ΕΝΑΤΗ ΔΙΑΛΕΞΗ

Εισαγωγή στη Standard Library

- Οτιδήποτε παρέχεται στη C++ μέσω header αποτελεί τη Standard Library (SL).
- Η SL παρέχει τις δυνατότητες που βρίσκουμε στις περισσότερες γλώσσες προγραμματισμού (αρχεία, μαθηματικές συναρτήσεις κλπ.).
- Επιπλέον, η SL επεκτείνει τη βασική γλώσσα παρέχοντας
  - δομές αποθήκευσης στοιχείων,
  - δομές χειρισμού αυτών και
  - συναρτήσεις που υλοποιούν βασικές "πράξεις".

Στα επόμενα θα επικεντρωθούμε στο τελευταίο χαρακτηριστικό.

Βοηθητικές συναρτήσεις (1/2)

Συναρτήσεις ελάχιστου/μέγιστου

Στο header `<algorithm>` ορίζονται οι συναρτήσεις που διακρίνουν

- το ελάχιστο, `std::min()`,
- το μέγιστο, `std::max()`, και
- το ελάχιστο και το μέγιστο ταυτόχρονα, `std::minmax()`,

κάποιων αντικειμένων.
Στην απλή τους μορφή οι δύο πρώτες δέχονται δύο ποσότητες ή μια λίστα ποσοτήτων *ίδιου τύπου* και επιστρέφουν την (πρώτη) μικρότερη ή μεγαλύτερη αντίστοιχα, αφού τις συγκρίνουν με τον τελεστή '<'.
Η συνάρτηση `std::minmax()` δέχεται δύο ορίσματα ή μια λίστα ορισμάτων και επιστρέφει ένα **struct** με δύο μέλη: το `first` είναι το (πρώτο) ελάχιστο και το `second` το (τελευταίο) μέγιστο από αυτά.

Παράδειγμα

```
auto a = std::min(3.1,5.5);       // a = 3.1
auto b = std::max({12,3,5,17,9});// b = 17
auto c = std::minmax({3,2,9,-1});// c.first = -1, c.second = 9
```

Βοηθητικές συναρτήσεις (2/2)

Συνάρτησεις εναλλαγής/ανταλλαγής

Στο header `<utility>` ορίζονται οι συναρτήσεις

- *εναλλαγής*, `std::swap()`. Δέχεται δύο ποσότητες ίδιου τύπου και εναλλάσσει τις τιμές τους.

Παράδειγμα

```
double a{3.1}, b{4.2};
std::swap(a,b);  // a = 4.2, b = 3.1
```

- *ανταλλαγής*, `std::exchange()`. Δέχεται ως πρώτο όρισμα ένα αντικείμενο και ως δεύτερο μια τιμή που μπορεί να εκχωρηθεί (με πιθανή μετατροπή τύπου) στο αντικείμενο. Εκχωρεί την τιμή στο αντικείμενο και επιστρέφει την παλιά τιμή του.

Παράδειγμα

```
int b{3}, c{5};
auto a = std::exchange(b,c); //(Σχεδόν) ισοδύναμο με a=b; b=c;
```

Συνιστώσες της Standard Library

Η Standard Library έχει τρεις κύριες συνιστώσες:

- τους containers, δομές με κατάλληλα χαρακτηριστικά για την αποθήκευση και διαχείριση δεδομένων οποιουδήποτε τύπου, η κάθε μία με διαφορετικές ιδιότητες και δυνατότητες.
- τους iterators, ένα είδος δείκτη σε θέσεις μιας συλλογής στοιχείων (π.χ. ενός container, ενός string ή ενός αρχείου). Οι iterators έχουν την ίδια μορφή για όλες τις ακολουθίες στοιχείων με αποτέλεσμα να παρέχουν συγκεκριμένο, ενιαίο τρόπο για τη διαχείρισή τους.
- τους αλγόριθμους, συναρτήσεις που υλοποιούν με πολύ αποτελεσματικό τρόπο τμήματα κώδικα που χρειάζονται συχνά (π.χ. ταξινόμηση στοιχείων μιας ακολουθίας, αναζήτηση ή αντικατάσταση στοιχείου με συγκεκριμένη τιμή σε αυτή κλπ.). Είναι ανεξάρτητοι από τη δομή αποθήκευσης καθώς δρουν σε iterators.

