ΕΝΔΕΚΑΤΗ ΔΙΑΛΕΞΗ

Αλγόριθμοι: εισαγωγή

- Η Standard Library παρέχει δεκάδες συναρτήσεις που εκτελούν συνήθεις ή θεμελιώδεις πράξεις σε ακολουθίες στοιχείων: αντιγραφή ή τροποποίηση, αναζήτηση στοιχείων με συγκεκριμένη ιδιότητα, ταξινόμηση, αναδιάταξη στοιχείων, κλπ. Οι συναρτήσεις λέγονται *αλγόριθμοι*.
- Οι αλγόριθμοι είναι ανεξάρτητοι από τους containers. Δρουν σε συλλογές στοιχείων που υποδεικνύονται από iterators. Κάθε αλγόριθμος χρειάζεται συγκεκριμένο τύπο iterators (και άνω, στην ιεραρχία).
- Οι iterators ως ορίσματα συχνά εμφανίζονται σε ζεύγη: ο πρώτος προσδιορίζει την αρχή κάποιου διαστήματος. Το τέλος του διαστήματος είναι *μία θέση πριν* τη θέση στην οποία δείχνει ο δεύτερος.

Η πλειονότητα των αλγορίθμων παρέχονται από το header `<algorithm>`.
Κάποιοι περιλαμβάνονται στο `<numeric>`.
Όλοι ορίζονται στο χώρο ονομάτων `std`.

Συνάρτηση λάμδα

Οι *συναρτήσεις λάμδα* είναι μικρές, ανώνυμες συναρτήσεις που μπορούν να δηλωθούν σχεδόν οπουδήποτε. Συνήθως δηλώνονται ως όρισμα συνήθους συνάρτησης.
Έχουν τη δυνατότητα πρόσβασης σε μεταβλητές του περιβάλλοντός τους (της συνάρτησης στην οποία ορίζονται), με διαφορετικό μηχανισμό από τα ορίσματα.

Παραδείγματα

- Μια συνάρτηση λάμδα που δέχεται δύο ακέραια ορίσματα και επιστρέφει αυτό που έχει τη μικρότερη απόλυτη τιμή, είναι η

```
[](int x, int y) -> int
{ return (std::abs(x) < std::abs(y) ? x : y); }
```

- Μια συνάρτηση λάμδα, που ορίζεται μέσα σε κάποια συνήθη συνάρτηση, δέχεται ένα ακέραιο όρισμα και επιστρέφει το άθροισμα αυτού και μιας ακέραιας ποσότητας a της περικλείουσας συνάρτησης, είναι η

```
[a](int x) { return x+a; }
```

Παρατηρήστε ότι είναι συνάρτηση ενός ορίσματος. Ο τύπος της ποσότητας που επιστρέφεται μπορεί να παραληφθεί, καθώς προσδιορίζεται αυτόματα από την τιμή που επιστρέφεται με το **return**.

Αλγόριθμος `std::accumulate()`, Α' μορφή

```
template<typename InputIterator, typename Type>
Type
accumulate(InputIterator beg, InputIterator end, Type value);
```

- Ο αλγόριθμος επιστρέφει το άθροισμα της τιμής value και των στοιχείων του διαστήματος [beg,end).
- Η άθροιση γίνεται με τον τελεστή '+' (με όποιο νόημα έχει για τα στοιχεία στο διάστημα [beg,end)).

Ο αλγόριθμος παρέχεται από το `<numeric>`.

Παράδειγμα

Ο κώδικας

```
auto sum  = std::accumulate(v.cbegin(), v.cend(), 0.0);
```

υπολογίζει το άθροισμα των πραγματικών στοιχείων ενός container με όνομα v.

