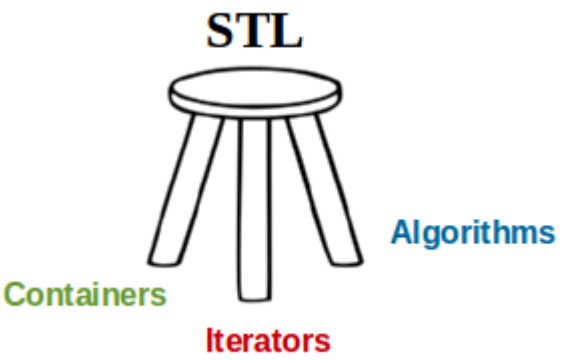
נושאים מתקדמים בתכנות מונחה עצמים
הרצאה 1

פרופ' עפר שיר
ofersh@telhai.ac.il

החוג למדעי המחשב
תל-חי
המכללה האקדמית

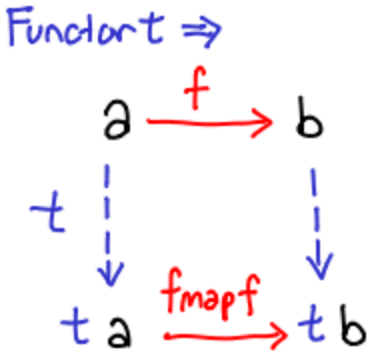
מבנה ההרצאה

- Functors
- Iterators



The reason that STL containers and algorithms work so well together is that they know nothing of each other.

– Alex Stepanov



FUNCTORS

Functors

- שם נוסף לאובייקטי פונקציה:
  מחלקות אשר עוטפות את `operator()`
  - יתרון: פונקציות המסוגלות לזכור "מצב"
- שימוש נרחב ב-STL בעבודה עם אלגוריתמים גנריים

```
class MultiplyBy {
int factor;
public:
  MultiplyBy(int x) : factor(x) { }
  int operator () (int other) const {return factor * other;}
};
```

```
int array[5] = {1, 2, 3, 4, 5};
std::transform(array, array + 5, array, MultiplyBy(3));
// The array reads {3, 6, 9, 12, 15}
```

Functors cont'd

- יתרונות
  - ייצוג פונקציה בעלת "מצב"
  - תואם עקרונות תכנות מונחה-עצמים
- חסרונות
  - קוד ארוך יותר: בעייתי כאשר השימוש חד-פעמי
  - תיתכן תקורה בזמן הקומפילציה; `inline` פותר זאת
  - לא ניתן להחלפה עם פונקציה אחרת בזמן-ריצה (אלא אם מתקיים פולימורפיזם – בעל תקורה)

21.7 Standard Function Objects

Function objects (`#include <functional>`)

STL function objects	Type
`divides< T >`	arithmetic
`equal_to< T >`	relational
`greater< T >`	relational
`greater_equal< T >`	relational
`less< T >`	relational
`less_equal< T >`	relational
`logical_and< T >`	logical
`logical_not< T >`	logical
`logical_or< T >`	logical
`minus< T >`	arithmetic
`modulus< T >`	arithmetic
`negate< T >`	arithmetic
`not_equal_to< T >`	relational
`plus< T >`	arithmetic
`multiplies< T >`	arithmetic

```
1      // Fig. 21.42: fig21_42.cpp
2      // Demonstrating function objects.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <vector>      // vector class-template definition
9  #include <algorithm>   // copy algorithm
10 #include <numeric>     // accumulate algorithm
11 #include <functional>  // binary_function definition
12
13 // binary function adds square of its second argument and
14 // running total in its first argument, then returns sum
15 int sumSquares( int total, int value )
16 {
17   return total + value * value;
18
19 } // end function sumSquares
20
```

```
21    // binary function class template defines overloaded operator()
22    // that adds square of its second argument and running total in
23    // its first argument, then returns sum
24    template< class T >
25    class SumSquaresClass : public std::binary_function< T, T, T > {
26
27    public:
28
29       // add square of value to total and return result
30       const T operator()( const T &total, const T &value )
31       {
32          return total + value * value;
33
34       } // end function operator()
35
36    }; // end class SumSquaresClass
37
```

```
38    int main(void)
39    {
40       const int SIZE = 10;
41       int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
42
43       std::vector< int > integers( array, array + SIZE );
44
45       std::ostream_iterator< int > output( cout, " " );
46
47       int result = 0;
48
49       cout << "vector v contains:\n";
50       std::copy( integers.begin(), integers.end(), output );
51
52       /* calculate sum of squares of elements of vector integers
53          using binary function sumSquares */
54       result = std::accumulate( integers.begin(), integers.end(),
55         0, sumSquares );
56
57       cout << "\n\nSum of squares of elements in integers using "
58            << "binary\nfunction sumSquares: " << result;
59
```

