Δ.Ι.Ε.Κ. ΜΕΤΑΞΟΥΡΓΕΙΟΥ

ΣΗΜΕΙΩΣΕΙΣ

Εισαγωγή στη γλώσσα C++

ΑΘΗΝΑ 2018

Περιεχόμενα

Πρόλογος

Οι παρούσες σημειώσεις αποτελούν μια εισαγωγή στον προγραμματισμό ηλεκτρονικών υπολογιστών. Ως γλώσσα προγραμματισμού χρησιμοποιείται η C++, όπως αυτή διαμορφώθηκε με το Standard του 2014 (ISO/IEC 14882 : 2014) [1].

Η γλώσσα C++ είναι κατά γενική ομολογία ιδιαίτερα πλούσια στις δυνατότητες οργάνωσης του κώδικα και έκφρασης που παρέχει στον προγραμματιστή. Συγχρόνως, το εύρος αυτό των δυνατοτήτων την καθιστά πιο απαιτητική στην εκμάθηση σε σύγκριση με άλλες γλώσσες όπως η Fortran και η C.

Οι παρούσες σημειώσεις δεν έχουν στόχο να υποκαταστήσουν την ήδη υπάρχουσα σχετική βιβλιογραφία (αγγλική κυρίως αλλά και ελληνική) ούτε και να καλύψουν τη γλώσσα C++ σε όλες τις επιμέρους δυνατότητές της. Φιλοδοξία μου είναι παρουσιαστούν οι βασικές έννοιες του προγραμματισμού και να δοθεί μια όσο το δυνατόν πιο άρτια και πλήρης περιγραφή/περίληψη ενός μέρους της γλώσσας C++. Ελπίζω να αποτελέσουν οι σημειώσεις μια στέρεη βάση πάνω στην οποία ο κάθε αναγνώστης, με δική του πλέον πρωτοβουλία, θα μπορέσει να αναπτύξει περαιτέρω την απαιτούμενη δεξιότητα του προγραμματισμού στις υπολογιστικές επιστήμες.

Καθώς οι σημειώσεις απευθύνονται σε αρχάριους προγραμματιστές, επιλέχθηκε να γίνει παρουσίαση της C++ ως μιας βελτιωμένης C· δίνεται προτεραιότητα στη χρήση έτοιμων δομών και εννοιών έναντι των μηχανισμών δημιουργίας τους, τα κεφάλαια για τη Standard Library προηγούνται του κεφαλαίου για τις κλάσεις, κ.α. Ίσως αυτή η προσέγγιση βοηθήσει στο να ανατραπεί η εικόνα που έχουν πολλοί για τη C++ ως γλώσσας αποκλειστικά για *αντικειμενοστρεφή*[1] προγραμματισμό («κάτι προχωρημένο που δε μας χρειάζεται»).

[1] http://www.dmst.aueb.gr/dds/faq/academic.html#oo

Μέρος Ι

Βασικές Έννοιες

Κεφάλαιο 1

Εισαγωγή

Ένας ηλεκτρονικός υπολογιστής έχει τη δυνατότητα να *προγραμματιστεί* ώστε να εκτελέσει μια συγκεκριμένη διαδικασία.

Το πρώτο στάδιο του προγραμματισμού είναι να αναλύσουμε την επιθυμητή διεργασία σε επιμέρους στοιχειώδεις έννοιες και να προσδιορίσουμε τον τρόπο και τη σειρά αλληλεπίδρασης αυτών. Το βήμα αυτό αποτελεί την ανάπτυξη της μεθόδου, του *αλγορίθμου* όπως λέγεται, που θα επιτύχει την εκτέλεση της διεργασίας. Απαραίτητη προϋπόθεση, βέβαια, είναι να έχουμε *ορίσει* σαφώς και να έχουμε *κατανοήσει* πλήρως την επιθυμητή διαδικασία.

Ο αλγόριθμος συνήθως απαιτεί *δεδομένα*, τιμές που πρέπει να προσδιοριστούν πριν την εκτέλεσή του, τα οποία πρόκειται να επεξεργαστεί ώστε να παραγάγει κάποιο αποτέλεσμα. Όμως, η μέθοδος σχεδιάζεται και αναπτύσσεται *ανεξάρτητα* από συγκεκριμένες τιμές των δεδομένων αυτών.

Κατά την ανάπτυξη του αλγορίθμου χρησιμοποιούμε *σταθερές* και *μεταβλητές* ποσότητες, δηλαδή, θέσεις στη μνήμη του υπολογιστή, για την αποθήκευση των ποσοτήτων (δεδομένων, ενδιάμεσων τιμών και αποτελεσμάτων) του προγράμματος. Υπάρχει η δυνατότητα *εκχώρησης* τιμής στις μεταβλητές, *επιλογής* του επόμενου βήματος, ανάλογα με κάποια συνθήκη, καθώς και η δυνατότητα *επανάληψης* ενός η περισσότερων βημάτων. Ανάλογα με τις δυνατότητες της γλώσσας προγραμματισμού που θα χρησιμοποιήσουμε στο επόμενο στάδιο, μπορούμε να κάνουμε χρήση δομών για την ομαδοποίηση ποσοτήτων (π.χ. πίνακας), ομαδοποίηση και παραμετροποίηση πολλών εντολών (π.χ. συνάρτηση), ή ακόμα και ορισμό νέων τύπων (π.χ. κλάση). Επιπλέον, μπορούμε να επιλέξουμε και να εφαρμόσουμε ένα από διάφορους τρόπους οργάνωσης και αλληλεπίδρασης των βασικών εννοιών του προγράμματός μας. Έτσι, μπορούμε να ακολουθήσουμε την τεχνική του δομημένου/διαδικαστικού προγραμματισμού, τον αντικειμενοστρεφή προγραμματισμό κλπ. ανάλογα με το πρόβλημα και την πολυπλοκότητά του.

Το επόμενο στάδιο του προγραμματισμού είναι να μεταγράψουμε, σε κάποια γλώσσα, τον αλγόριθμο με τις συγκεκριμένες έννοιες και αλληλεπιδράσεις, να γράψουμε δηλαδή τον κώδικα. Κατόπιν, ο ηλεκτρονικός υπολογιστής θα μεταφράσει τον κώδικά μας από τη μορφή που κατανοούμε εμείς σε μορφή που κατανοεί ο ίδιος, ώστε να μπορέσει να ακολουθήσει τα βήματα που προσδιορίζουμε στον κώδικα και να ολοκληρώσει την επιθυμητή διεργασία.

Η επιλογή της κατάλληλης γλώσσας βασίζεται στις δομές και τις έννοιες προγραμματισμού που αυτή υλοποιεί. Ανάλογα με το πεδίο στο οποίο εστιάζει, μια γλώσσα μπορεί να παρέχει ειδικές δομές και τρόπους οργάνωσης του κώδικα, κατάλληλους για συγκεκριμένες εφαρμογές.

Στις παρούσες σημειώσεις θα περιγράψουμε έννοιες της γλώσσας προγραμματισμού C++ και θα αναφερθούμε στις βασικές δομές που αυτή παρέχει. Έμφαση θα δοθεί στην τεχνική του δομημένου προγραμματισμού για την ανάπτυξη του κώδικα. Στο τέλος, θα παρουσιαστούν εισαγωγικά οι βασικές έννοιες του αντικειμενοστρεφούς προγραμματισμού και οι δομές που παρέχει η γλώσσα για να τις υλοποιήσει.

Παρακάτω θα παραθέσουμε ένα τυπικό κώδικα σε C++ και θα περιγράψουμε τη λειτουργία του. Στα επόμενα κεφάλαια θα αναφερθούμε στις εντολές και δομές της C++ που χρειάζονται για να αναπτύξουμε σχετικά πολύπλοκους κώδικες.

1.1 Παράδειγμα

Ας εξετάσουμε μία απλή εργασία που θέλουμε να εκτελεστεί από ένα ηλεκτρονικό υπολογιστή: να μας ζητά ένα πραγματικό αριθμό και να τυπώνει στην οθόνη το τετράγωνό του. Η διαδικασία που ακολουθούμε, ο *αλγόριθμος*, είναι η εξής:

1. Ανάγνωση αριθμού από το πληκτρολόγιο — [εισαγωγή του στη μνήμη].
2. [ανάκληση του αριθμού από τη μνήμη] — υπολογισμός του τετραγώνου — [εισαγωγή του αποτελέσματος στη μνήμη].
3. [ανάκληση του αποτελέσματος από τη μνήμη] — Εκτύπωση στην οθόνη.

Η ενέργειες που περιλαμβάνονται σε αγκύλες μπορεί να μη φαίνονται αναγκαίες σε πρώτη ανάγνωση. Είναι όμως, καθώς ο υπολογιστής κρατά στη μνήμη του (RAM) οποιαδήποτε πληροφορία, σε μεταβλητές ή σταθερές ποσότητες.

Πολλές γλώσσες προγραμματισμού, ανάμεσά τους και η C++, χρειάζονται ένα επιπλέον, προκαταρκτικό, βήμα σε αυτή τη διαδικασία. Προτού μεταγράψουμε τον αλγόριθμο πρέπει να κάνουμε το:

0. Δήλωση μεταβλητών (δηλαδή, *ρητή* δέσμευση μνήμης).

Ας δούμε ένα πλήρες πρόγραμμα C++ που εκτελεί την παραπάνω εργασία ακολουθώντας τα βήματα που περιγράψαμε. Με αυτό έχουμε την ευκαιρία να παρουσιάσουμε βασικά στοιχεία της δομής του κώδικα. Στα επόμενα κεφάλαια ακολουθούν πιο αναλυτικές περιγραφές τους.

```
#include <iostream>

/*
   main:
         Den pairnei orismata.
         Zhta ena pragmatiko kai typwnei to tetragwno tou.
         Epistrefei 0.
*/
int
main()
{
    double a;  // Dhlwsh pragmatikhs metavlhths.

    std::cout << "Dose␣pragmatiko␣arithmo:␣";
    // Mhnyma sthn othoni

    std::cin >> a; // Eisagwgi timhs apo plhktrologio

    std::cout << "To␣tetragwno␣einai:␣"; // Mhnyma sthn othoni

    double b;   // Dhlwsh allhs metablhths

    b = a * a;  // Ypologismos

    std::cout << b << '\n';
    // Ektypwsi apotelesmatos kai allagh grammhs

    return 0; // Epistrofh me epityxia.
}
```

Ας το αναλύσουμε:

1.2 Οδηγίες προεπεξεργαστή

Η γλώσσα C++ έχει λίγες ενσωματωμένες εντολές, σε σύγκριση με άλλες γλώσσες προγραμματισμού. Μία πληθώρα άλλων εντολών και δυνατοτήτων παρέχεται από τη Standard Library, τμήματα της οποίας μπορούμε να συμπεριλάβουμε με «οδηγίες» (directives), `#include` προς τον προεπεξεργαστή. Η προεπεξεργασία του κώδικα γίνεται αυτόματα ως πρώτη φάση της μεταγλώττισης. Η οδηγία στην πρώτη γραμμή του παραδείγματος,

```
#include <iostream>
```

προκαλεί την εισαγωγή του *header* `<iostream>` και δίνει τη δυνατότητα στον κώδικά μας να χρησιμοποιήσει, ανάμεσα σε άλλα, το πληκτρολόγιο και την οθόνη

για είσοδο και έξοδο δεδομένων. Οι κατάλληλες οδηγίες `#include` (αν υπάρχουν) κανονικά πρέπει να εμφανίζονται στην αρχή κάθε αρχείου με κώδικα C++. Επιπλέον, πρέπει να βρίσκονται μόνες τους στη γραμμή (ή να ακολουθούνται μόνο από σχόλια) και ο πρώτος μη κενός χαρακτήρας τους να είναι ο '#'.

1.3 Σχόλια

Ο μεταγλωττιστής (compiler) αγνοεί τους χαρακτήρες που περιλαμβάνονται μεταξύ

- του `//` και του τέλους της γραμμής στην οποία εμφανίζεται αυτός ο συνδυασμός χαρακτήρων,
- των `/*` και `*/` ανεξάρτητα από το πλήθος γραμμών που περιλαμβάνουν.

Οι χαρακτήρες αυτοί αποτελούν τα *σχόλια* και πρέπει να είναι μόνο λατινικοί (σχεδόν πάντα από το σύνολο χαρακτήρων ASCII).

Ένα σχόλιο μεταξύ των `/*` και `*/` μπορεί να εμφανίζεται όπου επιτρέπεται να υπάρχει ο χαρακτήρας tab, κενό ή αλλαγή γραμμής (δείτε το §1.9). Προσέξτε ότι τέτοιου τύπου σχόλιο δεν μπορεί να περιλαμβάνει άλλο σχόλιο μεταξύ των ίδιων συμβόλων.

Η ύπαρξη *επαρκών* και *σωστών* σχολίων σε ένα κώδικα βοηθά σημαντικά στην κατανόησή του από άλλους ή και εμάς τους ίδιους, όταν, μετά από καιρό, θα έχουμε ξεχάσει τι και πώς ακριβώς το κάνει το συγκεκριμένο πρόγραμμα. Προσέξτε όμως ότι η ύπαρξη σχολίων δυσνόητων ή υπερβολικά σύντομων, που δεν αντιστοιχούν στην τρέχουσα μορφή του κώδικα ή που δεν διευκρινίζουν τι ακριβώς γίνεται, είναι χειρότερη από την πλήρη έλλειψή τους.

Χρήση των παραπάνω στοιχείων της γλώσσας γίνεται συχνά για την απομόνωση κώδικα. Εναλλακτικά, η χρήση του προεπεξεργαστή προσφέρει ένα ιδιαίτερα βολικό μηχανισμό για κάτι τέτοιο: ο compiler αγνοεί τμήμα κώδικα που περικλείεται μεταξύ των

```
#if 0
.....
#endif
```

1.4 Κυρίως πρόγραμμα

Η δήλωση `int main() {...}` ορίζει τη βασική συνάρτηση (§7.9) σε κάθε πρόγραμμα C++: το όνομά της είναι `main`, επιστρέφει ένα ακέραιο αριθμό (`int`, §2.5.1), ενώ, στο συγκεκριμένο ορισμό, δε δέχεται ορίσματα· δεν υπάρχουν ποσότητες μεταξύ των παρενθέσεων που ακολουθούν το όνομα. Οι εντολές (αν υπάρχουν) μεταξύ των αγκίστρων '{}' που ακολουθούν την κενή λίστα ορισμάτων είναι ο κώδικας

που εκτελείται με την κλήση της. Η συγκεκριμένη συνάρτηση πρέπει να υπάρχει και να είναι μοναδική σε ένα ολοκληρωμένο πρόγραμμα C++. Η εκτέλεση του προγράμματος ξεκινά με την κλήση της από το λειτουργικό σύστημα και τελειώνει με την *επιστροφή τιμής* σε αυτό, είτε ρητά, όπως στο παράδειγμα (`return 0;`) είτε εμμέσως, όταν η ροή συναντήσει το καταληκτικό άγκιστρο '}' (οπότε επιστρέφεται αυτόματα το 0)[1]. Η επιστροφή της τιμής 0 από τη `main()` υποδηλώνει την επιτυχή εκτέλεσή της. Οποιαδήποτε άλλη ακέραια τιμή ενημερώνει το λειτουργικό σύστημα για κάποιο σφάλμα.

Ένα αρχείο κώδικα μπορεί να περιλαμβάνει και άλλες συναρτήσεις, ορισμένες πριν ή μετά τη `main()`. Αυτές εκτελούνται μόνο αν κληθούν από τη `main()` ή από συνάρτηση που καλείται από αυτή.

1.5 Δήλωση μεταβλητής

Οι *μεταβλητές* (variables) είναι θέσεις στη μνήμη που χρησιμοποιούνται για την αποθήκευση των ποσοτήτων (δεδομένων, αποτελεσμάτων) του προγράμματος. Προτού χρησιμοποιηθούν πρέπει να *δηλωθούν*, δηλαδή να ενημερωθεί ο compiler για το όνομά τους και τον τύπο τους. Επιπλέον, *προτού* συμμετάσχουν σε υπολογισμούς ή εντολές που χρειάζονται την τιμή τους πρέπει *να αποκτήσουν συγκεκριμένη τιμή*, αν θέλουμε προβλέψιμα αποτελέσματα.

Η εντολή

```
double a;
```

αποτελεί *δήλωση* μίας μεταβλητής του προγράμματός μας. Η συγκεκριμένη εντολή ζητά από τον compiler να δεσμεύσει χώρο στη μνήμη για ένα πραγματικό αριθμό διπλής ακρίβειας (`double`), με το όνομα a. Οι πραγματικοί διπλής ακρίβειας (§2.5.2) έχουν (συνήθως) δεκαπέντε σημαντικά ψηφία[2] σωστά.

Παρατηρήστε ότι στον αλγόριθμό μας έχουμε δύο εισαγωγές πραγματικών στη μνήμη οπότε θέλουμε δύο κατάλληλες μεταβλητές. Η δήλωση της δεύτερης μεταβλητής γίνεται με την εντολή `double b;` λίγο πριν χρησιμοποιηθεί αυτή. Οι δηλώσεις ποσοτήτων στη C++ μπορούν να γίνουν σε οποιοδήποτε σημείο του κώδικά μας. Καλό είναι μια μεταβλητή να δηλώνεται αμέσως πριν χρησιμοποιηθεί, ή, όπως θα δούμε παρακάτω, να γίνεται δήλωση με απόδοση αρχικής τιμής, στο σημείο του κώδικα που θα γνωρίζουμε την αρχική τιμή της.

1.6 Χαρακτήρες

Ένας ή περισσότεροι χαρακτήρες μεταξύ διπλών εισαγωγικών (") αποτελούν μια *σταθερή σειρά χαρακτήρων*, ένα C-style string. Τέτοια σταθερά είναι το κείμενο

[1] Προσέξτε ότι η τελευταία περίπτωση ισχύει μόνο για τη `main()` και όχι για άλλες συναρτήσεις.

[2] Σε ένα αριθμό, τα ψηφία από το αριστερότερο μη μηδενικό έως το δεξιότερο (μηδενικό ή όχι), ανεξάρτητα από τη θέση της υποδιαστολής (ή τελείας), χαρακτηρίζονται ως *σημαντικά*.

`"To␣tetragwno␣einai:␣"`.

Ένας μόνο χαρακτήρας μεταξύ απλών εισαγωγικών (') αποτελεί μια *σταθερή ποσότητα τύπου χαρακτήρα* (character literal, §2.5.4). Τέτοιες είναι οι χαρακτήρες `'a'`, `'1'`, κλπ.

Εκτός από τους απλούς χαρακτήρες, υπάρχουν και οι *ειδικοί*. Ο ειδικός χαρακτήρας εισάγεται με '\' και ακολουθούν ένας ή περισσότεροι συγκεκριμένοι χαρακτήρες· το σύμπλεγμα θεωρείται όμως ως *ένας*. Τέτοιος είναι ο `'\n'`, ο οποίος προκαλεί την αλλαγή γραμμής. Οι ειδικοί χαρακτήρες συνήθως δεν εκτυπώνονται όταν αποστέλλονται στην «έξοδο» του προγράμματος αλλά εκτελούν συγκεκριμένες λειτουργίες. Καθώς δεν είναι μεταβλητές, δεν έχει νόημα (και είναι λάθος) να χρησιμοποιηθούν κατά την «είσοδο» δεδομένων.

Οι χαρακτήρες που εμφανίζονται στον κώδικα, απλοί ή σε σειρά, καλό είναι να είναι μόνο λατινικοί (σχεδόν πάντα, αλλά όχι απαραίτητα, από το σύνολο χαρακτήρων ASCII). Υπάρχει η δυνατότητα να έχουμε π.χ. ελληνικούς χαρακτήρες σε σταθερή σειρά χαρακτήρων, αν η σειρά έχει ακριβώς πριν το αρχικό διπλό εισαγωγικό τους χαρακτήρες `u8` και η εισαγωγή των χαρακτήρων έγινε στο σύστημα utf-8, π.χ.

`u8`"Το τετράγωνο είναι"

1.7 Είσοδος και έξοδος δεδομένων

Τα αντικείμενα `std::cin` και `std::cout` αντιπροσωπεύουν το πληκτρολόγιο και την οθόνη αντίστοιχα, ή γενικότερα, το standard input (είσοδο) και standard output (έξοδο) του εκτελέσιμου αρχείου.

Η εντολή

```
std::cout << "Dose␣arithmo";
```

προκαλεί την εκτύπωση στην οθόνη ενός συγκεκριμένου κειμένου, των χαρακτήρων μεταξύ των εισαγωγικών. Αντίστοιχα, η εντολή

```
std::cout << b;
```

εκτυπώνει *την τιμή* που είναι αποθηκευμένη στη μεταβλητή b (και η οποία ανακαλείται από τη μνήμη).

Ανάλογα, η εντολή

```
std::cin >> a;
```

αναμένει να «διαβάσει» από το πληκτρολόγιο ένα πραγματικό αριθμό και να τον αποθηκεύσει στη μεταβλητή a. Η συγκεκριμένη εντολή αποτελεί τον ένα από τους τρεις βασικούς τρόπους για άμεση εκχώρηση τιμής σε μεταβλητή. Οι άλλοι είναι η *εντολή εκχώρησης* (§2.4) και η *απόδοση τιμής κατά τη δήλωση* («αρχικοποίηση», §2.2.1).[3]

[3] Επιπλέον, υπάρχουν οι μηχανισμοί απόδοσης τιμής μέσω αναφοράς (§2.17) ή δείκτη (§2.18).

Γενικότερα, ο τελεστής '<<', όταν χρησιμοποιείται για εκτύπωση («έξοδο») δεδομένων στην οθόνη, «στέλνει» την ποσότητα που τον ακολουθεί (το δεξί όρισμά του) στο `std::cout` (που είναι πάντα το αριστερό όρισμά του). Αντίστοιχα, ο τελεστής '>>' διαβάζει από το `std::cin` (το αριστερό όρισμά του) τιμή που την εκχωρεί στην (υποχρεωτικά μεταβλητή) ποσότητα που τον ακολουθεί. Η χρήση των παραπάνω προϋποθέτει, όπως αναφέρθηκε, τη συμπερίληψη του header `<iostream>`.

Παρατηρήστε ότι στη C++ δεν προσδιορίζουμε συγκεκριμένη διαμόρφωση για την είσοδο/έξοδο δεδομένων. Η σχετική πληροφορία συνάγεται από τον τύπο των μεταβλητών που τα αντιπροσωπεύουν. Αυτό, βέβαια, δε σημαίνει ότι δεν μπορούμε να καθορίσουμε π.χ. το πλήθος των σημαντικών ψηφίων ή τη στοίχιση των αριθμών που θα τυπωθούν, όπως θα δούμε αργότερα (§6.5). Όταν μεταγλωττίσουμε και εκτελέσουμε το πρόγραμμα, θα παρατηρήσουμε ότι η εκτύπωση κάποιας πληροφορίας στην οθόνη δεν συνοδεύεται από αλλαγή γραμμής. Η αλλαγή γραμμής πρέπει να γίνει *ρητά* στέλνοντας στην «έξοδο» του προγράμματος τον ειδικό χαρακτήρα '`\n`'. Προσέξτε επίσης πώς γίνεται, σε μία εντολή, εκτύπωση πολλών ποσοτήτων (ή γενικότερα, αποστολή πληροφορίας από πολλές «πηγές») στην έξοδο: στην εντολή

```
std::cout << b << '\n';
```

η μεταβλητή `b` και η αλλαγή γραμμής στέλνονται στο `std::cout` με τη χρήση του τελεστή '<<' *πριν από κάθε ποσότητα που εκτυπώνεται.*

Εκτός από τα `std::cin` και `std::cout`, η συμπερίληψη του header `<iostream>` παρέχει στον κώδικά μας και το `std::cerr`. Το συγκεκριμένο αντικείμενο συνδέεται αυτόματα στο standard error του προγράμματός μας και κανονικά χρησιμοποιείται για τα μηνύματα λάθους. Όταν στέλνουμε δεδομένα σε αυτό με τον τελεστή '<<', εμφανίζονται στην οθόνη (η οποία είναι το προκαθορισμένο standard error). Κατά την εκτέλεση του προγράμματος υπάρχει η δυνατότητα ανακατεύθυνσης (π.χ. από/σε αρχείο) των standard input, standard output, standard error.

1.8 Υπολογισμοί και εκχώρηση

Η εντολή

```
b = a * a;
```

αποτελεί μια *εντολή εκχώρησης* (§2.4) τιμής σε μεταβλητή. Στο δεξί μέλος της συγκεκριμένης γίνεται ανάκληση από τη μνήμη του αριθμού που είναι αποθηκευμένος στη μεταβλητή `a` και εκτελείται η προσδιοριζόμενη πράξη, ο πολλαπλασιασμός με τον εαυτό του. Ο τελεστής '`*`' μεταξύ πραγματικών αριθμών αντιπροσωπεύει τη γνωστή πράξη του πολλαπλασιασμού. Αφού υπολογιστεί το αποτέλεσμα, αποθηκεύεται στη μεταβλητή του αριστερού μέλους.

1.9 Διαμόρφωση του κώδικα

Κάθε εντολή τελειώνει με ελληνικό ερωτηματικό, ';', και εκτελείται με τη σειρά που εμφανίζεται στο αρχείο (εκτός, βέβαια, αν αλλάξει η ροή εκτέλεσης με κατάλληλες εντολές). Παρατηρήστε ότι οι οδηγίες προς τον προεπεξεργαστή δεν επιτρέπεται να έχουν τελικό ';'[4].

Η C++ δεν προβλέπει κάποια συγκεκριμένη διαμόρφωση του κώδικα· τα κενά, οι αλλαγές γραμμής κλπ. δεν έχουν κάποιο ιδιαίτερο ρόλο παρά μόνο να διαχωρίζουν διαδοχικές λέξεις της C++ ή ονόματα ποσοτήτων. Οι θέσεις αυτών είναι ελεύθερες (δείτε πόσο ακραίες διαμορφώσεις μπορείτε να συναντήσετε στο Παράρτημα Α΄). Ο χαρακτήρας tab, το κενό, τα σχόλια και η αλλαγή γραμμής δεν μπορούν

- να διαχωρίζουν τα σύμβολα που αποτελούν σύνθετους τελεστές (`+=`, `==`, `<<`,`>>`, `/*`, `//`,...),
- να βρίσκονται στο «εσωτερικό» πολυψήφιων αριθμών ή ονομάτων μεταβλητών, σταθερών, κλπ.

Επιπλέον, σταθερές σειρές χαρακτήρων, σχόλια που αρχίζουν με `//` και οδηγίες προς τον προεπεξεργαστή δεν επιτρέπεται να εκτείνονται σε περισσότερες από μία γραμμές, παρά μόνο αν στο τέλος της γραμμής που θέλουμε να συνεχιστεί στην επόμενη υπάρχει ο χαρακτήρας '\' *μόνο*, χωρίς να ακολουθείται από κανένα άλλο, ούτε καν από τον κενό.

Στη C++ τα κεφαλαία και πεζά γράμματα *είναι διαφορετικά*. Οι προκαθορισμένες λέξεις της γλώσσας και τα ονόματα των headers, συναρτήσεων, χώρων ονομάτων, κλπ. που παρέχει αυτή, γράφονται με πεζά.

1.10 Ασκήσεις

1. Γράψτε, μεταγλωττίστε και εκτελέστε τον κώδικα του παραδείγματος.[5]
2. Ποιο είναι το πιο σύντομο σωστό πρόγραμμα C++;
3. Τέσσερα διαφορετικά λάθη υπάρχουν στον παρακάτω κώδικα C++· ποια είναι;

```
#include <iostream>
main(){std:cout << 'Hello␣World!\n'}
```

[4] Αν το περιλαμβάνουν, αποτελεί μέρος της εντολής και όχι κατάληξή της.

[5] Το πώς θα τα κάνετε αυτά εξαρτάται από το λειτουργικό σύστημα και τον compiler που χρησιμοποιείτε και γι' αυτό δε δίνονται εδώ λεπτομέρειες.

Κεφάλαιο 2

Τύποι και Τελεστές

2.1 Εισαγωγή

Η C++ παρέχει ένα σύνολο θεμελιωδών τύπων που αντιστοιχούν στα πιο συνηθισμένα είδη δεδομένων. Επιπλέον, δίνει τη δυνατότητα στον προγραμματιστή να δημιουργήσει δικούς του τύπους ή να χρησιμοποιήσει σύνθετους τύπους που έχουν οριστεί στη Standard Library.

Ο τύπος, δηλαδή το είδος μιας ποσότητας, προσδιορίζει

- το πλήθος των θέσεων στη μνήμη που καταλαμβάνει μια ποσότητα,
- τις δυνατές τιμές που μπορεί να πάρει αυτή,
- τις πράξεις στις οποίες μπορεί να συμμετέχει.

Η C++ ενσωματώνει τύπους για το χειρισμό ποσοτήτων που είναι ακέραιες, πραγματικές, χαρακτήρες ή λογικές (boolean).

Ο μιγαδικός τύπος στη C++ δεν περιλαμβάνεται στους θεμελιώδεις αλλά παράγεται από ζεύγη αριθμητικών ποσοτήτων και παρέχεται από τη Standard Library μέσω *class template* (§14.9). Θα τον παρουσιάσουμε στο παρόν κεφάλαιο (§2.13) λόγω της μεγάλης χρησιμότητας των αριθμών τέτοιου τύπου σε επιστημονικούς κώδικες.

Προτού αναπτύξουμε την περιγραφή των τύπων, θα δούμε πώς δηλώνεται μια μεταβλητή, ποιοι κανόνες διέπουν το όνομά της και θα εξηγήσουμε τη βασική εντολή με την οποία αποκτά τιμή, την εντολή εκχώρησης. Θα παρουσιάσουμε το πώς ορίζουμε σταθερές ποσότητες και θα αναφερθούμε στην εμβέλεια των μεταβλητών, σταθερών, κλπ. που δηλώνουμε. Θα ακολουθήσει η περιγραφή των αριθμητικών και άλλων τελεστών, των κανόνων μετατροπής της τιμής μιας ποσότητας από ένα τύπο σε άλλο καθώς και του χώρου ονομάτων. Θα παρουσιάσουμε τις

έννοιες της αναφοράς και του δείκτη και θα κλείσουμε το κεφάλαιο με την παρουσίαση των μαθηματικών συναρτήσεων που παρέχει η C++, λόγω της μεγάλης χρησιμότητάς τους σε επιστημονικούς κώδικες.

2.2 Δήλωση μεταβλητής

Όπως αναφέραμε, κάθε ποσότητα προτού χρησιμοποιηθεί πρέπει να δηλωθεί, δηλαδή να ενημερωθεί ο compiler για το όνομα και τον τύπο της. Η δήλωση μπορεί να γίνει σε όποιο σημείο του κώδικα χρειαζόμαστε νέα μεταβλητή και έχει τη γενική μορφή

τύπος όνομα_μεταβλητής;

Έτσι, δήλωση ακέραιας μεταβλητής με όνομα k γίνεται με την εντολή

```
int k;
```

Μεταβλητή θεμελιώδους τύπου που δηλώνεται όπως παραπάνω, *δεν αποκτά* κάποια συγκεκριμένη τιμή, εκτός αν ορίζεται

- στον καθολικό χώρο ονομάτων (§2.16), δηλαδή έξω από κάθε συνάρτηση (Κεφάλαιο 7), κλάση (Κεφάλαιο 14), απαρίθμηση (§2.6) και άλλο χώρο ονομάτων (§2.16). Τέτοια μεταβλητή σε αυτή την περίπτωση χαρακτηρίζεται ως *καθολική* (global).
- σε άλλο χώρο ονομάτων εκτός του καθολικού.
- ως τοπική στατική μεταβλητή (§7.14).
- ως στατικό μέλος κλάσης.

Οι μεταβλητές αυτών των κατηγοριών χαρακτηρίζονται ως *στατικές* και αποκτούν μια προκαθορισμένη τιμή για κάθε τύπο (είναι η τιμή 0 αφού μετατραπεί στον συγκεκριμένο τύπο σύμφωνα με σχετικούς κανόνες).

Κάποιες φορές χρειάζεται να δηλώσουμε μια μεταβλητή με τον τύπο που έχει μια άλλη ποσότητα ή έκφραση. Μπορούμε να προσδιορίσουμε τον τύπο της ποσότητας ή έκφρασης a με την εντολή `decltype(a)`. Επομένως, στον παρακάτω κώδικα

τύπος μεταβλητή_Α;
`decltype`(μεταβλητή_Α) μεταβλητή_Β;

δημιουργούμε τη μεταβλητή «μεταβλητή_Β» με τον τύπο της ποσότητας «μεταβλητή_Α». Έτσι, οι δηλώσεις δύο ακέραιων μεταβλητών με ονόματα i,j μπορούν να γίνουν με τις εντολές

```
int i;
decltype(i) j;
```

Σε ειδικές περιπτώσεις ο μηχανισμός αυτός είναι πολύ χρήσιμος.

Όλα τα αντικείμενα τύπων που παρέχονται από τη Standard Library (*εκτός* από τον `std::array<>`), ή έχουν δημιουργηθεί από τον προγραμματιστή με προσδιορισμένο default constructor (§14.5.4), αποκτούν κατά τον ορισμό τους, οπουδήποτε γίνει αυτός, μια προκαθορισμένη αρχική τιμή για κάθε τύπο (αν δεν έχουμε προσδιορίσει άλλη κατά τη δήλωση).

Αν επιθυμούμε, μπορούμε να συνδυάσουμε τις δηλώσεις πολλών ποσοτήτων ταυτόχρονα, χωρίζοντας τα ονόματά τους με ‘,’. Προϋπόθεση βέβαια είναι *να είναι του ίδιου τύπου*[1]:

```
τύπος όνομα_μεταβλητής_Α, όνομα_μεταβλητής_Β;
```

Όμως, ένας κώδικας είναι ευκρινέστερος αν η κάθε δήλωση γίνεται σε ξεχωριστή γραμμή καθώς αυτό μας διευκολύνει να παραθέτουμε σχόλια για την ποσότητα που ορίζεται και ελαχιστοποιεί την πιθανότητα λάθος δήλωσης κάποιας ποσότητας.

Συνήθως, όποτε χρειαζόμαστε μια ποσότητα, γνωρίζουμε την τιμή που θέλουμε να έχει αρχικά. Η δυνατότητα που μας δίνει η C++ να γράφουμε τη δήλωση στο σημείο που θα χρειαστούμε για πρώτη φορά μια ποσότητα, μας διευκολύνει να χρησιμοποιούμε όσο περισσότερο γίνεται το μηχανισμό της δήλωσης με ταυτόχρονη απόδοση αρχικής τιμής. Θα τον περιγράψουμε παρακάτω.

2.2.1 Δήλωση με αρχικοποίηση

Όταν γνωρίζουμε την αρχική τιμή που θα έχει μια ποσότητα, μπορούμε να τη δηλώσουμε με ταυτόχρονη απόδοση της τιμής αυτής (αρχικοποίηση). Η γενική μορφή τέτοιας δήλωσης είναι

```
τύπος όνομα_μεταβλητής{αρχική_τιμή};
```

ή, ισοδύναμα,

```
τύπος όνομα_μεταβλητής = {αρχική_τιμή};
```

Εναλλακτικά μπορούμε να χρησιμοποιήσουμε τις μορφές

```
τύπος όνομα_μεταβλητής = αρχική_τιμή;
τύπος όνομα_μεταβλητής(αρχική_τιμή);
```

Η «αρχική_τιμή» δεν είναι απαραιτήτως κάποια σταθερή ποσότητα· μπορεί να είναι άλλη ποσότητα του ίδιου ή διαφορετικού τύπου (αρκεί να προβλέπεται αυτόματη μετατροπή).

Οι δηλώσεις με απόδοση αρχικής τιμής που χρησιμοποιούν τα άγκιστρα γύρω από την τιμή (με ή χωρίς το ‘=’), είναι προτιμότερες καθώς

[1] ή δείκτες ή αναφορές (με αρχικοποίηση) στον ίδιο τύπο.

- μπορούν να επεκταθούν στην περίπτωση αρχικοποίησης ποσότητας που χρειάζεται περισσότερες από μία τιμές για τον προσδιορισμό της αρχικής της τιμής.
- Εξασφαλίζουν ότι δεν γίνεται μετατροπή με απώλεια ακρίβειας της ποσότητας «αρχική_τιμή» κατά την αρχικοποίηση της μεταβλητής[2].

Η δήλωση με αρχικοποίηση χωρίς άγκιστρα επιτρέπει στην «αρχική_τιμή» να μην είναι του ίδιου τύπου με τη μεταβλητή που δηλώνουμε· σε αυτή την περίπτωση, θα υποστεί αυτόματη μετατροπή από το μεταγλωττιστή σε αυτό τον τύπο (κάτι που δεν είναι επιθυμητό όταν συνοδεύεται με απώλεια ακρίβειας), σύμφωνα με κάποιους κανόνες.

Αν παραλείψουμε να προσδιορίσουμε αρχική τιμή μεταξύ των '{}', δηλαδή, δηλώσουμε μια μεταβλητή με την εντολή

```
τύπος όνομα_μεταβλητής{};
```

τότε αυτή αποκτά ως αρχική τιμή την προκαθορισμένη τιμή για τον τύπο της (συνήθως το 0 αφού μετατραπεί). Προσέξτε ότι δεν είναι σωστό να αντικαταστήσουμε την κενή λίστα με κενές παρενθέσεις[3].

Είναι απαραίτητο να διευκρινήσουμε σε αυτό το σημείο ότι κάποιοι σύνθετοι τύποι ποσοτήτων, που θα δούμε σε επόμενα κεφάλαια, υποστηρίζουν αρχικοποίηση με λίστα τιμών που περικλείεται σε άγκιστρα. Αν επιθυμούμε να δημιουργήσουμε ποσότητες τέτοιων τύπων με αντιγραφή από άλλες ποσότητες τέτοιων τύπων, δεν μπορούμε να χρησιμοποιήσουμε τα άγκιστρα για να περιβάλουμε τις αρχικές ποσότητες· πρέπει να χρησιμοποιήσουμε παρενθέσεις.

Αυτόματη αναγνώριση τύπου

Εκτός από τις παραπάνω μορφές δήλωσης με απόδοση αρχικής τιμής, μπορούμε να έχουμε δήλωση στην οποία ο τύπος προσδιορίζεται αυτόματα από την αρχική τιμή:

```
auto όνομα_μεταβλητής = αρχική_τιμή;
```

Με την παραπάνω δήλωση δημιουργούμε και αρχικοποιούμε μια μεταβλητή ίδιου τύπου με την «αρχική_τιμή». Προφανώς, αν κατά τη δήλωση απουσιάζει η αρχική τιμή δεν μπορεί να χρησιμοποιηθεί ο αυτόματος προσδιορισμός τύπου.

Προσέξτε ότι η δήλωση της μορφής

```
auto όνομα_μεταβλητής{αρχική_τιμή};
```

είναι επιτρεπτή αλλά το «όνομα_μεταβλητής» *δεν αποκτά* τον τύπο της ποσότητας «αρχική_τιμή», όπως πιθανότατα επιδιώκουμε. Η μεταβλητή μας δημιουργείται με

[2] Αν πρόκειται να γίνει κάτι τέτοιο ο μεταγλωττιστής μας ενημερώνει με μήνυμα σφάλματος.

[3] Θα δούμε ότι η εντολή τύπος όνομα(); είναι δήλωση συνάρτησης και όχι μεταβλητής.

τύπο `std::initializer_list<>` και αποκτά αρχική τιμή μια λίστα αυτού του τύπου με ένα μέλος (την «αρχική_τιμή»).

Στην περίπτωση που κάνουμε χρήση της αυτόματης αναγνώρισης τύπου κατά τη δήλωση με απόδοση αρχικής τιμής σε πολλές ποσότητες ταυτόχρονα, δηλαδή, γράψουμε κάτι σαν

```
auto i=1, j=5;
```

θα πρέπει οι αρχικές τιμές να είναι *ίδιου τύπου* ώστε ο αυτόματα προσδιοριζόμενος τύπος να είναι κοινός.

Συνοψίζοντας, καλό είναι να προτιμούμε να αρχικοποιούμε κάθε ποσότητα χωριστά, χρησιμοποιώντας τη δήλωση με τα '{}' όταν προσδιορίζουμε ρητά τον τύπο, και να χρησιμοποιούμε το '=' όταν επιδιώκουμε την αυτόματη απόδοση τύπου με το `auto`.

2.3 Κανόνες σχηματισμού ονόματος

Πίνακας 2.1: Προκαθορισμένες λέξεις της C++.

Προκαθορισμένες λέξεις της C++				
`alignas`	`alignof`	`and`	`and_eq`	`asm`
`auto`	`bitand`	`bitor`	`bool`	`break`
`case`	`catch`	`char`	`char16_t`	`char32_t`
`class`	`compl`	`const`	`constexpr`	`const_cast`
`continue`	`decltype`	`default`	`delete`	`do`
`double`	`dynamic_cast`	`else`	`enum`	`explicit`
`extern`	`false`	`float`	`for`	`friend`
`goto`	`if`	`inline`	`int`	`long`
`mutable`	`namespace`	`new`	`noexcept`	`not`
`not_eq`	`nullptr`	`operator`	`or`	`or_eq`
`private`	`protected`	`public`	`register`	`reinterpret_cast`
`return`	`short`	`signed`	`sizeof`	`static`
`static_assert`	`static_cast`	`struct`	`switch`	`template`
`this`	`thread_local`	`throw`	`true`	`try`
`typedef`	`typeid`	`typename`	`union`	`unsigned`
`using`	`virtual`	`void`	`volatile`	`wchar_t`
`while`	`xor`	`xor_eq`		

Τα ονόματα μεταβλητών, σταθερών, συναρτήσεων, χώρων ονομάτων, τύπων που δημιουργεί ο προγραμματιστής (κλάσεις) και των λοιπών δομών της C++ επιτρέπεται να απαρτίζονται από λατινικά γράμματα (a–z, A–Z), αριθμητικά ψηφία (0–9), και το underscore, '_'. Όπως αναφέραμε, *κεφαλαία και πεζά γράμματα είναι διαφορετικά*. Δεν υπάρχει περιορισμός από τη C++ στο μήκος των ονομάτων, ενώ

δεν επιτρέπεται να αρχίζουν από αριθμητικό ψηφίο. Ονόματα που αρχίζουν με underscore (‘_’) ή περιέχουν διπλό underscore (‘__’) είναι δεσμευμένα για χρήση από τον compiler οπότε δεν επιτρέπεται να χρησιμοποιούνται από τον προγραμματιστή. Επίσης, δεν επιτρέπεται η χρήση των προκαθορισμένων λέξεων της C++ (keywords, Πίνακας 2.1) και της δεσμευμένης λέξης export ως ονόματα. Προσέξτε ότι η λέξη main είναι δεσμευμένη για την κύρια συνάρτηση του προγράμματος και δεν επιτρέπεται να χρησιμοποιηθεί για άλλο σκοπό στον καθολικό χώρο ονομάτων (§2.16).

Παράδειγμα

Μη αποδεκτά ονόματα:

`ena␣lathos␣onoma`, άλφα, 1234qwer, new, .onoma.

Αποδεκτά ονόματα:

```
timi, value12, ena_onoma_me_megalo_mikos, sqrt, New
```

2.4 Εντολή εκχώρησης

Η *εντολή εκχώρησης* έχει τη γενική μορφή

μεταβλητή = [έκφραση με σταθερές, μεταβλητές, κλπ.]

Σε αυτή την εντολή εκτελούνται *καταρχάς* όλες οι πράξεις, κλήσεις συναρτήσεων κλπ. που πιθανόν εμφανίζονται στο δεξί μέλος[4]. *Κατόπιν*, το αποτέλεσμα *μετατρέπεται (αν χρειάζεται)* στον τύπο της μεταβλητής του αριστερού μέλους και η τιμή που προκύπτει εκχωρείται σε αυτή.

Είναι δυνατό, στο αριστερό μέλος, η μεταβλητή να προσδιορίζεται μέσω αναφοράς (§2.17), δείκτη (§2.18), ή iterator (Κεφάλαιο 10).

Ο τελεστής εκχώρησης ‘=’ *δεν υποδηλώνει ισότητα* όπως στα μαθηματικά.

Παράδειγμα

```
double b, c;
b = 3.2;
c = 5.5;
c = c + 2.0 * b;
```

Προσέξτε την τελευταία εντολή: δεν σημαίνει ότι `b=0`. Πρώτα εκτελείται το δεξί μέλος, υπολογίζεται η τιμή (`5.5+2.0*3.2`) και το αποτέλεσμα, 11.9, εκχωρείται στη μεταβλητή c, αντικαθιστώντας την παλιά τιμή της.

[4] εκτός αν εμφανίζεται στο δεξί μέλος ο τελεστής ‘,’, ο οποίος είναι ο μόνος που έχει χαμηλότερη προτεραιότητα από τον τελεστή εκχώρησης (Πίνακας 2.3).

Ένα βασικό χαρακτηριστικό της C++ είναι ότι μια εντολή εκχώρησης έχει η ίδια κάποια τιμή που μπορεί να εκχωρηθεί σε κάποια μεταβλητή ή γενικότερα, να χρησιμοποιηθεί. Π.χ.

```
int a, b;
b = a = 3;  //First a = 3; then b = 3;
int c = (b = 4) + a; // First b = 4; then c = 7;
```

Το χαρακτηριστικό αυτό είναι χρήσιμο για την ταυτόχρονη εκχώρηση ίδιας τιμής σε πολλές μεταβλητές αλλά καλό είναι να αποφεύγεται σε άλλες περιπτώσεις καθώς περιπλέκει τον κώδικα.

2.5 Θεμελιώδεις τύποι

Για την αποθήκευση ακέραιων ποσοτήτων η γλώσσα παρέχει τους τύπους **int**, **short int**, **long int**, **long long int**—με τις εμπρόσημες (**signed**) και απρόσημες (**unsigned**) παραλλαγές τους. Για πραγματικά δεδομένα διαθέτει τους **float**, **double**, **long double**. Για τις λογικές ποσότητες παρέχει τον τύπο **bool** ενώ για το χειρισμό χαρακτήρων μπορούμε να επιλέξουμε μεταξύ των **char**, **signed char**, **unsigned char**, **wchar_t**, **char16_t**, **char32_t**. Επιπλέον, ως θεμελιώδης τύπος θεωρείται και ο τύπος **void**, με ειδική σημασία και χρήση.

Δεν θα αναφερθούμε στους τύπους με συγκεκριμένο (ή ελάχιστο) πλήθος bits που παρέχει η γλώσσα και ορίζονται στο `<cstdint>`. Επίσης, θα παραλείψουμε τους τύπους `double_t`, `float_t` του `<cmath>`. Όλοι οι παραπάνω είναι άλλα ονόματα για ενσωματωμένους τύπους της C++[5].

Ο τύπος **bool**, οι τύποι ακεραίων και οι τύποι χαρακτήρα θεωρούνται και συμπεριφέρονται ως *ακέραιοι τύποι (integral types)*· ποσότητες αυτών των τύπων μπορούν να συμμετέχουν μαζί σε εκφράσεις. Η γλώσσα προβλέπει συγκεκριμένους κανόνες μετατροπής μεταξύ αυτών.

2.5.1 Τύποι ακεραίων

Η C++ παρέχει διάφορους τύπους για την αναπαράσταση των ακέραιων ποσοτήτων στον κώδικά μας. Ο βασικός τύπος για ακέραιο είναι ο **int**. Μια μεταβλητή τέτοιου τύπου με όνομα π.χ. i, δηλώνεται ως εξής:

```
int i;
```

Στη C++ υπάρχουν τέσσερα είδη ακεραίων, **short int**, **int**, **long int** και **long long int**, με ελάχιστα μεγέθη τα 16, 16, 32 και 64 bit αντίστοιχα. Επιπλέον, σε μια υλοποίηση, το μέγεθος του **short int** είναι υποχρεωτικά μικρότερο ή ίσο από το μέγεθος του **int**, αυτό με τη σειρά του είναι μικρότερο ή ίσο από το μέγεθος του **long int** και το οποίο είναι μικρότερο ή ίσο από το μέγεθος του

[5] μέσω της εντολής **using** (§2.15).

long long int. Το ακριβές μέγεθος, *σε πολλαπλάσια του μεγέθους του* **char**, ενός τύπου, δίνεται από τον τελεστή **sizeof()** (§2.11.1) με όρισμα τον τύπο.

Καθένας από τους τύπους ακεραίων μπορεί να ορίζεται ως **signed** ή **unsigned** (δηλαδή με πρόσημο ή χωρίς). Αν δεν συμπληρώσουμε τον τύπο με κάποιο από αυτά, θεωρείται ότι δώσαμε **signed**[6].

Οι τιμές που μπορεί να λάβει μια ακέραια μεταβλητή καθορίζονται από την υλοποίηση. Αναφέραμε ότι το μέγεθος του τύπου **int** απαιτείται να είναι τουλάχιστο 16 bit, επομένως μπορεί να αναπαραστήσει αριθμούς στο διάστημα $[-2^{15}, 2^{15}-1] = [-32768, 32767]$ τουλάχιστον. Τα ακριβή όριά του για συγκεκριμένη υλοποίηση προσδιορίζονται από τις επιστρεφόμενες τιμές των συναρτήσεων

```
std::numeric_limits<int>::min()
```

και

```
std::numeric_limits<int>::max()
```

οι οποίες δηλώνονται στο header <limits>. Μπορούμε να τα τυπώσουμε στην οθόνη με τον ακόλουθο κώδικα:

```
#include <limits>
#include <iostream>

int
main()
{
  std::cout << std::numeric_limits<int>::min() << '\n';
  std::cout << std::numeric_limits<int>::max() << '\n';
}
```

Όπως αναφέραμε, μια μεταβλητή τύπου ακεραίου δεν αποκτά κατά τη δήλωσή της κάποια συγκεκριμένη τιμή εκτός και αν είναι στατική, οπότε γίνεται 0.

Αν δεν υπάρχει κάποιος ειδικός λόγος για το αντίθετο, καλό είναι να χρησιμοποιείται ο απλός τύπος **int** για την αναπαράσταση ακεραίων.

Ακέραιες Σταθερές

Μια σειρά αριθμητικών ψηφίων, χωρίς κενά ή άλλα σύμβολα[7], *που δεν αρχίζει από 0*, αποτελεί μια ακέραια σταθερά στο δεκαδικό σύστημα. Π.χ., τρεις ακέραιες σταθερές στο δεκαδικό σύστημα είναι οι παρακάτω

3, 12, 123456

[6] Συντομογραφίες των **short int**, **long int**, **long long int** είναι τα **short**, **long**, **long long** αντίστοιχα, ενώ οι **signed int** και **unsigned int** μπορούν να γραφούν **signed** και **unsigned** αντίστοιχα.

[7] εκτός από την απόστροφο, (').

Αν το πρώτο ψηφίο είναι `0` (και δεν ακολουθείται από 'x' ή 'X') θεωρείται οκταδικός αριθμός και πρέπει να ακολουθείται μόνο από κάποια από τα ψηφία 0–7. Αν οι δύο πρώτοι χαρακτήρες είναι `0x` ή `0X`, ο αριθμός θεωρείται δεκαεξαδικός και μπορεί να περιλαμβάνει εκτός των αριθμητικών ψηφίων, τους χαρακτήρες (a–f) ή (A–F). Αν ο αριθμός αρχίζει με `0b` ή `0B` τότε είναι δυαδικός και επιτρέπεται να περιλαμβάνει τα ψηφία 0 ή 1.

Επιτρέπεται να χωρίζουμε τα ψηφία ενός ακέραιου αριθμού με την απόστροφο, (᾽), ώστε ο αριθμός να είναι ευανάγνωστος (όχι απαραίτητα σε τριάδες): ο ακέραιος 1234567890 μπορεί να γραφεί `1'234'567'890`. Οι απόστροφοι μεταξύ των ψηφίων αγνοούνται από τον μεταγλωττιστή.

Ακέραιος αριθμός που ακολουθείται αμέσως μετά τη σειρά των ψηφίων

- από το χαρακτήρα 'L' ή 'l' θεωρείται τύπου **`long int`** ή, αν δε «χωρά» σε αυτόν, **`long long int`**. Έτσι, ο `12l` είναι **`long int`** ενώ σε κάποια υλοποίηση ο `9223372036854775805L` μπορεί να είναι **`long long int`**.
- από τους χαρακτήρες 'LL' ή 'll' είναι τύπου **`long long int`**[8].
- από το χαρακτήρα 'U' ή 'u' είναι **`unsigned int`** ή ο αμέσως μεγαλύτερος τύπος ακεραίου που επαρκεί (**`unsigned long int`** ή **`unsigned long long int`**).
- από το συνδυασμό 'U' και 'L' (με οποιαδήποτε σειρά, πεζά ή κεφαλαία) είναι **`unsigned long int`** ή **`unsigned long long int`**.
- από το συνδυασμό 'U' και 'LL' (με οποιαδήποτε σειρά, πεζά ή κεφαλαία) είναι **`unsigned long long int`**.

Αν μια ακέραια σταθερά δεν συμπληρώνεται με κάποιο χαρακτήρα που να υποδηλώνει τον τύπο της, ο μεταγλωττιστής θεωρεί για τον τύπο της ότι είναι ο μικρότερος από τους **`int`**, **`long int`**, **`long long int`** που μπορεί να την αναπαραστήσει.

Από τεχνικής άποψης, αρνητικές ακέραιες σταθερές δεν υπάρχουν. Αν μια σταθερά αρχίζει με πρόσημο, '+' ή '-', θεωρείται ότι δρα σε αυτήν ο αντίστοιχος μοναδιαίος τελεστής.

2.5.2 Τύποι πραγματικών

Στη C++ ορίζονται τρεις τύποι πραγματικών αριθμών, ανάλογα με το πόσα ψηφία αποθηκεύονται στον καθένα:

- απλής ακρίβειας (**`float`**),
- διπλής ακρίβειας (**`double`**) και
- εκτεταμένης ακρίβειας (**`long double`**).

[8] δεν επιτρέπονται οι συνδυασμοί 'lL' ή 'Ll'.

Μια πραγματική μεταβλητή διπλής ακρίβειας, με όνομα π.χ. g, δηλώνεται ως εξής:

```
double g;
```

Η γλώσσα εγγυάται ότι το μέγεθος του `float` είναι μικρότερο ή ίσο με το μέγεθος του `double` και αυτό με τη σειρά του είναι μικρότερο ή ίσο από το μέγεθος του `long double`. Οι τιμές που μπορεί να λάβει μια πραγματική μεταβλητή καθορίζονται από την υλοποίηση: το μέγιστο πλήθος των σημαντικών ψηφίων ενός αριθμού που μπορεί να αναπαρασταθεί από ένα πραγματικό τύπο π.χ. `double`, και τα ακριβή όριά του (ελάχιστος/μέγιστος), μπορούν να βρεθούν αν τυπώσουμε τον ακέραιο `std::numeric_limits<double>::digits10` και τις επιστρεφόμενες τιμές των συναρτήσεων

```
std::numeric_limits<double>::min()
std::numeric_limits<double>::max()
```

που δηλώνονται στο `<limits>`. Αντίστοιχα ισχύουν και για τους άλλους τύπους (όπου στα παραπάνω εμφανίζεται `double` γράφουμε άλλο τύπο).[9] Συνήθως, αλλά όχι υποχρεωτικά, ο `float` κρατά έξι σημαντικά ψηφία στο δεκαδικό σύστημα, ο `double` δεκαπέντε και ο `long double` δεκαοκτώ. Αυτή η διαφορά ακρίβειας εξηγεί την ονομασία τους.

Αν δεν υπάρχει κάποιος ειδικός λόγος για το αντίθετο, καλό είναι να χρησιμοποιείται για πραγματικούς αριθμούς ο τύπος `double`, καθώς αντιπροσωπεύει τον βέλτιστο τύπο πραγματικών αριθμών της κάθε υλοποίησης. Ο `double` συνήθως επαρκεί για να αναπαραστήσει με ικανοποιητική ακρίβεια αριθμούς με απόλυτη τιμή από $2^{-1023} \approx 10^{-308}$ έως $2^{1024} \approx 10^{308}$.

Πραγματικές Σταθερές

Η C++ παρέχει πραγματικές σταθερές στο δεκαδικό σύστημα. Μια πραγματική σταθερά έχει δύο δυνατές μορφές. Μπορεί να είναι

- μια σειρά των αριθμητικών ψηφίων 0–9, χωρίς κενά, που *υποχρεωτικά* περιλαμβάνει την τελεία (στη θέση της υποδιαστολής που χρησιμοποιούμε στην αριθμητική). Πριν ή μετά την τελεία μπορεί να μην υπάρχουν ψηφία.

