ΗΥ352 : ΕΡΓΑΣΤΗΡΙΟ ΤΕΧΝΟΛΟΓΙΑΣ ΛΟΓΙΣΜΙΚΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

7ο ΦΡΟΝΤΙΣΤΗΡΙΟ

ΔΙΔΑΣΚΟΝΤΑΣ
Αντώνιος Σαββίδης

Περιεχόμενα

- Iterators
- STL
  - Containers
  - Algorithms
  - Function Objects

Iterators (1/4)

- Ο iterator αποτελεί ένα σχεδιαστικό πρότυπο που μας βοηθάει να διατρέξουμε τα στοιχεία ενός container χωρίς να γνωρίζουμε (ή να έχουμε πρόσβαση) στην υλοποίησή του
- Επιπλέον χρησιμοποιούμε το ίδιο interface για τη χρήση του iterator ανεξάρτητα από το container
- Στη C++ ο iterator είναι ένα είδος «έξυπνου pointer» και συνήθως εξομοιώνει τις περισσότερες λειτουργίες ενός pointer (π.χ. increment, και dereference)
  - Βέβαια παρέχει μεγαλύτερη ασφάλεια από ένα απλό pointer

Iterators (2/4)

```
class Stack {
   int stack[100], top;
public:
   Stack() : top(0) {}
   void push(int i)
      { stack[top++]; }
   int pop()
      { return stack[--top]; }
   friend class StackIter;
};
```

```
class StackIter {
   Stack& s;
   int index;
public:
  StackIter(IntStack& s):s(s),index(0) {}
  int operator++() //prefix
      { return s.stack[++index]; }
  int operator++(int) //postfix
      { return s.stack[index++]; }
};
```

```
Stack s; s.push(1); s.push(2); s.push(3); s.push(4);
StackIter iter(s);
for(int i = 0; i < 4; ++i)              //iterate the stack
    std::cout << iter++ << std::endl; //prints 1 2 3 4
```

Iterators (3/4)

- Γενικά, κάθε φορά που φτιάχνουμε ένα container, φτιάχνουμε και ένα iterator class με το οποίο θα μπορούμε να διατρέχουμε τα στοιχεία του container
- Καλό είναι να έχουμε και standard API για τη χρήση των iterators (ακολουθώντας το παράδειγμα της STL)
- Υλοποιούμε το iterator class κάνοντας του overload τις απαραίτητες συναρτήσεις (increment, dereference, equality checks, κλπ)
- Στο container class προσθέτουμε τις μεθόδους begin και end που επιστρέφουν iterators στην αρχή και στο τέλος του container αντίστοιχα
  - Συγκεκριμένα ο iterator που δείχνει στο τέλος (end sentinel) δε γίνεται **ποτέ** dereferenced, και χρησιμοποιείται μόνο για να ξέρουμε πότε φτάσαμε στο τέλος της ακολουθίας (past the end)

Iterators (4/4)

```
class Stack {
   int stack[100], top;

public:

   Stack() : top(0) {}

   void push(int i) { stack[top++]; }

   int pop() { return stack[--top]; }

   friend class iterator;
   class iterator {
       Stack& s; int index;
   public:
       iterator(Stack& s, int i) : s(s), index(i) {}
       int& operator*() const { return s[index]; }
       iterator& operator++() { ++index; return *this; }
       iterator operator++(int) { iterator iter(s, index); ++index; return iter; }
       bool operator == (const iterator& i) { return index == i.index; }
       bool operator != (const iterator& i) { return !operator==(i); }
   };
   iterator begin() { return iterator(*this, 0); }
   iterator end()   { return iterator(*this, top); }

};

Stack s; s.push(10); s.push(20); s.push(30); s.push(40);

for(Stack::iterator i = s.begin(); i != s.end(); ++i) //iterate the stack

   std::cout << *i << std::endl;                      //prints 10 20 30 40
```

Εισαγωγή στην STL (1/2)

- Η STL (Standard Template Library) είναι μια πολύ ισχυρή βιβλιοθήκη που έχει ως στόχο να ικανοποιήσει τις περισσότερες ανάγκες μας για *containers* και *αλγορίθμους*
- **Containers**
  - Π.χ. vector, deque (double ended queue), set, list, hash map, queue, stack, κλπ.
  - Υλοποιούνται ως template classes
  - Παρέχουν συγκεκριμένους iterators και ένα κοινό API για τη χρήση τους
  - Ικανοποιούν πολύ μεγάλο ποσοστό των αναγκών μας όσον αφορά τα containers

