i przypisania

Analogiczne problemy pojawiają się dla listy

```
List list1;
for(int i =0;i<1000000;i++)list1.pushBack(i);

List list2=list1;
```



Wektor - modyfikacje

```
class Vector{
    friend class VectorIterator;
protected:
    double*start;
    double*end;
    int capacity;
    void copy(const Vector&other);
    void free();

public:
    Vector(int size = 0);
    Vector(const Vector&other){ copy(other); }
    ~Vector(){free();}
    Vector&operator=(const Vector&other);
//...
```

copy() – kopiuje zawartość drugiego obiektu

free() – usuwa dane

Wektor - modyfikacje

```
void Vector::free(){
    if (start!=0)delete []start;
    start=end=0;
    capacity=0;
}

void Vector::copy(const Vector&other){
    start=end=0;
    reserve(other.capacity);
    if(other.getSize()){
        for(int i=0;i<other.getSize();i++){
            start[i] = other.start[i];
        }
        end = start + other.getSize();
    }
}
```

Wektor – operator przypisania

```
Vector&Vector::operator=(const Vector&other){
    if(&other!=this){
        free();
        copy(other);
    }
    return *this;
}
```

Lista - modyfikacje

```
class List
{
    friend class ListIterator;
protected:
    void copy(const List&other);
    void free();
public:
    ListElement*start;
    ListElement*end;

    List();
    List(const List&other){
        copy(other);
    }
    ~List(){free();}
    List&operator=(const List&other);
    //...
```

Lista - modyfikacje

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

void List::copy(const List&other){
    start=end=0;
    for(ListElement*i=other.start;i!=0;i=i->next){
        pushBack(i->value);
    }
}
List&List::operator=(const List&other){
    if(&other!=this){
        free();
        copy(other);
    }
    return *this;
}
```

Wielokrotne wykorzystanie kodu

Implementacje kontenerów danego rodzaju (np.: list, wektorów) różniących się jedynie typem przechowywanych obiektów są najczęściej bardzo podobne.

W języku C++ można wskazać co najmniej trzy techniki wielokrotnego wykorzystania stworzonego kodu kontenera.

- kopiowanie kodu
- skorzystanie z dziedziczenia
- użycie szablonów

Kopiowanie kodu

Kopiowanie kodu i zmiana typów przechowywanych obiektów jest techniką najprostszą. Ze względu na konieczność ręcznej modyfikacji kodu łatwo jest wprowadzić wiele błędów. Technika powiększa rozmiary kodu wykonywalnego programów.

```
class Vector{
 // ...
    double*start;
    double*end;
// ...
public:
    bool pushFront(double v);
    bool pushBack(double v);
    bool insertAt(int where,double v);
    bool deleteFront();
    bool deleteBack();
    double&elementAt(int i)const;
 // ...
```

Dziedziczenie

Jest to rozwiązanie typowe dla języków obiektowych typu Java, a także dla wczesnych implementacji bibliotek w C++. Zakłada się, że drogą do wielokrotnego użycia kodu kontenera jest dziedziczenie.

- Kontener implementowany jest w ten sposób, by przechowywać obiekty klasy bazowej (`Object`, `CObject`, itd.). **Kontener przechowuje wskaźniki do elementów**.
- Sam **kontener jest również klasą potomną klasy `Object`**, stąd możliwa jest implementacja skomplikowanych struktur, w których kontenery mogą zawierać inne kontenery.
- Chcąc umieścić własne dane w kontenerze należy stworzyć klasę dziedziczącą po klasie `Object` lub innej klasie, której przodkiem jest `Object`.

Dziedziczenie

```
class Object
{
public:
    virtual ~Object(){}
};
```

- `Object**start` -- tablica zawiera wskaźniki do obiektów klasy `Object` lub potomnych.
- Kontener może przechowywać obiekty różnych typów.
- Pamięć dla obiektów przydzielona jest na stercie.

```
class VectorObj : public Object
{
protected:
    Object**start;
    Object**end;
    int capacity;
    void free() {
        if (start!=0) {
            for (int i = 0; i<getSize();i++)
                delete start[i];
            delete start;
        }
        start=end=0;
    }
public:
    VectorObj(){start=end=0;capacity=0;}
    virtual ~ VectorObj(){free();}
    //...
```

Dziedziczenie

W kontenerze nie można przechowywać zmiennych typów wbudowanych. Konieczne jest obiektowe opakowanie.

```
class Int: public Object
{
public:
    int value;
    Int(int v=0){value=v;}
    operator int(){return value;}
};


void foo(){
    VectorObj v;
    for(int i=0;i<10;i++)v.pushBack(new Int(i));
}
```