Библиотека стандардних шаблона (Standard template library – STL)

Уобичајени задаци које обављамо у програмирању

- Смештање података у контејнере
- Организовање података
  - за испис
  - за брз приступ
- Добављање вредности
  - по индексу („дај ми n-ти елемент“)
  - по вредности („дај ми први елемент чија је вредност 67“)
  - по особини („дај ми све елементе чија вредност је мања од 67“)
- Додавање података
- Уклањање података
- Уређивање и претраживање
- Једноставне нумеричке операције

Подизање алгоритма

```
double sum(double array[], int n)
{
  double s{0.0};
  for (int i = 0; i < n; ++i) s = s + array[i];
  return s;
}
```

```
struct Node { Node* next; int data; };

int sum(Node* first)
{
  int s{0};
  while (first != nullptr) {
       s += first->data;
       first = first->next;
  }
  return s;
}
```

Неутрални елемент

```
double sum(double array[], int n)
{
  double s{0.0};
  for (int i = 0; i < n; ++i) s = s + array[i];
  return s;
}
```

```
struct Node { Node* next; int data; };

int sum(Node* first)
{
  int s {0};
  while (first != nullptr) {
       s += first->data;
       first = first->next;
  }
  return s;
}
```

Сабирање

```
double sum(double array[], int n)
{
  double s{0.0};
  for (int i = 0; i < n; ++i) s = s + array[i];
  return s;
}
```

```
struct Node { Node* next; int data; };

int sum(Node* first)
{
  int s{0};
  while (first != nullptr) {
       s += first->data;
       first = first->next;
  }
  return s;
}
```

Почетак

```
double sum(double array[], int n)
{
  double s{0.0};
  for (int i = 0; i < n; ++i) s = s + array[i];
  return s;
}
```

```
struct Node { Node* next; int data; };

int sum(Node* first)
{
  int s{0};
  while (first != nullptr) {
       s += first->data;
       first = first->next;
  }
  return s;
}
```

Није крај

```
double sum(double array[], int n)
{
  double s{0.0};
  for (int i = 0; i < n; ++i) s = s + array[i];
  return s;
}
```

```
struct Node { Node* next; int data; };

int sum(Node* first)
{
  int s{0};
  while (first != nullptr) {
       s += first->data;
       first = first->next;
  }
  return s;
}
```

Добави вредност

```
double sum(double array[], int n)
{
  double s{0.0};
  for (int i = 0; i < n; ++i) s = s + array[i];
  return s;
}
```

```
struct Node { Node* next; int data; };

int sum(Node* first)
{
  int s{0};
  while (first != nullptr) {
       s += first->data;
       first = first->next;
  }
  return s;
}
```

Добави следећи елемент

```
double sum(double array[], int n)
{
  double s{0.0};
  for (int i = 0; i < n; ++i) s = s + array[i];
  return s;
}
```

```
struct Node { Node* next; int data; };

int sum(Node* first)
{
  int s{0};
  while (first != nullptr) {
      s += first->data;
      first = first->next;
  }
  return s;
}
```

Подизање алгоритма

```
// псеудо код

   data_type sum(data) {
    data_type s = 0;
    while (not at end) {
          s = s + get value;
          get next data element;
    }
    return s;
   }
```

- Потребне су нам две операције над типом елемената:
  - Постављање на неутралну вредност
  - Сабирање

Подизање алгоритма

```
// псеудо код

  data_type sum(data) {
   data_type s = 0;
   while (not at end) {
        s = s + get value;
        get next data element;
   }
   return s;
  }
```

- Потребне су нам три операције над структуром података:
  - Није крај
  - Добави вредност
  - Добави следећи елемент
- Плус, почетак (имплицитно део „добави вредност“ и саме структуре подата)

Подизање алгоритма

```
double r1 = sum(niz, N);
int r2 = sum(head);
```

```
double sum(double array[], int n)
{
  double s{0.0};
  for (int i = 0; i < n; ++i) s = s + array[i];
  return s;
}
```

```
struct Node { Node* next; int data; };

int sum(Node* first)
{
  int s{0};
  while (first != nullptr) {
       s += first->data;
       first = first->next;
  }
  return s;
}
```

Подизање алгоритма

```
double r1 = sum(niz, 0, N);
int r2 = sum(head, nullptr);
```

```
double sum(double array[], int first, int last)
{
  double s{0.0};
  for (int i = first; i < last; ++i) s = s + array[i];
  return s;
}
```

```
struct Node { Node* next; int data; };

int sum(Node* first, Node* last)
{
  int s{0};
  while (first != last) {
       s += first->data;
       first = first->next;
  }
  return s;
}
```

Подизање алгоритма

```
double r1 = sum(niz, 0, N, 0);
int r2 = sum(head, nullptr, 0.0);
```

```
double sum(double array[], int first, int last, double start)
{
  double s{start};
  for (int i = first; i < last; ++i) s = s + array[i];
  return s;
}
```

```
struct Node { Node* next; int data; };

int sum(Node* first, Node* last, int start)
{
  int s{start};
  while (first != last) {
       s += first->data;
       first = first->next;
  }
  return s;
}
```

Подизање алгоритма

```
double r1 = sum(niz, 0, N, 0);
int r2 = sum(head, nullptr, 0.0);
```

```
double sum(double array[], int first, int last, double s)
{
  for (int i = first; i < last; ++i) s = s + array[i];
  return s;
}
```

```
struct Node { Node* next; int data; };

int sum(Node* first, Node* last, int s)
{
  while (first != last) {
       s += first->data;
       first = first->next;
  }
  return s;
}
```

Подизање алгоритма

```
double r1 = sum(niz, 0, N, 0);
int r2 = sum(head, nullptr, 0.0);
```

```
double sum(double array[], int first, int last, double s)
{
  for (int i = first; i < last; ++i) s = s + array[i];
  return s;
}
```

```
struct Node { Node* next; int data; };

int sum(Node* first, Node* last, int s)
{
  while (first != last) {
       s = s + first->data;
       first = first->next;
  }
  return s;
}
```

Подизање алгоритма

```
double r1 = sum(niz, 0, N, 0);
int r2 = sum(head, nullptr, 0.0);
```

```
double sum(double array[], int first, int last, double s) {
  while (first < last) {
       s = s + array[first];
       ++first;
  }
  return s;
}
```

```
struct Node { Node* next; int data; };

int sum(Node* first, Node* last, int s) {
  while (first != last) {
       s = s + first->data;
       first = first->next;
  }
  return s;
}
```

