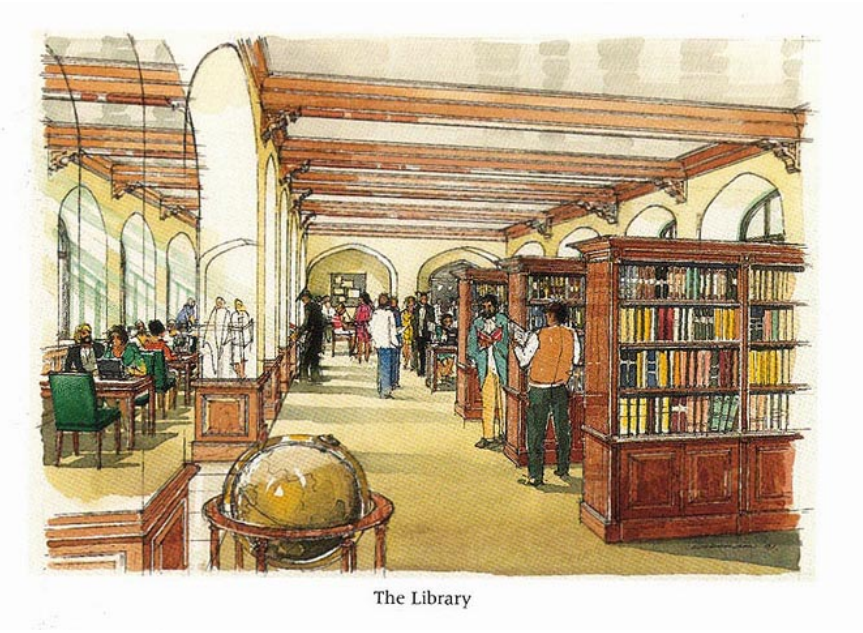
ספריית התבניות התיקנית (STL)

אוהד ברזילי

אוניברסיטת תל אביב



The Library

מיכלים ואיטרטורים

מיכלים (containers)

- מיכל (container) הוא כינוי למבנה נתונים המכיל עצמים אחרים
- בספרייה הסטנדרטית של C++ קיימים מספר טיפוסי מיכלים
- למיכלים השונים פעולות משותפות וכן תכונות פרטיות המייחדות רק אותם
- מיכלים שונים מותאמים לאופי עבודה שונה. לדוגמא:
  - הוספה\הסרה תכופה של איברים מלב המיכל יעילה יותר ברשימה מקושרת
  - גישה אקראית לאברים לפי מיקומם יעילה יותר במערך (vector)

עלות הפעולות על מיכלים

Standard Container Operations					
	[]	List Operations	Front Operations	Back (Stack) Operations	Iterators
	§16.3.3 §17.4.1.3	§16.3.6 §20.3.9	§17.2.2.2 §20.3.9	§16.3.5 §20.3.12	§19.2.1
vector	const	O(n)+		const+	Ran
list		const	const	const	Bi
deque	const	O(n)	const	const	Ran
stack				const+	
queue			const	const+	
priority_queue			O(log(n))	O(log(n))	
map	O(log(n))	O(log(n))+			Bi
multimap		O(log(n))+			Bi
set		O(log(n))+			Bi
multiset		O(log(n))+			Bi
string	const	O(n)+	O(n)+	const+	Ran
array	const				Ran
valarray	const				Ran
bitset	const				

האיטרטור (סודר? אצן? סורק?)

- איטרטור הוא הפשטה של מעבר סדרתי על מבנה נתונים כלשהו
- כדי לבצע פעולה ישירה על מבנה נתונים, יש לדעת כיצד הוא מיוצג בזכרון
- גישה בעזרת איטרטור למבנה הנתונים מאפשרת למשתמש לבצע מגוון רחב של פעולות ללא צורך להכיר את המבנה הפנימי של המיכל

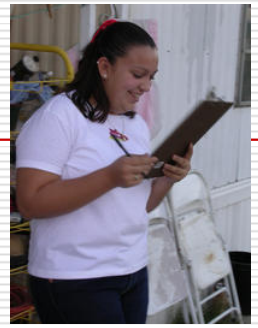
חיפוש במערך (מצביעים)

```
char arr[7] = "abcdef";

char *find_d_ptr(char *arr)
{
    for (char *iter = arr ; *iter ; iter++)
    {
        if (*iter == 'd')
            return iter;
    }
    return iter;
}
```

חיפוש במערך (אינדקסים)

```
char arr[7] = "abcdef";

char *find_d_ind(char *arr)
{
   for (int i=0 ; i < 7 ; i++)
   {
      if (arr[i] == 'd')
            return &arr[i];
   }
   return &arr[i];
}
```