- μια σειρά των αριθμητικών ψηφίων 0–9, χωρίς κενά, που *επιτρέπεται να* περιλαμβάνει την τελεία (στη θέση της υποδιαστολής που χρησιμοποιούμε στην αριθμητική). Πριν ή μετά την πιθανή τελεία μπορεί να μην υπάρχουν ψηφία. Ακολουθείται, χωρίς προηγούμενο κενό, από το χαρακτήρα 'e' ή 'E' και από μια σειρά αριθμητικών ψηφίων η οποία μπορεί να αρχίζει με πρόσημο. Ο ακέραιος μετά το e/E αποτελεί τον εκθέτη του 10, με τη δύναμη του οποίου πολλαπλασιάζεται ο αμέσως προηγούμενος τού e/E αριθμός. Επομένως, ο αριθμός της μορφής

[9] Προσέξτε ότι η εξειδίκευση της `std::numeric_limits<>::min()` για τους τύπους πραγματικών μας επιστρέφει το μικρότερο *θετικό* αριθμό.

xxx.xxxxxxxE$\pm yyy$

έχει αριθμητική τιμή

xxx.xxxxxxx $\times 10^{\pm yyy}$

Οι παραπάνω μορφές επιτρέπεται να ακολουθούνται, χωρίς προηγούμενο κενό, από 'f', 'F', 'l', 'L'. Αν η αριθμητική σταθερά δεν συμπληρώνεται με τους συγκεκριμένους χαρακτήρες, είναι τύπου `double`. Αν ο αριθμός συμπληρώνεται με το 'F' ή 'f', είναι τύπου `float`· αν τελειώνει σε 'L' ή 'l', είναι τύπου `long double`.

Παράδειγμα

- Οι ποσότητες

  ```
  2.034,  .44,  23.
  ```

  είναι πραγματικές τύπου `double`.

- Επίσης πραγματικοί αριθμοί τύπου `double` είναι οι

  ```
  2E-4  (≡ 0.0002),  2.3e2  (≡ 230.0).
  ```

- Οι αριθμοί

  ```
  32.3f,  3E-3F
  ```

  είναι πραγματικοί τύπου `float`.

- Οι αριθμοί

  ```
  1.234L,  0.02E-2L,  7E3L
  ```

  είναι `long double`.

Επιτρέπεται να χωρίζουμε τα ψηφία ενός πραγματικού αριθμού με την απόστροφο, (’), ώστε ο αριθμός να είναι ευανάγνωστος: ο πραγματικός 12.3456789 μπορεί να γραφεί 12.345′678′9. Οι απόστροφοι μεταξύ των ψηφίων αγνοούνται από τον μεταγλωττιστή.

Και για τις πραγματικές σταθερές, τεχνικά, αρνητικές τιμές δεν υπάρχουν. Αν μια σταθερά αρχίζει με πρόσημο, '+' ή '−', θεωρείται ότι δρα σε αυτήν ο αντίστοιχος μοναδιαίος τελεστής.

Προσέξτε ότι ο *ίδιος* αριθμός της αριθμητικής μπορεί να γραφεί στη C++ ως πραγματικός, απλής, διπλής ή εκτεταμένης ακρίβειας[10]. Οι παραπάνω μορφές *είναι διαφορετικές* για τη γλώσσα, παρόλο που αντιστοιχούν στον ίδιο αριθμό.

Τονίζουμε ότι οι πραγματικές σταθερές αποθηκεύονται κρατώντας όσα δεκαδικά ψηφία επιτρέπει ο τύπος τους, *ανεξάρτητα από το πόσα ψηφία θα παραθέσουμε*

[10] είτε, βέβαια, ως μιγαδικός ή και ως ακέραιος, αν τυχαίνει να είναι ακέραιος στα μαθηματικά.

όταν γράφουμε τη σταθερά. Τυπώστε, για παράδειγμα, την τιμή της μεταβλητής που δηλώνεται στην εντολή

```
auto pi = 3.14159265358979323846f;
```

δηλαδή, το π με 21 σημαντικά ψηφία, αλλά γραμμένο σε σταθερά τύπου `float`. Ποια τιμή προκύπτει; Θα χρειαστεί, πριν την εκτύπωση, να δώσετε την εντολή

```
std::cout.precision(21);
```

όπως αναφέρεται στην §6.5, για να ορίσετε 21 ψηφία στην εκτύπωση αντί για τα 6 που είναι προκαθορισμένα.

Με βάση τα παραπάνω, ποια τιμή αναμένετε να πάρει η μεταβλητή στην παρακάτω δήλωση

```
auto a = 3.141592653589793238f6 - 3.1415927f;
```

Τυπώστε τη.

2.5.3 Λογικός τύπος

Μια ποσότητα λογικού τύπου, `bool`, είναι κατάλληλη για την αναπαράσταση μεγεθών που μπορούν να πάρουν *δύο* τιμές (π.χ. ναι/όχι, αληθές/ψευδές, on/off, ...). Η δήλωση μεταβλητής λογικού τύπου, με όνομα π.χ. a, γίνεται ως εξής:

```
bool a;
```

Οι τιμές που μπορεί να πάρει είναι `true` ή `false`. Όπως όλες οι μεταβλητές θεμελιωδών τύπων, η a δεν αποκτά κάποια συγκεκριμένη τιμή με την παραπάνω δήλωση εκτός αν είναι στατική μεταβλητή (§2.2) οπότε αποκτά την τιμή `false`.

Δήλωση με ταυτόχρονη απόδοση της συγκεκριμένης αρχικής τιμής `true` είναι η παρακάτω:

```
bool a{true};
```

ή οι ισοδύναμες μορφές που παρουσιάστηκαν στην §2.2.

Ο τύπος `bool` συμπεριλαμβάνεται στους *ακέραιους τύπους (integral types)*. Ποσότητες τύπου `bool` και τύπων ακεραίου μπορούν να μετατραπούν η μια στον τύπο της άλλης και επομένως, να αναμιχθούν σε αριθμητικές και λογικές εκφράσεις. Όποτε χρειάζεται, μια λογική μεταβλητή με τιμή `true` ισοδυναμεί με 1 ενώ με τιμή `false` ισοδυναμεί με 0. Αντίστροφα, μη μηδενικός ακέραιος μετατρέπεται σε `true` ενώ ακέραιος με τιμή 0 ισοδυναμεί με `false`. Αντίστοιχα ισχύουν για οποιαδήποτε αριθμητική τιμή (π.χ. πραγματική). Επίσης, ένας δείκτης διάφορος του `nullptr`, μετατρέπεται αυτόματα σε `true` ενώ ο `nullptr` ισοδυναμεί με `false`.

Καλό είναι να αποφεύγουμε την ανάμιξη αριθμητικών με λογικές ποσότητες στην ίδια έκφραση. Προβλέπεται, όπως αναφέραμε, η μετατροπή, ως «κληρονομιά» από τη C· τι νόημα όμως έχει π.χ. η πρόσθεση του `true` με ένα αριθμό;

2.5.4 Τύπος χαρακτήρα

Μια μεταβλητή τύπου χαρακτήρα, `char`, με όνομα π.χ. c, δηλώνεται ως εξής:

```
char c;
```

Ας αναφερθεί, χωρίς να υπεισέλθουμε σε λεπτομέρειες, ότι ένας `char` μπορεί να δηλωθεί ότι είναι `signed` ή `unsigned`. Ο απλός τύπος `char` ταυτίζεται είτε με τον `signed char` είτε με τον `unsigned char`, ανάλογα με την υλοποίηση. Όμως, οι τρεις αυτοί τύποι είναι διαφορετικοί.

Οι τιμές που μπορεί να πάρει μια μεταβλητή `char` είναι ένας χαρακτήρας από το σύνολο χαρακτήρων της υλοποίησης· αυτό σχεδόν πάντοτε, αλλά όχι υποχρεωτικά, είναι υπερσύνολο του συνόλου ASCII. Ο τύπος `char` *εξ ορισμού* καταλαμβάνει ένα byte στη μνήμη. Σχεδόν πάντα, αλλά όχι απαραίτητα, αποτελείται από 8 bit. Το ακριβές πλήθος των bits ενός χαρακτήρα σε μια υλοποίηση μπορούμε να το βρούμε στη σταθερή ποσότητα `CHAR_BIT` που ορίζεται στον header `<climits>`.

Όπως όλες οι μεταβλητές θεμελιωδών τύπων, μια μεταβλητή τύπου `char` δεν αποκτά κάποια συγκεκριμένη τιμή με την παραπάνω δήλωση εκτός αν είναι στατική μεταβλητή (§2.2) οπότε αποκτά ως αρχική τιμή το μηδενικό χαρακτήρα. Προσέξτε ότι άλλος χαρακτήρας είναι ο μηδενικός (`'\0'`, ο χαρακτήρας με οκταδική τιμή 0 στο σύνολο ASCII), και άλλος ο `'0'` (ο χαρακτήρας με δεκαδική τιμή 48 στο σύνολο ASCII).

Η δήλωση

```
char c{'a'};
```

ή οι ισοδύναμες μορφές που παρουσιάστηκαν στην §2.2, ορίζει μεταβλητή τύπου χαρακτήρα με όνομα c και με συγκεκριμένη αρχική τιμή, το σταθερό χαρακτήρα `'a'`. Παρατηρήστε ότι ο τελευταίος περικλείεται σε απόστροφους (`'`)· σε διπλά εισαγωγικά (`"`) αποτελεί C-style string[11], που δεν μπορεί να αποδοθεί αυτόματα σε μεταβλητή τύπου `char`.

Ο τύπος `char` συγκαταλέγεται στους *ακέραιους τύπους*. Χαρακτήρες που συμμετέχουν σε εκφράσεις με άλλους ακέραιους τύπους, ισοδυναμούν με την τιμή τους στο σύνολο χαρακτήρων του συστήματος. Έτσι, ο χαρακτήρας `'a'` συμμετέχει σε εκφράσεις με τη δεκαδική τιμή 97 (αν το σύνολο χαρακτήρων του συστήματος περιλαμβάνει το ASCII).

Ειδικοί χαρακτήρες

Κάποιοι από τους χαρακτήρες του συστήματος χρειάζονται ειδικό συμβολισμό για να αναπαρασταθούν. Εισάγονται με '\' και ακολουθούν ένας ή περισσότεροι συγκεκριμένοι χαρακτήρες. Ο συνδυασμός τους αναπαριστά *ένα* χαρακτήρα.

[11] τύπου «δείκτη σε σταθερούς χαρακτήρες», `char const *`.

Πίνακας 2.2: Ειδικοί Χαρακτήρες της C++

Ειδικός Χαρακτήρας	**Περιγραφή**
\'	Απόστροφος
\"	Εισαγωγικά
\?	Ερωτηματικό
\\	Ανάποδη κάθετος
\a	Κουδούνι
\b	Διαγραφή προηγούμενου χαρακτήρα
\f	Αλλαγή σελίδας
\n	Αλλαγή γραμμής
\r	Μετακίνηση στην αρχή της γραμμής
\t	Οριζόντιο tab
\v	Κατακόρυφο tab
\ooo	Χαρακτήρας με οκταδική αναπαράσταση ooo
\xhhh	Χαρακτήρας με δεκαεξαδική αναπαράσταση hhh
\unnnn	Ο χαρακτήρας unicode U+nnnn
\Unnnnnnnn	Ο χαρακτήρας unicode U+nnnnnnnn

Οι ειδικοί χαρακτήρες της C++ παρουσιάζονται στον Πίνακα 2.2, μαζί με τους γενικούς τρόπους προσδιορισμού, σε δεκαεξαδικό και οκταδικό σύστημα, οποιουδήποτε χαρακτήρα του συνόλου της υλοποίησης. Π.χ.

```
char newline{'\n'};
char bell{'\a'};
char Alpha{'\x61'}; // Alpha = 'a' in ASCII, hex
char alpha{'\141'}; // alpha = 'a' in ASCII, octal
```

Οι ειδικοί χαρακτήρες μπορούν βεβαίως να περιλαμβάνονται και σε C-style string, μια σειρά χαρακτήρων εντός διπλών εισαγωγικών. Οι χαρακτήρες '?' και '"' μπορούν να αναπαρασταθούν και χωρίς να εισάγονται με '\', εκτός από ειδικές περιπτώσεις.

Παράδειγμα

Με την εντολή

```
std::cout << "This\nis\ta\ntest\nShe␣\bsaid:␣\"How␣are␣you\?\"\n";
```

εμφανίζεται στην οθόνη

```
This
is      a
test
Shesaid: "How are you?"
```

2.5.5 Εκτεταμένοι τύποι χαρακτήρα

Η C++ παρέχει τον τύπο `wchar_t` και κατάλληλες δομές και συναρτήσεις για την αποθήκευση και χειρισμό όλων των χαρακτήρων που υποστηρίζει μια υλοποίηση στο locale της. Οι υποστηριζόμενοι χαρακτήρες μπορεί να ανήκουν σε σύνολο πολύ μεγαλύτερο (π.χ. Unicode ή ελληνικά) από το βασικό σύνολο χαρακτήρων.

Επίσης, η γλώσσα παρέχει τους τύπους `char16_t` και `char32_t` για το χειρισμό χαρακτήρων που καταλαμβάνουν 16 ή 32 bit αντίστοιχα (π.χ. τα μέλη των συνόλων UTF-16, UTF-32). Δεν θα αναφερθούμε περισσότερο σε αυτούς.

2.5.6 void

Ο θεμελιώδης τύπος `void` χρησιμοποιείται κυρίως ως τύπος του «αποτελέσματος» μιας συνάρτησης για να δηλώσει ότι η συγκεκριμένη συνάρτηση δεν επιστρέφει αποτέλεσμα. Μπορεί επίσης να χρησιμοποιηθεί ως μοναδικό όρισμα μιας συνάρτησης, υποδηλώνοντας με αυτό τον εναλλακτικό τρόπο την κενή λίστα ορισμάτων. Η μόνη άλλη χρήση του είναι στον τύπο `void *` (δείκτης σε `void`) ως δείκτης σε ποσότητα άγνωστου τύπου. Με αυτή τη μορφή χρησιμοποιείται ως τύπος ορίσματος ή επιστρεφόμενης τιμής συνάρτησης.

2.6 Enumeration

Enumeration (απαρίθμηση) είναι ένας τύπος οι επιτρεπτές τιμές του οποίου προσδιορίζονται ρητά από τον προγραμματιστή κατά τη δημιουργία του. Εισάγεται με τις λέξεις `enum class`, ακολουθεί το όνομα του τύπου και μέσα σε άγκιστρα απαριθμώνται οι τιμές που μπορεί να πάρει μια ποσότητα αυτού του τύπου. Π.χ.

```
enum class Color {red, green, blue};
```

Η παραπάνω δήλωση ορίζει ένα νέο τύπο, τον τύπο `Color`, και απαριθμεί τις τιμές που μπορεί να πάρει μια μεταβλητή αυτού του τύπου: red, green, blue. Δήλωση μεταβλητής τέτοιου τύπου με απόδοση αρχικής τιμής είναι η ακόλουθη:

```
Color c{Color::red};
```

Παρατηρήστε τον τρόπο προσδιορισμού της τιμής: το όνομά της, red, έχει συμπληρωθεί με την enum στην οποία ανήκει[12].

Μια απαρίθμηση είναι χρήσιμη για να συγκεντρώνει τις τιμές στα `case` ενός `switch` (§3.6), δίνοντας τη δυνατότητα στον compiler να μας ειδοποιεί αν παραλείψουμε κάποια. Επίσης, είναι χρήσιμη ως τύπος επιστροφής μιας συνάρτησης (§7.2).

Οι «τιμές» μιας απαρίθμησης παίρνουν ακέραια τιμή ανάλογα με τη θέση τους στη λίστα της: η κάθεμια είναι κατά ένα μεγαλύτερη από την προηγούμενή της,

[12] αντίστοιχο μηχανισμό θα συναντήσουμε στο χώρο ονομάτων (§2.16) και στις κλάσεις (Κεφάλαιο 14).

με την πρώτη να παίρνει την τιμή 0. Στο παράδειγμά μας το red είναι 0, το green είναι 1 και το blue είναι 2. Για κάποιες ή όλες από τις «τιμές» μιας απαρίθμησης μπορεί να αντιστοιχηθεί άλλη ακέραια τιμή· σε αυτή την περίπτωση, οι ποσότητες για τις οποίες δεν έχει οριστεί ρητά συγκεκριμένη ακέραια τιμή είναι πάλι κατά 1 μεγαλύτερες από την αμέσως προηγούμενή τους:

```
enum class Color {red, green=5, blue}; // red=0, green=5, blue=6
```

Αν τυχόν χρειάζεται να χρησιμοποιήσουμε τις αριθμητικές τιμές, θα πρέπει να κάνουμε ρητή μετατροπή των ποσοτήτων τύπου απαρίθμησης σε ακέραιο. Έτσι μπορούμε να γράψουμε

```
int k{static_cast<int>(Color::green)};
```

Με τον αμέσως προηγούμενο ορισμό για το Color, το k αποκτά την τιμή 5. Αντίστροφα, μπορούμε να μετατρέψουμε ρητά ένα ακέραιο σε τιμή μιας ποσότητας τύπου **enum class**[13]:

```
Color g{static_cast<Color>(6)}; // g is Color::blue
```

Καλό είναι να αποφεύγουμε να κάνουμε τέτοια μετατροπή. Δεν έχει ιδιαίτερη χρησιμότητα αν ο ακέραιος δεν αντιστοιχεί σε κάποια τιμή της απαρίθμησης.

Δεν θα αναφερθούμε αναλυτικά στις «απλές» enumerations που παρέχει η C++ (είναι αυτές που δηλώνονται χωρίς το **class** στον ορισμό). Σε αυτές, οι μετατροπές από/σε ακέραιο είναι αυτόματες και οι «τιμές» της απαρίθμησης δεν «ανήκουν» στην απαρίθμηση (οπότε το όνομά τους δεν χρειάζεται συμπλήρωση με το όνομα της **enum**). Καλό είναι να μην χρησιμοποιούνται σε νέο κώδικα.

2.7 Σταθερές ποσότητες

Ποσότητες που έχουν γνωστή αρχική τιμή και δεν αλλάζουν σε όλη την εκτέλεση του προγράμματος, είναι καλό να δηλώνονται ως σταθερές ώστε ο compiler να μπορεί να προβεί σε βελτιστοποίηση του κώδικα και ταυτόχρονα, να μπορεί να μας ειδοποιήσει αν κατά λάθος προσπαθήσουμε να μεταβάλουμε στο πρόγραμμα την τιμή ποσότητας που λογικά είναι σταθερή. Η δήλωση ποσότητας που μπορεί να δημιουργηθεί *κατά τη μεταγλώττιση*, γίνεται χρησιμοποιώντας την προκαθορισμένη λέξη **constexpr** και συνοδεύεται υποχρεωτικά με απόδοση της αρχικής (και μόνιμης) τιμής:

```
double constexpr pi{3.141592653589793};
auto   constexpr maximum = 100;  // maximum is int
```

Η αρχική τιμή μιας τέτοιας σταθεράς μπορεί να προκύπτει από οποιαδήποτε έκφραση (με πράξεις, κλήση συνάρτησης **constexpr** κλπ.) αρκεί να μπορεί να υπολογιστεί *κατά τη μεταγλώττιση*. Εναλλακτικά, μια σταθερή ποσότητα μπορούμε να

[13] αρκεί η ακέραια τιμή να μπορεί να αναπαρασταθεί στον ακέραιο τύπο στον οποίο πραγματικά αποθηκεύονται οι τιμές της **enum class**. Ο προκαθορισμένος τύπος αποθήκευσης είναι ο **int**.

τη δηλώσουμε ως `const` οπότε ο περιορισμός χαλαρώνει· η τιμή της μπορεί να υπολογιστεί και *κατά την εκτέλεση* του προγράμματος.

Σε ποσότητες που έχουν δηλωθεί ως `constexpr` ή `const` αυτονόητο είναι ότι δεν μπορεί να γίνει εκχώρηση τιμής (δηλαδή, αλλαγή της αρχικής τιμής).

Καλό είναι να χρησιμοποιούνται συμβολικές σταθερές για να αποφεύγεται η χρήση «μαγικών αριθμών» στον κώδικα. Αν μια ποσότητα που είναι σταθερή (π.χ. πλήθος στοιχείων σε διάνυσμα, φυσικές ή μαθηματικές σταθερές) χρησιμοποιείται με την αριθμητική της τιμή και όχι με συμβολικό όνομα, καθίσταται ιδιαίτερα δύσκολη η αλλαγή της καθώς πρέπει να αναγνωριστεί και να τροποποιηθεί σε όλα τα σημεία του κώδικα που εμφανίζεται.

2.8 Εμβέλεια

Όλες οι μη στατικές μεταβλητές, σταθερές, συναρτήσεις, τύποι μπορούν να χρησιμοποιηθούν από το σημείο της δήλωσής τους[14] έως το καταληκτικό άγκιστρο του block εντολών στο οποίο ανήκουν. Οι μεταβλητές χάνουν την τιμή τους μετά από αυτό το σημείο, ελευθερώνεται ο χώρος μνήμης που καταλαμβάνουν και λέμε ότι *καταστρέφονται*, όταν η ροή της εκτέλεσης φύγει, με οποιονδήποτε τρόπο, από το block εντολών στο οποίο έχουν οριστεί. Αν η ροή επανέλθει στο block (με κάποια εντολή επανάληψης, `goto`, κλήση συνάρτησης κλπ.) πριν το σημείο ορισμού τους, οι μεταβλητές δημιουργούνται ξανά.

Οι καθολικές ποσότητες έχουν εμβέλεια μέχρι το τέλος του αρχείου στο οποίο γίνεται η δήλωση (και σε όσα αρχεία συμπεριλαμβάνεται αυτό με οδηγία `#include`). Καθώς αυτές οι ποσότητες είναι διαθέσιμες σε μεγάλα τμήματα του κώδικα, οι αλλαγές τους είναι δύσκολο να εντοπιστούν. Για το λόγο αυτό, η χρήση τους θα πρέπει να είναι εξαιρετικά σπάνια, μόνο για τις περιπτώσεις που δε γίνεται να την αποφύγουμε.

Οι εντολές ελέγχου και επανάληψης που θα συναντήσουμε στα επόμενα κεφάλαια καθώς και οι συναρτήσεις αποτελούν ξεχωριστά block, με δικές τους εμβέλειες. Επιπλέον, οποιοδήποτε σύνολο εντολών μπορεί να αποτελέσει ξεχωριστό block αν περιληφθεί σε άγκιστρα '{}'. Εννοείται ότι κάθε block πρέπει να περιλαμβάνεται *εξ ολοκλήρου* σε άλλο block (ή, αλλιώς, το κάθε ανοιχτό άγκιστρο '{' ταιριάζει με το *πλησιέστερο επόμενο* κλειστό άγκιστρο '}').

Με βάση τα παραπάνω, δύο ανεξάρτητα block εντολών μπορούν να περιέχουν ποσότητες με το ίδιο όνομα και ίδιο ή διαφορετικό τύπο. Οι ποσότητες αυτές είναι τελείως ανεξάρτητες μεταξύ τους. Εννοείται, βέβαια, ότι στο ίδιο block δεν μπορεί να χρησιμοποιηθεί το ίδιο όνομα για μεταβλητές διαφορετικού τύπου. Προσοχή θέλει η περίπτωση που χρησιμοποιείται το ίδιο όνομα για διαφορετικές ποσότητες σε δύο block που το ένα εσωκλείει το άλλο. Π.χ.

```
#include <iostream>
```

[14] για τις συναρτήσεις και τις καθολικές μεταβλητές, η δήλωση και ο ορισμός δεν ταυτίζονται απαραίτητα.

```
int
main()
{                    // begin block A
  double x{3.2};
  {                  // begin block B
    int x{5};
    std::cout << x; // prints 5
  }                  // end   block B
  std::cout << x;    // prints 3.2
}                    // end   block A
```

Η μεταβλητή x στο εσωτερικό block «κρύβει» σε αυτό τη x του εξωτερικού block· οποιαδήποτε εκχώρηση ή χρήση τιμής του x στο εσωτερικό block αναφέρεται στην ακέραια ποσότητα x. Όταν κλείσει το εσωτερικό block, καταστρέφεται η ακέραια μεταβλητή x και «ξαναφαίνεται» η πραγματική μεταβλητή x. Καλό είναι να αποφεύγεται αυτή η κατάσταση.

2.9 Αριθμητικοί τελεστές

Στη C++ υπάρχουν διάφοροι τελεστές που εκτελούν συγκεκριμένες αριθμητικές πράξεις:

- Οι μοναδιαίοι '+', '−' δρουν σε ένα αριθμό a και μας δίνουν τον ίδιο αριθμό ή τον αντίθετό του αντίστοιχα.
- Οι δυαδικοί '+', '−, '∗' δρουν μεταξύ δύο αριθμών a,b και μας δίνουν το άθροισμα, τη διαφορά και το γινόμενό τους αντίστοιχα.
- Ο δυαδικός '/' μεταξύ *πραγματικών* a,b δίνει το λόγο a/b.
- Ο δυαδικός '/' μεταξύ *ακεραίων* δίνει το *πηλίκο* της διαίρεσης του πρώτου με το δεύτερο, δηλαδή, εκτελεί τη διαίρεση *και αποκόπτει το δεκαδικό μέρος του αποτελέσματος*.
- Ο δυαδικός '%' μεταξύ ακεραίων δίνει το *υπόλοιπο* της διαίρεσης του πρώτου με τον δεύτερο.

Επομένως

```
int a{5};
int b{3};
int p{a/b};  // Piliko: p ← 1
int y{a%b};  // Ypoloipo: y ← 2
```

Προσέξτε ότι δεν υπάρχει τελεστής για ύψωση σε δύναμη. Αντ' αυτού χρησιμοποιείται η συνάρτηση `std::pow()` που περιλαμβάνεται στον header `<cmath>` (Πίνακας 7.1). Όπως θα εξηγήσουμε συνοπτικά στο §7.15, η μαθηματική έκφραση x^a γράφεται στη C++ ως `std::pow(x,a)`, αφού γράψουμε στην αρχή του αρχείου με τον κώδικά μας την οδηγία `#include <cmath>`.

Παρατήρηση: Η πεπερασμένη αναπαράσταση των πραγματικών αριθμών οδηγεί σε σφάλματα στρογγύλευσης στις πράξεις και ορισμένες μαθηματικές ιδιότητές τους (π.χ. η αντιμεταθετική και η προσεταιριστική της πρόσθεσης) δεν ισχύουν. Τι αναμένετε να τυπωθεί με τις επόμενες εντολές; Δοκιμάστε τις.

```
std::cout << 0.1+0.2-0.3 << '\n';
std::cout << 0.1-0.3+0.2 << '\n';
```

Ένας ιδιωματισμός της C++ είναι οι συντμήσεις των παραπάνω τελεστών με το '=': η εντολή

```
a = a + b;
```

γράφεται συνήθως ως

```
a += b;
```

Οι δύο εκφράσεις παραπάνω είναι ισοδύναμες, εκτός από την περίπτωση που ο υπολογισμός της μεταβλητής a παρουσιάζει «παρενέργειες»: με τη χρήση του συντετμημένου τελεστή ο υπολογισμός της ποσότητας του αριστερού μέλους γίνεται μία φορά ενώ χωρίς αυτόν γίνεται δύο. Δεν θα αναφερθούμε περισσότερο σε αυτό το σημείο για τις συνέπειες αυτής της διαφοράς.

Αντίστοιχα ισχύουν και για τους άλλους τελεστές· προκύπτουν οι συντμήσεις '`+=`', '`-=`', '`*=`', '`/=`', '`%=`' *χωρίς κενά μεταξύ του τελεστή και του* `=`.

Άλλος ιδιωματισμός της C++ είναι οι μοναδιαίοι τελεστές '`++`' και '`--`' (χωρίς κενά) οι οποίοι δρουν είτε πριν είτε μετά την αριθμητική μεταβλητή. Αν δρουν πριν, π.χ. όπως στην έκφραση

```
b = ++a + c;
```

τότε αυξάνεται κατά 1 η τιμή του a, αποθηκεύεται σε αυτό η νέα τιμή και μετά χρησιμοποιείται για να υπολογιστεί η έκφραση. Αν δρουν μετά, π.χ. όπως στην έκφραση

```
b = a++ + c;
```

τότε *πρώτα* υπολογίζεται η έκφραση (με την τρέχουσα τιμή του a) και *μετά* αυξάνεται κατά 1 το a. Αντίστοιχα (με μειώσεις κατά 1) ισχύουν για το '`--`'. Άρα το

```
b = --a + c;
```

ισοδυναμεί με

```
a = a - 1;
b = a + c;
```

ενώ το

```
b = a-- + c;
```

ισοδυναμεί με

```
b = a + c;
a = a - 1;
```

Παρατηρήστε ότι οι τελεστές '++' και '--' μετά την αριθμητική μεταβλητή χρειάζονται μια προσωρινή ποσότητα για αποθήκευση κατά την εκτέλεση της αντίστοιχης πράξης τους και, επομένως είναι προτιμότερο, αν δεν υπάρχει λόγος, να γίνεται η αύξηση ή μείωση με τους τελεστές *πριν* την αριθμητική μεταβλητή.

Πίνακας 2.3: Σχετικές προτεραιότητες (κατά φθίνουσα σειρά) κάποιων τελεστών της C++. Τελεστές στην ίδια θέση του πίνακα έχουν ίδια προτεραιότητα.

Σχετικές προτεραιότητες τελεστών της C++	
παρενθέσεις	`()`
τελεστής εμβέλειας	`::`
επιλογή μέλους κλάσης	`.`
επιλογή μέλους δείκτη σε κλάση	`->`
προσπέλαση τιμής σε διάνυσμα	`[]`
κλήση συνάρτησης	`()`
δημιουργία από τιμή	`{}`
μετατροπή τύπου	τύπος(έκφραση)
μετατροπή τύπου	**`static_cast`**
απόρριψη **`const`**	**`const_cast`**
αύξηση, μείωση (μετά τη μεταβλητή)	`++`, `--`
μέγεθος ποσότητας	**`sizeof`**
μέγεθος ποσότητας ή τύπου	**`sizeof`** `()`
αύξηση, μείωση (πριν τη μεταβλητή)	`++`, `--`
bitwise NOT	`~`
λογικό NOT	`!`
μοναδιαίο συν, πλην	`+`, `-`
εξαγωγή διεύθυνσης	`&`
προσπέλαση τιμής δείκτη ή iterator	`*`
μετατροπή τύπου	(τύπος) έκφραση
επιλογή δείκτη μέλους κλάσης	`.*`

(Συνεχίζεται...)

Σχετικές προτεραιότητες τελεστών της C++ (συνέχεια)	
επιλογή δείκτη μέλους δείκτη σε κλάση	`->*`
πολλαπλασιασμός	`*`
διαίρεση (ή πηλίκο)	`/`
υπόλοιπο	`%`
άθροισμα, διαφορά	`+, -`
μετατόπιση bit δεξιά, αριστερά	`>>, <<`
μικρότερο, μεγαλύτερο	`<, >`
μικρότερο ή ίσο, μεγαλύτερο ή ίσο	`<=, >=`
ίσο, άνισο	`==, !=`
bitwise AND	`&`
bitwise XOR	`^`
bitwise OR	`|`
λογικό AND	`&&`
λογικό OR	`||`
τριαδικός τελεστής[15]	`?:`
λίστα αρχικοποίησης	`{}`
εκχώρηση	`=`
πολλαπλασιασμός και εκχώρηση	`*=`
διαίρεση και εκχώρηση	`/=`
υπόλοιπο και εκχώρηση	`%=`
άθροισμα και εκχώρηση	`+=`
διαφορά και εκχώρηση	`-=`
μετατόπιση bit αριστερά με εκχώρηση	`<<=`
μετατόπιση bit δεξιά με εκχώρηση	`>>=`
bitwise AND με εκχώρηση	`&=`
bitwise XOR με εκχώρηση	`^=`
bitwise OR με εκχώρηση	`|=`
τελεστής κόμμα	`,`

Στον Πίνακα 2.3 παρατίθενται οι σχετικές προτεραιότητες κάποιων τελεστών. Για τελεστές ίδιας προτεραιότητας, οι πράξεις εκτελούνται από αριστερά προς τα

[15] Για την προτεραιότητα του '`?:`' ως προς τους τελεστές εκχώρησης, άλλο τριαδικό τελεστή ή τον τελεστή λίστας, δείτε την §3.5.

δεξιά. Εξαίρεση αποτελούν οι τελεστές εκχώρησης (απλός και σύνθετοι) και οι μοναδιαίοι· σε αυτούς μεγαλύτερη προτεραιότητα έχει ο δεξιότερος όμοιος τελεστής. Σημειώστε ότι συνεχόμενα σύμβολα (χωρίς κενά) ομαδοποιούνται από αριστερά προς τα δεξιά από τον compiler ώστε να σχηματιστεί ο μακρύτερος σύνθετος τελεστής, και δεν αντιμετωπίζονται χωριστά. Π.χ. η έκφραση `a--b` θεωρείται ως `(a--)b` (και είναι λάθος) παρά ως `a-(-b)`. Οι παρενθέσεις έχουν την υψηλότερη προτεραιότητα και με τη χρήση τους επιβάλλουμε διαφορετική σειρά εκτέλεσης των πράξεων.

2.10 Κανόνες μετατροπής

Σε εκφράσεις που συμμετέχουν ποσότητες διαφορετικών τύπων γίνονται αυτόματα από τον compiler οι κατάλληλες μετατροπές (αν είναι εφικτές, αλλιώς στη μεταγλώττιση βγαίνει λάθος) *ώστε να γίνουν όλες ίδιου τύπου και συγχρόνως να μη χάνεται η ακρίβεια*. Έτσι π.χ. σε πράξη μεταξύ **`int`** και **`double`** γίνεται μετατροπή της τιμής του **`int`** στον αντίστοιχο **`double`** και μετά εκτελείται η κατάλληλη πράξη μεταξύ πραγματικών ποσοτήτων. Οι πραγματικοί αριθμοί που τυχόν συμμετέχουν σε έκφραση με μιγαδικούς, μετατρέπονται στους αντίστοιχους μιγαδικούς κλπ. Ας σημειωθεί ότι ποσότητες ακέραιου τύπου «μικρότερου» από **`int`** (όπως **`bool`**, **`char`**, **`short int`**) μετατρέπονται σε **`int`** και κατόπιν εκτελείται η πράξη, *ακόμα και όταν είναι ίδιες* δεξιά και αριστερά του τελεστή. Το αποτέλεσμα της πράξης είναι **`int`**. Για να είναι κατανοητός ο κώδικας είναι καλό να αποφεύγονται «αφύσικες» εκφράσεις παρόλο που η γλώσσα προβλέπει κανόνες μετατροπής: γιατί π.χ. να χρειάζεται να προσθέσω **`bool`** και **`char`**;

Οι μετατροπές από ένα θεμελιώδη τύπο σε άλλον, «μεγαλύτερο» (με την έννοια ότι επαρκεί για να αναπαραστήσει την αρχική τιμή) δεν κρύβουν ιδιαίτερες εκπλήξεις. Προσοχή χρειάζεται όταν γίνεται μετατροπή σε «μικρότερο» τύπο, π.χ. στην εκχώρηση ενός πραγματικού αριθμού σε ακέραιο ή κατά τη δήλωση ακεραίου με απόδοση πραγματικής αρχικής τιμής μέσω του τελεστή '='. Σε τέτοια περίπτωση γίνεται στρογγυλοποίηση του πραγματικού σε ακέραιο με αποκοπή του δεκαδικού μέρους και κατόπιν γίνεται η εκχώρηση. Επιπλέον, είναι δυνατόν η στρογγυλοποιημένη τιμή να μην είναι μέσα στα όρια τιμών της μεταβλητής του αριστερού μέλους οπότε η συμπεριφορά του προγράμματος (και όχι μόνο το αποτέλεσμα) είναι απροσδιόριστη[16]. Έτσι

```
int a = 3.14; // a is 3
short int b = 12121212121.3;  // b = ??
```

2.10.1 Ρητή μετατροπή

Υπάρχουν περιπτώσεις που ο προγραμματιστής θέλει να καθορίζει συγκεκριμένη μετατροπή. Τέτοια είναι η περίπτωση της διαίρεσης ακεραίων. Όπως ανα-

[16] Ένας καλός compiler αναγνωρίζει τέτοια περίπτωση και προειδοποιεί.

φέρθηκε, δεν υπάρχει τελεστής που να εκτελεί αυτή την πράξη και να υπολογίζει πραγματικό αποτέλεσμα. Θυμηθείτε ότι ο τελεστής '/' εκτελεί κάτι διαφορετικό μεταξύ πραγματικών αριθμών (διαίρεση) απ' ό,τι μεταξύ ακεραίων (πηλίκο). Στον παρακάτω κώδικα που υπολογίζει την (πραγματική) μέση τιμή κάποιων ακεραίων είναι απαραίτητο να προσδιοριστεί συγκεκριμένη δράση του '/'. Αυτό επιτυγχάνεται με τη ρητή μετατροπή τουλάχιστον ενός[17] ακέραιου ορίσματός του σε πραγματικό με την εντολή `static_cast<>`:

```
int sum{2 + 3 + 5};
int N{3};
double mean1{sum / N}; // Wrong value
double mean2{static_cast<double>(sum) / N}; // Correct value
```

Προσέξτε ότι η μετατροπή αφορά *την τιμή* που έχει η ποσότητα στο όρισμα. Ο *τύπος* της δεν αλλάζει.

Μια άλλη περίπτωση που χρειάζεται ρητή μετατροπή σε συγκεκριμένο τύπο εμφανίζεται κατά την κλήση overloaded συνάρτησης (§7.10) όταν η επιλογή της κατάλληλης υλοποίησης δεν είναι μονοσήμαντη.

Η σύνταξη της εντολής μετατροπής, του `static_cast<>`, είναι:

`static_cast`<νέος_τύπος>(έκφραση);

Από τη C έχει κληρονομηθεί η δυνατότητα μετατροπής με τη σύνταξη

(νέος_τύπος) έκφραση

Επίσης, μετατροπή μπορεί να γίνει και ως εξής

νέος_τύπος(έκφραση)

Οι δύο τελευταίες μορφές μετατροπής χρησιμοποιούνται σε παλαιότερους κώδικες. Αποφύγετε τη χρήση τους· προτιμήστε το `static_cast<>`.

Εναλλακτικά, αντί για τη ρητή μετατροπή μέσω του `static_cast<>` μπορούμε να εκμεταλλευτούμε τους αυτόματους κανόνες μετατροπής και να γράψουμε κάτι σαν

```
double mean2{1.0 * sum / N};
```

ή

```
double mean2{(sum + 0.0) / N};
```

Οι τιμές της μεταβλητής mean2 θα είναι τότε οι επιθυμητές (γιατί;).

[17] Οι κανόνες αυτόματης μετατροπής φροντίζουν για τη μετατροπή και του άλλου.

2.11 Άλλοι τελεστές

2.11.1 Τελεστής sizeof

Ο τελεστής **sizeof** δέχεται ως όρισμα μια ποσότητα ή έναν τύπο και επιστρέφει το μέγεθός τους σε bytes[18]. Στο παρακάτω παράδειγμα δίνονται οι τρόποι κλήσης του τελεστή **sizeof**:

```
int a;
std::cout << sizeof(int); //parentheses are necessary
std::cout << sizeof(a);
std::cout << sizeof a;
```

Προσέξτε ότι ο τελεστής ακολουθείται από το όνομα του τύπου ή της ποσότητας σε παρενθέσεις. Οι παρενθέσεις μπορούν να παραλείπονται αν το όρισμα είναι το όνομα ποσότητας (ή αναφορά ή δείκτης σε ποσότητα).

Ο τελεστής **sizeof** υπολογίζεται κατά τη μεταγλώττιση και το αποτέλεσμά του θεωρείται σταθερή ποσότητα· μπορεί, επομένως, να χρησιμοποιείται όπου χρειάζεται τέτοια.

Ο επιστρεφόμενος τύπος από το **sizeof** είναι ο `std::size_t` που ορίζεται στο `<cstddef>`. Είναι άλλο όνομα (ορισμένο μέσω του **using**) για ένα απρόσημο ακέραιο τύπο. Ποσότητες τέτοιου τύπου είναι ιδιαίτερα κατάλληλες για διαστάσεις πινάκων καθώς και ως δείκτες αρίθμησης για να προσπελάζουμε τα στοιχεία τους.

2.11.2 Τελεστές bit

Υπάρχουν τελεστές που αντιμετωπίζουν τα ορίσματά τους ως σύνολο bit σε σειρά, δηλαδή ως ακολουθίες από τα ψηφία 0 ή 1. Η δράση τους ελέγχει ή θέτει την τιμή του κάθε bit χωριστά. Τα ορίσματά τους είναι ποσότητες με ακέραιο τύπο ή **enum class**. Οι τελεστές παρουσιάζονται στον Πίνακα 2.4.

Ο τελεστής '~' δρα σε ένα ακέραιο και επιστρέφει νέο ακέραιο έχοντας μετατρέψει τα bit του αρχικού με τιμή 0 σε 1 και αντίστροφα. Εναλλακτικό όνομα του τελεστή είναι το **compl**.

Οι τελεστές '<<', '>>', δημιουργούν νέα τιμή με μετατοπισμένα προς τα αριστερά ή τα δεξιά, αντίστοιχα, τα bit του αριστερού τους ορίσματος κατά τόσες θέσεις όσες ορίζει το δεξί τους όρισμα. Τα επιπλέον bit χάνονται. Ο τελεστής '<<' συμπληρώνει τις κενές θέσεις με 0 ενώ ο '>>' κάνει το ίδιο αν το αριστερό όρισμα είναι **unsigned**. Οι συνδυασμοί τους με το '=' εκτελούν τη μετατόπιση και εκχωρούν το αποτέλεσμα στο αριστερό τους όρισμα.

Οι τελεστές '&', '^', '|' επιστρέφουν ακέραιο με bit pattern που προκύπτει αν εκτελεστεί το AND, XOR, OR αντίστοιχα στα ζεύγη bit των ορισμάτων τους. Ο πίνακας αλήθειας τους για όλες τις δυνατές τιμές δύο bit p και q, που συνδυάζονται με καθένα από αυτούς τους τελεστές, είναι ο Πίνακας 2.5. Εναλλακτικά

[18] Εξ ορισμού, το byte είναι το μέγεθος ενός **char**.

Πίνακας 2.4: Τελεστές bit της C++

Τελεστής	Όνομα	Χρήση
`~`	bitwise NOT	`~expr`
`<<`	μετατόπιση αριστερά	`expr1 << expr2`
`>>`	μετατόπιση δεξιά	`expr1 >> expr2`
`&`	bitwise AND	`expr1 & expr2`
`^`	bitwise XOR	`expr1 ^ expr2`
`|`	bitwise OR	`expr1 | expr2`
`<<=`	μετατόπιση αριστερά με εκχώρηση	`expr1 <<= expr2`
`>>=`	μετατόπιση δεξιά με εκχώρηση	`expr1 >>= expr2`
`&=`	bitwise AND με εκχώρηση	`expr1 &= expr2`
`^=`	bitwise XOR με εκχώρηση	`expr1 ^= expr2`
`|=`	bitwise OR με εκχώρηση	`expr1 |= expr2`

Πίνακας 2.5: Πίνακας αλήθειας των δυαδικών τελεστών AND, XOR, OR

p	q	AND	XOR	OR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

ονόματα των παραπάνω τελεστών είναι τα **`bitand`**, **`xor`**, **`bitor`** αντίστοιχα. Οι συνδυασμοί τους με το '=', '`&=`', '`^=`', '`|=`', εκτελούν τη μετατροπή των bit απευθείας στο αριστερό τους όρισμα. Εναλλακτικά ονόματα των παραπάνω τελεστών είναι τα **`and_eq`**, **`xor_eq`**, **`or_eq`** αντίστοιχα.

Η αποθήκευση bit σε ακέραιους και η χρήση τελεστών για το χειρισμό τους είναι σημαντική όταν θέλουμε να καταγράψουμε την κατάσταση (true/false, on/off, ...) ενός πλήθους αντικειμένων. Η C++ έχει εισαγάγει την κλάση `std::bitset<>` και την εξειδίκευση της κλάσης `std::vector<>` για **`bool`**, `std::vector<bool>`[19], που διευκολύνουν πολύ αυτό το σκοπό, και βέβαια είναι προτιμότερες.

Η προφανής εναλλακτική λύση ενός διανύσματος με ποσότητες τύπου **`bool`**, παρόλο που είναι πιο εύχρηστη, κάνει πολύ μεγάλη σπατάλη μνήμης καθώς για την αποθήκευση ενός bit δεσμεύει τουλάχιστον ένα byte (§2.5.1).

2.11.3 Τελεστής κόμμα ','

Δύο ή περισσότερες εκφράσεις μπορούν να διαχωρίζονται με τον τελεστή ','. Ο υπολογισμός ή γενικότερα, η εκτέλεσή τους γίνεται από αριστερά προς τα δεξιά και η τιμή της συνολικής έκφρασης είναι η τιμή της δεξιότερης.

[19] Δείτε την παρατήρηση στο §11.5.2, σελίδα 236.

Άσκηση
Πώς εκτελείται ο παρακάτω κώδικας[20]; Τι αναμένετε να τυπωθεί;

```
auto a = -1;
auto b =  1;
std::cout <<  a, b  << '\n';
std::cout << (a, b) << '\n';
```

2.12 Μαθηματικές συναρτήσεις της C++

Η C++ παρέχει μέσω της Standard Library ορισμένες μαθηματικές συναρτήσεις, χρήσιμες για συνήθεις υπολογισμούς σε επιστημονικούς κώδικες. Όπως αναφέραμε, ακόμα και η ύψωση σε δύναμη γίνεται με συνάρτηση. Δείτε την §7.15 για την πλήρη καταγραφή τους.

Στην παρούσα παράγραφο θα εξηγήσουμε συνοπτικά τα βασικά για να τις χρησιμοποιήσουμε: τι σημαίνει η δήλωση π.χ. `double sqrt(double x)` για την τετραγωνική ρίζα, καθώς και τον τρόπο χρήσης, την *κλήση* της συνάρτησης. Στη δήλωση, εντός των παρενθέσεων, προσδιορίζεται ο τύπος της ποσότητας στην οποία θέλουμε να δράσει η συνάρτηση. Πριν το όνομά της εμφανίζεται ο τύπος του αποτελέσματος.

Έστω, λοιπόν, ότι σε κάποιο σημείο του κώδικα έχουμε την πραγματική μεταβλητή x και θέλουμε να υπολογίσουμε την τετραγωνική ρίζα της και να την αποθηκεύσουμε στην πραγματική μεταβλητή y. Για να χρησιμοποιήσουμε την `std::sqrt()` πρέπει καταρχάς να συμπεριλάβουμε το header στον οποίο ανήκει, τον `<cmath>`, με κατάλληλη οδηγία προς τον προεπεξεργαστή. Κατόπιν, όταν χρειαζόμαστε τον υπολογισμό, δίνουμε την εντολή `y=std::sqrt(x);`. Ένα ολοκληρωμένο πρόγραμμα που διαβάζει ένα πραγματικό αριθμό και τυπώνει τη ρίζα του είναι το ακόλουθο:

```
#include <iostream>  // for std::cin, std::cout
#include <cmath>     // for std::sqrt

int
main()
{
  double x;
  std::cin >> x;
  auto y = std::sqrt(x);
  std::cout << y << '\n';
}
```

Περισσότερα θα αναφέρουμε στο Κεφάλαιο 7.

[20] Συμβουλευτείτε τους Πίνακες 2.3 και 2.4. Λάβετε υπόψη ότι ο ειδικός χαρακτήρας `'\n'` έχει δεκαδική τιμή 10.

2.12.1 Γεννήτρια τυχαίων αριθμών

Από τη C έχει κληρονομηθεί η συνάρτηση `std::rand()` του header `<cstdlib>`, για την παραγωγή τυχαίων αριθμών (χωρίς ιδιαίτερες απαιτήσεις «ποιότητας»). Η `std::rand()`, καλούμενη χωρίς όρισμα, επιστρέφει ένα ψευδοτυχαίο *ακέραιο* αριθμό από 0 έως και `RAND_MAX`. Η σταθερή ποσότητα `RAND_MAX` προσδιορίζεται στο `<cstdlib>` και, φυσικά, μπορούμε μόνο να «διαβάσουμε» την τιμή της και όχι να την αλλάξουμε.

Η χρήση της είναι όπως στον παρακάτω κώδικα

```
#include <cstdlib>
int
main()
{
  auto r = std::rand(); // r is random in [0,RAND_MAX]
  ... // use r
}
```

Κάθε φορά που καλείται η `std::rand()`, επιστρέφει άλλο τυχαίο αριθμό. Αν εκτελέσουμε ξανά το ίδιο πρόγραμμα, η ακολουθία των τυχαίων αριθμών *θα είναι η ίδια*.

Μπορούμε να επηρεάσουμε την ακολουθία τυχαίων που παράγεται από τη `std::rand()` αν καλέσουμε τη συνάρτηση `std::srand()` του `<cstdlib>` με όρισμα ένα **`unsigned int`**. Ανάλογα με την τιμή του ορίσματος αλλάζει και η ακολουθία των τυχαίων αριθμών. Αν επιθυμούμε να έχουμε άλλη ακολουθία αριθμών σε κάθε εκτέλεση του προγράμματός μας, πρέπει να δίνουμε ως όρισμά της άλλο ακέραιο κάθε φορά. Μπορούμε να αξιοποιήσουμε για αυτό το σκοπό την τιμή που επιστρέφεται από τη κλήση της `std::time()` του header `<ctime>` με όρισμα το **`nullptr`**. Η συγκεκριμένη συνάρτηση επιστρέφει την τρέχουσα ώρα σε ποσότητα αριθμητικού τύπου. Οι υλοποιήσεις της C++ συνήθως (αλλά όχι υποχρεωτικά) επιλέγουν για την επιστρεφόμενη τιμή της `std::time()` τον αριθμό των δευτερολέπτων που έχουν περάσει μέχρι την κλήση της από την ώρα 00:00 της 1ης Ιανουαρίου 1970. Επομένως, πριν αρχίσουμε τις κλήσεις της `std::rand()` για την παραγωγή τυχαίων αριθμών, μπορούμε να έχουμε την εντολή

```
std::srand(std::time(nullptr));
```

Αν επιθυμούμε να αλλάξουμε το πεδίο τιμών της τυχαίας μεταβλητής r και να έχουμε τυχαίο ακέραιο x στο διάστημα $[m, n]$ (με ακέραια όρια m, n), μπορούμε να ακολουθήσουμε δύο δυνατότητες:

- Να κάνουμε τη μετατροπή x$= \alpha r + \beta$ και να επιλέξουμε τους συντελεστές α, β ώστε η ποσότητα x να βρίσκεται εντός των επιθυμητών ορίων. Μπορείτε να επαληθεύσετε ότι ο x που υπολογίζεται από την εντολή

  ```
  int x{m + std::round(static_cast<double>(n-m)/RAND_MAX*r)};
  ```

ανήκει στο διάστημα $[m, n]$.

- Να χρησιμοποιήσουμε την πιο απλή εντολή

```
int x{m + r % (n-m+1)};
```

Και οι δύο επιλογές επηρεάζουν αρνητικά την «ποιότητα» των ψευδοτυχαίων.

Παρατηρήστε ότι αν r είναι η τιμή που επιστρέφει η `std::rand()`, ο αριθμός `static_cast<double>(r) / RAND_MAX` είναι πραγματικός στο διάστημα $[0, 1]$. Επομένως, αν επιθυμούμε να παράγουμε τυχαίους *πραγματικούς* αριθμούς στο διάστημα $[a, b]$ μπορούμε να χρησιμοποιήσουμε την εντολή

```
double x{a + static_cast<double>(std::rand()) / RAND_MAX * (b-a)};
```

Η C++ παρέχει στο header `<random>` εκτεταμένη συλλογή συναρτήσεων και κλάσεων για την παραγωγή τυχαίων αριθμών με συγκεκριμένα χαρακτηριστικά. Δεν θα αναφερθούμε περισσότερο σε αυτές. Όταν όμως χρειαζόμαστε ποιοτικούς τυχαίους αριθμούς, πρέπει να τις χρησιμοποιήσουμε.

2.13 Μιγαδικός τύπος

Η χρήση μιγαδικών αριθμών σε κώδικα C++ προϋποθέτει τη συμπερίληψη του header `<complex>` με την οδηγία

```
#include <complex>
```

προς τον προεπεξεργαστή.

Μετά τη συμπερίληψη του `<complex>`, ένας αριθμός που ακολουθείται από το i αποτελεί ένα φανταστικό αριθμό διπλής ακρίβειας (δηλαδή μιγαδικό αριθμό με πραγματικό μέρος 0). Αν ακολουθείται από το if ή το il συμβολίζει φανταστικό αριθμό απλής ή εκτεταμένης ακρίβειας.

2.13.1 Δήλωση

Δήλωση μεταβλητής (με όνομα π.χ. z) μιγαδικού τύπου, με πραγματικό και φανταστικό μέρος τύπου **double**, γίνεται με την ακόλουθη εντολή:

```
std::complex<double> z;    // z = 0.0 + 0.0 i
```

Στη δήλωση, αντί για **double** μπορούμε να χρησιμοποιήσουμε οποιοδήποτε άλλο αριθμητικό τύπο (**int**, **float**, **long double**,...). Αν δεν προσδιοριστεί αρχική τιμή, δίνεται αυτόματα στη μεταβλητή η προκαθορισμένη τιμή $0 + 0\mathrm{i}$, *ανεξάρτητα από το αν αυτή είναι στατική ή τοπική.* Όπως αναφέραμε, αυτόματη αρχικοποίηση συμβαίνει σε όλα τα αντικείμενα τύπων που ορίζονται στη Standard Library (εκτός από το `std::array<>`).

Δήλωση με ταυτόχρονη απόδοση συγκεκριμένης αρχικής τιμής γίνεται με έναν από τους παρακάτω τρόπους:

- προσδιορίζουμε το πραγματικό και το φανταστικό μέρος της αρχικής τιμής σε άγκιστρα, '{}',

```
std::complex<double> z{3.4,2.8}; // z = 3.4 + 2.8i
```

ή ισοδύναμα, μέσα σε παρενθέσεις, '()',

```
std::complex<double> z(3.4,2.8);
```

- προσδιορίζουμε μόνο το πραγματικό μέρος της αρχικής τιμής. Το φανταστικό θεωρείται αυτόματα 0. Π.χ.

```
std::complex<double> z{3.41};  // z = 3.41 + 0.0i
```

ή ισοδύναμα

```
std::complex<double> z(3.41);
```

ή ακόμα και

```
std::complex<double> z = 3.41;
```

- Προσδιορίζουμε ως αρχική τιμή ένα φανταστικό αριθμό. Το πραγματικό μέρος της δηλούμενης ποσότητας γίνεται αυτόματα 0. Π.χ.

```
std::complex<double> z{1.2i};  // z = 0.0 + 1.2i
```

- προσδιορίζουμε μια άλλη μιγαδική ποσότητα την οποία αντιγράφουμε στη νέα μεταβλητή[21]. Π.χ.

```
std::complex<double> z1{3.41}; // z1 = 3.41 + 0.0i
std::complex<double> z2{z1};   // z2 = z1;
auto z3 = z2;                  // z3 = z2;
```

Στις παραπάνω δηλώσεις, οι αριθμοί ή η μιγαδική ποσότητα που προσδιορίζουμε μπορούν να προκύπτουν από οποιαδήποτε έκφραση παράγει πραγματικό ή μιγαδικό αριθμό ή ποσότητα που μπορεί να μετατραπεί σε τέτοιο (π.χ. αριθμητικές πράξεις, κλήσεις συναρτήσεων που επιστρέφουν αριθμό, κλπ.).

2.13.2 Πράξεις και συναρτήσεις μιγαδικών

Οι αριθμητικοί τελεστές '+', '−', '*', '/' και οι συντμήσεις '+=', '−=', '*=', '/=', που περιγράψαμε στην §2.9, μεταξύ μιγαδικών και ακέραιων ή πραγματικών, εκτελούν τις αναμενόμενες και γνωστές πράξεις από τα μαθηματικά και έχουν μιγαδικό αποτέλεσμα. Επίσης, οι μαθηματικές συναρτήσεις της C++ (§7.1) που έχουν νόημα για μιγαδικούς αριθμούς, δέχονται μιγαδικά ορίσματα[22], επιστρέφοντας το αντίστοιχο

[21] ο compiler καλεί τον copy constructor της κλάσης `std::complex<double>` (§14.5.2).
[22] αφού συμπεριλάβουμε το header `<complex>`.

