Programowanie obiektowe w języku C++

Stanisław Gepner

sgepner@meil.pw.edu.pl

STL
Standard Template Library

- Biblioteka szablonów - generyczna
- Kontenery (wektor, mapa, lista, ...)
- Z typami wbudowanymi oraz programisty
- Iteratory - służą do poruszania się po kolekcjach
- ... trochę jak wskaźniki, można inkrementować przez $++$, wyłuskać przez * i porównywać przez $!=$
- Algorytmy - czyli wzorce funkcji
- STL jest bardzo popularny

Iteratory

```
std::nazwa_kolekcji<parametr template>::iterator nazwa iteratora
```

- Pozwalają na generyczność STL
- Sposób na dostęp do elementów przechowywanych przez kontener
- W zależności od kontenera mogą być różne
  - input iterator: pozwala na odczyt, nie na zapis
  - output iterator: pozwala na zapis
  - forward iterator: odczyt, zapis, tylko ruch do przodu, od początku do końca
  - bidirectional iterator: także do tyłu
  - random access: dowolny dostęp
  - np. vector posiada random access, a lista tylko bidirectional
  - to znaczy, ze możemy przesuwać się tylko o jeden element

Kontenery

- Obiekty do przechowywania danych
- Sekwencyjne: wektor, lista, ...
- Asocjacyjne - przechowuje pary (klucz, wartość), umożliwia dostęp przez podanie klucza: set, map ...
- Też: kolejka (queue) i stack

vector

Tablica dynamiczna

- Jak tablica dynamiczna w C - ciągły obszar
- Możliwy dostęp do dowolnego elementu w stałym czasie
- Może automatycznie zmienić rozmiar przy dodawaniu / usuwaniu elementu
- Dodawanie / usuwanie na końcu - stały czas
- Może zaistnieć konieczność przeniesienia
- Dodawanie / usuwanie 'w środku' - czas liniowy bo trzeba przenieść elementy
- Dobry jeżeli dodajemy tylko na końcu, znamy rozmiar, często potrzebny dostęp do elementów

```
#include <vector>

std::vector<typ_przechowywany>
     nazwa; //pusty
std::vector<typ_przechowywany>
     nazwa2; = {a1,a2,a3};//3
     elementy
std::vector<int> nazwa3(4,100);
     // 4 int o wartosci 100
std::vector<int> nazwa4(third);
     // kopia nazwa3

//ale tez:
int myints[] = {16,2,77,29};
std::vector<int> nazwa5 (myints,
      myints + 4 );

//dostep przez []
cout << nazwa5[3] << endl;
```

Vector
Dostęp

- [] dostęp do elementu
- front() - pierwszy element
- back() - ostatni element

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> vec(5, 1); // 5 jedynek
  vec.front() = 9;
  vec.back() = 2;

  std::cout << "vec contains:";
  for (unsigned i=0; i<vec.size() ; i++)
    std::cout << ' ' << vec[i];
  std::cout << '\n';

  return 0;
}
```

Vector
Iteratory

- Dowolny dostęp
- Bidirectional
- *begin()* - iterator do pierwszego elementu
- *end()* - iterator za ostatni element

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> moj_vec = {16,2,77,29}

  std::cout << "moj_vec:";
  for (std::vector<int>::iterator it = moj_vec.begin() ; it !=
       moj_vec.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Vector
Rozmiar

- Wektor zajmuje więcej pamięci niż konieczne, by uniknąć realokacji
- *size* - bieżący rozmiar,
- *max size* - rozmiar maksymalny
- *capacity* - Za alokowany rozmiar, niekoniecznie *size*
- *empty* - test czy pusty
- *reserve* - zarezerwój rozmiar, zmienia *capacity*. Jeżeli wystąpi realokacja koszt najwyżej liniowy z rozmiarem
- *resize* - Zmienia rozmiar. Liniowy z liczbą dodawanych/usuwanych elementów, jeżeli realokacja koszt liniowy z rozmiarem