חיפוש ברשימה



```
struct node {
    char val;
    node *next;
};

node *find_d_lst(node *list)
{
    for (node *iter = list ; iter->val; iter = iter->next)
    {
        if (iter->val == 'd')
            return iter;
    }
    return iter;
}
```

הכרות אינטימית עם מבנה הנתונים

- 3 הדוגמאות הקודמות חושפות ידע מוקדם שיש לכותבת הפונקציה על מבנה הנתונים:
  - היא יודעת איפה הוא מתחיל ואיפה הוא נגמר
  - היא מכירה את מבנה הטיפוס שבעזרתו ניתן לקבל את המידע השמור במצביע
  - היא יודעת איך לעבור מאיבר לאיבר שאחריו

האיטרטור

- טיפוס יקרא איטרטור אם ניתן לבצע עליו 3 פעולות:
  - השוואה בין שני עצמים מאותו טיפוס (==)
  - קידום (++)
  - פעולת * או -> (dereference)
- בעזרת כל מחלקה אשר ממשת את 3 הפעולות ניתן לבצע מגוון רחב של פעולות על מכלים
- לדוגמא: מצביע ל-T הוא איטרטור כי הוא עונה על 3 הקריטריונים עבור מערך של טיפוס T
- נראה איך ניתן להכליל את פעולת החיפוש במכל

אלגוריתם כללי לחיפוש במכל

```
for (iterator iter = starting_point ;
    iter != end_point ; iter.inspect_next_element)
{
    if (iter.getElement == 'd')
        return iter;
}
return end_point;
```

- הקוד האדום הוא שינויים שיש להכניס לקוד החיפוש במערך עם מצביעים
- ע"י העמסת האופרטורים * ו- ++ אפשר להפשט קצת את הקוד

חיפוש במכל ע"י איטרטור
עם העמסת אופרטורים

```
for (iterator iter = starting_point ;
    iter != end_point ; iter++)
{
    if (*iter == 'd')
        return iter;
}
return end_point;
```

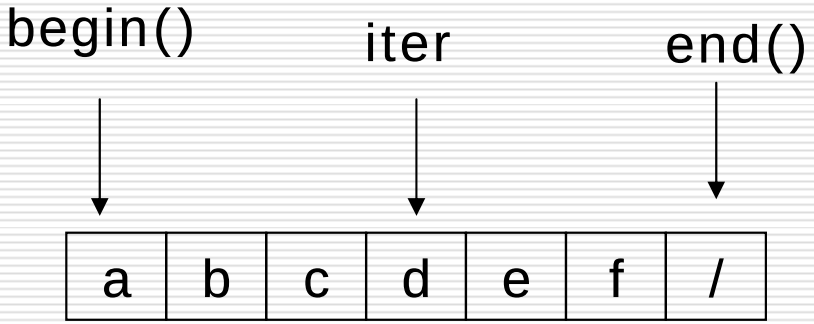
- ראשית, אנו מעוניינים לא לדעת את טיפוס האיטרטור וכן לא לדעת היכן מתחיל ונגמר מבנה הנתונים
- לשם כך נוסיף למחלקה השומרת את מבנה הנתונים עצמו (מערך, רשימה וכו') את הניהול של האיטרטורים שלה

חיפוש במערך עם מצביעים
- לפני -

```
char arr[7] = "abcdef";

char *find_d_ptr(char *arr)
{
    for (char *iter = arr ; *iter ; iter++)
    {
        if (*iter == 'd')
            return iter;
    }
    return iter;
}
```

חיפוש במערך עם מצביעים
- אחרי -

```
class Vector {
    char *arr;
public:
    typedef char * iterator;
    iterator begin() {return arr;}
    iterator end() {return &arr[6];}
};
```




```
Vector::iterator find_d_Vec(Vector& v)
{
    for (Vector::iterator iter = v.begin() ;
                                iter != v.end() ; iter++)
    {
        if (*iter == 'd')
                return iter;
    }
    return iter;
}
```

חיפוש ברשימה
- לפני -

```
struct node {
    char val;
    node *next;
};

node *find_d_lst(node *list)
{
    for (node *iter = list ; iter->val; iter = iter->next)
    {
        if (iter->val == 'd')
            return iter;
    }
    return iter;
}
```