Подизање алгоритма

```
double r1 = sum(niz, 0, N, 0);
int r2 = sum(head, nullptr, 0.0);
```

```
double sum(double array[], int first, int last, double s) {
  while (first != last) {
      s = s + array[first];
      ++first;
  }
  return s;
}
```

```
struct Node { Node* next; int data; };

int sum(Node* first, Node* last, int s) {
  while (first != last) {
      s = s + first->data;
      first = first->next;
  }
  return s;
}
```

Подизање алгоритма

```
double r1 = sum(niz, 0, N, 0);
int r2 = sum(head, nullptr, 0.0);
```

```
double sum(double array[], int first, int last, double s) {
  while (first != last) {
      s = s + array[first];
      toNext(first);
  }
  return s;
}
```

```
void toNext(int& x) { ++x; }
void toNext(Node*& x) { x = x->next; }
```

```
struct Node { Node* next; int data; };

int sum(Node* first, Node* last, int s) {
  while (first != last) {
      s = s + first->data;
      toNext(first);
  }
  return s;
}
```

Подизање алгоритма

```
double r1 = sum(&niz[0], &niz[N], 0);
int r2 = sum(head, nullptr, 0.0);
```

```
double sum(double* first, double* last, double s) {
  while (first != last) {
      s = s + *first;
      toNext(first);
  }
  return s;
}
```

```
void toNext(double*& x) { ++x; }
void toNext(Node*& x) { x = x->next; }
```

```
struct Node { Node* next; int data; };

int sum(Node* first, Node* last, int s) {
  while (first != last) {
      s = s + first->data;
      toNext(first);
  }
  return s;
}
```

Подизање алгоритма

```
double r1 = sum(&niz[0], &niz[N], 0);
int r2 = sum(head, nullptr, 0.0);
```

```
double sum(double* first, double* last, double s) {
  while (first != last) {
      s = s + getVal(first);
      toNext(first);
  }
  return s;
}
```

```
void toNext(double*& x) { ++x; }
void toNext(Node*& x) { x = x->next; }
double getVal(double* x) { return *x; }
int getVal(Node* x) { return x->data; }
```

```
struct Node { Node* next; int data; };

int sum(Node* first, Node* last, int s) {
  while (first != last) {
      s = s + getVal(first);
      toNext(first);
  }
  return s;
}
```

Подизање алгоритма

```
double r1 = sum(&niz[0], &niz[N], 0);
int r2 = sum(head, (Node*)nullptr,
0.0);
```

```
double sum(double* first, double* last, double s) {
  while (first != last) {
      s = s + getVal(first);
      toNext(first);
  }
  return s;
}
```

```
void toNext(double*& x) { ++x; }
void toNext(Node*& x) { x = x->next; }
double getVal(double* x) { return *x; }
int getVal(Node* x) { return x->data; }
```

```
struct Node { Node* next; int data; };

int sum(Node* first, Node* last, int s) {
  while (first != last) {
      s = s + getVal(first);
      toNext(first);
  }
  return s;
}
```

```
template<class Iter, class T>
T sum(Iter first, Iter last, T s) {
  while (first != last) {
    s = s + getVal(first);
    toNext(first);
  }
  return s;
}
```

Подизање алгоритма

```
double r1 = sum(&niz[0], &niz[N], 0);
int r2 = sum(head, (Node*)nullptr,
0.0);
```

```
double sum(double* first, double* last, double s) {
  while (first != last) {
      s = s + *first;
      ++first;
  }
  return s;
}
```

```
void toNext(double*& x) { ++x; }
void toNext(Node*& x) { x = x->next; }
double getVal(double* x) { return *x; }
int getVal(Node* x) { return x->data; }
```

```
struct Node { Node* next; int data; };

int sum(Node* first, Node* last, int s) {
  while (first != last) {
      s = s + getVal(first);
      toNext(first);
  }
  return s;
}
```

Подизање алгоритма

```
double r1 = sum(&niz[0], &niz[N], 0);
int r2 = sum(head, NodeIter(nullptr),
0.0);
```

```
double sum(double* first, double* last, double s) {
  while (first != last) {
       s = s + *first;
       ++first;
  }
  return s;
}
```

```
NodeIter& operator++(NodeIter& x) {
  x->t = x->t->next; return x; }
int operator*(NodeIter x) {
  return x->t->data; }
bool operator!=(NodeIter x, NodeIter y) {
  return x->t != t->t;
}
```

```
struct Node { Node* next; int data; };
struct NodeIter { Node* t; ... };

int sum(NodeIter first, NodeIter last, int s) {
  while (first != last) {
       s = s + *first;
       ++first;
  }
  return s;
}
```

Подизање алгоритма

```
template<class Iter, class T>
T sum(Iter first, Iter last, T s)
{
  while (first != last) {
     s = s + *first;
     ++first;
 }
 return s;
}
```

- Ради за обичне низове:

```
int a[] = { 1,2,3,4,5,6,7,8 };
double d = 0;
d = sum(a, a + n, d);
```

STL

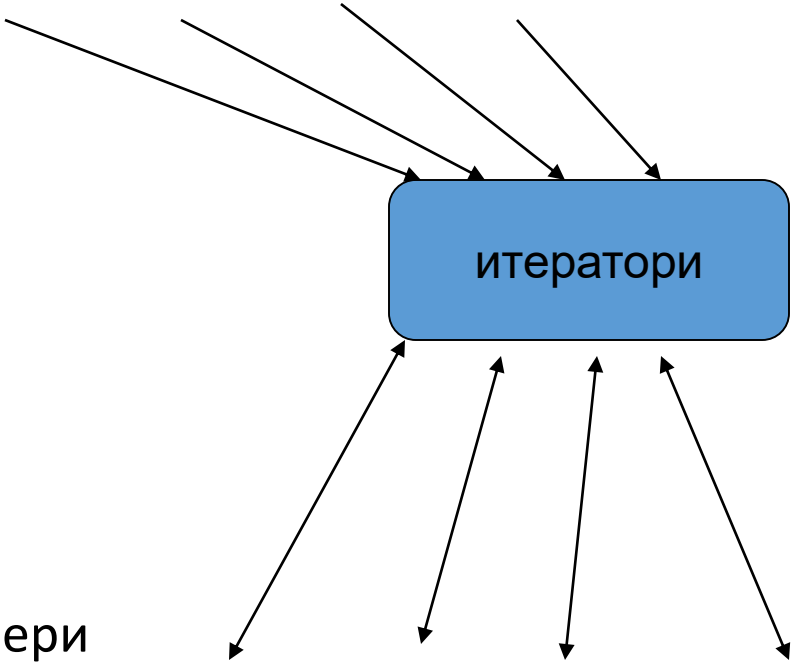
- Осмишљено од стране Александра Степанова (енгл. Alexander Stepanov; рус. Александр Степанов). Прочитати: www.stlport.org/resources/StepanovUSA.html