```
#include <iostream>
#include <vector>

int main (){
  std::vector<int> vec;
  vec.resize(100, 0);
  for (int i=0; i<100; i++) vec[i]=i;;

  std::cout << "size:␣" << (int) vec.size() << '\n';
  std::cout << "capacity:␣" << (int) vec.capacity() << '\n';
  std::cout << "max_size:␣" << (int) vec.max_size() << '\n';
  return 0;
}
```

Vector
Zmiany

- *push back* - dodaje element na końcu. koszt stały chyba, że realokacja
- *pop back* - Kasuje ostatni. Stały
- *insert* - wstawia element - Koszt liniowy z liczbą wstawionych elementów plus liczba elementów po - przesuwanych
- *erase* - usuwa elementy. Koszt liniowy z liczną usuwanych plus przesunięcia
- *swap* - wymienia zawartość z innym. Stały
- *clear* - czyści zawartość. Liniowy

```
#include <iostream>
#include <vector>
int main (){
  std::vector<int> vec;
  int val=10;
  do {
    vec.push_back (val);
    --val;
  } while (val);
  std::cout << "vec stores " << vec.size() << " numbers.\n";
  return 0;
}
```

list

- Pozwala na szybkie dodawanie usuwanie elementów
- Nie koniecznie ciągła w pamięci
- Elementy w rożnych miejscach
- Brak dostępu bezpośredniego
- Np. Dostęp do 7 elementu wymaga przejści przez 6 poprzednich
- Wymaga dodatkowej informacji o sąsiadach - więcej pamięci
- Dobre do częstego wstawiania, usuwania, przenoszenia

```
#include <list>

std::list<int> first;//pusta
std::list<int> second(4,100);//4
      100
std::list<int> fourth(third);//
     kopia
```

list

Dostęp

- front() - pierwszy element
- back() - ostatni element

```
#include <iostream>
#include <list>

int main ()
{
  std::list<int> mylist;

  mylist.push_back(77);
  mylist.push_back(22);

  // now front equals 77, and back 22

  mylist.front() -= mylist.back();

  std::cout << "mylist.front()␣is␣now␣" << mylist.front() << '\n';

  return 0;
}
```

list

Iteratory

- Bidirectional do typu
- *begin()* - iterator do pierwszego elementu
- *end()* - iterator za ostatni element

```
#include <iostream>
#include <list>

int main ()
{
  int myints[] = {75,23,65,42,13};
  std::list<int> mylist (myints,myints+5);

  std::cout << "mylist␣contains:";
  for (std::list<int>::iterator it=mylist.begin(); it != mylist.end
       (); ++it)
    std::cout << '␣' << *it;

  std::cout << '\n';

  return 0;
}
```

list

Rozmiar

- *size* - bieżący rozmiar,
- *max size* - rozmiar maksymalny
- *empty* - test czy pusty

```
#include <iostream>
#include <list>
int main (){
  std::list<int> mylist;
  std::cout << "0. size: " << mylist.size() << '\n';

  for (int i=0; i<10; i++) mylist.push_back(i);
  std::cout << "size: " << mylist.size() << '\n';
  int sum (0);
  while (!mylist.empty())
  {
     sum += mylist.front();
     mylist.pop_front();
     std::cout << "size: " << mylist.size() << '\n';
  }
  return 0;
}
```

list

Zmiany

- *push front* i *pop front* - dodaj / usuń na początku - Stały
- *push back* i *pop back* - dodaj / usuń na końcu - Stały
- *insert* - wstawia element - liniowy z liczbą elementów
- *erase* - usuwa elementy - liniowy z liczbą elementów
- *swap* - wymienia zawartość z innym. Stały
- *resize* - zmienia rozmiar
- *clear* - czyści zawartość.