accumulate initially passes **0** as the first argument, with the first element as the second. It then uses the return value as the first argument, and iterates through the other elements.

```
60        /* calculate sum of squares of elements of vector integers
61           using binary-function object */
62        result = std::accumulate( integers.begin(), integers.end(),
63           0, SumSquaresClass< int >() );
64
65        cout << "\n\nSum of squares of elements in integers using "
66             << "binary\nfunction object of type "
67             << "SumSquaresClass< int >: " << result << endl;
68
69        return 0;
70
71     } // end main
```

```
vector v contains:
1 2 3 4 5 6 7 8 9 10

Sum of squares of elements in integers using binary
function sumSquares: 385

Sum of squares of elements in integers using binary
function object of type SumSquaresClass< int >: 385
```

```
template< class F, class T >
std::vector<T> fmap(F f, const std::vector<T>&
                    vec) {
    std::vector<T> result;
    std::transform(vec.begin(), vec.end(),
result.begin, f);
    return result;
}
```

```
struct identity {
    template<class T>
    T operator()(T x) const   { return x; }
};
```

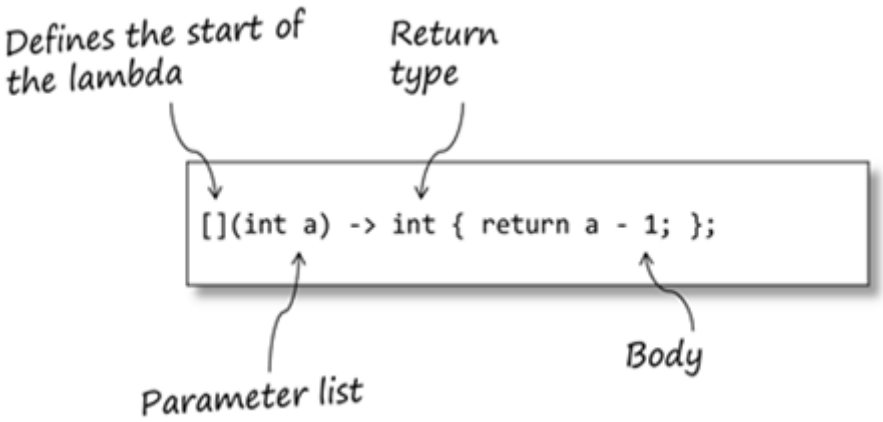
```
identity I = {};
std::vector<int> is = { 1, 2, 3 };
std::vector<int> is1 = fmap(identity(), is);
std::vector<int> is2 = I(is);
```

C++11 Proudly Presents: λ

- מענה על חסרונות אובייקטי הפונקציה:
  פונקציית אינליין אנונימית המוגדרת בסקופ מוגבל
- מדובר באובייקט הנוצר ע"י הקומפיילר
  - `operator()` הינו קבוע כברירת מחדל
- תחביר קומפקטי ושונה:
  - ניתן להשמיט ערך החזרה אם הוא מובן או `void`
- אין אפשרות לערכי ברירת-מחדל

```
void sugar (std::vector<double>& v) {
  std::transform(v.begin(), v.end(), v.begin(),
                 [](double d) { return d < 0.00001 ? 0 : d; }
                 );
}
```