חיפוש ברשימה
- אחרי -

```
class List{
    node *head;
    node *tail;

public:
    class iterator {
        node *cur;
    public:
        iterator(node *pos) : cur(pos){}
        iterator operator++(){ cur = cur->next; return *this; }
        char operator*(){ return cur->val; }
        bool operator==(iterator other) { return cur == other.cur; }
        bool operator!=(iterator other) { return cur != other.cur; }
    };

    iterator begin() {return iterator(head);}
    iterator end() {return iterator(tail->next);}
};
```

חיפוש ברשימה
- אחרי -

```
List::iterator find_d_List(List& l)
{
   for (List::iterator iter = l.begin() ;
                       iter != l.end() ; iter++)
   {
      if (*iter == 'd')
         return iter;
   }
   return iter;
}
```

הפונקציה find לא תלויה במיכל !

```
template<class C>
C::iterator find_d(C& c)
{
   for (C::iterator iter = c.begin() ;
                        iter != c.end() ; iter++)
   {
      if (*iter == 'd')
         return iter;
   }
   return iter;
}
```

רב צורתיות אד-הוק

- כדי שפונקציית התבנית find_d תעבוד, על המחלקה C לממש ממשק של מכל
- כלומר להגדיר מתודות ושדות כגון: begin(), end(), iterator אשר המשתמש מצפה למצוא בכל מיכל
- שימוש במחלקה ללא ידיעת סוגה האמיתי, אלא מתוך הכרת הממשק שלה בלבד נקרא **רב צורתיות** (polymorphism)
- במהלך הקורס נראה דוגמא נוספת לרב צורתיות המושגת ע"י מנגנון הירושה

סוגי איטרטורים

Iterator Operations and Categories					
Category: Abbreviation:	output *Out*	input *In*	forward *For*	bidirectional *Bi*	random-access *Ran*
Read:		`=*p`	`=*p`	`=*p`	`=*p`
Access:		`->`	`->`	`->`	`-> []`
Write:	`*p=`		`*p=`	`*p=`	`*p=`
Iteration:	`++`	`++`	`++`	`++ --`	`++ -- + - += -=`
Comparison:		`== !=`	`== !=`	`== !=`	`== != < > >= <=`

- כל מכל מגדיר איטרטורים רגילים ואיטרטורים קבועים const_iterator
- לא כל סוגי האיטרטורים קיימים בכל מכל (משיקולי יעילות)

מיכלים

Containers

המערך מהדור הישן

```
struct Entry{
    string name;
    int number;
};

Entry phone_book[1000];

void print_entry(int i) // simple use
{
    cout << phone_book[i].name << ´ ´ <<
       phone_book[i].number << ´\ n ´;
}
```

המערך של הדור החדש vector<T>

```
vector<Entry> phone_book(1000);

// simple use, exactly as for array
void print_entry(int i)
{
    cout << phone_book[i].name << ´ ´
              << phone_book [i].number << ´\ n ´;
}

void add_entries(int n) // increase size by n
{
    phone_book.resize(phone_book.size()+n );
}
```

רשימה מקושרת

- אם הוספה והסרה של כניסות לספר הטלפונים הן שכיחות נשקול מעבר לרשימה:

```
list<Entry> phone_book;
```

- נדפיס את ספר הטלפונים תוך שימוש באיטרטור:

```
void print_entry(const string& s)
{
    typedef list<Entry>::const_iterator LI;
    for(LI i = phone_book.begin();
                    i != phone_book.end(); ++i )
    {
        Entry& e = *i ; // reference used as shorthand
        if (s == e.name)
            cout << e.name << ´ ´ << e.number << ´\ n ´;
    }
}
```

הוספה של אברים לרשימה

```
void add_entry(Entry& e , list<Entry>::iterator i)
{
   phone_book.push_front(e); // add at beginning
   phone_book.push_back(e); // add at end
   phone_book.insert(i,e); // add before the
                           // element 'i' refers to
}
```

מיפוי
map<key,value>

- מיפוי הוא מבנה נתונים אשר מאפשר שליפה יעילה של מידע (value) על פי מפתח (key)
- טיפוסי המידע והמפתח הם כלליים, ונקבעים בזמן יצירת המיפוי

- ספר הטלפונים שלנו הוא מיפוי – לכל שם יש מספר טלפון:

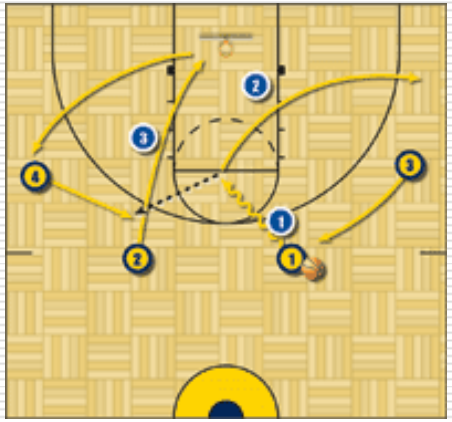
```
map<string , int> phone_book;
```

- ניתן לחפש כניסות בספר הטלפונים כך:

```
void print_entry(const string& s)
{
    if (int i = phone_book[s])
        cout << s << ´ ´ << i << ´\n´;
}
```