Αλγόριθμος std::accumulate(), Β' μορφή

```
template<typename InputIterator, typename Type,
         typename BinaryFunctor>
Type
accumulate(InputIterator beg, InputIterator end, Type value,
           BinaryFunctor op);
```

- Στη δεύτερη μορφή ο αλγόριθμος δέχεται ως επιπλέον τελευταίο όρισμα μια συνάρτηση op() που παίρνει δύο ορίσματα τύπου Type και επιστρέφει τιμή τέτοιου τύπου.
- Η συνάρτηση προσδιορίζει την εκτελούμενη πράξη μεταξύ των στοιχείων. Αν {a1, a2, a3, ...} είναι τα στοιχεία του διαστήματος, ο αλγόριθμος υπολογίζει διαδοχικά τα

  ```
  value = op(value,a1);
  value = op(value,a2);
  ...
  ```

  και επιστρέφει το τελικό value.

Ο αλγόριθμος παρέχεται από το <numeric>.

Αλγόριθμος `std::accumulate()`: Παράδειγμα

Το γινόμενο των τιμών μιας ακολουθίας ακέραιων στο διάστημα [beg,end) μπορεί να υπολογιστεί ως εξής:

```
#include <numeric>

/*
template<typename InputIterator, typename Type,
         typename BinaryFunctor>
Type
accumulate(InputIterator beg, InputIterator end, Type value,
           BinaryFunctor op);
*/

int p = std::accumulate(beg, end, 1,
                        [](int x, int y) {return x*y;});
```

Αλγόριθμος `std::min_element()`

```
template<typename ForwardIterator>
ForwardIterator
min_element(ForwardIterator beg, ForwardIterator end);

template<typename ForwardIterator, typename BinaryFunctor>
ForwardIterator
min_element(ForwardIterator beg, ForwardIterator end,
            BinaryFunctor comp);
```

- Η συνάρτηση επιστρέφει iterator στη θέση του μικρότερου στοιχείου (ή του πρώτου από όλα τα μικρότερα) στο διάστημα [beg,end). Αν το διάστημα είναι κενό (δηλαδή, αν `beg==end`), επιστρέφει το beg.
- Η σύγκριση των στοιχείων στην πρώτη μορφή γίνεται με τον τελεστή '<'.
- Στη δεύτερη, η σύγκριση γίνεται με βάση τη συνάρτηση `comp()`, η οποία δέχεται δύο ορίσματα και επιστρέφει **`true`** ή **`false`** αν το πρώτο όρισμά της είναι «μικρότερο» (ό,τι κι αν σημαίνει αυτό) από το δεύτερο ή όχι.

Παρέχεται από το `<algorithm>`.

Αλγόριθμος `std::copy()`

```
template<typename InputIterator, typename OutputIterator>
OutputIterator
copy(InputIterator beg1, InputIterator end1,
     OutputIterator beg2);
```

- Ο αλγόριθμος *αντιγράφει* τα στοιχεία του διαστήματος [beg1, end1) στο διάστημα που ξεκινά με το beg2. Αν το beg1 δείχνει στην ίδια θέση με το end1 ή σε επόμενη θέση, ο αλγόριθμος δεν κάνει τίποτε.
- Ο iterator beg2 μπορεί να δείχνει στον ίδιο container με το beg1 αλλά δεν επιτρέπεται να ανήκει στο διάστημα [beg1, end1).
- Η συνάρτηση επιστρέφει iterator στην ακολουθία εξόδου, στην επόμενη θέση από την τελευταία στην οποία έγινε εγγραφή.

Παρέχεται από το `<algorithm>`.

Αλγόριθμος `std::reverse()`

```
template<typename BidirectionalIterator>
void
reverse(BidirectionalIterator beg, BidirectionalIterator end);
```

- Ο αλγόριθμος αναστρέφει τη σειρά των στοιχείων του διαστήματος [beg,end).
- Οι `std::list<>` και `std::forward_list<>` παρέχουν γρηγορότερη συνάρτηση–μέλος.

Παρέχεται από το `<algorithm>`.