μιγαδικό αποτέλεσμα (εκτός από τη συνάρτηση `std::abs()` όπως θα αναφέρουμε αμέσως παρακάτω).

Παρακάτω παρουσιάζονται οι συναρτήσεις για μιγαδικές ποσότητες που περιέχονται στο `<complex>`:

- η συνάρτηση `std::abs()` με μιγαδικό όρισμα επιστρέφει το (πραγματικό) μέτρο (magnitude) του ορίσματος. Αν `z` $= \alpha + \mathrm{i}\beta$ τότε

  `std::abs(z) =` $\sqrt{\texttt{zz}^\star} \to \sqrt{\alpha^2 + \beta^2}$.

- Η συνάρτηση `std::polar()` επιστρέφει μιγαδικό αριθμό με μέτρο (magnitude) το πρώτο όρισμα και φάση (phase angle) (σε rad) το δεύτερο. Αν το δεύτερο όρισμα δεν προσδιορίζεται, θεωρείται 0. Επομένως, η `std::polar(r,t)` επιστρέφει το μιγαδικό αριθμό `z` $= r\mathrm{e}^{\mathrm{i}t}$ ενώ η `std::polar(r)` επιστρέφει τον `z` $= r\mathrm{e}^{\mathrm{i}0} \equiv r(1 + 0\mathrm{i}) = r$. Με τη χρήση αυτής της συνάρτησης μπορούμε να κατασκευάσουμε μιγαδικούς με συγκεκριμένο μέτρο και φάση:

  ```
  std::complex<double> z1{std::polar(2.0,0.75)}; // z1 = 2 exp(0.75i)
  auto z2 = std::polar(2.0); // z2 = 2 exp(0i) = 2.0 + 0.0i
  ```

- Η συνάρτηση `std::norm()` υπολογίζει το τετράγωνο του μέτρου του μιγαδικού ορίσματός της. Αν `z` $= \alpha + \mathrm{i}\beta$ τότε

  `std::norm(z) = zz`$^\star \to \alpha^2 + \beta^2$.

- Η συνάρτηση `std::arg()` επιστρέφει τη φάση του μιγαδικού ορίσματός της. Αν `z` $= \alpha + \mathrm{i}\beta$ τότε

  `std::arg(z)` $\to \tan^{-1}(\beta/\alpha)$.

- Η συνάρτηση `std::conj()` επιστρέφει το συζυγή του μιγαδικού ορίσματός της. Αν `z` $= \alpha + \mathrm{i}\beta$ τότε

  `std::conj(z)` $\to \alpha - \mathrm{i}\beta$.

- Η συνάρτηση `std::proj()` επιστρέφει την προβολή του μιγαδικού ορίσματός της στη σφαίρα Riemann.

- Η συνάρτηση `std::real()` επιστρέφει το πραγματικό μέρος του μιγαδικού ορίσματός της. Αν `z` $= \alpha + \mathrm{i}\beta$ τότε

  `std::real(z)` $\to \alpha$.

  Το πραγματικό μέρος μιας μιγαδικής ποσότητας επιστρέφεται επίσης από τη συνάρτηση–μέλος `real()` της κλάσης `std::complex<>`. Η κλήση της για μια μιγαδική ποσότητα z είναι `z.real()`.

- Η συνάρτηση `std::imag()` επιστρέφει το φανταστικό μέρος του μιγαδικού ορίσματός της. Αν `z` $= \alpha + \mathrm{i}\beta$ τότε

`std::imag(z)` $\rightarrow \beta$.

Το φανταστικό μέρος μιας μιγαδικής ποσότητας επιστρέφεται επίσης από τη συνάρτηση–μέλος `imag()` της κλάσης `std::complex<>`. Η κλήση της για μια μιγαδική ποσότητα z είναι `z.imag()`.

Για να αποδώσουμε νέα τιμή στο πραγματικό ή φανταστικό μέρος μιας μιγαδικής μεταβλητής, μπορούμε να χρησιμοποιήσουμε τις συναρτήσεις–μέλη `real()` και `imag()` με ένα όρισμα, τη νέα τιμή:

```
std::complex<double> z{3.0,1.5}; // z = 3.0 + 1.5i
z.real(4.2);  // z = 4.2 + 1.5i
z.imag(-0.9); // z = 4.2 - 0.9i
```

2.13.3 Είσοδος–έξοδος μιγαδικών δεδομένων

Η εκτύπωση μιγαδικών δεδομένων στην έξοδο (δηλαδή στην οθόνη ή σε αρχείο) με τον τελεστή '<<' γίνεται με τη μορφή

(πραγματικό μέρος,φανταστικό μέρος)

Επομένως, με τις εντολές

```
std::complex<double> z{3.2,1.5};
std::cout << z;
```

θα εμφανιστεί στην οθόνη

```
(3.2,1.5)
```

Η ανάγνωση μιγαδικών δεδομένων από την είσοδο (δηλαδή το πληκτρολόγιο ή αρχείο) με τον τελεστή '>>' γίνεται με μια από τις παρακάτω διαμορφώσεις:

(πραγματικό μέρος,φανταστικό μέρος)
(πραγματικό μέρος)
πραγματικό μέρος

Επομένως, με τις εντολές

```
std::complex<double> z;
std::cin >> z;
```

το πρόγραμμα αναμένει να δώσουμε από το πληκτρολόγιο κάτι σαν

(3.2, 1.5)

2.14 Τύπος string

Ο κατάλληλος τύπος για να χειριστούμε σειρές χαρακτήρων τύπου **char** στη C++ είναι ο `std::string` που παρέχεται από το header `<string>`.

2.14.1 Δήλωση

Δήλωση ποσότητας τύπου `std::string` είναι η

```
std::string s;
```

Η παραπάνω δήλωση δημιουργεί ένα κενό `std::string`. Ισοδύναμα μπορούμε να γράψουμε

```
std::string s{};
```

Ως αρχική τιμή ενός `std::string` μπορούμε να έχουμε

- μια λίστα χαρακτήρων:

  ```
  std::string s{'a','b','c','d'}; // s <- "abcd"
  ```

- μια σειρά χαρακτήρων (εντός διπλών εισαγωγικών):

  ```
  std::string s{"Dose␣arithmo:"}; // s <- "Dose arithmo:"
  ```

- Τους πρώτους χαρακτήρες από μια σειρά χαρακτήρων:

  ```
  std::string s{"Dose␣arithmo:", 6}; // s <- "Dose a"
  ```

- Ένα άλλο `string`, με αντιγραφή του:

  ```
  std::string s1{"Dose␣arithmo:"};
  std::string s2{s1}; // s2 <- "Dose arithmo:"
  ```

  Εναλλακτικά, μπορεί να γίνει απόδοση αρχικής τιμής με μετακίνηση (§2.17.1) άλλου `string`:

  ```
  std::string s1{"Dose␣arithmo:"};
  std::string s2{std::move(s1)};
  // s2 <- "Dose arithmo:", s1 undefined
  ```

- Ένα τμήμα άλλου `string`, από μια θέση[23] και πέρα:

  ```
  std::string s1{"Dose␣arithmo:"};
  // copy all chars from position 3
  std::string s2{s1,3}; // s2 <- "e arithmo:"
  ```

- Ένα τμήμα άλλου `string`, από μια συγκεκριμένη θέση και με συγκεκριμένο πλήθος χαρακτήρων:

  ```
  std::string s1{"Dose␣arithmo:"};
  // copy 5 chars from position 3
  std::string s2{s1,3,5}; // s2 <- "e ari"
  ```

[23] η αρχική θέση είναι η μηδενική.

- Επανάληψη ενός χαρακτήρα

```
int n{6};
std::string s(n, '*'); // s <- "******"
```

  Προσέξτε ότι πρέπει να χρησιμοποιήσουμε παρενθέσεις αντί για άγκιστρα. Με άγκιστρα γίνεται απόπειρα να μετατραπεί ο ακέραιος στο πρώτο όρισμα σε χαρακτήρα ώστε να χρησιμοποιηθεί ο πρώτος τρόπος αρχικοποίησης.

- Ένα διάστημα σε ακολουθία εισόδου, που προσδιορίζεται από iterators αρχής και τέλους. Θα τους περιγράψουμε στο Κεφάλαιο 10.

2.14.2 Χειρισμός string

Ο τελεστής '=' αντιγράφει το δεξί μέλος του (ένα `string`, μια λίστα χαρακτήρων, μια σειρά χαρακτήρων, κλπ.) στο `string` που βρίσκεται στο αριστερό του μέλος, σβήνοντας τους χαρακτήρες που τυχόν έχει αυτό:

```
std::string s{"in"}; // s <- "in"
s = "out";  // s <- "out"
```

Ο τελεστής '+' μπορεί να χρησιμοποιηθεί για να ενώσει δύο `string` ή ένα `string` με μια σειρά χαρακτήρων ή ένα απλό χαρακτήρα, παράγοντας άλλο `string`:

```
std::string s1{"Dose"};
std::string s2{"arithmo:"};
std::string s3 = s1 + '␣' + s2; // s3 <- "Dose arithmo:"
```

Η σύντμησή του με το '=' συμπληρώνει ένα `string` στο τέλος του με άλλο `string` ή χαρακτήρες:

```
std::string s{"Dose"};
s+= '␣';   // s <- "Dose "
s+="arithmo:"; // s <- "Dose arithmo:"
```

Ο τελεστής '[]' με ένα ακέραιο αριθμό μεταξύ των αγκυλών, όταν γράφεται μετά από ένα `string`, μας δίνει το χαρακτήρα που βρίσκεται στη συγκεκριμένη θέση (οι θέσεις αριθμώνται από το 0):

```
std::string s{"Dose␣arithmo:"};
std::cout << s[0] << '\n';  // 'D'
std::cout << s[10] << '\n';  // 'm'
```

Οι τελεστές '>>' και '<<', όταν έχουν στο αριστερό τους μέλος μια ροή (π.χ. `std::cin` ή `std::cout` αντίστοιχα) διαβάζουν χαρακτήρες σε ένα `string` ή τυπώνουν τους χαρακτήρες που αυτό έχει.

Το `std::string`, παρόλο που δεν είναι container της Standard Library, συμπεριφέρεται σε πολλές περιπτώσεις ως τέτοιος. Δεν θα αναφερθούμε περισσότερο εδώ στις δυνατότητες που έχει.

Εκτός από το `std::string`, η C++ παρέχει ακόμα τους τύπους `std::wstring`, `std::u16string`, `std::u32string` για την αποθήκευση σειρών χαρακτήρων τύπου `wchar_t`, `char16_t` και `char32_t` αντίστοιχα.

2.15 using

Θα δούμε σε επόμενα κεφάλαια ότι τα ονόματα τύπων στη C++ μπορεί να γίνουν ιδιαίτερα μεγάλα σε μήκος ή πολύπλοκα και συνεπώς όχι ιδιαίτερα εύχρηστα, ειδικά αν χρειάζονται σε πολλά σημεία του κώδικα. Μπορούμε να ορίσουμε μια άλλη, ισοδύναμη αλλά πιο σύντομη *ονομασία* για τέτοιο τύπο με τη βοήθεια του **using**. Η σύνταξη της σχετικής εντολής είναι

using νέο_όνομα_τύπου = παλαιό_όνομα_τύπου;

Προσέξτε ότι με αυτή την εντολή δεν δημιουργείται νέος τύπος αλλά μόνο αποκτά νέο όνομα. Το νέο όνομα, αφού προσδιοριστεί, μπορεί να χρησιμοποιηθεί σε δηλώσεις. Παύει να ισχύει όταν η ροή εκτέλεσης συναντήσει το καταληκτικό άγκιστρο '}' του block στο οποίο δόθηκε η σχετική εντολή **using**.

Παράδειγμα

Το όνομα του τύπου για μιγαδικές ποσότητες, `std::complex<double>`, είναι σχετικά μεγάλο· μπορεί να χρησιμοποιείται με το πιο σύντομο όνομα `complex` αν προηγηθεί η εντολή:

```
using complex = std::complex<double>;
```

Μια δήλωση μιγαδικής μεταβλητής μπορεί κατόπιν να γίνει ως εξής:

```
complex z;
```

Ένα όνομα τύπου που έχει οριστεί μέσω της εντολής **using**, δεν επιτρέπεται να συμπληρωθεί με επιπλέον προσδιοριστές, στην προσπάθειά μας να φτιάξουμε άλλο τύπο.

Παράδειγμα

```
using integer = int;
integer a;      // Correct: a is int
long integer b; // Error: b is not long int
```

Η πρώτη εντολή δίνει το επιπλέον όνομα integer στον τύπο **int**. Η δήλωση της μεταβλητής a είναι επιτρεπτή. Η δεύτερη δήλωση όμως, αποτυγχάνει, δεν δημιουργεί μεταβλητή τύπου **long int**.

Μια άλλη χρήση του **using** είναι για να «εντοπιστεί» η δήλωση ενός τύπου ώστε να μπορεί να αλλάξει πολύ εύκολα: έστω ότι ορίζουμε μια συνάρτηση που χειρίζεται πραγματικούς αριθμούς διπλής ακρίβειας. Ο τύπος τους θα είναι **double**. Αν σε

άλλο πρόγραμμα χρειαστούμε την ίδια συνάρτηση αλλά για πραγματικούς αριθμούς απλής ακρίβειας, θα πρέπει να κάνουμε εκτεταμένες τροποποιήσεις ώστε σε κάθε εμφάνιση του τύπου `double` να τον αντικαταστήσουμε με το `float`. Εναλλακτικά, στην αρχική ρουτίνα μπορούμε να δηλώσουμε όλους τους πραγματικούς ως `real` (ή κάποιο άλλο όνομα) έχοντας δώσει πιο πριν την εντολή

```
using real = double;
```

Η μετατροπή των πραγματικών μεταβλητών της ρουτίνας σε `float` θα είναι τότε άμεση, με την εξής μοναδική αλλαγή:

```
using real = float;
```

Η τελευταία εφαρμογή της `using` μπορεί να γίνει πιο αποτελεσματικά με τη βοήθεια των συναρτήσεων template (§7.11).

2.15.1 typedef

Από τη C έχει κληρονομηθεί αντίστοιχη εντολή με την `using`, η `typedef`. Η σύνταξή της είναι

`typedef` παλαιό_όνομα_τύπου νέο_όνομα_τύπου;

Η νέα εντολή `using` μπορεί να χρησιμοποιηθεί και για την εξιδείκευση ενός template κλάσης ενώ η `typedef` δεν μπορεί. Καλό είναι πλέον, σε νέο κώδικα, να χρησιμοποιείται η `using`.

2.16 Χώρος ονομάτων (namespace)

Ένα σημαντικό πρόβλημα που συναντούμε όταν θέλουμε να συνδυάσουμε κώδικες γραμμένους από διαφορετικούς προγραμματιστές (ή ακόμα και από τον ίδιο) εμφανίζεται όταν οι κώδικες χρησιμοποιούν το ίδιο όνομα για συναρτήσεις ή καθολικές μεταβλητές. Π.χ. μπορεί να γράψουμε μια συνάρτηση που να επιλύει ένα γραμμικό σύστημα εξισώσεων με το πολύ φυσικό όνομα solve, όμως όταν αργότερα θελήσουμε να επεκτείνουμε τη συλλογή συναρτήσεών μας δε θα μπορέσουμε να χρησιμοποιήσουμε αυτό το όνομα για τη συνάρτηση[24] επίλυσης ενός συστήματος διαφορικών εξισώσεων, ενώ θα ήταν επίσης φυσικό. Η C++ υλοποιεί την έννοια του `namespace` (χώρου ονομάτων) για να αντιμετωπιστεί αυτή η κατάσταση. Η χρήση του είναι απλή: ο κώδικας

```
namespace onoma {
  ...
  double a;
  ...
}
```

[24] εκτός αν διαφέρει από την πρώτη στο πλήθος ή στον τύπο των ορισμάτων της

θέτει τη μεταβλητή a (και όποιες άλλες δηλώσεις μεταβλητών, σταθερών, συναρτήσεων, κλπ. περιέχει) στο **namespace** με όνομα onoma. Για να έχουμε πρόσβαση σε αυτή σε κώδικα μετά τη δήλωση του **namespace** πρέπει να χρησιμοποιήσουμε το πλήρες όνομά της ως εξής: `onoma::a`. Στο εσωτερικό του συγκεκριμένου **namespace** που ορίστηκε χρησιμοποιούμε απλά το όνομά της, a. Με αυτό τον τρόπο αποφεύγουμε τη σύγκρουση ονομάτων από διαφορετικούς κώδικες. Γενικότερα, είναι καλό να περικλείουμε σε κατάλληλα ονομασμένο **namespace** όλο τον κώδικα που παρουσιάζει κάποια λογική συνοχή.

Το όνομα του χώρου ονομάτων ακολουθεί τους κανόνες ονοματοδοσίας της C++ (§2.3). Ένα **namespace** επιτρέπεται να περιέχει άλλο χώρο ονομάτων. Ο προσδιορισμός της ποσότητας a που ανήκει στο χώρο ονομάτων n2, ο οποίος περιέχεται στο χώρο n1 γίνεται ως εξής: `n1::n2::a`.

Όλες οι ποσότητες που παρέχονται από τη Standard Library ορίζονται στο χώρο ονομάτων std. Αυτός είναι ο λόγος που στα ονόματα των `cin`, `cout`, `complex<>`, χρησιμοποιούμε το πρόθεμα `std::`. Αν χρειαστεί να καλέσουμε πολλές φορές σε ένα μικρό τμήμα κώδικα, συναρτήσεις από ένα χώρο ονομάτων π.χ. το std, μπορούμε να δώσουμε την εντολή

```
using namespace std;
```

στο block που περικλείει τον κώδικά μας. Από το σημείο της δήλωσης μέχρι το τέλος του συγκεκριμένου block μπορούμε να παραλείπουμε το `std::`. Π.χ.

```
#include <iostream>
#include <complex>

int
main()
{
  using namespace std;  // "std::" not needed below
  complex<double> a{2.4,3.7};
  cout << a << '\n';
}
```

Η εντολή `using namespace std;` εισάγει στην εμβέλεια στην οποία περιλαμβάνεται, *όλα* τα ονόματα ποσοτήτων, συναρτήσεων κλπ. που περιέχονται στο χώρο ονομάτων std. Αυτό έχει ως συνέπεια να μην μπορούμε να τα χρησιμοποιήσουμε για ονόματα δικών μας ποσοτήτων ή, χειρότερα, μια συνάρτηση από το std μπορεί να εκτελείται όταν (νομίζουμε ότι) καλούμε κάποια δική μας. Παρατηρήστε ότι τα ονόματα στο std μάς είναι άγνωστα.

Εναλλακτικά (και καλύτερα), μπορούμε να ορίσουμε συγκεκριμένη ποσότητα για την οποία δεν είναι απαραίτητο να προσδιορίσουμε το όνομα του **namespace**, με εντολή σαν κι αυτή:

```
using std::cout;
```

Μετά από αυτή την εντολή, μπορούμε να χρησιμοποιούμε απλά το όνομα `cout`:

```
#include <iostream>
#include <complex>

int
main()
{
  using std::cout;
  std::complex a{2.4,3.7}; // std:: necessary
  cout << a << '\n';  // std:: not necessary
}
```

Τέτοια εντολή μπορεί να επαναλαμβάνεται για κάθε αντικείμενο που επιθυμούμε να φέρουμε στην εμβέλεια που εμφανίζεται η εντολή χωρίς να χρειάζεται ο ρητός προσδιορισμός χώρου ονομάτων.

Χρήσιμος επίσης είναι και ο ανώνυμος χώρος ονομάτων,

```
namespace {
   ...
}
```

Οι ποσότητες που ορίζονται σε αυτόν, χρησιμοποιούνται απευθείας με το όνομά τους μόνο στο αρχείο που ορίζεται ο ανώνυμος `namespace` (και σε όσα το συμπεριλαμβάνουν με `#include`). Δεν μπορούν να χρησιμοποιηθούν σε άλλο αρχείο και, επομένως, να «συγκρουστούν» με ποσότητες που ορίζονται εκεί.

Τεχνικά, ο χώρος έξω από κάθε `namespace` αποτελεί τον *καθολικό χώρο ονομάτων (global namespace)*. Είναι ανώνυμος αλλά οι ποσότητες που είναι ορισμένες σε αυτόν απευθείας (και όχι μέσα σε άλλο χώρο ονομάτων, συνάρτηση ή κλάση) είναι διαθέσιμες σε όλο τον κώδικα.

Η συνάρτηση `main()` δεν επιτρέπεται να ανήκει σε κανένα άλλο χώρο ονομάτων εκτός από τον καθολικό.

2.17 Αναφορά

Η *αναφορά (reference)* αποτελεί ένα εναλλακτικό όνομα για μια ποσότητα. Αν, π.χ., έχει δηλωθεί μια ακέραια μεταβλητή με το όνομα a

```
int a;
```

τότε μπορεί να δοθεί ένα ισοδύναμο με το a όνομα (π.χ. r) στη μεταβλητή αυτή ως εξής

```
int & r{a};
```

Η αναφορά r *δεν αποτελεί νέα μεταβλητή*· αντιπροσωπεύει την ίδια ποσότητα με το a. Η δήλωση μιας αναφοράς γίνεται με τον τύπο της ποσότητας στην οποία αναφέρεται, ακολουθούμενο από το σύμβολο '`&`'. Είναι απαραίτητο να γίνει αρχική (και *μόνιμη*) σύνδεσή της με την ποσότητα στην οποία αναφέρεται (και η οποία,

βεβαίως, πρέπει να έχει δηλωθεί πιο πριν). Επιπλέον, από τη στιγμή που γίνει ο ορισμός της αναφοράς δεν μπορούμε να αλλάξουμε τον τύπο ή τη μεταβλητή με την οποία σχετίζεται.

Παράδειγμα

Στον ακόλουθο κώδικα, οποιαδήποτε αλλαγή της τιμής του a εμφανίζεται αυτόματα και στο r και αντίστροφα:

```
int a;
int & r{a};
a = 3;          // r = 3
r = 2;          // a = 2
int b{a};       // b = 2
int c{r--};     // c = 2, a = 1
```

Σημειώστε ότι αν μια ποσότητα έχει οριστεί ως `const` ή `constexpr`, πρέπει και οι αναφορές σε αυτή να δηλώνονται ως `const`:

```
int constexpr a{5};
int & p{a};   // Error
int const & q{a}; // Correct. Value of a cannot change through q.
```

Αν η δήλωση του p ήταν αποδεκτή, θα μπορούσαμε να μεταβάλουμε την τιμή του a, μιας σταθερής ποσότητας, μέσω αυτού.

Η αναφορά βρίσκει σημαντική εφαρμογή στον ορισμό συναρτήσεων, όπως θα δούμε στο §7.2. Επίσης, χρήση μιας αναφοράς γίνεται συνήθως για να «συντομεύσει» ονόματα ποσοτήτων, που στη C++ μπορεί να είναι ιδιαίτερα μεγάλα, χωρίς να γίνεται ορισμός νέας μεταβλητής και, πιθανόν χρονοβόρα, αντιγραφή της αρχικής.

Παράδειγμα

Είδαμε ότι η ποσότητα `std::numeric_limits<double>::digits10` ορίζεται ως σταθερά στο `<limits>`. Ένα πιο εύχρηστο όνομα για αυτή ορίζεται και χρησιμοποιείται στο παρακάτω πρόγραμμα:

```
#include <limits>
#include <iostream>

int
main()
{
  auto const & digits = std::numeric_limits<double>::digits10;
  std::cout << digits << '\n';
}
```

Ας αναφέρουμε, χωρίς να επεκταθούμε, ότι μπορούμε να έχουμε αναφορά συνδεόμενη με συνάρτηση ή ενσωματωμένο διάνυσμα.

2.17.1 Αναφορά σε προσωρινή ποσότητα (rvalue)

Σταθερή αναφορά

Υπάρχει η δυνατότητα να ορίσουμε μια (υποχρεωτικά) σταθερή αναφορά σε σταθερή, μεταβλητή ή γενικότερα, έκφραση κατάλληλου τύπου, όπως παρακάτω:

```
int const & p{4};
int a{3};
int const & q{a};
int const & r{2*a};
```

Η δήλωση των συγκεκριμένων αναφορών ισοδυναμεί με τον ορισμό μιας ανώνυμης, προσωρινής ποσότητας με αρχική τιμή τη σταθερή ή μεταβλητή ή έκφραση (με πιθανή μετατροπή τύπου)· κατόπιν, η αναφορά ορίζεται σε σχέση με αυτή την ποσότητα. Φυσικά, δεν μπορεί να αλλάξει η τιμή της προσωρινής ποσότητας μέσω των σταθερών αναφορών.

Μη σταθερή αναφορά

Ας επαναλάβουμε τι ακριβώς γίνεται σε μια δήλωση μεταβλητής με αρχικοποίηση, όταν η αρχική τιμή προκύπτει από μια έκφραση ή είναι επιστρεφόμενη τιμή συνάρτησης. Π.χ.

```
int a{3};
int b{4};
int c{a+b};
```

Κατά τη δήλωση της μεταβλητής c υπολογίζεται η έκφραση `a+b`, η τιμή της οποίας αποθηκεύεται σε μια *ανώνυμη, προσωρινή ποσότητα* που δημιουργείται με κατάλληλο τύπο και δεσμεύει μη προσδιορίσιμη περιοχή μνήμης. Κατόπιν, η προσωρινή ποσότητα χρησιμοποιείται για τη δημιουργία (δηλαδή, τη δέσμευση μνήμης) της δηλούμενης μεταβλητής και την απόδοση αρχικής τιμής με αντιγραφή. Μετά την ολοκλήρωση της δήλωσης, η προσωρινή μεταβλητή καταστρέφεται. Αντίστοιχα ισχύουν και όταν η αρχική τιμή προκύπτει με κλήση συνάρτησης. Η δημιουργία και καταστροφή ανώνυμης, προσωρινής μεταβλητής, σε απλές περιπτώσεις, μπορεί να παρακαμφθεί από το μεταγλωττιστή, γενικά όμως δεν ισχύει αυτό.

Μπορούμε να δώσουμε όνομα και παράταση «ζωής» σε τέτοια ανώνυμη, προσωρινή μεταβλητή που προκύπτει από έκφραση, επιστροφή συνάρτησης (γενικότερα, από ποσότητα που δεν έχει συγκεκριμένο όνομα) με κατάλληλο είδος αναφοράς:

```
int a{3};
int b{4};
int c{a+b};
int && d{a-b};
d = 6;
```

Προσέξτε τα σύμβολα '`&&`' μεταξύ του ονόματος και του τύπου στη δήλωση του d. Το όνομα d *δεν αποτελεί νέα μεταβλητή* όπως n c. Αντίθετα, είναι ένα όνομα που συνδέεται με την προσωρινή ποσότητα που προκύπτει κατά τον υπολογισμό της έκφρασης `a-b`. Η ποσότητα δεν καταστρέφεται με το τέλος της εντολής, όπως θα γινόταν σε άλλη περίπτωση, αλλά μπορεί να χρησιμοποιηθεί μέσω του ονόματός της σε όλη την περιοχή εμβέλειας που έχει αυτό. Με το συγκεκριμένο είδος αναφοράς, το οποίο σχετίζεται με ανώνυμη, προσωρινή ποσότητα, αποφεύγουμε τη δημιουργία μια μεταβλητής και καταστροφή της προσωρινής, πράξεις που πιθανόν να έχουν μεγάλες απαιτήσεις σε χρόνο εκτέλεσης ή μνήμη.

Γενικότερα, μια μεταβλητή της οποίας θα χρειαστούμε την τιμή και τη μνήμη που αυτή καταλαμβάνει, για τελευταία φορά, μπορούμε να τη χειριστούμε όπως μια ανώνυμη, προσωρινή μεταβλητή από την οποία μπορούμε να *μεταφέρουμε*, και όχι μόνο να *αντιγράψουμε*, την κατάστασή της. Σε ορισμένες περιπτώσεις, ο compiler είναι ικανός να αντιληφθεί ότι αυτό είναι εφικτό. Συχνά όμως, πρέπει να καλέσουμε τη συνάρτηση `std::move()` του `<utility>` με όρισμα τη συγκεκριμένη μεταβλητή για να διευκολύνουμε το μεταγλωττιστή. Το όνομα της συνάρτησης είναι παραπλανητικό· δεν προκαλεί κάποια μετακίνηση. Στην ουσία, ενημερώνει το μεταγλωττιστή ότι επιτρέπεται να «μετακινήσει» την τιμή της αντί να την αντιγράψει.

Ας το δούμε με ένα παράδειγμα: έστω ότι θέλουμε να εναλλάξουμε τις τιμές δύο μεταβλητών, a,b. Ο σχετικός κώδικας είναι

```
auto c = a;
a = b;
b = c;
```

Παρατηρήστε ότι οι μεταβλητές a,b χρησιμοποιούνται μία φορά για ανάγνωση της τιμής τους και μετά αποκτούν άλλη τιμή. Στον παραπάνω κώδικα, δημιουργείται η μεταβλητή c με αντιγραφή από την a και αντιγράφονται οι τιμές των b,c στις a,b. Η μεταβλητή c θα καταστραφεί όταν τελειώσει η εμβέλειά της.

Προσέξτε την παρακάτω τροποποίηση:

```
auto c = std::move(a);
a = std::move(b);
b = std::move(c);
```

Η μεταβλητή c δημιουργείται και αποκτά *χωρίς αντιγραφή* αλλά με *μετακίνηση* την τιμή (και τη μνήμη) της a. Η a δεν έχει καταστραφεί· η εσωτερική της αναπαράσταση, όμως, είναι απροσδιόριστη. Μετά τη μετακίνηση, η μεταβλητή a μπορεί μόνο να καταστραφεί (όταν λήξει η εμβέλειά της) ή να αποκτήσει με μετακίνηση την τιμή (και γενικότερα, κατάσταση) άλλης όμοιας μεταβλητής. Στην επόμενη εντολή, η a αποκτά την τιμή (και τη μνήμη) της b *χωρίς αντιγραφή*. Αντίστοιχα ισχύουν και για την b και την τιμή της c. Αν οι μεταβλητές a,b,c είναι σύνθετου τύπου, η αντιγραφή τους είναι χρονοβόρα ενώ η μετακίνηση των τιμών τους γίνεται πιο γρήγορα. Βέβαια, θα πρέπει να έχουν οριστεί για τον τύπο των ποσοτήτων κατάλληλος κατασκευαστής με μετακίνηση (move constructor) και τελεστής εκχώρησης με μετακίνηση (move assignment). Οι containers της Standard Library έχουν καθορισμένες,

είτε ρητά είτε αυτόματα, τέτοιες συναρτήσεις–μέλη.

Συνοψίζοντας, στις παρακάτω εντολές

```
auto x = y;
auto z = std::move(y);
```

έχουμε δημιουργίες μεταβλητών με απόδοση αρχικής τιμής, μιας μεταβλητής y. Στην πρώτη εντολή γίνεται *με αντιγραφή* και επομένως, αργά, και στη δεύτερη *με μετακίνηση*, και επομένως, γρήγορα. Ανάλογα, στις εντολές

```
x = y;
z = std::move(y);
```

έχουμε εκχωρήσεις της τιμής μιας μεταβλητής y, με αντιγραφή (στην πρώτη) και με μετακίνηση (στη δεύτερη). Στα παραπάνω, μετά τη μετακίνηση, η μεταβλητή y απομένει σε απροσδιόριστη κατάσταση. Όταν θέλουμε να εκμεταλλευτούμε την ταχύτητα της μετακίνησης από μια μεταβλητή, και έχουμε τη δυνατότητα να το κάνουμε (δηλαδή, δεν χρειαζόμαστε τη συγκεκριμένη μεταβλητή σε επόμενη εντολή) χρησιμοποιούμε το `std::move()` με όρισμα τη μεταβλητή ώστε να ενημερώσουμε σχετικά τον μεταγλωττιστή.

2.18 Δείκτης

Οι μεταβλητές που ορίζονται σε ένα πρόγραμμα, όπως είναι γνωστό, αποθηκεύονται για το διάστημα της «ζωής» τους σε κατάλληλες θέσεις μνήμης. Ο αριθμός της θέσης στην οποία βρίσκεται μια μεταβλητή, *η διεύθυνσή της* δηλαδή, εξάγεται με τη δράση του τελεστή '`&`' στη μεταβλητή από αριστερά. Αυτός ο αριθμός μπορεί να εκχωρηθεί σε ένα *δείκτη* σε τύπο ίδιο με τον τύπο της μεταβλητής. Η δήλωση του δείκτη γίνεται με τη μορφή

τύπος * όνομα_δείκτη;

Έτσι, αν έχουμε τον ορισμό

```
int a{3};
```

η ποσότητα `&a` είναι η θέση μνήμης στην οποία βρίσκεται η a· ορισμός ενός δείκτη σε `int`, με όνομα p, και ταυτόχρονη απόδοση τιμής γίνεται με την εντολή

```
int * p{&a};
```

Η προσπέλαση της μεταβλητής που βρίσκεται στη θέση μνήμης p επιτυγχάνεται με τη δράση του τελεστή '`*`' στο δείκτη p από αριστερά. Συνεπώς, με τους παραπάνω ορισμούς, το `*p` αποτελεί ένα άλλο όνομα για τη μεταβλητή που ορίστηκε με όνομα a· η ποσότητα αυτή μπορεί να χρησιμοποιηθεί ή αλλάξει, είτε με το όνομα `*p` είτε με το a. Π.χ.

```
double r{5.0};
double * q{&r};
*q = 3.0; // r becomes 3.0
```

Ένας δείκτης μπορεί να εκχωρηθεί σε άλλο δείκτη *ίδιου τύπου*:

```
int a{4};
int * p;
p = &a;
int * q{p};
```

Η τελευταία εντολή ορίζει το q ως δείκτη σε ακέραιο και του δίνει ως αρχική τιμή το p. Πλέον, q και p δείχνουν την ίδια θέση μνήμης και, βέβαια, τα `*q` και `*p` είναι η ίδια μεταβλητή (η a).

Επίσης, ένας δείκτης που δεν δείχνει σε συνάρτηση ή μέλος κλάσης, μπορεί να εκχωρηθεί σε δείκτη σε `void`. Η μετατροπή γίνεται αυτόματα. Έτσι, η τελευταία εντολή από τις παρακάτω

```
int a{5};
int * p{&a};
void * t{p};
```

ορίζει ένα δείκτη σε `void` και του αποδίδει ως αρχική τιμή το p, ένα δείκτη σε `int`, ή, ισοδύναμα, τη διεύθυνση του ακεραίου a. Η δράση του τελεστή '`*`' στο p μας δίνει πρόσβαση στη μεταβλητή a· αντίθετα, *η δράση του '`*`' στο t δεν επιτρέπεται.* Ένας δείκτης σε `void` πρέπει πρώτα να μετατραπεί με `static_cast<>` στον αρχικό τύπο (που ο προγραμματιστής πρέπει να γνωρίζει) και μετά να χρησιμοποιηθεί για πρόσβαση και χειρισμό της ποσότητας στην οποία «δείχνει». Σύμφωνα με τους παραπάνω ορισμούς, χρειάζεται να γράψουμε

```
int * v{static_cast<int*>(t)};
*v = 4;
```

για να δώσουμε στο a την τιμή 4.

Γενικότερα, η μετατροπή ενός `void *` σε άλλο τύπο δείκτη γίνεται μόνο ρητά με τη χρήση του `static_cast<>`.

Προσέξτε ότι ένας δείκτης που δεν του έχει αποδοθεί αρχική τιμή—είτε άλλος δείκτης είτε διεύθυνση—δείχνει σε τυχαία, απροσδιόριστη περιοχή μνήμης. Η δράση του '`*`' δεν είναι λάθος αλλά θα δώσει μια *τυχαία* τιμή κατάλληλου τύπου. Αν προσπαθήσουμε να γράψουμε στην τυχαία θέση μνήμης θα προκαλέσουμε λάθος κατά την εκτέλεση του προγράμματος *αν η συγκεκριμένη θέση δεν έχει δοθεί από το λειτουργικό σύστημα στο πρόγραμμά μας*. Αν έχει δοθεί, θα γράψουμε πάνω σε (άρα θα καταστρέψουμε) άλλη «δική μας» μεταβλητή *χωρίς να βγει λάθος*.

Επιτρέπεται να συγκρίνουμε δύο δείκτες με τους τελεστές που θα δούμε στον Πίνακα 3.1. Αν δύο δείκτες είναι ίσοι σημαίνει ότι δείχνουν στην ίδια θέση μνήμης. Επιτρέπεται αλλά δεν έχει ουσιαστική χρησιμότητα να γνωρίζουμε αν κάποιος δείκτης είναι μικρότερος ή μεγαλύτερος από άλλον, αν δηλαδή δείχνει σε προηγούμενη ή επόμενη θέση μνήμης.

Σε ένα οποιοδήποτε δείκτη μπορεί να γίνει εκχώρηση της τιμής `nullptr`. Η συγκεκριμένη ποσότητα είναι δείκτης που δε δείχνει σε καμία ποσότητα. Σε τέτοιο δείκτη δεν επιτρέπεται η δράση του '`*`' (προκαλεί λάθος κατά την *εκτέλεση*

του προγράμματος αλλά όχι κατά τη *μεταγλώττιση* του κώδικα). Η σύγκριση ενός άγνωστου δείκτη (π.χ. όρισμα συνάρτησης) με το `nullptr` πρέπει να προηγείται οποιασδήποτε απόπειρας προσπέλασης της θέσης μνήμης στην οποία θεωρούμε ότι δείχνει ο δείκτης. Προσέξτε ότι σε αυτό το σημείο υπάρχει μια βασική διαφορά με την αναφορά: ένας δείκτης μπορεί να μη συνδέεται με κανένα αντικείμενο ενώ, αντίθετα, μια αναφορά είναι οπωσδήποτε συνδεδεμένη με κάποια ποσότητα.

Για την έννοια του δείκτη σε συνάρτηση, δείτε την §7.7.

2.18.1 Σύνοψη

Ας διευκρινίσουμε εδώ ένα λεπτό σημείο στις δηλώσεις δεικτών. Προσέξτε τις παρακάτω δηλώσεις (διευκολύνεται η κατανόησή τους αν διαβαστούν από το τέλος της γραμμής προς την αρχή):

```
int a;
int * p1{&a};
int const * p2{&a};
int * const p3{&a};
int const * const p4{&a};
int * & p5{p1};
int const * & p6{p2};
int * const & p7{p3};
int const * const & p8{p4};
```

- Ο p1 είναι δείκτης σε ακέραιο.
- Ο p2 είναι δείκτης σε σταθερό ακέραιο. Αυτό σημαίνει ότι δεν μπορεί να χρησιμοποιηθεί για να αλλάξει τιμή στη μεταβλητή `*p2`.
- Ο p3 είναι σταθερός δείκτης σε ακέραιο. Αυτό σημαίνει ότι μπορεί να χρησιμοποιηθεί για να αλλάξει τιμή στη μεταβλητή `*p3` αλλά πρέπει υποχρεωτικά να πάρει αρχική τιμή και δεν μπορεί να αποκτήσει άλλη κατά τη διάρκεια της ζωής του.
- Ο p4 είναι σταθερός δείκτης σε σταθερό ακέραιο. Δεν μπορεί να αλλάξει ούτε η αρχική τιμή του ούτε να μεταβληθεί μέσω αυτού η ποσότητα στην οποία δείχνει.
- Ο p5 είναι αναφορά σε δείκτη σε ακέραιο· ταυτίζεται με τον p1.
- Ο p6 είναι αναφορά σε δείκτη σε σταθερό ακέραιο· ταυτίζεται με τον p2.
- Ο p7 είναι αναφορά σε σταθερό δείκτη σε ακέραιο· ταυτίζεται με τον p3.
- Ο p8 είναι αναφορά σε σταθερό δείκτη σε σταθερό ακέραιο· ταυτίζεται με τον p4.

Και στην περίπτωση των δεικτών ισχύει η παρατήρηση που κάναμε για τις αναφορές: ένας δείκτης για να δείξει σε σταθερή ποσότητα πρέπει να δηλωθεί κατάλληλα:

```
double x{1.2};
double * const p{&x};
double y{0.1};
p = &y; // error
double const * q{&x};
*q -= 0.2; // error
int const a{2};
int * r{&a}; // error
```

2.18.2 Αριθμητική δεικτών

Αν p είναι ένας δείκτης σε ποσότητα κάποιου τύπου, είναι προφανές ότι όλοι οι τελεστές που η δράση τους έχει νόημα για αυτό τον τύπο επιτρέπεται να δράσουν στο `*p`. Αντίθετα, στο δείκτη p μπορούν να δράσουν συγκεκριμένοι τελεστές. Από τους αριθμητικούς, οι '`++`', '`--`' (είτε πριν είτε μετά το δείκτη) έχουν νόημα: ένας δείκτης σε τύπο T μετακινείται μετά τη δράση τους κατά τόσα bytes όσα είναι το μέγεθος του T, μετά ή πριν την αρχική του τιμή. Όπως θα δούμε στην §2.11.1, το μέγεθος ποσότητας ενός τύπου T δίνεται από τον τελεστή **`sizeof`** και είναι **`sizeof`**`(T)` bytes. Επίσης, μπορούμε να προσθέσουμε ή να αφαιρέσουμε ένα ακέραιο αριθμό στο δείκτη (π.χ. `p+2`) και να μετακινηθούμε κατά το αντίστοιχο πολλαπλάσιο του **`sizeof`**`(T)`, παράγοντας νέο δείκτη. Ακόμα, το αναμενόμενο νόημα έχουν και οι τελεστές '`+=`', '`-=`' που μετακινούν το δείκτη που βρίσκεται στο αριστερό τους μέλος κατά όσα πολλαπλάσια του **`sizeof`**`(T)` προσδιορίζει ο ακέραιος στο δεξί τους μέλος. Προφανώς, πρόσθεση δεικτών δεν έχει νόημα, ενώ αντίθετα, η διαφορά δεικτών ίδιου τύπου (μόνο!) δίνει το πλήθος των θέσεων μνήμης που μεσολαβούν[25], ως πολλαπλάσιο του **`sizeof`**`(T)`:

```
int a{4};
int *p{&a};
int *q{&a+10};
auto k = q - p; // int k = 10;
```

Οι μόνες πράξεις που μπορούμε να κάνουμε σε δείκτη σε **`void`** είναι η εκχώρηση δείκτη ίδιου ή άλλου τύπου, η ρητή μετατροπή σε άλλο τύπο, ο έλεγχος για ισότητα και ανισότητα (με άλλο **`void`** `*`). Επιτρέπεται μεν η σύγκριση δεικτών σε **`void`** με τους υπόλοιπους τελεστές σύγκρισης, αλλά δεν έχει ουσιαστικό νόημα.

[25] ως ακέραιο τύπου `std::ptrdiff_t`, που ορίζεται στο `<cstddef>`.

2.19 Ασκήσεις

1. Το παρακάτω πρόγραμμα υπολογίζει και τυπώνει στην οθόνη το άθροισμα δύο ακεραίων αριθμών που διαβάζει από το πληκτρολόγιο. Γράψτε το (όχι με copy-paste!) στο αρχείο *athroisi.cpp*, μεταγλωττίστε το και εκτελέστε το.

```
#include <iostream>

int
main()
{
  std::cout << "Dose␣dyo␣akeraious\n";
  int a, b;
  std::cin >> a >> b;
  int c{a + b};
  std::cout << "To␣athroisma␣tous␣einai:␣" << c << '\n';
}
```

2. Συμπληρώστε τον παραπάνω κώδικα ώστε να υπολογίζει και τη διαφορά, το γινόμενο, το πηλίκο και το υπόλοιπο της διαίρεσης των ακέραιων αριθμών εισόδου.

3. Επαναλάβετε το παραπάνω αλλά για πραγματικούς αριθμούς (προσέξτε ότι δεν ορίζεται πηλίκο και υπόλοιπο για πραγματικούς αλλά μόνο ο λόγος τους).

4. Γράψτε κώδικα στον οποίο θα δηλώνετε μεταβλητές κατάλληλου τύπου και θα εκχωρείτε σε αυτές τους αριθμούς $4 \cdot 10^3$, 10^{-2}, $3/2$. Τυπώστε τις τιμές των μεταβλητών. Είναι αυτές που αναμένετε;

5. Γράψτε κώδικα που να υπολογίζει τις παρακάτω εκφράσεις και να τυπώνει τις τιμές τους, αφού διαβάσει από το χρήστη τους πραγματικούς αριθμούς x, y, z:

   - d = $x^2 + y^2 + z^2$
   - d = $x^2/y + z$
   - d = $2.45(x + 1.5) + 3.1(y + 0.4) + 5.2 - z/2$
   - d = $(12.8x + 5y)/(11.3y + 4z)$
   - d = $x^{2/3} + y^{2/3} + z^{2/3}$

   Αν δώσετε $x = 1.5$, $y = 2.5$, $z = 3.5$ οι εκφράσεις πρέπει να έχουν τις τιμές: 20.75, 4.4, 19.79, 0.7502958579881658, 5.457604592453865. Θα χρειαστεί, πριν την εκτύπωση, να δώσετε την εντολή `std::cout.precision(16);` για να τυπωθούν 16 σημαντικά ψηφία.

6. Να γραφεί κώδικας ο οποίος θα κάνει τα παρακάτω:

(α΄) θα εμφανίζει το μήνυμα «Δώστε την ακτίνα του κύκλου»,

(β΄) θα διαβάζει από το πληκτρολόγιο την ακτίνα,

(γ΄) θα υπολογίζει και θα εμφανίζει την περίμετρο του κύκλου (μαζί με κατάλληλο μήνυμα),

(δ΄) θα υπολογίζει και θα εμφανίζει το εμβαδόν του κύκλου (μαζί με κατάλληλο μήνυμα).

Δίνεται ότι η περίμετρος ενός κύκλου δίνεται από τη σχέση $\Gamma = 2\pi R$ και το εμβαδόν από τη σχέση $E = \pi R^2$.

7. Τρεις ακέραιοι αριθμοί a, b, c που ικανοποιούν τη σχέση $a^2+b^2 = c^2$ αποτελούν μία Πυθαγόρεια τριάδα. Μπορούμε να παραγάγουμε μια τέτοια τριάδα από δύο οποιουσδήποτε ακέραιους m, n με $m > n$, σχηματίζοντας τους αριθμούς $a = m^2 - n^2$, $b = 2mn$, $c = m^2 + n^2$.[26] Γράψτε πρόγραμμα που να διαβάζει δύο ακεραίους και να τυπώνει την αντίστοιχη Πυθαγόρεια τριάδα.

8. Να γράψετε κώδικα που θα δέχεται έναν τριψήφιο ακέραιο αριθμό και θα εμφανίζει το άθροισμα των ψηφίων του.

   Υπόδειξη: Βρείτε το πηλίκο και το υπόλοιπο της διαίρεσης του αριθμού με το 10. Τι παρατηρείτε;

9. Να γραφεί κώδικας ο οποίος θα δέχεται ένα χρονικό διάστημα σε δευτερόλεπτα και θα εμφανίζει τις μέρες, τις ώρες, τα λεπτά και τα υπόλοιπα δευτερόλεπτα στα οποία αντιστοιχεί.

   Για παράδειγμα: εάν δώσουμε ως είσοδο 200000, θα πρέπει να εμφανιστεί στην οθόνη το μήνυμα: "2 days, 7 hours, 33 min & 20 seconds".

10. Ο αλγόριθμος του Gauss για τον υπολογισμό της ημερομηνίας του Πάσχα των Ορθοδόξων σε συγκεκριμένο έτος (μέχρι το 2099) είναι ο εξής:

    - Θεωρούμε ως δεδομένο εισόδου το έτος που μας ενδιαφέρει.
    - Ορίζουμε κάποιες ακέραιες ποσότητες σύμφωνα με τους ακόλουθους τύπους:

      (α΄) r_1 = υπόλοιπο διαίρεσης του έτους με το 19.
      (β΄) r_2 = υπόλοιπο διαίρεσης του έτους με το 4.
      (γ΄) r_3 = υπόλοιπο διαίρεσης του έτους με το 7.
      (δ΄) $r_a = 19r_1 + 16$.
      (ε΄) r_4 = υπόλοιπο διαίρεσης του r_a με το 30.
      (στ΄) $r_b = 2(r_2 + 2r_3 + 3r_4)$.
      (ζ΄) r_5 = υπόλοιπο διαίρεσης του r_b με το 7.
      (η΄) $r_c = r_4 + r_5 + 3$.

[26] Μπορείτε να το επαληθεύσετε εύκολα κάνοντας την αντικατάσταση.

- Το r_c είναι η ημερομηνία του Απριλίου του συγκεκριμένου έτους, που πέφτει το Πάσχα.[27]

Γράψτε σε κώδικα τον παραπάνω αλγόριθμο. Φροντίστε να τυπώνει κατάλληλο μήνυμα και τον αριθμό r_c, αφού τον υπολογίσει.

11. Γράψτε κώδικα που

    (α΄) να διαβάζει τιμές σε δύο ακέραιες μεταβλητές,

    (β΄) να εναλλάσσει τις τιμές αυτών των μεταβλητών,

    (γ΄) να τυπώνει στην οθόνη τις νέες τους τιμές.

12. Πόσο κάνει `5^2`;

13. Δίνεται ο κώδικας

```
int m{217};
int n{813};

m ^= n;
n ^= m;
m ^= n;
```

    Ποιες είναι οι τιμές των m,n μετά την εκτέλεσή του;

14. Ρομπότ με σταθερό μήκος βήματος καταφθάνει στον πλανήτη Άρη για να περισυλλέξει πετρώματα. Κάθε βήμα του είναι $80\,\mathrm{cm}$. Το Ρομπότ διαθέτει μετρητή βημάτων. Διένυσε στον Άρη μία ευθεία από σημείο Α σε σημείο Β και ο μετρητής βημάτων καταμέτρησε Ν βήματα.

    Να γραφεί πρόγραμμα που:

    (α΄) να διαβάζει τον αριθμό Ν των βημάτων του Ρομπότ,

    (β΄) να υπολογίζει και να τυπώνει την απόσταση ΑΒ που διανύθηκε σε cm,

    (γ΄) να αναλύει και να τυπώνει αυτή την απόσταση σε km, m και cm.

    Για παράδειγμα, αν τα βήματα είναι 1253 τότε θέλουμε να τυπώνει: απόσταση $100\,240\,\mathrm{cm}$ ή $1\,\mathrm{km}$, $2\,\mathrm{m}$, $40\,\mathrm{cm}$. (ΟΕΦΕ, 2001)

15. Από τα Μαθηματικά γνωρίζουμε ότι ισχύει

$$\begin{aligned} \pi &= 4\tan^{-1}(1)\,, \\ \pi &= 8\tan^{-1}(1/3) + 4\tan^{-1}(1/7)\,. \end{aligned}$$

    Γράψτε πρόγραμμα που να υπολογίζει και να τυπώνει το π από τις δύο σχέσεις, εφαρμόζοντας για τον υπολογισμό του τόξου εφαπτομένης μιας γωνίας

[27] Αν το r_c είναι μεγαλύτερο από 30, τότε το Πάσχα είναι στις $(r_c - 30)$ Μαΐου.

x και τους τρεις παρακάτω τύπους (για να κάνετε σύγκριση της ακρίβειας μεταξύ των τύπων)

$$\tan^{-1}(x) = \cos^{-1}\left(1/\sqrt{1+x^2}\right) = \sin^{-1}\left(x/\sqrt{1+x^2}\right) \ .$$

Η ακριβής τιμή είναι $\pi = 3.14159265358979323846264338\ldots$. Πόσα ψηφία προκύπτουν σωστά με το πρόγραμμά σας; Θα χρειαστεί, πριν την εκτύπωση, να δώσετε την εντολή `std::cout.precision(16);` για να τυπωθούν 16 σημαντικά ψηφία.

16. Γράψτε πρόγραμμα που να δέχεται δύο μιγαδικούς αριθμούς από το πληκτρολόγιο και να τυπώνει στην οθόνη, *στην πολική αναπαράσταση* (δηλαδή, με μέτρο και φάση), το άθροισμα, τη διαφορά, το γινόμενο και το λόγο τους.

Κεφάλαιο 3

Εντολές Επιλογής

3.1 Εισαγωγή

Κάθε γλώσσα προγραμματισμού παρέχει τουλάχιστον μία εντολή με την οποία επιλέγεται το τμήμα κώδικα που θα εκτελεστεί κάθε φορά. Οι βασικές εντολές της C++ που ελέγχουν την ροή εκτέλεσης του προγράμματος είναι οι **if** και **switch**. Η πρώτη επιλέγει τις εντολές που θα εκτελεστούν ανάλογα με την τιμή μιας συνθήκης λογικού τύπου. Η δεύτερη κατευθύνει τη ροή εκτέλεσης ανάλογα με την τιμή μιας ποσότητας ακέραιου τύπου. Προτού παρουσιάσουμε τη σύνταξή τους, θα εξετάσουμε το σχηματισμό της λογικής συνθήκης με τους τελεστές σύγκρισης και τους λογικούς τελεστές.

3.2 Τελεστές σύγκρισης

Η C++ υποστηρίζει τη σύγκριση ποσοτήτων με τη βοήθεια των *τελεστών σύγκρισης*, Πίνακας 3.1. Το αποτέλεσμα μιας σύγκρισης σαν την `x>10` ή την `a!=b` είναι

Πίνακας 3.1: Τελεστές σύγκρισης στη C++.

Τελεστής	**Σύγκριση**	**Τελεστής**	**Σύγκριση**
==	ίσο	!=	άνισο
>	μεγαλύτερο	<	μικρότερο
>=	μεγαλύτερο ή ίσο	<=	μικρότερο ή ίσο

λογική ποσότητα και επομένως έχει τιμή **true** ή **false**. Π.χ. το `3>2` είναι **true** ενώ το `2!=1+1` είναι **false**. Οι αριθμητικοί τελεστές έχουν μεγαλύτερη προτεραιότητα

από τους τελεστές σύγκρισης, όπως παρουσιάζεται στον Πίνακα 2.3. Τελεστές σύγκρισης για μιγαδικούς αριθμούς ορίζονται, όπως είναι αναμενόμενο, μόνο οι '==' (ισότητα) και '!=' (ανισότητα). Εναλλακτικό όνομα του τελεστή '!=' είναι το **not_eq**.

Αν σε μια λογική έκφραση εμφανίζονται ποσότητες διαφορετικού τύπου, γίνονται οι προβλεπόμενες μετατροπές, §2.10, ώστε όλες οι ποσότητες να είναι ίδιου τύπου.

Παρατήρηση: Να είστε πολύ προσεκτικοί αν χρειαστεί σύγκριση για ισότητα μεταξύ πραγματικών ποσοτήτων· προσπαθήστε να την αποφεύγετε. Η πεπερασμένη αναπαράσταση των πραγματικών αριθμών οδηγεί σε σφάλματα στρογγύλευσης.

3.3 Λογικοί Τελεστές

Για τη σύνδεση λογικών εκφράσεων η C++ παρέχει τους λογικούς τελεστές

- `!` (NOT),
- `&&` (AND),
- `||` (OR).

Απολύτως ισοδύναμες με αυτούς τους τελεστές, αλλά λιγότερο χρησιμοποιούμενες, είναι οι προκαθορισμένες λέξεις **not**, **and** και **or** αντίστοιχα.

Οι τελεστές '`&&`' και '`||`' δρουν μεταξύ δύο ποσοτήτων ή εκφράσεων που είναι λογικού τύπου (ή μπορούν να μετατραπούν σε τέτοιο) και σχηματίζουν μια νέα λογική ποσότητα ενώ ο τελεστής '`!`' δρα σε μία έκφραση:

- Ο τελεστής '`!`' δρα στη λογική έκφραση που τον ακολουθεί και της αλλάζει την τιμή:

  Η έκφραση `!(4 > 3)` είναι **false**.
  Η έκφραση `!(4 < 3)` είναι **true**.

- Η λογική έκφραση που σχηματίζεται συνδέοντας δύο εκφράσεις με τον τελεστή '`&&`' έχει τιμή **true** μόνο αν και οι δύο ποσότητες είναι **true**. Σε άλλη περίπτωση είναι **false**:

  Η έκφραση `(4 > 3) && (3.0 > 2.0)` είναι **true**.
  Η έκφραση `(4 < 3) && (3.0 > 2.0)` είναι **false**.

- Ο τελεστής '`||`' μεταξύ δύο λογικών εκφράσεων σχηματίζει μια νέα ποσότητα με τιμή **true** αν έστω και μία από τις δύο ποσότητες είναι **true**, αλλιώς είναι **false**:

  Η έκφραση `(4 > 3) || (3.0 < 2.0)` είναι **true**.
  Η έκφραση `(4 < 3) || (3.0 < 2.0)` είναι **false**.