Εισαγωγή στους containers (1/2)

- Οι containers χρησιμοποιούνται για την αποθήκευση και διαχείριση συλλογών από αντικείμενα οποιουδήποτε (ενιαίου) τύπου.
- Κάθε container έχει πλεονεκτήματα και μειονεκτήματα όταν συγκρίνεται με τους υπόλοιπους. Ο κατάλληλος container υπερέχει όμως σε σύγκριση με τη μοναδική αντίστοιχη δομή που κληρονομήθηκε από τη C, το ενσωματωμένο στατικό ή δυναμικό διάνυσμα.
- Είναι εσωτερικά πιο πολύπλοκοι από το ενσωματωμένο διάνυσμα καθώς έχουν τη δυνατότητα για π.χ. δυναμική διαχείριση μνήμης, γρήγορη εισαγωγή/διαγραφή στοιχείων, ταχύτατη αναζήτηση, κλπ.

Εισαγωγή στους containers (2/2)

Οι containers διακρίνονται σε τρεις γενικές κατηγορίες:

- sequence containers (σειριακοί).
- associative containers (συσχέτισης).
- unordered associative containers (συσχέτισης χωρίς τάξη).

Όλοι οι containers είναι templates. Κατά τη δήλωση προσδιορίζουμε τον τύπο των στοιχείων που μπορεί να αποθηκεύσει, το κριτήριο ταξινόμησης, κ.α.

Κατηγορίες containers (1/3)

Sequence containers

- Είναι συλλογές ομοειδών στοιχείων στις οποίες κάθε ένα έχει συγκεκριμένη θέση ως προς τα υπόλοιπα. Η θέση προσδιορίζεται από τον προγραμματιστή κατά την εισαγωγή κάθε στοιχείου και είναι ανεξάρτητη από την τιμή του.
- Παρέχουν γρήγορη πρόσβαση σε οποιοδήποτε στοιχείο ή γρήγορη εισαγωγή/διαγραφή στοιχείου (ανάλογα με τη θέση).
- Η κατηγορία περιλαμβάνει τους
  - `array<>`,
  - `vector<>`,
  - `deque<>`,
  - `forward_list<>`,
  - `list<>`.

Κατηγορίες containers (2/3)

Associative containers

- Είναι ταξινομημένες συλλογές στις οποίες η θέση κάθε στοιχείου εξαρτάται μόνο από την τιμή του και την τιμή των υπόλοιπων στοιχείων, και προσδιορίζεται σύμφωνα με κάποιο κριτήριο ταξινόμησης.
- Παρέχουν τη δυνατότητα ταχύτατης *αναζήτησης* στοιχείου.
- Η κατηγορία περιλαμβάνει τους
  - `set<>`,
  - `multiset<>`,
  - `map<>`,
  - `multimap<>`.

Κατηγορίες containers (3/3)

Unordered associative containers

- Είναι σύνολα στοιχείων χωρίς καθορισμένη σειρά. Η θέση κάθε στοιχείου σε τέτοια συλλογή είναι μεν συγκεκριμένη αλλά μπορεί να αλλάξει κατά την εκτέλεση του προγράμματος (π.χ. μετά από εισαγωγή ή διαγραφή στοιχείων).
- Παρέχουν ταχύτατο *έλεγχο ύπαρξης* στοιχείου.
- Η κατηγορία περιλαμβάνει τους
  - `unordered_set<>`,
  - `unordered_multiset<>`,
  - `unordered_map<>`,
  - `unordered_multimap<>`.

Σύνταξη

`std::array<`*τύπος*, *πλήθος*`>`

Χαρακτηριστικά

- Είναι sequence container. Η σειρά των στοιχείων προσδιορίζεται κατά την εισαγωγή τους.
- Παρέχεται από το header `<array>`.

Ιδιότητες

Ένα αντικείμενο τύπου `std::array<>`

- Δημιουργείται *κατά τη μεταγλώττιση* με γνωστό πλήθος στοιχείων, που δεν μπορεί να αλλάξει κατά την εκτέλεση. Δεν επιτρέπεται εισαγωγή/διαγραφή θέσεων.
- Αποθηκεύει τα στοιχεία σε *συνεχόμενες θέσεις μνήμης*.
- Παρέχει *ισόχρονη* και γρήγορη πρόσβαση σε οποιοδήποτε στοιχείο.

vector<>

Σύνταξη

`std::vector<τύπος>`

Χαρακτηριστικά

- Είναι sequence container. Η σειρά των στοιχείων προσδιορίζεται κατά την εισαγωγή τους.
- Κάνει διαχείριση μνήμης χωρίς παρέμβαση του προγραμματιστή.
- Παρέχεται από το header `<vector>`.