```
#include <iostream>
#include <list>
int main (){
  std::list<int> mylist (2,100);// two ints with a value of 100
  mylist.push_front (200);
  mylist.push_front (300);
  std::cout << "mylist␣contains:";
  for (std::list<int>::iterator it=mylist.begin(); it!=mylist.end();
       ++it)
    std::cout << '␣' << *it;
  std::cout << '\n';
  return 0;
}
```

pair

- Przechowują unikalne elementy
- Zawsze posortowane
- Element jest jednocześnie kluczem
- Tylko jeden element o danej wartości

```
template <class T1, class T2>
struct pair;
```

```
#include <utility> // std::pair, std::make_pair
#include <string>
#include <iostream>
int main () {
  std::pair<std::string,double> product1;
  std::pair<std::string,double> product2 ("
      tomatoes",2.30);
  std::pair<std::string,double> product3 (
      product2);
  product1 = std::make_pair(std::string("
      lightbulbs"),0.99);
  product2.first = "shoes";                    //
      the type of first is string
  product2.second = 39.90;                     //
      the type of second is double
  std::cout << "The␣price␣of␣" << product1.firs
       << "␣is␣$" << product1.second << '\n';
  std::cout << "The␣price␣of␣" << product2.firs
       << "␣is␣$" << product2.second << '\n';
  std::cout << "The␣price␣of␣" << product3.firs
       << "␣is␣$" << product3.second << '\n';
  return 0;
}
```

set

- Przechowują unikalne elementy
- Zawsze posortowane
- Element jest jednocześnie kluczem
- Tylko jeden element o danej wartości

```
#include <iostream>
#include <set>
bool fncomp (int lhs, int rhs) {return lhs<rhs;}
struct clcomp {
  bool operator() (const int& lhs, const int&
       rhs) const {return lhs<rhs;}
};

int main ()
{
  std::set<int> first;//pusty
  int myints[]= {10,20,30,40,50};
  std::set<int> second (myints,myints+5);
  std::set<int> third (second);//kopia
  std::set<int> fourth (second.begin(), second.
       end());  // iterator ctor.
  std::set<int,clcomp> fifth;                    //
       class as Compare
  bool(*fn_pt)(int,int) = fncomp;
  std::set<int,bool(*)(int,int)> sixth (fn_pt);
       // function pointer as Compare
  return 0;
}
```

set

Iteratory

- Bidirectional do typu
- *begin()* - iterator do pierwszego elementu
- *end()* - iterator za ostatni element

```
#include <iostream>
#include <set>

int main ()
{
  int myints[] = {75,23,65,42,13};
  std::set<int> myset (myints,myints+5);

  std::cout << "myset␣contains:";
  for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++
       it)
    std::cout << '␣' << *it;

  std::cout << '\n';

  return 0;
}
```

set
Rozmiar

- *size* - bieżący rozmiar,
- *max size* - rozmiar maksymalny
- *empty* - test czy pusty

```
#include <iostream>
#include <set>

int main ()
{
  std::set<int> myints;
  std::cout << "0. size: " << myints.size() << '\n';

  for (int i=0; i<10; ++i) myints.insert(i);
  std::cout << "1. size: " << myints.size() << '\n';

  myints.insert (100);
  std::cout << "2. size: " << myints.size() << '\n';

  myints.erase(5);
  std::cout << "3. size: " << myints.size() << '\n';

  return 0;
}
```

set
Zmiany

```
#include <iostream>
#include <set>
int main (){
  std::set<int> myset;
  std::set<int>::iterator it;
  std::pair<std::set<int>::iterator,bool> ret;
  // set some initial values:10 20 30 40 50
  for (int i=1; i<=5; ++i) myset.insert(i*10);
  ret = myset.insert(20);
  if (ret.second==false)
    it=ret.first;  // "it" now points 20
  myset.insert (it,25);  // max efficiency
  myset.insert (it,24);  // max efficiency
  myset.insert (it,26);  // no max efficiency
  int myints[]= {5,10,15};// 10 already in set
  myset.insert (myints,myints+3);
  std::cout << "myset␣contains:";
  for (it=myset.begin(); it!=myset.end(); ++it)
    std::cout << '␣' << *it;
  std::cout << '\n';
  return 0;
}
```

- *insert* - wstawia element - logarytmiczny, lub stały
- *erase* - usuwa elementy - logarytmiczny, stały lub liniowy
- *swap* - wymienia zawartość z innym. Stały
- *clear* - czyści zawartość.

map

- Przechowują pary (klucz, wartość), klucz jest unikalny
- Zawsze posortowane po kluczu
- podobne do set
- Dostęp przez [] lub find jest obarczony kosztem logarytmicznym

