κεφάλαιο

26

Βιβλιοθήκη Παγίων Περιγραμμάτων – Standard Template Library (STL)

Ο στόχος μας σε αυτό το κεφάλαιο:
Θα κάνουμε μια σύντομη γνωριμία με την STL.
Προσδοκώμενα αποτελέσματα:
Θα μπορείς να χρησιμοποιείς στα προγράμματά σου έτοιμες και καλές λύσεις για πολλά προβλήματα και δεν θα χρειάζεται να γράφεις τα πάντα από την αρχή. Όπως θα καταλάβεις, θα είναι σαν να γράφεις τα προγράμματά σου σε γλώσσα προγραμματισμού πολύ υψηλού επιπέδου.
Έννοιες κλειδιά:

- *περιέχουσα κλάση, περιέχον*
- *προσεγγιστής*
- *αλγόριθμοι*
- *συναρτησιακό αντικείμενο*
- *ακολουθίες*
- *συνειρμικά περιέχοντα*
- *επιλογή περιέχοντος*

Περιεχόμενα:

26.1 Περιέχοντα, Προσεγγιστές και Αλγόριθμοι 991
26.1.1 Περιέχουσες Κλάσεις 992
26.1.2 Περί Προσεγγιστών… 994
26.1.3 Αλγόριθμοι 996
26.1.3.1 Τρεις Συναρτήσεις για «τα Πάντα» 1000
26.1.4 Συναρτησιακά Αντικείμενα 1003
26.2 Ακολουθίες 1005
26.2.1 Το Περίγραμμα “*vector*” 1008
26.2.1.1 Εξαιρέσεις, Απόδοση και Άλλα 1011
26.2.1.2 Και μια Εξειδίκευση: “vector<bool>” 1012
26.2.2 Το Περίγραμμα “*deque*” 1013
26.2.3 Το Περίγραμμα “*list*” 1014
26.2.4 Ποια Ακολουθία να Διαλέξω; 1018
26.3 Συνειρμικά Περιέχοντα 1018
26.3.1 Το Περίγραμμα “*set*” 1020
26.3.1.1 Σχέσεις και Πράξεις Συνόλων 1023
26.3.2 Το Περίγραμμα “*map*” 1027
26.3.3 Τα Περιγράμματα “*multiset*” και “*multimap*” 1030
26.3.4 Διάταξη Στοιχείων 1031
26.4 Ποιο Περιέχον να Διαλέξω; 1033
26.5 Άλλα Περιγράμματα 1034
26.5.1 Το Περίγραμμα “*bitset*” 1034
26.5.2 Το Περίγραμμα “*complex*” 1037
26.6 Τι (Πρέπει να) Έμαθες στο Κεφάλαιο Αυτό 1038

Ασκήσεις **1038**

Εισαγωγικές Παρατηρήσεις:
Η **Βιβλιοθήκη Παγίων Περιγραμμάτων** (Standard Template Library – STL) περιλαμβάνει τέσσερα είδη συνιστωσών:

- *περιέχουσες κλάσεις,*
- *προσεγγιστές,*
- *αλγόριθμους,*
- *συναρτησιακά αντικείμενα.*

Για την υλοποίησή τους χρησιμοποιούνται

- **παραχωρητές μνήμης** (allocators) και
- κλάσεις χαρακτηριστικών.

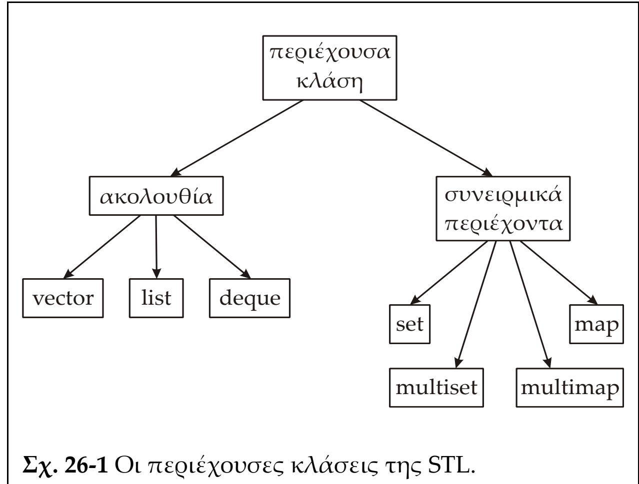
Στα παραδείγματα κλάσεων με δυναμικούς πίνακες βάλαμε έναν μηχανισμό που ήταν παντού ο ίδιος (με ή χωρίς τη *renew()*). Ο πειρασμός να γράψουμε ένα περίγραμμα κλάσης για δυναμικό μονοδιάστατο πίνακα με αυτορυθμιζόμενο μέγεθος είναι μεγάλος! Δεν χρειάζεται: η STL έχει τέτοιο εργαλείο.

Ας δούμε ένα απλό πρόβλημα και ας το λύσουμε με αυτά που ξέρουμε:

Θέλουμε ένα πρόγραμμα που θα διαβάζει από το πληκτρολόγιο άγνωστο πλήθος φυσικών, θα τους ταξινομεί και θα τους δείχνει στην οθόνη.

Χρησιμοποιώντας τη *renew()* από το `MyTmpltLib.h` και την *qsort()* που είδαμε στην §16.10 έχουμε:

```
#include <iostream>
#include <cstdlib>

#include "MyTmpltLib.h"

using namespace std;

int compare( const void* a, const void* b )
{
   return ( *reinterpret_cast<const unsigned int*>(a) -
            *reinterpret_cast<const unsigned int*>(b) );
} // compare

int main()
{
   const unsigned int inc( 5 );
   size_type size( inc );
   unsigned int* array( new unsigned int[inc] );

   int x;
   size_type count( 0 );
   cin >> x;
   while ( x >= 0 )
   {
      if ( count >= size )
      {
         renew( array, count, size+inc );
         size += inc;
      }
      array[count] = x;
      ++count;
      cin >> x;
   } // while
   qsort( array, count, sizeof(int), compare );
   for ( int k(0); k < count; ++k )
      cout << array[k] << endl;
}
```



Σχ. 26-1 Οι περιέχουσες κλάσεις της STL.

Στη συνέχεια θα δούμε διαφορετικές λύσεις που μπορούμε να γράψουμε με τα εργαλεία της STL.

26.1 Περιέχοντα, Προσεγγιστές και Αλγόριθμοι

Ένας άλλος τρόπος να λύσουμε το πρόβλημα που δώσαμε στην εισαγωγή είναι ο εξής:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
   vector< unsigned int > array;

   int x;
   cin >> x;
   while ( x >= 0 )
   {
      array.push_back( x );
      cin >> x;
   } // while
   sort( array.begin(), array.end() );
   size_type count( array.size() );
   for ( int k(0); k < count; ++k )
      cout << array[k] << endl;
}
```

Όπως βλέπεις, αυτό είναι αρκετά απλούστερο από το αρχικό πρόγραμμα.

Το "`vector`" είναι *περιέχουσα κλάση* της STL και εδώ χρησιμοποιούμε ένα στιγμιότυπο της για περιεχόμενο "`unsigned int`".

Ο `array` είναι ένας δυναμικός πίνακας. Πρόσεξε ότι το μόνο σημείο που αναφερόμαστε στο μέγεθος του πίνακα είναι το "`array.size()`" με το οποίο ζητάμε από τον `array` να μας πει το μέγεθός του. Η "`array.push_back(x)`" αντιγράφει την τιμή της x στο τέλος του πίνακα. Αν το μέγεθος του πίνακα δεν είναι αρκετό φροντίζει να τον μεγαλώσει.

Το "`sort`" είναι ένα περίγραμμα συνάρτησης που περιμένει δύο προσεγγιστές που να ορίζουν μια περιοχή μέσα σε κάποιο περιέχον και ταξινομεί τις τιμές μέσα στην περιοχή. Η ταξινόμηση γίνεται με βάση τον τελεστή "`<`" που πρέπει να είναι ορισμένος για τον περιεχόμενο τύπο.

Τα “`array.begin()`” και “`array.end()`” είναι κλήσεις μεθόδων του `array` που μας δίνουν (σταθερούς) *προσεγγιστές* προς την αρχή και το τέλος του. Θυμίσου τις μεθόδους με τα ίδια ονόματα που είδαμε στην *SListT*.

26.1.1 Περιέχουσες Κλάσεις

Όπως λέγαμε και στο προηγούμενο κεφάλαιο (§25.7) «*Περιέχον (container) ή περιέχουσα κλάση (container class) είναι περίγραμμα για κλάση-συλλογή: κάθε αντικείμενο ενός στιγμιοτύπου της είναι συλλογή άλλων αντικειμένων.*» Στη συνέχεια θα χρησιμοποιούμε τον όρο «περιέχον» και για στιγμιότυπα κάποιας περιέχουσας κλάσης.

Μπορούμε να δούμε τα περιέχοντα της STL χωρισμένα σε δύο κατηγορίες (Σχ. 26-1):

- **Ακολουθίες** (sequences) στις οποίες η πρόσβαση σε κάποιο στοιχείο γίνεται με βάση τη θέση του. Στις ακολουθίες ανήκουν τα
  - *vector* (διάνυσμα). Με τη δήλωση “`vector<K> vt;`” ο *vt* δηλώνεται ως δυναμικός μονοδιάστατος πίνακας (διάνυσμα) με στοιχεία τύπου *K*.
  - *list*. Με τη δήλωση “`list<K> lt;`” η *lt* δηλώνεται ως λίστα με διπλή σύνδεση με στοιχεία τύπου *K*.
  - *deque* (*d*ouble *e*nded *que*ue, ουρά δύο άκρων). Μπορεί να γίνεται εισαγωγή και εξαγωγή από τα δύο άκρα.

  και οι προσαρμογείς
  - *priority_queue* (ουρά προτεραιότητας). Γενικώς, τα στοιχεία εισάγονται μαζί με βαθμό προτεραιότητας και εξάγεται πρώτο το στοιχείο με τη μεγαλύτερη προτεραιότητα. Στην STL πρέπει να οριστεί η διάταξη “<” (για την περιεχόμενη κλάση) και πρώτο εξάγεται το «μεγαλύτερο».
  - *queue* (ουρά) FIFO. Επιτρέπει εισαγωγή μόνον στο τέλος της ουράς και εξαγωγή μόνον από την αρχή. Μπορείς να τη δεις σαν ουρά προτεραιότητας όπου τα εισερχόμενα στοιχεία έχουν *φθίνουσα* προτεραιότητα.
  - *stack* (στοίβα) LIFO. Επιτρέπει εισαγωγή και εξαγωγή μόνον από ένα άκρο. Μπορείς να τη θεωρήσεις ουρά προτεραιότητας όπου τα εισερχόμενα στοιχεία έχουν *αύξουσα* προτεραιότητα.
- **Συνειρμικά Περιέχοντα** (associative containers) στα οποία η πρόσβαση σε κάποιο στοιχείο γίνεται με βάση κάποιο κλειδί. Αυτά είναι:
  - *set* (σύνολο).
  - *multiset* (πολυσύνολο).
  - *map* (απεικόνιση), σύνολο ζευγών (κλειδί, υπόλοιπα στοιχεία).
  - *multimap* (πολυαπεικόνιση), πολυσύνολο ζευγών (κλειδί, υπόλοιπα στοιχεία).

Όλα τα παραπάνω περιέχοντα έχουν:

- Ερήμην δημιουργό.
- Δημιουργό αντιγραφής.
- Καταστροφέα.
- *Αντιγραφικό* τελεστή εκχώρησης.
- Μέθοδο *size()* που επιστρέφει το πλήθος στοιχείων της συλλογής.
- Μέθοδο *max_size()* που επιστρέφει το μέγιστο πλήθος στοιχείων που μπορεί να αποθηκευτεί στη συλλογή.
- Μέθοδο *empty()* που επιστρέφει τιμή `true` αν η περιεχόμενη συλλογή δεν έχει στοιχεία.
- Μέθοδο *swap()* για ανταλλαγή τιμών δύο περιεχόντων (του ίδιου τύπου).
- Μέθοδο *get_allocator()* που επιστρέφει αντίγραφο του παραχωρητή μνήμης του περιέχοντος.

Έχουν ακόμη τέσσερις μεθόδους που τις βλέπουμε στη συνέχεια.

Με περιγράμματα καθολικών συναρτήσεων επιφορτώνονται οι

- "==", "!=" (το $a \neq b$ υπολογίζεται ως !(a == b)),
- "<", ">=" ((a >= b) ≡ !(a < b)),
- ">", "<=" ((a > b) ≡ (b < a)).

Η σύγκριση "<" γίνεται *λεξικογραφικώς* (§22.7.3). Για να γίνει αυτό θα πρέπει να έχουμε (εκτός από τον "==") και τον "<" για την περιεχόμενη κλάση.

Κάθε μια από τις περιέχουσες κλάσεις έχουν τις δικές τους *κλάσεις προσεγγιστών*. Πρώτη η *iterator* που είναι σαν αυτήν που είδαμε στην *SListT*. Και αυτή έχει επιφορτωμένους τους "++", "*", "->", "==", "!=". Ακόμη, έχει επιφορτωμένον και τον "--": αυτοί οι προσεγγιστές είναι **αμφίδρομοι** (bidirectional) ενώ της *SListT* ήταν *μονόδρομοι*.

Η δεύτερη κλάση είναι η *const_iterator*. Αν δηλώσεις

```
C<T>::const_iterator it;
```

δεν έχεις δικαίωμα να αλλάξεις την τιμή του "*it".

Δύο μέθοδοι μας δίνουν την αρχή και το τέλος των περιεχομένων της περιέχουσας κλάσης:

- Μέθοδος *begin()*: που επιστρέφει προσεγγιστή προς το πρώτο στοιχείο της περιεχόμενης συλλογής.

```
iterator begin();
const_iterator begin() const;
```

- Μέθοδος *end()*: που επιστρέφει προσεγγιστή προς την πρώτη θέση(!) μετά το τελευταίο στοιχείο της περιεχόμενης συλλογής.

```
iterator end();
const_iterator end() const;
```

(Η δεύτερη μορφή –*const_iterator*– για αντικείμενα-συλλογές "const".) Αν κάποιο περιέχον *a* είναι *κενό* –η *a.empty()* επιστρέφει true– τότε έχουμε *a.begin()* == *a.end()*.

Φυσικά, θα πεις: το «*πρώτο στοιχείο*» και η «*πρώτη θέση(!) μετά το τελευταίο στοιχείο*» μπορεί να έχουν νόημα για πίνακα, μπορεί να έχουν νόημα και για λίστα (όπως είδαμε στα προηγούμενα κεφάλαια, π.χ. §25.7.1)· τι νόημα μπορεί να έχουν για το σύνολο; Θα καταλάβεις στη συνέχεια…

Για οποιαδήποτε περιέχουσα κλάση C της STL, με τη:

```
for ( C<T>::iterator it( co.begin() ); it != co.end(); ++it )
{ /* . . . */ }
```

μπορείς να περάσεις από όλα τα περιεχόμενα αντικείμενα τύπου *T* του αντικειμένου-συλλογή *co*. Αν δεν θέλεις να αλλάζεις τις τιμές των περιεχομένων μπορείς να χρησιμοποιήσεις προσεγγιστή "C<T>::const_iterator" ώστε να κάνεις το πρόγραμμά σου κάπως ταχύτερο.

Ας πούμε ότι η:

```
list< int > il;
```

έχει περιεχόμενο τους 0, 1, 2, 3, 4. Οι

```
for ( list<int>::iterator it( il.begin() );
      it != il.end(); ++it )
   cout << *it << "  ";
cout << endl;
```

θα δώσουν:

```
0  1  2  3  4
```

Δύο άλλες κλάσεις προσεγγιστών είναι οι *reverse_iterator* και *const_reverse_iterator*. Με τέτοιους προσεγγιστές διασχίζεις τη συλλογή από το τέλος προς την αρχή. Δύο μέθοδοι σου δίνουν αρχή και τέλος για τέτοια διάσχιση:

- Μέθοδος *rbegin()*: που επιστρέφει προσεγγιστή προς το τελευταίο στοιχείο της περιεχόμενης συλλογής.

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

- Μέθοδος *rend()*: που επιστρέφει προσεγγιστή προς την πρώτη θέση(!) πριν το πρώτο στοιχείο της περιεχόμενης συλλογής.

```
reverse_iterator rend();
const_reverse_iterator rend() const;
```

Οι εντολές

```
for ( list<int>::reverse_iterator rit( il.rbegin() );
      rit != il.rend(); ++rit )
   cout << *rit << "  ";
cout << endl;
```

για τη λίστα που είδαμε παραπάνω θα δώσουν:

```
4  3  2  1  0
```

Πρόσεξε ότι διασχίζουμε τη λίστα με την "`++rit`".

Αυτά προς το παρόν για τις περιέχουσες κλάσεις. Στη συνέχεια θα δούμε και άλλα που ισχύουν ειδικότερα για συγκεκριμένες κατηγορίες τους.

Υπάρχουν όμως και ορισμένες απαιτήσεις από τον τύπο της *περιεχόμενης* κλάσης. Στην §25.7.2 συνοψίσαμε μερικές από αυτές που επισημάναμε γράφοντας την περιέχουσα κλάση *SListT*:

- ***Ο τύπος περιεχομένου K θα πρέπει να έχει ερήμην δημιουργό.***
- ***Ο τύπος περιεχομένου K θα πρέπει να έχει σωστό δημιουργό αντιγραφής.***
- ***Θα πρέπει να υπάρχει σωστός (αντιγραφικός) τελεστής εκχώρησης για τον περιεχόμενο τύπο K.***
- ***Ο τύπος περιεχομένου K θα πρέπει να έχει καταστροφέα.***

Για ειδικές περιπτώσεις εμφανίζονται και άλλες απαιτήσεις. Ας πούμε,

- ***Αν χρειάζονται αναζητήσεις τιμών ο τύπος περιεχομένου K θα πρέπει να έχει τη δυνατοτητα για συγκρίσεις.***

26.1.2 Περί Προσεγγιστών...

Πριν προχωρήσουμε στην παρουσίαση των περιεχουσών κλάσεων της STL θα πούμε λίγα ακόμη για τους προσεγγιστές.

Η έννοια του προσεγγιστή είναι γενίκευση της έννοιας του βέλους. Με ποιον στόχο; Τη χρήση πάγιων τεχνικών και αλγορίθμων των μονοδιάστατων πινάκων σε άλλες δομές δεδομένων.

Αρχικώς, στην §22.9, γνωρίσαμε τον *πρόσθιο προσεγγιστή*. Σε αυτόν επιφορτώνουμε τους τελεστές "`*`", "`->`", "`++`", "`==`", "`!=`".

Ο *αμφίδρομος προσεγγιστής*, που είδαμε πιο πάνω, έχει όλα τα χαρακτηριστικά του *πρόσθιου προσεγγιστή* αλλά σε αυτόν επιφορτώνουμε και τον "`--`".

Στη συνέχεια θα δούμε τους **προσεγγιστές τυχαίας πρόσβασης** (random access iterators) που δεν έχουν να ζηλέψουν κάτι από τα βέλη: έχουν όλα τα χαρακτηριστικά του αμφίδρομου προσεγγιστή και επιπλέον αριθμητική σαν αυτήν των βελών (§12.3): "*προσεγγιστής θ ακέραιος*" όπου *θ* κάποιος από τους "`+`", "`-`", "`+=`", "`-=`" ή "*προσεγγιστής* – *προσεγγιστής*" (που μας δίνει την «απόσταση» των στοιχείων που δείχνουν οι δύο προσεγγιστές).

Ας δούμε τώρα μια τρίτη μορφή του παραδείγματός μας:

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```
#include <iterator>

using namespace std;

int main()
{
   vector< unsigned int > array;
   istream_iterator< unsigned int > cinIt( cin );
   istream_iterator< unsigned int > cinEoStream;

   while ( cinIt != cinEoStream )
   {
      array.push_back( *cinIt );
      ++cinIt;
   } // while
   sort( array.begin(), array.end() );
   size_type count( array.size() );
   ostream_iterator< unsigned int > coutIt( cout, "  " );
   for ( int k(0); k < count; ++k )
      *coutIt = array[k];
   cout << endl;
}
```

και ένα παράδειγμα εκτέλεσης:

```
19  0  17  3  11  5  7 e
0  3  5  7  11  17  19
```

Βάζοντας στο πρόγραμμα την "`#include <iterator>`" μπορούμε να χρησιμοποιήσουμε το *ostream_iterator* που είναι ένα περίγραμμα που μπορεί να μας δώσει στιγμιότυπα για οποιονδήποτε τύπο *T* για τον οποίον υπάρχει επιφορτωμένος ο "`<<`" προς μορφοποιημένο αρχείο. Ο *coutIt* είναι ένας προσεγγιστής *ostream*. Με τη δήλωση του *coutIt* λέμε ότι θα είναι ένας προσεγγιστής που θα γράφει τιμές τύπου `unsigned int` στο *cout* και μετά από κάθε τιμή θα «γράφει» δύο διαστήματα (`"  "`). Πώς γράφει μια τιμή *v*; Η εντολή: "`*coutIt = v;`" Ισοδυναμεί με "`cout << v << "  ";`". Έτσι, η **for** του προγράμματος βγάζει το αποτέλεσμα που είδες παραπάνω.