ספריית האלגוריתמים הסטנדרטית

<algorithm>



אלגוריתמים כלליים

- עצמים לא רק "מונחים" בתוך מיכלים, אנחנו משחקים איתם
- לפעולות סטנדרטיות רבות שנרצה לבצע על עצמים בתוך מיכל יש כבר פונקציות ספרייה שממשות אותן ביעילות
- לדוגמא: חיפוש, החלפה, מיון, מיזוג, העתקה, סינון ועוד...
- הפונקציות מופיעות בכותרת <algorithm>

מיון והעתקה

- הפונקציה הבאה ממיינת וקטור ומעתיקה אותו, תוך הסרת כפילויות, לרשימה מקושרת:

```
void f(vector<Entry>& ve, list<Entry>& le)
{
    sort(ve.begin(), ve.end());
    unique_copy(ve.begin(), ve.end(), le.begin());
}
```

- העתקה לתוך מיקום הנתון ע"י איטרטור דורסת את המידע הנמצא שם

איטרטור הכנסה

- כדי לשרשר את האברים למיקום המבוקש יש להשתמש באיטרטור הכנסה (insert iterator)

```
void f(vector<Entry>& ve, list<Entry>& le)
{
    sort(ve.begin(), ve.end());
    unique_copy(ve.begin(), ve.end(), back_inserter(le));
}
```

חיפוש במיכל

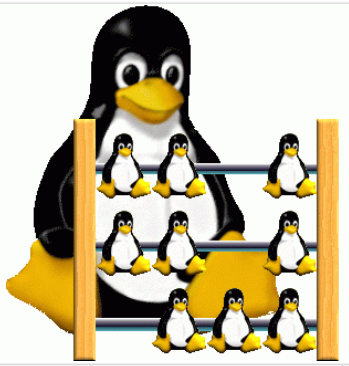
- הפונקציה הבאה סופרת את מספר הפעמים שמופיע התו c במחרוזת s

```
int count(const string& s , char c)
{
   string::const_iterator i =
                     find(s.begin(), s.end(), c );
   int n = 0 ;
   while(i != s.end()) {
      ++n;
      i = find(i+1 ,s.end(), c);
   }
   return n ;
}
```

מנייה

- ניתן להכליל את הפונקציה המונה שתעבוד על כל מיכל וערך מהטיפוס שנמצא במיכל, ולקבל:

```
template<class C , class T> int count(const C& v , T val)
{
    typename C::const_iterator i =
                                find(v.begin(), v.end(), val);
    int n = 0;
    while(i != v.end()) {
        ++n;
        ++i; // skip past the element we just found
        i = find(i ,v.end(), val);
    }
    return n ;
}
```



מנייה

- וכך נקבל:

```
void f(list<complex>& lc, vector<string>& vc, string s)
{
    int i1 = count(lc ,complex(1 ,3));
    int i2 = count(vc,"Chrysippus");
    int i3 = count(s ,´x´);
}
```

- count של הספרייה הסטנדרטית משתמשת ברצפים ואיטרטורים במקום במבני הנתונים עצמם:

```
void f(list<complex>& lc , vector<string>& vs , string s)
{
    int i1 = count(lc.begin(), lc.end(), complex(1,3));
    int i2 = count(vs.begin(), vs.end(), "Diogenes");
    int i3 = count(s.begin(), s.end(), ´x´);
}
```

חיפוש ומנייה מותנים

```
bool gt_42(const pair<const string ,int>& r)
{
    return r.second > 42;
}

void f(map<string ,int>& m)
{
    typedef map<string ,int>::const_iterator MI;
    MI i = find_if(m.begin(), m.end(), gt_42);
    int c42 = count_if(m.begin(), m.end(), gt_42);
    // ...
}
```

for_each

```
map<string ,int> histogram;

void print(const pair<const string ,int>& r)
{
    cout << r.first << ´ ´ << r.second << ´\n´;
}

int main()
{
    for_each(histogram.begin(), histogram.end(), print);
}
```

for_each עם מתודה של האיבר

```
void draw(Shape *p)
{
    p -> draw();
}

void f(list<Shape *>& sh)
{
    for_each(sh.begin(), sh.end(), draw);
}
```