Lambda Basic Syntax




```
void sugar1 (std::vector<double>& v) {
  std::transform(v.begin(),v.end(),v.begin(),
        [](double d) -> double {
            if (d < 0.0001) {
                return 0;
            } else {
                return d;
            }
        });
}
```

Lambda Capture

The 'context' is the set of objects in scope

```
int main()
{
  vector<X> v;

  // Add elements to the vector…

  int i = 10;

  for_each(v.begin(), v.end(),
    [i](X& elem)
    {
       cout << elem.getVal() * i << endl;
    }
  );
}
```

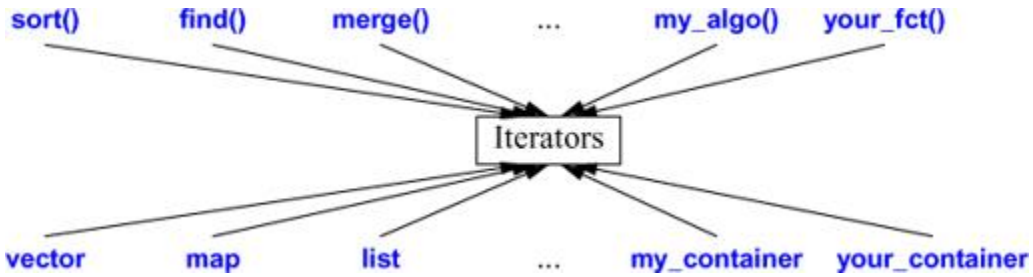
'Capture' i by value

this i is the lambda's local copy, not the original

```
void honey(std::vector<double>& v, const double& eps)
{
    std::transform(v.begin(), v.end(), v.begin(),
        [&eps](double d) -> double {
            if (d < eps) {
                return 0;
            } else {
                return d;
            }
        });
}
```

λ

פרטים נוספים וכמובן תרגילים יינתנו במפגש התרגול.


sort()
find()
merge()
...
my_algo()
your_fct()
Iterators
vector
map
list
...
my_container
your_container

ITERATORS

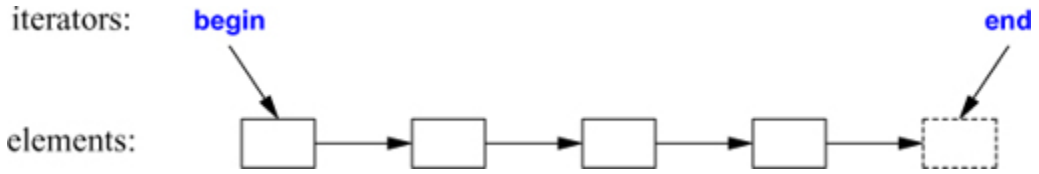
מוטיבציה

- נתונה פונקציית החיפוש הבאה:

```
template <class T>
T *find (T *begin, T *end, const T& look_for) {
      while (begin != end && *begin != look_for)
            ++begin;
      return begin;
}
```

- האם הפונקציה היתה עובדת אילו הפרמטרים היו כתובות של items ברשימה מקושרת?

מודל האיטרטור



לשם מעבר שיטתי על סדרת איברים, אלגוריתם יעבוד לרוב עם צמד איטרטורים `(b,e)` , ויבצע מהלך שיטתי בעזרת ++ עד לסוף הסדרה:

```
while (b!=e) { // use != rather than <
   // do something
   ++b;   // go to next element
}
```

תכונות האיטרטור

האיטרטור הוא למעשה קונספקט המכליל את מושג הפוינטר עבור container כלשהו. תכונות האיטרטור:

- הוא אובייקט ה"מצביע" על איבר (טיפוס `T`) ב container
- הפעלת `++` (או `--`) על איטרטור תגרום לו להצביע על האיבר הבא (הקודם) ב container.
- הפעלת `operator*` (אופרטור התוכן של פוינטר) על איטרטור תחזיר את תוכנו (האיבר עליו הוא מצביע)
- האיטרטור יהיה מסוגל להשוות עצמו עם איטרטור אחר כדי לבדוק אם הם מצביעים על אותו איבר
- אפשרות המרה ל `T*`