Ο *ostream_iterator* είναι το απλούστερο παράδειγμα **προσεγγιστή εξόδου** (output iterator). Ένας προσεγγιστής εξόδου έχει τα εξής χαρακτηριστικά:

- Έχει δημιουργό αντιγραφής, (αντιγραφικό) τελεστή εκχώρησης και καταστροφέα. Ακόμη μπορεί να «αυξηθεί» (π.χ. με τον "`++`").
- Με την αποπαραπομπή του παίρνουμε *τιμή-l* (§11.3). Στην πραγματικότητα η αποπαραπομπή ενός προσεγγιστή εξόδου εμφανίζεται πάντοτε στο αριστερό μέρος μιας εκχώρησης όπως φαίνεται και στο παράδειγμά μας ("`*coutIt = array[k];`").

Όπως καταλαβαίνεις και οι τρεις κατηγορίες προσεγγιστών που είδαμε –πρόσθιοι, αμφίδρομοι, τυχαίας πρόσβασης– έχουν τα χαρακτηριστικά του προσεγγιστή εξόδου.

Όπως πιθανότατα μαντεύεις, υπάρχουν και *προσεγγιστές istream*. Έτσι μπορούμε να διαβάσουμε από κάποιο *istream* ως εξής:

```
   istream_iterator< unsigned int > cinIt( cin );
   istream_iterator< unsigned int > cinEoStream;

   while ( cinIt != cinEoStream )
   {
      array.push_back( *cinIt );
      ++cinIt;
   } // while
```

Ο *cinIt* είναι ένας προσεγγιστής κλάσης `istream_iterator<unsigned int>`: Διαβάζει τιμές τύπου `unsigned int` από ένα ρεύμα *istream*, στην περίπτωσή μας από το *cin*. Δηλώνοντας έναν *istream_iterator* χωρίς ορίσματα –όπως τον *cinEoStream*– παίρνεις έναν προσεγγιστή που όμως «δείχνει μετά το τέλος του ρεύματος τιμών `unsigned int`» και χρησιμοποιείται για τον τερματισμό της ανάγνωσης.

Η τιμή που διαβάζουμε δίνεται με την αποπαραπομπή του προσεγγιστή (`*cinIt`). Ο προσεγγιστής διαβάζει με τον “`>>`”. Προχωρούμε στην επόμενη τιμή με τον τελεστή “`++`”.

Αν φτάσουμε σε τέλος αρχείου ή σε τέλος ρεύματος (δηλαδή –στην περίπτωσή μας– τιμή που δεν είναι `unsigned int`) η τιμή του *cinIt* θα γίνει ίση με αυτήν του *cinEoStream*.

Ο *istream_iterator* είναι το απλούστερο παράδειγμα **προσεγγιστή εισόδου** (input iterator) που έχει τα εξής χαρακτηριστικά:

- Έχει δημιουργό αντιγραφής, (αντιγραφικό) τελεστή εκχώρησης και καταστροφέα. Αν δεν έχει φτάσει στο τέλος ρεύματος, μπορεί να «αυξηθεί» (π.χ. με τον “`++`”). Ακόμη, μπορεί να συγκριθεί με τον “`==`” και τον “`!=`”.
- Με την αποπαραπομπή του παίρνουμε *τιμή-r* (μπορεί να εμφανιστεί μόνον στο δεξιό μερος της εκχώρησης και όχι στο αριστερό).

Τα χαρακτηριστικά του προσεγγιστή εισόδου τα βρίσκουμε και στους προσεγγιστές που είδαμε: εξόδου, πρόσθιους, αμφίδρομους και τυχαίας πρόσβασης.

Βάζοντας στο πρόγραμμά σου “`#include <iterator>`” μπορείς να χρησιμοποιήσεις το περίγραμμα συνάρτησης

```
template < typename InputIterator, typename Distance >
void advance( InputIterator& it, Distance n );
```

που δίνει τη λειτουργικότητα του “`+=`” σε οποιονδήποτε προσεγγιστή εισόδου. Ο *Distance* τι τύπος είναι; Κατ' αρχήν είναι ακέραιος τύπος. Αν ο *InputIterator* είναι τύπος προσεγγιστή αμφίδρομου ή τυχαίας πρόσβασης ο *Distance* μπορεί να είναι και ο `int`. Αλλιώς πρέπει να είναι `unsigned int`.

Η “`advance(it, n);`”

- αν ο *it* είναι τυχαίας πρόσβασης, θα εκτελεσθεί ως “`it += n;`”
- αλλιώς θα εκτελεσθεί με επαναληπτική (*n* φορές) δράση του “`++it;`” ή “`--it;`”.

26.1.3 Αλγόριθμοι

Έχουμε ήδη μάθει ότι βάζοντας στο πρόγραμμά μας “`#include <algorithm>`” μπορούμε να χρησιμοποιήσουμε τα περιγράμματα συναρτήσεων:

- (*std*::)*max()*, (*std*::)*min()*,
- (*std*::)*swap()*,
- (*std*::)*find()*,
- (*std*::)*sort()*.

Στη συνέχεια θα δούμε μερικά ακόμη, αλλά δεν θα παραθέσουμε και τα πάντα. Ψάξε τα εγχειρίδια της C++ που χρησιμοποιείς για να δεις τι υπάρχει.

Εδώ θα δούμε άλλη μια μορφή του παραδείγματός μας χρησιμοποιώντας το περίγραμμα:

```
template< class InputIterator, class OutputIterator >
OutputIterator copy( InputIterator first, InputIterator last,
                     OutputIterator result );
```

που αντιγράφει τα στοιχεία

από την περιοχή [*first*, *last*) στην περιοχή [*result*, *result* + (*last* - *first*)).

Κάνει δηλαδή τις εκχωρήσεις “`*result = *first`”, “`*(result+1) = *(first +1)` ”, κ.ο.κ. με αυτήν τη σειρά. Επιστρέφει ως τιμή το *result* + (*last* - *first*)· δηλαδή έναν προσεγγιστή προς μια «θέση» μετά το τελευταίο στοιχείο που ήλθε από την αντιγραφή.

Δες πώς γράφεται το πρόγραμμά μας:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
```

```
using namespace std;

int main()
{
   vector< unsigned int > array;
   istream_iterator< unsigned int > cinIt( cin );
   istream_iterator< unsigned int > cinEoStream;
   back_insert_iterator< vector<unsigned int> > dest( array );
   copy( cinIt, cinEoStream, dest );
   sort( array.begin(), array.end() );
   ostream_iterator< unsigned int > coutIt( cout, "  " );
   copy( array.begin(), array.end(), coutIt );
   cout << endl;
}
```

Πρόσεξε πώς γίνεται η έξοδος των στοιχείων: με αντιγραφή όλων των στοιχείων του πίνακα –αυτά περιλαμβάνονται στην περιοχή [*array*.*begin*(), *array*.*end*())– στο ρεύμα *cout*. Αυτό είναι το εύκολο.

Η δυσκολία υπάρχει στην ανάγνωση. Θα μπορούσαμε να διαβάσουμε τα στοιχεία με την *copy()*; Να γράψουμε δηλαδή κάτι σαν:

```
copy( cinIt, cinEoStream, array.begin() );
```

Όχι! Εδώ έχουμε πρόβλημα: Ο *array* δεν έχει την απαιτούμενη μνήμη και –γενικώς– δεν ξέρουμε πόσα είναι τα στοιχεία για να την κρατήσουμε πριν ζητήσουμε την αντιγραφή.

Η λύση είναι η εξής:

```
back_insert_iterator< vector<unsigned int> > dest( array );
copy( cinIt, cinEoStream, dest );
```

Ο *back_insert_iterator* είναι ένας *προσαρμογέας*: από τη μια είναι ένας προσεγγιστής, όπως τον θέλει η *copy()*, από την άλλη φροντίζει να παίρνει την απαραίτητη μνήμη ώστε να γίνεται η εισαγωγή μιας τιμής στον *array* χωρίς πρόβλημα.

Πριν προχωρήσουμε, αξίζει να «θαυμάσουμε» αυτή τη μορφή του προγράμματος που μοιάζει με ψευδοκώδικα! copy από την είσοδο – sort – copy στην έξοδο! Ή, αλλιώς: έχουμε προγραμματισμό σε ψηλότερο επίπεδο!

Παρατηρήσεις:▶

1. Η STL έχει περιγράμματα συναρτήσεων που δρουν σε *περιοχές* ενός περιέχοντος (ή σε *ολόκληρο* περιέχον). Δηλαδή, για ένα πρόβλημα σαν αυτό που είδαμε εδώ, η αντιπροσωπευτική λύση είναι η τελευταία και όχι οι προηγούμενες με τις **while** και **for**.
2. Όπως χρησιμοποιείς τα περιέχοντα της STL μπορείς να χρησιμοποιείς και τα περιέχοντα της C++ –δηλαδή τους πίνακες– και περιέχοντα δικά σου. Δες μια άλλη μορφή του παραδείγματος που είδαμε πιο πάνω:

```
#include <algorithm>
#include <iterator>
#include <iostream>

using namespace std;

int main()
{
   int  array[100];
   int* pEnd;

   pEnd = copy( istream_iterator<int>(cin), istream_iterator<int>(),
                array );
   sort( array, pEnd );
   copy( array, pEnd, ostream_iterator<int>(cout, "\n") );
}
```

Εδώ, περιέχον είναι ένας συνήθης πίνακας. Ως προσεγγιστές χρησιμοποιούμε δύο συμβατικά βέλη, τα `array` και `pEnd`. Κατά τα άλλα χρησιμοποιούμε τα περιγράμματα *copy()* και *sort()*.◀

Πριν προχωρήσουμε είναι καλό να ρίξουμε μια ματιά στις προδιαγραφές των περιγραμμάτων που ήδη είδαμε και χρησιμοποιούμε.

Ξεκινούμε από τα *min()* και *max()* που δίνονται με δύο μορφές:

```
template< class T >
const T& min( const T& a, const T& b );
template< class T, class Compare >
const T& min( const T& a, const T& b, Compare comp );
```

και

```
template< class T >
const T& max( const T& a, const T& b );
template< class T, class Compare >
const T& max( const T& a, const T& b, Compare comp );
```

Για τις απλές περιπτώσεις ο τύπος *T* θα πρέπει να έχει

- τον τελεστή "<" και
- δημιουργό αντιγραφής.

Έτσι, για παράδειγμα η *min()* μπορεί να είναι:

```
template < class T >
inline const T& min( const T& a, const T& b )
{  return b < a ? b : a;  }
```

Οι μορφές με τις τρεις παραμέτρους μπορεί να χρησιμοποιηθούν όταν ο *T* δεν έχει τον "<" ή θέλουμε να κάνουμε σύγκριση με άλλον τρόπο· αρκεί να δοθεί η *comp()* που μας λέει ποια από δύο τιμές *T* είναι η «μικρότερη» ή «προηγείται». Στην περίπτωση αυτήν η *min()* μπορεί να είναι:

```
template < class T, class Compare >
inline const T& min( const T& a, const T& b, Compare comp )
{  return comp(b, a) ? b : a;  }
```

Επομένως η *Compare* είναι ένα συναρτησοειδές, δηλαδή κλάση στην οποίαν έχουμε επιφορτώσει τον τελεστή "()" ώστε να δέχεται δύο παραμέτρους τύπου *T* και να επιστρέφει `true` αν η πρώτη είναι η «μικρότερη».

Αν έχεις να υπολογίσεις τον ελάχιστο από δύο ακέραιους χρησιμοποιείς την απλή μορφή: "`min( i1, i2 )`".

Αν όμως έχεις να συγκρίνεις τα `"abc"` και `"aggr"` θα χρειαστείς τη δεύτερη.[1] Για να τη χρησιμοποιήσεις έχεις δύο επιλογές:

- Να γράψεις κλάση που να ταιριάζει με την "`class Compare`":

```
struct CStrLT
{
   bool operator()( const char cs1[], const char cs2[] )
   {  return strcmp(cs1, cs2) < 0;  }
}; // CStrLT
```

  και στη συνέχεια να δώσεις "`min("abc", "aggr", cstrLT)`" όπου το *cstrLT* είναι (συναρτησιακό) αντικείμενο κλάσης *CStrLT*.

- Να γράψεις συνάρτηση (κατηγόρημα):

```
bool cstrLTf( const char cs1[], const char cs2[] )
{  return strcmp(cs1, cs2) < 0;  }
```

  και στη συνέχεια να δώσεις "`min("abc", "aggr", cstrLTf)`".

[1] Βέβαια μπορείς να βολευτείς με την απλή μορφή αν γράψεις "`min( string("abc"), string( "aggr") )`".

Παρόμοια ισχύουν και για τη *max()* αλλά προσοχή: μη θεωρήσεις ότι εδώ θα πρέπει να ασχοληθείς με τον “`>`”· και πάλι ο “`<`” θα κάνει τη δουλειά μας. Για παράδειγμα η “`max("abc", "aggr", cstrLTf)`” θα μας δώσει `"aggr"`.

Ας έλθουμε τώρα στο περίγραμμα:

```
template< class InputIterator, class T >
InputIterator find( InputIterator first, InputIterator last,
                    const T& value );
```

Στον τύπο *T* πρέπει να είναι δυνατές οι συγκρίσεις “`==`” και “`!=`”.

Πάντως υπάρχει και το περίγραμμα *find_if()* που είναι γενίκευση του *find()*:

```
template< class InputIterator, class Predicate >
InputIterator find_if( InputIterator first, InputIterator last,
                       Predicate pred );
```

Μας επιστρέφει προσεγγιστή *it* προς την πρώτη τιμή για την οποίαν το `pred(*it)` θα επιστρέψει `true`. Αν δεν βρει τέτοια τιμή επιστρέφει *last*.

Για το *pred* έχουμε τις ίδιες επιλογές που είχαμε και για το *comp* προηγουμένως αλλά το *pred* είναι κατηγόρημα με μια παράμετρο.

Και η *sort()* έχει δύο μορφές:

```
template< class RandomAccessIterator >
void sort( RandomAccessIterator first, RandomAccessIterator last );
template< class RandomAccessIterator, class Compare >
void sort( RandomAccessIterator first, RandomAccessIterator last,
           Compare comp );
```

Όπως λέει και το όνομα της κλάσης-παραμέτρου πρέπει να έχουμε προσεγγιστές τυχαίας πρόσβασης. Στην πρώτη περίπτωση η ταξινόμηση γίνεται με βάση τον τελεστή “`<`” ενώ στη δεύτερη με βάση το κατηγόρημα (δύο παραμετρων) *comp()* για το οποίο ισχύουν αυτά που είπαμε πιο πάνω.

Τώρα θα πεις, αφού μας δίνει εργαλείο για ταξινόμηση δεν θα μας δώσει και ένα εργαλείο για δυαδική αναζήτηση; Μας δίνει και όχι μόνον ένα.

```
template< class ForwardIterator, class T >
bool binary_search( ForwardIterator first, ForwardIterator last,
                    const T& value );
template< class ForwardIterator, class T, class Compare >
bool binary_search( ForwardIterator first, ForwardIterator last,
                    const T& value, Compare comp );
```

Εδώ έχεις μια απογοήτευση: αν βρεθεί στο [*first*, *last*) κάποιος *it* τέτοιος που να έχουμε `*`*it* `==` *value* η *binary_search()* θα σου επιστρέψει `true`. Αυτό συνήθως δεν είναι αρκετό.

Ας δούμε μιαν άλλη συνάρτηση (δηλαδή *δύο άλλες*):

```
template< class ForwardIter, class T >
ForwardIter lower_bound( ForwardIter first, ForwardIter last,
                         const T& value );
template< class ForwardIter, class T, class Compare >
ForwardIter lower_bound( ForwardIter first, ForwardIter last,
                         const T& value, Compare comp );
```

Ο προσεγγιστής που επιστρέφεται –ας τον πούμε *rv*– δείχνει την πρώτη θέση στην οποία μπορεί να εισαχθεί η *value* χωρίς να χαλάσει η ταξινόμηση· δείχνει το πρώτο στοιχείο που δεν είναι μικρότερο από *value*. Δηλαδή για κάθε προσεγγιστή *it* στο [*first*, *rv*) έχουμε `*`*it* `<` *value*. Αν *rv* `!=` *last* για κάθε *it* στο [*rv*, *last*) έχουμε `*`*it* `>=` *value* (ακριβέστερα: `!(*`*it* `<` *value*`)`.)

Αν χρησιμοποιήσεις τη δεύτερη μορφή τότε στο [*first*, *rv*) έχουμε `comp(*it, value)` (`== true`) ενώ στο [*rv*, *last*) έχουμε `!comp(*it, value)`.

Αν η αναζητούμενη τιμή υπάρχει στο [*first*, *last*) *τότε θα τη δείχνει ο rv δηλαδή θα έχουμε:* `*`*rv* `==` *value* (ακριβέστερα: `!(*`*rv* `<` *value*`) && !(`*value* `< *`*rv*`)`.)

Παρατηρήσεις:▶

1. Και οι τέσσερις συναρτήσεις έχουν πολυπλοκότητα ανάλογη του *log*(*last*–*first*).

2. Η *comp* πρέπει να είναι ίδια (γενικότερα: συμβατή) με αυτήν που χρησιμοποιήθηκε για την ταξινόμηση. Αλλιώς η αναζήτηση δεν γίνεται σωστά.
3. Υπάρχει και ένα ζεύγος περιγραμμάτων συναρτήσεων *upper_bound()* που επιστρέφουν προσεγγιστή προς τη μικρότερη τιμή που είναι μεγαλύτερη από αυτήν που ψάχνουμε.◀

26.1.3.1 Τρεις Συναρτήσεις για «τα Πάντα»

Πριν προχωρήσουμε στα συναρτησιακά αντικείμενα θα πούμε λίγα λόγια για τρία περιγράμματα συναρτήσεων που *δρουν σε όλα τα αντικείμενα μιας περιοχής*: πρόκειται για τις *for_each()*, *transform()* και *random_shuffle()*.

Η συνάρτηση

```
template< class InputIterator, class Function >
Function for_each( InputIterator first, InputIterator last, Function f );
```

σαρώνει την περιοχή [*first*, *last*) με έναν προσεγγιστή (*InputIterator*) *it* και για κάθε τιμή του υπολογίζει το "`f(*it)`". Αν η *f()* δεν είναι `void` το αποτέλεσμα που επιστρέφει αγνοείται.

Τα αντικείμενα κλάσης *Function* έχουν επιφορτωμένο τον "`()`" (συναρτησιακά αντικείμενα) έτσι ώστε να δέχονται μοναδική παράμετρο τα αντικείμενα που προκύπτουν από την αποπαραπομπή προσεγγιστών κλάσης *InputIterator*. Το όνομα της κλάσης-παραμέτρου μας δείχνει ότι τα χαρακτηριστικά των προσεγγιστών: είναι *προσεγγιστές εισόδου*. Αν πάρουμε υπόψη μας ότι η αποπαραπομπή τους είναι *τιμή-r* καταλαβαίνουμε ότι *η* "`f(*it)`" *δεν μπορεί να αλλάζει την τιμή του αντικειμένου που δείχνει ο it*.

Η *for_each()* επιστρέφει το συναρτησιακό αντικείμενο *f* όπως αυτό διαμορφώθηκε μετά την εφαρμογή σε όλα τα αντικείμενα της περιοχής.

Το παρακάτω πρόγραμμα

```
#include <iostream>
#include <algorithm>

using namespace std;

template< typename T >
struct Print
{
   void operator()( T x )
   {  cout << x << ' ';  }
}; // Print

int main()
{
   int        intArr[] = { 34, 72, -23, 8, -11, 31, 41 };
   Print<int> pr;

   for_each( intArr, intArr + 7, pr );
   cout << endl;
}
```

θα δώσει:

```
34 72 -23 8 -11 31 41
```

Θα μπορούσαμε να πάρουμε τα ίδια αποτελέσματα με τις:

```
   for_each( intArr, intArr+7, printF<int> );
   cout << endl;
```

όπου

```
template< typename T >
void printF( T x )  {  cout << x << ' ';  }
```

Για να δεις πώς μπορούμε να χρησιμοποιήσουμε αυτό που επιστρέφει η *for_each()* αλλάζουμε λίγο την *Print* (SGI 1999):

```
template< typename T >
struct Print
```

```
{
   Print() :  count(0) {};
   void operator()( T x )
   {  cout << x << ' ';
      ++count;  }
   unsigned int count;
}; // Print
```

και τον τρόπο που την καλούμε:

```
   pr = for_each( intArr, intArr+7, pr );
   cout << endl << pr.count << " numbers printed" << endl;
```

Αποτέλεσμα:

```
34 72 -23 8 -11 31 41
7 numbers printed
```

Αν θέλεις να αλλάξεις «τα πάντα» θα χρησιμοποιήσεις ένα από τα δύο περιγράμματα:

```
template< class InputIter, class OutputIter,
          class UnaryOperation >
OutputIterator transform( InputIter first, InputIter last,
                          OutputIter result, UnaryOperation op );
template< class InputIter1, class InputIter2,
          class OutputIter, class BinaryOperation >
OutputIterator transform( InputIter1 first1, InputIter1 last1,
                          InputIter2 first2, OutputIter result,
                          BinaryOperation binary_op );
```

Στα στιγμιότυπα της πρώτης μορφής σαρώνεται η περιοχή [*first*, *last*) με έναν προσεγγιστή (*InputIterator*) *it* και για κάθε τιμή υπολογίζεται το `op(*it)` που φυλάγεται σε διαδοχικές θέσεις ξεκινώντας από αυτήν που δείχνει ο *result*. Ο *result* μπορεί να δείχνει την ίδια θέση που δείχνει και ο *first*. Επιστρέφει προσεγγιστή προς την πρώτη θέση μετά την τελευταία όπου αποθηκεύτηκε αποτέλεσμα.

Ένα απλό παράδειγμα χρήσης της πρώτης μορφής είναι το παρακάτω:

```
#include <iostream>
#include <algorithm>

using namespace std;

template< class T >
struct Sqr
{
   T operator()( T x )  {  return x*x;  }
}; // Sqr

int main()
{
   int      intArr[] = { 34, 72, -23, 8, -11, 31, 41 };
   int      intArr1[7];
   Sqr<int> sqr;

   copy( intArr, intArr+7, ostream_iterator<int>(cout, " ") );
   cout << endl;
   int* res1( transform(intArr, intArr+7, intArr1, sqr) );
   copy( intArr1, intArr1+7, ostream_iterator<int>(cout, " ") );
   cout << endl << (res1-intArr1) << endl;

   int* res( transform(intArr, intArr+7, intArr, sqr) );
   copy( intArr, intArr+7, ostream_iterator<int>(cout, " ") );
   cout << endl << (res-intArr) << endl;
}
```

Αποτέλεσμα:

```
34 72 -23 8 -11 31 41
1156 5184 529 64 121 961 1681
7
1156 5184 529 64 121 961 1681
```

7

Το συναρτησιακό αντικείμενο *Sqr*[2] υπολογίζει και επιστρέφει το τετράγωνο της μοναδικής παραμέτρου του.

Η πρώτη κλήση της *transform()* έχει ως αποτέλεσμα να υπολογιστούν τα τετράγωνα των στοιχείων του `intArr` και να φυλαχτούν στα αντίστοιχα στοιχεία του `intArr1`. Ο προσεγγιστής (στην περίπτωσή μας απλό βέλος) που επιστρέφει (*res1*) απέχει από την αρχή του `intArr1` 7 θέσεις τύπου `int`.

Στη δεύτερη κλήση της *transform()* κάθε τετράγωνο που υπολογίζεται αντικαθιστά το αρχικό στοιχείο του `intArr`. Ο προσεγγιστής που επιστρέφει (*res*) απέχει από την αρχή του `intArr` 7 θέσεις τύπου `int`.

Πρόσεξε τώρα τη δεύτερη μορφή: το συναρτησιακό αντικείμενο που έχουμε εδώ έχει δύο παραμέτρους και δρα στα αντίστοιχα στοιχεία δύο περιοχών με ίσα μήκη. Το αποτέλεσμα κατά τη φύλαξή του μπορεί να αντικαταστήσει είτε την πρώτη περιοχή είτε τη δεύτερη (είτε καμία από τις δύο). Στις παραμέτρους υπάρχει μια «ασυμμετρία»:

- Για την πρώτη περιοχή μας δίνεται αρχή και τέλος: [*first1*, *last1*).
- Για τη δεύτερη περιοχή μας δίνεται μόνο η αρχή *first2* και το τέλος συνάγεται από το ότι θα πρέπει να έχει το ίδιο μήκος με την πρώτη.

Αν στο προηγούμενο παράδειγμα ορίσουμε επί πλέον

```
template< typename T >
struct Sum
{
   T operator()( T x, T y )  {  return x + y;  }
};
```

και δηλώσουμε:

```
   Sum<int> sum;
   int      intArr2[7];
```

τότε οι:

```
   transform( intArr, intArr+7, intArr1, intArr2, sum );
   copy( intArr2, intArr2+7, ostream_iterator<int>(cout, " ") );
   cout << endl;
```

θα μας δώσουν:

```
1190 5256 506 72 110 992 1722
```

Όπως βλέπεις, το 1190 είναι 34+1156, 5256 == 72+5184 κ.ο.κ. Δηλαδή, η *sum* καλείται 7 φορές για να υπολογίσει τα `intArr[k] + intArr1[k]`. Το αποτέλεσμα φυλάγεται στο `intArr2[k]`. Θα μπορούσαμε, αντί για `intArr2`, να είχαμε βάλει `intArr` ή `intArr1`.

Η τρίτη «συνάρτηση για τα πάντα» είναι η *random_shuffle()* που έχει δύο μορφές:

```
template< class RandomAccessIter >
void random_shuffle( RandomAccessIter first, RandomAccessIter last);
template< class RandomAccessIter, class RandomNumberGenerator >
void random_shuffle( RandomAccessIter first, RandomAccessIter last,
                     RandomNumberGenerator& rand );
```

Αυτή ανακατεύει με τυχαίο τρόπο τις τιμές που δείχνουν οι προσεγγιστές της περιοχής [*first1*, *last1*). Στη δεύτερη μορφή έχεις τη δυνατότητα να ορίσεις δική σου γεννήτρια τυχαίων αριθμών με μορφή συναρτησιακού αντικειμένου *rand*, με επιφορτωμένο τον "`()`", τέτοια ώστε η "`rand(n)`" να επιστρέφει τυχαίο ακέραιο στο [0..*n*).

Ως παράδειγμα δες το ανακάτεμα της τράπουλας:

```
#include <iostream>
#include <algorithm>

using namespace std;
```

[2] Δεν είναι απαραίτητο να είναι περίγραμμα.

```
int main()
{
   char* cardDeck[52] = { "S2","S3","S4","S5","S6","S7","S8","S9","S10","SJ",
                                                   "SQ","SK","SA",
                          "H2","H3","H4","H5","H6","H7","H8","H9","H10","HJ",
                                                   "HQ","HK","HA",
                          "D2","D3","D4","D5","D6","D7","D8","D9","D10","DJ",
                                                   "DQ","DK","DA",
                          "C2","C3","C4","C5","C6","C7","C8","C9","C10","CJ",
                                                   "CQ","CK","CA" };
   random_shuffle( cardDeck, cardDeck+52 );
   for ( int k(0); k < 52; ++k )
   {
      cout << cardDeck[k] << ' ';
      if ( k%13 == 12 )  cout << endl;
   }
}
```

Αποτέλεσμα:[3]

```
DK SK D5 HQ H7 D3 C4 CQ H9 H4 S6 D8 C10
S7 C9 C2 D9 DA H3 D10 CJ S10 H10 CA S9 H8
C3 SA C6 SJ H2 H5 HJ D6 D7 CK S2 C7 DJ
HA D2 H6 C8 S4 C5 S8 S3 SQ DQ HK S5 D4
```

26.1.4 Συναρτησιακά Αντικείμενα

Στη συνέχεια θα δούμε το περίγραμμα κλάσης “*set*” που είναι:

```
template < class Key, class Compare = less<Key>,
           class Allocator = allocator<Key> >
class set;
```

Αφού στα περιγράμματα κλάσεων δεν μπορούμε να έχουμε επιφόρτωση, όπως έχουμε στα περιγράμματα συναρτήσεων, αλλά μπορούμε να έχουμε ερήμην «τιμές» των παραμέτρων του περιγράμματος, αντί να έχουμε δύο περιγράμματα “*set*”, όπως έχουμε δύο περιγράμματα *sort()*, έχουμε αυτό το “`class Compare = less<Key>`” που μας λέει ότι αν γράψω “`set<T>`” εννοώ ότι οι συγκρίσεις θα γίνονται με το συναρτησιακό αντικείμενο `less<T>` όπου (§25.6.3):

```
template < typename T >
struct less
{
   bool operator()( const T& x, const T& y ) const
   {  return x < y;  }
};
```

Βάζοντας στο πρόγραμμά σου “`#include <functional>`” έχεις τη δυνατοτητα να χρησιμοποιείς συναρτησιακά αντικείμενα σαν το παραπάνω και τα άλλα παρόμοια περιγράμματα που υπάρχουν εκεί.

Όλα ξεκινούν με τα[4]

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

template < typename Arg1, typename Arg2, typename Result >
```

---

[3] Πού είναι η τράπουλα; Βλέπε το "Sx" ως "♠x", το "Hx" ως "♥x", το "Dx" ως "♦x" και το "Cx" ως "♣x".

[4] Στο C++11 αποθαρρύνεται η χρήση των *unary_function*, *binary_function* (που μπορεί να μην υπάρχουν σε επόμενες τυποποιήσεις).

```
struct binary_function
{
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Result result_type;
}; // binary_function
```

Τα άλλα περιγράμματα κληρονομούν αυτά τα δύο, για παράδειγμα:

```
template < class T >
struct negate : public unary_function< T, T >
{
   T operator() ( const T& x ) const { return -x; }
};

template < class T >
struct plus : public binary_function<T, T, T>
{
   T operator() ( const T& x, const T& y ) const { return x + y; }
};

template < typename T >
struct less : public binary_function< T, T, bool >
{
   bool operator()( const T& x, const T& y ) const
   {  return x < y;  }
};
```

- Το *binary_function* κληρονομούν τα
  - *plus()* ("+"), *minus()* ("-"), *multiplies()* ("*"), *divides()* ("/"), *modulus()* ("%")
  - *equal_to()* ("=="), *not_equal_to()* ("!="), *greater()* (">"), *less()* ("<"), *greater_equal()* (">="), *less_equal()* ("<=")
  - *logical_and()* ("&&"), *logical_or()* ("||")
- Το *unary_function* κληρονομούν τα
  - *negate()* ("-")
  - *logical_not()* ("!")

Γυρνώντας στο παράδειγμα για τη δεύτερη μορφή της *transform()* βλέπουμε ότι δεν χρειάζεται να γράψουμε το περίγραμμα *Sum*. Βάζουμε στο πρόγραμμα την "`#include <functional>`", αλλάζουμε τη δήλωση του *sum* σε:

```
   plus<int> sum;
```

και παίρνουμε τα ίδια αποτελέσματα.

Τα εργαλεία αυτά μας βοηθούν να χρησιμοποιούμε τις συναρτήσεις του `algorithm` και τα περιέχοντα της STL. Για τον λόγο αυτόν μας δίνονται και διάφοροι προσαρμογείς. Με ένα παράδειγμα ας δούμε εργαλεία που μας δίνονται ώστε να χρησιμοποιούμε συναρτησεις με δύο παραμέτρους εκεί που απαιτείται συνάρτηση με μια.

Παράδειγμα

Στο Project04, θέλουμε να εφοδιάσουμε την κλάση *CourseCollection* με μια μέθοδο:

```
Course* CourseCollection::findDep( string code )
```

που θα μας επιστρέφει βέλος προς στοιχείο (τύπου *Course*) που έχει ως προαπαιτούμενο (*prereq*) το στοιχείο με κωδικό *code*. Αν δεν υπάρχει τέτοιο στοιχείο θα επιστρέφει "`0`" (*NULL*).

Ένας τρόπος να λύσουμε το πρόβλημα είναι αυτός που έχουμε στη μέθοδο *delete1-Course()*· εδώ όμως θέλουμε λύσουμε το πρόβλημα χρησιμοποιώντας το

```
template< class InputIterator, class Predicate >
InputIterator find_if( InputIterator first, InputIterator last,
                       Predicate pred );
```

Το στιγμιότυπο που μας ενδιαφέρει θα είναι:

```
Course* find_if( ccArr, ccArr+ccNOfCourses, pred???? );
```

Τι θα είναι εκείνο το “`pred????`”; Εμείς ψάχνουμε ένα στοιχείο *s* για το οποίο “`s.prereq == code`”. Αυτό θα μπορούσαμε να το κωδικοποιήσουμε με ένα κατηγόρημα:

```
struct EqPrereq
{
   bool operator()( const Course& x, const string& y ) const
   {  return ( strcmp(x.getPrereq(), y.c_str()) == 0 );  }
}; // EqPrereq
```

Αλλά εδώ έχουμε ένα κατηγόρημα με δύο παραμέτρους ενώ το “`pred????`” πρέπει να έχει μια παράμετρο που θα αντικαθίσταται με τα `ccArr[0]`, `ccArr[1]` κ.ο.κ. Αυτό μπορεί να προκύψει από το *EqPrereq* ως εξής:

```
bind2nd(EqPrereq(),code)
```

που σημαίνει: συνάρτηση με μια παράμετρο που προκύπτει από την *EqPrereq()* αν βάλουμε τη δεύτερη παράμετρο ίση με “`code`”.

Για να δουλέψει η *bind2nd()* θα πρέπει το (πρώτο) όρισμά της να κληρονομεί ένα στιγμιότυπο της *binary_function*. Γράφουμε λοιπόν:

```
struct EqPrereq :  binary_function< Course, string, bool >
{
   bool operator()( const Course& x, const string& y ) const
   {  return ( strcmp(x.getPrereq(), y.c_str()) == 0 );  }
}; // EqPrereq
```

Η συνάρτηση που θέλουμε να γράψουμε θα είναι:

```
Course* CourseCollection::findDep( string code )
{
   Course* k;
   k = find_if( ccArr, ccArr+ccNOfCourses, bind2nd(EqPrereq(),code) );
   if ( k == ccArr+ccNOfCourses )  k = 0;
   return k;
} // findDep
```

❦❦❦

Εκτός από το περίγραμμα *bind2nd()* υπάρχει και το *bind1st()* που μας επιτρέπει να «καρφώσουμε» το πρώτο όρισμα. Τα δύο περιγράμματα στηρίζονται στα περιγράμματα-προσαρμογείς *binder2nd* και *binder1st* αντιστοίχως. Να επαναλάβουμε ότι για να χρησιμοποιήσουμε τα περιγράμματα συναρτήσεων σε μια συνάρτηση *f()* με δύο παραμέτρους θα πρέπει η αντίστοιχη συναρτησιακή κλάση *f* να κληρονομεί την *binary_function*.[5]

26.2 Ακολουθίες

Όπως είπαμε παραπάνω στις *ακολουθίες* η πρόσβαση σε κάποιο στοιχείο γίνεται με βάση τη θέση του. Έτσι:

- Ένα στιγμιότυπο της *vector* είναι κλάση μονοδιάστατων δυναμικών πινάκων. Αν έχουμε “`vector<K> vt`” τότε το “`vt[j]`” μας δίνει πρόσβαση στο αντίστοιχο στοιχείο του πίνακα με την ίδια ευκολία για οποιαδήποτε (νόμιμη) τιμή του “`j`”.
- Ένα στιγμιότυπο της *list* είναι κλάση που κάθε της αντικείμενο είναι λίστα με διπλή σύνδεση. Αν έχουμε “`list<K> lt`” μπορείς να αρχίσεις να διασχίζεις την *lt* από την αρχή της (με τον *lt.begin()*) ή από το τέλος της (με τον *lt.end()*). Αν έχεις κάποιον προσεγγιστή *it* προς κάποιο στοιχείο της λίστας μπορείς να τον μετακινήσεις προς το επόμενο στοιχείο (με την “`++it`”) ή προς το προηγούμενο (με την “`--it`”) αν, φυσικά, υπάρχουν.

---

[5] Στο C++11 αποθαρρύνεται η χρήση των *bind2nd()*, *bind1st()*, *binder2nd*, *binder1st* όπως και της *binary_function* (που μπορεί να μην υπάρχουν σε επόμενες τυποποιήσεις).

- Ένα στιγμιότυπο της *deque* είναι κλάση που κάθε της αντικείμενο είναι ουρά δύο άκρων. Η λειτουργικότητα ενός τέτοιου αντικειμένου είναι παρόμοια με αυτήν ενός αντικειμένου *vector*, έχουμε όμως και τη δυνατότητα να εισάγουμε και να εξάγουμε αντικείμενα της περιεχόμενης κλάσης από την αρχή.

Όλες οι ακολουθίες `S<K>` έχουν –πέρα από αυτά που είπαμε για τα περιέχοντα γενικώς– και άλλες συναρτήσεις-μέλη που είναι:

- **Δημιουργοί πλήρωσης** (fill constructors).[6]

  ```
  explicit S<K>( size_type sz, const K& val = K() );
  ```

  Με τη δήλωση:

  ```
  S<K> a( n, c );
  ```

  –όπου $n \geq 0$ και *c* αντικείμενο κλάσης *K*– το *a* περιέχει *n* αντίγραφα του *c*. Για παράδειγμα, μετά την

  ```
  list< Date >  ld( 3, Date(2010, 1, 1) );
  ```

  η *ld* είναι λίστα με τρία αντίγραφα της "`1.1.2010`". Παρόμοια λίστα (χωρίς όνομα) δημιουργείται από την παράσταση:

  ```
  list< Date >( 3, Date(2010, 1, 1) )
  ```

  Με τη δήλωση:

  ```
  S<K> a( n );
  ```

  το *a* περιέχει *n* αντίγραφα του *K()*. Έτσι, αν δηλώσουμε:

  ```
  vector< Date >  vd( 5 );
  ```

  ο *vt* είναι πίνακας με πέντε αντίγραφα της "`1.1.1`". Παρόμοιος πίνακας (χωρίς όνομα) δημιουργείται και από την παράσταση: "`vector<Date>(5)`"

- **Δημιουργοί περιοχής** (range constructors).

  ```
  template < class InputIterator >
  S<K>( InputIterator first, InputIterator last );
  ```

  Με τη δήλωση:

  ```
  S<K> a( j, k );
  ```

  –όπου τα *j*, *k* είναι προσεγγιστές σε κάποιο άλλο περιέχον (ή απλά βέλη προς στοιχεία απλού πίνακα) και η `[`*j*, *k*`)` είναι μια μη κενή περιοχή με αντικείμενα κλάσης *K*– το *a* περιέχει αντίγραφα όλων των αντικειμένων της περιοχής. Για παράδειγμα, μετά τις

  ```
  int v0[10] = { 0,1,2,3,4,5,6,7,8,9 };
  int* p1( &v0[5] );  int* p2( &v0[8] );
  deque< int > dqt( p1, p2 );
  ```

  η *dqt* περιέχει 3 στοιχεία: τα "`5`", "`6`", "`7`".

  Ακόμη, στις τρεις «νέες» μορφές του παραδείγματος με το οποίο ξεκινήσαμε, μπορείς να προσθέσεις άλλη μια όπου η ανάγνωση των στοιχείων γίνεται με τη δήλωση του πίνακα: αφού δηλώσεις τους δύο *istream_iterator*s χρησιμοποίησε αυτόν τον δημιουργό για να δηλώσεις τον `array`:

```
    istream_iterator< unsigned int > cinIt( cin );
    istream_iterator< unsigned int > cinEoStream;
    vector< unsigned int > array( cinIt, cinEoStream );
    sort( array.begin(), array.end() );
//  . . .
```

- Μέθοδοι για την **κεφαλή** (head) του περιέχοντος:

  ```
  K& front();
  const K& front() const;
  ```

  (η δεύτερη για αντικείμενα `const`.) Επιστρέφει (αναφορά προς) το πρώτο αντικείμενο (κεφαλή) του περιέχοντος. Με άλλα λόγια το: `a.front()` είναι ίσο με `*(a.begin())`.

[6] Παραλείπουμε την τρίτη παράμετρο (παραχωρητής μνήμης).

- Μέθοδοι εισαγωγής στοιχείων:

```
S<K>::iterator insert( S<K>::iterator pos, const K& val );
void insert( S<K>::iterator pos, size_type sz, const K& val );
template < class InputIterator >
void insert( S<K>::iterator pos,
             InputIterator first, InputIterator last );
```

  Η παράμετρος "`S<K>::iterator pos`" πρέπει να είναι προσεγγιστής στο περιέχον που κάνουμε την εισαγωγή.

  Η πρώτη μέθοδος εισάγει αντίγραφο της *val* πριν από το στοιχείο που δείχνει ο *pos* και αυξάνει το *size()* κατά 1. Επιστρέφει προσεγγιστή που δείχνει το στοιχείο που εισάχθηκε.

  Η δεύτερη εισάγει *sz* αντίγραφα της *val* πριν από το στοιχείο που δείχνει ο *pos*· αυξάνει το *size()* κατά *sz*.

  Η τρίτη (περίγραμμα) εισάγει ολόκληρη την περιοχή [*first*, *last*) πριν από το στοιχείο που δείχνει ο *pos*· αυξάνει το *size()* κατά *last - first*. Η προς αντιγραφή περιοχή μπορεί να βρίσκεται στο ίδιο ή σε άλλο αντικείμενο.

  Προσοχή: Η εισαγωγή στοιχείων σε μια ακολουθία μπορεί να προκαλέσει την ακύρωση προσεγγιστών που έδειχναν στοιχεία της ακολουθίας. Αυτό ισχύει και για το *vector* αφού με την εισαγωγή στοιχείου μπορεί να γίνει νέα παραχώρηση μνήμης. Έτσι, ο προσεγγιστής που επιστρέφει η πρώτη μορφή της *insert()* δεν είναι απαραίτητο να ισούται με τον (εισερχόμενο) *pos*.

- Μέθοδοι διαγραφής στοιχείων:

```
S<K>::iterator erase( S<K>::iterator pos );
S<K>::iterator erase( S<K>::iterator first,
                      S<K>::iterator last );
void clear();
```

  Η πρώτη μέθοδος διαγράφει το στοιχείο που δείχνει ο *pos* και μειώνει το *size()* κατά 1. Επιστρέφει προσεγγιστή που δείχνει το στοιχείο που ακολουθούσε αυτό που διαγράφηκε.

  Η δεύτερη μέθοδος διαγράφει όλα τα στοιχεία της περιοχής [*first*, *last*). Επιστρέφει προσεγγιστή που δείχνει το στοιχείο που αρχικώς έδειχνε ο *last*.

  Η τρίτη διαγράφει όλα τα περιεχόμενα. Η `a.clear()` είναι ισοδύναμη με `a.erase( a.begin(), a.end() )`.

  Προσοχή: Και η διαγραφή στοιχείων ακολουθίας μπορεί να προκαλέσει την ακύρωση προσεγγιστών που έδειχναν στοιχεία της ακολουθίας.