Ιδιότητες

Ένα αντικείμενο τύπου `std::vector<>`

- Δημιουργείται κατά την εκτέλεση του προγράμματος.
- Αποθηκεύει τα στοιχεία σε *συνεχόμενες θέσεις μνήμης*.
- Επιτρέπει την εισαγωγή ή διαγραφή θέσεων. Είναι ταχύτατες όταν γίνονται *στο τέλος* του αντικειμένου.
- Παρέχει *ισόχρονη* και γρήγορη πρόσβαση σε οποιοδήποτε στοιχείο.

deque<>

Σύνταξη

`std::deque<`*τύπος*`>`

Χαρακτηριστικά

- Είναι sequence container. Η σειρά των στοιχείων προσδιορίζεται κατά την εισαγωγή τους.
- Κάνει διαχείριση μνήμης χωρίς παρέμβαση του προγραμματιστή.
- Παρέχεται από το header `<deque>`.

Ιδιότητες

Ένα αντικείμενο τύπου `std::deque<>`

- Δημιουργείται κατά την εκτέλεση του προγράμματος.
- Αποθηκεύει τα στοιχεία σε *διάσπαρτα τμήματα μνήμης*.
- Επιτρέπει την εισαγωγή ή διαγραφή θέσεων. Είναι ταχύτατες όταν γίνονται *στα άκρα* (αρχή ή τέλος) του αντικειμένου.
- Παρέχει *ισόχρονη* πρόσβαση σε οποιοδήποτε στοιχείο (αλλά πιο αργά απ' ό,τι ένα `vector<>`).

list<>

Σύνταξη

std::list<τύπος>

Χαρακτηριστικά

- Είναι sequence container. Η σειρά των στοιχείων προσδιορίζεται κατά την εισαγωγή τους.
- Κάνει διαχείριση μνήμης χωρίς παρέμβαση του προγραμματιστή.
- Παρέχεται από το header <list>.

Ιδιότητες

Ένα αντικείμενο τύπου std::list<>

- Δημιουργείται κατά την εκτέλεση του προγράμματος.
- Αποθηκεύει τα στοιχεία σε *διάσπαρτες θέσεις μνήμης*.
- Επιτρέπει την εισαγωγή ή διαγραφή θέσεων στον *ίδιο χρόνο* σε *οποιοδήποτε σημείο* του αντικειμένου.
- Παρέχει πρόσβαση σε κάποιο στοιχείο, σε χρόνο ανάλογο του πλήθους θέσεων που μεσολαβούν από την τρέχουσα θέση.

forward_list<>

Σύνταξη

`std::forward_list<τύπος>`

Χαρακτηριστικά

- Είναι sequence container. Η σειρά των στοιχείων προσδιορίζεται κατά την εισαγωγή τους.
- Κάνει διαχείριση μνήμης χωρίς παρέμβαση του προγραμματιστή.
- Παρέχεται από το header `<forward_list>`.

Ιδιότητες

Ένα αντικείμενο τύπου `std::forward_list<>`

- Δημιουργείται κατά την εκτέλεση του προγράμματος.
- Αποθηκεύει τα στοιχεία σε *διάσπαρτες θέσεις μνήμης*.
- Επιτρέπει την εισαγωγή ή διαγραφή θέσεων στον *ίδιο χρόνο* σε *επόμενη θέση* από την τρέχουσα (αλλά όχι στο τέλος).
- Παρέχει πρόσβαση σε κάποιο *επόμενο* στοιχείο, σε χρόνο *ανάλογο του πλήθους θέσεων που μεσολαβούν από την τρέχουσα θέση*.

set<> και multiset<> (1/2)

Σύνταξη

std::set<τύπος,κριτήριο>
std::multiset<τύπος,κριτήριο>

Χαρακτηριστικά

- Είναι associative containers. Τα στοιχεία *ταξινομούνται* κατά την εισαγωγή τους με βάση το κριτήριο ταξινόμησης.
- Αν δεν προσδιορίσουμε κριτήριο ταξινόμησης, εφαρμόζεται το προκαθορισμένο. Σύμφωνα με αυτό, το μικρότερο στοιχείο προηγείται του μεγαλύτερου.
- Παρέχουν ταχύτατη αναζήτηση (π.χ. με δυαδικό αλγόριθμο).
- Κάνουν διαχείριση μνήμης χωρίς παρέμβαση του προγραμματιστή.
- Παρέχονται από το header <set>.

set<> και multiset<> (2/2)

Ιδιότητες

Ένα αντικείμενο τύπου `std::set<>` ή `std::multiset<>`

- Δημιουργείται κατά την εκτέλεση του προγράμματος.
- Τα στοιχεία είναι πάντα *ταξινομημένα* με βάση το κριτήριο ταξινόμησης.
- Δεν επιτρέπει μεταβολή της τιμής των στοιχείων παρά μόνο διαγραφή και νέα εισαγωγή.