הכללת פונקצית החיפוש, בעזרת איטרטורים, תיראה כך:

```
template <class Iterator, class T>
Iterator find (Iterator begin, Iterator end,
const T& look_for) {
      while (begin != end && *begin != look_for)
            ++begin;
      return begin;
}
```

פעולות איטרטורים

Iterator operations	
++p	Pre-increment: make **p** refer to the next element in the sequence or to one-beyond-the-last-element ("advance one element"); the resulting value is **p+1**.
p++	Post-increment: make **p** refer to the next element in the sequence or to one-beyond-the-last-element ("advance one element"); the resulting value is **p** (before the increment).
--p	Pre-decrement: make **p** point to previous element ("go back one element"); the resulting value is **p−1**.
p--	Post-decrement: make **p** point to previous element ("go back one element"); the resulting value is **p** (before the decrement).
***p**	Access (dereference): ***p** refers to the element pointed to by **p**.
p[n]	Access (subscripting): **p[n]** refers to the element pointed to by **p+n**; equivalent to ***(p+n)**.
p->m	Access (member access); equivalent to **(*p).m**.
p==q	Equality: true if **p** and **q** point to the same element or both point to one-beyond-the-last-element.
p!=q	Inequality: **!(p==q)**.

Iterator operations (*continued*)	
p<q	Does **p** point to an element before what **q** points to?
p<=q	**p<q || p==q**
p>q	Does **p** point to an element after what **q** points to?
p>=q	**p>q || p==q**
p+=n	Advance **n**: make **p** point to the **n**th element after the one it points to.
p−=n	Advance **−n**: make **p** point to the **n**th element before the one it points to.
q=p+n	**q** points to the **n**th element after the one pointed to by **p.**
q=p−n	**q** points to the **n**th element before the one pointed to by **p**; afterward, we have **q+n==p.**
advance(p,n)	Advance: like **p+=n**; **advance()** can be used even if **p** is not a random-access iterator; it may take **n** steps (through a list).
x=difference(p,q)	Difference: like **q−p**; **difference()** can be used even if **p** is not a random-access iterator; it may take **n** steps (through a list).