Παρατήρηση: Μια έκφραση στην οποία συμμετέχουν λογικές ποσότητες που συνδυάζονται με λογικούς τελεστές, έχει λογική τιμή και μπορεί να εκχωρηθεί σε μεταβλητή τύπου `bool`:

```
bool a{3==2};
auto b = ( (i > 0) && (i < max) );
```

3.3.1 short circuit evaluation

Ας αναφέρουμε εδώ ένα σημαντικό χαρακτηριστικό της C++: ο υπολογισμός των λογικών εκφράσεων εκτελείται από αριστερά προς τα δεξιά και σταματά μόλις έχει προσδιοριστεί η τελική τιμή της συνολικής έκφρασης. Το χαρακτηριστικό αυτό λέγεται short-circuit evaluation («υπολογισμός με βραχυκύκλωμα»). Π.χ. στην έκφραση

```
(i < 0) || (i > max)
```

αν το i είναι αρνητικό, η συνολική έκφραση είναι `true` ανεξάρτητα από τη δεύτερη συνθήκη, η οποία δεν υπολογίζεται αφού δεν χρειάζεται για τον προσδιορισμό της τιμής. Ανάλογα, στην έκφραση

```
(i >= 0) && (i < max)
```

αν το i είναι αρνητικό η συνολική έκφραση είναι `false` και δεν υπολογίζεται το `i < max`. Το χαρακτηριστικό αυτό είναι σημαντικό καθώς το τμήμα της λογικής έκφρασης που παραλείπεται μπορεί να περιλαμβάνει «παρενέργειες» (side effects) όπως μεταβολή κάποιας ποσότητας ή κλήση συνάρτησης με μεγάλες απαιτήσεις σε χρόνο εκτέλεσης ή μνήμη.

3.4 if

Η εντολή `if` είναι μία από τις βασικές δομές διακλάδωσης κάθε γλώσσας προγραμματισμού. Ελέγχει τη ροή του κώδικα ανάλογα με τη τιμή μιας λογικής συνθήκης, δηλαδή μιας ποσότητας ή έκφρασης λογικού τύπου. Στη C++ συντάσσεται ως εξής:

```
if (συνθήκη) {
        ...             // block A
} else {
        ...             // block B
}
```

Εάν η «συνθήκη» είναι αληθής ή μπορεί να μετατραπεί και να ισοδυναμεί με αληθή τιμή, εκτελείται το block των εντολών που περικλείονται μεταξύ των πρώτων {} (block A). Αν η «συνθήκη» είναι ψευδής ή ισοδυναμεί με ψευδή τιμή τότε εκτελείται το block των εντολών μετά το `else` (block B).

Το κάθε block μπορεί να αποτελείται από καμία, μία ή περισσότερες εντολές. Στην περίπτωση που το block περιλαμβάνει μία μόνο εντολή (*αρκεί να μην είναι δήλωση*), μπορούν να παραληφθούν τα άγκιστρα '{}' που την περικλείουν. Π.χ.

```
if (val > max)
   max = val;
else {
   max = 1000.0;
   ++i;
}
```

Στην περίπτωση που το block εντολών μετά το **else** είναι κενό, μπορεί να παραληφθεί ολόκληρο:

```
if (συνθήκη) {
      ...
}
```

Κάθε block μπορεί να περιλαμβάνει οποιεσδήποτε εντολές και βέβαια άλλο **if**. Σημειώστε ότι το σύμπλεγμα

```
if (συνθήκη) {
      ...
} else {
      ...
}
```

θεωρείται μία εντολή. Παρατηρήστε στην περιγραφή της σύνταξης και, στο παράδειγμα, τη στοίχιση των εντολών σε κάθε block. Η στοίχιση που επιλέχθηκε υποδηλώνει ότι βρίσκονται στο «εσωτερικό» μιας σύνθετης εντολής. Η συγκεκριμένη στοίχιση δεν είναι υποχρεωτική αλλά διευκολύνει τον προγραμματιστή ή τον αναγνώστη στην κατανόηση του κώδικα.

Στην περίπτωση ενός εσωκλειόμενου **if**, θέλει ιδιαίτερη προσοχή το επόμενο **else** του κώδικα. Το κάθε **else** ταιριάζει με το αμέσως προηγούμενό του **if** στο ίδιο block. Επομένως, ο παρακάτω κώδικας κάνει κάτι διαφορετικό απ' ό,τι υποδηλώνει η στοίχιση:

```
if (i == 0)
   if (val > max)
      max = val;
else
   max = 10;
```

Στην πραγματικότητα, η εντολή `max = 10;` εκτελείται όταν είναι αληθής η συνθήκη `(i == 0)` και ψευδής η `(val > max)` και όχι όταν δεν ισχύει το `(i == 0)`. Σε τέτοιες περιπτώσεις η χρήση των αγκίστρων μπορεί να επιβάλει την πρόθεση του προγραμματιστή:

```
if (i == 0) {
    if (val > max)
    max = val;
}
else
  max = 10;
```

Η συνθήκη στο **if** μπορεί να είναι οποιαδήποτε ποσότητα ή έκφραση (που μπορεί να περιέχει και κλήση συνάρτησης), με αποτέλεσμα που έχει τιμή λογικού τύπου ή που μπορεί να μετατραπεί σε τέτοια· αναφέραμε στο §2.5.3 τον κανόνα αυτόματης μετατροπής ενός ακεραίου ή ενός δείκτη σε **bool**.

Επίσης, η συνθήκη μπορεί να είναι δήλωση μίας μόνο ποσότητας με αρχική τιμή, η εμβέλεια της οποίας περιορίζεται στην εντολή **if** (δηλαδή, στα block πριν και μετά το πιθανό **else**):

```
if (int j = 3) {
   max = 10 + j;
}
```

Στο παραπάνω παράδειγμα, η τιμή του j αποτελεί την τιμή της συνθήκης και είναι μη μηδενική (3)· επομένως η συνθήκη θεωρείται **true**. Η μεταβλητή j έχει εμβέλεια μόνο στην εντολή **if**.

Η συνθήκη μπορεί να είναι, ακόμα, εντολή εκχώρησης:

```
if (j = 3) {
   max = 10 + j;
}
```

Σύμφωνα με όσα αναφέραμε στις §2.4 και §2.5.3, η εντολή που χρησιμοποιείται στη θέση της συνθήκης έχει τιμή 3 και επομένως είναι **true**, *ανεξάρτητα από την τιμή που είχε πριν η μεταβλητή* j. Σε αντιδιαστολή, προσέξτε πώς γράφεται η συνθήκη με σύγκριση:

```
if (j == 3) {
   max = 10 + j;
}
```

3.5 Τριαδικός τελεστής (?:)

Ο τριαδικός τελεστής, '`?:`', είναι ένας ιδιαίτερα διαδεδομένος ιδιωματισμός της C++. Δέχεται τρία ορίσματα: μια λογική συνθήκη και δυο εκφράσεις οποιουδήποτε είδους. Η έκφραση

συνθήκη ? έκφρασηΑ : έκφρασηΒ

έχει την τιμή «έκφρασηΑ» όταν η «συνθήκη» είναι αληθής, ενώ έχει την τιμή «έκφρασηΒ» όταν η συνθήκη είναι ψευδής. Ο υπολογισμός της κατάλληλης έκφρασης γίνεται μετά την επιλογή της. Επομένως, μπορούμε να γράψουμε

```
val = (condition ? value1 : value2);
```

Η val αποκτά την τιμή value1 ή την τιμή value2 ανάλογα με τη λογική τιμή της ποσότητας (ή έκφρασης) condition. Εύκολα αντιλαμβανόμαστε ότι το συγκεκριμένο παράδειγμα ισοδυναμεί με τον ακόλουθο κώδικα:

```
if (condition) {
   val = value1;
} else {
   val = value2;
}
```

Οι παρενθέσεις που περιβάλλουν τον τριαδικό τελεστή με τα ορίσματά του στο συγκεκριμένο παράδειγμα δεν είναι απαραίτητες, βοηθούν όμως στην κατανόηση του κώδικα.

Ο τριαδικός τελεστής μπορεί να χρησιμοποιηθεί και για την επιλογή του αριστερού μέλους μιας εντολής εκχώρησης ή του ορίσματος μιας συνάρτησης:

```
(k == 5 ? a : b) = 3;
```

Στο συγκεκριμένο παράδειγμα, όταν το k είναι 5 επιλέγεται η μεταβλητή a και εκτελείται η εντολή `a=3;`. Σε αντίθετη περίπτωση, επιλέγεται η b και εκτελείται η `b=3;`.

Στο τελευταίο παράδειγμα οι παρενθέσεις είναι απαραίτητες καθώς για τον τριαδικό τελεστή *δεν μπορεί να καθοριστεί μονοσήμαντα* η προτεραιότητά του σε σχέση με τον τελεστή '=' ή τους σύνθετους τελεστές εκχώρησης '+=', '-=', '*=', '/=', κλπ. Ισχύει ο ακόλουθος κανόνας: οι τελεστές '?:' και '=' έχουν ίδια προτεραιότητα, δεχόμενοι ότι οι πράξεις εκτελούνται από τα δεξιά προς τα αριστερά. Επομένως, η έκφραση `a = b ? c : d` ισοδυναμεί με `a = (b ? c : d)` ενώ η έκφραση `a ? b : c = d` εκτελείται ως `a ? b : (c = d)`. Στην τελευταία έκφραση, η εκχώρηση γίνεται μόνο όταν το a είναι **`true`**. Γενικά, η έκφρασηΒ έχει υψηλότερη προτεραιότητα όταν είναι εντολή εκχώρησης (απλής ή συνδυασμός μεταβολής και εκχώρησης), άλλος τριαδικός τελεστής ή τελεστής δημιουργίας λίστας, '{}'. Εκτελείται ή υπολογίζεται στην περίπτωση που η συνθήκη του τριαδικού τελεστή είναι ψευδής. Όσον αφορά την έκφρασηΑ, η προτεραιότητά της είναι πάντα υψηλότερη από τον τριαδικό τελεστή, σαν να περιβάλλεται από παρενθέσεις.

Καλό είναι να περιβάλλουμε πάντα με παρενθέσεις τον τριαδικό τελεστή συνολικά ώστε να εξασφαλίζουμε ότι εκτελείται η επιδιωκόμενη πράξη.

3.6 switch

Η σύνθετη εντολή **`switch`** αποτελεί μια πιο κομψή και πιο κατανοητή εκδοχή πολλαπλών εσωκλειόμενων ή διαδοχικών **`if`**. Η σύνταξή της είναι:

```
switch (i) {
  case value1:
```

```
  ...
  case value2:
  ...
  ...
  ...
  case valueN:
  ...
  default:
  ...
}
```

Η τιμή ελέγχου, i, πρέπει να είναι *ακέραιου* τύπου (χαρακτήρας, `bool`, ακέραιος) ή `enum class`. Αυτή η τιμή μπορεί να προέρχεται από ποσότητα τέτοιου τύπου ή μεταβλητή για την οποία προβλέπεται κανόνας μετατροπής σε ακέραιο τύπο. Επίσης, μπορούμε να έχουμε μια έκφραση με πράξεις και κλήσεις συναρτήσεων αρκεί ο τελικός τύπος του αποτελέσματος να είναι ακέραιος.

Οι τιμές value1, value2,...,valueN πρέπει να είναι *σταθερές*[1] ακέραιου τύπου ή `enum class` και διακριτές μεταξύ τους (να μην επαναλαμβάνονται).

Κατά την εκτέλεση, η τιμή ελέγχου συγκρίνεται με καθεμία από τις value1, value2,..., valueN. Αν η τιμή της περιλαμβάνεται σε αυτές, τότε εκτελούνται οι εντολές που ακολουθούν το αντίστοιχο `case`. *Η εκτέλεση συνεχίζει* με τις εντολές των επόμενων `case` ή/και του `default` αν έπεται, έως ότου αυτή διακοπεί με `break` (ή άλλες εντολές αλλαγής της ροής π.χ. `goto`, `return`, `throw`). Μετά τη διακοπή, η εκτέλεση συνεχίζει με την πρώτη εντολή μετά το καταληκτικό άγκιστρο της `switch`. Αν δεν συγκαταλέγεται η τιμή ελέγχου στις value1, value2,..., valueN, εκτελείται το block των εντολών που ακολουθεί το `default`, *αν υπάρχει*. Αλλιώς, η εκτέλεση συνεχίζει μετά το καταληκτικό άγκιστρο του `switch`.

Οι σχετικές θέσεις των `case` και του `default` μπορούν να είναι οποιεσδήποτε.

Παράδειγμα

Έστω ότι θέλουμε να γράψουμε κώδικα που να «διαβάζει» δύο πραγματικούς αριθμούς και ένα χαρακτήρα και να εκτελεί την πράξη μεταξύ των αριθμών που υποδηλώνει ο συγκεκριμένος χαρακτήρας. Αυτός θα είναι ένας από τους '+', '-', '*', '/'. Οποιοσδήποτε άλλος δεν είναι αποδεκτός και θα προκαλεί την εκτύπωση ενός μηνύματος που θα ενημερώνει τον χρήστη για το λάθος του και θα διακόπτει την εκτέλεση του προγράμματος. Μπορούμε να γράψουμε το εξής πρόγραμμα:

```
#include <iostream>

int
main()
```

[1] να μπορούν να υπολογιστούν κατά τη μεταγλώττιση.

```
{
    double a, b, res;
    char c;
    std::cin >> a >> b;
    std::cin >> c;

    switch (c) {
    case '+':
        res = a + b;
        break;
    case '-':
        res = a - b;
        break;
    case '*':
        res = a * b;
        break;
    case '/':
        res = a / b;
        break;
    default:
        std::cerr << "wrong␣character\n";
        return -1;
    }
    std::cout << "the␣result␣is␣" << res << '\n';
}
```

Στην περίπτωση που επιθυμούμε να εκτελείται το ίδιο block εντολών για περισσότερες από μία τιμές μπορούμε να εκμεταλλευτούμε τη μετάπτωση από το ένα `case` στο επόμενο. Έτσι, αν επιθυμούμε να εκτελέσουμε συγκεκριμένες εντολές όταν μία ακέραια ποσότητα i έχει τις τιμές 0, 1, 2 και κάποιες άλλες εντολές για τις τιμές 4, 5 μπορούμε να γράψουμε

```
switch (i) {
case 0:
case 1:
case 2:
 ....
 break;
case 4:
case 5:
 ....
 break;
}
```

3.7 Ασκήσεις

1. Γράψτε πρόγραμμα που θα δέχεται ένα ακέραιο αριθμό από το χρήστη και θα ελέγχει αν είναι άρτιος, τυπώνοντας κατάλληλο μήνυμα.

   Υπόδειξη: Άρτιος είναι ο ακέραιος που το υπόλοιπο της διαίρεσής του με το 2 είναι 0.

2. Γράψτε πρόγραμμα που θα υπολογίζει τον όγκο και το εμβαδόν της επιφάνειας μιας σφαίρας, αφού ζητήσει από το χρήστη την ακτίνα της. Αν ο χρήστης δώσει κατά λάθος αρνητικό αριθμό, να τυπώνει κατάλληλο μήνυμα.

3. Γράψτε πρόγραμμα που να επιλύει την πρωτοβάθμια εξίσωση $ax = b$, με τιμές των a, b που θα παίρνει από το χρήστη. Προσέξτε να κάνετε διερεύνηση ανάλογα με τις τιμές των a, b, δηλαδή: Ποια είναι η λύση αν (α΄) $a \neq 0$, (β΄) αν το $a = 0$ και $b = 0$, (γ΄) αν το $a = 0$ και $b \neq 0$.

4. Γράψτε πρόγραμμα που να τυπώνει τις λύσεις της δευτεροβάθμιας εξίσωσης $ax^2 + bx + c = 0$, με τιμές των (πραγματικών) a, b, c που θα παίρνει από το χρήστη. Προσέξτε να κάνετε διερεύνηση ανάλογα με τις τιμές των a,b,c, τυπώνοντας πέρα από τις λύσεις και κατάλληλα μηνύματα. Όταν οι λύσεις είναι μιγαδικές (δηλαδή, όταν η διακρίνουσα είναι αρνητική), το πρόγραμμα να πληροφορεί το χρήστη για αυτό, χωρίς να τις υπολογίζει.

5. Να τροποποιήσετε τον κώδικα που γράψατε για την επίλυση του τριωνύμου ώστε να λάβετε υπόψη την περίπτωση που υπάρχουν μιγαδικές λύσεις.

6. Γράψτε πρόγραμμα που να υπολογίζει το μέγιστο/ελάχιστο από 5 ακέραιους αριθμούς.

7. Το Υπουργείο Οικονομικών ανακοίνωσε ότι φορολογεί τα εισοδήματα των μισθωτών που αποκτήθηκαν κατά το έτος 2015 με βάση την παρακάτω κλίμακα:

Εισόδημα (σε Ευρώ)	Συντελεστής Φόρου
0 - 20000	22%
20000,01 - 30000	29%
30000,01 - 40000	37%
από 40000,01 και πάνω	45%

   Για παράδειγμα, εάν κάποιος έχει εισόδημα 48000 ευρώ, για τα πρώτα 20000 ευρώ θα φορολογηθεί με ποσοστό 22% (φόρος 4400 Ευρώ), για τα επόμενα 10000 ευρώ θα φορολογηθεί με ποσοστό 29% (φόρος 2900 Ευρώ), για τα επόμενα 10000 ευρώ θα φορολογηθεί με ποσοστό 37% (φόρος 3700 Ευρώ) και για τα υπόλοιπα 8000 ευρώ θα φορολογηθεί με ποσοστό 45% (φόρος 3600 Ευρώ). Συνολικά θα πληρώσει 14600 Ευρώ.

   Γράψτε κώδικα που θα διαβάζει από το πληκτρολόγιο ένα ποσό (το εισόδημα) και θα υπολογίζει το φόρο που του αναλογεί.

8. Έχετε τις εξής πληροφορίες:

   - Οι μήνες Ιανουάριος, Μάρτιος, Μάιος, Ιούλιος, Αύγουστος, Οκτώβριος, Δεκέμβριος έχουν 31 ημέρες.
   - Οι μήνες Απρίλιος, Ιούνιος, Σεπτέμβριος, Νοέμβριος έχουν 30 ημέρες.
   - Ο Φεβρουάριος έχει 28 ημέρες εκτός αν το έτος είναι δίσεκτο, οπότε έχει 29.
   - Δίσεκτα είναι τα έτη που διαιρούνται ακριβώς με το 4, εκτός από τις εκατονταετίες. Οι εκατονταετίες είναι δίσεκτες όταν διαιρούνται με το 400. Επομένως: ένα έτος που διαιρείται ακριβώς με το 4 αλλά όχι με το 100 είναι δίσεκτο. Είναι επίσης δίσεκτο αν διαιρείται με το 400.
   - Η αλλαγή από το παλαιό στο νέο ημερολόγιο έγινε στις 16 Φεβρουαρίου 1923 (με το παλαιό) που ορίστηκε ως 1η Μαρτίου 1923 (στο νέο). Συνεπώς, ο Φεβρουάριος του 1923 είχε 15 ημέρες.

   Γράψτε κώδικα, χρησιμοποιώντας το `switch`, που να διαβάζει μήνα και έτος από το χρήστη και να τυπώνει στην οθόνη τις ημέρες του μήνα.

9. Ο αλγόριθμος του Zeller υπολογίζει την ημέρα (Κυριακή, Δευτέρα, ...) κάποιας ημερομηνίας ως εξής:

   Έστω d είναι η ημέρα του μήνα $(1, 2, 3, \ldots, 31)$, m ο μήνας $(1, 2, \ldots, 12)$ και y το έτος. Αν ο μήνας είναι 1 (Ιανουάριος) ή 2 (Φεβρουάριος) προσθέτουμε στο m το 12 και αφαιρούμε 1 από το έτος y. Κατόπιν,

   (α′) Ορίζουμε το a να είναι το πηλίκο της διαίρεσης του $13(m + 1)$ με το 5.

   (β′) Ορίζουμε τα j, k να είναι το πηλίκο και το υπόλοιπο αντίστοιχα, της διαίρεσης του έτους με το 100.

   (γ′) Ορίζουμε το b να είναι το πηλίκο της διαίρεσης του j με το 4.

   (δ′) Ορίζουμε το c να είναι το πηλίκο της διαίρεσης του k με το 4.

   (ε′) Ορίζουμε το h να είναι το άθροισμα των a, b, c, d, k και του πενταπλάσιου του j.

   Το υπόλοιπο της διαίρεσης του h με το 7 είναι η ημέρα: αν είναι 0 η ημέρα είναι Σάββατο, αν είναι 1 η ημέρα είναι Κυριακή, κλπ.

   Γράψτε πρόγραμμα που να δέχεται μια ημερομηνία από το χρήστη και να τυπώνει την ημέρα της εβδομάδας αυτής της ημερομηνίας.

Κεφάλαιο 4

Εντολές επανάληψης

4.1 Εισαγωγή

Βασική ανάγκη ύπαρξης ενός υπολογιστή είναι να μας απαλλάξει από απλές επαναληπτικές διαδικασίες, εκτελώντας τις με ακρίβεια και ταχύτητα.

Για να καταλάβουμε την ανάγκη ύπαρξης μιας εντολής επανάληψης, ας δούμε το εξής πρόβλημα: Έστω ότι θέλουμε να τυπώσουμε στην οθόνη τους αριθμούς 1, 2, 3, 4, 5, σε ξεχωριστή γραμμή τον καθένα. Η εκτύπωση, σύμφωνα με όσα ξέρουμε μέχρι τώρα, μπορεί να γίνει ως εξής

```
std::cout << 1 << '\n';
std::cout << 2 << '\n';
std::cout << 3 << '\n';
std::cout << 4 << '\n';
std::cout << 5 << '\n';
```

Η προσέγγιση αυτή είναι εφικτή καθώς το πλήθος των αριθμών είναι μικρό. Πώς θα μπορούσαμε να επεκτείνουμε αυτές τις εντολές, σύντομα και σωστά, αν θέλαμε να τυπώσουμε μέχρι π.χ. το 500;

Μπορούμε να τροποποιήσουμε τις παραπάνω εντολές ώστε να τις φέρουμε σε κατάλληλη μορφή για επανάληψη. Έτσι, η εκτύπωση μπορεί να γίνει ως εξής

```
int i{1};
std::cout << i << '\n';
i = 2;
std::cout << i << '\n';
i = 3;
std::cout << i << '\n';
i = 4;
```

```
std::cout << i << '\n';
i = 5;
std::cout << i << '\n';
```

Παρατηρήστε ότι φέραμε τον κώδικα στη μορφή που μια εντολή επαναλαμβάνεται αυτούσια ενώ, μετά από τη συγκεκριμένη εντολή, μια ακέραια μεταβλητή αυξάνεται διαδοχικά κατά σταθερό βήμα, εδώ 1.

Η C++ παρέχει ενσωματωμένες τέσσερις εντολές επανάληψης[1]: τη **for**, την παραλλαγή της, **range for**, τη **while** και τη **do while**. Οι **for**, **while**, **do while** είναι ισοδύναμες, με την έννοια ότι ένας βρόχος υλοποιημένος με μία από αυτές μπορεί εύκολα να μετατραπεί σε βρόχο βασισμένο σε άλλη (με πιθανή εφαρμογή και της εντολής **break**).

Η χρήση της γενικότερης εντολής από όλες, της **for**, απλοποιεί τον κώδικα του παραδείγματος ως εξής

```
for (int i{1}; i <= 5; ++i) {
  std::cout << i << '\n';
}
```

Σε αυτή τη μορφή, η τροποποίηση του κώδικα ώστε να εκτυπώνει τους αριθμούς π.χ. μέχρι το 500 είναι απλή: αλλάζουμε το άνω όριο της μεταβλητής i.

Προτού εξηγήσουμε πώς εκτελείται η εντολή **for**, ας παραθέσουμε την υλοποίηση του παραδείγματος με τις άλλες δύο εντολές επανάληψης της γλώσσας, την εντολή **while** και την εντολή **do while**:

```
int i{1};
while (i <= 5) {
  std::cout << i << '\n';
  ++i;
}
```

και

```
int i{1};
do {
  std::cout << i << '\n';
  ++i;
} while (i <= 5);
```

4.2 for

Η εντολή **for** είναι η γενικότερη και πιο πολύπλοκη εντολή επανάληψης της C++. Η γενικευμένη σύνταξή της είναι:

[1] πέρα από την **goto** (§4.6.3) που μπορεί αλλά καλό είναι να μην χρησιμοποιηθεί για το σκοπό αυτό.

```
for (αρχική_εντολή; συνθήκη; τελική_εντολή) {
...
}
```

Η εντολή `for` εκτελείται ως εξής:

1. Εκτελείται η «αρχική_εντολή». Αυτή η εντολή μπορεί να είναι και δήλωση μεταβλητής (ή ακόμα και σύνολο εντολών χωριζόμενων με τον τελεστή κόμμα ‘,’ (§2.11.3)).
2. Ελέγχεται η «συνθήκη».
   - Αν είναι ψευδής, η ροή συνεχίζει με την πρώτη εντολή μετά το σύμπλεγμα `for`.
   - Αν είναι αληθής, εκτελείται το block εντολών μεταξύ των αγκίστρων ‘{}’. Αν δεν υπάρξει αλλαγή της ροής στο block (με `break`, `return`, `goto`, `throw`,...) εκτελείται η «τελική_εντολή».
3. Αν εκτελέστηκε το block χωρίς αλλαγή ροής, επαναλαμβάνεται το βήμα 2 (έλεγχος της συνθήκης). Η τιμή της «συνθήκης» μπορεί να έχει μεταβληθεί στο προηγούμενο βήμα, καθώς μπορεί να περιλαμβάνει ποσότητες που αλλάζουν κατά την επανάληψη.

Ένα ή περισσότερα από τα «αρχική_εντολή», «συνθήκη», «τελική_εντολή» μπορεί να απουσιάζουν. Αν λείπει η «συνθήκη», οι έλεγχοί της στην εκτέλεση του `for` θεωρούνται αληθείς.

Προσέξτε ότι αν η «αρχική_εντολή» περιλαμβάνει δήλωση μεταβλητής, η εμβέλεια αυτής (§2.8) περιορίζεται στο σύμπλεγμα της `for`, μέχρι, δηλαδή, το καταληκτικό ‘}’ του σώματός της.

Ας δούμε λοιπόν, πώς γίνεται η εκτύπωση της προηγούμενης παραγράφου με τη χρήση του `for`:

```
for (int i{1}; i <= 5; ++i) {
  std::cout << i << '\n';
}
```

Αρχικά, εκτελείται η δήλωση της μεταβλητής `i`, με απόδοση του 1 ως αρχικής της τιμής. Ελέγχεται αν η τιμή του `i` είναι μικρότερη ή ίση με το 5. Καθώς είναι, εκτελείται η εντολή εκτύπωσης. Κατόπιν, εκτελείται η τρίτη εντολή στο `for`, η αύξηση του `i`, και επαναλαμβάνεται ο βρόχος από τον έλεγχο της συνθήκης.

Παράδειγμα

Έστω ότι θέλουμε να υπολογίσουμε το άθροισμα των περιττών ακέραιων αριθμών από το 1 έως και το 9 στην ακέραια μεταβλητή `s`. Προφανώς, μπορούμε να γράψουμε

```
int s{1 + 3 + 5 + 7 + 9};
```

Ας κάνουμε τον υπολογισμό με τη χρήση εντολής επανάληψης `for`. Μπορούμε, σε πρώτο στάδιο, να γράψουμε το ακόλουθο:

```
int s{0};
s += 1;
s += 3;
s += 5;
s += 7;
s += 9;
```

Στον κώδικα αυτό προσθέτουμε σταδιακά, έναν–έναν, τους όρους στη μεταβλητή s, την οποία μηδενίζουμε αρχικά ώστε να έχει ουδέτερη τιμή στην πρώτη χρήση της σε άθροισμα.

Σε δεύτερο στάδιο, διαμορφώνουμε τον κώδικα ώστε να έχουμε μια εντολή που επαναλαμβάνεται αυτούσια ενώ αυξάνεται σταθερά μία ακέραια μεταβλητή:

```
int s{0};
int i{1};

s += i;
i += 2;

s += i;
i += 2;

s += i;
i += 2;

s += i;
i += 2;

s += i;
```

Η χρήση της εντολής `for`, σύμφωνα με όσα περιγράψαμε, απλοποιεί τον τελικό κώδικά μας:

```
int s{0};
for (int i{1}; i<=9; i+=2) {
  s += i;
}
```

Προσέξτε ότι στην παραπάνω επανάληψη εκτελείται αναπόφευκτα, μετά την τελευταία αύξηση του s, μια (περιττή) επιπλέον εντολή, η `i+=2;`.

Επιπλέον, παρατηρήστε ότι η μεταβλητή i δεν μπορεί να χρησιμοποιηθεί μετά το βρόχο. Αν είναι επιθυμητό κάτι τέτοιο, η δήλωση `int i;` πρέπει να

γίνει πριν το `for` ώστε να μεγαλώσει η εμβέλεια της μεταβλητής:

```
int s{0};
int i{1};
for (   ; i<=9; i+=2) {
  s += i;
}
```

Παρατηρήστε ότι μετά το `for` το `i` έχει την τιμή 11.

4.2.1 Χρήση

Βασιζόμενοι στην παραπάνω ανάλυση, αν έχουμε εντολές που επαναλαμβάνονται, φροντίζουμε να τις φέρουμε στη μορφή

- σύνολο εντολών (εξαρτώμενες ή μη από μια «ακέραια_μεταβλητή», αλλά με την ίδια μορφή ανεξάρτητα από την τιμή της μεταβλητής).
- «ακέραια_μεταβλητή» += «βήμα_αύξησης».

Αν το επιτύχουμε αυτό, τότε μπορούμε να γράψουμε την παραπάνω σειρά επαναλαμβανόμενων εντολών πιο συνοπτικά και κατανοητά με τη χρήση του `for`. Παρόλο που δεν υπάρχει περιορισμός στον τύπο της μεταβλητής ελέγχου (ούτε καν είναι αναγκαία η ύπαρξή της), καλό είναι να χρησιμοποιούμε το `for` όταν μπορούμε να διαμορφώσουμε τις εντολές μας ώστε να έχουμε *ακέραια μεταβλητή* που αλλάζει τιμή με σταθερό βήμα. Σε αντίθετη περίπτωση, οι άλλες εντολές επανάληψης, `while` (§4.4) και `do while` (§4.5), είναι καταλληλότερες. Ας επαναλάβουμε την παρατήρηση στο §3.2: πρέπει να αποφεύγουμε τη σύγκριση για ισότητα μεταξύ πραγματικών αριθμών.

Παραδείγματα

- Το άθροισμα των πρώτων 100 ακεραίων αριθμών, από το 1 ως το 100, υπολογίζεται με τον ακόλουθο κώδικα:

  ```
  int s{0};
  for (int i{1}; i<= 100; ++i) {
    s += i;
  }
  ```

- Το άθροισμα των άρτιων ακεραίων αριθμών στο διάστημα $[0, 1000]$ υπολογίζεται με τον ακόλουθο κώδικα:

  ```
  int s{0};
  for (int i{0}; i<= 1000; i+=2) {
    s += i;
  }
  ```

- Η εκτύπωση των αριθμών 99, 97, 95,..., 3, 1, με αυτή τη σειρά, μπορεί να γίνει με τον κώδικα

```
for (int i{99}; i >= 1; i-=2) {
  std::cout << i << '␣';
}
```

- Το άθροισμα των αριθμών 0.0, 0.1, 0.2,...,1.9 μπορεί να βρεθεί ως εξής

```
double s{0.0};
for (int i{0}; i <= 19; ++i) {
  s += 0.1 * i;
}
```

Θα υπέθετε κανείς ότι το ίδιο ακριβώς επιτυγχάνεται με τον κώδικα

```
double s{0.0};
for (double x{0.0}; x <= 1.9; x+=0.1) {
  s += x;
}
```

Ποιο αποτέλεσμα αναμένετε; Δοκιμάστε τον κώδικα και δείτε την παρατήρηση στο §3.2.

- Το πλήθος των ακεραίων που είναι πολλαπλάσιοι του 2 ή του 3 στο διάστημα $[5, 108]$ μπορεί να υπολογιστεί ως εξής

```
int k{0};
for (int i{5}; i <= 108; ++i) {
  if (i%2 == 0 || i%3 == 0) {
     ++k;
  }
}
```

Στο παράδειγμα αυτό, το πλήθος υπολογίζεται στη μεταβλητή k που χρησιμοποιείται ως *μετρητής*. Ο μετρητής στον προγραμματισμό είναι μια ακέραια μεταβλητή που δηλώνεται και παίρνει αρχική τιμή 0 *ακριβώς πριν* την εντολή επανάληψης και αυξάνει κατά ένα όταν ικανοποιείται κάποια συνθήκη. Με αυτή την τεχνική μετράμε πόσες φορές σε μια επανάληψη είναι αληθής μια συνθήκη.

Στην περίπτωση που ενδιαφερόμαστε όχι για το *πόσες φορές* αληθεύει μια συνθήκη σε κάποια επανάληψη αλλά μόνο για το *αν* αληθεύει, ταιριάζει να χρησιμοποιήσουμε ως μετρητή μια μεταβλητή λογικού τύπου. Η μεταβλητή αυτή θα ξεκινά πριν την επανάληψη με κάποια τιμή και θα αλλάζει όταν ικανοποιηθεί η συνθήκη.

- Η ακολουθία αριθμών 0,1,1,2,3,5,8,13,..., στην οποία κάθε μέλος της, από το τρίτο και μετά, είναι το άθροισμα των δύο προηγούμενων μελών, είναι η ακολουθία Fibonacci. Αν θέλουμε να τυπώσουμε στην οθόνη τους n πρώτους όρους, θα πρέπει να υπολογίσουμε ένα άθροισμα $n-2$ φορές. Προφανώς, θα χρησιμοποιήσουμε εντολή επανάληψης. Παρατηρήστε ότι για κάθε «νέο» όρο πρέπει να αθροίσουμε τον τρέχοντα όρο και τον προηγούμενό του όρο. Χρειαζόμαστε επομένως τρεις μεταβλητές: έστω η a είναι ο προηγούμενος όρος, η b ο τρέχων και η c ο νέος όρος. Η εντολή που πρέπει να επαναλάβουμε είναι η `c=a+b;`.

  Προσέξτε ότι η «ζωή» ενός όρου στο πρόγραμμά μας είναι περιορισμένη: όταν υπολογίζουμε ένα νέο όρο θα τον τυπώνουμε και κατόπιν θα τον χρησιμοποιήσουμε ως τρέχοντα στην επόμενη επανάληψη και ως προηγούμενο όρο στη μεθεπόμενη. Μετά παύει να χρειάζεται. Επομένως, μπορούμε να χρησιμοποιήσουμε μόνο τρεις μεταβλητές, αρκεί σε κάθε επανάληψη να τους αλλάζουμε κατάλληλα τις τιμές: η μεταβλητή που αποθηκεύει τον προηγούμενο, κάθε φορά, όρο να παίρνει τον τρέχοντα όρο· η μεταβλητή του τρέχοντα να παίρνει την τιμή του νέου όρου. Ετοιμάζονται με αυτό τον τρόπο για την επόμενη επανάληψη. Ο σχετικός κώδικας είναι

  ```
  int a{0};
  int b{1};
  std::cout << a << '\n' << b << '\n';
  for (int k{2}; k <= n; ++k) {
     int c{a+b};
     std::cout << c << '\n';
     a = b;
     b = c;
  }
  ```

  Παρατηρήστε ότι πριν αρχίσει η επανάληψη, οι μεταβλητές του προηγούμενου και του τρέχοντα όρου παίρνουν κατάλληλες τιμές ώστε να ξεκινήσει σωστά η επαναλαμβανόμενη εντολή.

4.3 Range for

Μια παραλλαγή του `for` διευκολύνει όταν επιθυμούμε να «διατρέξουμε» μια ομάδα μεταβλητών. Όπως θα δούμε, η C++ παρέχει δομές για ομαδοποίηση ποσοτήτων ίδιου τύπου. Μεταξύ αυτών, το ενσωματωμένο διάνυσμα, οι λίστες, οι containers της Standard Library. Έστω a είναι μια τέτοια ομάδα ποσοτήτων. Αν θέλουμε να χρησιμοποιήσουμε τις τιμές των στοιχείων της διαδοχικά, μπορούμε να γράψουμε τον κώδικα

```
for (auto x : a) {
  ....  // use value of x
```

```
}
```

Σε αυτή την εντολή δημιουργούμε μια μεταβλητή x με τύπο ίδιο με τα στοιχεία του a (με τη χρήση του `auto`). Το `range for` εξασφαλίζει ότι η μεταβλητή x θα πάρει διαδοχικά τις τιμές των στοιχείων του a, με τη σειρά από το πρώτο έως το τελευταίο, και με αυτές θα συμμετέχει στις εντολές του σώματος του `for`. Η εμβέλεια της μεταβλητής του `range for` είναι στο εσωτερικό της εντολής.

Παράδειγμα

Η εκτύπωση των αριθμών $4, 5, 8, -6$ μπορεί να γίνει ως εξής

```
for (auto x : {4,5,8,-6}) {
  std::cout << x << '\n';
}
```

Στην περίπτωση που θέλουμε να τροποποιήσουμε τις τιμές των στοιχείων του a, π.χ. να δώσουμε τιμές στο a από το πληκτρολόγιο, μπορούμε να γράψουμε

```
for (auto & x : a) {
  std::cin >> x;
}
```

Προσέξτε τη χρήση της αναφοράς στα στοιχεία του a· θέλουμε να περάσει σε αυτά η μεταβολή του x.

Το `range for` μπορεί να χρησιμοποιηθεί και για να διατρέξουμε container που έχει δημιουργήσει ο προγραμματιστής, αρκεί στον τύπο του να έχουν οριστεί οι συναρτήσεις–μέλη `begin()` και `end()`, με ανάλογες ιδιότητες όπως στους containers της Standard Library (§11.4).

4.4 while

Άλλη διαθέσιμη εντολή επανάληψης της C++ είναι η εντολή `while`. Αποτελεί την πιο απλή υλοποίηση βρόχου (δηλαδή επαναληπτικής διαδικασίας). Ταιριάζει να τη χρησιμοποιήσουμε όταν υπάρχει η ανάγκη να επαναλάβουμε ένα τμήμα εντολών χωρίς να γνωρίζουμε εκ των προτέρων το πλήθος των επαναλήψεων. Συντάσσεται ως εξής:

```
while (συνθήκη) {
...
}
```

Κατά την εκτέλεση της εντολής `while`:

1. Ελέγχεται η «συνθήκη»:
   - Αν είναι ψευδής, η ροή συνεχίζει με την πρώτη εντολή μετά το σύμπλεγμα.

- Αν είναι αληθής εκτελείται το block εντολών μεταξύ των αγκίστρων '{}'.

2. Αν εκτελέστηκε το block χωρίς αλλαγή ροής (από **break**, **return**, **goto**, **throw**, ...), επαναλαμβάνεται η διαδικασία από το βήμα 1 (έλεγχος της «συνθήκης»). Η τιμή της «συνθήκης» μπορεί να μεταβληθεί κατά την εκτέλεση του block εντολών.

Παράδειγμα

Έστω ότι θέλουμε να υπολογίσουμε σε μια μεταβλητή s το άθροισμα των ακέραιων που δίνει ο χρήστης από το πληκτρολόγιο, έως ότου δώσει τον αριθμό 0. Μπορούμε να γράψουμε τον παρακάτω κώδικα

```
int i;
std::cin >> i;

int s{0};
while (i != 0) {
  s += i;
  std::cin >> i;
}
```

4.5 do while

Η εντολή **do while** είναι μια παραλλαγή του **while** (§4.4), στην οποία το σώμα του βρόχου εκτελείται τουλάχιστον μία φορά. Η σύνταξή της είναι:

do { ...
} **while** (συνθήκη);

Κατά την εκτέλεση του συμπλέγματος **do while**:

1. Εκτελείται το block εντολών μεταξύ των αγκίστρων '{}'.

2. Αν δεν υπήρξε αλλαγή ροής (από **break**, **return**, **goto**, **throw**,...), ελέγχεται η «συνθήκη»:

   - Αν είναι ψευδής, η ροή συνεχίζει με την πρώτη εντολή μετά την εντολή.
   - Αν είναι αληθής επαναλαμβάνεται η διαδικασία από το βήμα 1 (εκτέλεση του block).

   Η τιμή της «συνθήκης» μπορεί να μεταβάλλεται σε κάθε επανάληψη.

Παράδειγμα

Έστω ότι θέλουμε να εξασφαλίσουμε ότι ένας ακέραιος που το πρόγραμμά μας θα διαβάζει από το πληκτρολόγιο είναι θετικός. Αν ο χρήστης δώσει αρνητικό ή μηδέν, το πρόγραμμα να επαναλαμβάνει το διάβασμα. Μπορούμε να γράψουμε τον παρακάτω κώδικα

```
int i;
do {
  std::cout << "Dose␣thetiko␣akeraio:␣";
  std::cin >> i;
  std::cout << '\n';
} while (i <= 0);
```

4.6 Βοηθητικές εντολές

4.6.1 break

Η εντολή **break** μπορεί να εμφανίζεται μόνο σε εντολή επιλογής **switch** ή στο σώμα βρόχου (**for** ή **range for**, **while**, **do while**). Η εκτέλεσή της προκαλεί έξοδο από την εντολή στην οποία περικλείεται, μεταφέροντας τη ροή στην αμέσως επόμενη εντολή από αυτή.

Παράδειγμα

Παράδειγμα χρήσης είναι το ακόλουθο: Ας υποθέσουμε ότι θέλουμε να εξασφαλίσουμε ότι ένας ακέραιος αριθμός εισόδου είναι θετικός. Αν ο χρήστης δώσει αρνητικό ή μηδέν το πρόγραμμα να βγάζει σχετικό μήνυμα και να επαναλαμβάνει το διάβασμα. Μπορούμε να γράψουμε τον παρακάτω κώδικα

```
int i;
while (true) {
  std::cout << "Dose␣thetiko␣akeraio:␣";
  std::cin >> i;
  std::cout << '\n';
  if (i > 0) {
    break;
  }
  std::cerr << "Edwses␣arnhtiko\n";
}
 ... use i
```

Παρατηρήστε ότι η «συνθήκη» στην εντολή επανάληψης έχει τη σταθερή τιμή **true**. Υλοποιούμε έτσι ένα ατέρμονα βρόχο. Εναλλακτικά, θα μπορούσαμε

να εντάξουμε τις επαναλαμβανόμενες εντολές σε `do {...} while (true);` ή `for (;;) {...}`.

4.6.2 continue

Η εντολή `continue` μπορεί να εμφανίζεται μόνο στο σώμα βρόχου (`for` ή range `for`, `while`, `do while`). Η εκτέλεσή της μεταφέρει τη ροή του προγράμματος στο καταληκτικό άγκιστρο του βρόχου, απ' όπου συνεχίζει η εκτέλεσή του.

Παράδειγμα

Παράδειγμα χρήσης είναι το ακόλουθο: Έστω ότι θέλουμε να τυπώσουμε τις τετραγωνικές ρίζες των πρώτων 10 αριθμών εισόδου, αγνοώντας όμως τους αρνητικούς. Ο σχετικός κώδικας μπορεί να είναι

```
for (int i{1}; i <= 10; ++i) {
  double x;
  std::cin >> x;
  if (x < 0.0) {
    continue;
  }
  std::cout << "H␣tetragwnikh␣riza␣einai" << std::sqrt(x) << '\n';
}
```

4.6.3 goto

Σε μια εντολή στο σώμα μιας συνάρτησης μπορεί να δοθεί κάποια ετικέτα (label). Το όνομά της (π.χ. labelname) σχηματίζεται με τους ίδιους κανόνες που ισχύουν για τα ονόματα των μεταβλητών (§2.3). Η απόδοση ετικέτας σε μια εντολή γίνεται ως εξής

```
labelname : statement;
```

δηλαδή, γράφουμε πριν την εντολή την ετικέτα, ακολουθούμενη από το χαρακτήρα ':'. Από άλλη θέση στο σώμα της *ίδιας* συνάρτησης, πριν[2] ή μετά την ετικέτα, μπορούμε να συνεχίσουμε την εκτέλεση με τη συγκεκριμένη εντολή με τη χρήση της `goto`:

```
goto labelname;
```

Όταν η ροή εκτέλεσης συναντήσει μια εντολή `goto` μεταπηδά στο σημείο που αυτή υποδεικνύει.

Προσέξτε ότι με το `goto` δεν επιτρέπεται

[2] η ετικέτα αποτελεί το μοναδικό (πέρα από τα μέλη κλάσης) χαρακτηριστικό της C++ που μπορεί να χρησιμοποιηθεί προτού το «συναντήσει» ο μεταγλωττιστής.

- να «υπερπηδήσουμε» έναν ορισμό ποσότητας, καθώς αυτή δε θα μπορεί να χρησιμοποιηθεί στο σημείο του κώδικα που θα μεταβούμε,
- να εισέλθουμε σε block κάποιου `catch`, σε περιοχή, δηλαδή, που διαχειρίζεται μια εξαίρεση (exception), καθώς αυτή δεν θα έχει συλληφθεί.

Η χρήση της `goto` πρέπει να αποφεύγεται. Η C++ παρέχει τις κατάλληλες εντολές ελέγχου και επαναληπτικές δομές καθιστώντας την `goto` περιττή. Μοναδική περίπτωση που η χρήση της είναι προτιμότερη από τις εναλλακτικές λύσεις, εμφανίζεται όταν επιθυμούμε έξοδο από πολλαπλούς βρόχους ή πολλαπλά `switch`, το ένα μέσα στο άλλο. Σε τέτοια περίπτωση μπορούμε να χρησιμοποιήσουμε το `goto` ουσιαστικά ως πιο ευέλικτο `break`. Ακόμη και τότε, η «μετακίνηση» με τη `goto` προς προηγούμενο σημείο του κώδικα πρέπει να αποφεύγεται.

4.7 Παρατηρήσεις

Όπως αναφέραμε και στο §3.4, κάθε block μπορεί να αποτελείται από καμία, μία, ή περισσότερες εντολές. Στην περίπτωση που περιλαμβάνει μία μόνο εντολή μπορούν να παραλειφθούν τα άγκιστρα '{}' που την περικλείουν.

Οπουδήποτε εμφανίζεται λογική συνθήκη στις εντολές επανάληψης, μπορούμε να έχουμε οποιαδήποτε έκφραση (που μπορεί να περιέχει και κλήση συνάρτησης), αρκεί να έχει ή να ισοδυναμεί με λογική τιμή. Δείτε τη σχετική συζήτηση στο §3.4.

Στο Κεφάλαιο 12 θα δούμε μια άλλη κατηγορία βρόχων, κάποιες έτοιμες συναρτήσεις που εκτελούν επαναλαμβανόμενες εργασίες σε containers ή λίστες.

4.8 Ασκήσεις

1. Γράψτε πρόγραμμα που να τυπώνει στην οθόνη τους αριθμούς 0.0, 0.1, 0.2, 0.3, 0.4, ..., 2.5.

2. Τυπώστε τις πρώτες 20 δυνάμεις του 2 (2^0, 2^1, ...,2^{19}).

3. Τυπώστε το παραγοντικό αριθμού που θα δίνει ο χρήστης. Το n! ορίζεται ως εξής:

$$n! = 1 \times 2 \times 3 \times \ldots \times n, \qquad 0! = 1.$$

   Φροντίστε ώστε το πρόγραμμά σας να μη δέχεται n μεγαλύτερο από 12.

4. Τυπώστε στην οθόνη 53 ισαπέχοντα σημεία στο διάστημα $[-3.5, 6.5]$ (συμπεριλαμβανόμενων και των άκρων).

   Υπόδειξη: n ισαπέχοντα σημεία στο διάστημα $[a, b]$ έχουν άγνωστη απόσταση μεταξύ τους, έστω h. Το $x_1 = a$, το $x_2 = a + h$, το $x_3 = a + 2h$, ..., το $x_n = a + (n-1)h$. Αλλά πρέπει $x_n \equiv b$, άρα $h = (b-a)/(n-1)$.

5. Γράψτε κώδικα που να τυπώνει στην οθόνη 30 ισαπέχουσες τιμές της μαθηματικής συνάρτησης $f(x) = x(x^2 + 5\sin(x))$ στο διάστημα $[-5, 5]$. Τα άκρα να περιλαμβάνονται σε αυτές.

6. Κάποιος καταθέτει 1000 ευρώ σε ένα απλό τραπεζικό λογαριασμό. Η τράπεζα δίνει τόκο που παραμένει στο λογαριασμό, με ετήσιο επιτόκιο 1.3%. Γράψτε πρόγραμμα που να υπολογίζει πόσα χρήματα θα υπάρχουν στο λογαριασμό αυτό μετά από 15 χρόνια.

7. Σύμφωνα με την Εθνική Στατιστική Υπηρεσία, ο πληθυσμός της Ελλάδας κατά την απογραφή του 2011 ήταν 10815197. Εάν αυξάνεται σταθερά κατά 0.53% το χρόνο, γράψτε πρόγραμμα που να υπολογίζει σε πόσα χρόνια θα ξεπεράσει τα 15000000.

8. Βρείτε το μέγιστο κοινό διαιρέτη (ΜΚΔ) δύο ακέραιων αριθμών a,b. Χρησιμοποιήστε τον αλγόριθμο του Ευκλείδη[3]. Σύμφωνα με αυτόν, για δύο μη αρνητικούς ακέραιους αριθμούς a και b:

   - αν ισχύει $a < b$ εναλλάσσουμε τις τιμές τους.
   - αν ο $b = 0$ τότε ο a είναι ο ΜΚΔ.
   - αν ο $b > 0$ τότε επαναλαμβάνουμε τη διαδικασία χρησιμοποιώντας ως νέους ακέραιους τον b και το υπόλοιπο της διαίρεσης του a με τον b.

   Χρησιμοποιήστε τον αλγόριθμο για να βρείτε το μέγιστο κοινό διαιρέτη των αριθμών 135 και 680.

   Απάντηση: 5

[3] http://en.wikipedia.org/wiki/Euclidean_algorithm

9. Γράψτε πρόγραμμα που να ελέγχει αν ένας ακέραιος αριθμός είναι τέλειος. Τέλειος είναι ο αριθμός, του οποίου το άθροισμα των διαιρετών του είναι ίσο με το διπλάσιο του. (Π.χ. το 6 είναι τέλειος αριθμός, γιατί διαιρείται ακριβώς με τους αριθμούς 1, 2, 3, 6 και το άθροισμά τους είναι το $1+2+3+6 = 12 = 2\times 6$).

10. Γράψτε πρόγραμμα που να δέχεται ένα θετικό ακέραιο αριθμό με οποιοδήποτε πλήθος ψηφίων και να εμφανίζει τα ψηφία του.

11. Γράψτε πρόγραμμα που να τυπώνει τους πρώτους N όρους της ακολουθίας Fibonacci[4]:

$$f_{i+2} = f_{i+1} + f_i, \qquad i \geq 0, \text{ με } f_0 = 0, f_1 = 1\ .$$

    Η ακολουθία επομένως είναι: $0, 1, 1, 2, 3, 5, 8, \ldots$. Το κάθε στοιχείο μετά το δεύτερο είναι το άθροισμα των δύο προηγούμενών του. Το πλήθος n να δίνεται από το χρήστη. Να φροντίσετε ώστε το πρόγραμμά σας να μην το δέχεται αν δεν ισχύει $0 \leq n < 31$.

12. Γράψτε πρόγραμμα που να ελέγχει αν ένας ακέραιος αριθμός που θα τον δίνει ο χρήστης, είναι πρώτος.

    Υπόδειξη Ι: θα σας βοηθήσει ο ακόλουθος ορισμός για τους πρώτους αριθμούς:

    Κάθε θετικός ακέραιος αριθμός είναι πρώτος εκτός αν διαιρείται ακριβώς με κάποιο αριθμό εκτός από το 1 και τον εαυτό του.

    Υπόδειξη ΙΙ: Ψάξτε να βρείτε κάποιον θετικό ακέραιο που να διαιρεί ακριβώς (δηλαδή χωρίς υπόλοιπο) τον αριθμό εισόδου, εκτός από το 1 και τον εαυτό του. Μπορούμε να αποκλείσουμε όλους τους μεγαλύτερους του αριθμού εισόδου καθώς κανένας δεν θα τον διαιρεί ακριβώς.

13. Γράψτε πρόγραμμα που να ζητά από το χρήστη μια σειρά από θετικούς πραγματικούς αριθμούς. Το πλήθος τους δεν θα είναι γνωστό εκ των προτέρων αλλά το «διάβασμα» των τιμών θα σταματά όταν ο χρήστης δώσει αρνητικό αριθμό. Το πρόγραμμά σας να υπολογίζει το μέσο όρο αυτών των αριθμών.

14. Χρησιμοποιήστε τον αλγόριθμο του Gauss για την ημερομηνία του ορθόδοξου Πάσχα από την άσκηση 10 στη σελίδα 56, για να βρείτε

    (α΄) ποια χρονιά το Πάσχα έπεσε πιο νωρίς,

    (β΄) ποια χρονιά έπεσε πιο αργά,

    (γ΄) πόσες φορές έπεσε το Μάιο,

    (δ΄) ποιες χρονιές έπεφτε στις 18 Απριλίου,

    μεταξύ των ετών 1930 έως και πέρυσι.

[4] http://oeis.org/A000045

15. Χρησιμοποιήστε τον αλγόριθμο της άσκησης 8 στη σελίδα 68 για να υπολογίσετε πόσες ημέρες έχουν περάσει από την ημερομηνία γέννησής σας (δηλαδή, την ηλικία σας σε ημέρες).

16. Ο Μιχάλης γεννήθηκε στις 8/2/2015. Χρησιμοποιήστε τις πληροφορίες της άσκησης 8 της σελίδας 68 για να βρείτε την ημερομηνία που θα συμπληρώσει 17000 ημέρες ζωής.

17. Γράψτε πρόγραμμα που να υπολογίζει και να τυπώνει το διπλό παραγοντικό αριθμού που θα δίνει ο χρήστης. Το διπλό παραγοντικό (n!!) ορίζεται ως

$$n!! = \begin{cases} 1 \times 3 \times 5 \times \cdots \times (n-2) \times n & \text{αν το } n \text{ είναι περιττός,} \\ 2 \times 4 \times 6 \times \cdots \times (n-2) \times n & \text{αν το } n \text{ είναι άρτιος,} \\ 1 & \text{αν το } n \text{ είναι } 0. \end{cases}$$

    Φροντίστε ώστε το πρόγραμμά σας να μη δέχεται n μεγαλύτερο από 15.

18. Γράψτε κώδικες που να υπολογίζουν

    - το e^x εφαρμόζοντας τη σχέση

$$\mathrm{e}^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} \,,$$

    - το $\sin x$ εφαρμόζοντας τη σχέση

$$\sin x = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{(2k+1)!} \,,$$

    - το $\cos x$ εφαρμόζοντας τη σχέση

$$\cos x = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k}}{(2k)!} \,.$$

    Για τη διευκόλυνσή σας παρατηρήστε ότι ο κάθε όρος στα αθροίσματα προκύπτει από τον αμέσως προηγούμενο αν αυτός πολλαπλασιαστεί με κατάλληλη ποσότητα.

    Στα αθροίσματα να σταματάτε τον υπολογισμό τους όταν ο όρος που πρόκειται να προστεθεί είναι κατ' απόλυτη τιμή μικρότερος από 10^{-10}.

19. Σύμφωνα με θεώρημα του Gauss, για κάθε θετικό ακέραιο a ισχύει ότι

$$2a = n(n+1) + m(m+1) + k(k+1)$$

    όπου n,m,k μη αρνητικοί και όχι απαραίτητα διαφορετικοί ακέραιοι. Να γράψετε ένα πρόγραμμα που να διαβάζει από τον χρήστη ένα ακέραιο a, να

υπολογίζει όλες τις τριάδες n,m,k για αυτόν και να τις τυπώνει στην οθόνη. Οι τριάδες που προκύπτουν με εναλλαγή των n,m,k να παραλείπονται (δηλαδή τυπώστε αυτές για τις οποίες ισχύει $n \leq m \leq k$).

Δοκιμάστε το για τους αριθμούς 16 ($[0, 1, 5]$ ή $[0, 3, 4]$ ή $[2, 2, 4]$), 104 ($[2, 4, 13]$ ή ...), και 111 ($[1, 10, 10]$ ή $[0, 9, 11]$ ή ...).

20. Γράψτε πρόγραμμα που να επαληθεύει το Θεώρημα των τεσσάρων τετραγώνων του Lagrange[5]. Σύμφωνα με αυτό, κάθε θετικός ακέραιος αριθμός μπορεί να γραφεί ως άθροισμα τεσσάρων (ή λιγότερων) τετραγώνων ακεραίων αριθμών.