- Циљ: Најопштија, најефикаснија, најфлексибилнија репрезентација концепата (идеја и алгоритама)
  - Представити раздвојене концепте у раздвојеном коду
  - Слободно комбиновати концепте кад год је смислено

Основни модел

- Алгоритми

  sort, find, search, copy, …



- Контејнери

  vector, list, map, unordered_map, …

- Алгоритми баратају подацима, али их се не тичу контејнери
- Контејнери складиште податке, али их се не тичу алгоритми
- Веза између алгоритама и контејнера су итератори:
  - Сваки контејнер има своју врсту итератора, али сви они имају исту (врло сличну) спрегу

STL

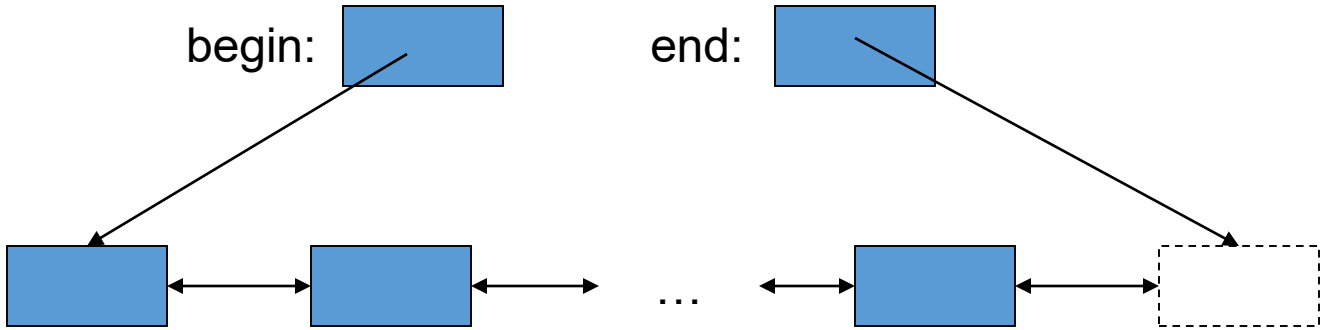
- Стандардна библиотека шаблона је стандардна Це++ библиотека са ~15 контејнера и преко 60 алгоритама
  - Постоје и друге библиотеке које су направљене у том стилу:
    - Boost.org, TBB, Microsoft, SGI, …

STL

- Ако знате основне концепте и неколико примера, онда лако можете користити и остатак библиотеке.
- Документација
  - Cppreference
    - https://en.cppreference.com/w/cpp/container
    - https://en.cppreference.com/w/cpp/iterator
    - https://en.cppreference.com/w/cpp/algorithm
  - SGI
    - http://www.sgi.com/tech/stl/
  - Dinkumware
    - http://www.dinkumware.com/refxcpp.html
  - Rogue Wave
    - http://www.roguewave.com/support/docs/sourcepro/stdlibug/index.html
- Формална спецификација у Це++ стандарду:
  - Appendix B

Основни модел

- Пар итератора одређује секвенцу
  - Почетак (beginning) – показује на први елемент (ако га уопште има)
  - Крај (end) – показује **иза последњег** елемента



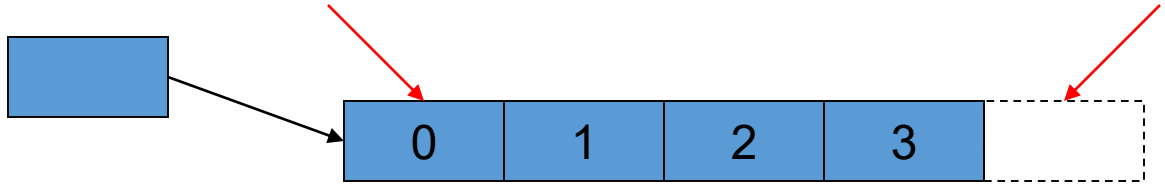
- Итератор је тип који подржава итераторске операције:
  - == (и !=) Да ли итератори показују на исти елемент? (за одређивање краја)
  - * (унарно) Добави вредност
  - ++ (префиксно и постфиксно) Пређи на следећи елемент
- Итератори могу подржавати још неке операције, нпр.:

  --, +, [ ] ...

Контејнери

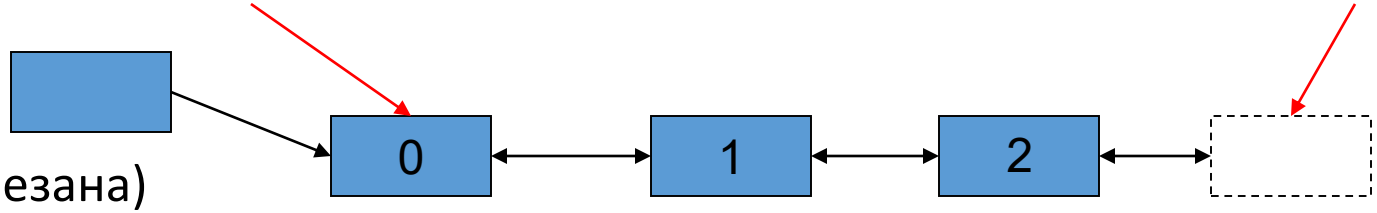
(складиште секвенцу података на различите начине)