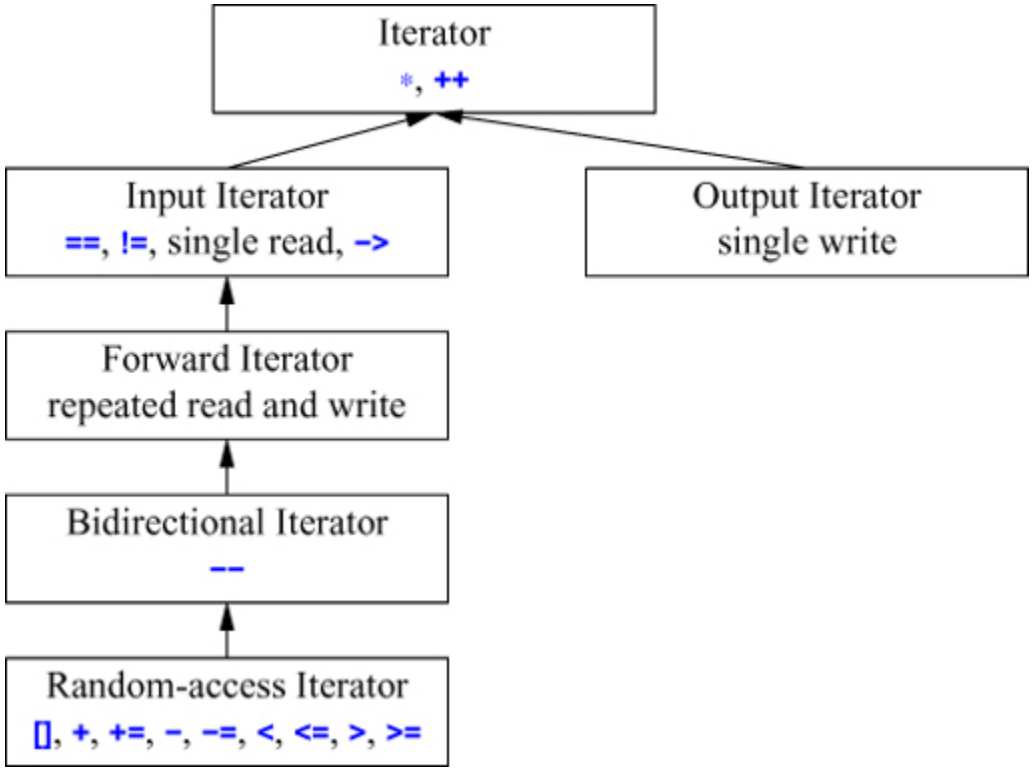
קטגוריות איטרטורים

STL מספקת איטרטורים השייכים לחמש קטגוריות :

Iterator categories	
input iterator	We can iterate forward using **++** and read each element once only using *****. We can compare iterators using **==** and **!=**. This is the kind of iterator that **istream** offers; see §21.7.2.
output iterator	We can iterate forward using **++** and write each element once only using *****. This is the kind of iterator that **ostream** offers; see §21.7.2.
forward iterator	We can iterate forward repeatedly using **++** and read and write (unless the elements are **const**) elements using *****. If it points to a class object, it can use **->** to access a member.

Iterator categories (*continued*)	
bidirectional iterator	We can iterate forward (using **++**) and backward (using **--**) and read and write (unless the elements are **const**) elements using *****. This is the kind of iterator that **list**, **map**, and **set** offer.
randomaccess iterator	We can iterate forward (using **++** or **+=**) and backward (using **—** or **-=**) and read and write (unless the elements are **const**) elements using ***** or **[]**. We can subscript, add an integer to a random-access iterator using **+**, and subtract an integer using **-**. We can find the distance between two random-access iterators to the same sequence by subtracting one from the other. We can compare iterators using **<**, **<=**, **>**, and **>=**. This is the kind of iterator that **vector** offers.

היראכיה


Iterator
*, ++
Input Iterator
==, !=, single read, ->
Output Iterator
single write
Forward Iterator
repeated read and write
Bidirectional Iterator
--
Random-access Iterator
[], +, +=, -, -=, <, <=, >, >=

Iterator Traits & Tags

Iterator Traits (§iso.24.4.1)	
iterator_traits<Iter>	Traits type for a non-pointer Iter
iterator_traits<T*>	Traits type for a pointer T*
iterator<Cat,T,Dist,Ptr,Re>	Simple class defining the basic iterator member types
input_iterator_tag	Category for input iterators
output_iterator_tag	Category for output iterators
forward_iterator_tag	Category for forward iterators; derived from input_iterator_tag; provided for forward_list, unordered_set, unordered_multiset, unordered_map, and unordered_multimap
bidirectional_iterator_tag	Category for bidirectional iterators; derived from forward_iterator_tag; provided for list, set, multiset, map, multimap
random_access_iterator_tag	Category for random-access iterators; derived from bidirectional_iterator_tag; provided for vector, deque, array, built-in arrays, and string

Iterator Traits

לשם השגת גנריות מלאה, STL מספקת מחלקת תבנית לייצוג כל התכונות האפשריות של האיטרטור:

```
namespace std {
    template <class T>
    struct iterator_traits {
        typedef typename T::value_type            value_type;
        typedef typename T::difference_type       difference_type;
        typedef typename T::iterator_category     iterator_category;
        typedef typename T::pointer               pointer;
        typedef typename T::reference             reference;
    };
}
```

T מייצג אובייקט איטרטור, כך שהמבנה מבטיח שכל טיפוסי המשתנים הללו מוגדרים היטב.