    Υπόδειξη: Δημιουργήστε τέσσερις βρόχους, ο ένας μέσα στον άλλο. Όταν το άθροισμα των τετραγώνων των μεταβλητών ελέγχου γίνει ίσο με το ζητούμενο αριθμό, τυπώστε τις μεταβλητές ελέγχου και συνεχίστε για την επόμενη τετράδα. Λάβετε υπόψη ότι κάποιες από τις μεταβλητές ελέγχου μπορεί να είναι 0 ή να είναι ίσες. Επιλέξτε κατάλληλα τα διαστήματα στα οποία αυτές παίρνουν τιμές.

21. Να επαληθεύσετε ότι

$$\sum_{n=1}^{\infty} \left[\frac{12}{n^2} \cos\left(\frac{9}{n\pi + \sqrt{(n\pi+3)(n\pi-3)}} \right) \right] = -\frac{\pi^2}{e^3}$$

    ως εξής: υπολογίσετε τα δύο μέλη της εξίσωσης χωριστά και τυπώστε αυτά καθώς και τη διαφορά τους (που θα πρέπει να πλησιάζει στο 0). Στον υπολογισμό του αθροίσματος δε θα πάρετε φυσικά άπειρους όρους· να σταματήσετε στον πρώτο όρο που είναι κατ' απόλυτη τιμή μικρότερος από 10^{-7}.

22. Ο *δυαδικός αλγόριθμος* για τον πολλαπλασιασμό δύο ακεραίων έχει ως εξής: σχηματίζουμε δύο στήλες με επικεφαλής τους δύο αριθμούς. Κάθε αριθμός της *πρώτης* στήλης είναι το ακέραιο μέρος (πηλίκο) της διαίρεσης με το 2 του αμέσως προηγούμενού του στη στήλη. Κάθε αριθμός της *δεύτερης* στήλης είναι το διπλάσιο του αμέσως προηγούμενού του στη στήλη. Οι διαιρέσεις/πολλαπλασιασμοί στις στήλες σταματούν όταν στην πρώτη εμφανιστεί ο αριθμός 0. Το γινόμενο των δύο αρχικών αριθμών είναι το άθροισμα των αριθμών της δεύτερης στήλης που αντιστοιχούν σε περιττό αριθμό στην πρώτη στήλη.

    Γράψτε κώδικα που θα δέχεται από το χρήστη δύο ακέραιους, θα υπολογίζει το γινόμενό τους με το συγκεκριμένο αλγόριθμο και θα το τυπώνει.

23. Γράψτε πρόγραμμα που να βρίσκει τις τριάδες διαδοχικών πρώτων αριθμών που διαφέρουν κατά έξι[6] (δηλαδή οι p, $p+6$, $p+12$ να είναι διαδοχικοί πρώτοι αριθμοί) και να τυπώνει το μικρότερο. Περιοριστείτε στους πρώτους που είναι μικρότεροι από 10000.

[5] http://mathworld.wolfram.com/LagrangesFour-SquareTheorem.html
[6] http://oeis.org/A047948

24. Τα κέρματα του ευρώ έχουν αξία 1 λεπτό, 2 λεπτά, 5 λεπτά, 10 λεπτά, 20 λεπτά, 50 λεπτά, 100 λεπτά (= 1€) και 200 λεπτά (= 2€).

    Ένα συγκεκριμένο ποσό μπορεί να σχηματιστεί με συνδυασμό διάφορων κερμάτων. Πόσοι είναι όλοι οι συνδυασμοί που έχουν αξία 300 λεπτών;

    Υπόδειξη: Προφανώς, κάθε συνδυασμός θα έχει το πολύ 300 κέρματα του ενός λεπτού, 150 κέρματα των δύο λεπτών, 60 κέρματα των 5 λεπτών κλπ. Σχηματίστε όλους τους δυνατούς συνδυασμούς και μετρήστε όσους έχουν αξία 300 λεπτών.

    Απάντηση: 471363

25. Γράψτε κώδικα που να παράγει 120 τυχαίους ακέραιους αριθμούς στο διάστημα $[-100, 100]$. Για την παραγωγή τους χρησιμοποιήστε τη συνάρτηση `std::rand()` (§2.12.1). Μετρήστε πόσοι από αυτούς είναι θετικοί και πόσοι αρνητικοί.

26. Από τα Μαθηματικά γνωρίζουμε ότι ισχύει

$$\pi = 3 + 2\sum_{k=1}^{\infty} \frac{k(5k+3)(2k-1)!k!}{2^{k-1}(3k+2)!} \; .$$

    Χρησιμοποιήστε την παραπάνω σχέση για να υπολογίσετε το π με ακρίβεια 10^{-6}· αυτό σημαίνει ότι στον υπολογισμό του αθροίσματος θα σταματήσετε στον πρώτο όρο που είναι μικρότερος από 10^{-6}. Συγκρίνετε το αποτέλεσμά σας με τη «σωστή» τιμή.

    Υπόδειξη: Στον υπολογισμό σας μπορείτε να βασιστείτε στο ότι ο κάθε όρος στο άθροισμα προκύπτει από τον προηγούμενο με πολλαπλασιασμό κατάλληλης ποσότητας.

27. Ο θετικός ακέραιος 65728 μπορεί να γραφεί ως άθροισμα δύο κύβων (ακέραιων υψωμένων στην τρίτη) με μόνο δύο τρόπους:

$$65728 = 12^3 + 40^3 = 31^3 + 33^3 \; .$$

    Το ίδιο ισχύει και για τον 64232:

$$64232 = 17^3 + 39^3 = 26^3 + 36^3 \; .$$

    Βρείτε το μικρότερο k που ικανοποιεί τη σχέση $k = i^3 + j^3$ με δύο (και μόνο δύο) διαφορετικά ζευγάρια i, j. Τα i, j, k είναι θετικοί ακέραιοι με $i \leq j < k$.

28. Πολλοί περιττοί ακέραιοι αριθμοί μπορούν να γραφτούν ως άθροισμα ενός πρώτου αριθμού και του διπλάσιου κάποιου τετραγώνου μη μηδενικού αριθμού:

$$3 \;\; = \;\; 1 + 2 \times 1^2$$

$$\begin{aligned} 5 &= 3 + 2 \times 1^2 \\ 9 &= 1 + 2 \times 2^2 = 7 + 2 \times 1^2 \\ 15 &= 7 + 2 \times 2^2 \\ 27 &= 19 + 2 \times 2^2 \\ \vdots &= \vdots \end{aligned}$$

Βρείτε τους πρώτους πέντε θετικούς περιττούς αριθμούς που ΔΕΝ είναι ίσοι με ένα τέτοιο άθροισμα.

Απάντηση: 17, 137, 227, 977, 1187

29. Ένας ακέραιος αριθμός με n ψηφία χαρακτηρίζεται ως παμψήφιος αν περιέχει όλα τα ψηφία από το 1 ως το n ακριβώς μία φορά. Π.χ. το 3142 είναι παμψήφιος τεσσάρων ψηφίων. Βρείτε τον μεγαλύτερο παμψήφιο τεσσάρων ψηφίων που είναι πρώτος.

    Επισήμανση: Ένας αριθμός χαρακτηρίζεται ως πρώτος αν διαιρείται ακριβώς μόνο με το 1 και τον εαυτό του.

    Απάντηση: 4231

30. Ο αριθμός 12 μπορεί να γραφεί ως γινόμενο ακεραίων με τις μορφές 2×6, 3×4, $2 \times 2 \times 3$. Οι αριθμοί σε κάθε γινόμενο αποτελούν τους διαιρέτες του αρχικού αριθμού. Στην τελευταία μορφή, οι διαιρέτες είναι πρώτοι αριθμοί (διαιρούνται ακριβώς μόνο από το 1 και τον εαυτό τους).

    Γράψτε πρόγραμμα που θα δέχεται ένα ακέραιο αριθμό από το χρήστη και θα τον αναλύει σε γινόμενο πρώτων διαιρετών. Το πρόγραμμα θα τυπώνει τους διαιρέτες σε μία γραμμή στην οθόνη, με ένα κενό μεταξύ τους. Έτσι, αν δώσουμε 12 θα πρέπει να τυπώσει: 2 2 3, ενώ αν δώσουμε πρώτο αριθμό, π.χ. 13, θα τυπώσει μόνο ένα διαιρέτη: 13.

31. **Υπόθεση Goldbach**. Αριθμός Goldbach λέγεται ένας άρτιος θετικός ακέραιος που μπορεί να γραφεί ως άθροισμα δύο περιττών αριθμών που είναι πρώτοι. Σύμφωνα με την υπόθεση του Goldbach, κάθε άρτιος αριθμός μεγαλύτερος του 4 είναι αριθμός Goldbach.

    Δείξτε ότι ισχύει για τους άρτιους ακέραιους στο διάστημα $[6, 10000]$.

Κεφάλαιο 5

Διανύσματα–Πίνακες–Δομές

5.1 Εισαγωγή

Κατά την υπολογιστική αντιμετώπιση ενός προβλήματος παρουσιάζεται πολύ συχνά η ανάγκη να αποθηκεύσουμε και να χειριστούμε ένα πλήθος ποσοτήτων, ίδιου ή διαφορετικού τύπου. Με βάση τους θεμελιώδεις τύπους που παρέχει μια γλώσσα προγραμματισμού μπορούν να οριστούν άλλοι, σύνθετοι τύποι, με κατάλληλο τρόπο ώστε να αναπαριστούν έννοιες του προβλήματός μας ή να ανταποκρίνονται σε ανάγκες του προγράμματός μας. Μια σύγχρονη γλώσσα προγραμματισμού παρέχει δομές κατάλληλες τουλάχιστον για την αποθήκευση και εύκολη προσπέλαση ομοειδών ποσοτήτων. Η C++ παρέχει, είτε ενσωματωμένα είτε μέσω της Standard Library, πληθώρα τέτοιων δομών, με διαφορετικές ιδιότητες η κάθε μία. Ενδεικτικά, στη C++ μπορούμε να αποθηκεύσουμε ποσότητες ίδιου τύπου με δυνατότητα *τυχαίας προσπέλασης* (δηλαδή, πρόσβασης σε οποιαδήποτε ποσότητα από αυτές σε ίσο χρόνο) ή ταχύτατης αναζήτησης. Μπορούμε επίσης να χρησιμοποιήσουμε δομές με δυνατότητα προσθήκης ή αφαίρεσης στοιχείων. Θα τις περιγράψουμε αναλυτικά στο Κεφάλαιο 11.

Στο τρέχον κεφάλαιο θα αναφερθούμε σε δύο δομές ομαδοποίησης όμοιων ποσοτήτων με δυνατότητα τυχαίας προσπέλασης: το `std::array<>` από το header `<array>` και το `std::vector<>` από το header `<vector>`. Σε σύγκριση με τις αντίστοιχες δομές που κληρονομήθηκαν από τη C—το ενσωματωμένο στατικό διάνυσμα και το δυναμικό διάνυσμα, έχουν όλες τις δυνατότητές τους, πολλά πλεονεκτήματα και δεν υστερούν από αυτές σε ταχύτητα. Καθώς θα δείτε σε κώδικες να χρησιμοποιούνται οι παλαιές δομές, θα τις περιγράψουμε συνοπτικά. Καλό είναι να μην βασίζεται νέος κώδικας σε αυτές.

Η C++ παρέχει επιπλέον τη δυνατότητα να ομαδοποιήσουμε ποσότητες διαφορετικού (ή και ίδιου) τύπου χρησιμοποιώντας τη *δομή* (`struct`), που θα δούμε

παρακάτω, και την επέκτασή της, την *κλάση* (`class`), που θα αναπτύξουμε στο Κεφάλαιο 14.

5.2 Διάνυσμα

Έστω ότι στον κώδικά μας χρειαζόμαστε τις πέντε πρώτες δυνάμεις του 2. Μπορούμε να ορίσουμε ισάριθμες ανεξάρτητες σταθερές ποσότητες για να τις αποθηκεύσουμε:

```
int constexpr po2_0{1};
int constexpr po2_1{2};
int constexpr po2_2{4};
int constexpr po2_3{8};
int constexpr po2_4{16};
```

Αν θελήσουμε να τις τυπώσουμε στην οθόνη θα πρέπει να δώσουμε τις παρακάτω εντολές:

```
std::cout << po2_0 << '\n';
std::cout << po2_1 << '\n';
std::cout << po2_2 << '\n';
std::cout << po2_3 << '\n';
std::cout << po2_4 << '\n';
```

Τι θα κάναμε αν χρειαζόμαστε τις τριάντα πρώτες δυνάμεις; δεν είναι πρακτικό να κάνουμε τριάντα δηλώσεις ούτε είναι εύχρηστες ισάριθμες *ανεξάρτητες* ποσότητες. Προφανώς χρειαζόμαστε κάποια εντολή επανάληψης.

Στην προσπάθεια να εκτελέσουμε τις παραπάνω εντολές με βρόχο, θα μπορούσε να σκεφτεί κανείς ότι ο κώδικας

```
for (int i{0}; i <=4; ++i) {
  std::cout << po2_i << '\n';
}
```

το επιτυγχάνει. Ο κώδικας δεν έχει λάθος στη σύνταξη, προσέξτε όμως την εντολή που επαναλαμβάνεται: είναι η εκτύπωση της (μίας) ποσότητας με όνομα po2_i και όχι των po2_0, po2_1, κλπ. Ο μεταγλωττιστής *δεν κάνει* αντικατάσταση του i στη λέξη po2_i. Δεν υπάρχει η δυνατότητα να φέρουμε τις εντολές που αφορούν ανεξάρτητες ποσότητες σε κατάλληλη μορφή για ένταξη σε εντολή επανάληψης· πρέπει να τις γράψουμε μία–μία. Το ίδιο ισχύει και για τον ορισμό τέτοιων ποσοτήτων· δεν μπορεί να απλοποιηθεί ιδιαίτερα.

Η C++ μας δίνει τη δυνατότητα να δηλώσουμε μια ομάδα σχετιζόμενων ποσοτήτων με μία εντολή, ως *ένα* αντικείμενο, και να τη χειριζόμαστε με απλό τρόπο. Η γλώσσα παρέχει για γενική χρήση δύο δομές· μπορούμε να επιλέξουμε μεταξύ του `std::array<>` και του `std::vector<>`. Οι δυο τους διαφοροποιούνται ως προς το στάδιο δημιουργίας τους (κατά τη μεταγλώττιση ή κατά την εκτέλεση του προγράμματος) και ως προς τη δυνατότητα μεταβολής (προσθήκης ή αφαίρεσης) στοιχείων.

Συγκεκριμένα, ένα `std::array<>` δημιουργείται κατά τη μεταγλώττιση και δεν επιτρέπεται η αλλαγή του πλήθους των στοιχείων μετά τη δημιουργία του, ενώ ένα `std::vector<>` δημιουργείται κατά την εκτέλεση και επιτρέπεται η προσθήκη ή αφαίρεση στοιχείων σε αυτό. Μια τεχνική διαφορά που παρουσιάζεται επίσης, είναι ότι μπορεί να είναι πιο γρήγορη η πρόσβαση των στοιχείων σε `std::array<>` παρά σε `std::vector<>`.

5.2.1 Διάνυσμα με γνωστή και σταθερή διάσταση (στατικό)

Η C++ μας δίνει τη δυνατότητα να δηλώσουμε μια ομάδα ποσοτήτων *ίδιου* τύπου, με πλήθος *γνωστό κατά τη μεταγλώττιση* και *σταθερό σε όλο το πρόγραμμα*, χρησιμοποιώντας το `std::array<>` από το header `<array>`.

Δήλωση

Η δήλωση έχει τη γενική μορφή

```
std::array<τύπος,πλήθος> όνομα_μεταβλητής;
```

Ο «τύπος» μπορεί να είναι οποιοσδήποτε (όχι μόνο θεμελιώδης). Το «πλήθος» επιτρέπεται να είναι

- μια ακέραιη *σταθερά*,
- μια ακέραιη *σταθερή ποσότητα* (δηλωμένη ως **`constexpr`**),
- μια έκφραση, με πιθανή κλήση συναρτήσεων **`constexpr`** (§7.12), που έχει ακέραιη τιμή, γνωστή κατά τη μεταγλώττιση.

Με την παραπάνω εντολή δημιουργούμε *μία* ποσότητα, σύνθετη: αποτελείται από συγκεκριμένο πλήθος στοιχείων, συγκεκριμένου τύπου. Είναι ένα διάνυσμα που έχει τα στοιχεία του στη σειρά και έχει δυνατότητα τυχαίας προσπέλασης σε αυτά, χρειάζεται, δηλαδή, τον ίδιο χρόνο για την πρόσβαση σε *οποιοδήποτε* από αυτά.

Παράδειγμα

Έστω ότι σε κάποιο πρόγραμμά μας χρειάζεται να χειριστούμε τις μέσες θερμοκρασίες κάθε ημέρας, σε ένα τόπο, για μια συγκεκριμένη εβδομάδα. Μπορούμε να δηλώσουμε το διάνυσμα με όνομα temper ως εξής:

```
std::array<double,7> temper;
```

Εννοείται ότι θα έχουμε συμπεριλάβει στην αρχή του κώδικά μας το header `<array>`, με κατάλληλη εντολή **`#include`**.

Η δήλωση ενός διανύσματος μπορεί να γίνει ταυτόχρονα με άλλα διανύσματα, ίδιας διάστασης και τύπου στοιχείων. Π.χ. η δήλωση

```
int constexpr n{5};
std::array<double,n> a, b;
```

δημιουργεί δύο πραγματικά διανύσματα 5 στοιχείων με ονόματα a,b.

Στη δήλωση όπως παρουσιάστηκε εδώ, τα στοιχεία του διανύσματος, αν είναι θεμελιώδους τύπου, έχουν *απροσδιόριστη τιμή*. Αν είναι τύπου που ορίζεται στη Standard Library (εκτός του `std::array<>`) ή τύπου που έχει δημιουργηθεί από τον προγραμματιστή με προσδιορισμένο default constructor, αποκτούν την προκαθορισμένη τους τιμή.

Αρχικοποίηση

Αν επιθυμούμε να δημιουργήσουμε διάνυσμα και ταυτόχρονα να αποδώσουμε συγκεκριμένες τιμές στα στοιχεία του, πρέπει να παραθέσουμε τις τιμές στη σειρά με τη μορφή λίστας κατά τη δήλωση: περικλείουμε δηλαδή, εντός αγκίστρων '{}' ποσότητες με τύπο ίδιο με τα στοιχεία του διανύσματος (ή τύπο που να μπορεί να μετατραπεί σε αυτόν). Επιπλέον, τα στοιχεία της λίστας δεν πρέπει να είναι περισσότερα από τη διάσταση του διανύσματος. Αν παρατίθενται λιγότερα, τα υπόλοιπα θεωρούνται 0 (μετατρεπόμενο στον αντίστοιχο τύπο). Π.χ.

```
std::array<char,4> letter{'a', 'b', 'c', 'd'};
// letter[0] = 'a', letter[1] = 'b', letter[2] = 'c', letter[3] = 'd'

std::array<int,5> prime{2,3,5,7}; // prime[4] = 0
std::array<double,10> a{};  // a[0]=a[1]=...=a[9] = 0.0
```

Εναλλακτικά, μπορούμε να δημιουργήσουμε διάνυσμα ως αντίγραφο άλλου διανύσματος με τον εξής τρόπο:

```
std::array<int,5> a{1,2,5,7,8};
std::array<int,5> b{a};
auto c = a;
```

Στη δεύτερη εντολή δημιουργούμε το διάνυσμα b αντιγράφοντας όλα τα στοιχεία από άλλο διάνυσμα a, ίδιου τύπου και πλήθους στοιχείων. Στην τρίτη, δημιουργούμε το c με ίδιο τύπο, πλήθος στοιχείων και τιμές όπως ο a.

Πρόσβαση στα στοιχεία

Πρόσβαση στα στοιχεία ενός διανύσματος έχουμε αν βάλουμε σε αγκύλες μετά το όνομα του διανύσματος, ένα ακέραιο *μεταξύ* 0 *και* $D-1$, όπου D το πλήθος στοιχείων (n *διάσταση*). Το πρώτο, δηλαδή, στοιχείο του διανύσματος είναι στη θέση 0, το δεύτερο στην 1, το τελευταίο στη $D-1$. Προσέξτε ότι αν δώσουμε ακέραιο εκτός των ορίων του πίνακα, δηλαδή μικρότερο από το 0 ή μεγαλύτερο από $D-1$, *δε* θα διαγνωστεί ως λάθος από τον compiler.

Για το διάνυσμα που δηλώνεται ως

```
std::array<double,7> temper;
```

το πρώτο στοιχείο είναι το `temper[0]`, το δεύτερο είναι το `temper[1]`, ενώ το τελευταίο είναι το `temper[6]`. Με αυτά τα «ονόματα» συμμετέχουν σε εκφράσεις και σε αυτά τα ονόματα γίνεται η εκχώρηση τιμής.

Η εκχώρηση τιμών στα στοιχεία του μπορεί να γίνει, μεταξύ άλλων τρόπων, ως εξής:

- με ξεχωριστές εντολές εκχώρησης

  ```
  temper[4] = 13.6;
  temper[5] = 15.0;
  temper[6] = 16.5;
  ```

- με ανάγνωση από το πληκτρολόγιο (ή αρχείο)

  ```
  std::cin >> temper[0];
  std::cin >> temper[1];
  ```

- με εκχώρηση άλλου array ίδιου πλήθους στοιχείων

  ```
  std::array<double,7> temp;
   ... // give values to temp
  temper = temp;
  ```

- με κλήση της συνάρτησης–μέλους `fill()` με όρισμα συγκεκριμένη τιμή. Η κλήση

  ```
  temper.fill(12.6);
  ```

  εκχωρεί σε όλα τα στοιχεία του temper την τιμή 12.6.

Με βάση τα παραπάνω, αν θέλουμε να υπολογίσουμε τον μέσο όρο των τριών πρώτων στοιχείων του temper πρέπει να γράψουμε την εντολή

```
double mo3 { (temper[0]+temper[1]+temper[2])/3.0 };
```

Η εκτύπωση των τιμών των στοιχείων γίνεται με εντολές σαν την

```
std::cout << "The␣temperature␣on␣Wednesday␣was␣"
          << temper[3] << "␣deg.␣Celsius\n";
```

Ο `std::array<>` είναι container της Standard Library και μπορούν να χρησιμοποιηθούν σε αυτόν όλες οι δυνατότητες που παρέχει αυτή (π.χ. αλγόριθμοι). Εσωτερικά, είναι «κέλυφος» για το ενσωματωμένο διάνυσμα που κληρονομήθηκε από τη C. Σε παλαιότερους κώδικες θα δείτε να χρησιμοποιείται αυτό απευθείας. Χρειάζεται συνεπώς να το περιγράψουμε συνοπτικά παρακάτω. Όμως, δεν υπάρχει κανένας λόγος να χρησιμοποιούμε απευθείας το ενσωματωμένο διάνυσμα. Ο `std::array<>` έχει μόνο πλεονεκτήματα έναντι αυτού.

5.2.2 Ενσωματωμένο στατικό διάνυσμα

Η δήλωση ενσωματωμένου διανύσματος με αρχικές τιμές, σύμφωνα με το μηχανισμό που κληρονομήθηκε από τη C, έχει τη γενική μορφή

τύπος όνομα[πλήθος] {λίστα_τιμών};

Για τον ορισμό αυτό ισχύουν όσα έχουμε αναφέρει για το `std::array<>`. Έτσι, το πλήθος πρέπει να είναι ακέραιο, γνωστό κατά τη μεταγλώττιση, και η λίστα αρχικοποίησης μπορεί να παραλείπεται οπότε τα στοιχεία του πίνακα έχουν απροσδιόριστη τιμή (αν είναι θεμελιώδους τύπου) ή 0 αν ο τύπος τους ορίζεται στη Standard Library.

Αν παραθέτουμε λίστα αρχικών τιμών στη δήλωση, μπορούμε να παραλείψουμε να προσδιορίσουμε ρητά το πλήθος. Θα υπολογιστεί από τον αριθμό των στοιχείων της λίστας και το διάνυσμα θα δημιουργηθεί με αυτό το πλήθος:

τύπος όνομα[] {τιμή0, τιμή1, τιμή2, τιμή3}; // 4 στοιχεία στο όνομα

Αν προσδιοριστεί και η λίστα και το πλήθος, θα πρέπει η λίστα να έχει το πολύ τόσα στοιχεία όσα και το διάνυσμα. Αν έχει λιγότερα, συμπληρώνονται με το 0.

Σύμφωνα με τα παραπάνω, η δήλωση της μεταβλητής a ως ενσωματωμένο διάνυσμα για 15 πραγματικούς γίνεται ως εξής

```
double a[15];
```

Η δήλωση με απόδοση τεσσάρων ακέραιων αρχικών τιμών στον πίνακα με όνομα b είναι η

```
int b[] {3,4,9,12};
```

Ό,τι αναφέραμε για την *ατομική* (όχι ως σύνολο) προσπέλαση των στοιχείων στο `std::array<>` ισχύει και για το ενσωματωμένο διάνυσμα. Δεν υπάρχει η δυνατότητα εκχώρησης ενός διανύσματος ή μιας λίστας σε διάνυσμα.

Ενσωματωμένο διάνυσμα και δείκτες

Η αριθμητική δεικτών (§2.18.2) είναι χρήσιμη στην περίπτωση που εκχωρήσουμε σε ένα δείκτη τη διεύθυνση ενός στοιχείου ενσωματωμένου διανύσματος[1]. Τότε, η μετακίνηση κατά πολλαπλάσια του μεγέθους του τύπου μας μεταφέρει σε επόμενο ή προηγούμενο στοιχείο του διανύσματος:

```
int a[10];
int * p{&a[3]};
int * q{p + 2};    // q == &a[5]
```

Μάλιστα, αν ισχύει

[1] ή `std::array<>`, `std::vector<>` ή οποιασδήποτε άλλης δομής αποθηκεύει τα στοιχεία σε *συνεχόμενες* θέσεις μνήμης.

```
int a[10];
int * p{&a[0]};
```

τότε η έκφραση `*(p+i)` είναι απόλυτα ισοδύναμη με την `a[i]` και, βέβαια, ισχύει ότι `p+i == &a[i]`. Προσέξτε ότι τίποτε δεν εμποδίζει να προσπελάσουμε στοιχείο που δεν ανήκει στο διάνυσμα· αυτό αποτελεί ένα πολύ συνηθισμένο λάθος για αρχάριους προγραμματιστές.

Σημειώστε ότι το όνομα ενός ενσωματωμένου διανύσματος έχει τιμή, τη διεύθυνση του πρώτου στοιχείου του. Επομένως η έκφραση `a[i]` είναι απόλυτα ισοδύναμη με την `*(a+i)`. Επίσης, επιτρέπεται να χρησιμοποιήσουμε τη διεύθυνση οποιουδήποτε στοιχείου ενός διανύσματος *καθώς και τη διεύθυνση του πρώτου στοιχείου μετά το τέλος του.*

Παράδειγμα

Ο κώδικας

```
double b[10];
double * p{b};
for (int i{0}; i < 10; ++i) {
   *p = 1.0;
   ++p;
}
```

εκχωρεί τιμές σε ένα ενσωματωμένο στατικό διάυνσμα χρησιμοποιώντας δείκτη για να το διατρέξει. Ισοδύναμος με τον παραπάνω κώδικα είναι ο

```
double b[10];
for (auto p = b; p != b+10; ++p) {
   *p = 1.0;
}
```

Παρατήρηση

Η δράση του τελεστή **sizeof** (§2.11.1) σε ενσωματωμένο διάνυσμα, επιστρέφει το μέγεθος σε bytes ολόκληρου του διανύσματος, δηλαδή, το πλήθος των στοιχείων επί το μέγεθος ενός στοιχείου. Έτσι στον παρακάτω κώδικα

```
double a[13];
int k { sizeof(a) / sizeof(a[0]) };  // k == 13
```

δρώντας κατάλληλα τον τελεστή υπολογίζεται το πλήθος των στοιχείων του διανύσματος.

5.2.3 Διάνυσμα με άγνωστη ή μεταβλητή διάσταση (δυναμικό)

Αν επιθυμούμε να δημιουργήσουμε ένα διάνυσμα

- με πλήθος στοιχείων που θα γίνει γνωστό *κατά την εκτέλεση του προγράμματος* και όχι πιο πριν, ή/και
- με δυνατότητα προσθήκης ή αφαίρεσης στοιχείων,

θα πρέπει να χρησιμοποιήσουμε άλλο container της Standard Library και όχι τον `std::array<>`. Ο `std::vector<>` από το header `<vector>` είναι ο πλησιέστερος στον `std::array<>` ως προς τα χαρακτηριστικά του.

Παράδειγμα

Έστω ότι θέλουμε να αποθηκεύσουμε ένα πλήθος πραγματικών αριθμών που θα δίνει ο χρήστης. Προφανώς θα χρειαστεί διάνυσμα αλλά η διάστασή του (το πλήθος των στοιχείων του) *δεν είναι γνωστή κατά τη μεταγλώττιση ή όταν γράφουμε τον κώδικα*. Μπορεί να δοθεί «εξωτερικά», από το χρήστη, πριν αρχίσει την εισαγωγή αριθμών. Ο σχετικός κώδικας θα είναι

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

int
main()
{
   int D;
   std::cin >> D; // get dimension
   std::vector<double> v(D);
   ...
}
```

Προσέξτε στο παράδειγμα το διαφορετικό τρόπο ορισμού του `std::vector<>` σε σύγκριση με το `std::array<>`. Η γενική μορφή της δήλωσης είναι

`std::vector`<τύπος> όνομα_μεταβλητής(πλήθος);

Το «πλήθος» μπορεί να είναι σταθερή ή μεταβλητή ποσότητα ή έκφραση, προφανώς με ακέραια τιμή.

Αν δεν ορίσουμε συγκεκριμένη τιμή (με λίστα ή τους άλλους τρόπους που θα δούμε στο Κεφάλαιο 11), τα στοιχεία ενός `std::vector<>` αποκτούν

- την τιμή 0 (αφού μετατραπεί στον κατάλληλο τύπο) αν είναι θεμελιώδους τύπου,
- όποια τιμή έχει προκαθορίσει η Standard Library ή ο προγραμματιστής που δημιούργησε τον τύπο τους, μέσω του default constructor.

Κατά τα λοιπά, η χρήση ενός `std::vector<>` με όνομα v είναι ακριβώς όμοια με το `std::array<>`. Το πρώτο στοιχείο είναι το `v[0]`, το δεύτερο είναι το `v[1]`, κλπ.

Τις επιπλέον δυνατότητες που μας παρέχει η κλάση `std::vector<>` (ανάμεσά τους τη δυνατότητα προσθήκης/αφαίρεσης στοιχείων) θα τις αναλύσουμε στο §11.5.2.

Επιτρέπεται η δήλωση `std::vector<>` με πλήθος στοιχείων γνωστό κατά τη μεταγλώττιση. Όμως, γενικά υστερεί έναντι του `std::array<>` σε ταχύτητα πρόσβασης στα στοιχεία. Από την άλλη, αν πρόκειται το πλήθος στοιχείων να μεταβληθεί κατά την εξέλιξη του προγράμματος, το `std::array<>` δεν μπορεί να χρησιμοποιηθεί. Προφανώς, περιορίζεται η επιλογή μας στο `std::vector<>` (ή άλλο container).

5.2.4 Ενσωματωμένο δυναμικό διάνυσμα

Ο μηχανισμός που κληρονομήθηκε από τη C για τη δημιουργία διανυσμάτων κατά την εκτέλεση του προγράμματος, βασίζεται στους δείκτες και σε συναρτήσεις δέσμευσης μνήμης (`std::malloc()`, `std::calloc()`, `std::realloc()`) και αποδέσμευσης μνήμης (`std::free()`). Οι συναρτήσεις αυτές παρέχονται από το header `<cstdlib>`. Θα αναφερθούμε συνοπτικά μόνο στις βασικές συναρτήσεις, `malloc()/free()`, καθαρά για λόγους κατανόησης παλαιότερων κωδίκων, καθώς πρέπει να αποφεύγεται πλέον η χρήση του συγκεκριμένου μηχανισμού.

Αν επιθυμούμε να δεσμεύσουμε κατά τη διάρκεια εκτέλεσης του προγράμματος, συνεχόμενο χώρο μνήμης για D στοιχεία, π.χ. πραγματικά, μπορούμε να κάνουμε το εξής: καλούμε τη συνάρτηση `std::malloc()` με όρισμα το πλήθος των bytes που επιθυμούμε να δεσμεύσουμε. Η συνάρτηση επιστρέφει δείκτη σε στην αρχή του χώρου αυτού στη μνήμη αλλά με τη μορφή δείκτη σε **`void`**. Για να χρησιμοποιήσουμε το νέο χώρο μνήμης πρέπει να μετατρέψουμε ρητά με **`static_cast<>`** αυτό το δείκτη σε δείκτη στον κατάλληλο τύπο. Στο τέλος, αφού ολοκληρώσουμε τη χρήση του νέου διανύσματος, πρέπει να αποδεσμεύσουμε τη μνήμη του *ρητά*, με την κλήση της συνάρτησης `std::free()`. Αυτή δέχεται ως όρισμα το δείκτη στην αρχή του χώρου αυτού στη μνήμη, τον μετατρέπει αυτόματα σε **`void *`**, ελευθερώνει τη μνήμη και δεν επιστρέφει τίποτε. Συνολικά, πρέπει να γράψουμε κάτι σαν

```
#include <cstdlib>

int
main()
{
   int D;
   std::cin >> D; // get dimension
   auto p = std::malloc(D * sizeof(double));
   double * v{static_cast<double *>(p)};

   v[0] = ...
   v[1] = ...
   ...
   v[D-1] = ...
```

```
   std::free(p);
}
```

Παρατηρήστε ότι στο όρισμα της `std::malloc()` χρησιμοποιήσαμε τον τελεστή **`sizeof`** για να βρούμε το μέγεθος σε bytes των στοιχείων που θα αποθηκεύει το διάνυσμα. Το αποτέλεσμά της εκχωρήθηκε σε δείκτη σε **`double`** με ρητή μετατροπή. Το νέο διάνυσμα πλέον μπορεί να χρησιμοποιηθεί όπως και ένα ενσωματωμένο στατικό διάνυσμα. Στο τέλος, καλείται η `std::free()` ώστε να αποδοθεί ξανά στο λειτουργικό σύστημα η δεσμεύμενη μνήμη. Η αποδέσμευση είναι πολύ βασικό να γίνεται όταν πλέον δεν χρειάζεται το διάνυσμα, καθώς, αν δεν γίνει ρητά από τον προγραμματιστή, η δεσμευμένη μνήμη «χάνεται» για όλη τη διάρκεια εκτέλεσης του προγράμματος.

Η C++ απλοποίησε κάπως τον μηχανισμό που παρουσιάστηκε, με την εισαγωγή των τελεστών **`new/delete[]`**. Ο κώδικας του παραδείγματος μπορεί να γραφεί

```
int
main()
{
   int D;
   std::cin >> D; // get dimension

   double * v{new double[D]};

   v[0] = ...
   v[1] = ...
   ...
   v[D-1] = ...

   delete[] v;
}
```

5.3 Πίνακας

Πολύ συχνά σε επιστημονικούς κώδικες, εμφανίζεται η ανάγκη να αναπαραστήσουμε ποσότητες σε 2 ή 3 διαστάσεις, π.χ. σε ένα καρτεσιανό πλέγμα.

Παράδειγμα

Έστω ότι θέλουμε να επεξεργαστούμε τις θερμοκρασίες ενός τόπου για κάθε ημέρα συγκεκριμένου έτους. Αυτές μπορεί να μας δίνονται με την παρακάτω

μορφή

Ημέρα	Θερμοκρασία (°C)
1	5.0
2	7.5
3	6.4
⋮	⋮
157	19.1
158	21.4
⋮	⋮
364	4.5
365	7.0

Οι ημέρες αριθμούνται από το 1 έως το 365.

Παρατηρήστε ότι για να προσδιορίσουμε μια συγκεκριμένη θερμοκρασία πρέπει να γνωρίζουμε σε ποια γραμμή είναι, δηλαδή σε ποια ημέρα αναφερόμαστε. Με άλλα λόγια, η πληροφορία μας (οι θερμοκρασίες) παραμετροποιείται με *ένα* ακέραιο αριθμό. Επακόλουθο είναι ότι η αποθήκευση των τιμών στο πρόγραμμά μας θα γίνει σε διάνυσμα.

Εναλλακτικά, οι θερμοκρασίες μπορεί να μας δίνονται στην ακόλουθη μορφή

	1	2	...	14	15	...	30	31
1	5.0	7.5	...	8.3	9.2	...	12.3	11.0
2	4.5	6.5	...	7.0	9.0	...		
⋮	⋮	⋮		⋮	⋮		⋮	⋮
6	25.0	27.5	...	28.3	29.2	...	27.3	
7	26.0	27.0	...	26.0	31.0	...	32.5	33.0
⋮	⋮	⋮		⋮	⋮		⋮	⋮
11	5.0	6.5	...	7.6	10.0	...	11.0	
12	5.0	7.5	...	8.5	9.5	...	12.0	12.5

Η πρώτη γραμμή παραθέτει τις ημέρες ενός μήνα ενώ η πρώτη στήλη παραθέτει τους μήνες.

Παρατηρήστε ότι για να προσδιορίσουμε μια συγκεκριμένη θερμοκρασία πρέπει να καθορίσουμε *δύο* ακέραιους αριθμούς: τον μήνα (γραμμή) και την ημέρα (στήλη). Στο πρόβλημά μας, η πληροφορία οργανώνεται σε δύο διαστάσεις. Στο πρόγραμμά μας, θα θέλαμε να έχουμε τη δυνατότητα να αποθηκεύσουμε τις θερμοκρασίες σε πίνακα δύο διαστάσεων.

5.3.1 Πίνακας με γνωστές και σταθερές διαστάσεις (στατικός)

Ένας τρόπος για να δημιουργήσουμε ένα διδιάστατο πίνακα είναι να ορίσουμε ως τύπο στοιχείων ενός `std::array<>` άλλο `std::array<>`:

```
std::array<std::array<τύπος,διάσταση2>,διάσταση1> name;
```

Οι ακέραιες ποσότητες διάσταση1 και διάσταση2, δηλαδή το πλήθος γραμμών και στηλών αντίστοιχα, πρέπει να είναι *γνωστές κατά τη μεταγλώττιση* και σταθερές για όλο το πρόγραμμα. Παρατηρήστε ότι ο διδιάστατος πίνακας είναι στην πραγματικότητα ένα array με στοιχεία άλλα arrays.

Παραδείγματος χάριν, ένας πραγματικός διδιάστατος πίνακας με 12 γραμμές («μήνες») και 31 στήλες («ημέρες») με όνομα tempr, μπορεί να δηλωθεί ως εξής

```
std::array<std::array<double,31>,12> tempr;
```

Εννοείται ότι έχουμε συμπεριλάβει το header `<array>` πιο πριν.

Η προσπέλαση των στοιχείων γίνεται βάζοντας μετά το όνομα του πίνακα δύο ακέραιους, τον καθένα εντός αγκυλών: η θερμοκρασία στις 14 Απριλίου θα αποθηκευτεί στη θέση `tempr[3][13]`, καθώς *η αρίθμηση των γραμμών και στηλών ξεκινά από το* 0[2]. Παρατηρήστε ότι *δεν είναι σωστός* ο προσδιορισμός του στοιχείου της θέσης (i, j) με τον τρόπο `tempr[i,j]`, όπως ίσως θα περίμενε κανείς. Η σύνταξη δεν είναι λάθος αλλά η ποσότητα `tempr[i,j]` είναι η γραμμή j του διανύσματος, ένα `std::array<double>` (γιατί;).

Αν επιθυμούμε να δώσουμε αρχικές τιμές στα στοιχεία, τις παραθέτουμε *κατά γραμμές*, διαδοχικά:

```
std::array<std::array<int,3>,2> b { 0, 1, 2, 3, 4, 5 };
```

Με τη συγκεκριμένη εντολή δημιουργείται ο διδιάστατος πίνακας

$$b = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{pmatrix} .$$

Όσα αναφέραμε στην παράγραφο §5.2.1 για τα διανύσματα που υλοποιούνται με `std::array<>` ισχύουν και για τους πίνακες. Έτσι, για παράδειγμα, μπορούμε να αντιγράψουμε με μία εντολή εκχώρησης ένα διδιάστατο πίνακα σε άλλο, όμοιό του.

Πίνακες περισσότερων διαστάσεων ορίζονται ανάλογα.

Στην §5.3.3 θα παρουσιάσουμε ένα πιο ευέλικτο, και προτιμότερο, τρόπο δημιουργίας ενός πολυδιάστατου πίνακα με τη χρήση διανύσματος.

5.3.2 Ενσωματωμένος στατικός πίνακας

Κατ᾽ αντιστοιχία του ενσωματωμένου διανύσματος, μπορούμε να ορίσουμε ένα πίνακα στη C++, με τον τρόπο που έχει κληρονομηθεί από τη C. Συμβολικά, ένας διδιάστατος πίνακας ορίζεται με την δήλωση

```
τύπος όνομα[διάσταση1][διάσταση2];
```

[2] Θα πρέπει να προσέχουμε βέβαια να μην δώσουμε για δείκτες συνδυασμούς που δεν έχουν νόημα, π.χ. [1][29] που αντιστοιχεί στις 30 Φεβρουαρίου ή ή [5][30] που υποδηλώνει την 31η Ιουνίου.

Οι ακέραιες ποσότητες διάσταση1 και διάσταση2, δηλαδή το πλήθος γραμμών και στηλών αντίστοιχα, πρέπει να είναι *γνωστές κατά τη μεταγλώττιση*. Παρατηρήστε ότι ο διδιάστατος πίνακας είναι στην πραγματικότητα ένα διάνυσμα με στοιχεία άλλα διανύσματα, ίδιας διάστασης.

Ένας πραγματικός διδιάστατος πίνακας με 12 γραμμές («μήνες») και 31 στήλες («ημέρες») είναι ο

```
double tempr[12][31];
```

Η προσπέλαση των στοιχείων γίνεται βάζοντας μετά το όνομα του πίνακα δύο ακέραιους, τον καθένα εντός αγκυλών:

```
tempr[3][13] = 16.5;
```

Παρατηρήστε ότι η σύνταξη `tempr[i,j]` προσδιορίζει το στοιχείο $(0, j)$ (γιατί;) και όχι το (i, j) που θα επιθυμούσαμε.

Αν επιθυμούμε να δώσουμε αρχικές τιμές στα στοιχεία, παραθέτουμε σε λίστα τις λίστες των στοιχείων κάθε γραμμής:

```
int b[2][3] = { {0, 1, 2}, {3, 4, 5} };
```

Αν επιθυμούμε, μπορούμε να παραλείψουμε τα «εσωτερικά» άγκιστρα, χάνοντας όμως τη δυνατότητα να συμπληρώνει ο compiler όσα στοιχεία δεν προσδιορίζουμε (δίνοντας σε αυτά την τιμή 0).

Πίνακες περισσότερων διαστάσεων ορίζονται ανάλογα: περικλείουμε σε αγκύλες την κάθε διάσταση. Πίνακες πολυδιάστατοι, *με γνωστές διαστάσεις κατά τη μεταγλώττιση του κώδικα*, μπορούν να ορίζονται με τον παραπάνω τρόπο. Είναι πιο απλή η δήλωσή τους από την περίπτωση που χρησιμοποιούσαμε `std::array<>` αλλά οι ενσωματωμένοι πίνακες είναι λιγότερο εύχρηστοι. Καλό είναι να μην χρησιμοποιούνται πλέον.

5.3.3 Πίνακας με άγνωστες ή μεταβλητές διαστάσεις (δυναμικός)

Ας υποθέσουμε ότι έχουμε ένα πλέγμα στις δύο διαστάσεις με 3 γραμμές και 4 στήλες. Οι γραμμές έχουν αρίθμηση $0, 1, 2$ και οι στήλες $0, 1, 2, 3$. Κάθε θέση στο πλέγμα μπορεί να προσδιοριστεί με δύο ακέραιους αριθμούς· ο πρώτος θα καθορίζει τη γραμμή και ο δεύτερος τη στήλη.

Εναλλακτικά, μπορούμε να προσδιορίσουμε μονοσήμαντα μια θέση στο πλέγμα χρησιμοποιώντας *ένα* ακέραιο αριθμό: ξεκινούμε την αρίθμηση από το 0 για τη θέση $(0, 0)$ και προχωρούμε κατά στήλες έτσι ώστε διαδοχικά στοιχεία στην ίδια στήλη να έχουν διαδοχική αρίθμηση (5.1). Παρατηρήστε ότι η θέση με συντεταγμένες (i, j) στο πλέγμα έχει αριθμηθεί με την τιμή $i + 3 * j$. Γενικότερα, οι θέσεις ενός διδιάστατου πίνακα A με M γραμμές και N στήλες έχουν αρίθμηση κατά στήλες την τιμή $k = i + M * j$. Μπορούμε φυσικά να επιλέξουμε αρίθμηση κατά γραμμές οπότε ο αριθμός της θέσης (i, j) είναι ο $k = i * N + j$. Συμπερασματικά, ένας διδιάστατος πίνακας $M \times N$ μπορεί να θεωρηθεί ως μονοδιάστατος (διάνυσμα) με $M * N$ στοιχεία. Όποτε χρειαζόμαστε το στοιχείο στη θέση (i, j) θα το βρίσκουμε

	0	1	2	3
0	$(0,0)$ 0	$(0,1)$ 3	$(0,2)$ 6	$(0,3)$ 9
1	$(1,0)$ 1	$(1,1)$ 4	$(1,2)$ 7	$(1,3)$ 10
2	$(2,0)$ 2	$(2,1)$ 5	$(2,2)$ 8	$(2,3)$ 11

Σχήμα 5.1: Αρίθμηση θέσεων διδιάστατου πίνακα κατά στήλες

στη θέση $i + M * j$, αν έχουμε επιλέξει αρίθμηση κατά στήλες ή στη θέση $i * N + j$ αν έχουμε επιλέξει αρίθμηση κατά γραμμές.

Παρατηρήστε ότι αν έχουμε αρίθμηση κατά στήλες ενός πίνακα $M \times N$ και γνωρίζουμε τον αριθμό της θέσης, k, μπορούμε να υπολογίσουμε τη γραμμή και τη στήλη της θέσης: είναι αντίστοιχα το υπόλοιπο και το πηλίκο της διαίρεσης του k με το πλήθος των γραμμών. Ανάλογα ισχύουν για την αρίθμηση κατά γραμμές.

Η παραπάνω ανάλυση μας χρειάζεται καθώς δεν υπάρχει η δυνατότητα στη C++ να ορίσουμε με άμεσο και απλό τρόπο, ένα πολυδιάστατο πίνακα με διαστάσεις που είναι άγνωστες κατά τη μεταγλώττιση ή πρόκειται να αλλάξουν κατά τη διάρκεια εκτέλεσης του προγράμματος. Μπορούμε όμως να τον ορίσουμε ως διάνυσμα, ώστε να χρησιμοποιήσουμε το `std::vector<>`, με την εξής αντιστοίχιση:

- Ένας διδιάστατος πίνακας $[a_{ij}]$ με διαστάσεις D1×D2, αντιστοιχεί σε διάνυσμα με πλήθος στοιχείων `D1*D2`. Το στοιχείο a_{ij} αντιστοιχεί στο `a[i+D1*j]` (αποθήκευση κατά στήλες) ή στο `a[i*D2+j]` (αποθήκευση κατά γραμμές).

- Ένας τριδιάστατος πίνακας $[a_{ijk}]$ με διαστάσεις D1×D2×D3, αντιστοιχεί σε διάνυσμα με πλήθος στοιχείων `D1*D2*D3`. Το στοιχείο a_{ijk} αντιστοιχεί

  - στο `a[i+D1*(j+D2*k)]`, αν αποθηκεύουμε πρώτα κατά την πρώτη διάσταση και μετά κατά τη δεύτερη,
  - στο `a[i+D1*(j*D3+k)]`, αν αποθηκεύουμε πρώτα κατά την πρώτη διάσταση και μετά κατά την τρίτη,
  - στο `a[(i*D2+j)*D3+k]`, αν αποθηκεύουμε πρώτα κατά την τρίτη διάσταση και μετά κατά τη δεύτερη,
  - στο `a[(i+j*D1)*D3+k]`, αν αποθηκεύουμε πρώτα κατά την τρίτη διάσταση και μετά κατά την πρώτη.

- Αντίστοιχα ισχύουν για περισσότερες διαστάσεις.

Έχοντας υπόψη τα παραπάνω, μπορούμε να χρησιμοποιήσουμε τους containers `std::array<>` και `std::vector<>` που περιγράψαμε στο §5.2.1 για να υλοποιήσουμε ένα πίνακα με τη μορφή διανύσματος. Έτσι, ο πραγματικός διδιάστατος πίνακας b με 8 γραμμές και 6 στήλες δηλώνεται με την εντολή

```
std::array<double,8*6> b;
```

Το στοιχείο του στην 5η γραμμή και 2η στήλη είναι το `b[4+8*1]`, αν στο πρόγραμμά μας αποφασίσαμε να αποθηκεύουμε τα στοιχεία του b κατά στήλες.

Είμαστε ελεύθεροι να αποφασίσουμε τη σειρά αποθήκευσης των στοιχείων ενός διδιάστατου πίνακα (κατά γραμμές ή κατά στήλες) αρκεί να είμαστε συνεπείς σε όλο τον κώδικα. Η αποθήκευση κατά στήλες καθιστά τους διδιάστατους πίνακες της C++ συμβατούς με τους διδιάστατους πίνακες της Fortran και, συνεπώς, κατάλληλους για χρήση στις εκτεταμένες συλλογές μαθηματικών ρουτινών που έχουν γραφεί στη γλώσσα αυτή. Στο Παράρτημα Γ΄.2 παρουσιάζεται το πώς μπορούμε να χρησιμοποιήσουμε στον κώδικά μας συναρτήσεις γραμμένες σε Fortran.

Όπως θα δούμε στο Κεφάλαιο 14, η C++ παρέχει το μηχανισμό για να απλοποιείται σημαντικά η χρήση των πολυδιάστατων πινάκων. Μπορούμε να δημιουργήσουμε δικό μας container που να αποτελεί «κέλυφος» για αυτούς με απλό και φυσικό τρόπο χρήσης.

5.4 Παρατηρήσεις

5.4.1 Σταθερός πίνακας

Ένα διάνυσμα ή πίνακας μπορεί, όπως και κάθε άλλη ποσότητα, να οριστεί ως σταθερός περιλαμβάνοντας στον ορισμό του τη λέξη **`const`** ή **`constexpr`** (§2.7) με ταυτόχρονη εκχώρηση αρχικής (και μόνιμης) τιμής:

```
std::array<int,6> constexpr powers_of_two{1,2,4,8,16,32};
```

5.4.2 Πλήθος στοιχείων

Σε οποιοδήποτε σημείο του κώδικα χρειαστούμε το πλήθος των στοιχείων ενός `std::array<>` ή ενός `std::vector<>` μπορούμε να το βρούμε με τη χρήση της συνάρτησης–μέλους `size()`. Αν a είναι τέτοιος container, το `a.size()` επιστρέφει το πλήθος των στοιχείων του, *όσο είναι κατά τη στιγμή της κλήσης* της `size()` (καθώς στο `std::vector<>` το πλήθος μπορεί να μεταβληθεί).

Είναι επιτρεπτή η δήλωση διανύσματος με μηδενικό πλήθος στοιχείων. Τότε όμως δεν έχει νόημα η απόπειρα προσπέλασης κάποιου στοιχείου του, και βέβαια η `size()` επιστρέφει 0.

5.4.3 Διάτρεξη διανυσμάτων και πινάκων

Για να διατρέξουμε όλα τα στοιχεία ενός διανύσματος ή πίνακα χρειαζόμαστε εντολές επανάληψης, τόσες όσες οι διαστάσεις του. Π.χ., η εκχώρηση τιμών στα στοιχεία ενός διανύσματος από το πληκτρολόγιο μπορεί να γίνει ως εξής

```
std::array<double,10> a;
for (int i{0}; i < a.size(); ++i) {
  std::cin >> a[i];
}
```

Εναλλακτικά, η χρήση του range for (§4.3) απλοποιεί την εντολή επανάληψης:

```
for (auto & x : a) {
    std::cin >> x;
}
```

Προσέξτε στην παραπάνω εντολή τη χρήση της αναφοράς ώστε οι εκχωρήσεις τιμής στο x να κατευθύνονται στα στοιχεία του a.

Για να διατρέξουμε ένα πίνακα δύο διαστάσεων χρειαζόμαστε δύο εντολές επανάληψης, με μεταβλητές ελέγχου που διατρέχουν η μία τις «γραμμές» και η άλλη τις «στήλες» του. Οι εντολές επανάληψης θα είναι *η μία μέσα στην άλλη*. Καλό είναι όταν διατρέχουμε ένα ενσωματωμένο διδιάστατο πίνακα να μεταβάλλεται πιο γρήγορα ο τελευταίος δείκτης καθώς τα στοιχεία αποθηκεύονται *κατά γραμμές* (row-major order)[3].

Σύμφωνα με τα παραπάνω, η δήλωση του ακέραιου πίνακα a με διάσταση 5×8 και η ανάγνωση τιμών σε αυτόν από το πληκτρολόγιο, γίνεται ως εξής

```
std::array<std::array<int,8>,5> a;
for (int i{0}; i < 5; ++i) {
  for (int j{0}; j < 8; ++j) {
    std::cin >> a[i][j];
  }
}
```

Η συγκεκριμένη επιλογή για τη σειρά των επαναλήψεων (αποκτά το i την πρώτη του τιμή και διατρέχουμε όλα τα j, μετά αλλάζει τιμή το i και ξαναδιατρέχουμε τα j, κοκ.) σημαίνει ότι όταν θα πληκτρολογούμε τις τιμές κατά τη διάρκεια εκτέλεσης του συγκεκριμένου κώδικα, πρέπει να δίνουμε τα στοιχεία του πίνακα κατά γραμμές.

Ένα πίνακα που στο πρόβλημά μας είναι διδιάστατος αλλά επιλέξαμε στον κώδικά μας να τον ορίσουμε ως μονοδιάστατο, τον διατρέχουμε με παρόμοιο τρόπο· μία εντολή επανάληψης για κάθε πραγματική διάσταση:

```
std::array<int,5*8> a;
for (int i{0}; i < 5; ++i) {
```

[3] Προσέξτε ότι στη Fortran η αποθήκευση γίνεται κατά στήλες (column-major order). Συνεπώς, ένας διδιάστατος πίνακας της C++ αντιμετωπίζεται ως ο ανάστροφός του από ρουτίνες της Fortran.