- Μέθοδος αλλαγής μήκους

```
void resize( size_type sz, K val = K() );
```

  Δίνοντας "`vt.resize( n, kv );`"

  αν *n* < *vt*.*size*() είναι σαν να δίνεις

  "`vt.erase( vt.begin()+n, vt.end() );`"

  ενώ αν *n* > *vt*.*size*() είναι σαν δίνεις

  "`vt.insert( vt.end(), n-vt.size(), kv );`"

Τα τρία περιγράμματα ακολουθιών έχουν μερικές ακόμη ιδιότητες (που δεν υπάρχουν για τα επιπλέον περιγράμματα που προδιαγράφονται στο C++11):

- Είναι **αναστρέψιμα** (reversible) περιέχοντα. Όπως είπαμε παραπάνω, αυτό σημαίνει ότι έχουν τύπο *reverse_iterator* και τις μεθόδους *rbegin()*, *rend()*.
- Δίνουν εργαλεία για **εισαγωγή** (και **διαγραφή**) **στο τέλος** (back insertion).
  - Μέθοδο για να πάρουμε το τελευταίο στοιχείο του περιέχοντος (αν δεν είναι κενό):

```
K& S<K>::back();
```

```
const K& S<K>::back();
```

Το "`a.back()`" είναι ισοδύναμο με το "`*(--a.end())`".

◦ Μέθοδο για εισαγωγή στο τέλος της συλλογής του περιέχοντος:

```
void S<K>::push_back( const K& );
```

Μετά την "`a.push_back( c );`" η "`a.back()`" μας δίνει το *c*.

◦ Μέθοδο για τη διαγραφή του τελευταίου στοιχείου της συλλογής του περιέχοντος (αν δεν είναι κενό):

```
void S<K>::pop_back();
```

26.2.1 Το Περίγραμμα "*vector*"

Πολλοί προγραμματιστές «βολεύονται» απλώς και μόνον με δυναμικούς πίνακες. Έτσι, «περιφρονούν» τις άλλες περιέχουσες κλάσεις της STL, χρησιμοποιούν μόνον τη *vector* (διάνυσμα) που γίνεται η ευρύτερα χρησιμοποιούμενη περιέχουσα κλάση της STL. «Βολεύονται» μόνο από τεμπελιά; Όχι!

♦ ***Το περίγραμμα "vector" είναι η μόνη περιέχουσα κλάση της STL[7] που εγγυάται αποθήκευση των στοιχείων σε συναπτές θέσεις της μνήμης.***

Έτσι, μπορεί κανείς να χρησιμοποιεί απλά βέλη –αλλά και συναρτήσεις– που έχουν γραφεί για συνήθεις (μονοδιάστατους) πίνακες για να χειριστεί πίνακες *vector*. Ας δούμε ένα

Παράδειγμα

Στην §12.3.5 είχαμε δει τη συνάρτηση

```
double vectorSum( const double x[], int n, int from, int upto )
{
   if ( x == 0 && n > 0 )
   {  cerr << "η vectorSum κλήθηκε με ανύπαρκτο πίνακα" << endl;
      exit( EXIT_FAILURE );  }
// άλλοι έλεγχοι
   double sum( 0 );
   for ( int m(from); m <= upto; ++m )  sum += x[m];
   return sum;
} // vectorSum
```

που υπολογίζει το άθροισμα των στοιχείων από `x[from]` μέχρι και `x[upto]` ενός πίνακα με *n* στοιχεία τύπου `double`. Μπορούμε να τη χρησιμοποιήσουμε για τον

```
   vector< double > vx;
```

Βεβαίως, και να πώς:

```
ut = vectorSum( &vx[0], vx.size(), 1, 2 );
```

Η αρχή του πίνακα βρίσκεται στη `&vx[0]`[8] και το πλήθος των στοιχείων του είναι `vx.size()`.

Το περίγραμμα *vector* δηλώνεται ως εξής:

```
template < class K, class Alloc = allocator<K> >
class vector;
```

Να δούμε τώρα τις δυνατότητες –για την ακρίβεια: τις μεθόδους– που προσφέρει η *vector* πέρα από αυτές που έχουν οι ακολουθίες γενικώς:

• Η πρόσβαση σε κάθε στοιχείο γίνεται (θυμίσου και τη *string*) με τον τελεστή "`[]`" και τη μέθοδο *at()*.

```
K& operator[]( size_type n );
const K& operator[]( size_type n ) const;
```

[7] Αυτό για τη C++03. Η STL της C++11 έχει και άλλες.

[8] Όπως θα δεις στη συνέχεια, στη *vector* επιφορτώνουμε τον "`[]`".

```
K& at( size_type n );
const K& at( size_type n ) const;
```

(Οι περιπτώσεις "`const`" για σταθερά αντικείμενα.) Όπως και στη *string*, η διαφορά έγκειται στο ότι η *at()* ελέγχει την τιμή του *n* και αν είναι ≥ *size()* ρίχνει εξαίρεση (*std*::)*out_of_range*.

- Όπως για τη *string* (§21.10.1) έτσι και για κάθε στιγμιότυπο του *vector* έχει νόημα η **χωρητικότητα** (capacity): το πλήθος αντικειμένων τύπου *K* που μπορούμε να αποθηκεύσουμε στη μνήμη που έχει ήδη δεσμευθεί για ένα αντικείμενο `vector<K>`. Μας τη δίνει η μέθοδος:

  ```
  size_type capacity() const;
  ```

  Προφανώς, για οποιοδήποτε αντικείμενο *a* τέτοιου τύπου θα έχουμε:

  `a.size() ≤ a.capacity()`

  Αν το `a.size()` πρέπει υπερβεί την τρέχουσα `a.capacity()` τότε θα πρέπει να γίνει νέα παραχώρηση μνήμης ώστε να μεγαλώσει η χωρητικότητα και να γίνουν οι απαραίτητες αντιγραφές (αυτά που κάνουμε με τη *renew()*). Βάζοντας αρκετά μεγάλη τιμή χωρητικότητας –με τη μέθοδο *reserve()* που ακολουθεί– περιορίζεις αυτές τις χρονοβόρες διαδικασίες.

- Η χωρητικότητα ενός αντικειμένου *vector* αλλάζει με τη

  ```
  void reserve( size_type n );
  ```

  Αν η *n* έχει τιμή μεγαλύτερη από την τρέχουσα χωρητικότητα (`a.capacity()`) τότε δεσμεύεται περισσότερη μνήμη ώστε να έχουμε `a.capacity()` ≥ *n*. Αυτό συνήθως απαιτεί αντιγραφή των περιεχομένων του *a*.

Επειδή με τις *size()*, *resize()*, *capacity()*, *reserve()* μπορεί να υπάρχουν μπερδεματα ας προσπαθήσουμε να τα ξεκαθαρίσουμε με ένα προγραμματάκι:

Παράδειγμα

```
 0: #include <iostream>
 1: #include <vector>
 2:
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:    vector<int> av( 3 );
 8:
 9:    cout << "capacity: " << av.capacity()
10:         << "   size: " << av.size() << endl;
11:    for ( int i(0); i <= 6; ++i )
12:    {
13:       if ( i < av.size() )  av[i] = i*10;
14:                       else  av.push_back( i*10 );
15:       cout << "i: " << i << "    capc: " << av.capacity()
16:            << "   size: " << av.size() << "    ";
17:       for ( int k(0); k < av.size(); ++k )
18:          cout << av[k] << ' ';
19:       cout << endl;
20:    }
21:
22:    av.reserve( 31 );
23:    av.resize( 4 );
24:    cout << "capacity: " << av.capacity()
25:         << "   size: " << av.size() << endl;
26:    for ( int k(0); k < av.size(); ++k )  cout << av[k] << ' ';
27:    cout << endl;
28:    try { cout << av.at(5) << endl; }
29:    catch( out_of_range& x )
30:    {  cout << x.what() << endl;  }
31: }
```

που μας δίνει:

```
capacity: 3   size: 3
i: 0   capc: 3   size: 3   0 0 0
i: 1   capc: 3   size: 3   0 10 0
i: 2   capc: 3   size: 3   0 10 20
i: 3   capc: 6   size: 4   0 10 20 30
i: 4   capc: 6   size: 5   0 10 20 30 40
i: 5   capc: 6   size: 6   0 10 20 30 40 50
i: 6   capc: 12   size: 7   0 10 20 30 40 50 60
capacity: 31   size: 4
0 10 20 30
index out of range in function: vector:: at(size_t)
 index: 5 is greater than max_index: 4
```

Με τη δήλωση της γραμμής 7 δηλώνουμε έναν (δυναμικό) πίνακα ακεραίων με αρχικό μέγεθος 3, δηλαδή με 3 στοιχεία. Από τη γραμμή 9 παίρνουμε το αποτέλεσμα "`capacity: 3    size: 3`". Η χωρητικότητα έγινε αυτομάτως `3`. Αυτή είναι η ελάχιστη τιμή για την οποίαν ικανοποιείται η: `av.size() ≤ av. capacity()`.

Στις πρώτες 3 εκτελέσεις (*i*: 0, 1, 2) της **for** δίνουμε τιμές στα `va[i]` με εκχώρηση (γραμμή 13) αφού τα στοιχεία αυτά υπάρχουν.

Για την εισαγωγή της 4ης τιμής ("`3`") χρησιμοποιούμε την "`av.push_back(i*10)`" διότι θέλουμε να μεγαλώσουμε ταυτοχρόνως τον πίνακα. Το ίδιο ισχύει και για τις επόμενες τιμές.

Όπως φαίνεται στη γραμμή "`i: 3`" η *av.size()* επέστρεψε τιμή "`4`" –αυξήθηκε κατά 1– αλλά η *av.capacity()* επέστρεψε "`6`"· διπλασιάστηκε με τη νέα παραχώρηση μνήμης που μεσολάβησε! Στη γραμμή "`i: 6`" βλέπουμε τι έγινε για να μπορέσουμε να βάλουμε και έβδομη τιμή: το μέγεθος αυξήθηκε κατά 1 αλλά η χωρητικότητα ξαναδιπλασιάστηκε· είχαμε και πάλι νέα παραχώρηση. Έτσι δουλεύει ο παραχωρητής μνήμης του *vector* και αυτό το έχουμε συζητήσει ήδη στην §16.13.3.

Η γραμμή "`capacity: 31    size: 4`" μας δείχνει τα αποτελέσματα των εντολών που δώσαμε στις γραμμές 22 και 23. Το αποτέλεσμα της **for** της γραμμής 26 μας επιβεβαιώνει το "`size: 4`".

Άλλη μια επιβεβαίωση για το μέγεθος μας δίνει και η απόπειρα χρήσης του 6ου στοιχείου του πίνακα με την "`av.at(5)`" μέσα σε μια `try`. Ρίχνεται εξαίρεση *out_of_range* από τη *what()* της οποίας παίρνουμε τις δύο τελευταίες γραμμές.

❖❖❖

Τώρα θα πούμε και μερικά πράγματα για τους προσεγγιστές του *vector*. Αν ψάξεις λίγο στο αρχείο `vector` της C++ που χρησιμοποιείς θα δεις κάτι σαν:

```
template < class T, class Allocator = allocator<T> >
class vector
{
// . . .
public:
   typedef T             value_type;
// . . .
   typedef value_type* iterator;
// . . .
   typedef ptrdiff_t   difference_type;
// . . .
```

Δηλαδή: ο προσεγγιστής "`vector<T>::iterator`" είναι βέλος "`T*`" προς στοιχείο πίνακα. Έτσι, γίνεται αυτομάτως προσεγγιστής τυχαίας πρόσβασης και έχουμε δυνατότητα για αριθμητική βελών.

Στο παραπάνω παράδειγμα θα μπορούσαμε, μετά τη γραμμή 20, να βάλουμε:

```
   cout << *(av.begin()+3) << endl;
   vector<int>::iterator it1( av.begin() ),
                         it2( av.begin() );
   it1 += 2;
```

```
    it2 += 6;
    vector<int>::difference_type dist( it2 - it1 );
    cout << "dist: " << dist << endl;
```

και να πάρουμε:

```
30
*it1: 20   *it2: 60
dist: 4
```

Και κάτι για τις ακυρώσεις προσεγγιστών:

♦ ***Αν έχεις προσεγγιστές προς στοιχεία κάποιου αντικειμένου* vector *και κάνεις κάποια εισαγωγή (insert) ή διαγραφή (erase) όσοι προσεγγιστές έδειχναν από το πρώτο στοιχείο που εισάχθηκε (ή διαγράφηκε) και μετά ακυρώνονται (δεν δείχνουν αυτό που έδειχναν).***

Αν το ξεχάσεις μπορεί να «υποφέρεις» από μερικά δύσκολα προγραμματιστικά λάθη.

26.2.1.1 Εξαιρέσεις, Απόδοση και Άλλα

Με πόση ασφάλεια και πόσο γρήγορα γίνονται οι πράξεις σε αντικείμενα *vector*; Ας τις δούμε μια προς μια.

Πρώτα οι διαγραφές: Ας πούμε ότι έχεις ένα αντικείμενο *v* τύπου `vector <K>` –με *n* στοιχεία τύπου *K*– και δίνεις την εντολή "`v.erase(f, l);`".

Αν στα προς διαγραφή στοιχεία περιλαμβάνεται και το τελευταίο στοιχείο (αν δηλαδή *l* `==` *v*`.`*end()*) δεν γίνονται αντιγραφές. Απλώς ο παραχωρητής παίρνει πίσω τα τελευταία *l - f* στοιχεία και καταστρέφει τα αντικείμενά τους καλώντας τον `~K()`. Στην περίπτωση αυτή (καλώς εχόντων των πραγμάτων) δεν εγείρεται εξαίρεση. Ο χρόνος που απαιτείται είναι αυτός που χρειάζεται για την καταστροφή των *l - f* αντικειμένων.

Έστω τώρα ότι ο *f* δείχνει το `v[j]` και ο *l* δείχνει το `v[k]` (όπου *k - j = l - f*), δύο ενδιάμεσα στοιχεία του πίνακα. Τι πρέπει να γίνει;

- Να αντιγραφούν τα `v[k]..v[n-1]` στα στοιχεία που ξεκινούν από το `v[j]` (`copy(l, v.end(), f)`).
- Να καταστραφούν τα τελευταία *l - f* στοιχεία και να μειωθεί το μέγεθος του πίνακα κατά *l - f*.

Δηλαδή: έχουμε γραμμική εξάρτηση του χρόνου από το πλήθος (*n*-1-*k*) των στοιχείων από το `v[k+1]` μέχρι το τέλος του πίνακα.

Από τις αντιγραφές μπορεί να έχουμε εξαίρεση. Πάντως δεν θα έχουμε διαρροή μνήμης και το *v* θα είναι διαχειρίσιμο (βασική εγγύηση).

Και για να επανέλθουμε σε αυτό που λέμε πιο πάνω: *αν έχεις προσεγγιστές που –πριν τη διαγραφή– δείχνουν σε στοιχεία από* `v[j]` *μέχρι το τέλος του πίνακα, μετά τη διαγραφή ακυρώνονται.*

Ας πάμε τώρα στην εισαγωγή. Έστω ότι θέλουμε να εισαγάγουμε *m* αντικείμενα ξεκινώντας από τη θέση `v[k]`. Θα πρέπει:

- Να μεγαλώσει το μέγεθος του πίνακα κατά *m* (σε *n*+*m*). Υποθέτουμε ότι δεν θα χρειαστεί νέα παραχώρηση μνήμης.
- Να αντιγραφούν τα `v[k]..v[n-1]` στα `v[k+m]..v[n+m-1]` αντιστοίχως.
- Να αντιγραφούν τα *m* εισαγόμενα αντικείμενα στις `v[k]..v[k+m-1]`.

Έχουμε δηλαδή και εδώ γραμμική εξάρτηση του χρόνου από το πλήθος (*n*-*k*) των στοιχείων από το `v[k]` μέχρι το τέλος του πίνακα και το πλήθος *m* των εισαγόμενων στοιχείων. Αλλά:

- Αν δεν ισχύει η υπόθεση που κάναμε και *έχουμε* νέα παραχώρηση μνήμης τότε θα πρέπει να γίνουν και *n* αντιγραφές. Στην περίπτωση αυτήν μπορεί –κατ' αρχήν τουλάχιστον– να πάρουμε *bad_alloc*.

- Ο χρόνος εισαγωγής στο τέλος εξαρτάται μόνο από το πλήθος των στοιχείων που εισάγονται. Φυσικά, για εισαγωγή ενός στοιχείου είτε με την *push_back()* είτε με κάποια *insert()* ο χρόνος είναι σταθερός. Σταθερός είναι ο χρόνος και την *pop_back()*.
- Για την τρίτη συνάρτηση (περίγραμμα) εισαγωγής

```
template < class InputIterator >
void insert( S<K>::iterator pos,
             InputIterator first, InputIterator last );
```

  τα πράγματα μπορεί να είναι πιο πολύπλοκα. Οι *first* και *last* είναι –γενικώς– προσεγγιστές εισόδου. Και αν μεν είναι πρόσθιοι μπορούν να γίνουν αυτά που είπαμε παραπάνω. Αν όμως είναι, ας πούμε, *istream_iterator*s τότε δεν μπορούμε να μετρήσουμε και να βρούμε το *m* εκ των προτέρων. Τα στοιχεία θα εισάγονται ένα προς ένα και θα γίνουν *m* φορές αυτά που είπαμε πιο πάνω (αν τα εξειδικεύσεις για *m* = 1).

Εκτός από τη *bad_alloc*, που προαναφέραμε, μπορεί να έχουμε εξαίρεση από τις αντιγραφές. Σε κάθε περίπτωση έχουμε και εδώ ασφάλεια επιπέδου βασικής εγγύησης.

26.2.1.2 Και μια Εξειδίκευση: “vector<bool>”

Όπως έχουμε παρατηρήσει και σε προηγούμενο κεφάλαιο, από τα 8 ψηφία της ψηφιολέξης με την οποίαν παριστάνεται μια τιμή τύπου `bool` χρησιμοποιούμε μόνον το ένα. Για να γίνει η σχετική οικονομία, η STL προσφέρει μια εξειδίκευση του *vector* για τον τύπο `bool`:

```
template <class Allocator>
class vector<bool, Allocator>
```

Πέρα από την οικονομία μνήμης, η εξειδίκευση έχει δύο επιπλέον συναρτήσεις:

- Τη μέθοδο:

```
void flip();
```

  που αλλάζει τις τιμές όλων των περιεχομένων από *b* σε *~b*· δηλαδή αλλάζει όλα τα `false` σε `true` και όλα τα `true` σε `false`.
- Πέρα από τη μέθοδο *swap()*, που ανταλλάσσει τις τιμές δύο πινάκων, τη στατική συνάρτηση:

```
static void swap( vector<bool>::reference ref1,
                  vector<bool>::reference ref2 );
```

  που ανταλλάσσει τις τιμές δύο στοιχείων του πίνακα. Το “`reference`” να το σκέφτεσαι ως “`bool&`” (αλλά διάβασε παρακάτω.)

Παρατήρηση:▶

Το περίγραμμα *vector* έχει γενικώς έναν τύπο “`reference`” που για το περίγραμμα `vector<K>` ορίζεται ως “`K&`”. Ειδικώς όμως για το `vector<bool>` υπάρχει πρόβλημα: Όπως έχουμε πει, οι αναφορές είναι «κρυμμένα» βέλη, δηλαδή διευθύνσεις. Αν λοιπόν «στριμώξουμε» το κάθε στοιχείο μιας τέτοιας κλάσης σε ένα δυαδικό ψηφίο τότε την ίδια διεύθυνση τη μοιράζονται 8 (ή περισσότερα) στοιχεία. Για να λυθεί αυτό το πρόβλημα ορίζεται (μέσα στο `vector<bool>`) ολόκληρη κλάση `reference`.◀

Δες το παρακάτω προγραμματάκι:

```
 0: #include <iostream>
 1: #include <vector>
 2: using namespace std;
 3:
 4: void display( ostream& tout, const vector<bool>& bv )
 5: {
 6:     tout << boolalpha;
 7:     for ( int k(0); k < bv.size(); ++k ) tout << bv[k] << ' ';
 8:     tout << endl;
 9: }
10:
11: int main ()
12: {
```

```
13:    vector<bool> bv1( 3 ) , bv2( 2 );
14:
15:    bv1[0] = true;  bv1[1] = false;  bv1[2] = true;
16:    cout << "bv1: ";  display( cout, bv1 );
17:    bv1.flip();
18:    cout << "bv1: ";  display( cout, bv1 );
19:
20:    bv2[0] = true;  bv2[1] = false;
21:    cout << "bv2: ";  display( cout, bv2 );
22:
23:    vector<bool>::swap( bv1[0], bv1[1] );
24:    cout << "bv1: ";  display( cout, bv1 );
25:    vector<bool>::swap( bv2.front(), bv2.back() );
26:    cout << "bv2: ";  display( cout, bv2 );
27:    vector<bool>::swap( bv1[1], bv2[1] );
28:    cout << "bv1: ";  display( cout, bv1 );
29:    cout << "bv2: ";  display( cout, bv2 );
30:
31:    bv1.swap( bv2 );
32:    cout << "bv1: ";  display( cout, bv1 );
33:    cout << "bv2: ";  display( cout, bv2 );
34: }
```

που βγάζει:

```
bv1: true false true
bv1: false true false
bv2: true false
bv1: true false false
bv2: false true
bv1: true true false
bv2: false false
bv1: false false
bv2: true true false
```

Πρόσεξε τα εξής:

- Η αλλαγή του *bv1* από την πρώτη γραμμή στη δεύτερη προκλήθηκε από την "`bv1.flip();`" της γραμμής 17.
- Η αλλαγή του *bv1* από τη δεύτερη στην τέταρτη γραμμή είναι αποτέλεσμα της *swap()* της γραμμής 23 ενώ η αλλαγή του *bv2* από τη τρίτη στην πέμπτη γραμμή είναι αποτέλεσμα της *swap()* της γραμμής 25.
- Η *swap()* της γραμμής 27 ανταλλάσσει τις τιμές στοιχείων διαφορετικών αντικειμένων.
- Τέλος, στη γραμμή 31 βλέπεις την «άλλη *swap()*», αυτήν που έχουν όλα τα περιέχοντα της STL. Το αποτέλεσμα φαίνεται στις δύο τελευταίες του απότελέσματος: τα *bv1* και *bv2* έχουν ανταλλάξει τιμές. Το ίδιο αποτέλεσμα θα είχαμε με την "`bv2.swap(bv1);`".