- **vector**



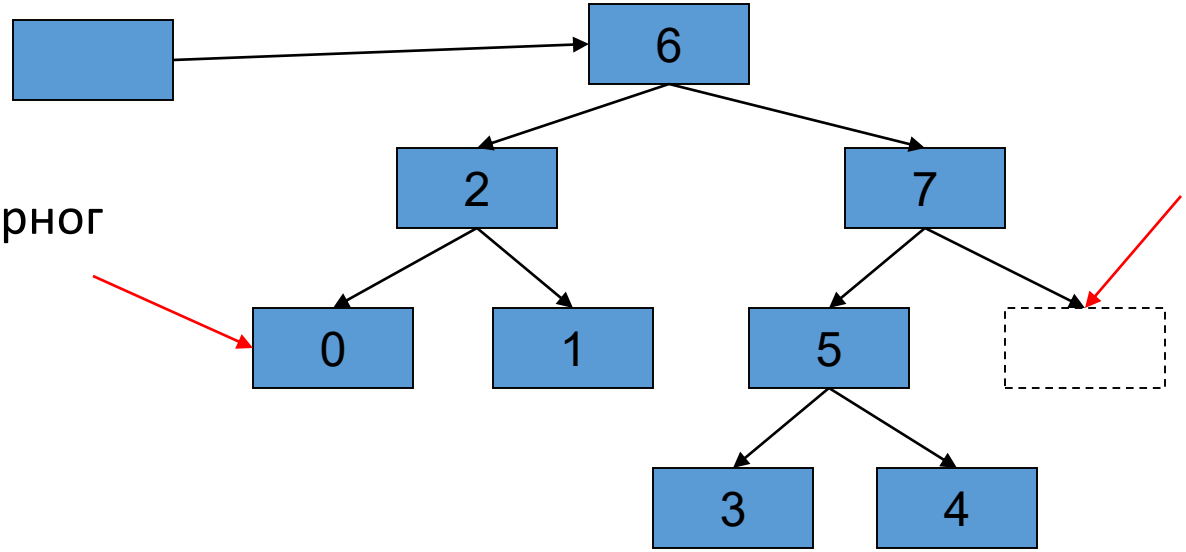
- **list**

(двоструко повезана)

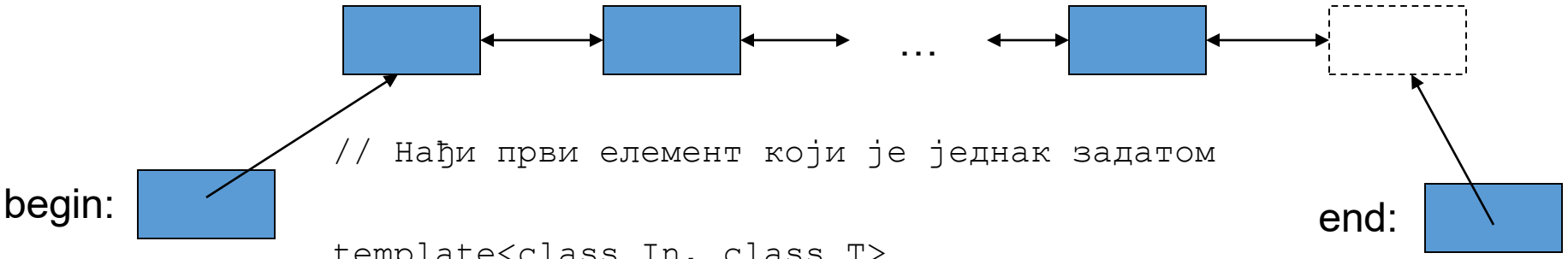


- **set**

(у облику бинарног стабла)



Пример алгоритма: find()




```
// Нађи први елемент који је једнак задатом

template<class In, class T>
In find(In first, In last, const T& val)
{
  while (first != last && *first != val) ++first;
  return first;
}

void f(vector<int>& v, int x)
{
  vector<int>::iterator p = find(v.begin(), v.end(), x);
  if (p != v.end()) { /* нашли смо x */ }
  // …
}
```

find()

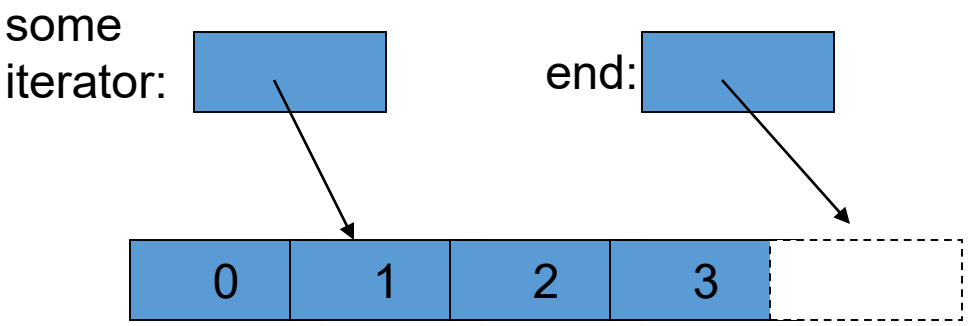
```
void f(vector<int>& v, int x)
{
  vector<int>::iterator p = find(v.begin(), v.end(), x);
  if (p != v.end()) { /* нашли смо x */ }
  // …
}

void f(list<string>& v, string x)
{
  list<string>::iterator p = find(v.begin(), v.end(), x);
  if (p != v.end()) { /* нашли смо x */ }
  // …
}

void f(set<double>& v, double x)
{
  set<double>::iterator p = find(v.begin(), v.end(), x);
  if (p != v.end()) { /* нашли смо x */ }
  // …
}
```

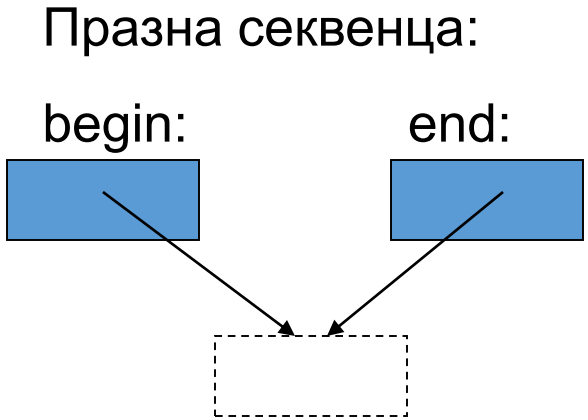
find()

```
void f(vector<int>& v, int x)
{
  auto p = find(v.begin(), v.end(), x);
  if (p != v.end()) { /* нашли смо x */ }
  // …
}

void f(list<string>& v, string x)
{
  auto p = find(v.begin(), v.end(), x);
  if (p != v.end()) { /* нашли смо x */ }
  // …
}

void f(set<double>& v, double x)
{
  auto p = find(v.begin(), v.end(), x);
  if (p != v.end()) { /* нашли смо x */ }
  // …
}
```

Алгоритми и итератори

- Итератор показује на елемент у секвенци
- Крај секвенце је отворен, тј. итератор на крају секвенце показује иза последњег елемента
  - Згодно за елегантно представљање празне секвенце
  - Крајњи итератор целог контејнера показује иза последњег елемента
    - Тај итератор се не може дереференцирати

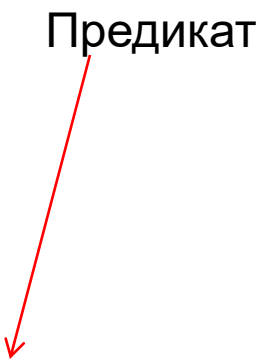




Пример алгоритма: find_if()