Εισαγωγή στην STL (2/2)

- **Αλγόριθμοι**
  - Δουλεύουν σε containers μεσω iterator interface
    - Δηλαδή δουλεύουν σε ***διάστημα από iterators***
  - Π.χ. for_each, count, equal, search, copy, reverse, sort, binary_serch, κλπ.
  - Υλοποιούνται ως template functions
- Θα δούμε στη συνέχεια μερικά παραδείγματα χρήσης από containers και αλγορίθμους της STL
  - Για περισσότερες λεπτομέρειες επισκεφτείτε το site http://www.sgi.com/tech/stl/

Containers - vector

- **template <typename T, typename Alloc> class *vector*;**
  - Ορίζεται στο header <vector>
- Το vector είναι μια ακολουθία που επιτρέπει τυχαία προσπέλαση στα στοιχεία του και εισαγωγή και αφαίρεση στοιχείων από το τέλος σε σταθερό χρόνο (O(1)) και από την αρχή και τη μέση σε O(n)
- Το template argument T είναι ο τύπος των δεδομένων που θα εμπεριέχονται στην ακολουθία
- Το *Alloc* χρησιμοποιείται για τη δέσμευση μνήμης μέσα στο container και έχει default τιμή (που σπάνια την αλλάζουμε)

```
#include <vector>
std::vector<int> v;
v.push_back(1);
assert(!v.empty()); assert(v.size() == 1);
v.pop_back();
v.erase(v.begin(), v.end());  // same as v.clear()
v.insert(v.begin(), 2); v.insert(v.end(), 3);
for(int i = 0; i < v.size(); ++i)
    std::cout << v[i] << " ";  // prints 2 3
for(std::vector<int>::iterator i = v.begin(); i != v.end(); ++i)
    std::cout << *i << " ";  // prints 2 3
```

Containers - list

- **template <typename T, typename Alloc> class *list*;**
  - Ορίζεται στο header <list>
- Η list είναι μια ακολουθία που επιτρέπει διάσχιση από την αρχή προς το τέλος και αντίστροφα
- Η εισαγωγή και αφαίρεση στοιχείων γίνεται σε κατά μέσο όρο σταθερό χρόνο (O(1)) (ανεξάρτητα αν γίνεται σε αρχή, μέση ή τέλος)
- Το template argument T είναι ο τύπος των δεδομένων που θα εμπεριέχονται στην ακολουθία

```
#include <list>
std::list<int> l;
l.push_back(1); l.push_back(1); l.push_front(2);
assert(!l.empty());
l.remove(1); assert(l.size() == 1);
l.pop_front();
l.erase(l.begin(), l.end());  // same as v.clear()
l.insert(l.begin(), 2); l.insert(l.end(), 3);
for(std::list<int>::iterator i = l.begin(); i != l.end(); ++i)
    std::cout << *i << " ";  // prints 2 3
for(std::list<int>::reverse_iterator i = l.rbegin(); i != l.rend(); ++i)
    std::cout << *i << " "; // prints 3 2
```

Containers – map (1/3)

- **template <typename Key, typename Data, typename Compare, typename Alloc> class *map*;**
  - Ορίζεται στο header <map>
- Το map είναι ένα ταξινομημένο container που συσχετίζει *κλειδιά* (που έχουν τύπο Key) με *τιμές* (που έχουν τύπο Data)
- Κάθε κλειδί είναι μοναδικό
  - Αν θέλουμε ένα κλειδί να μπορεί να αντιστοιχιστεί σε πολλές τιμές υπάρχει και το container multimap
- Τα βασικά template arguments του map είναι ο **τύπος των κλειδιών (Key)** και ο **τύπος των δεδομένων (Data)**
- Το *Compare* χρησιμοποιείται για να δηλώσουμε τον τρόπο σύγκρισης που θα χρησιμοποιηθεί στην ταξινόμηση και έχει default τιμή (αλλά συχνά χρειάζεται να δώσουμε εμείς συγκεκριμένη τιμή)
- Το *Alloc* και πάλι χρησιμοποιείται για τη δέσμευση μνήμης μέσα στο container και έχει default τιμή

Containers – map (2/3)