Αλγόριθμος `std::find()`

```
template<typename InputIterator, typename Type>
InputIterator
find(InputIterator beg, InputIterator end, Type const & value);
```

- Η συνάρτηση επιστρέφει iterator στη θέση του πρώτου στοιχείου στο [beg,end) που είναι ίσο με value. Η σύγκριση γίνεται με τον τελεστή '=='. Οι associative και unordered containers παρέχουν ανάλογη συνάρτηση–μέλος που είναι πιο γρήγορη.
- Η συνάρτηση επιστρέφει το end αν δεν υπάρχει στοιχείο που να ικανοποιεί την αντίστοιχη συνθήκη.
- Αν τα στοιχεία στο διάστημα [beg,end) είναι ταξινομημένα, υπάρχουν πιο γρήγοροι αλγόριθμοι για την εύρεση συγκεκριμένου στοιχείου.

Παρέχεται από το `<algorithm>`.

Αλγόριθμος `std::sort()`

```
template<typename RandomIterator>
void sort(RandomIterator beg, RandomIterator end);

template<typename RandomIterator, typename BinaryFunctor>
void
sort(RandomIterator beg, RandomIterator end, BinaryFunctor comp);
```

- Ο αλγόριθμος *ταξινομεί* τα στοιχεία στο διάστημα [beg,end) κάνοντας συγκρίσεις με πλήθος ανάλογο του $n \log n$, όπου n το πλήθος των στοιχείων.
- Η σύγκριση των στοιχείων στην πρώτη μορφή γίνεται με τον τελεστή '<'.
- Στη δεύτερη, η σύγκριση γίνεται με βάση τη συνάρτηση `comp()`, η οποία δέχεται δύο ορίσματα και επιστρέφει **`true`** ή **`false`** αν το πρώτο όρισμά της είναι «μικρότερο» (ό,τι κι αν σημαίνει αυτό) από το δεύτερο ή όχι.
- Η σχετική θέση ισοδύναμων στοιχείων δεν διατηρείται απαραίτητα.

Παρέχεται από το `<algorithm>`.

Εισαγωγή στον Αντικειμενοστρεφή Προγραμματισμό (1/3)

Έστω ότι θέλουμε να γράψουμε κώδικα που να χειρίζεται ημερομηνίες.

Πρώτη (απλοϊκή) προσέγγιση

Για μια ημερομηνία ορίζουμε τρεις ακέραιες μεταβλητές: d, m, y (για ημέρα, μήνα, έτος). Π.χ.

```
int d{3}, m{12}, y{2001};
```

Αν έχουμε πολλές ημερομηνίες, ορίζουμε *τρία* `std::vector<int>`.

Μειονεκτήματα

- Τα d,m,y είναι ασύνδετα (ενώ θα έπρεπε να σχετίζονται) και ισοδύναμα με οποιοδήποτε άλλο ακέραιο του προγράμματος. Στην περίπτωση πολλών ημερομηνιών σε διανύσματα, συνδέονται "άσχετες" ποσότητες μεταξύ τους.
- Δεν μπορούμε να επιβάλουμε όρια αποδεκτών τιμών ή να ελέγξουμε αυτόματα την εγκυρότητα της ημερομηνίας που αποθηκεύεται, π.χ. οι 31/4/2009, 29/2/2005, 29/2/1900 είναι σωστές;
- Δεν μπορούμε να εμποδίσουμε τροποποιήσεις κατά λάθος.

Δεύτερη προσέγγιση

Για μια ημερομηνία ορίζουμε ένα **struct** με τρία μέλη:

```
struct date {
   int d, m, y;
};
date z{3,5,2001};
```

Αν έχουμε πολλές ημερομηνίες, ορίζουμε *ένα* `std::vector<date>`.

Πλεονέκτημα

Οι μεταβλητές που αντιπροσωπεύουν ημερομηνία σχετίζονται ισχυρά μεταξύ τους, μεταφέρονται ως σύνολο σε όρισμα συνάρτησης, κλπ.