```
  for (int j{0}; j < 8; ++j) {
    std::cin >> a[i+5*j];
  }
}
```

Η ίδια παρατήρηση ισχύει και εδώ: η συγκεκριμένη επιλογή για τη σειρά των επαναλήψεων επιβάλλει να παραθέτουμε τα στοιχεία κατά γραμμές. Προσέξτε ότι η επιλογή της μορφής `a[i+D1*j]` για την πρόσβαση των στοιχείων σημαίνει ότι αυτά δεν αποθηκεύονται με τη σειρά που τα δίνουμε σε διαδοχικές θέσεις. Αν θέλαμε να ισχύει αυτό (για λόγους ταχύτερης αποθήκευσης) θα έπρεπε να εναλλάξουμε τη σειρά των βρόχων.

Εννοείται ότι μπορούμε να διατρέξουμε ένα, στην ουσία, πολυδιάστατο πίνακα που έχει οριστεί ως διάνυσμα, με μία εντολή επανάληψης:

```
std::array<int,5*8> a;
... // give values to a
for (int k{0}; k < a.size(); ++k) {
    std::cout << a[k] << '\n';
}
```

ή, ισοδύναμα,

```
std::array<int,5*8> a;
... // give values to a
for (auto const & x : a) {
    std::cout << x << '\n';
}
```

Η συγκεκριμένη επανάληψη θα τυπώσει τα στοιχεία του a με τη σειρά που βρίσκονται στη μνήμη του υπολογιστή και η οποία καθορίστηκε κατά την εισαγωγή τους (κατά στήλες ή κατά γραμμές), ανάλογα με το αν χρησιμοποιήσαμε τη μορφή `a[i*D2+j]` ή `a[i+D1*j]` για την προσπέλασή τους.

5.5 Δομή (struct)

Όπως αναφέραμε ήδη, οι σχετιζόμενες ποσότητες του προβλήματός μας, με ίδιο τύπο, είναι προτιμότερο να αναπαρίστανται στον κώδικά μας με διάνυσμα ή πίνακα παρά με ισάριθμες ανεξάρτητες μεταβλητές. Στην οργάνωση του κώδικα, αλλά και στη διαχείριση των μεταβλητών, τα διανύσματα και οι πίνακες παρουσιάζουν σημαντικά πλεονεκτήματα. Παρ' όλα αυτά, δεν μπορούν να αναπαραστήσουν συνολικά σχετιζόμενες ποσότητες που δεν είναι ίδιου τύπου.

Ας δούμε πώς μπορούμε να περιγράψουμε σε κώδικα μια σύνθετη έννοια, ένα χημικό στοιχείο. Όπως ξέρουμε, το στοιχείο προσδιορίζεται από το όνομά του, το χημικό του σύμβολο, τον ατομικό του αριθμό, τη μάζα του, κλπ. Αυτά τα σχετιζόμενα δεδομένα είναι διαφορετικού τύπου και αναπαριστώνται καλύτερα ως ένα σύνολο που συνδυάζει σειρές χαρακτήρων, ακέραιους και πραγματικούς αριθμούς

κλπ. Θυμηθείτε ότι ο κατάλληλος τύπος για την αναπαράσταση σειράς χαρακτήρων στη C++ είναι ο `std::string` (§2.14).

Στη C++ υπάρχει η σύνθετη δομή με όνομα **`struct`**, η οποία είναι κατάλληλη για τη συνολική αναπαράσταση μιας σύνθετης ποσότητας με ανόμοιες συνιστώσες. Η σύνταξή της είναι

```
struct όνομα_δομής {
τύποςΑ μέλοςΑ;
τύποςΒ μέλοςΒ;
...
};
```

και μπορεί να εμφανίζεται είτε στο σώμα μιας συνάρτησης (και να έχει περιορισμένη εμβέλεια) είτε εκτός, κατά προτίμηση σε αρχείο header. Επομένως, ο νέος τύπος εισάγεται με την προκαθορισμένη λέξη **`struct`** ακολουθούμενη από το όνομά του. Το όνομα είναι της επιλογής του προγραμματιστή και συντάσσεται με τους γνωστούς κανόνες ονομάτων. Ακολουθούν εντός αγκίστρων και χωρίς συγκεκριμένη σειρά, δηλώσεις ποσοτήτων είτε θεμελιωδών είτε άλλων σύνθετων τύπων. Οι συνιστώσες ποσότητες αποτελούν τα *μέλη* της δομής και η εμβέλειά τους περιορίζεται στο σώμα της δομής. Παρατηρήστε το ';' που ακολουθεί το καταληκτικό '}'. Η δήλωση δομής (ή κλάσης) είναι ένα από τα λίγα σημεία της C++ που εμφανίζεται ο συνδυασμός '};'[4]. Αν το επιθυμούμε, μπορούμε να έχουμε στους ορισμούς των μελών και αποδόσεις αρχικών τιμών.

Μια ποσότητα του νέου τύπου ορίζεται με τον τρόπο που ισχύει για οποιονδήποτε θεμελιώδη τύπο, ως εξής:

```
όνομα_δομής όνομα_μεταβλητής;
```

Έχοντας υπόψη τα παραπάνω, μπορούμε να ορίσουμε ένα νέο τύπο για την αναπαράσταση ενός χημικού στοιχείου ως εξής:

```
struct ChemicalElement {
   double mass;
   int Z; // atomic number
   std::string name;
   std::string symbol;
};
```

Μια μεταβλητή τύπου ChemicalElement και όνομα, π.χ. hydrogen, ορίζεται με τον κώδικα

```
ChemicalElement hydrogen;
```

Η παραπάνω εντολή δημιουργεί τη μεταβλητή hydrogen με απροσδιόριστες τιμές για όσα μέλη της είναι θεμελιώδους τύπου και τις προκαθορισμένες τιμές για τα

[4]Τον συναντούμε και στις αποδόσεις αρχικών τιμών με λίστα καθώς και στις απαριθμήσεις.

υπόλοιπα. Έτσι τα mass, Z είναι απροσδιόριστα και τα name, symbol έχουν την τιμή "".

Απόδοση αρχικής τιμής σε μεταβλητή τέτοιου τύπου μπορεί να γίνει με λίστα· μέσα σε άγκιστρα παραθέτουμε ποσότητες που αντιστοιχούν στα μέλη της δομής, *με τη σειρά που δηλώθηκαν στον ορισμό της*, π.χ.

```
ChemicalElement hydrogen{1.008, 1, "Hydrogen", "H"};
```

Ισοδύναμα μπορούμε να γράψουμε

```
ChemicalElement hydrogen = {1.008, 1, "Hydrogen", "H"};
```

Αν ήδη έχουμε μία ποσότητα ίδιου τύπου, μπορούμε να την αντιγράψουμε κατά μέλη σε άλλη κατά τη δημιουργία της δεύτερης:

```
ChemicalElement hydrogen{1.008, 1, "Hydrogen", "H"};
ChemicalElement elem{hydrogen};
```

Με την τελευταία εντολή ή τις ισοδύναμές της,

```
ChemicalElement elem = hydrogen;
ChemicalElement elem(hydrogen);
```

γίνεται ταυτόχρονα δήλωση και αρχικοποίηση των μελών της μεταβλητής elem.

Ατομική πρόσβαση στα μέλη μιας δομής γίνεται με τον τελεστή '.'· το όνομα της δομής ακολουθείται από '.' και το όνομα του μέλους:

```
ChemicalElement oxygen;
oxygen.name = "Oxygen";
oxygen.mass = 15.99494;
oxygen.Z = 8;
oxygen.symbol = "O";
std::cout << "The␣mass␣of␣element␣" << oxygen.name
          << "␣is␣" << oxygen.mass  << '\n';
```

Στην περίπτωση που έχουμε δείκτη p σε ποσότητα τύπου struct, η προσπέλαση στο μέλος της με όνομα member γίνεται (λαμβάνοντας υπόψη τις σχετικές προτεραιότητες των '*' και '.', όπως παρουσιάζονται στον Πίνακα 2.3) ως εξής

```
(*p).member
```

Τέτοια έκφραση χρησιμοποιείται συχνά στη C++ και γι' αυτό έχει εισαχθεί ειδικός συμβολισμός, τελείως ισοδύναμος με τον παραπάνω:

```
p->member
```

Η δομή (`struct`) που υπάρχει στη C, αποτέλεσε τη βάση για την ανάπτυξη των κλάσεων στη C++, όπως θα δούμε στο Κεφάλαιο 14. Οι κλάσεις επιτρέπουν επιπλέον τη δήλωση συναρτήσεων ως μέλη σε μια δομή.

5.6 Ασκήσεις

1. Δημιουργήστε ένα διάνυσμα με 100 ακέραια στοιχεία. Στο στοιχείο του διανύσματος στη θέση i ($i = 0, \ldots, 99$) δώστε την τιμή i^2+3i+1. Κατόπιν, υπολογίστε το μέσο όρο των στοιχείων του διανύσματος.

2. Δημιουργήστε διάνυσμα με πλήθος στοιχείων N που θα το προσδιορίζει ο χρήστης. Στο στοιχείο j ($j = 0, \ldots, N-1$) δώστε την τιμή $\sin(\pi j/N)$. Κατόπιν, υπολογίστε τη μέγιστη και την ελάχιστη τιμή σε αυτό το διάνυσμα καθώς και το πλήθος των στοιχείων που είναι κατ' απόλυτη τιμή μεγαλύτερα από 0.4.

3. Γράψτε κώδικα που να δημιουργεί δύο πραγματικούς πίνακες A, B με διαστάσεις 20×30. Σε κάθε στοιχείο (i, j) ($i = 0, \ldots, 19$, $j = 0, \ldots, 29$) του A δώστε την τιμή $(i + j)/3$ ενώ στο $B(i, j)$ δώστε την τιμή $2i - j/3$.

   (α΄) Υπολογίστε τον ανάστροφο πίνακα B^T του B.

   (β΄) Υπολογίστε το γινόμενο[5] των πινάκων A, B^T.

   (γ΄) Υπολογίστε το άθροισμα των στοιχείων της κύριας διαγωνίου (το ίχνος) του πίνακα $A \cdot B^T$.

4. Δημιουργήστε ένα διάνυσμα a, 100 πραγματικών στοιχείων. Στο στοιχείο j ($j = 0, \ldots, 99$) του διανύσματος δώστε την τιμή $\cos(\pi j/100)$. Κατόπιν, εναλλάξτε τα πρώτα 50 στοιχεία με τα 50 τελευταία, δηλαδή, $a[0] \leftrightarrow a[50]$, $a[1] \leftrightarrow a[51]$, ..., $a[49] \leftrightarrow a[99]$.

5. Γράψτε πρόγραμμα που να υπολογίζει και να αποθηκεύει σε διάνυσμα τα παραγοντικά των αριθμών από το 0 ως το 12. Κατόπιν, να υπολογίζει το e^x από το άθροισμα

$$e^x \approx x^0/0! + x^1/1! + x^2/2! + \cdots + x^{12}/12! \; .$$

   Το x θα το δίνει ο χρήστης. Συγκρίνετε το αποτέλεσμα με αυτό που δίνει η συνάρτηση `std::exp()` του `<cmath>`.

6. Να υπολογίσετε το π από τον τύπο

$$\frac{1}{\pi} = \sum_{n=0}^{\infty} \frac{((2n)!)^3(42n+5)}{(n!)^6 16^{3n+1}} \; ,$$

   κρατώντας τους πέντε πρώτους όρους στο άθροισμα. Το αποτέλεσμα με 15 ψηφία θα πρέπει να πλησιάζει την τιμή 3.1415926535898.

[5] Το γινόμενο των πινάκων $\boldsymbol{A}_{M\times N}$, $\boldsymbol{B}_{N\times P}$ με στοιχεία τα A_{ij}, B_{ij}, είναι ο πίνακας $\boldsymbol{C}_{M\times p}$ με στοιχεία

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj} \; .$$

Υπόδειξη: Πρώτα υπολογίστε και αποθηκεύστε σε διάνυσμα τα παραγοντικά που θα χρειαστείτε.

7. Γράψτε πρόγραμμα που να βρίσκει και να τυπώνει όλους τους πρώτους αριθμούς μέχρι το 1000 εφαρμόζοντας το «κόσκινο του Ερατοσθένη»: διαγράψτε τα *πολλαπλάσια* των αριθμών από το 2 και μετά (όχι τους ίδιους τους αριθμούς). Όποιοι απομείνουν είναι πρώτοι.

8. Ο Μανώλης ο επιστάτης, είναι υπεύθυνος για να ανάβει και να σβήνει τα φώτα σε διάδρομο ενός κτηρίου. Έστω ότι ο διάδρομος έχει n λαμπτήρες στη σειρά. Καθένας έχει ένα χαρακτηριστικό αριθμό: $1, 2, 3, \ldots, n$.

   Κάθε λαμπτήρας έχει το δικό του διακόπτη. Το είδος του διακόπτη είναι τέτοιο ώστε πατώντας τον ανάβει ο λαμπτήρας (αν είναι σβηστός) ή σβήνει (αν είναι αναμμένος). Ο Μανώλης κάνει n διαδρομές πήγαινε–έλα (όσοι οι λαμπτήρες στο διάδρομο). Στη διαδρομή i διασχίζει το διάδρομο και πατάει το διακόπτη κάθε λαμπτήρα που ο χαρακτηριστικός αριθμός του είναι πολλαπλάσιος του i. Στην επιστροφή κάθε διαδρομής δεν πατά κανένα διακόπτη.

   Πόσοι είναι οι αναμμένοι λαμπτήρες μετά τη διαδρομή n, αν υποθέσουμε ότι αρχικά ήταν όλοι αναμμένοι;

9. Δημιουργήστε ένα πίνακα $M \times N$ με $M = 20$, $N = 60$, στον οποίο $K = 400$ στοιχεία θα έχουν την τιμή 1 και τα υπόλοιπα θα είναι 0. Τα στοιχεία με τιμή 1 θα είναι επιλεγμένα με τυχαίο τρόπο (§2.12.1). Τυπώστε στην οθόνη τον πίνακα αυτόν κατά σειρές, βάζοντας `'x'` για τα μη μηδενικά στοιχεία και `'o'` για τα μηδενικά.

10. Έστω ένα πλέγμα 9×9 πάνω στο οποίο κινείται ένα μυρμήγκι. Σε κάθε βήμα του, το μυρμήγκι κινείται τυχαία σε τετράγωνο που γειτονεύει με την τρέχουσα θέση του (δηλαδή πάνω, κάτω, δεξιά ή αριστερά· όχι διαγωνίως). Δεν μπορεί να φύγει από το πλέγμα.

    Σε κάθε τετράγωνο της πρώτης γραμμής του πλέγματος υπάρχει αρχικά ένας σπόρος. Όταν το μυρμήγκι, στην τυχαία του κίνηση, βρεθεί σε τετράγωνο της πρώτης σειράς, «φορτώνεται» τον σπόρο και τον μεταφέρει έως ότου βρεθεί σε τετράγωνο της τελευταίας γραμμής του πλέγματος όπου και αφήνει τον σπόρο. Το μυρμήγκι μπορεί να μεταφέρει μόνο ένα σπόρο κάθε φορά· εάν βρεθεί σε τετράγωνο της πρώτης γραμμής που δεν έχει σπόρο (γιατί τον πήρε σε προηγούμενη επίσκεψη) προφανώς δεν παίρνει τίποτε. Επίσης, αν μεταφέρει σπόρο σε τετράγωνο της τελευταίας γραμμής που έχει ήδη σπόρο (από προηγούμενη επίσκεψη) δεν μπορεί να αφήσει το φορτίο του.

    Η κίνηση του μυρμηγκιού τελειώνει όταν μεταφέρει όλους τους σπόρους στην τελική γραμμή.

    Να γράψετε κώδικα που να προσομοιώνει την παραπάνω διαδικασία από την αρχική ως την τελική κατάσταση. Να τυπώνει το πλήθος των κινήσεων που έγιναν. Δώστε ως αρχική θέση του μυρμηγκιού το κεντρικό τετράγωνο.

Κεφάλαιο 6

Ροές (streams)

6.1 Εισαγωγή

Έχουμε δει μέχρι τώρα τις δύο *ροές χαρακτήρων (streams)* που μπορούμε να χρησιμοποιήσουμε για είσοδο (`std::cin`) και έξοδο (`std::cout`) ποσοτήτων. Επιπλέον, υπάρχουν και δύο άλλα standard streams, τα `std::cerr`, `std::clog`, που είναι συνδεδεμένα με το standard error του προγράμματός μας. Χρησιμοποιούνται για να μεταφέρουν πληροφορία που δεν έχει σχέση με τα κανονικά αποτελέσματα του προγράμματος, όπως π.χ. προειδοποιήσεις προς το χρήστη. Διαφέρουν στο ότι το πρώτο τυπώνει την πληροφορία που του έχει σταλεί αμέσως, οπότε είναι κατάλληλο π.χ. για επείγουσες ειδοποιήσεις ή επισημάνσεις λαθών, ενώ το δεύτερο τυπώνει όποτε συγκεντρωθεί συγκεκριμένο πλήθος χαρακτήρων, οπότε είναι κατάλληλο π.χ. για πληροφορία σχετική με τη γενική εξέλιξη της εκτέλεσης του κώδικα.

Ας αναφέρουμε απλά, χωρίς να επεκταθούμε, ότι για είσοδο/έξοδο χαρακτήρων τύπου `wchar_t` υποστηρίζονται οι αντίστοιχες με τις παραπάνω ροές χαρακτήρων `std::wcin`, `std::wcout`, `std::wcerr`, `std::wclog`.

Όλα τα παραπάνω streams ορίζονται στο header `<iostream>`.

6.2 Ροές αρχείων

Εκτός των προκαθορισμένων streams μπορούμε να ορίσουμε streams συνδεδεμένα με αρχεία. Στο header `<fstream>` και στο χώρο ονομάτων std ορίζονται οι τύποι (κλάσεις) `ifstream` και `ofstream`. Η δήλωση

```
std::ifstream inpstr{"filename"};
```

δημιουργεί ένα stream *μόνο για ανάγνωση* με όνομα inpstr, που συνδέεται με το αρχείο με όνομα *filename*. Προφανώς, το αρχείο πρέπει να προϋπάρχει. Αντίστοιχα, με την εντολή

```
std::ofstream outstr{"filename"};
```

δημιουργείται ένα stream *μόνο για εγγραφή* με όνομα outstr, που συνδέεται με το αρχείο με όνομα *filename*. Αν το αρχείο αυτό δεν υπάρχει, θα δημιουργηθεί.

Η εντολή

```
std::ofstream outstr{"filename", std::ios_base::app};
```

συνδέει στο αρχείο *filename* το stream με όνομα outstr έτσι ώστε να γίνεται εγγραφή *στο τέλος του*. Ακόμα, η εντολή

```
std::ofstream outstr{"filename", std::ios_base::trunc};
```

«ανοίγει» το αρχείο *filename* καταστρέφοντας τα περιεχόμενά του.

Η χρήση τους είναι απλή: οι μεταβλητές inpstr, outstr (τα ονόματα των οποίων είναι, βεβαίως, της επιλογής του προγραμματιστή) υποκαθιστούν τα `std::cin` και `std::cout` που είδαμε μέχρι τώρα. Έτσι, στον κώδικα

```
double a{10.0};
outstr << a;

char c;
inpstr >> c;
```

η εκτύπωση της μεταβλητής a γίνεται στο αρχείο με το οποίο συνδέεται το outstr ενώ η ανάγνωση του χαρακτήρα c γίνεται από το αντίστοιχο αρχείο του inpstr.

Το κλείσιμο των αρχείων γίνεται *αυτόματα* μόλις η ροή του κώδικα φύγει από την εμβέλεια στην οποία ορίστηκαν τα αντικείμενα που συνδέονται με αυτά. Στη σπάνια περίπτωση που χρειάζεται να κλείσει ένα stream με όνομα str μέσα στην εμβέλεια ορισμού του (π.χ. για να συνδεθεί σε άλλο αρχείο), μπορεί να κληθεί η κατάλληλη συνάρτηση–μέλος: `str.close();`. Το stream str συνδέεται ξανά με αρχείο με την εντολή `str.open("filename");`.

6.3 Ροές strings

Σε διάφορες γλώσσες προγραμματισμού υπάρχει η δυνατότητα να χρησιμοποιήσουμε «εσωτερικό» αρχείο που βρίσκεται στη μνήμη του υπολογιστή και όχι σε κάποιο μέσο αποθήκευσης. Στη C++ τέτοια αρχεία υλοποιούνται με ροές (streams) συνδεδεμένες με σειρές χαρακτήρων.

Με τη συμπερίληψη του header `<sstream>` στο πρόγραμμά μας, παρέχονται οι κλάσεις `std::istringstream` και `std::ostringstream`. Η εκτύπωση σε αντικείμενο τύπου `ostringstream` δημιουργεί ένα C++ string με συνένωση των εκτυπούμενων ποσοτήτων. Με αυτό το μηχανισμό μπορούμε να μετατρέψουμε αριθμούς

σε string, ενώνοντάς τους με χαρακτήρες. Ο παρακάτω κώδικας δημιουργεί ένα C++ string στο οποίο αποθηκεύει τη σειρά χαρακτήρων `"filename_3.dat"`:

```
#include <sstream>

int
main()
{
  std::ostringstream os;
  os << "filename_";
  os << 3;
  os << ".dat";
  // os contains the string "filename_3.dat"
}
```

Η εξαγωγή του string γίνεται καλώντας τη συνάρτηση `str()`, μέλος της κλάσης `ostringstream`:

```
std::cout << os.str(); // prints: filename_3.dat
```

Επομένως, για να συνδέσουμε στο πρόγραμμά μας, π.χ. για έξοδο, το αρχείο με όνομα που έχει αποθηκευτεί ως string στο `std::ostringstream` os, δίνουμε την παρακάτω εντολή:

```
std::ofstream outstr{os.str()};
```

Αντικείμενο τύπου `std::istringstream` χρησιμοποιείται για το «διάβασμα» τιμών από το string με το οποίο συνδέεται, όπως ακριβώς θα γινόταν από stream:

```
#include <sstream>

int
main()
{
  std::istringstream is{"5␣6␣7␣a"};
  int i,j,k;
  is >> i;      // i = 5
  is >> j;      // j = 6
  is >> k;      // k = 7
  char ch;
  is >> ch;     // ch = 'a'
}
```

6.4 Είσοδος–έξοδος δεδομένων

Η εκτύπωση των θεμελιωδών τύπων, καθώς και όσων τύπων παρέχονται από τη Standard Library, σε ροή (stream) συνδεδεμένη με το standard output, ή το

standard error ή με αρχείο ή με string, γίνεται με τον τελεστή '<<':

```
std::cout << 'a';
std::cerr << "Wrong␣value␣of␣b\n";
```

Η είσοδος δεδομένων από το `std::cin` ή από αρχείο γίνεται με τον τελεστή >>, και πάλι ανεξάρτητα από τον τύπο των δεδομένων:

```
double a;
int b;
std::cin >> a;
std::cin >> b;
```

Κενοί χαρακτήρες (και αλλαγές γραμμών) στην είσοδο αγνοούνται.

Ένα χαρακτηριστικό των τελεστών '<<', '>>' είναι ότι μπορούμε να συνδυάσουμε είσοδο ή έξοδο πολλών δεδομένων ταυτόχρονα. Έτσι π.χ. ο κώδικας

```
std::cout << "To␣athroisma␣toy␣"
std::cout << a;
std::cout << "␣kai␣toy␣";
std::cout << b;
std::cout << "␣einai:␣"
std::cout << a+b;
std::cout << '\n';
```

μπορεί να γραφτεί ισοδύναμα

```
std::cout << "To␣athroisma␣toy␣" << a << "␣kai␣toy␣" << b
          << "␣einai:␣" << a+b << '\n';
```

ενώ η ανάγνωση δύο τιμών από το πληκτρολόγιο μπορεί να γίνει με την εντολή

```
std::cin >> a >> b;
```

Δείτε επίσης την §2.11.3, αν έχετε την απορία γιατί η εντολή `std::cin >> a, b` δεν εκτελείται όπως θα νόμιζε κανείς.

6.4.1 Είσοδος–έξοδος δεδομένων λογικού τύπου

Η εκτύπωση στην οθόνη ή σε αρχείο μιας ποσότητας τύπου **`bool`** παρουσιάζει την ιδιαιτερότητα να τη μετατρέπει πρώτα στον αντίστοιχο ακέραιο (§2.5.3) και μετά να την τυπώνει. Έτσι η εντολή

```
std::cout << (3==2);
```

τυπώνει 0. Αν επιθυμούμε να τυπώσει τις λέξεις **`true`** ή **`false`**, πρέπει πρώτα να «στείλουμε» στην έξοδο το διαμορφωτή `std::boolalpha` που ορίζεται στο header `<ios>`:

```
std::cout << std::boolalpha << (3==2);
```

Η μετατροπή σε ακέραιο επανέρχεται αφού στείλουμε στην έξοδο το διαμορφωτή `std::noboolalpha`.

Το «διάβασμα» από το πληκτρολόγιο ή από αρχείο, μιας ποσότητας τύπου `bool` γίνεται μόνο με τις ακέραιες τιμές.

6.4.2 Επιτυχία εισόδου–εξόδου δεδομένων

Η επιτυχία εκτύπωσης ή ανάγνωσης από κάποιο stream ελέγχεται από την «τιμή» του stream: αν είναι `false` τότε η εγγραφή ή η ανάγνωση έχει αποτύχει. Έτσι, η ανάγνωση άγνωστου πλήθους ποσοτήτων (π.χ. ακεραίων) από ένα stream inp γίνεται με χρήση της δομής επανάληψης `while` ως εξής:

```
int i;
while (inp >> i) {
  ...
}
```

Στον παραπάνω κώδικα, εκτελείται η εντολή `inp >> i` και κατόπιν ελέγχεται η «τιμή» του inp. Αν η απόδοση τιμής στη μεταβλητή i έγινε κανονικά, η «τιμή» του ισοδυναμεί με `true` και εκτελείται το σώμα εντολών του `while`.

Από τις διάφορες συναρτήσεις–μέλη των streams χρήσιμες είναι

- η `get()`, η οποία επιστρέφει τον επόμενο χαρακτήρα από τη ροή εισόδου ή `EOF` αν φτάσουμε στο τέλος του αρχείου (οπότε η συνάρτηση–μέλος `eof()` γίνεται `true`). Ανάγνωση και απόρριψη μιας γραμμής στη ροή εισόδου is μπορεί επομένως να γίνει με τον κώδικα

  ```
  while (is.get() != '\n') {}
  ```

- οι `seekg()`/`seekp()` (για ροές εισόδου/εξόδου αντίστοιχα), με τις οποίες μετακινούμαστε σε συγκεκριμένο σημείο της ροής. Π.χ. η μετακίνηση στην αρχή της ροής εισόδου is γίνεται με την εντολή `is.seekg(0);`.

6.5 Διαμορφώσεις

Έχουμε ήδη δει στο §6.4.1 δύο διαμορφωτές (manipulators) που παρέχει η C++, τους `std::boolalpha` και `std::noboolalpha`, που καθορίζουν τη μορφή εκτύπωσης ποσοτήτων ενός συγκεκριμένου τύπου, του τύπου `bool`. Στο header `<ios>` υπάρχουν επίσης οι

- `std::noskipws`, `std::skipws`: (δεν) αγνοούνται οι κενοί χαρακτήρες στην ανάγνωση.

  Η αυτόματα προεπιλεγμένη τιμή είναι `std::skipws`.

- `std::noshowpos`, `std::showpos`: (δεν) τυπώνεται το πρόσημο + σε θετικούς αριθμούς.

  Η αυτόματα προεπιλεγμένη τιμή είναι `std::noshowpos`.

- `std::noshowpoint`, `std::showpoint`: (δεν) τυπώνεται η τελεία σε πραγματικό αριθμό που δεν έχει δεκαδικό μέρος. Αν ζητήσουμε να τυπωθεί, συμπληρώνεται με όσα μηδενικά καθορίζει, αυτόματα ή ρητά, το `std::setprecision()` (που θα αναφέρουμε παρακάτω).

  Η αυτόματα προεπιλεγμένη τιμή είναι `std::noshowpoint`.

- `std::scientific`: οι πραγματικοί τυπώνονται με τη μορφή $d.ddddddedd$.

- `std::fixed`: οι πραγματικοί τυπώνονται με τη μορφή $dddd.dd$.

- `std::defaultfloat`: οι πραγματικοί τυπώνονται με τη μορφή που επιλέγει ο μεταγλωττιστής. Είναι η προκαθορισμένη επιλογή μορφής εκτύπωσης.

- `std::left`, `std::right`: προκαλεί αριστερή/δεξιά στοίχιση στην εκτύπωση (μέσα στο διάστημα που θα καθορίσουμε με το `std::setw()` που θα αναφέρουμε παρακάτω).

  Η αυτόματα προεπιλεγμένη τιμή είναι `std::right`.

Στο header `<iomanip>` υπάρχουν επίσης

- ο `std::setprecision()` που ως όρισμα δέχεται ένα ακέραιο αριθμό.

  Αν έχουμε ορίσει `std::fixed` ή `std::scientific` για την επιθυμητή μορφή εκτύπωσης, το όρισμα καθορίζει το πλήθος των *δεκαδικών* ψηφίων που θα εμφανιστούν (μετά την τελεία). Αν ο αριθμός μπορεί να αναπαρασταθεί με λιγότερα ψηφία από τα ζητούμενα, θα συμπληρωθεί με μηδενικά.

  Αν δεν έχει οριστεί συγκεκριμένη μορφή εκτύπωσης, ή έχουμε προσδιορίσει το `std::defaultfloat`, τότε το όρισμα του `std::setprecision()` προσδιορίζει το πλήθος των *σημαντικών* ψηφίων με το οποίο θέλουμε να τυπώνονται οι πραγματικοί αριθμοί. Η προεπιλεγμένη τιμή είναι 6. Αν ο αριθμός μπορεί να αναπαρασταθεί με λιγότερα ψηφία από τα ζητούμενα, *δεν* συμπληρώνεται με μηδενικά στα δεκαδικά ψηφία που απομένουν.

- ο `std::setw()` που ως όρισμα δέχεται το ελάχιστο πλήθος των θέσεων στο οποίο θα τυπωθεί (ή θα διαβαστεί) η επόμενη ποσότητα. Η προεπιλεγμένη τιμή είναι 0.

- ο `std::setfill()` που ως όρισμα δέχεται τον χαρακτήρα με τον οποίο θα γεμίσουν οι κενές θέσεις αν το `std::setw()` όρισε περισσότερες από την ακρίβεια. Ο προεπιλεγμένος χαρακτήρας είναι ο κενός, '␣'.

Όλοι οι manipulators ανήκουν στο χώρο ονομάτων `std`.

Παράδειγμα

```
#include <ios>
#include <iomanip>
#include <iostream>
int
main()
{
  double b{3.25};
  std::cout << b << '\n';
  std::cout << std::showpoint << b << '\n';
  std::cout << std::noshowpoint;            // reset
  double a{256.123456789987};
  std::cout << "default\t" << a << '\n';
  std::cout << "scientific\t" << std::scientific << a << '\n';
  std::cout << "fixed\t" << std::fixed << a << '\n';
  std::cout << "with␣9␣digits\t" << std::setprecision(9)
            << a << '\n';
}
```

Αντί για τους διαμορφωτές `setprecision()`, `setw()` και `setfill()`, μπορούμε να χρησιμοποιήσουμε τις αντίστοιχες *συναρτήσεις-μέλη* κάθε ροής, `precision()`, `width()` και `fill()`:

```
#include <iostream>

int
main()
{
  std::cout.precision(9);
  std::cout << 256.123456789987 << '\n'; // 256.123457
  std::cout << std::fixed;
  std::cout << 256.123456789987 << '\n'; // 256.123456790
}
```

6.6 Ασκήσεις

1. - Να δημιουργήσετε με πρόγραμμα ένα αρχείο με όνομα *trig.dat*. Να τυπώσετε σε αυτό το ημίτονο, το συνημίτονο και την εφαπτομένη των γωνιών από $0°$ έως $359.9°$ ανά $0.1°$. Οι αριθμοί που εκτυπώνονται να έχουν 5 σημαντικά ψηφία και να είναι στοιχισμένοι σε τέσσερις στήλες: γωνία, ημίτονο, συνημίτονο και εφαπτομένη. Στην πρώτη γραμμή του αρχείου να γράψετε τον αριθμό των γραμμών που ακολουθούν.
   - Διαβάστε το αρχείο *trig.dat* με άλλο πρόγραμμα και βρείτε σε ποια από τις γωνίες του αρχείου αντιστοιχεί το μικρότερο συνημίτονο.

2. Γράψτε πρόγραμμα που να διαβάζει μήνα και έτος από τον χρήστη και να τυπώνει στο αρχείο με όνομα *calendar* τις ημέρες του μήνα με τη μορφή (παράδειγμα για Μάρτιο του 2014):

   03/2014

ΔΕΥ	ΤΡΙ	ΤΕΤ	ΠΕΜ	ΠΑΡ	ΣΑΒ	ΚΥΡ
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

   Θα χρειαστεί να βρείτε:

   (α΄) Ποια ημέρα (Δευτέρα, Τρίτη, ...) πέφτει η πρώτη του μηνός.

   (β΄) Πόσες ημέρες έχει ο συγκεκριμένος μήνας.

   Για το πρώτο, θα σας βοηθήσει ο αλγόριθμος του Zeller· δείτε την άσκηση 9 στη σελίδα 68. Για το δεύτερο, δείτε την άσκηση 8 στη σελίδα 68.

3. Να υπολογίσετε τους δεκαδικούς λογαρίθμους των αριθμών από 1.0 έως το 9.9 με βήμα 0.1. Να τυπώσετε τα αποτελέσματα με 3 δεκαδικά ψηφία με τη μορφή διδιάστατου πίνακα, όπως παρακάτω:

	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	0.000	0.041	0.079	0.114	0.146	0.176	0.204	0.230	0.255	0.279
2	0.301	0.322	0.342	0.362	0.380	0.398	0.415	0.431	0.447	0.462
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
9	0.954	0.959	0.964	0.968	0.973	0.978	0.982	0.987	0.991	0.996

   Η πρώτη στήλη έχει το ακέραιο μέρος ενός αριθμού ενώ η πρώτη γραμμή έχει το δεκαδικό μέρος. Το άθροισμα του πρώτου στοιχείου στη γραμμή i και του πρώτου στη στήλη j έχει λογάριθμο που δίνεται στην τομή τους, δηλαδή στο στοιχείο (i, j).

4. Δημιουργήστε τον πίνακα του Pascal. Ο πίνακας αυτός είναι διδιάστατος, $n \times n$, και έχει στοιχεία που ορίζονται από τις σχέσεις

$$\begin{array}{ll} P(i,1) = P(1,j) = 1 & \text{για κάθε } i,\ j \text{ και} \\ P(i,j) = P(i-1,j) + P(i,j-1) & \text{για } i,j > 1. \end{array}$$

   Το κάθε «εσωτερικό» στοιχείο επομένως είναι το άθροισμα των προηγούμενων στη στήλη του και στη γραμμή του.

   - Γράψτε κώδικα που να τυπώνει τον πίνακα του Pascal. Οι αριθμοί να είναι στοιχισμένοι κατά στήλες και κάθε γραμμή του πίνακα να τυπώνεται σε ξεχωριστή γραμμή. Εφαρμόστε τον για $n = 7$.
   - Τροποποιήστε το πρόγραμμα ώστε να εκτυπώνει τον πίνακα στο αρχείο *pascal*.

5. Έστω η ακόλουθη διαδικασία για ένα θετικό ακέραιο αριθμό («αριθμός εισόδου»):

   - αν ο αριθμός είναι άρτιος τον διαιρούμε με το 2.
   - αν ο αριθμός είναι περιττός τον πολλαπλασιάζουμε με το 3 και προσθέτουμε 1.

   Ξεκινώντας από ένα αριθμό n, επαναλαμβάνουμε τη διαδικασία θεωρώντας το αποτέλεσμα κάθε επανάληψης ως αριθμό είσοδου της επόμενης. Σύμφωνα με την υπόθεση του Collatz, η επανάληψη αυτής της διαδικασίας θα δώσει, μετά από πεπερασμένο αριθμό βημάτων, ως αποτέλεσμα το 1. Ο αριθμός βημάτων που θα χρειαστεί για αυτό εξαρτάται από τον αρχικό μας αριθμό n.

   Να γράψετε πρόγραμμα που

   - να τυπώνει στο αρχείο *collatz.dat* κάθε θετικό ακέραιο εισόδου από το 2 μέχρι το 100000 (πρώτη στήλη) μαζί με το αντίστοιχο πλήθος των βημάτων μέχρι να βγει αποτέλεσμα το 1 (δεύτερη στήλη).
   - Να τυπώνει στην οθόνη τον αριθμό εισόδου που είχε το μεγαλύτερο αριθμό βημάτων, μαζί με τον αριθμό βημάτων.

6. (α′) Δημιουργήστε με πρόγραμμα το αρχείο *random.txt* με 10000 τυχαίους ακεραίους στο διάστημα $[-20, 20]$.

   (β′) Γράψτε πρόγραμμα που να διαβάζει το αρχείο *random.txt* και να τυπώνει στην οθόνη πόσους θετικούς, αρνητικούς και ίσους με το 0 αριθμούς περιέχει.

7. Σύμφωνα με την υπόθεση Lemoine, κάθε περιττός θετικός ακέραιος αριθμός μεγαλύτερος του 5, μπορεί να γραφεί ως άθροισμα ενός πρώτου αριθμού και του διπλάσιου ενός άλλου πρώτου αριθμού (χωρίς να είναι απαραίτητα

διαφορετικοί οι δύο πρώτοι αριθμοί)[1]. Να ελέγξετε αυτή την υπόθεση για κάθε περιττό αριθμό m από το 7 μέχρι το 999999: βρείτε τους πρώτους αριθμούς p, q που ικανοποιούν τη σχέση $m = p + 2q$. Γράψτε τους αριθμούς m, p, q, με ένα κενό ανάμεσά τους, στο αρχείο *lemoine.dat*, ώστε να έχετε την κάθε τριάδα σε ξεχωριστή γραμμή.

8. Μια διαμόρφωση ενός αρχείου που μπορεί να χρησιμοποιηθεί για την αποθήκευση ασπρόμαυρης εικόνας είναι η ακόλουθη:

   - Η πρώτη γραμμή του αρχείου πρέπει να γράφει: P1.
   - Η δεύτερη να γράφει τις διαστάσεις της εικόνας: πλάτος ύψος (δηλαδή τους δύο αριθμούς με κενό μεταξύ τους).
   - Να ακολουθούν οι αριθμοί 0 ή 1· αυτοί αντιπροσωπεύουν τα pixels της εικόνας *κατά γραμμές*: κάθε λευκό pixel αντιστοιχεί στο 0 και κάθε μαύρο στο 1. Οι αριθμοί μπορούν να διαχωρίζονται από κενά αλλά δεν είναι απαραίτητο. Σε κάθε σειρά του αρχείου μπορούμε να έχουμε έως 70 χαρακτήρες.

   Η διαμόρφωση αυτή αποτελεί ένα αρχείο τύπου plain pbm (portable bitmap) που μπορούμε να το δούμε με προγράμματα απεικόνισης.

   Δημιουργήστε ένα πίνακα 512×512 στον οποίο τα στοιχεία που βρίσκονται μία θέση κάτω και μία θέση πάνω από τις δύο διαγωνίους, κύρια και δευτερεύουσα (δηλαδή, σε τέσσερις συγκεκριμένες γραμμές), θα έχουν την τιμή 1 και τα υπόλοιπα θα είναι 0. Έτσι, έχουμε αποθηκεύσει μια διδιάστατη εικόνα στον πίνακα. Τυπώστε την σε αρχείο με κατάληξη .ppbm και διαμόρφωση plain pbm.

9. Μια διαμόρφωση ενός αρχείου κειμένου (ASCII) που μπορεί να χρησιμοποιηθεί για την αποθήκευση εικόνας με αποχρώσεις του γκρι είναι η ακόλουθη:

   - η πρώτη γραμμή του αρχείου πρέπει να γράφει: P2.
   - η δεύτερη πρέπει να γράφει τις διαστάσεις: πλάτος ύψος (δηλαδή τους δύο αριθμούς με κενό μεταξύ τους).
   - η τρίτη πρέπει να έχει ένα θετικό ακέραιο αριθμό K που αντιπροσωπεύει τη μέγιστη τιμή του γκρι. Πρέπει να είναι μικρότερη από 256. Τυπική τιμή για αυτή είναι το 255.
   - ακολουθούν τα pixels της εικόνας *κατά γραμμές*. Κάθε pixel αντιπροσωπεύεται από ένα ακέραιο αριθμό από το 0 έως και το K. Μεταξύ των τιμών πρέπει να υπάρχει ένα τουλάχιστον κενό ή αλλαγή γραμμής. Οι γραμμές του αρχείου πρέπει να έχουν έως 70 χαρακτήρες. Διευκολύνει επομένως αν κάθε pixel είναι γραμμένο σε ξεχωριστή γραμμή.

[1] το 1 δεν θεωρείται πρώτος αριθμός.

Η διαμόρφωση αυτή αποτελεί ένα αρχείο τύπου plain pgm (portable graymap) που μπορούμε να το δούμε με προγράμματα απεικόνισης.

Γράψτε ένα πρόγραμμα που θα διαβάζει το αρχείο *input.ppgm* και θα δημιουργεί μία νέα εικόνα στο *output.ppgm* ως εξής: Το pixel (i, j) στη νέα εικόνα θα είναι ο μέσος όρος του pixel (i, j) της αρχικής και των γειτονικών του (μέχρι γείτονες τάξης p). Επομένως, αν το (i, j) είναι μακριά από τα άκρα, τα pixels που χρησιμοποιούμε στον υπολογισμό είναι αυτά που βρίσκονται στο τετράγωνο με κορυφές τα $(i \pm p, j \pm p)$. Αν το (i, j) είναι στα άκρα, οι γείτονες είναι λιγότεροι.

Το πρόγραμμά σας θα ζητά από το χρήστη τον ακέραιο θετικό αριθμό p. Για να το δοκιμάσετε, χρησιμοποιήστε το αρχείο στη διεύθυνση `http://fla.st/1KxdatB`.

10. Μια διαμόρφωση ενός αρχείου που μπορεί να χρησιμοποιηθεί για την αποθήκευση έγχρωμης εικόνας είναι η ακόλουθη:

    - Η πρώτη γραμμή του αρχείου πρέπει να γράφει: P3.
    - Η δεύτερη πρέπει να γράφει τις διαστάσεις της εικόνας: πλάτος ύψος (δηλαδή τους δύο αριθμούς με κενό μεταξύ τους).
    - Η τρίτη πρέπει να έχει ένα θετικό ακέραιο αριθμό K, μέχρι το 255, που αντιπροσωπεύει τη μέγιστη τιμή του κάθε χρώματος. Τυπική τιμή είναι το 255.
    - Να ακολουθούν τα pixels της εικόνας *κατά γραμμές*. Στο αρχείο θα γράφεται η τιμή των χρωμάτων «κόκκινο» (R), «πράσινο» (G), «μπλε» (B) για το κάθε pixel, με ένα κενό μεταξύ τους. Ένα pixel που έχει χρώμα κόκκινο θα αναπαρίσταται από την τριάδα K 0 0 (αν το K είναι 255 θα γράφουμε 255 0 0). Το pixel με «πράσινο» χρώμα θα αντιστοιχεί στη γραμμή 0 K 0. Το μαύρο χρώμα είναι το 0 0 0 ενώ το λευκό K K K. Το κίτρινο είναι K K 0. Σε κάθε συνιστώσα RGB μπορούμε γενικά να έχουμε οποιαδήποτε τιμή μεταξύ 0 και K ώστε να παράγουμε όλα τα χρώματα.
      Οι γραμμές του αρχείου πρέπει να έχουν έως 70 χαρακτήρες.

    Η διαμόρφωση αυτή αποτελεί ένα αρχείο τύπου plain ppm (portable pixmap) που μπορούμε να το δούμε με προγράμματα απεικόνισης.

    Δημιουργήστε ένα αρχείο με όνομα *france.pppm* με τη σημαία της Γαλλίας[2], σε 512×768 pixels, χρησιμοποιώντας την παραπάνω διαμόρφωση.

11. Γράψτε ένα πρόγραμμα που να υλοποιεί το Game of Life[3] του Dr. J. Conway. Αυτό προσομοιώνει την εξέλιξη ζωντανών οργανισμών βασιζόμενο σε συγκεκριμένους κανόνες.

[2] τρεις κατακόρυφες λωρίδες ίσου πλάτους: μπλε, λευκή, κόκκινη.
[3] `http://www.math.com/students/wonders/life/life.html`

Σε ένα πλέγμα $M \times N$, κάθε τετράγωνο έχει οκτώ πρώτους γείτονες (λιγότερους αν βρίσκεται στα άκρα). Τοποθετούμε σε τυχαίες θέσεις K οργανισμούς. Σε κάθε βήμα της εξέλιξης (νέα γενιά):

(α΄) Ένα κενό τετράγωνο με ακριβώς τρεις «ζωντανούς» γείτονες γίνεται «ζωντανό» (γέννηση).

(β΄) Ένα «ζωντανό» τετράγωνο με δύο ή τρεις «ζωντανούς» γείτονες παραμένει ζωντανό (επιβίωση).

(γ΄) Σε κάθε άλλη περίπτωση ένα τετράγωνο γίνεται ή παραμένει κενό δηλαδή «πεθαίνει» ή παραμένει «νεκρό» (από υπερπληθυσμό ή μοναξιά!).

Η «αποθήκευση» της επόμενης γενιάς γίνεται αφού ολοκληρωθεί ο υπολογισμός της για όλα τα τετράγωνα.

Να τυπώνετε την κάθε γενιά σε αρχεία τύπου plain pbm (δείτε την περιγραφή της διαμόρφωσης στην άσκηση 8), ώστε να μπορείτε να τις δείτε όλες μαζί διαδοχικά[4].

Σχηματίστε τετραγωνικό πλέγμα 512×512 και υπολογίστε 1000 γενιές. Δοκιμάστε να τοποθετήσετε αρχικά τους οργανισμούς όχι σε τυχαίες θέσεις αλλά σε μία θέση κάτω και μία θέση πάνω από τις δύο διαγωνίους, κύρια και δευτερεύουσα (δηλαδή, σε τέσσερις συγκεκριμένες γραμμές).

12. Ένα μυρμήγκι (Langton's ant[5]) βρίσκεται σε ορθογώνιο πλέγμα από 128×128 τετράγωνα. Τα τετράγωνα μπορούν να είναι είτε άσπρα είτε μαύρα. Αρχικά είναι όλα άσπρα. Το μυρμήγκι έχει αρχική θέση το κέντρο του πλέγματος (το σημείο $(64, 64)$), κατεύθυνση προς τα επάνω και κινείται σε κάθε βήμα του σύμφωνα με τους ακόλουθους κανόνες:

- Αν βρίσκεται σε μαύρο τετράγωνο, αλλάζει το χρώμα του τετραγώνου σε άσπρο, στρέφει κατά $90°$ αριστερόστροφα και προχωρά κατά ένα τετράγωνο.
- Αν βρίσκεται σε άσπρο τετράγωνο, αλλάζει το χρώμα του τετραγώνου σε μαύρο, στρέφει κατά $90°$ δεξιόστροφα και προχωρά κατά ένα τετράγωνο.

Γράψτε πρόγραμμα που να προσομοιώνει την κίνηση του μυρμηγκιού για 12000 βήματα. Κάθε 100 βήματα να αποθηκεύετε την εικόνα του πλέγματος σε αρχείο με τη διαμόρφωση plain pbm (δείτε την περιγραφή της διαμόρφωσης στην άσκηση 8). Δείτε όλες τις εικόνες· τι παρατηρείτε;

[4] Σε συστήματα UNIX, με εγκατεστημένο το πρόγραμμα imagemagick, η εντολή είναι `animate *.ppbm`

[5] http://mathworld.wolfram.com/LangtonsAnt.html

Κεφάλαιο 7

Συναρτήσεις

7.1 Εισαγωγή

Στα προηγούμενα κεφάλαια έχουν παρουσιαστεί κάποιες από τις βασικές εντολές και έννοιες της C++, αρκετές ώστε να μπορούμε να γράψουμε σχετικά πολύπλοκους κώδικες. Η συγκέντρωση όμως, όλου του κώδικα σε μία συνάρτηση, τη `main()`, καθιστά δύσκολη την κατανόησή του και, κυρίως, τη διόρθωση λαθών. Σχεδόν πάντα ο κώδικας αποτελείται από τμήματα που είναι σε μεγάλο βαθμό ανεξάρτητα μεταξύ τους. Αυτά μπορούν να απομονωθούν σε αυτόνομες *συναρτήσεις*, να αποτελούν, δηλαδή, ομάδες εντολών με συγκεκριμένο όνομα, οι οποίες θα καλούνται όπου και όσες φορές χρειάζεται από τη `main()` ή άλλες συναρτήσεις, χρησιμοποιώντας μόνο αυτό το όνομα. Αυτές οι ομάδες εντολών θα παραμετροποιούνται συνήθως από μία ή περισσότερες ποσότητες, τα *ορίσματα* της συνάρτησης.

Η οργάνωση του προγράμματός μας σε συναρτήσεις είναι ένα πρώτο βήμα στην απλοποίηση του κώδικα και μας επιτρέπει να επικεντρωνόμαστε σε συγκεκριμένες, κατά το δυνατόν απλές, εργασίες κατά την ανάπτυξη ή διόρθωση του προγράμματος. Έτσι π.χ., ένας αλγόριθμος μπορεί να υλοποιηθεί, να διορθωθεί και να βελτιστοποιηθεί αυτόνομα, ανεξάρτητα από τον υπόλοιπο κώδικα και, επομένως, να μπορεί να χρησιμοποιείται από εμάς ή άλλους σε διαφορετικά προγράμματα. Από τη στιγμή που θα υπάρξει απομόνωση του κώδικα σε αυτόνομη, ελεγμένη συνάρτηση, η χρήση του απλοποιείται σημαντικά καθώς μας απασχολεί μόνο το πώς τον καλούμε και τι ορίσματα πρέπει να «περάσουμε» στη συνάρτηση και όχι το ποιους ακριβώς υπολογισμούς εκτελεί.

Η οργάνωση του κώδικα σε δεδομένα και σε διαδικασίες που επιδρούν σε αυτά περιγράφεται ως *δομημένος (structured)* ή *διαδικαστικός (procedural)* προγραμματισμός και αποτελεί ένα από τα μοντέλα προγραμματισμού που υποστηρίζει η C++.

7.1.1 Η έννοια της συνάρτησης

Ας προσπαθήσουμε να κατανοήσουμε την έννοια της συνάρτησης στον προγραμματισμό με βάση τη γνωστή έννοια της μαθηματικής συνάρτησης. Στα μαθηματικά μπορούμε να ορίσουμε ότι

$$f(x) = x^2 + 5x - 2 \, .$$

Αυτό σημαίνει ότι οι συγκεκριμένες πράξεις, $x^2 + 5x - 2$, έχουν αποκτήσει ένα *όνομα*, f, μέσω του οποίου θα τις χρησιμοποιούμε όποτε χρειαζόμαστε το αποτέλεσμά τους. Παρατηρούμε ότι εξαρτώνται από ένα *σύμβολο*, το x, και επομένως, δεν μπορούν να εκτελεστούν και να μας δώσουν αποτέλεσμα. Ο συμβολισμός $f(x)$ υποδηλώνει ότι η συνάρτηση f έχει ως παράμετρο, ως *όρισμα* όπως λέμε, την ποσότητα x. Για τις συναρτήσεις πραγματικής μεταβλητής το x συμβολίζει έναν πραγματικό αριθμό. Όταν επιθυμούμε να εκτελέσουμε τις πράξεις x^2+5x-2 για κάποια τιμή του x μπορούμε να χρησιμοποιήσουμε (να *καλέσουμε*) τη συνάρτηση f με ταυτόχρονο προσδιορισμό της τιμής του ορίσματος, του συμβόλου x. Γι' αυτό γράφουμε, π.χ., $y = f(2.5)$ αντί για το ισοδύναμο αλλά πιο εκτεταμένο $y = 2.5^2 + 5 \times 2.5 - 2$. Οι πράξεις μπορούν να εκτελεστούν αφού δώσουμε τιμή στο σύμβολο x, θέσουμε, δηλαδή, τη δεδομένη τιμή όπου εμφανίζεται το x. Το αποτέλεσμά τους για τη συγκεκριμένη τιμή εκχωρείται (*επιστρέφεται*) στο όνομα της συνάρτησης και μπορούμε να το κρατήσουμε σε κάποια κατάλληλη ποσότητα. Μια μαθηματική συνάρτηση μπορεί να έχει περισσότερα από ένα ορίσματα (παραμέτρους). Αφού τα προσδιορίσουμε όλα, μπορούν να εκτελεστούν οι πράξεις τις οποίες αντιπροσωπεύει. Προφανώς, μια συνάρτηση μπορεί να κληθεί όσες φορές επιθυμούμε.

Στον προγραμματισμό, κατά πλήρη αντιστοιχία: ένα τμήμα κώδικα μπορούμε να το ξεχωρίσουμε από το κύριο σώμα των εντολών του προγράμματός μας και να του δώσουμε ένα όνομα. Αυτό το τμήμα κώδικα μπορεί να εξαρτάται από καμία, μία ή περισσότερες ποσότητες. Στον ορισμό της συνάρτησης, τα ορίσματα δεν είναι τίποτε άλλο παρά σύμβολα που αντιστοιχούν σε ποσότητες συγκεκριμένων τύπων. Όταν καλέσουμε τη συνάρτηση με το όνομά της, πρέπει να προσδιορίσουμε ταυτόχρονα και τα ορίσματά της, δίνοντας τιμές των αντίστοιχων τύπων σε καθένα από αυτά. Τότε μόνο μπορούν να εκτελεστούν οι εντολές που αυτή αντιπροσωπεύει. Όταν ολοκληρωθεί η εκτέλεση της συνάρτησης μπορεί να επιστρέφεται τιμή μέσω του ονόματός της. Η επιστρεφόμενη τιμή μπορεί να χρησιμοποιηθεί· έτσι μπορούμε να την αποθηκεύσουμε σε κατάλληλη μεταβλητή, ή να την τυπώσουμε σε αρχείο, ή να την συμπεριλάβουμε σε σύνθετη έκφραση. Στην περίπτωση που θέλουμε να λάβουμε περισσότερα του ενός αποτελέσματα, μπορούμε να υποκαταστήσουμε το τμήμα του κώδικα με συνάρτηση στην οποία θα χρησιμοποιήσουμε ως τύπο επιστρεφόμενης ποσότητας μια κατάλληλη δομή (§5.5) ή κλάση (§14). Εναλλακτικά, μπορούμε να το υποκαταστήσουμε με συνάρτηση η οποία έχει επιπλέον ορίσματα που αποκτούν τιμή μετά την ολοκλήρωση της εκτέλεσής της.

7.2 Ορισμός

Ένα τμήμα κώδικα που είναι σε μεγάλο βαθμό ανεξάρτητο από το υπόλοιπο πρόγραμμα μπορεί να αποτελέσει μια συνάρτηση. Το τμήμα αυτό περιλαμβάνει δηλώσεις ποσοτήτων και εκτελέσιμες εντολές και μπορεί να παραμετροποιείται από κάποιες σταθερές ή μεταβλητές ποσότητες—τα ορίσματα της συνάρτησης— ή, όπως θα δούμε στο §7.11, από τύπους ποσοτήτων. Μια συνάρτηση μπορεί να μην επιστρέφει τίποτε ή να επιστρέφει μία απλή ή σύνθετη ποσότητα. Συναρτήσεις που πρέπει να επιστρέφουν περισσότερες από μία ανεξάρτητες τιμές, περιλαμβάνουν στη λίστα ορισμάτων μεταβλητές κατάλληλου τύπου για να τις εξαγάγουν.

Ο ορισμός μιας συνάρτησης έχει μία από τις ακόλουθες γενικές μορφές:

```
τύπος_επιστρεφόμενης_ποσότητας
όνομα(τύπος_ορίσματος_Α όρισμαΑ, τύπος_ορίσματος_Β όρισμαΒ,...)
{
// κώδικας
}
```

ή

```
auto
όνομα(τύπος_ορίσματος_Α όρισμαΑ, τύπος_ορίσματος_Β όρισμαΒ,...)
-> τύπος_επιστρεφόμενης_ποσότητας
{
// κώδικας
}
```

ή, ακόμα,

```
auto
όνομα(τύπος_ορίσματος_Α όρισμαΑ, τύπος_ορίσματος_Β όρισμαΒ,...)
{
// κώδικας
}
```

ή και