26.2.2 Το Περίγραμμα "*deque*"

Το περίγραμμα *deque* (προφέρεται "deck") υλοποιεί αυτό που λέει το όνομά της: μια ουρά δύο άκρων (*d*ouble *e*nded *que*ue).

Για να μπορεί να λειτουργήσει ως ουρά δύο άκρων χρειάζεται –εκτός από τις *front()*, *back()*, *push_back()*, *pop_back()*– και τις *push_front()*, *pop_front()* για εισαγωγή και διαγραφή στην αρχή του περιέχοντος που τις έχουμε δει στο *SListT* (§25.7.1):

- Εισαγωγή στοιχείου στην αρχή του περιέχοντος:

  `void deque<K>::push_front( const K& val );`

  Μετά την "`a.push_front( c );`" η "`a.front()`" μας δίνει το *c*· το ίδιο και η "`*a.begin()`".
- Διαγραφή του πρώτου στοιχείου της ακολουθίας του περιέχοντος (αν δεν είναι κενό):

  `void deque<K>::pop_front();`

Οι δύο αυτές μέθοδοι έχουν σταθερό χρόνο εκτέλεσης, όπως άλλωστε και οι *push_back()*, *pop_back()*.

Κατά τα άλλα, παρ' όλο που ο τρόπος αποθήκευσης των περιεχομένων είναι πιο πολύπλοκος από τον απλό μονοδιάστατο πίνακα, η λειτουργικότητα του *deque* είναι ίδια με αυτήν του *vector* με μια διαφορά: *Το deque δεν έχει τις reserve() και capacity()*.

26.2.3 Το Περίγραμμα "*list*"

Το περίγραμμα *list*:

```
template < class T, class Allocator = allocator<T> >
class list;
```

υλοποιεί λίστα με διπλή σύνδεση με περιεχόμενο στοιχείων τύπου *T*.

Για το *list*, πέρα από αυτά που έχει το *deque*, υπάρχουν επιπλέον:

- Μέθοδοι για μεταφορά στοιχείων (προσοχή: *μεταφορά όχι αντιγραφή*) σε μια λίστα από άλλη λίστα με στοιχεία και παραχωρητή μνήμης ίδιου τύπου:

  ```
  void splice( iterator pos, list<T,Allocator>& x );
  void splice( iterator pos, list<T,Allocator>& x, iterator i);
  void splice( iterator pos, list<T,Allocator>& x,
               iterator first, iterator last );
  ```

  Αν οι *l1*, *l2* είναι λίστες `list<K>` τότε:

  - Η "`l1.splice(it, l2);`", όπου ο *it* είναι προσεγγιστής προς στοιχείο της *l1*, μεταφέρει στην *l1* όλα τα στοιχεία της *l2* και τα συνδέει πριν από αυτό που δείχνει ο *it*. Η *l2* παραμένει χωρίς στοιχεία.
  - Η "`l1.splice(it, l2, it2);`", όπου ο *it* είναι προσεγγιστής προς στοιχείο της *l1* και ο *it2* προσεγγιστής προς στοιχείο της *l2*, μεταφέρει στην *l1* το στοιχείο που δείχνει ο *it2* και το συνδέει πριν από αυτό που δείχνει ο *it* ενώ το αφαιρεί από την *l2*. Αν δώσεις "`l1.splice(it, l1, it2);`" κάνεις εσωτερική μετακίνηση στην *l1*.
  - Η "`l1.splice(it, l2, f, l);`", όπου ο *it* είναι προσεγγιστής προς στοιχείο της *l1* και οι *f*, *l* προσεγγιστές προς στοιχεία της *l2*, μεταφέρει στην *l1* το στοιχεία της περιοχής `[f,l)` της *l2* και τα συνδέει πριν από αυτό που δείχνει ο *it* ενώ τα αφαιρεί από την *l2*. Αν δώσεις "`l1. splice(it, l1, f, l);`" έχεις εσωτερική μετακίνηση στοιχείων στην *l1*. Στην περίπτωση αυτήν ο *it* πρέπει να είναι έξω από την `[f,l)`.

- Μέθοδος για διαγραφή στοιχείων με συγκεκριμένη τιμή και μέλος–περίγραμμα για διαγραφή στοιχείων με συγκεκριμένες προδιαγραφές:

  ```
  void remove( const T& v );
  template < class Predicate >
  void remove_if( Predicate pred );
  ```

  Η "`l3.remove( v );`" έχει ως αποτέλεσμα την αφαίρεση από την *l3* όλων των στοιχείων που έχουν τιμή *v*.

  Μέσω του *pred()*, που είναι ένα κατηγόρημα με μια παράμετρο τύπου *T*, περνάμε στη *remove_if()* τις προδιαγραφές. Δες ένα

Παράδειγμα

Το πρόγραμμα:

```
 0: #include <iostream>
 1: #include <list>
 2: using namespace std;
 3:
 4: template< typename K >
 5: void display( ostream& tout, list<K>& lv )
 6: {
 7:    for ( list<K>::iterator it(lv.begin());
```

```
 8:          it != lv.end(); ++it ) tout << *it << ' ';
 9:    tout << endl;
10: } // display
11:
12: struct Dec0
13: {
14:    bool operator()( const int& v )
15:    {  return ( (0 <= v) && (v < 10) );  }
16: }; // Dec0
17:
18: int main ()
19: {
20:    int ints[] = { 3, 23, 3, 19, 5, 19, 17, 7, 7, 17, 23 };
21:    list<int> l3( ints, ints+11 );
22:    display<int>( cout, l3 );
23:
24:    l3.remove( 23 );  display<int>( cout, l3 );
25:
26:    l3.remove( 7 );  display<int>( cout, l3 );
27:
28:    l3.remove_if( Dec0() );  display<int>( cout, l3 );
29: }
```

δίνει:

```
3 23 3 19 5 19 17 7 7 17 23
3 3 19 5 19 17 7 7 17
3 3 19 5 19 17 17
19 19 17 17
```

Με την εντολή της γραμμής 24 ζητούμε να αφαιρεθούν από τη λίστα όλα τα "23" και παίρνουμε τη δεύτερη γραμμή του αποτελέσματος. Παρομοίως, με την εντολή της γραμμής 26 αφαιρούμε τα "7" και παίρνουμε την τρίτη γραμμή του αποτελέσματος.

Στις γραμμές 12-16 ορίζουμε ένα συναρτησοειδές (*Dec0*) που επιλέγει μονοψήφιους φυσικούς. Με την εντολή της γραμμής 28 αφαιρούμε από τη λίστα τους μονοψήφιους φυσικούς "3", "3" και "5".

Αντί για το συναρτησοειδές θα μπορούσαμε να ορίσουμε:

```
bool dec0( int v )  {  return ( (0 <= v) && (v < 10) );  }
```

και να πάρουμε το ίδιο αποτέλεσμα γράφοντας:

```
   l3.remove_if( dec0() );
```

❖❖❖

- Μέθοδος για διαγραφή *διαδοχικών αντιγράφων ενός στοιχείου* και μέλος–περίγραμμα για διαγραφή διαδοχικών στοιχείων με συγκεκριμένες προδιαγραφές:

  ```
  void unique();
  template < class BinaryPredicate >
  void unique( BinaryPredicate binaryPred );
  ```

  Για την πρώτη περίπτωση διασχίζεται η λίστα με έναν προσεγγιστή *it* και συγκρίνονται για ισότητα ζεύγη περιεχόμενων τιμών (`*it`, `*(it-1)`). Αν βρεθούν ίσα διαγράφεται αυτό που δείχνει ο *it*.

  Για τη δεύτερη μορφή, το κατηγόρημα «τροφοδοτείται» με τα `*it`, `*(it-1)`. Αν το κατηγόρημα επιστρέψει τιμή `true` διαγράφεται το στοιχείο που δείχνει ο *it*. Όπως καταλαβαίνεις, το κατηγόρημα πρέπει να είναι δυαδικό.

Παράδειγμα

Το πρόγραμμα:

```
 0: #include <iostream>
 1: #include <list>
 2: using namespace std;
 3:
 4: template< typename K >
 5: void display( ostream& tout, list<K>& lv )
```

```
 6: // ΟΠΩΣ ΠΑΡΑΠΑΝΩ
 7:
 8: struct NotInOrder
 9: {
10:    bool operator()( const int& v1, const int& v2 )
11:    {  return ( v1 > v2 );  }
12: }; // NotInOrder
13:
14: int main ()
15: {
16:    int ints[] = { 2, 2, 2, 3, 3, 5, 5, 5, 5, 2, 2, 3 };
17:    list<int> l4( ints, ints+12 );
18:    display<int>( cout, l4 );
19:
20:    l4.unique();  display<int>( cout, l4 );
21:
22:    l4.unique( NotInOrder() );  display<int>( cout, l4 );
23: }
```

δίνει:

```
2 2 2 3 3 5 5 5 5 2 2 3
2 3 5 2 3
2 3 5
```

Τι γίνεται με την εντολή της γραμμής 20; Μετά το πρώτο "2" διαγράφονται τα δύο αντίγραφά του ("2") που το ακολουθούν· τα ίδια ισχύουν και για τα "3" και τα "5". Το "2", που είναι προτελευταίο στοιχείο της ακολουθίας, αφαιρείται διότι υπάρχει "2" ακριβώς πριν από αυτό.

Το συναρτησοειδές *NotInOrder* ορίζει ένα κατηγόρημα που επιστρέφει `true` αν το πρώτο όρισμα είναι μεγαλύτερο από το δεύτερο. Η εντολή της γραμμής 24 έχει ως αποτέλεσμα να αφαιρεθεί το "2" που ακολουθεί το "5" και στη συνέχεια, να αφαιρεθεί και το "3" όταν βρίσκεται και αυτό να ακολουθεί το "5".

❖❖❖

- Μέθοδος και μέλος–περίγραμμα για ταξινόμηση των περιεχομένων της λίστας:

```
void sort();
template < class Compare >
void sort( Compare comp );
```

Τα περιεχόμενα μιας λίστας τύπου `list<K>` ταξινομούνται με βάση τον τελεστή "<" της *K*.

Αν θέλουμε ταξινόμηση χωρίς να έχουμε ορισμένο τον "<" της *K* ή με άλλη διάταξη από αυτήν που μας δίνει ο γράφουμε ένα συναρτησοειδές-κατηγόρημα που να ορίζει τη διάταξη ("<" της *K*) που μας ενδιαφέρει. Για την ταξινόμηση η "*a* < *b*" έχει το νόημα «το *a* προηγείται του *b*».

Παράδειγμα

Σε μια λίστα έχουμε τις ημερομηνίες ιστορικών γεγονότων:

```
Date dates[] = {  Date(1821,3,25), Date(1940,10,28),
                  Date(1904,10,13), Date(1453,5,29),
                  Date(1912,10,26)  };
list<Date> dateList( dates, dates+5 );
```

Αν δώσουμε:

```
dateList.sort();
display<Date>( cout, dateList );
```

θα πάρουμε:

```
29.5.1453 25.3.1821 13.10.1904 26.10.1912 28.10.1940
```

Όπως βλέπεις, εδώ έχουμε ιστορικές ημερομηνίες τις επετείους των οποίων γιορτάζουμε πανελληνίως ή τοπικώς. Θα ήταν λοιπόν ενδιαφέρον να τις έχουμε ταξινομημένες κατά μήνα και ημέρα. Για να το κάνουμε αυτό γράφουμε το συναρτησοειδές:

```
struct MonthFirst
{
   bool operator()( const Date& d1, const Date& d2 )
   {
      bool fv( false );
      if ( d1.getMonth() < d2.getMonth() )  fv = true;
      else if ( d1.getMonth() == d2.getMonth() )
      {
         if ( d1.getDay() < d2.getDay() )  fv = true;
         else if ( d1.getDay() == d2.getDay() )
            fv = ( d1.getYear() < d2.getYear() );
      }
      return fv;
   }
}; // MonthFirst
```

και γράφοντας:

```
   dateList.sort( MonthFirst() );
   display<Date>( cout, dateList );
```

θα πάρουμε:

```
25.3.1821 29.5.1453 13.10.1904 26.10.1912 28.10.1940
```

Αν, τώρα, θέλουμε τις ημερομηνίες από τις πιο πρόσφατες προς τις παλαιότερες (κατά «φθίνουσα» τάξη) γράφουμε το συναρτησοειδές:

```
struct Decr
{
   bool operator()( const Date& d1, const Date& d2 )
   {  return ( d2 < d1 );  }
}; // Decr
```

και με τις:

```
   dateList.sort( Decr() );
   display<Date>( cout, dateList );
```

να πάρουμε:

```
28.10.1940 26.10.1912 13.10.1904 25.3.1821 29.5.1453
```

❧❧❧

Γιατί να προτιμήσουμε να δώσουμε "`l.sort()`" αντί για "`sort(l.begin(), l.end())`"; Η μέθοδος υπερέχει ως προς το ότι είναι **ευσταθής** (stable): η σχετική διάταξη δύο αντικειμένων με ίσα κλειδιά δεν αλλάζει κατά την ταξινόμηση. Προφανώς αυτό έχει νόημα όταν η λίστα φιλοξενεί σύνθετα αντικείμενα.

- Μέθοδος και μέλος–περίγραμμα για να συγχωνεύσουμε (τα περιεχόμενα από) δύο λίστες που είναι ήδη ταξινομημένες:

  ```
  void merge( list& x );
  template < class Compare >
  void merge( list& x, Compare comp );
  ```

  Αν έχουμε "`list<K> l1, l2`" και οι *l1*, *l2* είναι ταξινομημένες σύμφωνα με τον "`<`" της *K* τότε η "`l1.merge(l2);`" μεταφέρει (δεν αντιγράφει) και συγχωνεύει στην *l1* τα περιεχόμενα της *l2*. Τελικώς:

  - Η *l1* έχει όλα τα στοιχεία που είχαν αρχικώς οι *l1*, *l2* και είναι ταξινομημενα σύμφωνα με τον "`<`" της *K*.
  - Η *l2* είναι *κενή*.

  Αν οι *l1*, *l2* είναι ταξινομημένες με άλλη λογική χρησιμοποιούμε το περίγραμμα βάζοντας με τη δεύτερη παράμετρο τη διάταξη όπως κάναμε στην ταξινόμηση.

  Σε σχέση με τον αλγόριθμο (*std*::)*merge()* οι μέθοδοι έχουν ευστάθεια με την έννοια: μετά την "`l1.merge(l2);`" κάθε στοιχείο της *l2* εισάγεται μετά από το(τα) στοχείο(-α) της *l1* με το ίδιο κλειδί.

- Μέθοδος για την αντιστροφή μιας λίστας:

```
void reverse();
```

Για παράδειγμα, οι εντολές:

```
int ints[] = { 3, 23, 3, 19, 5, 19, 17, 7, 7, 17, 23 };
list<int> l3( ints, ints+11 );
display<int>( cout, l3 );

l3.reverse();  display<int>( cout, l3 );
```

θα δώσουν:

```
3 23 3 19 5 19 17 7 7 17 23
23 17 7 7 17 19 5 19 3 23 3
```

26.2.4 Ποια Ακολουθία να Διαλέξω;

Η πιο συνηθισμένη απάντηση είναι: *vector*! Πράγματι, αυτό το περιέχον φαίνεται να υπερέχει σε όλες τις περιπτώσεις αλλά και τα άλλα έχουν λόγο ύπαρξης.

- Διάλεξε το *deque* αν έχεις συλλογή-ακολουθία με συχνές εισαγωγές/εξαγωγές στα δύο άκρα.
- Διάλεξε το *list* αν έχεις συλλογή-ακολουθία με συχνές εισαγωγές/εξαγωγές στη «μέση» (μακριά από τα άκρα).

26.3 Συνειρμικά Περιέχοντα

Συνειρμική μνήμη (associative memory) ή **μνήμη προσπελάσιμη με το περιεχόμενο** (content-addressable memory) είναι τρόπος οργάνωσης της μνήμης ώστε η πρόσβαση στην κάθε μονάδα του περιεχομένου της να γίνεται όχι με τη διεύθυνσή της αλλά με τμήμα της μονάδας (κλειδί). Έτσι οργανώνουν τα περιεχόμενά τους τα **συνειρμικά περιέχοντα** (associative containers) της STL.

Δες τις επικεφαλίδες των τεσσάρων περιγραμμάτων:

```
template < class Key, class Compare = less<Key>,
           class Allocator = allocator<Key> >
class set;
template < class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T> > >
class map;
template < class Key, class Compare = less<Key>,
           class Allocator = allocator<Key> >
class multiset;
template < class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T> > >
class multimap;
```

Η πρώτη παράμετρος-κλάση (*Key*) είναι ο τύπος του αντικειμένου-κλειδιού με βάση το οποίο γίνονται οι αναζητήσεις στη συλλογή. Η παράμετρος-κλάση *Compare* είναι ένα συναρτησοειδές που καθορίζει διάταξη "<" για αντικείμενα κλάσης *Key*. Αν για την *Key* είναι ορισμένος ο "<" μπορείς να μην ορίσεις το συναρτησοειδές και τον ρόλο του να παίξει το *less*<*Key*>.[9]

Για να έχουμε γρήγορη αναζήτηση τα κλειδιά οργανώνονται –με βάση τη διάταξη "<" που είδαμε πιο πάνω– σε ένα **δένδρο δυαδικής αναζήτησης** (binary search tree).[10] Όπως

[9] Είδαμε το περίγραμμα "*less*" στην §25.6.3.

[10] Συνήθως ένα δένδρο red-black.

κατάλαβαίνεις, παρ' όλο που πραγματευόμαστε –γενικώς– σύνολα και πολυσύνολα, συζητούμε για **διαταγμένα** (ordered) σύνολα και πολυσύνολα.[11]

Κάθε στιγμιότυπο οποιουδήποτε από τα παραπάνω περιγράμματα έχει τη δική του κλάση (αμφίδρομων) προσεγγιστών.

Πέρα από τις μεθόδους που έχουν όλα τα περιέχοντα της STL, στα συνειρμικά περιέχοντα υπάρχουν:

- Μέθοδος που επιστρέφει το πλήθος των αντικειμένων που έχουν συγκεκριμενο κλειδί.
```
size_type A<K>::count( const K& x ) const;
```
  Προφανώς, αν το *A* είναι *set* ή *map* η *count()* μπορεί να επιστρέφει "`0`" ή "`1`".[12]
- Μέθοδοι για εντοπισμό κλειδιού στη συλλογή:
```
A<K>::iterator A<K>::find( const K& x );
A<K>::const_iterator A<K>::find( const K& x ) const;
```
  Αν το κλειδί *x* βρεθεί στη συλλογή επιστρέφεται προσεγγιστής προς αυτό το αντικείμενο. Αλλιώς ο προσεγγιστής δείχνει το "`A<K>::end()`".
- Μέθοδοι για εισαγωγές στοιχείων που θα τις δούμε ξεχωριστά για κάθε περιέχον.
- Μέθοδοι για διαγραφές στοιχείων:
```
void A<K>::erase( A<K>::iterator pos );
size_type A<K>::erase( const K& x );
void A<K>::erase( A<K>::iterator first, A<K>::iterator last );
void A<K>::clear();
```
  Η πρώτη διαγράφει το αντικείμενο που δείχνει ο *pos*.

  Η δεύτερη διαγράφει όλα τα αντικείμενα που έχουν κλειδί *x*. Επιστρέφει το πλήθος των αντικειμένων που διαγράφηκαν· το ίδιο που θα επέστρεφε το "`count(x)`" πριν από τη διαγραφή.

  Η τρίτη διαγράφει όλα τα αντικείμενα που τα κλειδιά τους ανήκουν στην περιοχή [*first*, *last*).