Μειονεκτήματα

- Δεν μπορούμε να επιβάλουμε όρια αποδεκτών τιμών ή να ελέγξουμε αυτόματα την εγκυρότητα της ημερομηνίας που αποθηκεύεται.
- Δεν μπορούμε να εμποδίσουμε κατά λάθος αλλαγές: π.χ. επιτρέπεται το `z.m=16;`.

Τρίτη προσέγγιση

Θα μας βόλευε να έχουμε έτοιμο τον τύπο `date` και να μπορούμε να κάνουμε:

```
date a{14,5,2005};    // a = 14/5/2005
date b{31,6,2009};    // error
date c{a};            // c = 14/5/2005
date d{c+3};          // d = 17/5/2005
date e{d-1000};       // e = 21/8/2002
int k{a-e};           // k = 997
--d;                  // d = 16/5/2005
++e;                  // e = 22/8/2002
d += 4;               // d = 20/5/2005
e = d;                // e = 20/5/2005
```

Στις γλώσσες που υποστηρίζουν αντικειμενοστρεφή προγραμματισμό (object-oriented programming) μπορούμε να κατασκευάσουμε νέους τύπους.

Κατασκευή νέου τύπου (1/2)

Η κατασκευή ενός νέου τύπου περιλαμβάνει συνοπτικά

- τη δήλωση των ποσοτήτων που χρειάζονται για την αποθήκευση των πληροφοριών και της κατάστασης ενός αντικειμένου αυτού του τύπου. Στην απλή περίπτωση οι ποσότητες αυτές είναι κάποιες μεταβλητές και αποτελούν την *εσωτερική αναπαράσταση* του αντικειμένου.
- τη λεπτομερή περιγραφή του τρόπου *δημιουργίας* ενός αντικειμένου. Η δημιουργία του μπορεί να γίνει
  - από ανεξάρτητες ποσότητες που θα δώσουν τιμές στις εσωτερικές μεταβλητές ή
  - από άλλο αντικείμενο της ίδιας κλάσης, με αντιγραφή ή μετακίνηση.
- τη λεπτομερή περιγραφή του τρόπου *καταστροφής* ενός αντικειμένου.
- τη συμπεριφορά του *τελεστή εκχώρησης* ενός αντικειμένου σε άλλο. Η εκχώρηση μπορεί να γίνει με αντιγραφή ή μετακίνηση.
- τον ορισμό συναρτήσεων για την προσπέλαση ή τροποποίηση των μελών της εσωτερικής αναπαράστασης.

Κατασκευή νέου τύπου (2/2)

Πιθανόν να χρειάζεται για νέο τύπο, αν έχουν νόημα, να ορίσουμε

- τελεστές που δρουν σε αντικείμενα του συγκεκριμένου τύπου (αριθμητικοί, σύγκρισης, τελεστής '( )', τελεστής '[ ]', κλπ.) και
- πώς γίνεται η μετατροπή του συγκεκριμένου τύπου σε άλλο.

Ορισμός κλάσης (1/2)

Ο ορισμός της κλάσης (του νέου τύπου, δηλαδή) εισάγεται με τη λέξη **`class`** (ή τη **`struct`**), ακολουθούμενης από το όνομα του τύπου που δημιουργούμε. Μέσα σε άγκιστρα '{}' παραθέτουμε με οποιαδήποτε σειρά τα *μέλη* του· αυτά υλοποιούν χαρακτηριστικά και ιδιότητές του. Ο ορισμός κλείνει με το καταληκτικό ';'.

Παράδειγμα

```
class date {
...
};
```

Ορισμός κλάσης (2/2)