```
decltype(auto)
όνομα(τύπος_ορίσματος_Α όρισμαΑ, τύπος_ορίσματος_Β όρισμαΒ,...)
{
// κώδικας
}
```

Ο «τύπος_επιστρεφόμενης_ποσότητας», αν προσδιορίζεται ρητά, μπορεί να είναι `void`· υποδηλώνεται έτσι ότι δεν επιστρέφεται τιμή. Εναλλακτικά, μπορεί να είναι οποιοσδήποτε απλός ή σύνθετος τύπος εκτός από ενσωματωμένο διάνυσμα

και συνάρτηση (επιτρέπεται, όμως, να είναι δείκτης σε ενσωματωμένο διάνυσμα ή συνάρτηση).

Η πρώτη μορφή ορισμού είναι η παλαιότερη στη C++ και πιο συνηθισμένη. Στην τρίτη και τέταρτη μορφή ορισμού ο compiler πρέπει να εξαγάγει (με κανόνες που διαφέρουν) τον τύπο από τον τύπο της ποσότητας που εμφανίζεται σε εντολή `return` στο σώμα της συνάρτησης. Φυσικά, αν υπάρχουν πολλές εντολές `return`, θα πρέπει όλες να παραθέτουν τιμές με κοινό τύπο. Επιπλέον, επιτρέπεται να γίνεται αναδρομική κλήση της συνάρτησης (§7.4.1) και επιστροφή του αποτελέσματός της αρκεί μέσα στο σώμα της συνάρτησης να υπάρχει άλλη εντολή `return` από την οποία μπορεί να εξαχθεί ο τύπος της επιστρεφόμενης ποσότητας.

Η λίστα ορισμάτων μπορεί να είναι κενή ή, ισοδύναμα, να περιέχει τη λέξη `void`. Τα ορίσματα, αν υπάρχουν, δεν μπορούν να επανοριστούν στο σώμα της συνάρτησης και η εμβέλειά τους εκτείνεται ως το καταληκτικό '}' του σώματος. Οι δηλώσεις στη λίστα ορισμάτων γίνονται όπως οι γνωστές δηλώσεις ποσοτήτων· ειδικά για την περίπτωση που θέλουμε να έχουμε ως όρισμα μιας συνάρτησης ένα ενσωματωμένο διάνυσμα (§5.2.2), χρησιμοποιούμε την ακόλουθη μορφή:

τύπος_στοιχείων όνομα_διανύσματος[ ]

Προσέξτε ότι μεταξύ των αγκυλών έχουμε κενό. Στην πραγματικότητα, σε μια τέτοια δήλωση ορίσματος, «περνά» ως όρισμα ένας δείκτης στο αρχικό στοιχείο του διανύσματος. Με άλλα λόγια, οι δηλώσεις ορίσματος `int a[]` και `int *a` είναι ισοδύναμες. Αυτό έχει ως συνέπεια να μην «περνά» ταυτόχρονα και η διάσταση του ενσωματωμένου διανύσματος οπότε, αν χρειάζεται, πρέπει να δοθεί με ξεχωριστό όρισμα. Οι containers της Standard Library που θα δούμε στο Κεφάλαιο 11 δεν έχουν τέτοιο πρόβλημα.

Αν τυχόν υπάρχει κάποιο όρισμα της συνάρτησης που δεν χρησιμοποιείται στο σώμα της—για διάφορους λόγους μπορεί να συμβεί—μπορούμε να παραλείψουμε το όνομά του (αλλά όχι τον τύπο του) στη λίστα ορισμάτων.

Ο ορισμός μιας συνάρτησης δεν μπορεί να γίνει στο σώμα άλλης συνάρτησης· πρέπει να γραφεί έξω από οποιαδήποτε συνάρτηση.

7.2.1 Επιστροφή

Η επιστροφή τιμής από τη συνάρτηση γίνεται με την εντολή

`return` τιμή;

που μπορεί να εμφανίζεται μία ή περισσότερες φορές στο σώμα της συνάρτησης. Εξαίρεση αποτελεί η `main()` στην οποία το `return` δεν είναι αναγκαίο: αν παραλείπεται, θεωρείται ότι δόθηκε ως τελευταία εκτελέσιμη γραμμή η εντολή `return 0;`. Η «τιμή» που προσδιορίζεται στο `return` μπορεί να είναι μια μεταβλητή ή σταθερή ποσότητα ή έκφραση (που μπορεί να περιέχει και κλήση συνάρτησης). Η τελική τιμή που θα προκύψει πρέπει να έχει τον τύπο της ποσότητας που επιστρέφει η συνάρτηση ή να μπορεί να μετατραπεί σε αυτόν.

Μια συνάρτηση που δεν επιστρέφει τιμή (δηλαδή «επιστρέφει» `void`), μπορεί, χωρίς να είναι απαραίτητο, να περιλαμβάνει εντολές `return;` (χωρίς τιμή). Επίσης, μια τέτοια συνάρτηση μπορεί να «επιστρέφει» την «τιμή» μιας συνάρτησης που «επιστρέφει» `void`. Οι ακόλουθες μορφές του `return` είναι, επομένως, αποδεκτές

```
void
f(int a)
{
  // ...
   return;
}

void
g(int b)
{
   return f(b);
}
```

Όταν η ροή του προγράμματος συναντήσει μέσα σε συνάρτηση την εντολή `return`, επιστρέφει στο σημείο που έγινε η κλήση.

Καλό είναι να υπάρχει μόνο *ένα* σημείο εξόδου από τη συνάρτηση. Μπορούμε να χρησιμοποιήσουμε μια μεταβλητή κατάλληλου τύπου για να αποθηκεύσουμε το αποτέλεσμα της συνάρτησης σε οποιοδήποτε σημείο παραχθεί αυτό· κατόπιν, μπορούμε να την «επιστρέψουμε» από *ένα* σημείο, στο τέλος του σώματος της συνάρτησης.

Αν επιθυμούμε, μπορούμε να χρησιμοποιήσουμε ως επιστρεφόμενες τιμές μιας συνάρτησης τα `EXIT_SUCCESS` και `EXIT_FAILURE` που ορίζονται στο `<cstdlib>`, για να υποδηλώσουμε ότι η εκτέλεση ήταν επιτυχής ή όχι, αντίστοιχα.

7.3 Δήλωση

Μια συνάρτηση για να κληθεί πρέπει προηγουμένως να έχει *δηλωθεί*, αλλά όχι απαραίτητα να έχει *οριστεί*. Ο compiler πρέπει να γνωρίζει το όνομά της, τα ορίσματα (τύπο και πλήθος τους) και τον τύπο επιστρεφόμενης ποσότητας ώστε να ελέγξει αν γίνεται σωστά η κλήση. Τα στοιχεία αυτά τα λαμβάνει

- είτε από τον ορισμό της συνάρτησης, *αν βρίσκεται στο ίδιο αρχείο με το σημείο που θα κληθεί και προηγείται αυτού*,
- είτε από τη *δήλωση* της συνάρτησης, η οποία πρέπει να βρίσκεται στο ίδιο αρχείο με την κλήση της. Ο ορισμός σε αυτή την περίπτωση μπορεί να βρίσκεται στο ίδιο ή άλλο αρχείο.

Οι δηλώσεις που αντιστοιχούν στις διάφορες μορφές του γενικού ορισμού είναι ακριβώς οι ίδιες με τον ορισμό αλλά το σώμα της συνάρτησης (το τμήμα μεταξύ των {}, συμπεριλαμβανομένων αυτών) έχει αντικατασταθεί από το ‘;’:

```
τύπος_επιστρεφόμενης_ποσότητας
όνομα(τύπος_ορίσματος_Α όρισμαΑ, τύπος_ορίσματος_Β όρισμαΒ,...);
```

ή

```
auto
όνομα(τύπος_ορίσματος_Α όρισμαΑ, τύπος_ορίσματος_Β όρισμαΒ,...)
-> τύπος_επιστρεφόμενης_ποσότητας;
```

ή

```
auto
όνομα(τύπος_ορίσματος_Α όρισμαΑ, τύπος_ορίσματος_Β όρισμαΒ,...);
```

ή

```
decltype(auto)
όνομα(τύπος_ορίσματος_Α όρισμαΑ, τύπος_ορίσματος_Β όρισμαΒ,...);
```

Σε μια δήλωση συνάρτησης μπορούν να παραληφθούν τα ονόματα των ορισμάτων ή να αλλάξουν σε σύγκριση με τον ορισμό.

Από εδώ και πέρα, θα χρησιμοποιείται στα παραδείγματα μόνο η πρώτη μορφή ορισμού και δήλωσης, αυτή με τον τύπο επιστρεφόμενης ποσότητας πριν το όνομα της συνάρτησης.

Η δήλωση μιας συνάρτησης επιτρέπεται να εμφανίζεται οπουδήποτε μπορούμε να ορίσουμε μια μεταβλητή. Συνήθως, οι δηλώσεις των συναρτήσεων που χρησιμοποιούμε σε ένα αρχείο, συγκεντρώνονται στην αρχή του αρχείου, μετά τις εντολές `#include`, έξω από κάθε συνάρτηση[1] ή, όπως θα δούμε στο §7.8, σε αρχείο header.

7.4 Κλήση

Η κλήση μιας συνάρτησης γίνεται παραθέτοντας το όνομά της, ακολουθούμενο σε παρενθέσεις από ποσότητες κατάλληλου τύπου ώστε να αντιστοιχούν στα ορίσματά της (ή να μπορούν να μετατραπούν σε αυτά). Οι ποσότητες αυτές πρέπει προφανώς να είναι ακριβώς τόσες όσα και τα ορίσματα, εκτός από την περίπτωση που στον ορισμό ή τη δήλωση της συνάρτησης καθορίζονται προεπιλεγμένες τιμές (§7.6) για κάποια από αυτά οπότε μπορούν να είναι λιγότερες.

Μια συνάρτηση που επιστρέφει τιμή μπορεί να χρησιμοποιηθεί όπου θα χρησιμοποιούσαμε σταθερή ποσότητα του ίδιου τύπου με την επιστρεφόμενη τιμή, π.χ. σε εκχώρηση, σύνθετη έκφραση, εκτύπωση κλπ.

Αν δεν επιθυμούμε να χρησιμοποιήσουμε το αποτέλεσμα μιας συνάρτησης, έχουμε τη δυνατότητα να μην το κάνουμε. Μπορούμε, δηλαδή, να έχουμε ως αυτόνομη εντολή την κλήση οποιασδήποτε συνάρτησης· προσέξτε την κλήση της read στο επόμενο παράδειγμα:

[1] εκτός οποιασδήποτε συνάρτησης επιτρέπεται ο ορισμός μεταβλητών. Καθώς είναι προσπελάσιμες από οποιαδήποτε συνάρτηση του αρχείου αποτελούν πηγή πολλών λαθών και γι᾽ αυτό πρέπει να αποφεύγεται η χρήση τους.

Παράδειγμα

```
#include <iostream>
#include <fstream>
#include <string>

//declarations
double func(double a, double b);  // The definition is elsewhere.
int read(double & a, std::string const & fname);
void print(char c);

// definition
int
read(double & a, std::string const & fname)
{
  std::ifstream file{fname};
  file >> a;

  return 0; // All ok
}

// definition
void
print(char c)  // definition
{
  std::cout << c << '\n';
}

int
main()
{
  double x{3.2};
  double y{3.4};
  double z{func(x,y)};  // Calls func with double, double.

  int i{3};
  double t{func(x,i)};  // Calls func with double, int.
                        // int is promoted to double.

  print('a'); // Calls a void function.

  double r;
```

```
   read(r, "input.dat");
   // Calls function and ignores returned value.
 }
```

Οι τιμές των ποσοτήτων που δίνονται κατά την κλήση στη συνάρτηση χρησιμοποιούνται ως αρχικές τιμές νέων μεταβλητών που αντιστοιχούν στα ορίσματα, *αν αυτά δεν είναι αναφορές*· οι τιμές τους, δηλαδή, *αντιγράφονται* στα ορίσματα. Οποιαδήποτε χρήση *και αλλαγή* των ορισμάτων στο σώμα της συνάρτησης αναφέρεται σε αυτές τις νέες μεταβλητές και όχι στις ποσότητες οι οποίες πέρασαν κατά την κλήση. Αν έχουμε όρισμα που είναι αναφορά, η ποσότητα που του δίνεται κατά την κλήση *ταυτίζεται* με το όρισμα. Οι νέες μεταβλητές ή οι αναφορές που αντιστοιχούν στα ορίσματα έχουν εμβέλεια το σώμα της συνάρτησης (ή αλλιώς, χρόνο «ζωής» τη διάρκεια κλήσης της συνάρτησης).

Τα παραπάνω έχουν ως συνέπεια να χρειάζεται ιδιαίτερος τρόπος δήλωσης των ορισμάτων αν επιθυμούμε να έχουμε τη δυνατότητα αλλαγής στις τιμές των αρχικών μας μεταβλητών. Π.χ.

```
#include <iostream>

void add3(double x);

void
add3(double x)
{
  x+=3.0;
}

int
main()
{
  double z{2.0};
  add3(z);           //  z = ???
  std::cout << z << '\n';   // z is 2.0
}
```

Στη συνάρτηση `add3()` του παραδείγματος, οποιαδήποτε μεταβολή στο όρισμά της γίνεται σε διαφορετική μεταβλητή από αυτή με την οποία κλήθηκε: το x δημιουργείται κατά την κλήση με αρχική τιμή αυτή που έχει το z (2.0), γίνεται 5.0 με την εντολή που περιέχεται στο σώμα, ενώ στο τέλος της συνάρτησης *καταστρέφεται*. Το z παραμένει 2.0.

Για να μπορέσουμε να εξαγάγουμε τις αλλαγές σε κάποιο όρισμα πρέπει αυτό να δηλωθεί είτε ως αναφορά, π.χ.

```
void add3(double & x) {  x+=3.0; }
```

είτε ως δείκτης, π.χ.

```
void add3(double * x) { *x+=3.0; }
```

Παρατηρήστε την αλλαγή στον τρόπο χρήσης του ορίσματος στο σώμα της συνάρτησης. Στην πρώτη περίπτωση, η κλήση παραμένει η ίδια, `add3(z)`, μόνο που τώρα το όνομα x είναι συνώνυμο του z· οποιαδήποτε αλλαγή στην τιμή του x εμφανίζεται αυτόματα και στο z. Στη δεύτερη, η κλήση αλλάζει· στη συνάρτηση περνά η *διεύθυνση του z*, `add3(&z)`. Το x «δείχνει» πλέον στη μεταβλητή z. Αλλαγή στο x δεν μπορεί να εξαχθεί· αντίθετα όμως, η μεταβολή του `*x` διατηρείται και μετά την επιστροφή της συνάρτησης. Το παραπάνω σημαίνει ότι αν το όρισμα είναι διάνυσμα ή ισοδύναμα, δείκτης σε διάνυσμα, δεν μπορούμε να το αλλάξουμε· *τα στοιχεία* του διανύσματος, όμως, μπορούν να μεταβληθούν.

Αναφέραμε ότι υπάρχουν ορίσματα μέσω των οποίων μπορεί να αλλάξει τιμή αυτό που «δείχνουν» ή στο οποίο αναφέρονται (είναι δείκτες ή αναφορές). Αν δεν επιθυμούμε να επιτρέπεται αυτή η τροποποίηση, καλό είναι να το υποδεικνύουμε στον μεταγλωττιστή προσθέτοντας στη δήλωση του ορίσματος το `const`. Έτσι, στον παρακάτω κώδικα

```
void
print(double const a[], int N)
{
  for (int i{0}; i < N; ++i){
    std::cout << a[i] << '\n';
  }
}
```

δηλώνουμε ότι τα στοιχεία του διανύσματος a είναι σταθερά μέσα στο σώμα της συνάρτησης. Αν τυχόν προσπαθούσαμε να τροποποιήσουμε κάποιο από αυτά, η μεταγλώττιση θα σταματούσε.

Προσέξτε το ακόλουθο παράδειγμα:

```
void
print(std::vector<double> const & a)
{
  for (int i{0}; i < a.size(); ++i){
    std::cout << a[i] << '\n';
  }
}
```

Το όρισμα της συνάρτησης έχει δηλωθεί ως αναφορά. Με τον τρόπο αυτό, *αποφεύγουμε την αντιγραφή* η οποία μπορεί να είναι χρονοβόρα, του πιθανώς μεγάλου vector που θα δοθεί ως όρισμα. Αν όμως αφήναμε το a απλώς ως αναφορά, θα επιτρέπαμε στη συνάρτηση να το τροποποιήσει. Κάτι τέτοιο δεν είναι απαραίτητο ή επιθυμητό στη συγκεκριμένη συνάρτηση και γι᾽ αυτό συμπληρώνουμε τη δήλωση του ορίσματος με το `const`. Το a, επομένως, είναι αναφορά σε σταθερό vector και η συνάρτηση είναι γρήγορη χωρίς να διακινδυνεύουμε την «ορθότητα» του προγράμματος.

7.4.1 Αναδρομική (recursive) κλήση

Στη C++ επιτρέπεται σε μια συνάρτηση να καλεί τον εαυτό της. Μια συνάρτηση που καλεί τον εαυτό της απλοποιεί πολύ οποιοδήποτε πρόβλημα, αρκεί ο αλγόριθμος επίλυσής του να μπορεί να γραφεί ώστε:

- ο υπολογισμός του αποτελέσματος να χρειάζεται την εφαρμογή του ίδιου αλγόριθμου αλλά σε διαφορετικές «τιμές» για τα δεδομένα εισόδου από αυτές που δέχτηκε αρχικά,
- ο αλγόριθμος να μπορεί να υπολογίσει το αποτέλεσμα για ένα συγκεκριμένο σύνολο «τιμών» με άλλο τρόπο και όχι με εφαρμογή του εαυτού του. Το συγκεκριμένο σύνολο πρέπει να μπορεί να το «φτάσει» σε κάποια από τις διαδοχικές εφαρμογές του εαυτού του.

Παράδειγμα

Ας δούμε πώς μπορούμε να υλοποιήσουμε μια συνάρτηση για το παραγοντικό ενός ακεραίου με *αναδρομικό (recursive)* τρόπο: σύμφωνα με τον ορισμό,

$$n! = \begin{cases} 1 \times 2 \times \cdots \times (n-1) \times n = (n-1)! \times n\ , & n > 0\ , \\ 1\ , & n = 0\ . \end{cases}$$

Επομένως, ο υπολογισμός του παραγοντικού του ακεραίου n απαιτεί τον υπολογισμό του παραγοντικού ενός άλλου ακεραίου (του n−1). Επιπλέον, για n= 1 ο υπολογισμός γίνεται με άλλο τρόπο (απευθείας) και όχι με υπολογισμό άλλου παραγοντικού.

Ο ορισμός που δόθηκε παραπάνω για το παραγοντικό εκφράζεται σε συνάρτηση C++ ως εξής

```
int
factorial(int n)
{
  int result;
  if (n > 0) {
    result = n * factorial(n-1);
  }
  if (n == 0) {
    result = 1;
  }
  return result;
}
```

Στη συνάρτηση αυτή έχουμε παραλείψει τους ελέγχους που κανονικά πρέπει να γίνονται (το n να μην είναι αρνητικό και το αποτέλεσμα να μπορεί να

αναπαρασταθεί στον τύπο της επιστρεφόμενης ποσότητας). Προσέξτε ότι η κλήση της `factorial()` στο σώμα της δεν είναι ανεξέλεγκτη· η ακολουθία

`factorial(n)`→`factorial(n-1)` →`factorial(n-2)`→ ...

σταματά (και επιστρέφεται τιμή που υπολογίζεται χωρίς την κλήση της) όταν το όρισμα γίνει 0.

Η παραπάνω υλοποίηση απλοποιείται αρκετά με τη χρήση του τριαδικού τελεστή '`?:`' (§3.5):

```
int
factorial(int n)
{
  return (n > 0 ? n * factorial(n-1) : 1);
}
```

Ας δούμε ένα άλλο, πιο πολύπλοκο παράδειγμα χρήσης της αναδρομικής συνάρτησης.

Παράδειγμα

Στη Μαθηματική Φυσική υπάρχουν οικογένειες πολυωνύμων που έχουν κάποιες ειδικές ιδιότητες. Μια από αυτές τις οικογένειες, τα πολυώνυμα Hermite, εμφανίζεται στην κβαντομηχανική αντιμετώπιση του αρμονικού ταλαντωτή. Κάποια από αυτά είναι

$$\begin{aligned} H_0(x) &= 1\,, \\ H_1(x) &= 2x\,, \\ H_2(x) &= 4x^2 - 2\,, \\ H_3(x) &= 8x^3 - 12x\,, \\ &\vdots \end{aligned}$$

Τα πολυώνυμα $H_n(x)$ ικανοποιούν την αναδρομική σχέση:

$$H_n(x) = 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x)\,, \qquad n \geq 2\,.$$

Παρατηρήστε ότι χρειαζόμαστε τα πολυώνυμα μηδενικού βαθμού ($H_0(x) = 1$) και πρώτου βαθμού ($H_1(x) = 2x$) για να υπολογίσουμε από την αναδρομική σχέση το πολυώνυμο δεύτερου βαθμού. Ανάλογα, χρειαζόμαστε τα $H_1(x)$ και $H_2(x)$ για να υπολογίσουμε το $H_3(x)$, κοκ.

Αν θελήσουμε να γράψουμε συνάρτηση που να υπολογίζει την τιμή των πολυωνύμων Hermite, $H_n(x)$, για κάποια τιμή του x, μπορούμε να μεταγράψουμε τον προηγούμενο μαθηματικό τύπο στην ακόλουθη αναδρομική συνάρτηση:

```
double
hermite(int n, double x)
{
  double h;

  if (n == 0) {
    h = 1.0;
  }

  if (n == 1) {
    h = 2.0*x;
  }

  if (n > 1) {
    h = 2.0 * (x*hermite(n-1,x) - (n-1)*hermite(n-2,x));
  }

  return h;
}
```

7.5 Παρατηρήσεις

7.5.1 Σταθερό όρισμα

Είναι περιττό να δηλώσουμε ως `const` ένα «απλό» όρισμα που δεν είναι αναφορά. Οι δηλώσεις

```
void f(double x);
```

και

```
void f(double const x);
```

είναι ισοδύναμες μεταξύ τους. Επίσης ισοδύναμες είναι και οι ακόλουθες:

```
void f(double * x);
void f(double * const x);
```

Προσέξτε ότι η τελευταία δήλωση, `void f(double * const x)`, διαφέρει από την `void f(double const * x)`, σύμφωνα με όσα αναφέραμε στην §2.18.

7.5.2 Σύνοψη δηλώσεων ορισμάτων

Ας συνοψίσουμε όσα αναφέραμε για τη δήλωση ορισμάτων ως προς τη δυνατότητα να εξάγουμε αλλαγές στην τιμή τους. Έχουμε τις ακόλουθες περιπτώσεις:

- Η τιμή της ποσότητας που θα περαστεί ως όρισμα δεν μπορεί να μεταβληθεί και *αντιγράφεται* στο x:

```
void f(double x); // argument cannot change
```

- Η τιμή της ποσότητας που θα περαστεί ως όρισμα μπορεί να μεταβληθεί και *ταυτίζεται* με το x:

```
void f(double & x);   // argument can change
```

- Η τιμή της ποσότητας που θα περαστεί ως όρισμα δεν μπορεί να μεταβληθεί και *ταυτίζεται* με το x:

```
void f(double const & x);//argument cannot change
```

- Η *διεύθυνση* που θα περαστεί ως όρισμα δεν μπορεί να μεταβληθεί, *αντιγράφεται* στο xp, ενώ μπορεί να αλλάξει το `*xp` (η τιμή στην οποία δείχνει):

```
void f(double * xp);
// argument cannot change, *xp can change
```

- Η *διεύθυνση* που θα περαστεί ως όρισμα δεν μπορεί να μεταβληθεί, ούτε όμως το `*xp` (η τιμή στην οποία δείχνει):

```
void f(double const * xp);
// argument cannot change, *xp cannot change
```

- Η *διεύθυνση* που θα περαστεί ως όρισμα μπορεί να μεταβληθεί, *ταυτίζεται* με το xp, ενώ μπορεί να αλλάξει και το `*xp` (η τιμή στην οποία δείχνει):

```
void f(double * & xp);
// argument can change, *xp can change
```

- Η «τιμή» του διανύσματος a δεν μπορεί να μεταβληθεί, μπορεί όμως να αλλάξουν τα στοιχεία του

```
void f(double a[]);
// argument cannot change, a[i] can change
```

- Η «τιμή» του διανύσματος a δεν μπορεί να μεταβληθεί, αλλά ούτε και τα στοιχεία του

```
void f(double const a[]);
// argument cannot change, a[i] cannot change
```

7.6 Προεπιλεγμένα ορίσματα

Ένα ιδιαίτερα χρήσιμο χαρακτηριστικό, ειδικά στους constructors όπως θα αναφέρουμε στο §14.5.1, είναι πως σε μια συνάρτηση μπορεί να δηλωθεί ότι ένα ή περισσότερα *από το τέλος*, ή και όλα τα ορίσματά της, παίρνουν προεπιλεγμένες τιμές:

```
int func(double a, double b = 5.0);
```

Η κλήση της `func()` μετά από τέτοια δήλωση μπορεί να γίνει είτε με δύο ορίσματα είτε με ένα όρισμα (που αντιστοιχεί στο a) οπότε το b παίρνει την προεπιλεγμένη τιμή, 5.0. Γενικότερα, οι ποσότητες που «περνούν» σε μια συνάρτηση κατά την κλήση της αντιστοιχίζονται στα ορίσματα διαδοχικά από την αρχή· αν είναι περισσότερες από αυτά η κλήση είναι λάθος, ενώ αν δεν επαρκούν, ο compiler αναζητά προκαθορισμένες τιμές για τα υπόλοιπα και δίνει λάθος αν δεν τις βρει.

Τα προκαθορισμένα ορίσματα καλό είναι να προσδιορίζονται στη *δήλωση* της συνάρτησης και όχι στον *ορισμό* της, καθώς μόνο η δήλωση είναι συνήθως «ορατή» στο σημείο κλήσης της.

7.7 Συνάρτηση ως όρισμα

Ένας χρήσιμος τύπος, ειδικά για όρισμα συνάρτησης, είναι ο δείκτης σε συνάρτηση.

Ας υποθέσουμε ότι θέλουμε να γράψουμε κώδικα με τον οποίο να σχεδιάζεται μια μαθηματική συνάρτηση $f(x)$ μίας μεταβλητής σε κάποιο διάστημα τιμών, να παράγεται δηλαδή μια σειρά σημείων (x, y). Μια απόπειρα είναι η ακόλουθη

```
#include <iostream>

double f(double x);

int
plot(double low, double high)
{
  double const step{(high - low) / 100};
  for (double x{low}; x < high; x+=step) {
    std::cout << x << '␣' << f(x) << '\n';
  }

  return 0;
}
```

Παρατηρήστε ότι η `plot()` δεν μπορεί να γενικευτεί για οποιαδήποτε συνάρτηση `f(x)` χωρίς να γίνει επέμβαση στον κώδικά της. Θα θέλαμε η `f(x)` να περνά στην

`plot()` ως όρισμα. Αυτό το επιτυγχάνουμε χρησιμοποιώντας ως τύπο ενός επιπλέον ορίσματος το δείκτη σε συνάρτηση. Για τη γενική δήλωση συνάρτησης

> τύπος_επιστρεφόμενης_ποσότητας
> όνομα(τύπος_ορίσματος_Α όρισμαΑ, τύπος_ορίσματος_Β όρισμαΒ,...);

ο δείκτης είναι

> τύπος_επιστρεφόμενης_ποσότητας
> (*όνομα_δείκτη)(τύπος_ορίσματος_Α όρισμαΑ,
> τύπος_ορίσματος_Β όρισμαΒ,...);

Οι παρενθέσεις γύρω από το «*όνομα_δείκτη» χρειάζονται καθώς το '*' (εξαγωγή τιμής από δείκτη) έχει μικρότερη προτεραιότητα από τις '()' (κλήση συνάρτησης), (δείτε τον Πίνακα 2.3). Μετά από τέτοια δήλωση, η μεταβλητή «όνομα_δείκτη» μπορεί να πάρει «τιμή» με εκχώρηση μιας συνάρτησης με αντίστοιχο πλήθος και είδος ορισμάτων και όταν ακολουθείται από κατάλληλες τιμές που να ανταποκρίνονται στα ορίσματα να επιστρέφει την τιμή που θα έδινε αυτή η συνάρτηση:

```
double f(double x);  // declaration of f(x)

double (*fptr)(double x); // declaration of a pointer

fptr = f; // assignment
double x{1.2};
auto y = f(x);
auto z = fptr(x);  // or z = (*fptr)(x);
// y == z
```

Με τους δείκτες σε συνάρτηση μας δίνεται η δυνατότητα να τροποποιήσουμε την plot ως εξής:

```
int
plot(double low, double high, double (*f)(double x))
{
  double const step{(high - low) / 100};
  for (double x{low}; x < high; x+=step) {
     std::cout << x << '␣' << f(x) << '\n';
  }

  return 0;
}
```

Έχουμε κρατήσει το σώμα της απαράλλαχτο και έχουμε προσθέσει, με κατάλληλο τρόπο, την `f(x)` στα ορίσματα. Έτσι μπορούμε να έχουμε

```
double sin(double x);   // sine
double cos(double x);   // cosine
```

```
double tan(double x);   // tangent
int plot(double low, double high, double (*f)(double x));

int
main()
{
  plot(1.0, 5.0, sin);  // plot of sine
  plot(1.0, 5.0, cos);  // plot of cosine
  plot(1.0, 5.0, tan);  // plot of tangent
}
```

Στην περίπτωση που θέλουμε να χρησιμοποιήσουμε την εντολή **using** (§2.15) για να ορίσουμε π.χ. τον τύπο «δείκτης σε συνάρτηση που επιστρέφει ακέραιο και δέχεται δύο πραγματικά ορίσματα», η σύνταξη είναι:

```
int func(double a, double b); // target function
using ftype = int (*)(double x, double y);
ftype g{func};   // declaration with assignment
```

7.8 Οργάνωση κώδικα

Συνήθως, οι δηλώσεις των συναρτήσεων που καλεί ένα τμήμα κώδικα συγκεντρώνονται σε ένα ή περισσότερους headers, αρχεία με συνήθη κατάληξη `.h`,[2] τα οποία συμπεριλαμβάνονται κατά την προεπεξεργασία του συγκεκριμένου τμήματος κώδικα· εμφανίζονται δηλαδή στην αρχή οδηγίες όπως η

```
#include "name.h"
```

όπου *name.h* το όνομα του header, όπως το αντιλαμβάνεται το λειτουργικό σύστημα (επομένως, μπορεί να περιλαμβάνεται και το path στο όνομα αυτό). Προσέξτε ότι οι headers που ορίζει ο προγραμματιστής–και η «φυσική» τους μορφή είναι αρχεία–περικλείονται σε διπλά εισαγωγικά (`"`). Αντίθετα, οι headers του συστήματος–που δεν είναι απαραιτήτως αρχεία–περικλείονται σε '<>'.

Με την συμπερίληψη των κατάλληλων headers, ο compiler γνωρίζει τον τρόπο κλήσης των συναρτήσεων που χρειάζεται ένα τμήμα κώδικα. Οι ορισμοί, δηλαδή η παράθεση του σώματος των συναρτήσεων, παρουσιάζονται κανονικά σε ένα ή περισσότερα αρχεία κώδικα, σε αντιστοιχία με τους headers.

Παράδειγμα

Έστω οι συναρτήσεις min/max που επιστρέφουν το μικρότερο/μεγαλύτερο από δύο αριθμούς:

min: αν όρισμαA < όρισμαB επίστρεψε το όρισμαA αλλιώς επίστρεψε το όρι-

[2] εξαρτώμενη από τον compiler.

σμαB

max: αν όρισμαA < όρισμαB επίστρεψε το όρισμαB αλλιώς επίστρεψε το όρισμαA

Μπορούμε να οργανώσουμε τον κώδικα ως εξής: στο αρχείο με όνομα π.χ. *utilities.h* θα γράψουμε τις δηλώσεις τους (π.χ. για ορίσματα τύπου `double`),

```
// utilities.h

// declarations
double min(double a, double b);
double max(double a, double b);
```

και στο αρχείο με όνομα *utilities.cpp* τους ορισμούς τους,

```
// utilities.cpp
#include "utilities.h"
// Not necessary but good practice

// definitions
double
min(double a, double b)
{
  return a<b ? a : b;
}
double
max(double a, double b)
{
  return a>b ? a : b;
}
```

Η χρήση τους σε ένα πρόγραμμα γίνεται ως εξής:

- συμπεριλαμβάνουμε το *utilities.h* στον κώδικά μας, π.χ.

```
#include <iostream>
#include "utilities.h"

int
main()
{
  double a, b;
  std::cout << "Give␣two␣real␣numbers\n";
  std::cin >> a >> b;
  std::cout << "Max␣is␣" << max(a,b) << '\n';
```

```
    std::cout << "Min␣is␣" << min(a,b) << '\n';
}
```

- κάνουμε ξεχωριστό compile στο *utilities.cpp* και στο αρχείο που περιέχει τη `main()` με την κατάλληλη διαδικασία για τον compiler που χρησιμοποιούμε και
- «ενώνουμε» τα ξεχωριστά τμήματα του συνολικού προγράμματος στο τελευταίο στάδιο πριν τη δημιουργία εκτελέσιμου αρχείου, στη φάση του linking.

7.9 main()

Έχουμε ήδη χρησιμοποιήσει ένα από τους δύο τρόπους σύνταξης της βασικής συνάρτησης κάθε ολοκληρωμένου προγράμματος, της `main()`:

```
int main() {.....}
```

Ο δεύτερος τρόπος επιτρέπει να «περάσουν» ορίσματα στη `main()` από το λειτουργικό σύστημα (το οποίο καλεί τη συνάρτηση) *κατά την έναρξη εκτέλεσης του προγράμματος*:

```
int main(int argc, char* argv[]) {........}
```

Ισοδύναμος με αυτόν τον τρόπο δήλωσης (δείτε την §7.4) είναι και ο εξής:

```
int main(int argc, char** argv) {........}
```

Το πρώτο όρισμα, ένας ακέραιος με το συμβατικό όνομα argc, παίρνει τιμή κατά 1 μεγαλύτερη από το πλήθος των ορισμάτων που δίνονται στη `main()` ή 0, αν το λειτουργικό σύστημα δεν μπορεί να περάσει ορίσματα. Το δεύτερο, ένας διάνυσμα δεικτών σε **char**, έχει διάσταση argc+1 και περιέχει σε μορφή C-style string τα ορίσματα. Η τιμή `argv[0]` είναι πάντα το όνομα με το οποίο έγινε η κλήση του προγράμματος, τα `argv[1]`, `argv[2]`, ... το πρώτο, δεύτερο, ... όρισμα, ενώ η τελευταία τιμή, `argv[argc]`, είναι 0. Το λειτουργικό σύστημα UNIX θεωρεί ως ορίσματα τις «λέξεις» (σειρές χαρακτήρων που περιβάλλονται από κενά) που ακολουθούν το όνομα του προγράμματος στη γραμμή εντολών κατά την κλήση του. Έτσι, αν η κλήση του εκτελέσιμου *a.out* είναι η

```
./a.out 12 input.dat output.dat 4.5
```

στη `main()`, *αν έχει γίνει ο ορισμός με τη δεύτερη μορφή*, το argc είναι 5, και οι τιμές του argv είναι:

```
argv[0] = "./a.out";
argv[1] = "12";
argv[2] = "input.dat";
argv[3] = "output.dat";
```

```
argv[4] = "4.5";
argv[5] = 0;
```

Προσέξτε ότι τα ορίσματα 1 και 4 δεν «περνούν» ως αριθμοί. Για να χρησιμοποιηθούν ως τέτοιοι στη `main()` πρέπει να μετατραπούν. Για το σκοπό αυτό παρέχονται από τη C++ στο header `<cstdlib>` οι συναρτήσεις:

```
int atoi(char const * p);       // C-string to int
long atol(char const * p);      // C-string to long int
double atof(char const * p);    // C-string to double
```

καθώς και η πιο γενική `strtod()`. Οι παραπάνω ορίζονται στο χώρο ονομάτων std. Με τη χρήση αυτών μπορούμε να έχουμε

```
#include <cstdlib>
#include <fstream>

int
main(int argc, char *argv[])
{
  int n{std::atoi(argv[1])};
  // n gets the value of the first argument

  std::ifstream filein{argv[2]};
  // open input file. Name is given by argv[2].

  std::ofstream fileout{argv[3]};
  // open output file. Name is given by argv[3].

  double x{std::atof(argv[4])};
  // x gets the value of the fourth argument

  ....
}
```

7.10 overloading

Ας εξετάσουμε την περίπτωση που θέλουμε να γράψουμε συναρτήσεις που να εκτελούν πολλαπλασιασμό αριθμού με διάνυσμα, αριθμού με διδιάστατο πίνακα, ή πολλαπλασιασμό δύο διδιάστατων πινάκων. Οι πράξεις γίνονται με διαφορετικούς αλγορίθμους αλλά στο χώρο των πινάκων περιγράφονται με το ίδιο όνομα. Η C++ μας δίνει τη δυνατότητα (overloading) να χρησιμοποιήσουμε για τις συναρτήσεις που υλοποιούν αυτούς τους αλγορίθμους το ίδιο όνομα, παρόλο που θα δέχονται ορίσματα διαφορετικού τύπου και, συνολικά, θα είναι διαφορετικές. Δεν είμαστε υποχρεωμένοι να επινοούμε μοναδικά ονόματα για τις συναρτήσεις μας έτσι ώστε

να μη «συγκρούονται» με άλλες παρόμοιες. Θα δούμε παρακάτω τις μαθηματικές συναρτήσεις της C++ που ορίζονται με το ίδιο όνομα παρόλο που πιθανόν εκτελείται διαφορετικός αλγόριθμος αν τα ορίσματα είναι `double`, `float` ή `long double`.

Όταν γίνεται η κλήση μιας συνάρτησης με πολλούς ορισμούς, ο compiler επιλέγει τον κατάλληλο με βάση τα ορίσματα (πλήθος και τύπο) που περνούν. Δε λαμβάνει υπόψη, όμως, τον τύπο της επιστρεφόμενης ποσότητας της συνάρτησης. Αν δε βρει μία μόνο συνάρτηση που να ταιριάζει ακριβώς, παίρνει υπόψη του τις «αυτόματες» μετατροπές (π.χ. `bool`, `char`, `short int` σε `int`, `float` σε `double`,...). Αν πάλι δε βρεθεί αντίστοιχη συνάρτηση, εξετάζει τα ορίσματα αφού μετατρέψει `int` σε `double`, `double` σε `long double`, δείκτες σε `void*`, κλπ. Υπάρχουν γενικά πολύπλοκοι κανόνες για την επιλογή της κατάλληλης, μοναδικής συνάρτησης· αν σε κάποιο στάδιο εμφανιστούν περισσότερες από μία «ισότιμες» επιλογές ή δε βρεθεί καμία, η κλήση είναι λάθος.

Καλό είναι να γράφονται οι συναρτήσεις με τα ακριβή ορίσματα (κατά τύπο και αριθμό) με τα οποία θα κληθούν ώστε να μη χρειαστεί να γίνονται μετατροπές από τον compiler που πιθανόν καλέσουν διαφορετική συνάρτηση από αυτή που είχε σκοπό ο προγραμματιστής.

7.11 Υπόδειγμα (template) συνάρτησης

Ένα ιδιαίτερα σημαντικό χαρακτηριστικό της C++ έναντι πολλών άλλων γλωσσών προγραμματισμού είναι η υποστήριξη των templates (υποδείγματα). Για συναρτήσεις αυτό σημαίνει ότι μπορούμε να τις παραμετροποιήσουμε όχι μόνο με ορίσματα αλλά και με τύπο ποσοτήτων στη λίστα ορισμάτων ή στο σώμα της συνάρτησης. Πάρτε για παράδειγμα μια συνάρτηση που αλλάζει τιμές μεταξύ των δύο ορισμάτων της (swap). Θα θέλαμε να έχουμε τέτοια συνάρτηση για όλους τους τύπους μεταβλητών[3], είτε είναι ενσωματωμένοι (`int`, `float`,...), είτε πρόκειται για τύπους που ορίζει ο προγραμματιστής (κλάσεις, Κεφάλαιο 14). Η δυνατότητα για overloading είναι ευπρόσδεκτη καθώς μπορούμε να χρησιμοποιήσουμε το ίδιο όνομα για όλες αυτές τις συναρτήσεις. Προσέξτε ότι όλες οι παραλλαγές διαφέρουν μόνο στον τύπο των μεταβλητών και όχι στον αλγόριθμο:

```
void
swap(int & a, int & b)
{
  int const temp{b};
  b = a;
  a = temp;
}

void
swap(float & a, float & b)
```

[3] έχουμε ήδη, την `std::swap()` στο `<utility>` (§9.2.4).

```
{
  float const temp{b};
  b = a;
  a = temp;
}

void
swap(double & a, double & b)
{
  double const temp{b};
  b = a;
  a = temp;
}

...
...
```

Εύκολα αντιλαμβανόμαστε ότι είναι κουραστικό και δύσκολο στη διόρθωση ή την αναβάθμιση το να επαναλαμβάνει κανείς ουσιαστικά τον ίδιο κώδικα κάθε φορά που θέλει να υποστηρίξει μια συνάρτηση για ένα νέο τύπο. Η C++ δίνει τη δυνατότητα να γράφει ο compiler την αναγκαία συνάρτηση κάθε φορά, αρκεί ο προγραμματιστής να του έχει παρουσιάσει ένα υπόδειγμα (template) για το πώς να το κάνει. Η σύνταξη του template γίνεται πιο εύκολα κατανοητή με ένα παράδειγμα:

```
template <typename T>
void
swap(T & a, T & b)
{
  T const temp{b};
  b = a;
  a = temp;
}
```

Η προσθήκη στον ορισμό της συνάρτησης του `template <typename T>` (που αποτελεί μέρος της δήλωσης) ορίζει ότι το όνομα T (που θα μπορούσε να είναι οποιοδήποτε της επιλογής του προγραμματιστή) συμβολίζει ένα τύπο. Με αυτό τον τύπο μπορούμε να δηλώσουμε τα ορίσματα, την επιστρεφόμενη τιμή της συνάρτησης, καθώς και όποιες ποσότητες χρειάζονται στο σώμα της. Γενικά μπορούν να υπάρχουν περισσότερα από ένα τέτοια ονόματα (παράμετροι του template). Επιπλέον, οι τελευταίες παράμετροι επιτρέπεται να έχουν προεπιλεγμένες «τιμές»:

```
template <typename X, typename Y = double, typename Z = int>
...
```

Η κλήση ενός template συνάρτησης γίνεται βάζοντας σε <> τους τύπους που αντιστοιχούν στις παραμέτρους του template κατά τη συγκεκριμένη κλήση, μεταξύ του ονόματος της συνάρτησης και της λίστας των ορισμάτων:

```
double a{2.0};
double b{3.0};
swap<double>(a,b);
```

Με αυτό τον τρόπο, δημιουργούμε ρητά μια εκδοχή του template. Στην περίπτωση που οι παράμετροι του template μπορούν να αναγνωριστούν από τον τύπο των ορισμάτων, η κλήση μπορεί να παραλείψει τη ρητή δήλωσή τους. Η κλήση στο παραπάνω παράδειγμα είναι ισοδύναμη με την `swap(a,b)`. Προσέξτε ότι αν η συνάρτηση περνά ως όρισμα σε άλλη, δεν μπορούμε να παραλείψουμε τον προσδιορισμό των παραμέτρων καθώς δεν μπορούν να εξαχθούν από τα (ανύπαρκτα) ορίσματα.

Εκτός από τύπος, μια παράμετρος ενός template μπορεί να είναι σταθερή ποσότητα, γνωστή κατά τη μεταγλώττιση, κάποιου ακέραιου τύπου ή **`enum class`**[4].

Έστω, π.χ., ότι θέλουμε να γράψουμε μια συνάρτηση που να ελέγχει αν το όρισμά της είναι ακέραιο πολλαπλάσιο ενός δεδομένου αριθμού. Μπορούμε να την υλοποιήσουμε (χωρίς ελέγχους για τα ορίσματα) ως εξής:

```
bool
mult(int a, int b)
{
  return !(a%b);
}
```

Η κλήση της είναι, βέβαια `mult(a,b)`. Εναλλακτικά, αν το `b` είναι γνωστό κατά τη μεταγλώττιση, μπορούμε να ορίσουμε το ακόλουθο template:

```
template<int b>
bool
mult(int a)
{
  return !(a%b);
}
```

Η κλήση τότε είναι `mult<b>(a)`.

Θα δούμε σε επόμενο κεφάλαιο ποια χρησιμότητα έχει αυτή η μορφή του template.

Ο τρόπος οργάνωσης του κώδικα σε αρχεία είναι ιδιαίτερος στην περίπτωση που περιλαμβάνεται μια συνάρτηση template. Πρέπει να περιλαμβάνεται στο header όχι μόνο η δήλωση αλλά και ο ορισμός του template συνάρτησης.

7.11.1 Εξειδίκευση

Στην περίπτωση που ο γενικός αλγόριθμος που προκύπτει από ένα template δε μας ικανοποιεί (π.χ. ως προς την ταχύτητα ή τον αλγόριθμο που υλοποιεί)

[4] ή δείκτης σε συνάρτηση ή αντικείμενο, αναφορά σε συνάρτηση ή σταθερό αντικείμενο, ή δείκτης σε μέλος κλάσης

για κάποιο συγκεκριμένο σύνολο παραμέτρων, μπορούμε να δηλώσουμε προς τον compiler ότι πρέπει να χρησιμοποιεί άλλη ρουτίνα όποτε χρειαστεί να παραγάγει κώδικα για τις συγκεκριμένες παραμέτρους. Έτσι π.χ. για ακέραιους αριθμούς στη `swap()` θα θέλαμε να χρησιμοποιεί τον αλγόριθμο XOR swap αντί για το γενικό που δόθηκε παραπάνω. Μπορούμε να συμπληρώσουμε το αρχείο που περιέχει το υπόδειγμα για τη `swap()` με τον εξής κώδικα:

```
template<>
void
swap(int & x, int & y)
{
  if (&x != &y) {
     x^=y;
     y^=x;
     x^=y;
  }
}
```

Σε αυτή την περίπτωση, η κλήση `swap<int>(a,b)` (ή, ισοδύναμα, η κλήση της swap με ακέραια ορίσματα) χρησιμοποιεί τον ειδικό αλγόριθμο, ενώ για οποιαδήποτε άλλη παράμετρο καλείται ο γενικός.

Παρατηρήστε ότι απαλοίφουμε από το template την παράμετρο που εξειδικεύουμε, δηλαδή, αφαιρούμε το `typename T`, και όπου εμφανίζεται η αυτή γράφουμε το συγκεκριμένο τύπο για τον οποίο εξειδικεύουμε.

Στην περίπτωση που θέλουμε να κάνουμε μερική εξειδίκευση για κάποιες παραμέτρους ενός template, σχηματίζουμε νέο template από το αρχικό, με λιγότερες παραμέτρους, στο οποίο έχουμε προσδιορίσει ρητά κάποιους τύπους. Έτσι, αν έχουμε το

```
template<typename T1, typename T2>
void
f(T1 a, T2 b)
{
...
}
```

δύο μερικώς εξειδικευμένα templates είναι

```
template<typename T1>
void
f(T1 a, int b)
{
...
}

template<typename T2>
```

```
void
f(double a, T2 b)
{
...
}
```

Παρατηρήστε ότι αποτελούν νέα templates.

7.12 Συνάρτηση constexpr

Μια συνάρτηση μπορεί να προσδιοριστεί ως `constexpr` αν είναι δυνατό να εκτελεστεί κατά τη διάρκεια της μεταγλώττισης (και όχι μόνο κατά την εκτέλεση του προγράμματος). Εννοείται ότι τα τυχόν ορίσματά της πρέπει να είναι και αυτά ποσότητες γνωστές στον compiler. Τέτοιες συναρτήσεις μπορούν να χρησιμοποιηθούν σε σταθερές εκφράσεις, για απόδοση τιμής σε σταθερές `constexpr`, κλπ.

Μια συνάρτηση `constexpr` επιτρέπεται να περιέχει ουσιαστικά οποιαδήποτε εντολή εκτός από εντολή `goto`, `try` και δηλώσεις μεταβλητών που είναι στατικές ή δεν αποκτούν αρχική τιμή.

Παραδείγματα

Μια συνάρτηση που βρίσκει το μεγαλύτερο δύο ακεραίων μπορεί να οριστεί ως `constexpr` ως εξής:

```
constexpr
int
max(int a, int b)
{
  return a > b ? a : b;
}
```

Η συνάρτηση που υπολογίζει το παραγοντικό με αναδρομική κλήση του εαυτού της, είναι κατάλληλη να οριστεί ως `constexpr`:

```
constexpr
int
factorial(int n)
{
  return  (n > 0 ? n * factorial(n-1) : 1);
}
```

Συνάρτηση ορισμένη με το `constexpr` πρέπει να είναι πλήρως γνωστή στον compiler πριν χρησιμοποιηθεί, δεν αρκεί μόνο η δήλωσή της όπως στις υπόλοιπες· ο compiler πρέπει να την εκτελέσει κατά τη μεταγλώττιση του κώδικα που την καλεί. Ο ορισμός μιας συνάρτησης `constexpr` πρέπει να «ορατός» στο σημείο που θα κληθεί αυτή και, υποχρεωτικά, ο ίδιος σε οποιοδήποτε άλλο σημείο κλήσης της.

Επομένως, ο ορισμός της πρέπει να περιλαμβάνεται στο header που κανονικά θα είχε μόνο τη δήλωσή της.

7.13 inline

Η εκτέλεση «μικρού» κώδικα μέσω κλήσης συνάρτησης που τον περιέχει είναι γενικά πιο χρονοβόρα απ᾽ ό,τι αν παρατεθούν αυτούσιες οι εντολές στο σημείο κλήσης. Η C++ δίνει τη δυνατότητα να ενημερώσουμε τον compiler ότι μια συνάρτηση είναι κατάλληλα μικρή και πρόκειται να χρησιμοποιηθεί συχνά ώστε, *εάν γίνεται*, να υποκαταστήσει τις κλήσεις της απευθείας με τον κώδικα που περιέχει. Με αυτόν τον τρόπο μπορούμε να εξαλείψουμε την καθυστέρηση της κλήσης. Η ενημέρωση του compiler γίνεται χρησιμοποιώντας την προκαθορισμένη λέξη `inline` στον ορισμό της συνάρτησης, πριν τον τύπο της επιστρεφόμενης ποσότητας. Παραδείγματος χάριν, μια συνάρτηση που βρίσκει το μεγαλύτερο δύο ακεραίων και πρόκειται να χρησιμοποιηθεί συχνά, μπορεί να οριστεί ως εξής:

```
inline
int
max(int a, int b)
{
  return a > b ? a : b;
}
```

Προφανώς δεν έχει νόημα, και είναι λάθος, να οριστεί `inline` η `main()`.

Όπως και στην περίπτωση της συνάρτησης `constexpr`, συνάρτηση ορισμένη με το `inline` πρέπει να είναι πλήρως γνωστή στον compiler πριν χρησιμοποιηθεί, δεν αρκεί μόνο η δήλωσή της όπως στις υπόλοιπες. Επομένως, ο ορισμός της πρέπει να περιλαμβάνεται στο header που κανονικά θα είχε μόνο τη δήλωσή της.

7.14 Στατικές ποσότητες

Οι μεταβλητές που ορίζονται στο σώμα μιας συνάρτησης έχουν διάρκεια ζωής όση και η διάρκεια εκτέλεσης της συνάρτησης. Επομένως, δημιουργούνται όταν η ροή του προγράμματος φτάσει στο σημείο δήλωσής τους στη συνάρτηση και καταστρέφονται όταν η ροή φύγει από την εμβέλειά τους. Μπορούμε να ορίσουμε κατάλληλα κάποια μεταβλητή έτσι ώστε να δημιουργηθεί και να πάρει αρχική τιμή (0 ή αυτή που θα δοθεί κατά τον ορισμό της) μόνο την πρώτη φορά που η ροή θα συναντήσει τη δήλωσή της και, επιπλέον, να μην καταστραφεί κατά την έξοδο από τη συνάρτηση. Αυτό γίνεται προσθέτοντας στον ορισμό της μεταβλητής την προκαθορισμένη λέξη `static`:

```
void
func(double a)
{
```

```
  static int howmany{0};
  // .....
  ++howmany;
}
```

Η μεταβλητή howmany στο παράδειγμα ουσιαστικά μετρά πόσες φορές κλήθηκε η συνάρτηση. Εννοείται ότι «φαίνεται» μόνο μέσα στη συνάρτηση `func()`.

Η ροή μεταγλώττισης του κώδικα στο προηγούμενο παράδειγμα έχει ως εξής: Την πρώτη φορά που ο compiler συναντήσει τη δήλωση με το **`static`**, δημιουργείται η δηλούμενη ποσότητα και της αποδίδεται η αρχική τιμή που προσδιορίζει η εντολή (ή 0 αν δεν υπάρχει αρχική τιμή). Προφανώς, η αρχική τιμή πρέπει να είναι τέτοια ώστε να μπορεί να την υπολογίσει ο μεταγλωττιστής. Όταν η συνάρτηση στην οποία δηλώνεται αυτή η ποσότητα ολοκληρωθεί, η ροή, δηλαδή, συναντήσει το καταληκτικό '}', η ποσότητα *δεν καταστρέφεται*. Την επόμενη φορά που η ροή συναντήσει τη δήλωση της στατικής ποσότητας, δεν τη δημιουργεί ξανά, ούτε της αποδίδει τιμή, αλλά χρησιμοποιεί την ποσότητα που ήδη υπάρχει (με όποια τιμή έχει).

Προφανώς, μια στατική ποσότητα καταστρέφεται (δηλαδή, ελευθερώνεται η αντίστοιχη μνήμη) μόνο όταν ολοκληρωθεί το πρόγραμμα[5].

7.15 Μαθηματικές συναρτήσεις της C++

Η C++ παρέχει μέσω της Standard Library ορισμένες μαθηματικές συναρτήσεις, χρήσιμες για συνήθεις υπολογισμούς σε επιστημονικούς κώδικες. Οι δηλώσεις των περισσότερων συναρτήσεων περιέχονται στο header `<cmath>` και ορίζονται με το ίδιο όνομα για πραγματικούς αριθμούς (τύπου **`float`**, **`double`**, **`long double`**) ή ποσότητες ακέραιου τύπου[6] (§2.5). Στον Πίνακα 7.1 παρατίθενται οι δηλώσεις για **`double`**. Όλες οι συναρτήσεις του `<cmath>`, όπως και όλη η Standard Library, ανήκουν στο χώρο ονομάτων `std`.

Για ιστορικούς λόγους, κάποιες μαθηματικές συναρτήσεις για ακέραιους και οι συνοδευτικές τους δομές δηλώνονται στο `<cstdlib>`. Και αυτές βέβαια, ανήκουν στο χώρο ονομάτων `std`. Τέτοιες είναι:

- οι συναρτήσεις απόλυτης τιμής:

  `int abs(int x)`.
  : Επιστρέφει την απόλυτη τιμή του x. Ορίζεται και για τους τύπους ορισμάτος **`long int`** και **`long long int`**, με αντίστοιχο τύπο επιστρεφόμενης ποσότητας.

  `long int labs(long int x)`.
  : Είναι ουσιαστικά άλλο όνομα για τη συνάρτηση

[5] αρκεί να μην διακοπεί το πρόγραμμα με την **`std::abort()`** ή ανάλογη συνάρτηση.
[6] τις οποίες μετατρέπουν σε **`double`** για να κάνουν τη σχετική πράξη.

`long int abs(long int x);`

`long long int llabs(long long int x);`.
Είναι ουσιαστικά ένα άλλο όνομα για την

`long long int abs(long long int x);`

- οι συναρτήσεις για πηλίκο και υπόλοιπο:

`div_t div(int n, int d)`.
Υπολογίζει το πηλίκο και το υπόλοιπο της διαίρεσης του n με το d και τα επιστρέφει στα μέλη μιας ποσότητας τύπου δομής `std::div_t` με ονόματα `quot` και `rem` αντίστοιχα. Ορίζεται και για `long int`, `long long int` με τύπο επιστρεφόμενης ποσότητας `std::ldiv_t` και `std::lldiv_t` αντίστοιχα.

`ldiv_t ldiv(long int n, long int d)`
Είναι ουσιαστικά άλλο όνομα για τη συνάρτηση

`ldiv_t div(long int n, long int d);`

`lldiv_t lldiv(long long int n, long long int d)`.
Είναι ουσιαστικά άλλο όνομα για τη συνάρτηση

`lldiv_t div(long long int n, long long int d);`

Στο `<cinttypes>` ορίζονται ακόμα οι συναρτήσεις