```
#include <iostream>
#include <map>
bool fncomp (char lhs, char rhs) {return lhs<rhs
    ;}
struct classcomp {
  bool operator() (const char& lhs, const char&
      rhs) const
  {return lhs<rhs;}
};
int main ()
{
  std::map<char,int> first;
  first['a']=10;
  first['b']=30;
  first['c']=50;
  first['d']=70;
  std::map<char,int> second (first.begin(),first
      .end());
  std::map<char,int> third (second);
  std::map<char,int,classcomp> fourth;
                          // class as Compare
  bool(*fn_pt)(char,char) = fncomp;
  std::map<char,int,bool(*)(char,char)> fifth (
      fn_pt); // function pointer as Compare
  return 0;
}
```

map
Iteratory

- Bidirectional do typu
- *begin()* - iterator do pierwszego elementu
- *end()* - iterator za ostatni element

```
#include <iostream>
#include <map>
int main (){
  std::map<char,int> mymap;

  mymap['b'] = 100;
  mymap['a'] = 200;
  mymap['c'] = 300;
  // show content:
  for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end
       (); ++it)
    std::cout << it->first << "␣=>␣" << it->second << '\n';
  return 0;
}
```

map
Rozmiar

- *size* - bieżący rozmiar,
- *max size* - rozmiar maksymalny
- *empty* - test czy pusty

```
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;

  mymap['a']=10;
  mymap['b']=20;
  mymap['c']=30;

  while (!mymap.empty())
  {
    std::cout << mymap.begin()->first << "␣=>␣" << mymap.begin()->
        second << '\n';
    mymap.erase(mymap.begin());
  }

  return 0;
}
```

map
Dostęp

```
#include <iostream>
#include <set>
int main (){
  std::set<int> myset;
  std::set<int>::iterator it;
  std::pair<std::set<int>::iterator,bool> ret;
  // set some initial values:10 20 30 40 50
  for (int i=1; i<=5; ++i) myset.insert(i*10);
  ret = myset.insert(20);
  if (ret.second==false)
    it=ret.first;  // "it" now points 20
  myset.insert (it,25);  // max efficiency
  myset.insert (it,24);  // max efficiency
  myset.insert (it,26);  // no max efficiency
  int myints[]= {5,10,15};// 10 already in set
  myset.insert (myints,myints+3);
  std::cout << "myset␣contains:";
  for (it=myset.begin(); it!=myset.end(); ++it)
    std::cout << '␣' << *it;
  std::cout << '\n';
  return 0;
}
```

- *insert* - wstawia element - logarytmiczny, lub stały, zwraca parę (iterator, bool)
- *find* - znajduje element, zwraca iterator do lub .end() - logarytmiczny
- *[]* - jak find, logarytmiczny
- *clear* - czyści zawartość.

for each

- Stosuje funkcję do wszystkich elementów z zakresu

```
#include <iostream>      // std::cout
#include <algorithm>     // std::for_each
#include <vector>        // std::vector

void myfunction (int i) {  // function:
  std::cout << ' ' << i;
}

struct myclass {             // function object type:
  void operator() (int i) {std::cout << ' ' << i;}
} myobject;

int main () {
  std::vector<int> myvector;
  myvector.push_back(10);
  myvector.push_back(20);
  myvector.push_back(30);

  std::cout << "myvector contains:";
  for_each (myvector.begin(), myvector.end(), myfunction);
  std::cout << '\n';

  // or:
  std::cout << "myvector contains:";
  for_each (myvector.begin(), myvector.end(), myobject);
  std::cout << '\n';
```

find

- Zwraca iterator do pierwszego wystąpienia elementu

```
// find example
#include <iostream>       // std::cout
#include <algorithm>      // std::find
#include <vector>         // std::vector

int main () {
  // using std::find with array and pointer:
  int myints[] = { 10, 20, 30, 40 };
  int * p;

  p = std::find (myints, myints+4, 30);
  if (p != myints+4)
    std::cout << "Element found in myints: " << *p << '\n';
  else
    std::cout << "Element not found in myints\n";