Tag Dispatch

- תיוג האיטרטורים נועד לאפשר חישוב אופטימלי של האלגוריתם הפועל (התאמה לטיפוס האיטרטור)
- למשל, איטרטור random-access מסוגל לבצע:

```
template<typename Iter>
void advance_helper(Iter& p, int n, random_access_iterator_tag)
{
  p+=n;
}
```

Tag Dispatch cont'd

- לעומת זאת, איטרטור bidirectional יתקדם בצעדים בודדים:

```
template<typename Iter>
void advance_helper(Iter& p, int n,
                        bidirectional_iterator_tag){
  if (n>0)
    while (n--) ++p;
  else if (n<0)
    while (n++) --p;
}
```

- באופן כזה, הפונקציה **advance()** תוכל לבצע בעקביות את החישוב האופטימלי:

```
template<typename Iter>
void advance(Iter& p, int n) {
     advance_helper( p, n,
typename iterator_traits<Iter>::iterator_category{} );
}
```

Specialization for Pointers

```
namespace std {
    template <class T>
    struct iterator_traits<T*> {
        typedef T                           value_type;
        typedef std::ptrdiff_t              difference_type;
        typedef random_access_iterator_tag  iterator_category;
        typedef T*                          pointer;
        typedef T&                          reference;
    };
}
```

- הייחוד הנ"ל מאפשר לראות במצביעים למערך כאיטרטורים מטיפוס random-access
- כך הושגה עקביות עבור מצביעים פרימיטיביים (אשר אינם מכילים את הטיפוסים הנ"ל) ועבור אובייקטי איטרטור של השפה

כתיבת פונקציה גנרית עבור איטרטורים

```
template<typename Iter> // NOT GENERAL
typename Iter::value_type read(Iter p, int n) {
     // ... do some checking ...
     return p[n];
}
```

← הרעיון הוא לבדוק את תכונות האיטרטור, במבנה **iterator_traits**, במקום את האיטרטור עצמו:

```
template<typename Iter> // More general
typename iterator_traits<Iter>::value_type read(Iter p, int n)
{
     // ... do some checking ...
     return p[n];
}
```

תכונות נוספות: מרחק בין איטרטורים

תכונת איטרטור כללית היא הגדרת המרחק במרחב הכתובות,
מטיפוס **difference_type**, באמצעות **std::distance** :

```
template<typename Iter>
void f(Iter p, Iter q) {

/* First attempt: SYNTAX ERROR: "typename" missing */
  Iter::difference_type d1 = std::distance(p,q);

/* Second attempt: wouldn't work for pointers! */
  typename Iter::difference_type d2 = std::distance(p,q);

/* Third attempt: OKAY */
  typename iterator_traits<Iter>::distance_type d3 =
       std::distance(p,q);

    // ...

}
```

כתיבת פונקציה גנרית עבור איטרטורים

```
template <typename Itr>
inline void my_func (Itr begin, Itr end)
{
  func_helper (begin, end,
      std::iterator_traits<Itr>::iterator_category{}
);
}
```

```
template <typename BidectionalIterator>
void func_helper (BidectionalIterator begin,
                  BidirectionalIterator end,
                  std::bidirectional_iterator_tag)
 {
       //Bidirectional Iterator specific code is here
 }

template <typename RandomIterator>
void  func_helper(RandomIterator begin,
                  RandomIterator end,
                  std::random_access_iterator_tag)
 {

       // Random access Iterator specific code is here
 }
```

struct iterator

לסיכום, מבנה האיטרטור הכללי מאגד את התכונות המוזכרות לכדי **struct** בסיסי עם ערכי ברירת מחדל:

```
template<typename Cat, typename T, typename Dist = ptrdiff_t,
typename Ptr = T*, typename Ref = T&>

struct iterator {

    using value_type = T;

    using difference_type = Dist ;    // type used by distance()

    using pointer = Ptr;              // pointer type

    using reference = Ref;            // reference type

    using iterator_category = Cat;    // category (tag)

};
```

Alias-declaration* in C++11 is equivalent to a *typedef-name