Παράδειγμα χρήσης map

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

typedef std::map<int, const char *> Map;
Map m;
m.insert(Map::value_type(1, "one")); //map<K, V>::value_type == std::pair<K, V>
m.insert(std::pair<int, const char *>(2, "two"));
m[3] = "three";
for(Map::iterator i = m.begin(); i != m.end(); ++i)
     std::cout << i->first << " --> " << i->second << std::endl;
assert(m.count(1) == 1 && m.count(4) == 0);
assert(!m.empty());
m.erase(1);
assert(m.size() == 2);
Map::iterator i = m.find(3);
assert(i != m.end());
std::cout << i->first << " --> " << i->second << std::endl; //prints 3 --> three
m.erase(i);
assert(m.size() == 1);
```

Containers – map (3/3)

Παράδειγμα χρήσης με συγκεκριμένο Compare

```
#include <map>
#include <cstring>
#include <iostream>
struct lessThanStr {
    bool operator()(const char* s1, const char* s2) const
        { return strcmp(s1, s2) < 0; }
};
int main() {
  typedef std::map<const char*, int, lessThanStr> Map;
  Map months;
  months["january"] = 31; months["february"] = 28; months["march"] = 31;
  months["april"] = 30; months["may"] = 31; months["june"] = 30;
  months["july"] = 31; months["august"] = 31; months["september"] = 30;
  months["october"] = 31; months["november"] = 30; months["december"] = 31;
  std::cout << "june -> " << months["june"] << std::endl;
  Map::iterator cur  = months.find("june");
  Map::iterator prev = cur, next = cur;
  ++next; --prev;
  std::cout << "Previous (alphabetically) is " << (*prev).first << std::endl; // july
  std::cout << "Next (alphabetically) is " << (*next).first << std::endl;     // march
}
```

Αλγόριθμοι - for_each (1/3)

- **template <class InputIterator, class UnaryFunction> UnaryFunction *for_each*(InputIterator first, InputIterator last, UnaryFunction f);**
  - *Ορίζεται στο header <algorithm> (όπως και όλοι οι αλγόριθμοι)*
- Εφαρμόζει τη συνάρτηση f σε όλα στοιχεία της από το first μέχρι το last
- Η συνάρτηση εφαρμόζεται πρώτα στο first, μετά στο επόμενο, κλπ μέχρι το last (δεν εφαρμόζεται στο last).
- Επιστρέφει τη συνάρτηση αφού την έχει εφαρμόσει σε όλα τα στοιχεία.

Εσωτερική Υλοποίηση

```
template <class It, class Func>
Func for_each(It begin, It end, Func f)
{
    for (; begin != end; ++begin)
        f(*begin);
    return f;
}
```

Αλγόριθμοι - for_each (2/3)

for_each χρησιμοποιώντας C πίνακα και functor object

```
#include <iostream>
#include <algorithm>
template<class T> struct print : public std::unary_function<T, void> {
  ostream& os; int count;
  print(ostream& out) : os(out), count(0)
    { os << "print functor created "; }
  void operator() (T x)
    { os << x << ' '; ++count; }
};
int A[] = {1, 4, 2, 8, 5, 7};
const int N = sizeof(A) / sizeof(int);
print<int> P = std::for_each(A, A + N, print<int>(cout));
std::cout << std::endl << P.count << " objects printed." << std::endl;
// 1)Δημιουργείται ένα temporary print instance και τυπώνεται το "print functor created".
// 2)Τρέχει η for_each με το instance αυτό καλώντας τον operator() για κάθε στοιχείο από
// την αρχή του πίνακα μέχρι το τέλος του, τυπώνοντας έτσι τα στοιχεία του πίνακα και
// αυξάνοντας το count τότες φορές όσα τα στοιχεία.
// 3)Η for_each μας επιστρέφει το instance της functor κλάσης μας και το εκχωρούμε στο P.
// 4)Χρησιμοποιούμε το P για να τυπώσουμε τον πλήθος των στοιχείων που τυπώθηκαν.
```

```
template <class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
};
```

Αλγόριθμοι - for_each (3/3)

Χρήση με ***stl container*** και **functor object**

```
#include <iostream>
struct accumulator : public std::unary_function<int, void> {
  int sum;
  accumulator(void) : sum(0) {}
  void operator() (int x) { sum += x; }
};

int main() {
  std::list<int> l;
  l.push_back(1); l.push_back(5); l.push_back(30);
  accumulator A = std::for_each(l.begin(), l.end(), accumulator());
  std::cout << A.sum << std::endl; // prints 36
}
```

Χρήση με ***stl container*** και **συνάρτηση**