- Нађи први елемент који испуњава неки критеријум (предикат)
  - Предикат прима један аргумент и враћа **bool** тип

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
  while (first != last && !pred(*first)) ++first;
  return first;
}

void f(vector<int>& v)
{
  vector<int>::iterator p = find_if(v.begin(), v.end(), odd);
  if (p != v.end()) { /* нашли смо први непарни елемент */ }
  // …
}
```



Предикати

- Пример:
  - Функција (сетимо се исцртавања функција)

```
bool odd(int i) { return i % 2; }
odd(7);
```

  - Функцијски објекат (функтор)

```
struct Odd {
     bool operator()(int i) const { return i % 2; }
};
Odd odd;
odd(7);
```

Предикати

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
...
```

- Пример:
  - Функција (сетимо се исцртавања функција)

```
bool odd(int i) { return i % 2; }
odd(7);

find_if(..., ..., odd);
// шта је тип Pred?
```

  - Функцијски објекат (функтор)

```
struct Odd {
     bool operator()(int i) const { return i % 2; }
};
Odd odd;
odd(7);

find_if(..., ..., odd);
// шта је тип Pred?
```

Предикати

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
...
```

- Пример:
  - Функција (сетимо се исцртавања функција)

```
bool odd(int i) { return i % 2; }
odd(7);

find_if(..., ..., odd);
// шта је тип Pred? bool ()(int);
```

  - Функцијски објекат (функтор)

```
struct Odd {
     bool operator()(int i) const { return i % 2; }
};
Odd odd;
odd(7);

find_if(..., ..., odd);
// шта је тип Pred? Odd
```

Предикати

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
...
```

- Пример:
  - Функција (сетимо се исцртавања функција)

```
bool even(int i) { return !(i % 2); }
even(7);

find_if(..., ..., even);
// шта је тип Pred? bool ()(int);
```

  - Функцијски објекат (функтор)

```
struct Even {
    bool operator()(int i) const { return !(i % 2); }
};
Even even;
even(7);

find_if(..., ..., even);
// шта је тип Pred? Even
```

Функтори наспрам функција

- Функтори и класичне функције које се позивају на синтаксно идентичан начин ипак нису истог типа! Зато се, на пример, не могу прослеђивати истим функцијама...

```
void foo(int x);
void bar(int x);

void apply1(void (*f)(int)) {
  ... f(elem); ...
}
apply1(foo);
apply1(bar);
apply1(foA); // не може
```

```
struct A {
  void operator()(int x);
};
struct B {
  void operator()(int x);
};

A foA; B foB;

void apply2(A f) {
  ... f(elem); ...
}

apply2(foA);
apply2(foo); // не може
apply2(foB); // не може
```

Функтори наспрам функција

```
template<typename T>
void foo(T func) {
  ...
  func(15);
  ...
}

void bar1(int x);
void bar2(int x);

struct Bar3 {
  void operator()(int x);
};

struct Bar4 {
  void operator()(int x);
};

Bar4 bar4;
```

```
typedef void fType(int);
void foo(fType func) {
  ... func(15); ...
}

void foo(Bar3 func) {
  ... func(15); ...
}

void foo(Bar4 func) {
  ... func(15); ...
}

foo(bar1);
foo(bar2);
foo(Bar3());
foo(bar4);
```

```
r1 <- mem[_bar1]
//r1 <- mem[_bar2]
foo_fType:
  call r1

foo_Bar3:
  call _Bar3_f

foo_Bar4:
  call _Bar4_f
```

...али се могу користити за инстанцирање истих функцијских шаблона, јер се исто синтаксно понашају.

Инстанцирање шаблона функторима може довести до бржег кода, јер компајлер тачно зна која функција ће се применити. Са обичним функцијама се не зна да ли је то bar1 или bar2 или нешто треће, па имамо индирекцију.

Функтори (из архиве)

```
int expN_number_of_terms = 6; // супер тајни аргумент функције expN

double expN(double x)
{
      return expe(x, expN_number_of_terms);
}

for (int n = 0; n < 50; ++n)
{
      ostringstream ss;
      ss << "exp approximation; n==" << n ;
      win.set_label(ss.str().c_str());
      expN_number_of_terms = n; // супер тајни аргумент функције expN

      // наредна апроксимација:
      Function e(expN, r_min,r_max, orig, 200, x_scale,y_scale);

      win.attach(e);
      win.wait_for_button();
      win.detach(e);
}
```

Функтори (из архиве)

```
struct expN
{
  int number_of_terms;
  expN(int x) : number_of_terms(x) {}
  double operator()(double x) const { return expe(x, number_of_terms); }
};


for (int n = 0; n < 50; ++n)
{
      ostringstream ss;
      ss << "exp approximation; n==" << n ;
      win.set_label(ss.str().c_str());
      //expN_number_of_terms = n; // супер тајни аргумент функције expN

      // наредна апроксимација:
      Function e(expN(n), r_min,r_max, orig, 200, x_scale,y_scale);

      win.attach(e);
      win.wait_for_button();
      win.detach(e);
}
```

Функтори

```
struct Less_than
{
 int val;
 Less_than(int& x) : val(x) { }
 bool operator()(const int& x) const { return x < val; }
};

p=find_if(v.begin(), v.end(), Less_than(43));
p=find_if(v.begin(), v.end(), Less_than(76));
```

Функтори

```
template<class T>
struct Less_than
{
 T val;
 Less_than(T& x) : val(x) { }
 bool operator()(const T& x) const { return x < val; }
};

p=find_if(v.begin(), v.end(), Less_than(43));
p=find_if(v.begin(), v.end(), Less_than(76));
q=find_if(ls.begin(), ls.end(), Less_than("perfection"));
```

Функтори

- Да закључимо:
- Зашто функтори?
  - Без индирекције
  - Могу да имају стање

Параметризација алгоритама

```
struct Record {
  string name;
  char addr[24];
  // …
};

vector<Record> vr;
// …
sort(vr.begin(), vr.end(), Cmp_by_name());   // уреди по имену
sort(vr.begin(), vr.end(), Cmp_by_addr());   // уреди по адреси
```

Параметризација алгоритама

```
struct  Cmp_by_name {
 bool operator()(const Rec& a, const Rec& b) const
     { return a.name < b.name; }
};

struct  Cmp_by_addr {
 bool operator()(const Rec& a, const Rec& b) const
     { return 0 < strncmp(a.addr, b.addr, 24); }
};

vector<Record> vr;
// …
sort(vr.begin(), vr.end(), Cmp_by_name());

sort(vr.begin(), vr.end(), Cmp_by_addr());
```