  // using std::find with vector and iterator:
  std::vector<int> myvector (myints,myints+4);
  std::vector<int>::iterator it;

  it = find (myvector.begin(), myvector.end(), 30);
  if (it != myvector.end())
    std::cout << "Element found in myvector: " << *it << '\n';
  else
    std::cout << "Element not found in myvector\n";

  return 0;
```

fill

- Wypełnia wartością

```
// fill algorithm example
#include <iostream>       // std::cout
#include <algorithm>      // std::fill
#include <vector>         // std::vector

int main () {
  std::vector<int> myvector (8);// myvector: 0 0 0 0 0 0 0 0

  std::fill (myvector.begin(),myvector.begin()+4,5);   // myvector:
       5 5 5 5 0 0 0 0
  std::fill (myvector.begin()+3,myvector.end()-2,8);   // myvector:
       5 5 5 8 8 8 0 0

  std::cout << "myvector␣contains:";
  for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.
       end(); ++it)
    std::cout << '␣' << *it;
  std::cout << '\n';

  return 0;
}
```

sort

- Sortuje rosnąco

```
// sort algorithm example
#include <iostream>       // std::cout
#include <algorithm>      // std::sort
#include <vector>         // std::vector
bool myfunction (int i,int j) { return (i<j); }
struct myclass {
  bool operator() (int i,int j) { return (i<j);}
} myobject;
int main () {
  int myints[] = {32,71,12,45,26,80,53,33};
  std::vector<int> myvector (myints, myints+8);
  // using default comparison (operator <):
  std::sort (myvector.begin(), myvector.begin()+4);
  // using function as comp
  std::sort (myvector.begin()+4, myvector.end(), myfunction);
  // using object as comp
  std::sort (myvector.begin(), myvector.end(), myobject);
  // print out content:
  std::cout << "myvector␣contains:";
  for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.
       end(); ++it)
    std::cout << '␣' << *it;
  std::cout << '\n';
  return 0;
}
```

unique

Działa na posortowanej kolekcji. Powtarzające elementy zostaję przesunięte na koniec kolekcji, zwraca iterator za ostatni unikalny element.

lower bound

- Zwraca iterator do elementu nie mniejszego niż argument
- Zakres musi być posortowany
- koszt logarytmiczny

```
// lower_bound/upper_bound example
#include <iostream>     // std::cout
#include <algorithm>    // std::lower_bound, std::upper_bound, std::
    sort
#include <vector>       // std::vector

int main () {
  int myints[] = {10,20,30,30,20,10,10,20};
  std::vector<int> v(myints,myints+8);           // 10 20 30 30 20
      10 10 20
  std::sort (v.begin(), v.end());                // 10 10 10 20 20
      20 30 30
  std::vector<int>::iterator low,up;
  low=std::lower_bound (v.begin(), v.end(), 20); //          ^
  up= std::upper_bound (v.begin(), v.end(), 20); //
                        ^
  std::cout << "lower_bound at position " << (low- v.begin()) << '\n
      ';
  std::cout << "upper_bound at position " << (up - v.begin()) << '\n
      ';
  return 0;
}
```

binary search

- Zwraca *true* gdy jest i *false* gdy nie
- Kolekcja musi być posortowana
- Koszt logarytmiczny

```
// binary_search example
#include <iostream>      // std::cout
#include <algorithm>     // std::binary_search, std::sort
#include <vector>        // std::vector
bool myfunction (int i,int j) { return (i<j); }
int main () {
  int myints[] = {1,2,3,4,5,4,3,2,1};
  std::vector<int> v(myints,myints+9);                         // 1
       2 3 4 5 4 3 2 1
  // using default comparison:
  std::sort (v.begin(), v.end());
  std::cout << "looking for a 3... ";
  if (std::binary_search (v.begin(), v.end(), 3))
    std::cout << "found!\n"; else std::cout << "not found.\n";
  // using myfunction as comp:
  std::sort (v.begin(), v.end(), myfunction);
  std::cout << "looking for a 6... ";
  if (std::binary_search (v.begin(), v.end(), 6, myfunction))
    std::cout << "found!\n"; else std::cout << "not found.\n";
  return 0;
}
```