```
#include <cstdio>
void print(char c) { putchar(c); }

int main() {
  std::list<char> l;
  l.push_back('a'); l.push_back('b'); l.push_back('c');
  std::for_each(l.begin(), l.end(), print)('d');  // prints abcd
}
```

Αλγόριθμοι - find

- **template<class InputIterator, class EqualityComparable>**
  InputIterator *find*(InputIterator first, InputIterator last, const EqualityComparable& value);
- Επιστρέφει τον πρώτο iterator στο διάστημα [first, last) που η τιμή του είναι ίση με την τιμή value
- Αν δεν υπάρχει τέτοιος iterator επιστρέφει last

```
typedef std::list<int> IntList;
IntList l;
l.push_back(3);
l.push_back(1);
l.push_back(7);

IntList::iterator result = std::find(l.begin(), l.end(), 7);
if(result != l.end())
     std::cout << "Element with value 7 found!" << std::endl
```

Αλγόριθμοι - find_if

- **template<class InputIterator, class Predicate> InputIterator *find_if*(InputIterator first, InputIterator last, Predicate pred);**
- Επιστρέφει τον πρώτο iterator i στο διάστημα [first, last) τέτοιο ώστε το pred(*i) να είναι αληθές
  - Προσοχή: δεν είναι καθορισμένο αν το predicate δίνεται στη συνάρτηση by value ή by reference (ισχύει για όλα τα predicates)
    - Καλό είναι λοιπόν σε αυτό να έχουμε μόνο ***immutable access***
- Αν δεν υπάρχει τέτοιος iterator επιστρέφει last

```
struct positive : public std::unary_function<int, bool>
   { bool operator()(int x) const { return x > 0; } };
typedef std::list<int> IntList;
IntList l;
l.push_back(-3); l.push_back(2); l.push_back(7);
IntList::iterator result = std::find_if(l.begin(), l.end(), positive());
    std::cout << "Result = " << *result << std::endl;  // prints 2
```

Αλγόριθμοι - count

- **template <class InputIterator, class EqualityComparable> iterator_traits<InputIterator>::difference_type *count* (InputIterator first, InputIterator last, const EqualityComparable& value);**
- Επιστρέφει τον αριθμό των στοιχείων στο διάστημα [first, last) που έχουν τιμή ίση με value
- Το iterator_traits<InputIterator>::difference_type είναι ένας προσημασμένος ακέραιος που μπορεί να αναπαραστήσει την απόσταση μεταξύ δύο iterators, ουσιαστικά δηλαδή είναι ακέραιος

```
using namespace std;
int A[] = { 2, 0, 4, 6, 0, 3, 1, -7 };
cout << "Number of zeros: " << count(A, A + 8, 0) << endl; // prints 2

vector<bool> b;
b.push_back(true); b.push_back(true); b.push_back(false); b.push_back(true);
cout << "True values: " << count(b.begin(), b.end(), true) << endl; // prints 3
```

Αλγόριθμοι - count_if

- **template<class** InputIterator**, class** Predicate**>** iterator_traits**<InputIterator>::difference_type** ***count_if*** **(InputIterator first, InputIterator last, Predicate pred);**
- Επιστρέφει τον αριθμό των στοιχείων στο διάστημα [first, last) που ικανοποιούν το κατηγόρημα pred (δηλαδή το πλήθος iterator i για τους οποίους το pred(*i) είναι αληθές

```
struct even : public std::unary_function<int, bool> {
      bool operator()(int x) { return x % 2 == 0; }
};
int A[] = { 2, 0, 4, 6, 0, 3, 1, -7 };
std::cout << “Evens: " << std::count_if(A, A + 8, even()) << std::endl;
// prints 5
```

Αλγόριθμοι - equal

- **template <class InputIterator1, class InputIterator2> bool *equal*(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);**
- **template <class InputIterator1, class InputIterator2, class BinaryPredicate> bool *equal*(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, BinaryPredicate binary_pred);**
- Επιστρέφει αληθές αν τα δύο διαστήματα [first, last) και [first2, first2 + (last1 - first1)) είναι ίδια συγκρινόμενα στοιχείο με στοιχείο
- Η πρώτη έκδοση χρησιμοποιεί τον operator == για να ελέγξει την ισότητα, ενώ η δεύτερη χρησιμοποιεί το κατηγόρημα binary_pred