Параметризација алгоритама
Ламбда изрази („функторски литерал“)

```
struct  Cmp_by_name {
 bool operator()(const Rec& a, const Rec& b) const
     { return a.name < b.name; }
};

struct  Cmp_by_addr {
 bool operator()(const Rec& a, const Rec& b) const
     { return 0 < strncmp(a.addr, b.addr, 24); }
};


vector<Record> vr;
// …
sort(vr.begin(), vr.end(), [] (const Rec& a, const Rec& b)
        { return a.name < b.name; } );
sort(vr.begin(), vr.end(), [] (const Rec& a, const Rec& b)
        { return 0 < strncmp(a.addr, b.addr, 24); });
```

Параметризација алгоритама
Ламбда изрази („функторски литерал“)

```
void foo(int x);
int a = 5;
int b = 6;
foo(a);
foo(b);


foo(5);
foo(6);
```

Тип литерала 5?

```
???  x = 5;
```

Параметризација алгоритама
Ламбда изрази („функторски литерал“)

```
void foo(int x);
int a = 5;
int b = 6;
foo(a);
foo(b);


foo(5);
foo(6);
```

Тип литерала 5? int

```
int  x = 5;
```

Параметризација алгоритама
Ламбда изрази („функторски литерал“)

```
void foo(int x);
int a = 5;
int b = 6;
foo(a);
foo(b);
```

Тип литерала 5? int

```
int  x = 5;
```

Тип овог литерала?

```
foo(5);
foo(6);
```

```
???  y = [] (const Rec& a, const Rec& b)
        { return a.name < b.name; };
```

Параметризација алгоритама
Ламбда изрази („функторски литерал“)

```
void foo(int x);
int a = 5;
int b = 6;
foo(a);
foo(b);
```

Тип литерала 5? int

```
int  x = 5;
```

Тип овог литерала?

```
foo(5);
foo(6);
```

```
auto y = [] (const Rec& a, const Rec& b)
        { return a.name < b.name; };
```

Гледаћете у следећој сезони... (у предмету Паралелно програмирање - TBB)

```
class ApplyFoo {
    vector<float>& m_a;
public:
    void operator()(const blocked_range<size_t>& r) const {
        for (size_t i = r.begin(); i != r.end(); ++i)
            Foo(m_a[i]);
    }
    ApplyFoo(vector<float>& a) : m_a(a) {}
};

vector<float> data;

parallel_for(blocked_range<size_t>(0, n), ApplyFoo(data));
```

Гледаћете у следећој сезони... (у предмету Паралелно програмирање - TBB)

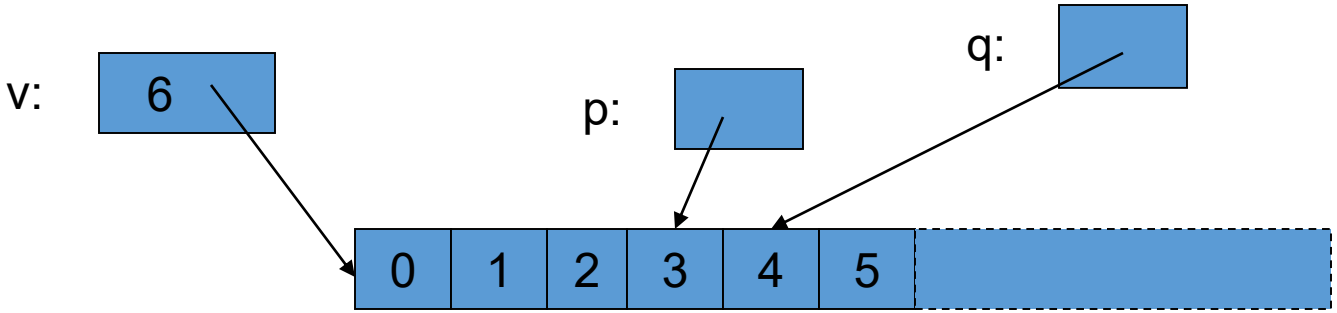
```
class ApplyFoo {
    vector<float>& m_a;
public:
    void operator()(const blocked_range<size_t>& r) const {
        for (size_t i = r.begin(); i != r.end(); ++i)
            Foo(m_a[i]);
    }
    ApplyFoo(vector<float>& a) : m_a(a) {}
};

vector<float> data;

parallel_for(blocked_range<size_t>(0, n), [data](const blocked_range<size_t>& r) {
    for (size_t i = r.begin(); i != r.end(); ++i)
        Foo(data[i]);
});
```

vector

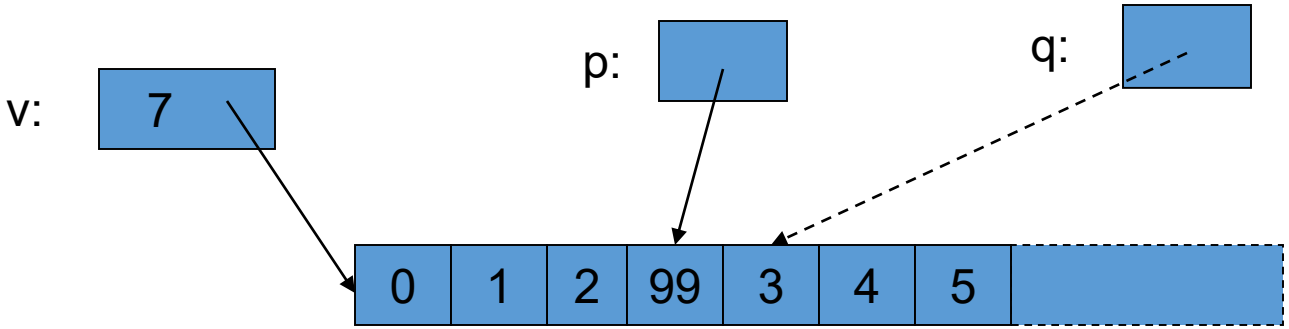
```
template<class T> class vector {
 T* elements;
 // …
 using value_type = T;
 using iterator = ???;
 using const_iterator = ???;

 iterator begin();
 const_iterator begin() const;
 iterator end();
 const_iterator end() const;

 iterator erase(iterator p);
 iterator insert(iterator p, const value_type& v);
};
```

insert() код вектора

```
vector<int>::iterator p = v.begin(); ++p; ++p; ++p;
vector<int>::iterator q = p; ++q;
```