- בספרייה הסטנדרטית יש תבנית מיוחדת לצורך כך mem_fun:

```
for_each(sh.begin(), sh.end(), mem_fun(&Shape::draw));
```

אלגוריתמים שאינם משנים את מבנה הנתונים

Nonmodifying Sequence Operations (§18.5) <algorithm>	
for_each()	Do operation for each element in a sequence.
find()	Find first occurrence of a value in a sequence.
find_if()	Find first match of a predicate in a sequence.
find_first_of()	Find a value from one sequence in another.
adjacent_find()	Find an adjacent pair of values.
count()	Count occurrences of a value in a sequence.
count_if()	Count matches of a predicate in a sequence.
mismatch()	Find the first elements for which two sequences differ.
equal()	True if the elements of two sequences are pairwise equal.
search()	Find the first occurrence of a sequence as a subsequence.
find_end()	Find the last occurrence of a sequence as a subsequence.
search_n()	Find the *n*th occurrence of a value in a sequence.

אלגוריתמים המשנים את מבנה הנתונים

Modifying Sequence Operations (§18.6) <algorithm>	
transform()	Apply an operation to every element in a sequence.
copy()	Copy a sequence starting with its first element.
copy_backward()	Copy a sequence starting with its last element.
swap()	Swap two elements.
iter_swap()	Swap two elements pointed to by iterators.
swap_ranges()	Swap elements of two sequences.
replace()	Replace elements with a given value.
replace_if()	Replace elements matching a predicate.
replace_copy()	Copy sequence replacing elements with a given value.
replace_copy_if()	Copy sequence replacing elements matching a predicate.
fill()	Replace every element with a given value.
fill_n()	Replace first *n* elements with a given value.
generate()	Replace every element with the result of an operation.
generate_n()	Replace first *n* elements with the result of an operation.
remove()	Remove elements with a given value.
remove_if()	Remove elements matching a predicate.

עוד אלגוריתמים המשנים את מבנה הנתונים

Modifying Sequence Operations (continued) (§18.6) <algorithm>	
remove_copy()	Copy a sequence removing elements with a given value.
remove_copy_if()	Copy a sequence removing elements matching a predicate.
unique()	Remove equal adjacent elements.
unique_copy()	Copy a sequence removing equal adjacent elements.
reverse()	Reverse the order of elements.
reverse_copy()	Copy a sequence into reverse order.
rotate()	Rotate elements.
rotate_copy()	Copy a sequence into a rotated sequence.
random_shuffle()	Move elements into a uniform distribution.

אלגוריתמים ממיינים

Sorted Sequences (§18.7) <algorithm>	
sort()	Sort with good average efficiency.
stable_sort()	Sort maintaining order of equal elements.
partial_sort()	Get the first part of sequence into order.
partial_sort_copy()	Copy getting the first part of output into order.
nth_element()	Put the nth element in its proper place.
lower_bound()	Find the first occurrence of a value.
upper_bound()	Find the first element larger than a value.
equal_range()	Find a subsequence with a given value.
binary_search()	Is a given value in a sorted sequence?
merge()	Merge two sorted sequences.
inplace_merge()	Merge two consecutive sorted subsequences.
partition()	Place elements matching a predicate first.
stable_partition()	Place elements matching a predicate first, preserving relative order.

אלגורתמים לקבוצות ולערמות

Set Algorithms (§18.7.5) <algorithm>	
includes()	True if a sequence is a subsequence of another.
set_union()	Construct a sorted union.
set_intersection()	Construct a sorted intersection.
set_difference()	Construct a sorted sequence of elements in the first but not the second sequence.
set_symmetric_difference()	Construct a sorted sequence of elements in one but not both sequences.

Heap Operations (§18.8) <algorithm>	
make_heap()	Make sequence ready to be used as a heap.
push_heap()	Add element to heap.
pop_heap()	Remove element from heap.
sort_heap()	Sort the heap.

