נושאים מתקדמים בתכנות מונחה עצמים

תרגיל/מעבדה 3

פרופ' עפר שיר

ofersh@telhai.ac.il

החוג למדעי המחשב

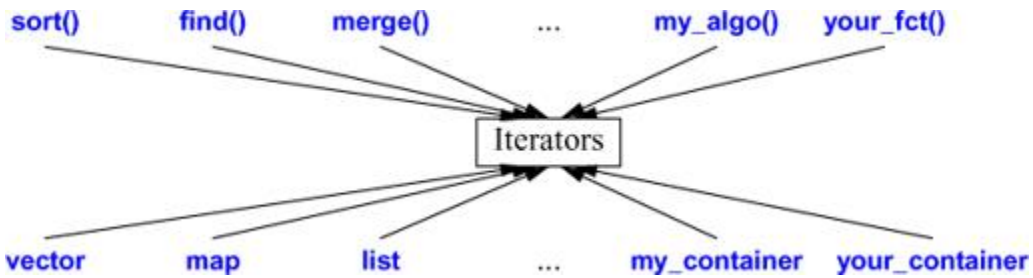
תל-חי

המכללה האקדמית

מטרת התרגיל

- λ solutions

- **`std::iterator<…>`**



$$\lambda$$

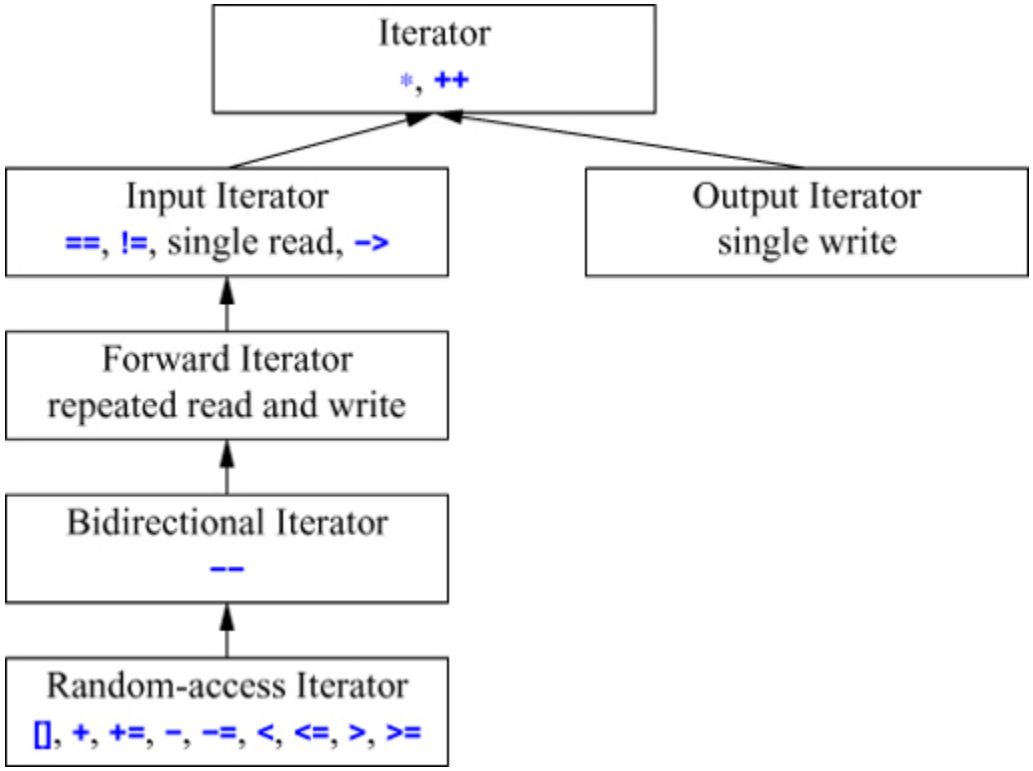
```
#include <iostream>
#include <map>
#include <iterator>
using namespace std;
template <typename T> class HBO {
public: bool operator()(const T& a, const T& b) const {
      return a > b;
   }
};
// typedef std::multimap< double, string, HBO<double> > mmh;
auto cmpFnc = [](double a, double b) {return a>b ? true : false;};
using mmh = std::multimap<double,string, decltype(cmpFnc) >;
int main(void) {
   mmh tvs(cmpFnc);
   tvs.insert( mmh::value_type(8.5,string("Olive Kitteridge")) );
   tvs.insert( mmh::value_type(9.4,string("The Sopranos")) );
   tvs.insert( mmh::value_type(9.4,string("The Wire")) );
   tvs.insert( mmh::value_type(9.1,string("Game of Thrones")) );
   tvs.insert( mmh::value_type(8.7,string("Boardwalk Empire")) );
   tvs.insert( mmh::value_type(9.4,string("True Detective")) );
   mmh::const_iterator iter = tvs.begin();
   unsigned short list = 4;
   while (list--) {
      cout << iter->first << '\t'<< iter->second << '\n';
      iter++;
   }
   return 0;
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
template<class T, class U>
struct ACDC {
   bool operator()(const std::pair<T,U>& a, const std::pair<T,U>& b) const {
      return *a.first < *b.first; }
 };
template<class IterIn, class IterOut>
void rock_n_roll(IterIn first, IterIn last, IterOut out) {
   std::vector< std::pair<IterIn,int> > s(last-first);
   for(int i=0; i < s.size(); ++i)
      s[i] = std::make_pair(first+i, i);

   std::sort(s.begin(), s.end(), [](const std::pair<IterIn,int>& a, const
std::pair<IterIn,int>& b){ return *a.first < *b.first; } );
   for(int i=0; i < s.size(); ++i, ++out)
      *out = s[i].second;
 }
int main(void) {
   int a[10] = { 15,12,13,14,18,11,10,17,16,19 };
   std::vector<int> b(10);
   rock_n_roll(a, a+10, b.begin());
   for(int i=0; i<10; ++i)
      std::cout << b[i] << ' ';
   std::cout << std::endl;
   return 0;
}
```