```
struct equal : public std::binary_function<int, int, bool> {
      bool operator()(int x, int y) { return x == y; }
};
int A1[] = { 3, 1, 4, 1, 5, 9, 3 };
int A2[] = { 3, 1, 4, 2, 8, 5, 7 };
std::cout << std::equal(A1, A1 + 7, A2) << std::endl;  // prints false
std::cout << std::equal(A1, A1 + 3, A2) << std::endl;  // prints true
std::list<int> l;
l.push_back(3); l.push_back(1); l.push_back(4);
std::cout << std::equal(A1, A1 + 3, l.begin(), equal()) << std::endl;  // prints true
```

Αλγόριθμοι - search

- **template <class ForwardIterator1, class ForwardIterator2> ForwardIterator1 *search*(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2);**
- **template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate> ForwardIterator1 *search*(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate binary_pred);**
- Η search ψάχνει για μια υποακολουθία στο διάστημα [first1, last1) που να είναι ίδια με το [first2, last2) συγκρινόμενα στοιχείο με στοιχείο και επιστρέφει τον iterator στην αρχή της υποακολουθίας ή last1 αν δεν υπάρχει τέτοια υποακολουθία
- Η πρώτη έκδοση χρησιμοποιεί τον operator == για να ελέγξει την ισότητα, ενώ η δεύτερη χρησιμοποιεί το κατηγόρημα binary_pred

```
struct equal : public std::binary_function<int, int, bool> {
      bool operator()(int x, int y) { return x == y; }
};
int A1[] = { 1, 2, 3, 1, 4, 1}, A2[] = { 3, 1, 4};
int * pos = std::search(A1, A1 + 6, A2, A2 + 3);
std::cout << "Index = " << pos – A1 << std::endl;  // prints 2
std::list<int> l;
l.push_back(3); l.push_back(1); l.push_back(4);
assert(std::search(l.begin(), l.end(), A1 + 2, A1 + 4, equal()) == l.begin());
```

Αλγόριθμοι - copy

- **template <class** InputIterator**, class** OutputIterator**> OutputIterator** ***copy*** **(InputIterator first, InputIterator last, OutputIterator result);**
- Η copy αντιγράφει στοιχεία από το διάστημα [first, last) στο διάστημα [result, result + (last - first))
- Η επιστρεφόμενη τιμή της copy είναι ο iterator result + (last - first)
- Οι εκχωρήσεις γίνονται με αυξανόμενη σειρά
- **Προσοχη:** Πρέπει να υπάρχει αρκετός χώρος για τα στοιχεία που αντιγράφονται

```
std::vector<int> v;
v.push_back(3); v.push_back(1); v.push_back(2);
std::list<int> l(v.size());
std::copy(v.begin(), v.end(), l.begin());
assert(std::equal(v.begin(), v.end(), l.begin()));
std::copy(v.begin(),v.end(),std::ostream_iterator<int>(std::cout, " "));
//prints 3 1 2
```

Αλγόριθμοι – transform (1/2)

- **template <class** InputIterator**, class** OutputIterator**, class** UnaryFunction**> OutputIterator *transform* (InputIterator first, InputIterator last, OutputIterator result, UnaryFunction op);**
- Εφαρμόζει τη συνάρτηση op(*i) για όλα τα στοιχεία στο διάστημα [first, last) και βάζει το αποτέλεσμα στο διάστημα [result, result + (last - first))
- Η επιστρεφόμενη τιμή της είναι iterator result + (last - first)
- **Προσοχη:** Πρέπει να υπάρχει αρκετός χώρος για τα στοιχεία εξόδου