Η εισαγωγή στοιχείου που υπάρχει ήδη,

- αγνοείται σε ένα `std::set<>` (συνεπώς τα στοιχεία ενός `std::set<>` είναι μοναδικά).
- δημιουργεί νέο στοιχείο με ίδια τιμή σε ένα `std::multiset<>`.

map<> και multimap<> (1/2)

Σύνταξη

std::map<τύποςK,τύποςV,κριτήριο>
std::multimap<τύποςK,τύποςV,κριτήριο>

Χαρακτηριστικά

- Είναι associative containers. Αποθηκεύουν *ζεύγη στοιχείων* (key,value) ταξινομημένα ως προς το key με βάση το κριτήριο ταξινόμησης.
- Αν δεν προσδιορίσουμε κριτήριο ταξινόμησης, το ζεύγος με το μικρότερο key προηγείται του ζεύγους με το μεγαλύτερο.
- Παρέχουν ταχύτατη αναζήτηση (π.χ. με δυαδικό αλγόριθμο).
- Κάνουν διαχείριση μνήμης χωρίς παρέμβαση του προγραμματιστή.
- Παρέχονται από το header <map>.

map<> και multimap<> (2/2)

Ιδιότητες

Ένα αντικείμενο τύπου `std::map<>` ή `std::multimap<>`

- Δημιουργείται κατά την εκτέλεση του προγράμματος.
- Τα στοιχεία είναι ζεύγη *ταξινομημένα* με βάση το κριτήριο ταξινόμησης.
- Δεν επιτρέπει μεταβολή της τιμής *του key* των στοιχείων παρά μόνο διαγραφή και νέα εισαγωγή. Επιτρέπεται η μεταβολή του value ενός ζεύγους.

Η εισαγωγή ζεύγους με key που υπάρχει ήδη,

- αγνοείται σε ένα `std::map<>`.
- δημιουργεί νέο ζεύγος με ίδια τιμή key σε ένα `std::multimap<>`.

unordered_set<>, unordered_multiset<>, unordered_map<>, unordered_multimap<>

- Είναι unordered associative containers, παρόμοιοι με τα set<>, multiset<>, map<>, multimap<>.
- Ως παραμέτρους των templates προσδιορίζουμε
  - τον τύπο των στοιχείων ή τους τύπους που αποτελούν τα ζεύγη που αποθηκεύουν,
  - προαιρετικά τη συνάρτηση κατακερματισμού (hash),
  - προαιρετικά το κριτήριο ισότητας.
- Παρέχονται από τους headers <unordered_set> (οι δύο πρώτοι) και <unordered_map> (οι δύο επόμενοι).
- Αποθηκεύουν τα στοιχεία οργανώνοντάς τα με τέτοιο τρόπο ώστε να παρέχουν ταχύτατο *έλεγχο ύπαρξης* κάποιας τιμής ή κλειδιού.

Εισαγωγή στους iterators

Οι iterators

- είναι δείκτες σε θέσεις κάποιας ακολουθίας στοιχείων ίδιου τύπου, καθώς και σε *μία θέση μετά το τέλος* της ακολουθίας,
- αποτελούν το μηχανισμό για τον ενιαίο και ομοιόμορφο χειρισμό οποιουδήποτε container, ροής αντικειμένων ίδιου τύπου (αρχείο), `std::string` ή και ενσωματωμένου διανύσματος,
- ορίζονται και δρουν μόνο σε τέτοιες ακολουθίες,
- παράγονται από συναρτήσεις–μέλη των containers, αλγόριθμους της Standard Library ή άλλους iterators,
- διαφέρουν ως έννοια από τους δείκτες (pointers) που κληρονομήθηκαν από τη C, παρόλο που συμπεριφέρονται παρόμοια.

Iterators: ορισμός (1/2)

Μια μεταβλητή με όνομα, π.χ., it, μπορεί να οριστεί ως iterator κατάλληλος για τις θέσεις ενός container με τύπο `cntr<T>` (και μόνο) ως εξής:

```
cntr<T>::iterator it;
```

Κάθε container έχει ως μέλη δύο συναρτήσεις που επιστρέφουν συγκεκριμένους iterators. Σε ένα container με όνομα c:

- η `c.begin()` επιστρέφει iterator που δείχνει στην πρώτη θέση αποθήκευσης του c.
- η `c.end()` επιστρέφει iterator στην *επόμενη θέση μετά την τελευταία* θέση αποθήκευσης του c.