מיכל לדוגמא - וקטור

vector

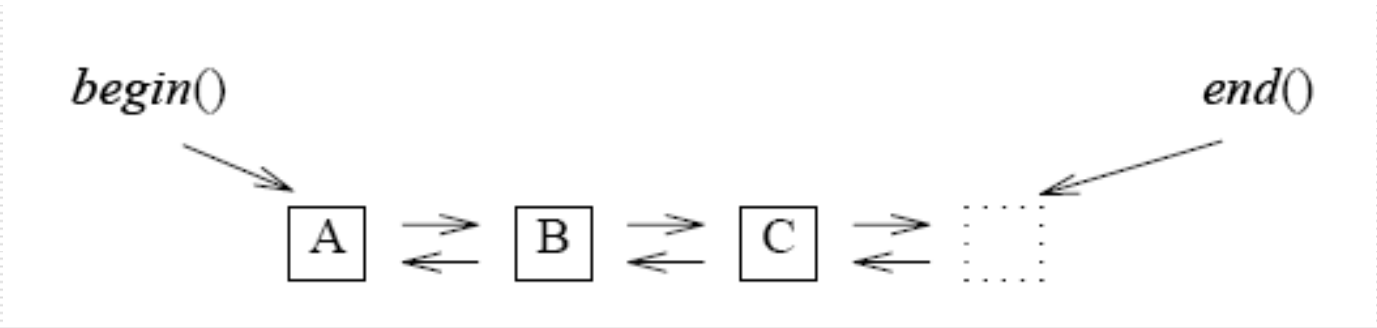
טיפוסים

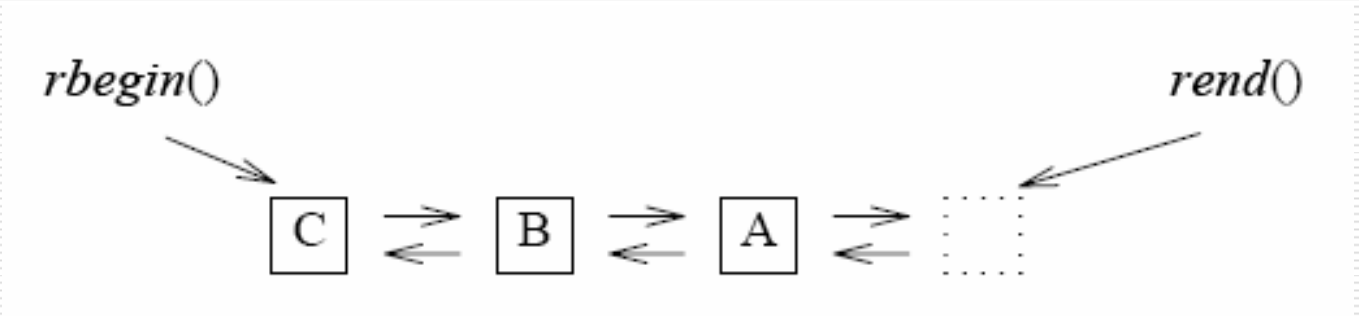
```
template<class T , class A = allocator<T>>class std::vector {
public:
    //types:
    typedef T value_type ; // type of element
    typedef A allocator_type ; // type of memory manager
    typedef typename A::size_type size_type;
    typedef typename A::difference_type difference_type ;
    typedef implementation_dependent1 iterator ; // T*
    typedef implementation_dependent2 const_iterator; // const T*
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator>
                                              const_reverse_iterator;
    typedef typename A::pointer pointer; // pointer to element
    typedef typename A::const_pointer const_pointer;
    typedef typename A::reference reference; // reference to element
    typedef typename A::const_reference const_reference;
// ...
};
```

איטרטורים

```
template <class T , class A = allocator<T>> class vector
{
public:
    // ...
    // iterators:
    iterator begin();          // points to first element
    const_iterator begin() const;
    iterator end();            // points to one past last element
    const_iterator end() const;
    reverse_iterator rbegin();      // points to first element of
                                    // reverse sequence
    const_reverse_iterator rbegin() const;
    reverse_iteratorr end();        // points to one past last
                                    // element of reverse sequence
    const_reverse_iterator rend() const ;
    // ...
};
```

איטרטור הפוך (reverse_iterator)





איטרטור הפוך (reverse_iterator)

```
template<class C>
typename C::iterator find_last(const C& c,
                               typename C::value_type v)
{
    return find_first(c.rbegin(), c.rend(),v).base();
}
```

גישה לאברים

```
template <class T , class A = allocator<T>> class vector
{
public:
    // ...
    // element access:
    reference operator[](size_type n); // unchecked access
    const_reference operator[](size_type n ) const;
    reference at(size_type n);          // checked access
    const_reference at(size_type n) const;
    reference front();                  // first element
    const_reference front() const;
    reference back();                   // last element
    const_reference back() const;
    // ...
};
```

גישה בטוחה מול גישה מהירה

```
void f(vector<int>& v , int i1 , int i2)
{
   try {
      for(int i = 0 ; i < v.size(); i ++) {
            // range already checked: use unchecked
            // v[i] here
      }

      v.at(i1) = v.at(i2); // check range on access
      // ...
   }
   catch(out_of_range) {
      // oops: outofrange error
   }
}
```