  Η τελευταία είναι ισοδύναμη με "`erase( begin(), end() )`" και έχει ως αποτέλεσμα κενό περιέχον.
- Μέθοδοι για εντοπισμό περιοχής κλειδιών με συγκεκριμένη τιμή:
```
A<K>::iterator A<K>::lower_bound( const K& x );
A<K>::const_iterator A<K>::lower_bound( const K& x ) const;
```
  Επιστρέφει προσεγγιστή προς το πρώτο στοιχείο που δεν είναι μικρότερο από *x*. Δες και αυτά που λέγαμε για τη συνάρτηση *lower_bound()* στην §26.1.3. Αν στο περιέχον υπάρχει (-ουν) στοιχείο(-α) με κλειδί *x* τότε ο προσεγγιστής που επιστρέφεται δείχνει το πρώτο (ή το μοναδικό) στοιχείο με τέτοιο κλειδί.
```
A<K>::iterator A<K>::upper_bound( const K& x );
A<K>::const_iterator A<K>::upper_bound( const K& x ) const;
```
  Επιστρέφει προσεγγιστή προς το πρώτο στοιχείο του περιέχοντος που είναι μεγαλύτερο από *x*.
```
pair< A<K>::iterator, A<K>::iterator > A<K>::equal_range( const K& x );
pair< A<K>::const_iterator, A<K>::const_iterator >
                                    A<K>::equal_range( const K& x ) const;
```
  Επιστρέφεται ζεύγος[13] προσεγγιστών *f*, *l* ώστε όλα τα κλειδιά στην [*f*, *l*) να έχουν τιμή *x*. Αν δεν υπάρχει το κλειδί οι δυο προσεγγιστές είναι ίσοι (περιοχή κενή). Αν για ένα

---

[11] Το νέο πρότυπο C++11 προδιαγράφει και *μη-διατεταγμένα* σύνολα/πολυσύνολα.

[12] Γράφουμε "`A<K>`" αλλά μπορεί να είναι και "`A<K,C>`" ή "`A<K,C,T>`". Απλώς όλες οι μέθοδοι έχουν να κάνουν μόνο με το κλειδί.

συγκεκριμένο περιέχον πάρουμε το *lb* (*lower_bound()*), το *ub* (*upper_bound()*) και το ζεύγος *eqr* (*equal_range()*) τότε θα έχουμε *eqr.first* == *lb* και *eqr.second* == *ub*. Το πρόγραμμα:

```
#include <iostream>
#include <functional>
#include <set>
using namespace std;
int main()
{
   int ints[] = { 3, 23, 3, 19, 5, 19, 17, 7, 7, 17, 23 };
   multiset<unsigned int> ms( ints, ints+11 );
   typedef multiset<unsigned int>::iterator msIterator;

   msIterator lb( ms.lower_bound(17) );
   msIterator ub( ms.upper_bound(17) );
   pair<msIterator, msIterator> eqr( ms.equal_range(17) );
   if ( lb == eqr.first )  cout << "lb == eqr.first" << endl;
   if ( ub == eqr.second )  cout << "ub == eqr.second" << endl;
}
```

δίνει

```
lb == eqr.first
ub == eqr.second
```

- Μέθοδος που δίνει αντίγραφο του αντικείμενου-συναρτησοειδούς που χρησιμοποιεί η κλάση:

  `C A<K,C>::key_comp() const;`

  Για τη μέθοδο αυτή θα συζητήσουμε στη συνέχεια.

Κάθε ένα από τα περιγράμματα έχει τη δική του κλάση *αμφίδρομων* προσεγγιστών για τους οποίους ισχύουν τα εξής:

♦ ***Οι πράξεις διαγραφής δεν ακυρώνουν προσεγγιστές εκτός από αυτούς που δείχνουν στοιχεία που διαγράφονται.***

Στη συνέχεια και για κάθε περίγραμμα θα δούμε τις σχετικές πράξεις εισαγωγής (*insert()*). Αλλά από τώρα θα τονίσουμε ότι:

♦ ***Οι πράξεις εισαγωγής δεν ακυρώνουν προσεγγιστές.***

26.3.1 Το Περίγραμμα "*set*"

Το περίγραμμα "*set*" με επικεφαλίδα:

```
template < class Key, class Compare = less<Key>,
           class Allocator = allocator<Key> >
class set;
```

«φιλοξενεί» συλλογές αντικειμένων που είναι **σύνολα** (sets). Αυτό σημαίνει ότι σε κάθε συλλογή το κάθε αντικείμενο είναι *μοναδικό*.

Δεν υπάρχουν μέθοδοι για το *set* πέρα από αυτές που είδαμε για τα συνειρμικά περιέχοντα γενικώς.

Για να το χρησιμοποιήσεις θα πρέπει να γράψεις στο πρόγραμμά σου:

```
#include <set>
```

Με τη:

```
set< T >  s;
```

δηλώνουμε το *s* ως σύνολο με στοιχεία τύπου *T*. Η τιμή εκκίνησης του *s* είναι το ∅. Π.χ. οι

```
   set< int > intSet;
// intSet == { }
   cout << "#intSet == " << intSet.size() << endl;
```

---

[13] Το περίγραμμα (*std*::)*pair* είναι ίδιο με το *PairT* που είδαμε στην §25.3.1.

```
   if ( intSet.empty() )  cout<< "intSet == {}" << endl;
                    else  cout<< "intSet != {}" << endl;
```

θα δώσουν:

```
#intSet == 0
intSet == {}
```

- Το "`0`" είναι ο **πληθάριθμος** (cardinality) του *intSet* και μας το δίνει η *intSet*.*size*`()`.
- Η *intSet*.*empty*`()` δίνει `true` που σημαίνει «Ναι, το *intSet* είναι κενό.»

Με την *intSet*.*clear*`()` δίνεις τιμή ∅ στο *intSet*.

Μπορείς να δώσεις ως αρχική τιμή μια υποπεριοχή είτε από συνήθη πίνακα είτε από κάποιο άλλο περιέχον της STL:

```
   int             intArr[] = { 34, 72, -23, 8, -11, 31, 41 };
   vector< int >   intVec( intArr+3, intArr+7 );
// . . .
   set< int > intSet1( intVec.begin(), intVec.end() );
   copy( intSet1.begin(), intSet1.end(),
                                ostream_iterator<int>(cout, " ") );
   cout << endl;
   set< int > intSet2( intArr, intArr+5 );
   copy( intSet2.begin(), intSet2.end(),
                                ostream_iterator<int>(cout, " ") );
   cout << endl;
```

Αποτέλεσμα:

```
-11 8 31 41
-23 -11 8 34 72
```

Πρόσεξε ότι τα στοιχεία των συνόλων μας δίνονται ταξινομημένα κατ' αύξουσα τάξη χωρίς να το ζητήσουμε. Στο θέμα αυτό θα επανέλθουμε στη συνέχεια.

Ο έλεγχος για να δούμε αν μια τιμή x ανήκει σε ένα σύνολο s ($x \in s$) μπορεί να γίνει με τρεις τρόπους:

- Με τη μέθοδο *lower_bound()*. Η "`*(s.lower_bound(x)) == x`" θα επιστρέψει `true` αν η τιμή x υπάρχει στο s. Οι

```
   if ( *(intSet1.lower_bound(31)) == 31 )
      cout << "31 in intSet1" << endl;
   else
      cout << "31 not in intSet1" << endl;
   if ( *(intSet1.lower_bound(61)) == 61 )
      cout << "61 in intSet1" << endl;
   else
      cout << "61 not in intSet1" << endl;
```

θα δώσουν:

```
31 in intSet1
61 not in intSet1
```

- Με τη μέθοδο *find()*. Η "`s.find(x)`" θα επιστρέψει "`s.end()`" αν η τιμή x δεν υπάρχει στο s. Αλλιώς θα επιστρέψει προσεγγιστή *it*, τύπου `set<int> ::iterator` (το "`int`" για το παράδειγμά μας), τέτοιον ώστε **it* == x. Τα ίδια αποτελέσματα με αυτά που πήραμε παραπάνω θα δώσουν και οι

```
   if ( intSet1.find(31)==intSet1.end() )
      cout << "31 not in intSet1" << endl;
   else
      cout << "31 in intSet1" << endl;
   if ( intSet1.find(61)==intSet1.end() )
      cout << "61 not in intSet1" << endl;
   else
      cout << "61 in intSet1" << endl;
```

- Με τη μέθοδο *count()*. Η "`s.count(x)`" θα επιστρέψει "`0`" αν η τιμή x δεν υπάρχει στο s· αλλιώς θα επιστρέψει "`1`". Οι

```
   if ( intSet1.count(31)==0 )  cout << "31 not in intSet1" << endl;
```

```
                              else  cout << "31 in intSet1" << endl;
   if ( intSet1.count(61)==0 )  cout << "61 not in intSet1" << endl;
                              else  cout << "61 in intSet1" << endl;
```

θα δώσουν τα ίδια αποτελέσματα.

Αν θέλεις μπορείς να δοκιμάσεις τα παραπάνω και με το στιγμιότυπο "`set<Date>`"· χρησιμοποίησε τον πίνακα "`dates`" που είδαμε στο παράδειγμα για την ταξινόμηση περιεχομένων λίστας.

Για να χειρίζεσαι ένα σύνολο έχεις ακόμη:

- Τις μεθόδους *erase()* –που είδαμε για τα συνειρμικά περιέχοντα– και επιτρέπουν τη διαγραφή στοιχείων από σύνολο
- Μεθόδους *insert()* που επιτρέπουν εισαγωγή στοιχείων σε σύνολο κλάσης `set<K>`:

```
pair< set<K>::iterator, bool > insert( const K& x );
set<K>::iterator insert( set<K>::iterator position, const K& x );
template < class InputIterator >
void insert( InputIterator first, InputIterator last );
```

Η πρώτη μορφή επιστρέφει ζεύγος (*pair*) του οποίου

  - Το μέλος *first* είναι τύπου `set<K>::iterator` και δείχνει το μέλος στο οποίο βρίσκεται η τιμή *x*.
  - Το μέλος *second* είναι τύπου `bool` και έχει τιμή `true` αν έγινε η εισαγωγή ή`false` αν δεν έγινε εισαγωγή διότι η τιμή υπήρχε.

Η δεύτερη μορφή είναι λίγο «περίεργη»: στην "`insert(it,v)`" με την πρώτη παράμετρο *it* κάνουμε μια υπόδειξη για το *πού* πρέπει να γίνει η εισαγωγή της *v*! Βεβαίως η εισαγωγή θα γίνει εκεί που πρέπει να γίνει αλλά αν ο *it* δείχνει *ακριβώς πριν*[14] από τη θέση που θα γίνει η εισαγωγή κερδίζουμε χρόνο. Επιστρέφει τιμή τύπου `set<K>::iterator` που δείχνει το μέλος στο οποίο βρίσκεται η τιμή *x*.

Με την τρίτη μορφή εισάγουμε στο σύνολο όλες τις τιμές μιας περιοχής [*first*, *last*).

Από τις εντολές:

```
   set<int> intSet;
   pair< set<int>::iterator, bool > retVal( intSet.insert(17) );
   cout << boolalpha
        << *(retVal.first) << "  " << retVal.second << endl;
```

παίρνουμε:

```
17  true
```

Ο προσεγγιστής `retVal.first` δείχνει το στοιχείο του *intSet* που έχει τιμή "`17`". Η τιμή `true` του `retVal.second` μας λέει ότι η εισαγωγή αυτής της τιμής έγινε τώρα.

Αν

```
   int      intArr[] = { 34, 72, -23, 8, -11, 31, 41 };
```

από τις εντολές:

```
   intSet.insert( intArr, intArr+4 );
   copy( intSet.begin(), intSet.end(),
                              ostream_iterator<int>(cout, " ") );
   cout << endl;
```

παίρνουμε:

```
-23 8 17 34 72
```

Μετά από αυτές δίνουμε:

```
   retVal = intSet.insert( 8 );
   cout << *(retVal.first) << "  " << retVal.second << endl;
```

και παίρνουμε:

---

[14] Με το νέο πρότυπο C++11 έχουμε βέλτιστη ταχύτητα αν ο *it* δείχνει *ακριβώς μετά* τη θέση που θα γίνει η εισαγωγή.

```
8 false
```

Και εδώ ο `retVal.first` δείχνει στοιχείο με την τιμή που εισάγουμε αλλά η `retVal.second` έχει τιμή `false`: το "`8`" υπάρχει ήδη στο *intSet*.

Συνεχίζουμε δίνοντας:

```
set<int>::iterator it;
it = intSet.insert( retVal.first, 11 );
cout << *it << endl;
```

και παίρνουμε:

```
11
```

Δηλαδή: χρησιμοποιώντας τη δεύτερη μορφή της *insert()* εισάγουμε την τιμή "`11`". Ως πρώτο όρισμα βάζουμε τον προσεγγιστή που πήραμε από την εισαγωγή του "`8`". Η αποπαραπομπή του προσεγγιστή που μας επιστρέφεται μας δίνει ακριβώς το "`11`".

Ε, τώρα ας κάνουμε και μερικές διαγραφές:

```
int n( intSet.erase(72) );
cout << n << endl;
copy( intSet.begin(), intSet.end(),
                          ostream_iterator<int>(cout, " ") );
cout << endl;
intSet.erase( intSet.begin(), intSet.find(17) );
copy( intSet.begin(), intSet.end(),
                          ostream_iterator<int>(cout, " ") );
cout << endl;
```

που δίνουν:

```
1
-23 8 11 17 34
17 34
```

Η "`intSet.erase(72)`" επιστρέφει (και κάνει τιμή της *n*) το πλήθος των στοιχείων που διαγράφηκαν. Το "`1`" της πρώτης γραμμής μας λέει ότι διαγράφηκε ένα στοιχείο· δηλαδή το "`72`" υπήρχε και διαγράφηκε.

Η επόμενη «μαζική» διαγραφή διαγράφει τα στοιχεία του συνόλου από την αρχή μέχρι να βρει το "`17`".

Τέλος, να επισημάνουμε κάτι για τους προσεγγιστές του *set*: Είτε γράψεις "`set<K>::iterator it`" είτε γράψεις "`set<K>::const_iterator it`" *δεν επιτρέπεται να τροποποιήσεις το *it*. Γιατί; Διότι *τα περιεχόμενα ενός set είναι κλειδιά και* –όπως έχουμε τονίσει και σε προηγούμενα κεφάλαια– *τα κλειδιά δεν τροποποιούνται*. Αν επιτρεπόταν η τροποποίηση του **it* θα ήταν πολύ απλό να καταστρέψεις και τη μοναδικότητα των στοιχείων του συνόλου και τη διάταξή τους.[15] Ο «νόμιμος» τρόπος να τροποποιήσεις το **it* είναι ο εξής:

```
   K temp( *it );
   oneSet.erase( it );
// άλλαξε την τιμή της temp
   oneSet( temp );
```

Αν ο *K* είναι τύπος με μεγάλα αντικείμενα αυτός ο τρόπος γίνεται χρονοβόρος. Για την περίπτωση αυτήν υπάρχει άλλο περιέχον για παράσταση συνόλου: το "*map*" που θα δούμε στη συνέχεια.

26.3.1.1 Σχέσεις και Πράξεις Συνόλων

Η σχέση "$\subseteq$" («είναι υποσύνολο», «περιλαμβάνεται») καθώς και οι πράξεις "$\cup$" (ένωση), "$\cap$" (τομή), "\" (διαφορά) και "**Δ**" (συμμετρική διαφορά) υλοποιούνται με περιγράμματα συναρτήσεων. Για να τα χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου "`#include <algorithm>`".

[15] Ο μεταγλωττιστής Borland C++ v.5.5 *επιτρέπει* τροποποίηση του **it*! Αν τον έχεις δοκίμασε να κάνεις τις «παρανομίες» που λέμε.

Η σχέση $A \subseteq B$ (διαβάζεται ως «B includes A») υλοποιείται με το περίγραμμα *includes()* που ορίζεται στο **algorithm**:

```
template< class InputIter1, class InputIter2 >
bool includes( InputIter1 first1, InputIter1 last1,
               InputIter2 first2, InputIter2 last2 );
template< class InputIter1, class InputIter2,
          class Compare >
bool includes( InputIter1 first1, InputIter1 last1,
               InputIter2 first2, InputIter2 last2,
               Compare comp);
```

Με τις:

```
#include <set>
#include <algorithm>
// . . .
   int      intArr[] = { 34, 72, -23, 8, -11, 31, 41 };
// . . .
   set<int> intSet1( intArr, intArr+3 );
   copy( intSet1.begin(), intSet1.end(),
         ostream_iterator<int>(cout, " ") );
   cout << endl;
   set<int> intSet2( intArr, intArr+7 );
   copy( intSet2.begin(), intSet2.end(),
         ostream_iterator<int>(cout, " ") );
   cout << endl;
   if ( includes(intSet2.begin(), intSet2.end(),
                 intSet1.begin(), intSet1.end()) )
      cout << "intSet2 includes intSet1" << endl;
   else
      cout << "intSet2 does not include intSet1" << endl;
```

θα πάρουμε:

```
-23 34 72
-23 -11 8 31 34 41 72
intSet2 includes intSet1
```

Η σχέση $A \subset B$ –**γνήσιο υποσύνολο** (proper ή strict subset)– δεν υλοποιείται απ' ευθείας· θα πρέπει να τη δεις ως ($A \subseteq B$) && (A != B).

Εδώ όμως θα πρέπει να παρατηρήσουμε μερικά πράγματα:

- Οι τύποι των παραμέτρων βάζουν πολύ λίγους περιορισμούς: είναι απλώς *προσεγγιστές εισόδου*, δηλαδή ό,τι πιο γενικό. Πουθενά δεν φαίνεται ότι θα πρέπει να είναι προσεγγιστές συνόλου. Στην πραγματικότητα γίνεται σύγκριση των περιεχομένων δύο περιοχών που μπορεί να βρίσκονται σε οποιοδήποτε περιέχον STL, σε πίνακα ή ακόμη και στα δεδομένα εισόδου! Πράγματι, αν γράψουμε:

```
// . . .
   istream_iterator< int > cinIt( cin ), cinEoStream;
   if ( includes(intSet2.begin(), intSet2.end(),
                 cinIt, cinEoStream) )
      cout << "intSet2 includes input" << endl;
// . . .
```

και κατά την εκτέλεση πληκτρολογίσουμε:

-11 8 31 e

θα πάρουμε:

```
intSet2 includes input
```

Αν γράψουμε:

```
// . . .
   copy( intArr+4, intArr+7, ostream_iterator<int>(cout, " ") );
   cout << endl;
   if ( includes(intSet2.begin(), intSet2.end(),
                 intArr+4, intArr+7) )
      cout << "intSet2 includes [intArr+4, intArr+7)" << endl;
```

```
// . . .
```

θα πάρουμε:

```
-11 31 41
intSet2 includes [intArr+4, intArr+7)
```

Τέλος, αν γράψουμε:

```
   deque<int> intDeque( intArr+1, intArr+6 );
   sort( intDeque.begin(), intDeque.end() );
   copy( intDeque.begin(), intDeque.end(),
         ostream_iterator<int>(cout, " ") );
   cout << endl;
   if ( includes(intSet2.begin(), intSet2.end(),
                 intDeque.begin(), intDeque.end()) )
      cout << "intSet2 includes intDeque" << endl;
```

παίρνουμε:

```
-23 -11 8 31 72
intSet2 includes intDeque
```

- Πρόσεξε την τελευταία περίπτωση: πριν καλέσουμε την *includes()* καλούμε τη *sort()* για να ταξινομήσει τα περιεχόμενα της *intDeque*. Αυτό είναι ουσιώδες: Αν δεν το κάνουμε θα πάρουμε λάθος αποτελέσματα:

```
72 -23 8 -11 31
intSet2 does not include intDeque
```

Για να καταλάβεις τι γίνεται αντιγράφουμε –«ξεκαθαρίζοντας» κάπως– το περίγραμμα *includes()* από τη gcc (αρχείο `stl_algo.h`):[16]

```
 0: template< typename InputIter1, typename InputIter2 >
 1: bool includes( InputIter1 first1, InputIter1 last1,
 2:                InputIter2 first2, InputIter2 last2 )
 3: {
 4:    while( first1 != last1 && first2 != last2 )
 5:       if( *first2 < *first1 )
 6:          return false;
 7:       else if( *first1 < *first2 )
 8:          ++first1;
 9:       else
10:       {  ++first1; ++first2;  }
11:
12:    return first2 == last2;
13: }
```

Όπως καταλαβαίνεις, το πρόβλημα είναι η σύγκριση της γραμμής 5: "`34 < 72`" και δεν υπάρχει αν κάνουμε την ταξινόμηση.

Στη δεύτερη μορφή του περιγράμματος η γραμμή 5 γίνεται:

```
if( comp(*first2,*first1) )
```

και η γραμμή 7:

```
else if( comp(*first1,*first2) )
```

Προϋποθέσεις για τον αλγόριθμο *includes()*: Οι δύο περιοχές

- Περιέχουν στοιχεία του ίδιου τύπου.[17]
- Τα στοιχεία είναι ταξινομημένα.
- Η ταξινόμηση έγινε με το ίδιο κατηγόρημα *comp()*.

Τα παραπάνω ισχύουν και για τις πράξεις μεταξύ συνόλων που βλέπουμε στη συνέχεια. Αν βάλεις στο πρόγραμμά σου "`#include <algorithm>`" μπορείς να χρησιμοποιείς και τις:

---

[16] Κάτι σου θυμίζει; Ξαναδές τον αλγόριθμο της συγχώνευσης (§9.5.3).

[17] Και τα στοιχεία είναι μοναδικά για την καθεμιά; Η αλήθεια είναι ότι αλγόριθμος δουλεύει και χωρίς τη μοναδικότητα, π.χ. και για πολυσύνολα.