Iterators: ορισμός (2/2)

Παράδειγμα

Οι δηλώσεις με εκχώρηση τιμής, δύο iterators με ονόματα beg, end, στην αρχή και *σε μία θέση μετά το τέλος* ενός container τύπου `std::vector<double>`, με όνομα v, είναι

```
std::vector<double>::iterator beg{v.begin()};
std::vector<double>::iterator end{v.end()};
```

ή, πιο απλά,

```
auto beg = v.begin();
auto end = v.end();
```

Ισοδύναμα μπορούμε να γράψουμε

```
auto beg = std::begin(v);
auto end = std::end(v);
```

Iterator σε σταθερή ποσότητα

- Iterator κατάλληλος για τις θέσεις ενός container με τύπο `cntr<T>`, μέσω του οποίου *δεν μπορούμε να μεταβάλουμε τις τιμές* των θέσεων, ορίζεται ως εξής:

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

- Κάθε container παρέχει τις συναρτήσεις–μέλη `cbegin()` και `cend()` που επιστρέφουν `const_iterators` στην αρχή και σε μία θέση μετά το τέλος του αντικειμένου για το οποίο καλούνται.
- Επιπλέον, παρέχονται οι συναρτήσεις `std::cbegin()` και `std::cend()` που επιστρέφουν τους αντίστοιχους `const_iterators` για το όρισμά τους.

Iterators: πράξεις (1/2)

Σε ένα iterator

- Η δράση από αριστερά του τελεστή '`*`' δίνει πρόσβαση στο στοιχείο στο οποίο δείχνει ο iterator, π.χ. `*it=3;`.
  Προσέξτε ότι σε iterator όπως ο `end()`, που δεν δείχνει σε θέση ενός container, η δράση του τελεστή '`*`' είναι λάθος.
- Η δράση του τελεστή '`->`' δίνει πρόσβαση σε μέλος του στοιχείου στο οποίο δείχνει ένας iterator. Επομένως, αν το στοιχείο στη θέση που δείχνει ο iterator it έχει μέλος με όνομα member, η έκφραση `(*it).member` ισοδυναμεί με `it->member`.
- Η δράση του τελεστή ++ *μετακινεί* τον iterator στην επόμενη θέση.

Iterators: πράξεις (2/2)

Σε iterators ειδικών κατηγοριών

- Η δράση του τελεστή `--` *μετακινεί* τον iterator στην προηγούμενη θέση.
- Η πρόσθεση ή αφαίρεση ενός ακέραιου αριθμού n έχει αποτέλεσμα ένα άλλο iterator που δείχνει n θέσεις μετά ή πριν.
  Επιπλέον, έχουν το αναμενόμενο νόημα οι τελεστές '+=' και '−='.
- Έχουν νόημα οι τελεστές σύγκρισης ==, != μεταξύ iterators στον ίδιο container. Ελέγχουν αν οι iterators δείχνουν ή όχι στην ίδια θέση.
  Σε iterators κατάλληλου τύπου έχουν νόημα και οι υπόλοιποι τελεστές σύγκρισης.

Iterators: Παράδειγμα Α'

Εκχώρηση τιμής (π.χ. 3.5) σε όλα τα στοιχεία ενός container που περιέχει πραγματικούς αριθμούς και έχει όνομα v:

```
for (auto it = v.begin(); it != v.end(); ++it) {
   *it = 3.5;
}
```

Προσέξτε πώς γράφουμε τη συνθήκη για τη συνέχιση της επανάληψης: ο iterator, που μετακινείται προς το τέλος σε κάθε επανάληψη, συγκρίνεται τον iterator της πρώτης θέσης μετά το τέλος του container. Όταν φτάσει εκεί, η επανάληψη διακόπτεται καθώς έχουμε διατρέξει όλο τον container.
Τελείως ισοδύναμος κώδικας με τον παραπάνω, γράφεται με τη χρήση του range **for**:

```
for (auto & x : v) {
   x = 3.5;
}
```

Iterators: Παράδειγμα Β'

Εκτύπωση στην οθόνη των στοιχείων container τύπου `std::vector<double>` με όνομα v:

```
for (auto it = v.cbegin(); it != v.cend(); ++it) {
   std::cout << *it << '\n';
}
```

Παρατηρήστε την επιλογή των συναρτήσεων–μελών για τον προσδιορισμό των iterators αρχής και τέλους. Σε συνδυασμό με την αυτόματη δήλωση, ο it είναι τύπου `std::vector<double>::const_iterator`. Δεν χρειαζόμαστε (και, για ασφάλεια, με τη συγκεκριμένη δήλωση δεν επιτρέπουμε) την τροποποίηση των στοιχείων που δείχνει ο συγκεκριμένος iterator.