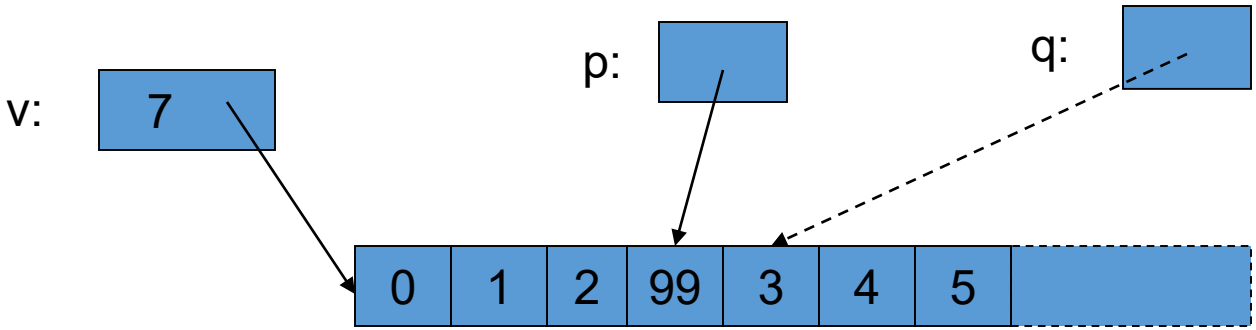



```
p = v.insert(p, 99);
```

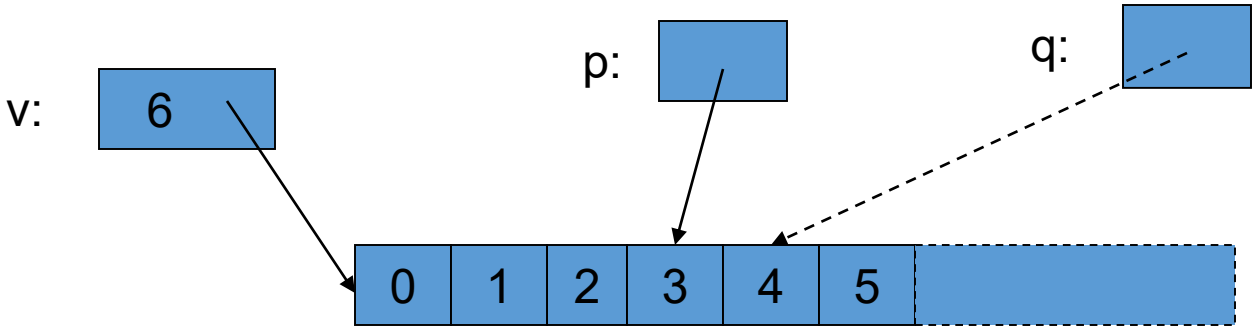


- Напомена: q није валидно након **insert()**
- Напомена: Неки елементи су се померили (могли су се сви померити)

erase() код вектора

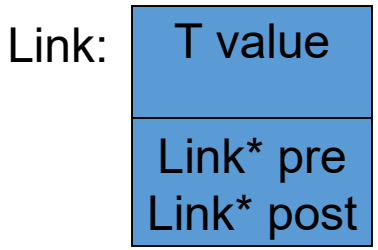



```
p = v.erase(p);
```



- Напомена: претходно постављени итератори нису валидни након **erase()**
- Напомена: Неки елементи су се померили (могли су се сви померити)

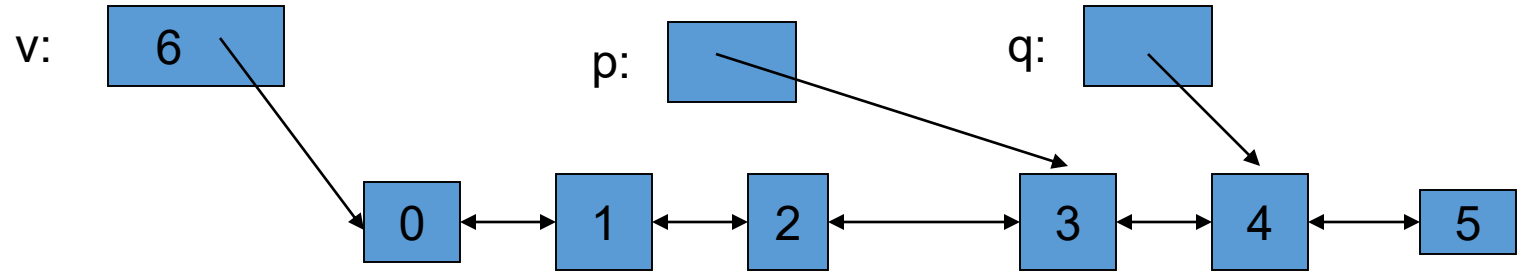
list



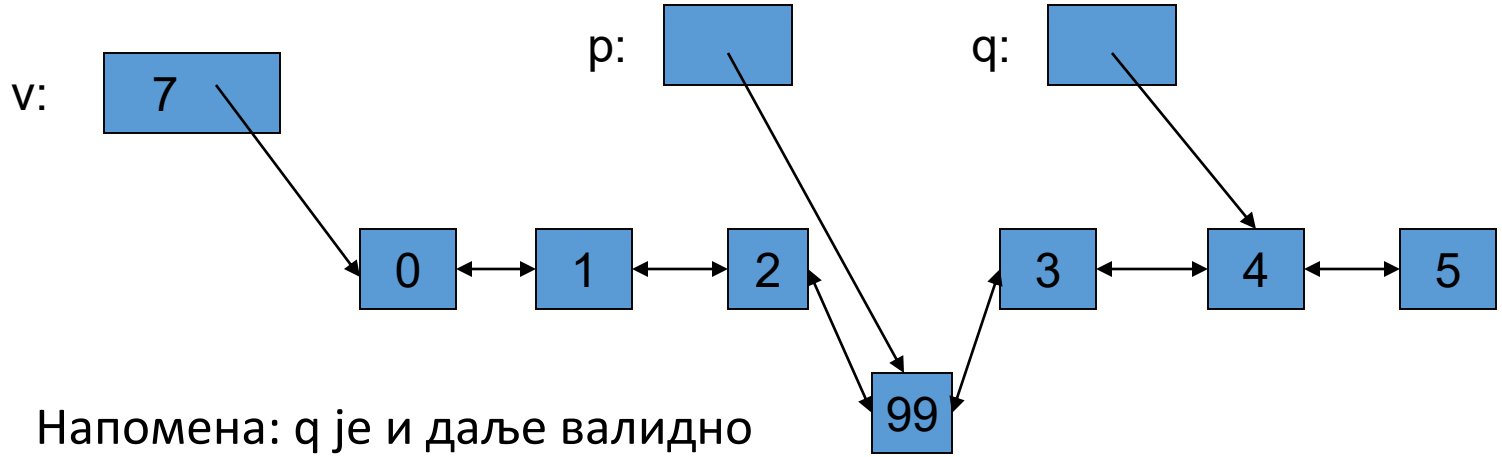

```
template<class T> class list {
 Link* elements;
 // …
 using value_type = T;
 using iterator = ???;
 using const_iterator = ???;

 iterator begin();
 const_iterator begin() const;
 iterator end();
 const_iterator end() const;

 iterator erase(iterator p);
 iterator insert(iterator p, const value_type& v);
};
```

insert() код листе

```
list<int>::iterator p = v.begin(); ++p; ++p; ++p;
list<int>::iterator q = p; ++q;
```