```
struct square : public std::unary_function<int, int> {
     int operator()(int x) { return x * x; }
};
int A1[] = { 1, 2, 3, 4, 5, 6, 7 };
std::transform(A1, A1 + 7, A1, square());
std::copy(A1, A1 + 7, std::ostream_iterator<int>(std::cout, " "));
//prints 1 4 9 16 25 36 49
```

Αλγόριθμοι – transform (2/2)

- **template <class InputIterator1, class InputIterator2, class OutputIterator, class BinaryFunction> OutputIterator *transform* (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, OutputIterator result, BinaryFunction binary_op);**
- Παίρνει N (last – first) στοιχεία από κάθε είσοδο (first1, first2) και τα συνδυάζει εφαρμόζει πάνω σε αυτά τη συνάρτηση op(*i1, *ι2)
- Το αποτέλεσμα μπαίνει στο διάστημα [result, result + (last - first))
- Η επιστρεφόμενη τιμή της είναι ο iterator result + (last - first)
- **Προσοχη:** Πρέπει να υπάρχει αρκετός χώρος για τα στοιχεία εξόδου

```
struct plus : public std::binary_function<int, int, int> {
   int operator()(int x, int y) { return x + y; }
};
int A1[] = { 1, 2, 3, 4, 5, 6, 7 };
std::transform(A1, A1 + 7, A1, A1, plus());
std::copy(A1, A1 + 7, std::ostream_iterator<int>(std::cout, " "));
//prints 2 4 6 8 10 12 14
```

Αλγόριθμοι - replace

- **template <class ForwardIterator, class T> void *replace*(ForwardIterator first, ForwardIterator last, const T& old_value, const T& new_value);**
- Αντικαθιστά κάθε στοιχείο του διαστήματος [first, last) που είναι ίσο με old_value με new_value

```
std::vector<int> v;
v.push_back(1); v.push_back(2);
v.push_back(3); v.push_back(1);
std::replace(v.begin(), v.end(), 1, 99);
assert(v[0] == 99 && v[3] == 99);
```

Αλγόριθμοι - replace_if

- **template <class ForwardIterator, class Predicate, class T> void *replace_if*(ForwardIterator first, ForwardIterator last, Predicate pred, const T& new_value);**
- Αντικαθιστά κάθε στοιχείο του διαστήματος [first, last) που ικανοποιεί το κατηγόρημα pred με new_value

```
struct negative : public std::unary_function<int, bool> {
      bool operator()(int x) { return x < 0; }
};
std::vector<int> v;
v.push_back(1); v.push_back(-3);
v.push_back(2); v.push_back(-1);
std::replace_if(v.begin(), v.end(), negative(), 0);
assert(v[1] == 0 && v[3] == 0);
```

Αλγόριθμοι - remove

- template <class ForwardIterator, class T> ForwardIterator *remove*(ForwardIterator first, ForwardIterator last, const T& value);
- Αφαιρεί από το διάστημα [first, last) τα στοιχεία ίσα με value
- Επιστρέφει ένα iterator new_last τέτοιο ώστε το διάστημα [first, new_last) δεν περιέχει στοιχείο ίσο με value
- **Προσοχή:**
  - Η remove δεν καταστρέφει iterators και δεν αλλάζει την απόσταση μεταξύ first και last
  - Οι iterators [new_last, last) εξακολουθούν να υπάρχουν (μπορούν να γίνουν και dereference) αλλά δείχνουν σε ακαθόριστα στοιχεία

```
std::vector<int> v;
v.push_back(3); v.push_back(1); v.push_back(4); v.push_back(1);
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(cout, " ")); // prints 3 1 4 1
std::vector<int>::iterator new_end = std::remove(v.begin(), v.end(), 1);
std::copy(v.begin(), new_end, std::ostream_iterator<int>(cout, " ")); // prints 3 4
std::cout << v.size() << std::endl;                                   // prints 4, not 2!
```

Αλγόριθμοι - remove_if

- **template <class ForwardIterator, class Predicate> ForwardIterator *remove_if*(ForwardIterator first, ForwardIterator last, Predicate pred);**
- Ακριβώς όπως τη remove απλά αφαιρεί τα στοιχεία που ικανοποιούν το predicate pred

```
struct odd : public std::unary_function<int, bool> {
      bool operator()(int x) { return x % 2 != 0; }
};
std::vector<int> v;
v.push_back(3); v.push_back(1); v.push_back(4); v.push_back(1);
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(cout, " "));
// prints 3 1 4 1
std::vector<int>::iterator new_end = std::remove_if(v.begin(), v.end(), odd());
std::copy(v.begin(), new_end, std::ostream_iterator<int>(cout, " "));
// prints 4
std::cout << v.size() << std::endl; // prints 4, not 1!
```

Αλγόριθμοι - reverse

- **template <class BidirectionalIterator> void *reverse* (BidirectionalIterator first, BidirectionalIterator last);**
- Αντιστρέφει τα περιεχόμενα του διαστήματος [first, last) ώστε το στοιχείο που βρίσκεται στο first να αλλάξει θέση με αυτό που βρίσκεται στο last – 1, κλπ.