Ετικέτες πρόσβασης

- Το τμήμα μεταξύ των {} στον ορισμό της κλάσης μπορεί να χωριστεί σε τμήματα που εισάγονται με τις ετικέτες **`public`**: και **`private`**:. Η πλησιέστερη, προς τα επάνω, ετικέτα είναι αυτή που καθορίζει την εμβέλεια των μελών που ακολουθούν.
- Για όσα μέλη οι δηλώσεις έπονται της ετικέτας **`public`**: επιτρέπεται η πρόσβαση και χρήση τους από οποιοδήποτε σημείο του κώδικα. Για όσα μέλη δηλώνονται μετά την ετικέτα **`private`**: η πρόσβαση επιτρέπεται μόνο από άλλα μέλη της κλάσης.
- Οι ετικέτες μπορούν να επαναλαμβάνονται στο σώμα της κλάσης, δεν έχουν προκαθορισμένη σειρά και δεν είναι απαραίτητο να ακολουθούνται από δηλώσεις μελών.
- Μετά το εναρκτήριο άγκιστρο της κλάσης υπονοείται η ετικέτα **`private`**: αν ο ορισμός εισάγεται με τη λέξη **`class`**, και η ετικέτα **`public`**: αν χρησιμοποιήθηκε η λέξη **`struct`**.

Εσωτερική αναπαράσταση κλάσης

Παράδειγμα

Η εσωτερική αναπαράσταση της κλάσης date μπορεί να αποτελείται από τρεις ακέραιους, d,m,y. Στο εσωτερικό της κλάσης (εντός των '{}'), μετά από μια ετικέτα **private**: γράφουμε

```
int d,m,y;
```

δηλαδή έχουμε

```
class date {
...
private:
int d,m,y;
...
};
```

Constructor (1/3)

Για την περιγραφή της δημιουργίας αντικειμένου από άλλες ποσότητες γράφουμε εντός της κλάσης ένα constructor (κατασκευαστή), μια *ιδιότυπη* συνάρτηση που δέχεται ως ορίσματα κάποιες ποσότητες (ή και τίποτε) και δίνει τιμές στα μέλη της υλοποίησης.
Ο constructor

- δεν είναι μοναδικός,
- έχει ως όνομα το όνομα της κλάσης,
- δεν έχει τύπο επιστρεφόμενης ποσότητας και
- κάνει ιδιότυπη εκχώρηση αρχικών τιμών στα μέλη της αναπαράστασης, αμέσως μετά τα ορίσματα και πριν το σώμα του, με λίστα που αρχίζει με το ‘:΄.
- Αν χρειάζεται, μπορεί να περιλαμβάνει εντολές στο σώμα του.

Constructor (2/3)

Για την κλάση date μπορούμε να δημιουργήσουμε ένα αντικείμενο

- από τρεις ακέραιες ποσότητες. Ο constructor είναι

```
date(int d1, int m1, int y1)
: d{d1}, m{m1}, y{y1}
{ ... }
```

- από ένα ακέραιο (π.χ. 20191204). Ο constructor είναι

```
date(int k)
: d{k%100}, m{(k/100)%100}, y{k/10000}
{ ... }
```

- με συγκεκριμένη ημερομηνία (π.χ. 1/1/1970). Ο constructor είναι

```
date()
: d{1}, m{1}, y{1970}
{}
```

Παρατηρήστε τη λίστα αρχικοποίησης της εσωτερικής αναπαράστασης. Στο σώμα των constructors μπορούμε να περιλάβουμε εντολές που θα ελέγχουν την εγκυρότητα της ημερομηνίας.

Constructor (3/3)

Δημιουργία αντικειμένου τύπου X έχουμε όταν, μεταξύ άλλων,

- δηλώνουμε μια μεταβλητή ή σταθερή του τύπου X,
- περνάμε μια ποσότητα τύπου X σε όρισμα συνάρτησης,
- επιστρέφουμε ποσότητα από συνάρτηση που επιστρέφει X.

Η επιλογή του constructor που θα κληθεί *αυτόματα*, γίνεται με βάση τη σειρά, το πλήθος και τον τύπο των τιμών αρχικοποίησης του αντικειμένου.

Παράδειγμα

Στις δηλώσεις

```
date a{3,12,1980}; // a= 3/12/1980
date b{20191204};  // b= 4/12/2019
date c;            // c= 1/1/1970
```

καλούνται για τα a,b,c ο πρώτος, ο δεύτερος και ο τρίτος constructor της κλάσης date αντίστοιχα.