iterator_traits

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

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

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

1. פונקציית הדפסה עבור אוסף כלשהו

```
template<typename Iter>
void print(Iter first, Iter last,
const char* label, const char* sep,
std::ostream& os);
```

– ניתן לעשות שימוש באלגוריתם `std::copy`, אך כיצד?

print

```
template<typename Iter>
void print(Iter first, Iter last, const
char* lb = "", const char* sep = "\n",
std::ostream& os = std::cout) {
  if(lb != 0 && *lb != '\0')
    os << lb << ": " << sep;
  typedef typename
std::iterator_traits<Iter>::value_type T;
  std::copy(first, last,
std::ostream_iterator<T>(os, sep));
  os << std::endl;
}
```

```
TokenIterator :
public std::iterator<…>
```

A completely reusable ***tokenizer***

```
template<class InputIter, class Pred = Isalpha>
class TokenIterator : public std::iterator<
std::input_iterator_tag, std::string, std::ptrdiff_t> {
  InputIter first, last;
  std::string word;
  Pred predicate;
public:
  TokenIterator(InputIter begin, InputIter end,  Pred pred =
Pred()) : first(begin), last(end), predicate(pred) { ++*this; }
  TokenIterator() {}
  /* SEE NEXT CHART for operator++() etc. */
  std::string operator*() const { return word; }
  const std::string* operator->() const { return &word; }
  bool operator==(const TokenIterator&) const {
    return word.size() == 0 && first == last;
  }
  bool operator!=(const TokenIterator& rv) const {
    return !(*this == rv);
  }
};
```

operator++(), operator++(int)

```
// Prefix increment:
  TokenIterator& operator++() {
    word.resize(0);
    first = std::find_if(first, last, predicate);
    while (first != last && predicate(*first))
      word += *first++;
    return *this;
  }
  // Postfix increment
  class Proxy {
    std::string word;
  public:
    Proxy(const std::string& w) : word(w) {}
    std::string operator*() { return word; }
  };
  Proxy operator++(int) {
    Proxy d(word);
    ++*this;
    return d;
  }
```

Isalpha, Delimeters

```
struct Isalpha {
  bool operator()(char c) {
    return isalpha(c);
  }
};

class Delimiters {
  std::string exclude;
public:
  Delimiters() {}
  Delimiters(const std::string& excl)
    : exclude(excl) {}
  bool operator()(char c) {
    return exclude.find(c) == std::string::npos;
  }
};
```

TokenIteratorTest.cpp

```
int main(void) {
  char* fname = "TokenIteratorTest.cpp";
  ifstream in(fname);
  ostream_iterator<string> out(cout, "\n");
  typedef istream_iterator<char> IsbIt;
  IsbIt begin(in), isbEnd;
  Delimiters delimiters(" \t\n~;()\"<>:{}[]+-
=&*#.,/\\");
  TokenIterator<IsbIt, Delimiters>
wordIter(begin, isbEnd, delimiters), end;
  vector<string> wordlist;
  copy(wordIter, end, back_inserter(wordlist));
  copy(wordlist.begin(), wordlist.end(), out);
```

function template

std::back_inserter

<iterator>

```
template <class Container>
  back_insert_iterator<Container> back_inserter (Container& x);
```

Construct back insert iterator

Constructs a *back-insert iterator* that inserts new elements at the end of *x*.

A *back-insert iterator* is a special type of *output iterator* designed to allow algorithms that usually overwrite elements (such as `copy`) to instead insert new elements automatically at the end of the container.

The type of *x* needs to have a `push_back` member function (such as the standard containers `vector`, `deque` and `list`).

Using the assignment operator on the returned iterator (either dereferenced or not), causes the container to expand by one element, which is initialized to the value assigned.

The returned iterator supports all other typical operations of output iterators but have no effect: all values assigned are inserted at the end of the container.

Parameters

`x`

Container on which the iterator will insert new elements.
`Container` should be a container class with member `push_back` defined.

Return value

A `back_insert_iterator` that inserts elements at the end of container *x*.

3. פונקציית התקדמות גנרית אך מותאמת

הציעו מנשק ומימוש עבור פונקציית התקדמות גנרית של האיטרטור הקאנוני; זוהי פונקציית תבנית (פרמטר התבנית הוא האיטרטור) המקבלת את מספר הצעדים בהם צריך להתקדם האיטרטור כארגומנט.

פונקציה זו תסתמך על העמסה של פונקציות עזר שתהיינה ספציפיות עבור טיפוסי האיטרטורים השונים.

לאחר מכן, ממשו את פונקציות העזר הנדרשות לפונקציה שהצעתם עבור איטרטורים חמש הקטגוריות הקיימות.