```
std::vector<char> v;
v.push_back('a');
v.push_back('b');
v.push_back('c');
std::reverse(v.begin(), v.end());
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(cout, "
"));  //prints c b a
```

Αλγόριθμοι - sort, stable_sort

- **template <class RandomAccessIterator> void *sort* (RandomAccessIterator first, RandomAccessIterator last);**
- **template <class RandomAccessIterator> void *stable_sort* (RandomAccessIterator first, RandomAccessIterator last);**
- Ταξινομούν τα στοιχεία του διαστήματος [first, last) σε αυξανόμενη σειρά χρησιμοποιώντας τον operator < για τη σύγκριση
  - Υπάρχουν και εκδόσεις που ταξινομούν βάση κάποιου predicate
- Η stable_sort διαφέρει από τη sort στο γεγονός ότι εγγυημένα διατηρεί τη σχετική διάταξη μεταξύ δύο ισότιμων (αλλά όχι απαραίτητα ίσων) στοιχείων

```
using namespace std;
int A[] = {3, 4, 1, 8};
sort(A, A + 4);
copy(A, A + 4,
  ostream_iterator<int>(cout, " ")
); // prints 1 3 4 8
```

```
bool lessThanIgnoreCase(char c1, char c2)
  { return tolower(c1) < tolower(c2); }
int main() {
  char A[] = "fdBeACFDbEac";   //size 12
  std::stable_sort(A, A + 12, lessThanIgnoreCase);
  printf("%s\n", A);  // prints AaBbCcdDeEfF
}
```

Αλγόριθμοι - binary_search

- **template <class ForwardIterator, class LessThanComparable> bool *binary_search*(ForwardIterator first, ForwardIterator last, const LessThanComparable& value);**
- **template <class ForwardIterator, class T, class StrictWeakOrdering> bool *binary_search*(ForwardIterator first, ForwardIterator last, const T& value, StrictWeakOrdering comp);**
- Ψάχνει το στοιχείο value στο διάστημα [first, last) χρησιμοποιώντας δυαδική αναζήτηση και επιστρέφει true αν βρήκε το στοιχείο, αλλιώς false
- Η πρώτη έκδοση χρησιμοποιεί τον operator < για τη σύγκριση, ενώ η δεύτερη χρησιμοποιεί το κατηγόρημα comp

```
using namespace std;

vector<int> v;
v.push_back(10); v.push_back(20);
v.push_back(30); v.push_back(50);
sort(v.begin(), v.end());
assert(
  binary_search(v.begin(),v.end(),20)
);
```

```
#include <cstring>
struct lessThanStr : public
  std::binary_function<const char*,const char*, bool>{
     bool operator()(const char* s1, const char* s2)
       { return strcmp(s1, s2) < 0; }
};
char *A[]={"alpha","beta","charlie","delta"};
// must have lexicographical order
assert(std::binary_search(A,A + 4,"beta",lessThanStr()));
```

Αλγόριθμοι - merge

- **template <class InputIterator1, class InputIterator2, class OutputIterator> OutputIterator *merge*(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);**
- **template <class InputIterator1, class InputIterator2, class OutputIterator, class StrictWeakOrdering> OutputIterator *merge*(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, StrictWeakOrdering comp);**
- Συνδυάζει δύο ταξινομημένες ακολουθίες [first1, last1) και [first2, last2) σε μια κοινή ακολουθία και επιστρέφει τον iterator result + (last1 - first1) + (last2 - first2)
- Η πρώτη έκδοση χρησιμοποιεί τον operator < για τη σύγκριση, ενώ η δεύτερη χρησιμοποιεί το κατηγόρημα comp
- **Προσοχή:** Πρέπει να υπάρχει αρκετός χώρος για τα στοιχεία που αντιγράφονται