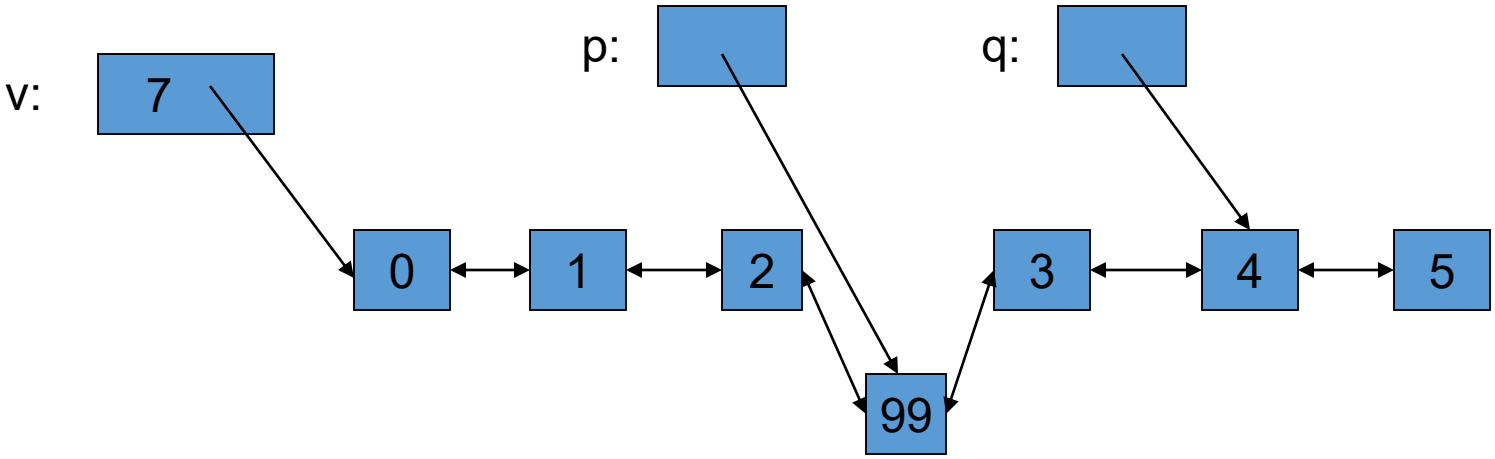



```
p = v.insert(p, 99);
```

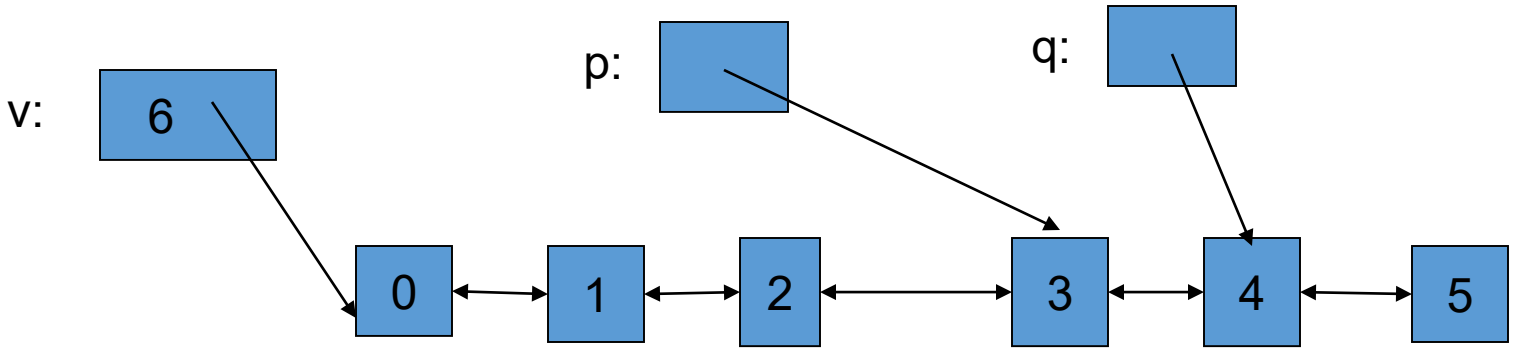


- Напомена: q је и даље валидно
- Напомена: Нема померања елемената

erase() код листе




```
p = v.erase(p);
```



- Напомена: q је и даље валидно
- Напомена: Нема померања елемената

Начини проласка кроз вектор

```
for (int i = 0; i < v.size(); ++i) // зашто баш int?
  …    // v[i]

for (vector<T>::size_type i = 0; i < v.size(); ++i) // дуже, али увек исправно
  …    // v[i]

for (vector<T>::iterator p = v.begin(); p != v.end(); ++p)
  …    // *p
```

Начини проласка кроз вектор

```
for (int i = 0; i < v.size(); ++i) // зашто баш int?
  …    // v[i]

for (vector<T>::size_type i = 0; i < v.size(); ++i) // дуже, али увек исправно
  …    // v[i]

for (vector<T>::iterator p = v.begin(); p != v.end(); ++p)
  …    // *p

for (vector<T>::value_type x : v) // нпр.: for(int x : v), ако је vector<int>
  …    // x

for (auto x : v) // или for (auto& x : v)
  …    // x
```

- Оваква фор петља је згодна да се користи:
  - када хоћемо да прођемо кроз цео контејнер, од почетка до краја
  - када нам није важна позиција елемента
  - када нам је довољно да обрађујемо по један елемент

Вектор наспрам листе

- Подразумевано користите **vector**
- Ако желите да померате елементе, користите **list**

- То нису једини контејнери!

Корисна стандардна заглавља

- **<iostream>** I/O streams, cout, cin, …
- **<fstream>** file streams
- **<algorithm>** sort, copy, …
- **<numeric>** accumulate, inner_product, …
- **<functional>** function objects
- **<string>**
- **<vector>**
- **<map>**
- **<unordered_map>** hash table
- **<list>**
- **<set>**