בנאים

```
template <class T , class A = allocator<T>> class vector
{
public:
    // ...
    // constructors, etc.:
    explicit vector(const A& =A ());
    explicit vector(size_type n, const T& val = T(),
                                        const A& =A()); // n copies of val
    template<class In> // In must be an input iterator
    vector(In first , In last , const A& = A()); // copy from [first:last[

    vector(const vector& x);
    ~vector();
    vector& operator=(const vector& x);

    template<class In> // In must be an input iterator
    void assign(In first , In last); // copy from [first:last[

    void assign(size_type n , const T& val); // n copies of val
    // ...
};
```

בניית וקטור ע"י שכפול טיפוס

```
class Num { // infinite precision
public:
    Num(long);
    // no default constructor
    // ...
};

vector<Num> v1(1000); // error: no default Num
vector<Num> v2(1000 ,Num(0)); // ok
```

המתודה assign

- מאפשרת השמה עם פרמטרים מרובים (ממש כמו בבנייה)

```
class Book { /* ... */ };

void f(vector <Num>& vn , vector<char>& vc ,
                          vector<Book>& vb , list<Book>& lb)
{
    vn.assign(10 ,Num(0)); // assign vector of 10 copies of
                           // Num(0) to vn. vn.size()==10
    char s[] = "literal";
    vc.assign(s ,&s[sizeof(s) - 1]); // assign "literal" to vc
    vb.assign(lb.begin(), lb.end()); // assign list elements
    // ...
}
```

פעולות מחסנית

- פעולות מחסנית הן פעולות המשפיעות על זנב הוקטור

```
template <class T , class A = allocator<T>> class vector
{
public:
    // ...
    // stack operations:
    void push_back(const T& x); // add to end
    void pop_back(); // remove last element
    // ...
};
```

פעולות מחסנית

- הוספה והסרה של אברים מגדילה ומקטינה את גודל (size()) הוקטור:

```
void f(vector<char>& s)
{
    s.push_back(´a´);
    s.push_back(´b´);
    s.push_back(‘c´);
    s.pop_back();
    assert(s[s.size() - 1] == ‘b’);
    s.pop_back();
    assert(s.back() == ‘a’);
}
```

פעולות רשימה

```
template <class T , class A = allocator<T>> class vector
{
public:
     // ...
     // list operations:
     iterator insert(iterator pos , const T& x); // add x before 'pos'
     void insert(iterator pos , size_type n , const T& x );

     template<class In> // In must be an input iterator (§19.2.1)
     void insert(iterator pos, In first, In last); // insert elements from sequence

     iterator erase(iterator pos); // remove element at pos
     iterator erase(iterator first , iterator last); // erase sequence

     void clear();
     // ...
};
```

פעולות רשימה - דוגמא

- נבנה רשימה של פירות:

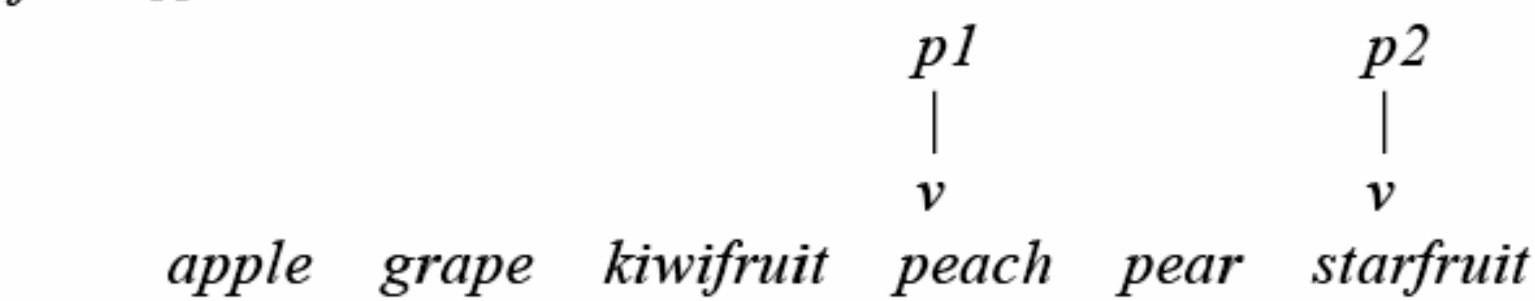
```
vector<string> fruit;
fruit.push_back("peach");
fruit.push_back("apple");
fruit.push_back("kiwifruit");
fruit.push_back("pear");
fruit.push_back("starfruit");
fruit.push_back("grape");
```