```
using namespace std;
int A1[] = { 1, 3, 5, 7 };
list<int> l;
l.push_back(2); l.push_back(4); l.push_back(6); l.push_back(8);
vector v(4 + l.size());    //should always have enough space for all elements!
merge(A1, A1 + 4, l.begin(), l.end(), v.begin());//same as giving predicate less<int>()
copy(v.begin(), v.end(), ostream_iterator<int>(cout, " ")); // prints 1 2 3 4 5 6 7 8
```

Function Objects – Standard objects

- Η STL παρέχει κάποια έτοιμα template function objects
  - Αριθμητικά
    - plus, minus, multiplies, divides, modulus, negative
  - Συγκριτικά
    - equal_to, not_equal_to, less, greater, less_equal, greater_equal
  - Λογικά
    - logical_and, logical_or, logical_not

```
using namespace std;
int A1[] = { 1, 2, 3, 4 };
int A2[] = { 1, 4, 2, 3 };
bool A3[] = { false, false, false, false };
transform(A1, A1 + 4, A2, A3, greater_equal<int>()); //A3 = {t, f, t, t}
copy(A3, A3 + 4, ostream_iterator<bool>(cout, " ")); //prints 1, 0, 1, 1
transform(A1, A1 + 4, A2, A2, plus<int>());          //A2 = {2, 6, 5, 7}
transform(A2, A2 + 4, A2, negate<int>());            //A2 = {-2, -6, -5, -7}
copy(A2, A2+4, ostream_iterator<int>(cout," "));     //prints -2, -6, -5, -7
```

Function Objects - Binders

- Οι binders είναι ειδικές συναρτήσεις που μπορούν να τροποποιούν άλλες συναρτήσεις ως προς το πλήθος ή τη σειρά των ορισμάτων τους
  - Μπορούν για παράδειγμα να μετατρέψουν μια συνάρτηση που δέχεται δύο ορίσματα σε μια συνάρτηση που δέχεται ένα όρισμα δίνοντας μια συγκεκριμένη τιμή στο άλλο όρισμα

```
int add(int a, int b) { return a + b; }
add5 = bind1st(add, 5); //add5: unary function, add(3) ➔ 5 + 3 = 8
ten = bind(add, 1, 9);  //ten: nullary function, ten() ➔ 1 + 9 = 10
```

Function Objects - bind1st, bind2nd (1/2)

- template<class AdaptableBinaryFunction > class ***binder1st***, ***binder2nd***;
  - Ορίζεται στο header <functional>
- Είναι *functor* κλάσεις που δέχονται ως όρισμα το function object που είναι να κληθεί καθώς και το σταθερό (bound) όρισμα
  - Το function object πρέπει να είναι αντικείμενο κλάσης που ορίζει τους τύπους των ορισμάτων και της επιστρεφόμενης τιμής
    - Π.χ. για μια unary συνάρτηση θα πρέπει να ορίζονται οι τύποι *argument_type* και *result_type*
  - Για να χρησιμοποιήσουμε απλές συναρτήσεις, τις μετατρέπουμε πρώτα σε function objects χρησιμοποιώντας βοηθητικές συναρτήσεις όπως οι ***ptr_fun***, η ***mem_fun*** και η ***mem_fun1***
- Πρακτικά, χρησιμοποιούμε τις βοηθητικές συναρτήσεις ***bind1st*** και ***bind2nd***

Function Objects - bind1st, bind2nd (2/2)

Χρήση με συνάρτηση

```
int greater(int x, int y) { return x > y; }
typedef std::pointer_to_binary_function<int, int, bool> BinaryFunc;
typedef std::binder1st<BinaryFunc> UnaryFunc;
UnaryFunc positive = std::bind2nd(std::ptr_fun(greater), 0);
std::cout << positive(5); // prints 1
```

Χρήση με member function

```
struct X {
   int x;
   void add(int amount) { x += amount; }
   X(int x = 0) : x(x) {}
};
X* x = new X;
std::bind1st(std::mem_fun1(&X::add), x)(5); //equivalent to x->add(5);
std::list<X *> l; l.push_back(new X(1)); l.push_back(new X(2));
std::for_each(l.begin(), l.end(), std::bind2nd(std::mem_fun1(&X::add), 5));
```

Χρήση με function object

```
struct greater : public std::binary_function<int, int, bool>
   { bool operator()(int l, int r) const { return l > r; } };
typedef std::list<int> IntList;
IntList l;
l.push_back(1); l.push_back(4); l.push_back(3); l.push_back(2);
IntList::iterator r = std::find_if(l.begin(), l.end(), std::bind2nd(greater(), 2));
std::cout << *result;  // prints 4, could also have used std::greater<int>()
```