- (*std*::)*set_union* (ένωση συνόλων),
- (*std*::)*set_intersection* (τομή συνόλων),
- (*std*::)*set_difference* (διαφορά συνόλων) και
- (*std*::)*set_symmetric_difference* (συμμετρική διαφορά συνόλων).[18]

Με αυτές όμως έχουμε και ένα άλλο πρόβλημα. Ας πάρουμε, για παράδειγμα, το:

```
template< class InputIter1, class InputIter2, class OutputIterator >
OutputIterator set_union( InputIter1 first1, InputIter1 last1,
                          InputIter2 first2, InputIter2 last2,
                          OutputIterator result );
template< class InputIter1, class InputIter2,
          class OutputIterator, class Compare >
OutputIterator set_union( InputIter1 first1, InputIter1 last1,
                          InputIter2 first2, InputIter2 last2,
                          OutputIterator result, Compare comp );
```

Το νέο στοιχείο εδώ είναι ότι έχουμε *αποτέλεσμα* που είναι (κατ' αρχήν) *σύνολο*. Γιατί «κατ' αρχήν»; Διότι ναι μεν είναι σύνολο αλλά μπορείς να το αποθηκεύσεις όπου σε βολεύει.

Με την παράμετρο *result* περνούμε την *αρχή* της περιοχής όπου θα αποθηκευτεί το απότέλεσμα. Η συνάρτηση επιστρέφει τιμή-προσεγγιστή προς το *τέλος* της περιοχής όπου έγινε η αποθήκευση. *Δεν πρέπει να υπάρχει επικάλυψη της περιοχής που θα πάρει το αποτέλεσμα με οποιαδήποτε από τις περιοχές που ενώνονται.*

Ας πούμε ότι έχουμε δύο σύνολα:

```
set< int > intSet1, intSet2;
```

και τους δίνουμε τιμές { 4, 5, 6 } και { 2, 4, 6, 8, 10 } αντιστοίχως. Μπορούμε να πάρουμε την ένωσή τους κατ' ευθείαν στο *cout*:

```
set_union( intSet1.begin(), intSet1.end(),
           intSet2.begin(), intSet2.end(),
           ostream_iterator<int>(cout, " ") );
cout << endl;
```

και να δούμε στην οθόνη μας:

```
2 4 5 6 8 10
```

Μπορούμε να πάρουμε την ένωσή τους στο:

```
vector<int> intVec1(11);
```

με την εντολή:

```
set_union( intSet1.begin(), intSet1.end(),
           intSet2.begin(), intSet2.end(),
           intVec1.begin() );
copy( intVec1.begin(), intVec1.end(),
      ostream_iterator<int>(cout, " ") );
cout << endl;
```

Αποτέλεσμα:

```
2 4 5 6 8 10 0 0 0 0 0
```

Τι είναι αυτά τα μηδενικά; Για τον πίνακα έχουμε κρατήσει 11 θέσεις ενώ η ένωση έχει μόνον 6. Δηλαδή αν δηλώσουμε:

```
vector<int> intVec1;
```

δεν θα βγούνε; Δεν θα βγούνε αλλά θα πρέπει να χρησιμοποιήσουμε τον προσαρμογέα που μάθαμε στην §26.1.3:

```
back_insert_iterator< vector<int> > destV( intVec1 );
```

και να δώσουμε:

---

[18] Η **συμμετρική διαφορά** δύο συνόλων A, B ορίζεται ως $A \Delta B = (A \setminus B) \cup (B \setminus A)$· δηλαδή τα στοιχεία του A που δεν ανήκουν στο B και τα στοιχεία του B που δεν ανήκουν στο A. Άρα: $A \Delta B = (A \cup B) \setminus (A \cap B)$.

```
    set_union( intSet1.begin(), intSet1.end(),
               intSet2.begin(), intSet2.end(),
               destV );
```

Και αν θέλουμε να αποθηκεύσουμε το αποτέλεσμα σε ένα

```
    set<int> intSet3;
```

θα κάνουμε το ίδιο; Όχι, αλλά κάτι παρόμοιο: Το περίγραμμα *back_insert_iterator* μπορεί να δώσει στιγμιότυπα για κλάσεις που έχουν την *push_back()*. Ένα άλλο περίγραμμα-προσαρμογέας που μπορεί να δώσει στιγμιότυπα για κάθε κλάση που έχει την *insert()* είναι ο *inserter*. Για να τον χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου "`#include <iterator>`".

Αν δώσουμε:

```
    set_union( intSet1.begin(), intSet1.end(),
               intSet2.begin(), intSet2.end(),
               inserter(intSet3, intSet3.begin()) );
```

η ένωση θα φυλαχθεί στο *intSet3*.

Με τον ίδιο τρόπο δηλώνονται και οι άλλες συναρτήσεις (περιγράμματα) για σύνολα. Για παράδειγμα, για την τομή έχουμε:

```
template< class InputIter1, class InputIter2, class OutputIter >
OutputIterator set_intersection(InputIter1 first1, InputIter1 last1,
                                InputIter2 first2, InputIter2 last2,
                                OutputIter result );
template< class InputIter1, class InputIter2,
          class OutputIter, class Compare >
OutputIterator set_intersection(InputIter1 first1, InputIter1 last1,
                                InputIter2 first2, InputIter2 last2,
                                OutputIter result, Compare comp );
```

Ο χειρισμός τους γίνεται με τον ίδιο τρόπο.

26.3.2 Το Περίγραμμα "*map*"

Το περίγραμμα "*set*" δεν είναι το μοναδικό εργαλείο της STL για παράσταση συνόλου. Υπάρχει και η **απεικόνιση** (map) που έχει την επικεφαλίδα:

```
template < class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T> > >
class map;
```

Για να τη χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου

```
#include <map>
```

Σε σύγκριση με το "*set*" υπάρχει μια επί πλέον παράμετρος: "`class T`". Τι είναι αυτό; Τα περιεχόμενα του περιγράμματος είναι οργανωμένα σε ζεύγη τύπου:

```
    pair< Key, T >
```

Ο πρώτος τύπος (*Key*) είναι ο γνωστός μας (από τα σύνολα) *τύπος κλειδιών* (key type)· ο δεύτερος (*T*) είναι ο **τύπος τιμών** (value type). Έτσι, οι δυο συνιστώσες του ζεύγους είναι:

```
    Key first;
    T   second;
```

Μήπως το "`map<Key, T>`" είναι ίδιο με το "`set< pair<Key, T> >`"; Όχι! Όταν το ζεύγος μπαίνει στο περιέχον (`map`) η θέση του στη διάταξη καθορίζεται από την τιμή του *first* που μπαίνει στο δένδρο αναζήτησης και *δεν* (πρέπει να) *αλλάζει*.[19] Στο *second* βάζουμε όλα τα άλλα στοιχεία που *έχουμε τη δυνατοτητα να τροποποιούμε*. Στο "`set< pair<Key, T> >`" δεν έχουμε δυνατότητα για οποιαδήποτε τροποποίηση.

---

[19] Το «πρέπει να» για τους μεταγλωττιστές –σαν τον Borland C++ v.5.5– που αφήνουν την περιφρούρηση της δομής του συνόλου αποκλειστικώς στον προγραμματιστή.

Παράδειγμα
Γυρνάμε πίσω, στο Project 4, στην κλάση *CourseCollection* και αναγνωρίζουμε ότι έχουμε ένα σύνολο αντικειμένων *Course*. Θα πρέπει να αντικαταστήσουμε τη δήλωση "`Course* ccArr`" με μια δήλωση *map* αφού κάνουμε την κατάλληλη προεργασία.

- Ο τύπος κλειδιών είναι έτοιμος:

```
struct CourseKey
{
   enum { cCodeSz = 8 };
   char s[cCodeSz];
   explicit CourseKey( string aKey="" )
   {  strncpy( s, aKey.c_str(), cCodeSz-1 );
      s[cCodeSz-1] = '\0';  }
}; // CourseKey
```

- Όλα τα υπόλοιπα μπαίνουν στον τύπο τιμών:

```
class CourseVal
{
public:
   enum { cTitleSz = 80, cCategSz = 4 };
   explicit CourseVal( string aTitle="" );
// . . .
   void display( ostream& tout ) const;
private:
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
// . . .
   unsigned int cNoOfStudents;    // αριθ. φοιτητών
}; // Course
```

- Ο τύπος των ζευγών τιμών θα είναι:

```
   pair< CourseKey, CourseVal >
```

- Αφού για τον *CourseKey* δεν έχουμε επιφορτώσει τον "`<`" θα πρέπει να γράψουμε το κατάλληλο συναρτησοειδές:

```
struct CodeLT
{
   bool operator()( const CourseKey& ck1, const CourseKey& ck2 )
   {  return ( strcmp(ck1.s, ck2.s) < 0 );  }
}; // CodeLT
```

Έτσι, θα δηλώσουμε το σύνολό μας ως:

```
   map< CourseKey, CourseVal, CodeLT >  ccCourses;
```

Αν έχουμε:

```
   CourseKey aCk( "ΕΥ02010" );
   CourseVal aCv( "Αντικειμενοστρεφής Προγραμματισμός (Θ)" );
```

μπορούμε να εισαγάγουμε στο *ccCourses*:

```
   ccCourses.insert( make_pair(aCk, aCv) );
```

Αν

```
   map<CourseKey, CourseVal, CodeLT>::iterator it;
```

και ο *it* δείχνει το αντικείμενο που βάλαμε μπορούμε να κάνουμε τροποποίηση του *ccCateg*:

```
   (it->second).setCateg( "ΜΕΥ" );
```

Ο τελεστής "`[]`" –που δεν επιφορτώσαμε για την *CourseCollection* διότι μας φάνηκε «παράξενο» το "`allCourses["ΕΥ02010"]`"– επιφορτώνεται για το *map*. Δες το παρακάτω

Παράδειγμα
Στο παρακάτω πρόγραμμα έχουμε μια απεικόνιση από τα ονόματα των μηνών στα πλήθη ημερών τους.

```
  0: #include <iostream>
  1: #include <string>
```

```
 2: #include <map>
 3:
 4: using namespace std;
 5:
 6: int main()
 7: {
 8:    string monthN[] = { "Jan", "Feb", "Mar", "Apr",
 9:                        "May", "Jun", "Jul", "Aug",
10:                        "Sep", "Oct", "Nov", "Dec" };
11:    unsigned int noOfDays[] = { 31, 28, 31, 30, 31, 30,
12:                                31, 31, 30, 31, 30, 31 };
13:    map< string, unsigned int > months;
14:
15:    for ( int k(0); k < 6; ++k )
16:    {
17:       pair<string, unsigned int> amp;
18:       amp.first = monthN[k];
19:       amp.second = noOfDays[k];
20:       months.insert( amp );
21:    }
22:    for ( int k(6); k < 12; ++k )
23:       months[monthN[k]] = noOfDays[k];
24:
25:    for ( map<string, unsigned int>::const_iterator
26:                                   cit( months.begin() );
27:          cit != months.end(); ++cit )
28:       cout << cit->first << ", " << cit->second << endl;
29: }
```

Στις γραμμές 15-21 εισάγουμε τους πρώτους 6 μήνες δημιουργώντας τα κατάλληλα ζεύγη `pair<string, unsigned int>` και χρησιμοποιώντας τη μέθοδο *insert()*.

Στις γραμμές 22-23 εισάγουμε τους τελευταίους 6 μήνες αλλιώς:

```
   months["Jul"] = 31;
   months["Aug"] = 31;
// . . .
   months["Dec"] = 31;
```

Το “`months["Jul"] = 31`” είναι ισοδύναμο με “`months.insert(make_pair ("Jul",31))`”.

Γράφουμε στο *cout* το περιεχόμενο στηριγμένοι στο ότι ο προσεγγιστής *cit* δείχνει ζεύγος που έχει ως *first* το όνομα του μήνα και ως *second* το πλήθος των ημερών. Το πρόγραμμα θα δώσει:

```
Apr, 30
Aug, 31
Dec, 31
Feb, 28
Jan, 31
Jul, 31
Jun, 30
Mar, 31
May, 31
Nov, 30
Oct, 31
Sep, 30
```

Οι μήνες βγαίνουν κατ’ αλφαβητική σειρά. Αν αντικαταστήσουμε τις γραμμές 25-28 με τις:

```
   for ( int k(0); k < 12; ++k )
      cout << monthN[k] << ", " << months[monthN[k]] << endl;
```

όπου χρησιμοποιούμε τον “`[]`”, θα πάρουμε:

```
Jan, 31
Feb, 28
Mar, 31
Apr, 30
```

```
May, 31
Jun, 30
Jul, 31
Aug, 31
Sep, 30
Oct, 31
Nov, 30
Dec, 31
```

❦❦❦

Εκτός από την *insert()* που είδαμε στα δύο παραδείγματα υπάρχουμε και για το *map* οι άλλες δύο μορφές που είδαμε για το *set*.

Τα περιγράμματα *includes()*, *set_union()*, *set_intersection()*, *set_difference()* και *set_symmetric_difference()* δουλεύουν για τα στιγμιότυπα του *map* όπως για τα στιγμιότυπα του *set*.

26.3.3 Τα Περιγράμματα "*multiset*" και "*multimap*"

Βάζοντας στο πρόγραμμά σου την "`#include <set>`" έχεις τη δυνατότητα να χρησιμοποιήσεις και το περίγραμμα "*multiset*" με επικεφαλίδα:

```
template < class Key, class Compare = less<Key>,
           class Allocator = allocator<Key> >
class multiset;
```

Κάθε στιγμιότυπο του περιγράμματος έχει αντικείμενα που είναι **πολυσύνολα** (multisets, bags). Σε αντιδιαστολή με το σύνολο, ένα πολυσύνολο μπορεί να έχει πολλαπλά αντίγραφα των μελών του. Έτσι, ενώ η σχέση "*ανήκει*" ορίζεται και για πολυσύνολα ($x \sqsubset m$)[20], πιο πολύ νόημα έχει η συνάρτηση **πολλαπλότητα** (multiplicity, multitude) που στο *multiset* υλοποιείται με την *count()*.

Όπως στα σύνολα έτσι και εδώ με την:

```
multiset< int >  mulSet1;
```

δημιουργούμε ένα κενό πολυσύνολο (⟦ ⟧). Οι εντολές

```
cout << "#mulSet1 == " << mulSet1.size() << endl;
if ( mulSet1.empty() )  cout<< "mulSet1 == [[]]" << endl;
                  else  cout<< "mulSet1 != [[]]" << endl;
```

δίνουν:

```
#mulSet1 == 0
mulSet1 == [[]]
```

Θέλοντας να παραστήσουμε το ⟦ 1, 2, 4, 3, 2, 4, 4 ⟧ στο πρόγραμμά μας δίνουμε:

```
int intArr[] = { 1, 2, 4, 3, 2, 4, 4 };

multiset< int >  mulSet2( intArr, intArr+7 );
copy( mulSet2.begin(), mulSet2.end(),
                          ostream_iterator<int>(cout, " ") );
cout << endl;
```

Αποτέλεσμα:

```
1 2 2 3 4 4 4
```

Για να δούμε τις πολλαπλότητες των "2" και "7" στο *mulSet2* δίνουμε:

```
cout << "mulSet2 # 2 = " << mulSet2.count(2) << endl;
cout << "mulSet2 # 7 = " << mulSet2.count(7) << endl;
```

και παίρνουμε:

```
mulSet2 # 2 = 2
mulSet2 # 7 = 0
```

Για το *multiset* υπάρχουν τρεις μέθοδοι *insert()*:

[20] Συμβολισμός της γλώσσας προδιαγραφών Z.

```
multiset<K>::iterator multiset<K>::insert( const K& x );
multiset<K>::iterator multiset<K>::insert( multiset<K>::iterator pos,
                                           const K& x );
template < class InputIterator >
void insert( InputIterator first, InputIterator last );
```

Με την πρώτη μορφή εισάγουμε στο πολυσύνολο την τιμή *x*. Αν η τιμή υπάρχει ήδη θα αυξηθεί η πολλαπλότητά της κατά 1. Επιστρέφει προσεγγιστή προς το νεοεισαχθέν στοιχείο.

Και με τη δεύτερη μορφή εισάγουμε στο πολυσύνολο την τιμή *x* και μας επιστρέφει προσεγγιστή προς το νεοεισαχθέν στοιχείο. Η πρώτη παράμετρος (*pos*) μπορεί να επιταχύνει την εισαγωγή, όπως λέγαμε και στα σύνολα.

Με την τρίτη μορφή εισάγουμε όλα τα στοιχεία της περιοχής [*first*, *last*).

Τα περιγράμματα *includes()*, *set_union()*, *set_intersection()*, *set_difference()* και *set_symmetric_difference()* δουλεύουν και για το *multiset*.

Παρατηρήσεις:▶

1. Ως *ένωση* (*U*) και *τομή* (*I*) δύο πολυσυνόλων M_1 και M_2 ορίζονται τα πολυσύνολα που έχουν $U \# v = max(M_1 \# v, M_2 \# v)$ και $I \# v = min(M_1 \# v, M_2 \# v)$.

Στη βιβλιογραφία (π.χ. στη γλώσσα Z) θα βρεις να αναφέρεται ως «ένωση» και αυτό που άλλοι ονομάζουν «άθροισμα πολυσυνόλων» (bag sum) και έχει $S \# v = M_1 \# v + M_2 \# v$.

2. Ως *διαφορά* (*D*) των πολυσυνόλων $M_1 \setminus M_2$ ορίζεται το πολυσύνολο που έχει $D \# v = (M_1 \# v > M_2 \# v)\ ?\ M_1 \# v - M_2 \# v : 0$.◀

Με την "`#include <map>`" στο πρόγραμμά σου έχεις τη δυνατότητα να χρησιμοποιήσεις και το περίγραμμα:

```
template < class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T> > >
class multimap;
```

Όπως στο "*map*" έτσι και στο "*multimap*" τα περιεχόμενα είναι οργανωμένα σε ζεύγη τύπου:

```
pair< Key, T >
```

26.3.4 Διάταξη Στοιχείων

Τα στοιχεία ενός συνόλου (ή ενός πολυσυνόλου) δεν είναι κατ' ανάγκη διαταγμένα. Αλλά, όπως θα παρατήρησες σε όλα τα παραδείγματα που δώσαμε τα στοιχεία συνόλων και πολυσυνόλων βγαίνουν ταξινομημένα κατ' αύξουσα τάξη χωρίς να το ζητήσουμε. Γιατί; Διότι *«για να έχουμε γρήγορη αναζήτηση τα κλειδιά οργανώνονται –με βάση τη διάταξη "*`<`*" που είδαμε πιο πάνω– σε ένα δένδρο δυαδικής αναζήτησης.»* Έτσι, όταν δηλώνεις σύνολο με στοιχεία τύπου *K* θα πρέπει στον τύπο αυτόν να είναι ορισμένη η διάταξη "`<`".

Πάντως είναι δυνατόν να ανατρέψεις αυτήν τη διάταξη ορίζοντας δική σου όπως θα δεις στο παρακάτω παράδειγμα:

```
int      intArr[] = { 34, 72, -23, 8, -11, 31, 41 };

set< int >  intSet( intArr, intArr+7 );
copy( intSet.begin(), intSet.end(),
                      ostream_iterator<int>(cout, " ") );
cout << endl;

set< int, greater<int> > intSetG( intArr, intArr+7 );
copy( intSetG.begin(), intSetG.end(),
                      ostream_iterator<int>(cout, " ") );
cout << endl;
```

Αποτέλεσμα:

```
-23 -11 8 31 34 41 72
72 41 34 31 8 -11 -23
```

Το δεύτερο σύνολο μας δίνει τα μέλη του κατά φθίνουσα τάξη. Αυτό είναι αποτέλεσμα του συναρτησοειδούς που έχουμε ως δεύτερο όρισμα (`greater <int>`) στο στιγμιότυπο του περιγράμματος.

Για παράδειγμα δες πώς ορίζεται (στο `functional`) το περίγραμμα

```
template < class T >
struct greater : public binary_function<T, T, bool>
{
   bool operator() (const T& x, const T& y) const { return x > y; }
};
```

Μέσα στο πρόγραμμά σου μπορείς να γράψεις: `greater(5,2)` ή `greater(0.5, 1.2)` και να σου δώσουν τιμές `true` και `false` αντιστοίχως. Φυσικά, για να γίνει η εξειδίκευση θα πρέπει να ορίζεται ο τελεστής "`>`" για τον τύπο *T*.

Αν δεν ορίζεται ο τελεστής διάταξης που σε ενδιαφέρει τι κάνεις; Έχεις δύο επιλογές:

- Να τον επιφορτώσεις (αν φυσικά επιφορτώνεται).
- Να γράψεις το δικό σου συναρτησοειδές.

Για τη δεύτερη περίπτωση δες το παρακάτω παράδειγμα:

```
struct cstrGT
{
   bool operator()( const char cs1[], const char cs2[] )
   {
      return strcmp(cs1, cs2) > 0;
   }
};

int main()
{
   set<const char*, cstrGT> css;
   css.insert( "μια" );  css.insert( "κλάση" );
   css.insert( "στην" );  css.insert( "οποία" );
   css.insert( "επιφορτώνουμε" );  css.insert( "τον" );
   css.insert( "τελεστή" );  css.insert( "κλήσης" );
   css.insert( "συνάρτησης" );
   copy( css.begin(), css.end(),
                     ostream_iterator<const char*>(cout, " ") );
   cout << endl;
```

Αποτέλεσμα:

```
τον τελεστή συνάρτησης στην οποία μια κλήσης κλάση επιφορτώνουμε
```

Αν όμως ορίσουμε:

```
struct cstrLT
{
   bool operator()( const char cs1[], const char cs2[] )
   {
      return strcmp(cs1, cs2) < 0;
   }
};
```

δηλώσουμε

```
   set<const char*, cstrLT> css;
```

και εισαγάγουμε τις ίδιες λέξεις παίρνουμε:

```
επιφορτώνουμε κλάση κλήσης μια οποία στην συνάρτησης τελεστή τον
```

Δηλώνοντας:

```
set< T > s;
```

είναι σαν να δηλώνεις

```
set< T, less<T> > s;
```

με την προϋπόθεση ότι στον *T* είναι ορισμένος ο "`<`".