- נסיר ממנה את כל הפרות שמתחילים באות 'p' :

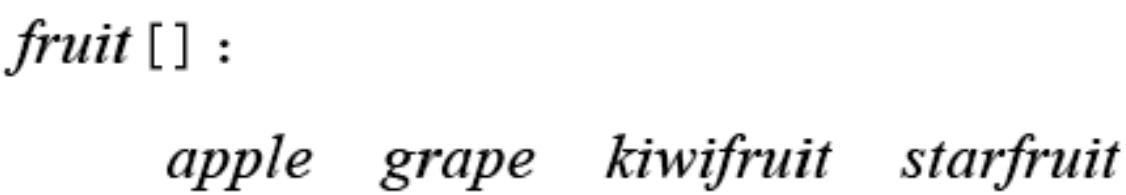
```
sort(fruit.begin(), fruit.end());
vector<string>::iterator p1 =
                find_if(fruit.begin(), fruit.end(), initial('p'));
vector<string>::iterator p2 =
                find_if(p1 ,fruit.end(), initial_not('p'));
fruit.erase(p1 ,p2);
```

פעולות רשימה - דוגמא

- לפני:



- אחרי:



איטרטור הפוך

- ניתן היה לממש את אותו רעיון עם איטרטור הפוך:

```
vector<string>::iterator p1 =
   find_if(fruit.begin(), fruit.end(), initial('p'));

vector<string>::reverse_iterator p2 =
   find_if(fruit.rbegin(), fruit.rend(),initial('p'));

fruit.erase(p1 ,p2+1); // oops!: type error
fruit.erase(p1 ,p2.base()); // extract iterator from
                            // reverse_iterator
```

מחיקה והוספה של אברים בודדים

fruit[]:

apple *kiwifruit*

- נמחוק את "starfruit" ו- "grape":

```
fruit.erase(find(fruit.begin(), fruit.end(), "starfruit"));
fruit.erase(fruit.begin()+1);
```

- ונוסיף את "cherry" ו- "cranberry":

```
fruit.insert(fruit.begin()+1 ,"cherry");
fruit.insert(fruit.end(), "cranberry"); //same as f.push_back
```

fruit[]:

apple *cherry* *kiwifruit* *cranberry*

הוספה של רצפים

- נוסיף פירות חמוצים למשפחת הפירות:

```
fruit.insert(fruit.begin()+2 ,citrus.begin(), citrus.end());
```

fruit [ ] :

apple *cherry* *kiwifruit* *cranberry*

citrus [ ] :

lemon *grapefruit* *orange* *lime*

- ונקבל:

fruit [ ] :

apple *cherry* *lemon* *grapefruit* *orange* *lime* *kiwifruit* *cranberry*

גישה לאברים
עשה ואל תעשה


```
template<class C> void f(C& c)
{
    c.erase(c.begin()+7 ); // ok
    c.erase(&c[7]) ; // not general

    c.erase(c+7);
    // error: adding 7 to a container makes no sense

    c.erase(c.back());
    // error: c.back() is a reference, not an iterator

    c.erase(c.end() - 2); // ok (second to last element)

    c.erase(c.rbegin()+2); // error: vector::reverse_iterator and
                           // vector::iterator are different types

    c.erase((c.rbegin()+2).base()); // obscure, but ok

}
```

גודל וקיבולת


```
template <class T , class A = allocator<T>> class vector
{
public:
    // ...
    // capacity:
    size_type size() const ; // number of elements
    bool empty() const { return size()==0 ; }
    size_type max_size() const ; // size of the largest possible vector

    void resize(size_type sz , T val = T()); // added elements initialized by val

    size_type capacity() const ; // size of the memory (in number of
    // elements) allocated

    void reserve(size_type n); // make room for a total of n elements;
    // don't initialize. throw a length_error if n>max_size()

    // ...
};
```

גודל וקיבולת


```
struct Link {
    Link *next;
    Link(Link *n =0) : next(n) {}
    // ...
};

vector<Link> v;

/** fill v with n Links so that each Link points to its predecessor
 */
void chain(size_t n)
{
    v.reserve(n);
    v.push_back(Link(0));
    for(int i = 1 ; i<n ; i++)
        v.push_back(Link(&v[i-1]));
    // ...
}
```

עוד STL

- תיעוד מקוון (MSDN)
- ברשת:
  - Sgi STL Tutorial
  - Rogue Wave's STL Guide
- ספרים:
  - ***Effective STL - Scott Meyers, Addison-Wesley***
  - ***The C++ Programming Language* - Bjarne Stroustrup,**