Αυτά που λέμε εδώ για τα στιγμιότυπα του "*set*" εφαρμόζονται και στα άλλα συνειρμικά περιέχοντα και όχι μόνον· τα είδαμε και στα περιγράμματα *min()*, *max()*, *sort()* κλπ. Δηλαδή, εκείνο το *comp()* είναι σαν τον "`<`"; Πρόσεξε:

- Όταν γράφουμε "`a < b`" εννοούμε ότι "το *a* είναι μικρότερο του *b*". Εννοούμε όμως και κάτι άλλο: αν έχουμε αύξουσα διάταξη το *a προηγείται του b*.
- Η παράσταση "`comp(a, b)`" έχει τιμή `true` αν και μόνον αν το *a προηγείται του b*. Δηλαδή το *comp()* καθορίζει τη διάταξη· το «μικρότερο» το ξεχνούμε.

Από την επιφόρτωση του "`<`" ή τον ορισμό του *comp()* έχουμε και υποκατάστατο του "`==`": "`!(a<b) && !(b<a)`" ή "`!comp(a,b) && !comp(b,a)`". Στην περίπτωση αυτή λέμε ότι τα κλειδιά *a* και *b* είναι **ισοδύναμα** (equivalent) κρατώντας τον όρο «*ίσα*» μόνο για την περίπτωση σύγκρισης με τον "`==`" (αν είναι ορισμένος).

Στις παρατηρήσεις για τη *lower_bound()* στην §26.1.3 γράφαμε «*Η comp πρέπει να είναι ίδια (γενικότερα: συμβατή) με αυτήν που χρησιμοποιήθηκε για την ταξινόμηση. Αλλιώς η αναζήτηση δεν γίνεται σωστά.*» Γενικώς:

♦ ***Αν μια συνάρτηση χρησιμοποιεί την ταξινόμηση των στοιχείων ενός περιέχοντος το συναρτησοειδές (comp ή less<T>) διάταξης θα πρέπει να είναι το ίδιο με αυτό που έγινε η ταξινόμηση.***

26.4 Ποιο Περιέχον να Διαλέξω;

Μετά την ερώτηση «*Ποια Ακολουθία να Διαλέξω;*» έρχεται η γενικότερη ερώτηση: «*Ποιο Περιέχον να Διαλέξω;*»

- Για πολλούς η απάντηση είναι η ίδια με αυτήν που θα έδιναν και στην πρώτη ερώτηση: «*vector και πάλι vector! Δεν νομίζω ότι υπάρχει κάτι άλλο!*»
- Άλλοι θα πουν: όπως λέγαμε στην §9.5.1 «*Πολύ συχνά, ένας πίνακας χρησιμοποιείται για την παράσταση κάποιου συνόλου.*» Θα πρέπει λοιπόν να χρησιμοποιούμε
  - το "*set*" αν ολόκληρα τα αντικείμενα των περιεχομένων είναι κλειδιά ή
  - το "*map*".

Η δεύτερη άποψη φαίνεται πιο σωστή αφού έχουμε πιο εύκολη εισαγωγή και διαγραφή στοιχείων· θυμίσου ότι στο σημείο αυτό το "*vector*" υστερεί και ως προς το "*list*". Ακόμη, οι αναζητήσεις στα συνειρμικά περιέχοντα έχουν λογαριθμική πολυπλοκότητα.

Βέβαια και στο "*vector*" μπορείς να έχεις γρήγορες αναζητήσεις αν έχεις ταξινομήσει τα περιεχόμενά του. Ακόμη, το "*vector*" έχει το πλεονέκτημα της αποθήκευσης των στοιχείων του σε συναπτές θέσεις της μνήμης πράγμα που επιτρέπει –εκτός από τη δυαδική αναζήτηση– τη χρήση συναρτήσεων που έχουν γραφεί για απλούς πίνακες. Το μειονέκτημά του είναι η χρονοβόρα εισαγωγή/διαγραφή στοιχείων έτσι ώστε να μην καταστρέφεται η ταξινόμηση.

Ας προσπαθήσουμε να βγάλουμε μερικούς κανόνες για τις επιλογές μας.

Ένας κάπως επιφανειακός αλλά όχι λανθασμένος κανόνας είναι αυτός που είπαμε πιο πάνω:

- Αν έχουμε μια συλλογή αντικειμένων που έχει χαρακτηριστικά συνόλου (μοναδικότητα στοιχείων) χρησιμοποιούμε
  - το "*set*" αν ολόκληρα τα αντικείμενα είναι κλειδιά ή
  - το "*map*".
- Αν η συλλογή μας είναι πολυσύνολο τότε
  - αν έχουμε συχνές εισαγωγές/διαγραφές χρησιμοποιούμε "*multiset*" ή "*multimap*",
  - αλλιώς χρησιμοποιούμε (ταξινομημένο) "*vector*".

Αν σε ενδιαφέρει πολύ η ταχύτητα θα πρέπει να εξετάσεις την περίπτωση του ταξινομημένου “*vector*” και σε σχέση με τα “*set*”/“*map*”. Φυσικά, σε μια τέτοια περίπτωση έχεις να πληρώσεις και κάτι ακόμη: είναι δική σου υποχρέωση να διασφαλίζεις τη μοναδικότητα των μελών του συνόλου.

26.5 Άλλα Περιγράμματα

Στη συνέχεια θα δούμε εν συντομία δύο περιγράμματα κλάσεων που δεν είναι περιέχοντα: *bitset*, *complex*.[21]

26.5.1 Το Περίγραμμα “*bitset*”

Βάζοντας στο πρόγραμμά σου την οδηγία “`#include <bitset>`” μπορείς να χρησιμοποιήσεις το περίγραμμα

```
template < size_t N >
class bitset;
```

που είναι σαν τη δική μας “*Bitmap*” αλλά χωρίς περιορισμούς από το μέγεθος (σε δυαδικά ψηφία) του ακέραιου: το μέγεθος καθορίζεται από την παράμετρο *N* και μπορεί να είναι πολύ μεγάλο.

Έχει κάποια σχέση με το “`vector<bool>`”; Μοιάζει πολύ αλλά και διαφέρει· η βασική διαφορά είναι ότι το “*bitset*”

- δεν είναι δυναμικό –η τιμή της *N* καθορίζεται όταν γίνεται η μεταγλώττιση και μετά δεν αλλάζει–
- δεν έχει προσεγγιστές και φυσικά δεν έχει “*push_back()*”.

Με μεθόδους επιφορτώνονται οι τελεστές “`&=`”, “`|=`”, “`^=`”, “`~`”, “`[]`”, “`<<=`”, “`>>=`”,“`==`”, “`!=`” ενώ με περιγράμματα καθολικών συναρτήσεων επιφορτώνονται οι “`&`”, “`|`”, “`^`”.

Οι “`<<`”, “`>>`” επιφορτώνονται δύο φορές

- με μεθόδους ως τελεστές ολίσθησης:

```
bitset<N> operator<<(size_t pos) const;
bitset<N> operator>>(size_t pos) const;
```

- με περιγράμματα καθολικών συναρτήσεων ως τελεστές εισόδου/εξόδου:

```
template < size_t N >
istream& operator>>( istream& is, bitset<N>& x );
template < size_t N >
ostream& operator<<( ostream& os, const bitset<N>& x );
```

Το περίγραμμα έχει αρκετούς δημιουργούς:

- Ερήμην δημιουργό:

```
bitset();
```

που ξεκινάει όλα τα ψηφία στο “`0`”:

```
bitset<12> bsd;
cout << bsd << endl;
```

Αποτέλεσμα:

```
000000000000
```

- Δημιουργό με αρχική τιμή `unsigned long int`:

```
bitset( unsigned long val );
```

Ας τον δούμε με ένα παράδειγμα: οι

```
bitset<12> bsu0( 21UL ), bsu1( 1000000UL );
cout << bsu0 << endl << bsu1 << endl;
```

[21] Δεν θα δούμε το περίγραμμα *valarray* και τα σχετικά με αυτό.

```
bitset<32> bsu2( 1000000UL );
cout << bsu2 << endl;
```

δίνουν:

```
000000010101
001001000000
00000000000011110100001001000000
```

Στην πρώτη γραμμή βλέπουμε την παράσταση του "21" στο δυαδικό σύστημα (16+4+1) με 12 ψηφία (το λιγότερο σημαντικό ψηφίο στα δεξιά).

Στην τρίτη γραμμή βλέπεις το 1 εκατομμύριο σο δυαδικό σύστημα (32 ψηφία). Και στη δεύτερη τι έχουμε; Τα 12 λιγότερο σημαντικά ψηφία αυτού που βλέπουμε στην τρίτη γραμμή· τόσα χωράνε στη *bsu1*!

Καταλαβαίνεις λοιπόν πώς δουλεύει αυτός ο δημιουργός:

- Αν η δυαδική παράσταση του ορίσματος χωράει στα δυαδικά ψηφία που έχουμε (*N*) όλα πάνε καλά (με πιθανή συμπλήρωση με μηδενικά από αριστερά).
- Αν δεν χωράει αποθηκεύονται τα λιγότερο σημαντικά ψηφία που χωράνε.

- Δημιουργό με αρχική τιμή `string`:

`explicit bitset( const string& str, size_type pos=0, size_type n=npos );`

Το πρέπει να έχει μόνον τους χαρακτήρες "`0`" και "`1`". Αν βρεθεί κάτι άλλο ρίχνεται εξαίρεση (*std*::)*invalid_argument*. Μπορείς να ορίσεις και υποορμαθό (*n* χαρακτήρες ξεκινώντας από *pos*). Οι εντολές

```
bitset<12> bss0( string("100100101010") ),
           bss1( string("100100101") ),
           bss2( string("11111110010010101001010l0") );
cout << bss0 << endl << bss1 << endl << bss2 << endl;
```

δίνουν:

```
100100101010
000100100101
010100101010
```

Στο *bss0* βάζουμε ως αρχική τιμή ορμαθό (με "`0`" και "`1`") με μήκος 12. Τα ψηφία γίνονται τιμές των δυαδικών ψηφίων της *bss0*.

Στη δεύτερη περίπτωση έχουμε ορμαθό με 9 ψηφία και συμπληρώνεται με τρία "`0`" στην αρχή.

Στην τελευταία περίπτωση ο ορμαθός έχει μήκος μεγαλύτερο από 12. Η παίρνει την τιμή της από τους *τελευταίους* 12 χαρακτήρες.

Ας δούμε τώρα τις αντιστοιχίες μεταξύ του (δικού μας) *Bitmap* και του *bitset*:

- Αντί για τη *bitValue()* έχουμε τον τελεστή "`[]`" που επιφορτώνεται με μεθόδους:

`bitset<N>::reference operator[]( size_t pos );`

`bool operator[]( size_t pos ) const;`

Για τον τύπο "`bitset<N>::reference`" ισχύουν αυτά που είπαμε στην §26.2.1.2 για τον "`vector<bool>::reference`".

Μπορείς ακόμη να πάρεις την τιμή του ψηφίου σε τύπο `bool` με τη μέθοδο

`bool test( size_t pos ) const;`

- Αντί για τη *setBit()* έχουμε τη

`bitset<N>& set( size_t pos, int val=1 );`

Πάντως αυτή μπορεί να χρησιμοποιηθεί και για να μηδενίσουμε ένα ψηφίο αν κληθεί με δεύτερο όρισμα "`0`".

Υπάρχει όμως και μια άλλη μορφή της *set()*

`bitset<N>& set();`

που βάζει τιμή "`1`" σε όλα τα ψηφία του `*this`.

- Αντί για την *clearBit()* υπάρχει η
```
bitset<N>& reset( size_t pos );
```
Είπαμε όμως ότι μπορείς να χρησιμοποιήσεις και τη *set()* με δεύτερο όρισμα "0".
Και για τη *reset()* υπάρχει η μορφή
```
bitset<N>& reset();
```
που μηδενίζει όλα τα ψηφία του *this.
- Αντί για την *count1()* υπάρχει η
```
size_t count() const;
```
- Αντί για τη *display()* χρησιμοποίησε τον τελεστή "<<" όπως κάναμε στα παραδείγματα που δώσαμε παραπάνω.

Για τα τις *part(), firstBitSet()* δεν υπάρχουν αντίστοιχες.
Άλλες μέθοδοι που υπάρχουν στο *bitset*:
- Μέθοδοι αντιστροφής δυαδικών ψηφίων: Η
```
bitset<N>& flip( size_t pos );
```
αντιστρέφει ("0"→"1" και "1"→"0") το ψηφίο στη θέση *pos* και επιστρέφει το *this και η
```
bitset<N>& flip();
```
επιστρέφει το *this αφού αντιστρέψει *όλα* τα ψηφία του. Οι "bs1 = ~bs0" και "bs1 = bs0.flip()" έχουν το ίδιο αποτέλεσμα.
- Μέθοδος μετατροπής σε unsigned long int:
```
unsigned long to_ulong() const;
```
Επιστρέφει την τιμή unsigned long int που αντιστοιχεί στα δυαδικά ψηφία που είναι αποθηκευμένα στο *this. Αν τα ψηφία είναι περισσότερα από αυτά που χρησιμοποιούνται για τον long ρίχνει εξαίρεση (*std*::)*overflow_error*.
- Μέθοδος μετατροπής σε string:
```
string to_string() const;
```
Επιστρέφει την τιμή string που προκύπτει από την αντιστοίχιση κάθε ψηφίου μηδέν στον χαρακτήρα '0' και κάθε ψηφίου ένα στον χαρακτήρα '1'.
- Μέθοδος:
```
bool any() const;
```
που επιστρέφει true αν έστω και ένα ψηφίο του *this έχει τιμή ένα.
- Μέθοδος:
```
bool none() const;
```
που επιστρέφει true αν δεν υπάρχει ψηφίο του *this με τιμή ένα.

Παρατήρηση:▶

Θα πεις τώρα: «Καλα, τα "*bits*" τα είδαμε· το "*set*" που είναι;» Ας δούμε ένα (όχι πολύ καλό) παράδειγμα: Γυρνάμε στο Project 4 και θέλουμε σε κάθε *Course* να βάλουμε και το σύνολο των φοιτητών που έχουν εγγραφεί σε αυτό. Τα στοιχεία των φοιτητών φυλάγονται σε *scNOfStudents* θέσεις του πίνακα scArr. Ένας τρόπος να λύσουμε το πρόβλημά μας είναι ο εξής: Σε κάθε αντικείμενο κλάσης *Course* προσθέτουμε ένα ακομή μέλος:
```
Bitset<allStudents.getNOfStudents()> cEnrolled;
```
Αν ο φοιτητής με τα στοιχεία του στη scArr[k] είναι γραμμένος στο μάθημα *crs* τότε το crs.cEnrolled[k] έχει τιμή "1" αλλιώς "0". Σε μια τέτοια περίπτωση αχρηστεύεται και το *cNoOfStudents* αφού θα μπορούμε να έχουμε:
```
unsigned int Course::getNoOfStudents() const
{  return cEnrolled,count();  }
```

Γιατί δεν είναι πολύ καλό το παράδειγμα; Διότι ο πίνακας φοιτητών δεν είναι και τοσο σταθερός: φοιτητές εισάγονται και διαγράφονται και οι θέσεις των στοιχείων στον πίνακα μπορεί να μεταβάλλονται.◀

26.5.2 Το Περίγραμμα "*complex*"

Βάζοντας στο πρόγραμμά σου "`#include <complex>`" μπορείς να χρησιμοποιήσεις το περίγραμμα

```
template< class T >
class complex
{
public:
   typedef T value_type;
   complex( const T& re=T(), const T& im=T() );
   complex( const complex& );
   template<class X>  complex( const complex<X>& );
   T real() const;
   T imag() const;
   complex<T>& operator=( const T& );
   complex<T>& operator+=( const T& );
   complex<T>& operator-=( const T& );
   complex<T>& operator*=( const T& );
   complex<T>& operator/=( const T& );
   complex& operator=( const complex& );
   template<class X> complex<T>& operator=( const complex<X>& );
   template<class X> complex<T>& operator+=( const complex<X>& );
   template<class X> complex<T>& operator-=( const complex<X>& );
   template<class X> complex<T>& operator*=( const complex<X>& );
   template<class X> complex<T>& operator/=( const complex<X>& );
}; // complex
```

για μιγαδικούς αριθμούς.

Στο `complex` υπάρχουν και τρεις εξειδικεύσεις του περιγράμματος για `float`, `double` και `long double`.

Ακόμη, με περιγράμματα καθολικών συναρτήσεων επιφορτώνονται:

- Οι τελεστές "+", "-", "*", "/".
- Οι συναρτήσεις *abs()* και *arg()* με τα

```
template< class T > T abs( const complex<T>& x )
{  return sqrt( x.real()*x.real()+x.imag()*x.imag() );  }
template< class T > T arg( const complex<T>& x )
{  return atan2( imag(x), real(x) )  }
```

που μας δίνουν τα συστατικά της πολικής μορφής.

- Η συνάρτηση (περίγραμμα)

```
template< class T >
complex<T> polar( const T& rho, const T& theta = 0 );
```

που μας δίνει έναν μιγαδικό από τα πολικά συστατικά του. Θα πρέπει να έχουμε:

$$x == polar(abs(x), arg(x))$$

αλλά, αφού έχουμε να κάνουμε με τιμές κινητής υποδιαστολής, αντί για το "==" καλύτερα να σκέφεται "≈".

- Η συνάρτηση

```
template< class T > complex<T> conj( const complex<T>& x );
```

που μας δίνει τον συζυγή του *x*.

- Περιγράμματα συναρτήσεων για *sqrt()*, *pow()*, *cos()*, *sin()*, *tan()*, *exp()*, *log()*, *log10()*, *cosh()*, *sinh()*.

26.6 Τι (Πρέπει να) Έμαθες στο Κεφάλαιο Αυτό

Όταν λέμε STL εννοούμε *περιέχοντα*, *προσεγγιστές*, *συναρτησιακά αντικείμενα* και *αλγόριθμους* (και όχι απλώς το *vector* όπως πιστεύουν μερικοί).

Τα *περιέχοντα* της STL είναι *vector*, *list*, *deque* (ακολουθίες) και *set*, *map*, *multiset* και *multimap* (συνειρμικά). Το ότι μέχρι τώρα παριστάναμε όλες τις συλλογές αντικειμένων με πίνακες (απλούς ή δυναμικούς) δεν σημαίνει ότι από εδώ και πέρα θα τις παριστάνουμε με αντικείμενα στιγμιοτύπων του *vector*. Το περιέχον που επιλέγουμε εξαρτάται από το πρόβλημα που έχουμε να λύσουμε.

Οι *προσεγγιστές* είναι πολύτιμα εργαλεία για την αξιοποίηση των περιεχόντων αλλά και των αλγορίθμων. Με βάση τις δυνατότητές τους κατηγοριοποιούνται σε *προσεγγιστές εισόδου*, *εξόδου*, *πρόσθιους*, *αμφίδρομους* και *τυχαίας πρόσβασης*.

Τα *συναρτησιακά αντικείμενα* είναι απαραίτητα για την παραμετροποίηση των περιγραμμάτων κλάσεων και συναρτήσεων. Δεν χρειάζεται να γράφουμε τα πάντα: η STL μας δίνει μια μικρή αλλά πολύ χρήσιμη συλλογή.

Για να χρησιμοποιήσουμε τα περιέχοντα της STL, εκτός από τις μεθόδους τους, μας δίνονται και αρκετοί *αλγόριθμοι* (περιγράμματα συναρτήσεων). Πολλοί από αυτούς δρουν σε συλλογές αντικειμένων. Σε τέτοιες περιπτώσεις στις παραμέτρους της συνάρτησης περιλαμβάνονται προσεγγιστές *first*, *last* και υπονοείται επεξεργασία των αντικειμένων **it* όταν ο *it* διατρέχει την περιοχή [*first*, *last*).

Στις βιβλιοθήκες της C++ υπάρχουν και άλλα περιγράμματα κλάσεων εκτός από τα περιέχοντα. Εν συντομία είδαμε δύο τέτοια παραδείγματα: *bitset* και *complex*.

Ασκήσεις

26-1 Αν

```
vector < T > v1;
```

τι αποτέλεσμα έχουν οι εντολές:

```
set< T > s1( v1.begin(), v1.end() );
vector< T > v2( s1.begin(), s1.end() );
```

Πώς αλλιώς (χωρίς να χρησιμοποιήσεις σύνολο) θα μπορούσες να βάλεις στο *v1* το τελικό περιεχόμενο του *v2*;

26-2 Δύο συναρτήσεις χρήσιμες για πολυσύνολα είναι οι εξής:

- *eqmult()*: η *eqmult*(v, M_1, M_2) μας δίνει τιμή `true` αν δύο πολυσύνολα M_1, M_2 έχουν ίσα πλήθη του ατόμου v.
- **κλιμάκωση** (scaling): η $n \otimes M_1$ (n: φυσικός) μας δίνει πολυσύνολο που περιέχει τα στοιχεία του M_1 και μόνον αυτά αλλά σε n-πλάσιο πλήθος από ότι τα έχει το M_1.

Να τις υλοποιήσεις με τα κατάλληλα περιγράμματα συναρτήσεων που θα είναι δυνατόν να χρησιμοποιηθούν για στιγμιότυπα των *multiset*, *multimap*.