```
intmax_t imaxabs(intmax_t x);
imaxdiv_t imaxdiv(intmax_t n, intmax_t d);
```

Αποτελούν overloads των παραπάνω για τον τύπο `std::intmax_t` του `<cstdint>`. Ο τύπος `std::imaxdiv_t` έχει δύο μέλη τύπου `std::intmax_t`, με ονόματα `quot` και `rem`.

Δείτε την §8.4 για το μηχανισμό μέσω του οποίου οι μαθηματικές συναρτήσεις ενημερώνουν για σφάλματα κατά την κλήση τους.

Πίνακας 7.1: Επιλεγμένες συναρτήσεις του <cmath> (μέρος α').

Συνάρτηση	**Επιστρεφόμενη τιμή**	**Παρατηρήσεις**
	Τριγωνομετρικές	
`double cos(double x)`	Συνημίτονο του x.	Το x σε rad.
`double sin(double x)`	Ημίτονο του x.	Το x σε rad.
`double tan(double x)`	Εφαπτομένη του x.	Το x σε rad.
`double acos(double x)`	Τόξο συνημιτόνου του x.	Το x στο $[-1, 1]$, το αποτέλεσμα στο $[0, \pi]$ σε rad.
`double asin(double x)`	Τόξο ημιτόνου του x.	Το x στο $[-1, 1]$, το αποτέλεσμα στο $[-\pi/2, \pi/2]$ σε rad.
`double atan(double x)`	Τόξο εφαπτομένης του x.	Το αποτέλεσμα στο $[-\pi/2, \pi/2]$ σε rad.
`double atan2(double x, double y)`	Τόξο εφαπτομένης $\tan^{-1}(x/y)$.	Τα πρόσημα των x,y καθορίζουν το τεταρτημόριο. Το αποτέλεσμα στο $[-\pi, \pi]$ σε rad.
	Υπερβολικές	
`double cosh(double x)`	Υπερβολικό συνημίτονο του x.	
`double sinh(double x)`	Υπερβολικό ημίτονο του x.	
`double tanh(double x)`	Υπερβολική εφαπτομένη του x.	
`double acosh(double x)`	Τόξο υπερβολικού συνημιτόνου του x.	Το $x \geq 1$, το αποτέλεσμα είναι μη αρνητικό.
`double asinh(double x)`	Τόξο υπερβολικού ημιτόνου του x.	
`double atanh(double x)`	Τόξο υπερβολικής εφαπτομένης του x.	Το x στο $(-1, 1)$.
	Δυνάμεις	
`double pow(double x, double a)`	Ύψωση σε δύναμη, x^a.	Πρέπει να ισχύει `a>0` αν `x=0` και ο a να είναι ακέραιος αν `x<0`.
`double sqrt(double x)`	Η τετραγωνική ρίζα του x.	Το x μη αρνητικό.
`double cbrt(double x)`	Η κυβική ρίζα του x.	
`double hypot(double x, double y)`	$\sqrt{x^2 + y^2}$.	Χωρίς overflow/underflow στις πράξεις.

Πίνακας 7.1: Επιλεγμένες συναρτήσεις του <cmath> (μέρος β').

Συνάρτηση	Επιστρεφόμενη τιμή	Παρατηρήσεις
	Εκθετικές/Λογαριθμικές	
`double exp(double x)`	Εκθετικό του x (e^x).	
`double exp2(double x)`	2^x.	
`double expm1(double x)`	Εκθετικό του x, μείον 1 ($e^x - 1$).	Δίνει πιο ακριβές αποτέλεσμα από το `exp(x)-1` για x με μικρό μέτρο.
`double log(double x)`	Φυσικός λογάριθμος του x ($\ln x$).	Το x μη αρνητικό.
`double log2(double x)`	Δυαδικός λογάριθμος του x ($\log_2 x$).	Το x μη αρνητικό.
`double log10(double x)`	Δεκαδικός λογάριθμος του x ($\log_{10} x$).	Το x μη αρνητικό.
`double log1p(double x)`	Φυσικός λογάριθμος του $1 + x$.	Δίνει πιο ακριβές αποτέλεσμα από το `log(1+x)` για x με μικρό μέτρο.
`double modf(double x, double* p)`	Το δεκαδικό μέρος του x, με το πρόσημο του x.	Το ακέραιο μέρος στο `*p`, με το πρόσημο του x.
`double frexp(double d, int* p)`	Βρίσκει x στο $[0.5, 1)$ και y ώστε $d = x\,2^y$. Επιστρέφει το x.	Θέτει το y στο `*p`.
`double ldexp(double d, int i)`	$d\,2^i$.	
	Στρογγυλοποιήσεις	
`double ceil(double x)`	Ο μικρότερος ακέραιος που δεν είναι μικρότερος από το x, ως πραγματικός.	
`double floor(double x)`	Ο μεγαλύτερος ακέραιος που δεν είναι μεγαλύτερος από το x, ως πραγματικός.	
`double trunc(double x)`	Ο πλησιέστερος ακέραιος που δεν έχει μεγαλύτερο μέτρο από τον x, ως πραγματικός.	
`double round(double x)`	Ο πλησιέστερος ακέραιος στον x, ως πραγματικός.	Αν το x έχει δεκαδικό μέρος το 0.5 η στρογγυλοποίηση γίνεται στον ακέραιο μεγαλύτερου μέτρου.
`long int lround(double x)`	Ο πλησιέστερος ακέραιος στον x, ως long int.	Αν το x έχει δεκαδικό μέρος το 0.5 η στρογγυλοποίηση γίνεται στον ακέραιο μεγαλύτερου μέτρου.
`long long int llround(double x)`	Ο πλησιέστερος ακέραιος στον x, ως long long int.	Αν το x έχει δεκαδικό μέρος το 0.5 η στρογγυλοποίηση γίνεται στον ακέραιο μεγαλύτερου μέτρου.

Πίνακας 7.1: Επιλεγμένες συναρτήσεις του `<cmath>` (μέρος γ').

Συνάρτηση	**Επιστρεφόμενη τιμή**	**Παρατηρήσεις**
	Υπολογισμός υπολοίπου	
`double fmod(double x, double y)`	x-n*y, όπου n το ακέραιο μέρος του x/y.	Το n στρογγυλοποιείται προς το 0.
`double remainder(double x, double y)`	x-n*y, όπου n το ακέραιο μέρος του x/y. Το n στρογγυλοποιείται προς τον πλησιέστερο ακέραιο.	Αν το x/y έχει δεκαδικό μέρος 0.5 η στρογγυλοποίηση του n γίνεται προς τον πλησιέστερο *άρτιο* ακέραιο.
`double remquo(double x, double y, int* q)`	Το `std::remainder(x,y)`.	Το πρόσημο και τουλάχιστον τα τρία τελευταία bit του πηλίκου της διαίρεσης του x με το y αποθηκεύονται στον ακέραιο `*q`.
	Συναρτήσεις σφάλματος και Γάμμα	
`double erf(double x)`	Συνάρτηση σφάλματος του x.	$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \mathrm{e}^{-t^2}\,\mathrm{d}t$.
`double erfc(double x)`	Συμπληρωματική συνάρτηση σφάλματος του x.	$\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$.
`double tgamma(double x)`	Συνάρτηση $\Gamma(x)$.	$\Gamma(x) = \int_0^\infty t^{x-1}\mathrm{e}^{-t}\,\mathrm{d}t$. Το x θετικό.
`double lgamma(double x)`	Φυσικός λογάριθμος της απόλυτης τιμής του $\Gamma(x)$.	Το x θετικό.
	Διάφορες	
`double copysign(double x, double y)`	Τιμή ίση με |x| και με πρόσημο του y.	
`double fmin(double x, double y)`	Το μικρότερο από τα x,y.	
`double fmax(double x, double y)`	Το μεγαλύτερο από τα x,y.	
`double fdim(double x, double y)`	Το μέγιστο των (x-y), 0.0.	
`double fabs(double x)`	Απόλυτη τιμή του x.	
`double abs(double x)`	Απόλυτη τιμή του x.	
`double fma(double x, double y, double z)`	$x \cdot y + z$	Χωρίς overflow/underflow στις ενδιάμεσες πράξεις. Πιο γρήγορη από την έκφραση $x \cdot y + z$ αν είναι ορισμένη από τον compiler η σταθερά FP_FAST_FMA (ή FP_FAST_FMAF για ορίσματα `float` ή FP_FAST_FMAL για ορίσματα `logn double`).

7.16 Ασκήσεις

1. Να γράψετε συνάρτηση που να δέχεται ως όρισμα έναν πραγματικό αριθμό x και να επιστρέφει την τιμή της ποσότητας $\mathrm{e}^{-x^2/2}$. Να τη χρησιμοποιήσετε στο πρόγραμμά σας για να υπολογίσετε και να τυπώσετε την τιμή της για $x = 0.3,\ 1.4,\ 5.6$.

2. Να γράψετε συνάρτηση που να δέχεται ως όρισμα την ακτίνα ενός κύκλου και να υπολογίζει το εμβαδόν του.

3. Να γράψετε συνάρτηση που να δέχεται ως όρισμα έναν (μικρό) ακέραιο αριθμό και να επιστρέφει το παραγοντικό του. Να την καλέσετε για να υπολογίσετε και να τυπώσετε τα $3!$, $5!$, $7!$.

4. Να γράψετε συνάρτηση που να δέχεται ως ορίσματα τρεις πραγματικούς αριθμούς και να υπολογίζει το άθροισμα των τετραγώνων τους. Να τη χρησιμοποιήσετε για την τριάδα $(3.2, 5.6, 8.1)$.

5. Να γράψετε συνάρτηση που να ελέγχει αν το όρισμά της, ένας ακέραιος αριθμός, είναι πρώτος ή όχι (η ποσότητα που θα επιστρέφει ποιου τύπου είναι;). Να τη χρησιμοποιήσετε για να ελέγξετε τους αριθμούς 89, 261, 1511.

6. Να γράψετε συνάρτηση που να υπολογίζει και να επιστρέφει το μέσο όρο των στοιχείων ενός `std::vector<>` ακεραίων που θα δέχεται ως όρισμα.

7. Να γράψετε συνάρτηση που να υπολογίζει και να επιστρέφει το μικρότερο στοιχείο ενός `std::vector<>` ακεραίων που θα δέχεται ως όρισμα.

8. Να γράψετε συνάρτηση που να υπολογίζει και να επιστρέφει τη *θέση* του μέγιστου στοιχείου ενός `std::vector<>` ακεραίων που θα δέχεται ως όρισμα.

9. Να γράψετε συνάρτηση που να εναλλάσσει τις τιμές δύο πραγματικών μεταβλητών. Κατόπιν, να γράψετε πρόγραμμα το οποίο να χρησιμοποιεί τη συνάρτηση αυτή.

10. Να γράψετε συνάρτηση που να υπολογίζει και να επιστρέφει το εσωτερικό γινόμενο δύο `std::vector<>` πραγματικών αριθμών με ίδιο πλήθος στοιχείων, δηλαδή υπολογίστε το $\sum_i a_i b_i$ αν a, b είναι τα vectors.

11. Να γράψετε συνάρτηση που να επιστρέφει ένα τυχαίο ακέραιο σε διάστημα $[a, b]$ που θα προσδιορίζεται από τα ορίσματά της.

12. Γράψτε συνάρτηση που να μετρά πόσες φορές εμφανίζεται το όρισμά της, ένας ακέραιος αριθμός, στο αρχείο στη διεύθυνση `http://tinyurl.com/114rndint`. Η πρώτη γραμμή του αρχείου περιέχει το πλήθος των αριθμών που ακολουθούν. Εφαρμόστε τη για τους αριθμούς 5744, 6789, 2774.

13. Να γράψετε συνάρτηση που να ελέγχει αν το όρισμά της, ένας θετικός ακέραιος αριθμός, είναι αριθμός Mersenne. Ένας ακέραιος αριθμός k είναι αριθμός Mersenne αν το $k+1$ είναι δύναμη του 2. Βρείτε τους αριθμούς Mersenne μέχρι το 10000.

14. Να γράψετε συνάρτηση που να υπολογίζει το e^x από τον τύπο

$$\mathrm{e}^x \approx x^0/0! + x^1/1! + x^2/2! + \cdots + x^{12}/12! \,.$$

Να βρείτε με αυτόν τον τύπο τις τιμές του e^x για $x = 0.5,\ 1.2,\ 4.1$.

15. Να γράψετε συνάρτηση που να υπολογίζει το ημίτονο από τον τύπο

$$\sin x \approx x^1/1! - x^3/3! + x^5/5! - x^7/7! + x^9/9! - x^{11}/11! \,.$$

Βασιστείτε στο ότι ο κάθε όρος στο άθροισμα προκύπτει από τον προηγούμενό του με πολλαπλασιασμό κατάλληλης ποσότητας. Να χρησιμοποιήσετε τη συνάρτησή σας για να υπολογίσετε το ημίτονο των 35°.

16. Γράψτε συναρτήσεις που να υπολογίζουν και να επιστρέφουν

(α΄) το e^x εφαρμόζοντας τη σχέση

$$\mathrm{e}^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} \,,$$

(β΄) το $\sin x$ εφαρμόζοντας τη σχέση

$$\sin x = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{(2k+1)!} \,,$$

(γ΄) το $\cos x$ εφαρμόζοντας τη σχέση

$$\cos x = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k}}{(2k)!} \,.$$

Για τη διευκόλυνσή σας παρατηρήστε ότι ο κάθε όρος στα αθροίσματα προκύπτει από τον αμέσως προηγούμενο αν αυτός πολλαπλασιαστεί με κατάλληλη ποσότητα.

Στα αθροίσματα να σταματάτε τον υπολογισμό τους όταν ο όρος που πρόκειται να προστεθεί είναι κατ' απόλυτη τιμή μικρότερος από 10^{-10}.

17. Γράψτε συνάρτηση που να δέχεται ως ορίσματα δύο μιγαδικούς αριθμούς και ένα χαρακτήρα. Ο χαρακτήρας θα είναι ένας από τους '+', '-', '*', '/'. Οποιοσδήποτε άλλος δεν είναι αποδεκτός και θα προκαλεί την εκτύπωση ενός μηνύματος που θα ενημερώνει τον χρήστη για το λάθος του και θα διακόπτεται η εκτέλεση της συνάρτησης. Ανάλογα με το χαρακτήρα, η συνάρτηση θα υπολογίζει την αντίστοιχη πράξη μεταξύ των μιγαδικών ορισμάτων και θα επιστρέφει το αποτέλεσμα. Το πρόγραμμά σας να την καλεί και να τυπώνει το αποτέλεσμα του πολλαπλασιασμού των αριθμών $2 + 3\mathrm{i},\ 5.7 - 9\mathrm{i}$.

18. Να γράψετε δύο συναρτήσεις που μετατρέπουν θερμοκρασία από βαθμούς Κελσίου σε βαθμούς Φαρενάιτ και αντίστροφα. Η σχέση των κλιμάκων Κελσίου (C) και Φαρενάιτ (F) είναι γραμμική. Ο τύπος μετατροπής είναι $F = 9/5\,C + 32$.

    Να τις χρησιμοποιήσετε στο πρόγραμμά σας για να υπολογίσετε τη θερμοκρασία σε βαθμούς Φαρενάιτ για:

    - τη θερμοκρασία $22\,^{\circ}\mathrm{C}$,
    - τη θερμοκρασία του απόλυτου 0 ($-273.15\,^{\circ}\mathrm{C}$),
    - τη μέση θερμοκρασία της επιφάνειας του Ήλιου ($6000\,^{\circ}\mathrm{C}$).

    και τη θερμοκρασία σε βαθμούς Κελσίου για τους $100\,^{\circ}\mathrm{F}$.

19. Να γράψετε συνάρτηση που να επιλύει τη δευτεροβάθμια εξίσωση $ax^2 + bx + c = 0$ και να μας επιστρέφει τις λύσεις. Προσέξτε να κάνετε διερεύνηση ανάλογα με τις τιμές των a, b, c, που θα δέχεται ως ορίσματα. Η συνάρτησή σας δε θα τυπώνει τις τιμές των ριζών αλλά θα τις επιστρέφει με ορίσματα.

20. Δύο διδιάστατοι πραγματικοί πίνακες περιέχονται στα αρχεία στις διευθύνσεις `http://tinyurl.com/114matrixA` και `http://tinyurl.com/114matrixB`. Η πρώτη γραμμή σε κάθε αρχείο είναι ο αριθμός των γραμμών και η δεύτερη ο αριθμός των στηλών. Οι επόμενες γραμμές περιέχουν τα στοιχεία των πινάκων κατά γραμμή, από αριστερά προς τα δεξιά, δηλαδή διαδοχικά τα στοιχεία $(0,0)$, $(0,1)$, ..., $(1,0)$, $(1,1)$, ...για κάθε πίνακα.

    - Να γράψετε συνάρτηση που να διαβάζει τα στοιχεία από ένα αρχείο και να τα εκχωρεί σε πίνακα. Ως ορίσματα θα δέχεται τον πίνακα και το όνομα του αρχείου. Χρησιμοποιήστε τη για να δώσετε τιμές σε δύο πίνακες A,B.
    - Να γράψετε άλλη συνάρτηση που να υπολογίζει το γινόμενο των δύο πινάκων.
    - Να γράψετε συνάρτηση που να τυπώνει το όρισμά της, ένα πραγματικό πίνακα, στοιχισμένο κατά στήλες, με 4 δεκαδικά ψηφία σε κάθε στοιχείο.

    Χρησιμοποιώντας τις παραπάνω συναρτήσεις, γράψτε ένα πρόγραμμα που να διαβάζει τους δύο πίνακες και να τυπώνει στην οθόνη το γινόμενό τους.

21. Γράψτε συνάρτηση που να υπολογίζει τον ερμιτιανό συζυγή ενός τετραγωνικού πίνακα μιγαδικών αριθμών. Ο συζυγής να αποθηκεύεται στον αρχικό πίνακα. Εφαρμόστε την για τον πίνακα

$$\begin{pmatrix} 2.3 - \mathrm{i} & 1 - 7\mathrm{i} & 5.8 & -2.9 - 3.7\mathrm{i} \\ -4.9\mathrm{i} & \mathrm{i} & 9 - 0.3\mathrm{i} & -2 + 0.72\mathrm{i} \\ 8.2 + 4\mathrm{i} & -0.8 + \mathrm{i} & 0.2 + 5\mathrm{i} & 9 - 3\mathrm{i} \\ 2.3\mathrm{i} & -7.1 + 9\mathrm{i} & 0.9 & -4\mathrm{i} \end{pmatrix}$$

22. Η απομάκρυνση από τη θέση ισορροπίας μιας μπάλας στην άκρη ενός ελατηρίου περιγράφεται χρονικά από την εξίσωση $x(t) = A\cos(\omega t) + B\sin(\omega t)$, με A =3 cm, B =2 cm, ω =12 Hz.

    (α΄) Να γράψετε συνάρτηση που να δέχεται το χρόνο t και να επιστρέφει την αντίστοιχη απομάκρυνση x.

    (β΄) Να γράψετε πρόγραμμα που να τυπώνει στο αρχείο *data* τις τιμές των t και $x(t)$ με 4 δεκαδικά ψηφία, για $t = 0.0, 0.5, 1.0, ...,100.0$ s. Κάθε ζεύγος να είναι σε ξεχωριστή γραμμή.

23. Να γράψετε συνάρτηση που να υπολογίζει το παραγοντικό ενός ακέραιου αριθμού βασιζόμενοι στο ότι

$$n! = \begin{cases} (n-1)! \times n\ , & n > 0\ , \\ 1\ , & n = 0\ . \end{cases}$$

24. Να γράψετε αναδρομική συνάρτηση που να ελέγχει αν το όρισμά της, μια ακέραιη ποσότητα, είναι δύναμη του 2. Να επιστρέφει τιμή λογικού τύπου. Να τη χρησιμοποιήσετε για να ελέγξετε αν οι αριθμοί 4096, 65534, 1855932 είναι δυνάμεις του 2.

25. (α΄) Γράψτε συνάρτηση που θα δέχεται ως πρώτο όρισμα ένα ακέραιο αριθμό, θα τον αναλύει στα ψηφία του και θα τα αποθηκεύει στο δεύτερο όρισμά του, ένα διάνυσμα τουλάχιστον 10 θέσεων.

    (β΄) Ένας θετικός ακέραιος αριθμός χαρακτηρίζεται ως παλίνδρομος αν «διαβάζεται» το ίδιο από αριστερά προς τα δεξιά και αντίστροφα. Ο παλίνδρομος αριθμός δηλαδή έχει ίδια το πρώτο (μη μηδενικό) και το τελευταίο ψηφίο, το δεύτερο και το προτελευταίο κλπ. Π.χ. οι ακέραιοι 19791, 4774 είναι παλίνδρομοι.

    Γράψτε συνάρτηση που θα δέχεται ένα ακέραιο αριθμό και θα ελέγχει αν είναι παλίνδρομος.

    (γ΄) Γράψτε στο αρχείο με όνομα *palindrome.dat* όλους τους παλίνδρομους ακέραιους που είναι γινόμενο δύο τριψήφιων αριθμών. Τον μεγαλύτερο από αυτούς γράψτε τον στην οθόνη μαζί με τους τριψήφιους αριθμούς που τον παρήγαγαν.

26. Γράψτε κώδικα που να τυπώνει στο αρχείο *palindrome.dat* όλους τους παλίνδρομους αριθμούς (δείτε τον ορισμό τους στην άσκηση 25) μέχρι το 1000000. Στην οθόνη να τυπώνει το πλήθος τους.

27. Γράψτε μία συνάρτηση με όνομα digit που να δέχεται δύο ακέραια ορίσματα, N και d. Η συνάρτηση θα επιστρέφει το ψηφίο στη θέση d του αριθμού N. Προσέξτε ότι το N μπορεί να είναι αρνητικός. Θεωρούμε ότι στην πρώτη θέση είναι το ψηφίο των μονάδων. Για παράδειγμα, η κλήση της digit με N=57960,

d=2 πρέπει να επιστρέφει τον αριθμό 6. Αν το d είναι μεγαλύτερο από το πλήθος των ψηφίων του N, η συνάρτηση θα επιστρέφει 0.

Αποθηκεύστε στον υπολογιστή σας το αρχείο στη διεύθυνση `http://tinyurl.com/ints201411`. Περιέχει 3590 ακέραιους, σε ξεχωριστή γραμμή ο καθένας. Χρησιμοποιήστε τη συνάρτηση που γράψατε για να βρείτε το τρίτο ψηφίο των αριθμών του αρχείου. Τα ψηφία που θα βρείτε, να τα γράψετε στο αρχείο *digit.txt*, ένα σε κάθε σειρά.

28. Γράψτε συνάρτηση που να δέχεται ένα διάνυσμα με στοιχεία οποιουδήποτε τύπου και να εντοπίζει και να επιστρέφει το στοιχείο που εμφανίζεται τις περισσότερες φορές συνεχόμενα (ή το τελευταίο από όσα εμφανίζονται με ίδιο πλήθος). Ελέγξτε την για τη σειρά στοιχείων $\{2, 8, 8, 3, 5, 5, 5, 8, 8, 1, 6, 7, 7, 7\}$· θα πρέπει να βρει το 7.

29. Η στροφή ενός τριδιάστατου διανύσματος $\vec{r} = (x, y, z)$ κατά γωνία θ γύρω από τον άξονα $\hat{x}$, μπορεί να αναπαρασταθεί με τον πολλαπλασιασμό του διανύσματος $\vec{r}$ με τον πίνακα

$$R_x(\theta) = \begin{vmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{vmatrix},$$

δηλαδή, το στραμμένο διάνυσμα έχει συνιστώσες

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{vmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{vmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}.$$

Οι πίνακες στροφής γύρω από τους άξονες $\hat{y}$, $\hat{z}$ είναι αντίστοιχα οι

$$R_y(\theta) = \begin{vmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{vmatrix}$$

και

$$R_z(\theta) = \begin{vmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix}.$$

(α′) Να γράψετε τρεις συναρτήσεις· η κάθε μια από αυτές θα εκτελεί τη στροφή γύρω από έναν άξονα. Κάθε συνάρτηση θα δέχεται ως ορίσματα

- τη γωνία στροφής θ και
- ένα διάνυσμα, οι συνιστώσες του οποίου θα τροποποιούνται.

(β′) Να γράψετε συναρτήσεις που θα υπολογίζουν το μέτρο ενός διανύσματος και τη γωνία μεταξύ δύο διανυσμάτων.

(γ′) Να γράψετε πρόγραμμα που θα χρησιμοποιεί τα παραπάνω για να κάνετε τα εξής:

i. Δημιουργήστε ένα διάνυσμα με συνιστώσες $x = 0.5$, $y = -0.3$, $z = 1.2$. Να το στρέψετε διαδοχικά κατά γωνία 30° γύρω από τον άξονα $\hat{y}$, κατόπιν κατά γωνία 35° γύρω από τον άξονα $\hat{x}$ και τέλος κατά γωνία 58° γύρω από τον άξονα $\hat{z}$. Τυπώστε στην οθόνη τις τελικές συνιστώσες.

ii. Υπολογίστε και τυπώστε στην οθόνη τα μέτρα του αρχικού και του τελικού (μετά τις στροφές) διανύσματος καθώς και τη μεταξύ τους γωνία.

30. Στη Μαθηματική Φυσική χρησιμοποιείται η οικογένεια πολυωνύμων Hermite, $H_n(x)$. Ο βαθμός n του πολυωνύμου είναι ακέραιος, 0, 1,.... Τα πρώτα πολυώνυμα Hermite είναι

$$\begin{aligned} H_0(x) &= 1 \\ H_1(x) &= 2x \\ H_2(x) &= 4x^2 - 2 \\ \vdots &= \vdots \end{aligned}$$

Για τα πολυώνυμα Hermite ισχύει η αναδρομική σχέση

$$H_n(x) = 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x)\ , \qquad n \geq 2\ .$$

Γράψτε συνάρτηση που να υπολογίζει την τιμή ενός πολυωνύμου Hermite. Αυτή θα δέχεται ως ορίσματα έναν ακέραιο αριθμό n που θα αντιπροσωπεύει το βαθμό του πολυωνύμου και ένα πραγματικό x που θα είναι το σημείο υπολογισμού. Θα επιστρέφει την τιμή του $H_n(x)$.

31. Η κβαντομηχανική αντιμετώπιση του μονοδιάστατου αρμονικού ταλαντωτή (μάζα m σε δυναμικό $V = kx^2/2$) καταλήγει στις ιδιοσυναρτήσεις με χωρικό τμήμα

$$\psi_n(y) = \sqrt{\frac{1}{2^n n! \sqrt{\pi}}} \left(\frac{\sqrt{km}}{\hbar}\right)^{1/4} H_n(y) \mathrm{e}^{-y^2/2}\ , \tag{7.1}$$

όπου $y = x\sqrt{\sqrt{km}/\hbar}$.

Χρησιμοποιήστε τη συνάρτηση που γράψατε για τα πολυώνυμα Hermite στην άσκηση 30 για να υπολογίσετε την πυκνότητα πιθανότητας ($\psi\psi^*$) της κυματοσυνάρτησης (7.1). Θα γράψετε μια νέα συνάρτηση γι᾽ αυτή, που θα δέχεται ως ορίσματα τα n,x. Θεωρήστε ότι $m = k = \hbar = 1$.

Να τυπώσετε στο αρχείο *harmonic.dat* τις τιμές της πυκνότητας πιθανότητας για $n = 5$ σε 60 ισαπέχοντα σημεία x στο διάστημα $[-6, 6]$, μαζί με τα αντίστοιχα σημεία x (δηλαδή το αρχείο θα περιέχει δύο στήλες, x και $\psi\psi^*$).

32. Στη Μαθηματική Φυσική χρησιμοποιείται η οικογένεια πολυωνύμων Legendre, $P_\ell(x)$, με $x \in [-1, 1]$. Ο βαθμός ℓ του πολυωνύμου είναι ακέραιος, 0, 1,.... Τα δύο πρώτα πολυώνυμα Legendre είναι $P_0(x) = 1$ και $P_1(x) = x$, ενώ για μεγαλύτερες τιμές του ℓ υπολογίζονται από την αναδρομική σχέση:

$$\ell P_\ell(x) = (2\ell - 1)xP_{\ell-1}(x) - (\ell - 1)P_{\ell-2}(x) \, .$$

Γράψτε συνάρτηση που να υπολογίζει την τιμή ενός πολυωνύμου Legendre. Αυτή θα δέχεται ως ορίσματα έναν ακέραιο αριθμό ℓ που θα αντιπροσωπεύει το βαθμό του πολυωνύμου και ένα πραγματικό x που θα είναι το σημείο υπολογισμού. Θα επιστρέφει την τιμή του $P_\ell(x)$.

33. Γράψτε συνάρτηση που να ελέγχει αν το όρισμά της, ένας ακέραιος αριθμός, είναι πρώτος ή όχι.

Το αρχείο στη διεύθυνση http://tinyurl.com/114rndint περιέχει ακέραιους αριθμούς, σε ξεχωριστή σειρά ο καθένας. Η πρώτη γραμμή περιέχει το πλήθος των αριθμών που ακολουθούν. Βρείτε πόσοι από αυτούς δεν είναι πρώτοι αριθμοί και τυπώστε στην οθόνη το πλήθος τους.

34. Επιλέξτε N τυχαία σημεία σε ένα τετράγωνο $-1 \leq x < 1$, $-1 \leq y < 1$. Βρείτε τον λόγο όσων βρίσκονται εντός ενός κύκλου με ακτίνα 1 ($x^2 + y^2 = 1$) προς τα συνολικά. Να επαναλάβετε τον κώδικα για διάφορες μεγάλες τιμές του N (επομένως, γράψτε τον σε συνάρτηση). Δοκιμάστε π.χ. τα $N = 10^3$, $N = 10^4$, ..., $N = 10^9$. Επαληθεύστε ότι ο λόγος αυτός για πολύ μεγάλα N προσεγγίζει το λόγο του εμβαδού του κύκλου με διάμετρο 2 προς το εμβαδόν του τετραγώνου με πλευρά 2 (πόσος είναι;)

35. Γράψτε δύο συναρτήσεις, η μία να δέχεται καρτεσιανές συντεταγμένες (x, y) στο επίπεδο και να υπολογίζει τις αντίστοιχες πολικές (ρ, θ) και η άλλη να κάνει την αντίστροφη μετατροπή.

Γράψτε πρόγραμμα που να τυπώνει στο αρχείο με όνομα *plotdata* τα σημεία x, y που αντιστοιχούν στις πολικές συντεταγμένες ρ, θ για θ =0°,2°,4°,...,360° και $\rho = \mathrm{e}^{\sin\theta} 2\cos(4\theta) + \sin^5((2\theta - \pi)/24)$. Τη συνάρτηση $\rho(\theta)$ γράψτε τη σε ξεχωριστή συνάρτηση. Η καμπύλη που σχηματίζεται είναι η «καμπύλη πεταλούδας».

36. Γράψτε συνάρτηση που να υπολογίζει όλους τους πρώτους αριθμούς μέχρι έναν ακέραιο N εφαρμόζοντας το «κόσκινο του Ερατοσθένη». Να τη χρησιμοποιήσετε για να βρείτε και να τυπώσετε στην οθόνη τους πρώτους αριθμούς μέχρι το 1000.

37. Υλοποιήστε τη γεννήτρια ψευδοτυχαίων αριθμών του Cliff Pickover[7]:

$$x_{i+1} = |(100 \ln(x_i)) \bmod 1| \, ,$$

[7] http://mathworld.wolfram.com/CliffRandomNumberGenerator.html

με $x_0 = 0.1$. Η έκφραση $a \bmod 1$ σημαίνει το δεκαδικό μέρος του a. Οι πραγματικοί αριθμοί x_i προκύπτουν τυχαίοι στο $[0, 1)$. Προσέξτε να γράψετε έτσι τη συνάρτηση ώστε να καλείται χωρίς ορίσματα[8].

38. Γράψτε αναδρομική συνάρτηση που να υπολογίζει την ορίζουσα ενός τετραγωνικού πίνακα A διάστασης $N \times N$ εφαρμόζοντας τον ακόλουθο τύπο[9]

$$\det A = \sum_{i=1}^{N} (-1)^{i+j} a_{ij} \det \widetilde{A}_{ij} \ ,$$

για σταθερό j, π.χ. 1. Το στοιχείο του A στην i γραμμή και j στήλη συμβολίζεται με a_{ij}, ενώ $\widetilde{A}_{ij}$ είναι ο πίνακας που προκύπτει από τον A με διαγραφή της i γραμμής και της j στήλης. Μπορείτε να γράψετε τη συνάρτηση ώστε να δέχεται x οποιουδήποτε τύπου[10];

39. Γράψτε πρόγραμμα που να χρησιμοποιεί τη συνάρτηση για την ορίζουσα που γράψατε στην άσκηση 38 ώστε να προσδιορίζει τη λύση γραμμικού συστήματος εφαρμόζοντας τη μέθοδο του Cramer[11].

40. Η κβαντομηχανική αντιμετώπιση του ατόμου του Υδρογόνου καταλήγει στις ιδιοσυναρτήσεις (σε σφαιρικές συντεταγμένες)

$$\psi_{n\ell m}(r, \theta, \phi) = R_{n\ell}(r) Y_{\ell m}(\theta, \phi) \ .$$

Το γωνιακό τμήμα τους είναι οι *σφαιρικές αρμονικές*,

$$Y_{\ell m}(\theta, \phi) = \sqrt{\frac{2\ell + 1}{4\pi} \frac{(\ell - m)!}{(\ell + m)!}} \, P_\ell^m(\cos\theta) \mathrm{e}^{\mathrm{i}m\phi} \ .$$

Τα *συναφή πολυώνυμα Legendre*, $P_\ell^m(x)$, ικανοποιούν τις σχέσεις

- αν $\ell = m$

$$P_\ell^m(x) = (-1)^m \, 1 \times 3 \times 5 \times \cdots \times (2m - 1) \, (1 - x^2)^{m/2} \ ,$$

- αν $\ell = m + 1$

$$P_\ell^m(x) = x(2m + 1) P_m^m(x) \ ,$$

- ενώ σε άλλη περίπτωση δίνονται από την αναδρομική σχέση

$$(\ell - m) P_\ell^m(x) = x(2\ell - 1) P_{\ell-1}^m(x) - (l + m - 1) P_{\ell-2}^m(x) \ .$$

Οι γωνίες θ και ϕ μεταβάλλονται στα διαστήματα $[0, \pi]$ και $[0, 2\pi)$ αντίστοιχα.

[8] Δείτε την §7.14.

[9] http://mathworld.wolfram.com/DeterminantExpansionbyMinors.html

[10] αρκεί να ορίζονται οι πράξεις πρόσθεσης και πολλαπλασιασμού.

[11] http://mathworld.wolfram.com/CramersRule.html

- Γράψτε συνάρτηση που να υπολογίζει το παραγοντικό ενός μικρού ακεραίου.
- Γράψτε συνάρτηση που να υπολογίζει το συναφές πολυώνυμο Legendre, $P_\ell^m(x)$.
- Γράψτε συνάρτηση που να υπολογίζει τη σφαιρική αρμονική, $Y_{\ell m}(\theta, \phi)$.
- Δημιουργήστε ένα καρτεσιανό πλέγμα 50×100 σημείων στο επίπεδο $\theta - \phi$ και υπολογίστε σε καθένα από αυτά τις τιμές των $Y_{\ell m}(\theta, \phi)$. Τυπώστε στο αρχείο με όνομα *ylm_data* τις τιμές των εκφράσεων $\sin\theta\cos\phi$, $\sin\theta\sin\phi$, $\cos\theta$, $Y_{\ell m}(\theta,\phi)Y^*_{\ell m}(\theta,\phi)$ (δηλαδή, ουσιαστικά τα x, y, z, $\psi\psi^*$) για κάθε σημείο, με $\ell = 2$, $m = 0$ (δηλαδή, ένα από τα d-τροχιακά).

41. Ένας αλγόριθμος για να βρούμε τη ρίζα μίας συνάρτησης $f(x)$, δηλαδή, την πραγματική ή μιγαδική τιμή $\bar{x}$ στην οποία η $f(x)$ μηδενίζεται ($f(\bar{x}) = 0$), είναι ο αλγόριθμος Müller. Σύμφωνα με αυτόν

(α΄) επιλέγουμε τρεις διαφορετικές τιμές x_0, x_1, x_2 στην περιοχή της αναζητούμενης ρίζας.

(β΄) Ορίζουμε τις ποσότητες

$$\begin{aligned} q &= \frac{x_2 - x_1}{x_1 - x_0}\,, \\ A &= q\left(f(x_2) - f(x_1)\right) - q^2\left(f(x_1) - f(x_0)\right)\,, \\ B &= (q+1)\left(f(x_2) - f(x_1)\right) + A\,, \\ C &= (q+1) f(x_2)\,. \end{aligned}$$

(γ΄) Η επόμενη προσέγγιση της ρίζας δίνεται από τη σχέση

$$x_3 = x_2 - \frac{2C(x_2 - x_1)}{D}\,,$$

όπου D ο αριθμός που έχει το μεγαλύτερο *μέτρο* μεταξύ των *μιγαδικών* $B + \sqrt{B^2 - 4AC}$, $B - \sqrt{B^2 - 4AC}$.

(δ΄) Αν η νέα προσέγγιση είναι ικανοποιητική πηγαίνουμε στο βήμα 41στ΄.

(ε΄) Θέτουμε $x_0 \leftarrow x_1$, $x_1 \leftarrow x_2$, $x_2 \leftarrow x_3$. Επαναλαμβάνουμε τη διαδικασία από το βήμα 41β΄.

(στ΄) Τέλος.

Προσέξτε ότι οι διαδοχικές προσεγγίσεις της ρίζας μπορεί να είναι *μιγαδικές* λόγω της τετραγωνικής ρίζας, οπότε οι ποσότητες x_n, q, A, B, C, D είναι γενικά *μιγαδικές*.

Βρείτε μία ρίζα της συνάρτησης $f(x) = x^3 - x + 1$ χρησιμοποιώντας τον αλγόριθμο Müller.

42. Η συνάρτηση Bessel πρώτου είδους, ακέραιας τάξης n, $J_n(x)$, μπορεί να οριστεί ως εξής

$$J_n(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m!(m+n)!} \left(\frac{x}{2}\right)^{2m+n} .$$

Να τυπώσετε στο αρχείο *bessel.dat* τις τιμές των συναρτήσεων $J_0(x)$, $J_1(x)$, $J_2(x)$ σε 150 ισαπέχοντα σημεία x_i στο διάστημα $[0, 20]$. Το αρχείο θα έχει σε κάθε γραμμή τις τιμές

$$x_i \qquad J_0(x_i) \qquad J_1(x_i) \qquad J_2(x_i)$$

Υπόδειξη Ι: Στο άθροισμα δεν μπορούμε, φυσικά, να πάρουμε άπειρους όρους. Να σταματήσετε τον υπολογισμό του στον πρώτο όρο που κατ' απόλυτη τιμή είναι μικρότερος από 10^{-12}.

Υπόδειξη ΙΙ: Παρατηρήστε ότι ο κάθε όρος στο άθροισμα προκύπτει από τον προηγούμενό του με πολλαπλασιασμό κατάλληλης ποσότητας. Μπορεί να σας βοηθήσει.

43. Η κυβική ρίζα ενός πραγματικού αριθμού a μπορεί να υπολογιστεί προσεγγιστικά ως εξής: Επιλέγουμε μία οποιαδήποτε μη μηδενική τιμή, x_0. Έστω $x_0 = 1$. Εφαρμόζουμε τον τύπο

$$x_{i+1} = x_i \frac{x_i^3 + 2a}{2x_i^3 + a}$$

για να παράγαγουμε διαδοχικά τις τιμές $x_1, x_2, \ldots$. Δηλαδή,

$$\begin{aligned} x_1 &= x_0 \frac{x_0^3 + 2a}{2x_0^3 + a} , \\ x_2 &= x_1 \frac{x_1^3 + 2a}{2x_1^3 + a} , \quad \text{κλπ.} \end{aligned}$$

Κάθε τιμή από τις $x_1, x_2, \ldots$ προσεγγίζει όλο και καλύτερα το $\sqrt[3]{a}$. Μπορούμε να σταματήσουμε σε κάποια τιμή x_k που ικανοποιεί τη σχέση $|x_k^3 - a| \leq \varepsilon$, όπου ε μία αρκετά μικρή θετική τιμή, π.χ. 10^{-12}.

Γράψτε συνάρτηση που να δέχεται ως όρισμα ένα πραγματικό αριθμό και να επιστρέφει την προσεγγιστική τιμή για την κυβική ρίζα του. Χρησιμοποιήστε τη για να υπολογίσετε τις κυβικές ρίζες των αριθμών 20.0, 20.1, 20.2,...,30.0. Να τυπώσετε σε αρχείο με όνομα *cbrt* δύο στήλες αριθμών: η πρώτη θα αποτελείται από τους αριθμούς 20.0, 20.1, 20.2,...,30.0 και η δεύτερη από τις κυβικές ρίζες τους. Να κρατήσετε 12 δεκαδικά ψηφία στις ρίζες.

44. Γράψτε ένα πρόγραμμα που να παίζει τρίλιζα με αντίπαλο το χρήστη. Σε αυτό το παιχνίδι, οι δύο παίκτες τοποθετούν διαδοχικά σε θέσεις πλέγματος 3×3 ή, γενικότερα, $N \times N$, το σύμβολό τους (π.χ. 'Χ' ή 'Ο') με σκοπό να

επιτύχουν το σχηματισμό τριάδας (ή, γενικότερα, N-άδας) ίδιων συμβόλων σε οριζόντια, κάθετη, ή διαγώνια γραμμή. Στην περίπτωση που δε σχηματιστεί τέτοια γραμμή, υπάρχει ισοπαλία.

Φροντίστε στον κώδικά σας να υπάρχει δυνατότητα επιλογής του ποιος παίζει πρώτος. Το πρόγραμμα θα πρέπει να δίνει επαρκείς οδηγίες στο χρήστη για το πώς επιλέγει θέση πλέγματος. Προφανώς, πρέπει ο υπολογιστής να επιδιώκει τη νίκη, καταρχάς, και, όσο είναι δυνατό, να αποφεύγει την ήττα. Το πρόγραμμα να τυπώνει σε στοιχειώδη μορφή το πλέγμα μετά από κάθε κίνηση· ας εμφανίζεται κάτι σαν

```
X|O|X
-----
 |  |O
-----
O|X|
```

Φροντίστε, επιπλέον, να περιγράφετε επαρκώς με σχόλια (τι κάνουν) τις ομάδες εντολών που χρησιμοποιείτε.

45. Γράψτε ένα πρόγραμμα που να παίζει four-in-a-row με αντίπαλο εσάς. Σε αυτό το παιχνίδι, δύο παίκτες τοποθετούν διαδοχικά τις «μάρκες» τους σε ένα κατακόρυφο πλέγμα $M \times N$ (εφαρμόστε το για 7 στήλες επί 6 γραμμές). Κάθε μάρκα τοποθετείται στην κορυφή μίας στήλης και πέφτει έως ότου συναντήσει άλλη μάρκα ή το άκρο του πλέγματος. Νικητής είναι ο παίκτης που σχηματίζει τέσσερις συνεχόμενες μάρκες οριζοντίως, καθέτως ή διαγωνίως. Εάν το πλέγμα γεμίσει χωρίς να έχει σχηματιστεί τέτοια γραμμή, έχουμε ισοπαλία.

    Να φροντίσετε ο υπολογιστής να μην επιλέγει τυχαία τη στήλη στην οποία θα ρίξει τη «μάρκα» του. Θα πρέπει προφανώς να την επιλέγει ώστε να προσπαθεί να σχηματίσει τετράδα. Αν δεν γίνεται αυτό, θα πρέπει να εμποδίζει τον αντίπαλο να σχηματίσει τετράδα (μόλις έρθει η σειρά του). Αλλιώς, μπορεί να επιλέγει μία τυχαία στήλη.

46. **Το πρόβλημα των N βασιλισσών**. Σε μία σκακιέρα $N \times N$, με $N > 3$, θέλουμε να τοποθετήσουμε N βασίλισσες σε τέτοιες θέσεις ώστε να μη βρίσκονται ανά δύο στην ίδια γραμμή, στήλη ή διαγώνιο. Γράψτε πρόγραμμα που να υπολογίζει και να τυπώνει στην οθόνη μια τέτοια τοποθέτηση.

    Κάθε γραμμή της σκακιέρας θα έχει προφανώς μία μόνο βασίλισσα. Το πρόγραμμά σας θα είναι πιο απλό αν «γεμίζετε» διαδοχικά τις γραμμές επιλέγοντας μόνο τη στήλη στην οποία θα τοποθετηθεί το κομμάτι.

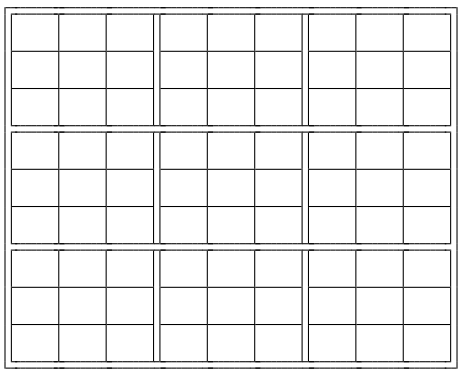
    Ακολουθήστε τον εξής αλγόριθμο:

    - Δημιουργήστε ένα πίνακα ακεραίων με διαστάσεις $N \times N$. Έστω ότι ονομάζεται board. Τα στοιχεία με τιμή 0 θα αντιπροσωπεύουν επιτρεπτές θέσεις.

- Δημιουργήστε ένα διάνυσμα ακεραίων με N στοιχεία και όνομα π.χ. column. Θα αποθηκεύει τις στήλες των βασιλισσών. Η γραμμή i θα έχει βασίλισσα στη θέση `column[i]`.
- Γράψτε μια συνάρτηση που θα δέχεται συγκεκριμένη γραμμή και στήλη, θα εντοπίζει τις «απαγορευμένες» θέσεις (γραμμή, στήλη, διαγώνιους) και θα αυξάνει την τιμή των αντίστοιχων στοιχείων του board. Έτσι, αν κάποιο στοιχείο είναι επιτρεπτό (έχει τιμή 0) θα γίνεται μη επιτρεπτό (με τιμή 1). Αν είναι ήδη απαγορευμένο θα γίνεται πιο «έντονη» η απαγόρευση.
- Γράψτε μια συνάρτηση που θα δέχεται συγκεκριμένη γραμμή και στήλη και θα «ακυρώνει» τη διαδικασία που έκανε η προηγούμενη. Αν κάποιο στοιχείο είναι απαγορευμένο με τιμή 1 θα γίνεται επιτρεπτό, αν είναι απαγορευμένο με μεγαλύτερη τιμή θα γίνεται λιγότερο απαγορευμένο.

  Οι δύο συναρτήσεις μπορούν εύκολα να συγχωνευθούν σε μία.
- Ξεκινήστε από την πρώτη γραμμή. Αν υπάρχουν διαθέσιμες θέσεις σε αυτή, επιλέξτε μία, κάντε κατάλληλες τροποποιήσεις στο διάνυσμα και στον πίνακα και συνεχίστε στην επόμενη γραμμή. Αν δεν υπάρχουν διαθέσιμες θέσεις σημαίνει ότι κάποια προηγούμενη επιλογή κενής στήλης δεν οδηγεί σε λύση. Αναιρέστε την τυχόν αποθηκευμένη στήλη για την τρέχουσα γραμμή, πηγαίνετε στην προηγούμενη, ακυρώστε τις αλλαγές που έγιναν στην προηγούμενη επιλογή στήλης. Επιλέξτε άλλη στήλη. Αν εξαντληθούν οι επιτρεπτές στήλες σε μία γραμμή, μετακινηθείτε στην προηγούμενή της και ακολουθήστε την ίδια διαδικασία.
- Όταν υπολογιστούν οι θέσεις όλων των βασιλισσών, τυπώστε στην οθόνη τη σκακιέρα (N σύμβολα σε N γραμμές, όπου υπάρχει βασίλισσα να εμφανίζεται ο χαρακτήρας 'Q' αλλιώς να εμφανίζεται ο χαρακτήρας '_').

47. **Sudoku**. Γράψτε ένα πρόγραμμα που να λύνει sudoku. Σε αυτή τη δραστηριότητα ο σκοπός είναι να γεμίσει το παρακάτω πλέγμα 9×9 με αριθμητικά ψηφία ώστε κάθε γραμμή, στήλη ή κουτί 3×3 να περιέχει όλα τα ψηφία $1-9$, από μία φορά το καθένα (χωρίς επανάληψη).



Το πρόγραμμα θα δέχεται ένα μερικώς συμπληρωμένο πλέγμα, θα προσδιορίζει τα ψηφία στα κενά τετράγωνα και θα το τυπώνει συμπληρωμένο.

Ο αλγόριθμος που μπορείτε να ακολουθήσετε είναι ο εξής:

(α΄) Ξεκινάμε από το πρώτο κενό τετράγωνο και τοποθετούμε εκεί το ψηφίο 1.

(β΄) Ελέγχουμε αν είναι αποδεκτό σύμφωνα με τους κανόνες που αναφέρθηκαν. Αν όχι, το αντικαθιστούμε με το 2, 3, κλπ. έως ότου βρούμε αποδεκτό ψηφίο. Αν εξαντλήσουμε τα ψηφία χωρίς να αποδεχθούμε κανένα, το πλέγμα δεν έχει λύση.

(γ΄) Προχωράμε στο επόμενο κενό τετράγωνο και ακολουθούμε την ίδια διαδικασία. Στην περίπτωση που εξαντλήσουμε τα ψηφία 1–9, το αφήνουμε κενό το συγκεκριμένο και μετακινούμαστε στο προηγούμενο τετράγωνο που έχουμε συμπληρώσει. Αυξάνουμε τον αριθμό του διαδοχικά, ελέγχοντας κάθε φορά τις συνθήκες. Αν αποδεχθούμε ψηφίο, προχωράμε στο επόμενο τετράγωνο, αν τα εξαντλήσουμε, μετακινούμαστε πιο πίσω κ.ο.κ.

Δοκιμάστε το για το πλέγμα

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

48. Ένας τρόπος να σχεδιάσουμε ένα διδιάστατο fractal είναι ο εξής: ξεκινάμε από ένα σημείο του επιπέδου, έστω το $(x = 0,\ y = 0)$, και το μετακινούμε στη θέση (x', y') όπου

$$\begin{aligned} x' &= a \cdot x + b \cdot y + e \\ y' &= c \cdot x + d \cdot y + f \end{aligned}$$

και a,b,c,d,e,f σταθερές.

Το νέο σημείο το μεταφέρουμε με τον ίδιο μετασχηματισμό στο επόμενο σημείο του fractal (δηλαδή, θέτουμε $x' \to x$ και $y' \to y$ και παράγουμε το νέο (x', y')). Τη διαδικασία αυτή την επαναλαμβάνουμε επ᾽ άπειρο. Η ακολουθία των σημείων (x, y) που παράγονται, αποτελεί το fractal.

Γράψτε ένα πρόγραμμα το οποίο:

(α΄) θα διαβάζει από το αρχείο *in.dat* 4 γραμμές. Σε κάθε γραμμή θα υπάρχουν 7 πραγματικοί αριθμοί: οι 6 πρώτοι αντιστοιχούν στους συντελεστές a, b, c, d, e, f και ο τελευταίος στην πιθανότητα p να γίνει ο συγκεκριμένος μετασχηματισμός. Κάθε γραμμή αντιστοιχεί σε άλλο μετασχηματισμό. Το άθροισμα των πιθανοτήτων, $\sum p_i$, όλων των μετασχηματισμών είναι 1.

(β΄) Θα επιλέγει ένα τυχαίο πραγματικό αριθμό r στο διάστημα $[0, 1)$. Ανάλογα με την τιμή του θα εφαρμόζεται διαφορετικός μετασχηματισμός. Δηλαδή, αν ισχύει $0 \leq r < p_1$ θα εκτελείται ο πρώτος μετασχηματισμός, αν ισχύει $p_1 \leq r < p_1 + p_2$ θα εκτελείται ο δεύτερος κλπ.

(γ΄) θα επαναλαμβάνει το προηγούμενο βήμα 1000 φορές σώζοντας κάθε φορά το σημείο που προκύπτει στο αρχείο *fractal.dat*.

Δοκιμάστε το πρόγραμμά σας με τις εξής παραμέτρους στο *in.dat*

$$\begin{array}{rrrrrrr} 0 & 0 & 0 & 0.16 & 0 & 0 & 0.01 \\ 0.85 & 0.04 & -0.04 & 0.85 & 0 & 1.6 & 0.85 \\ 0.2 & -0.26 & 0.23 & 0.22 & 0 & 1.6 & 0.07 \\ -0.15 & 0.28 & 0.26 & 0.24 & 0 & 0.44 & 0.07 \end{array}$$

και

$$\begin{array}{rrrrrrr} 0 & 0 & 0 & 0.25 & 0 & -0.4 & 0.02 \\ 0.95 & 0.005 & -0.005 & 0.93 & -0.002 & 0.5 & 0.84 \\ 0.035 & -0.2 & 0.16 & 0.04 & -0.09 & 0.02 & 0.07 \\ -0.04 & 0.2 & 0.16 & 0.04 & 0.083 & 0.12 & 0.07 \end{array}$$

Αν θέλετε, μπορείτε να σχεδιάσετε τα *fractal.dat* που προκύπτουν.

49. Γράψτε ένα πρόγραμμα που να παρέχει την υποδομή για να παίξουν «Ναυμαχία» δύο παίκτες. Ο ένας μπορεί να είναι ο ίδιος ο υπολογιστής. Σε αυτό το παιχνίδι κάθε παίκτης έχει ένα διδιάστατο πλέγμα 10×10 στο οποίο τοποθετεί τα πλοία του και ένα όμοιο πλέγμα για τις βολές του εναντίον του αντίπαλου παίκτη. Ο κάθε παίκτης τοποθετεί στο πλέγμα του, είτε οριζόντια είτε κάθετα, τα εξής πλοία:

(α΄) 1 Μεταγωγικό (5 θέσεις),

(β΄) 1 Θωρηκτό (4 θέσεις),

(γ΄) 1 Αντιτορπιλικό (3 θέσεις),

(δ΄) 1 Υποβρύχιο (3 θέσεις),

(ε΄) 1 Ναρκαλιευτικό (2 θέσεις).

Τα πλοία προφανώς δεν μπορούν να επικαλύπτονται στο πλέγμα και οι θέσεις τους δεν είναι γνωστές στον αντίπαλο.

Κάθε παίκτης, διαδοχικά, επιλέγει μια θέση στο πλέγμα του αντιπάλου. Αν χτυπήσει πλοίο, θα ενημερωθεί από τον αντίπαλο. Ένα πλοίο βυθίζεται όταν

χτυπηθεί σε όλες τις θέσεις που καταλαμβάνει. Σκοπός κάθε παίκτη είναι να βυθίσει όλα τα πλοία του αντιπάλου. Νικητής είναι αυτός που θα το επιτύχει.

Το πρόγραμμά σας να τυπώνει στην οθόνη τα δύο πλέγματα (πλοίων και βολών) του παίκτη που είναι η σειρά του να παίξει. Οι βολές να σημειώνονται με X αν είναι επιτυχείς και με O αν δεν έχουν βρει το στόχο. Ο υπολογιστής θα ζητά από τον παίκτη να προσδιορίσει τη βολή του. Αν ο παίκτης είναι ο ίδιος ο υπολογιστής δεν θα τυπώνετε το πλέγμα του στην οθόνη αλλά θα γίνεται η επιλογή στόχου και θα έρχεται η σειρά σας.

50. Έστω η μιγαδική συνάρτηση μιγαδικής μεταβλητής $p(z)$. Μια οποιαδήποτε αρχική τιμή z_0 (για την οποία ισχύει $p'(z_0) \neq 0$) θα συγκλίνει σε μία από τις ρίζες της $p(z)$, στα σημεία δηλαδή που μηδενίζεται η $p(z)$, αν την μεταβάλλουμε ως εξής:

$$z_{i+1} = z_i - \frac{p(z_i)}{p'(z_i)}\,, \quad i = 0, 1, 2, \ldots\ .$$

Δηλαδή, αν ξεκινήσουμε από μία τιμή z_0, η εφαρμογή του τύπου θα μας δώσει τη z_1. Με νέα εφαρμογή του τύπου θα υπολογίσουμε τη z_2, κλπ., έως ότου πλησιάσουμε όσο κοντά θέλουμε σε μία από τις ρίζες της $p(z)$, όταν δηλαδή $|p(z_i)| \leq \epsilon$ με ϵ ένα πολύ μικρό θετικό αριθμό. Αυτή η επαναληπτική διαδικασία αποτελεί τον *αλγόριθμο Newton–Raphson* για εύρεση ρίζας, εφαρμοσμένο σε μιγαδικές συναρτήσεις.

Η μιγαδική συνάρτηση μιγαδικής μεταβλητής $p(z) = z^3 - 1$ έχει ρίζες τα $a = 1$, $b = \mathrm{e}^{\mathrm{i}2\pi/3}$, $c = \mathrm{e}^{-\mathrm{i}2\pi/3}$. Οποιαδήποτε αρχική τιμή στο μιγαδικό επίπεδο, εκτός από την $z = 0 + \mathrm{i}0$, θα συγκλίνει σε μία από τις ρίζες. Μπορούμε να δημιουργήσουμε μία έγχρωμη εικόνα αν σε κάθε σημείο στο μιγαδικό επίπεδο αντιστοιχήσουμε ένα χρώμα ανάλογα με το σε ποια ρίζα καταλήγει. Έτσι, π.χ., όσα σημεία καταλήγουν στην a τα χρωματίζουμε κόκκινα (RGB = $(255, 0, 0)$). Όσα καταλήγουν στην b τα χρωματίζουμε πράσινα (RGB = $(0, 255, 0)$) και όσα καταλήγουν στη c τα χρωματίζουμε μπλε (RGB = $(0, 0, 255)$). Το σημείο $0 + \mathrm{i}0$ το χρωματίζουμε λευκό (RGB = $(255, 255, 255)$).

(α′) Επιλέξτε στον άξονα των πραγματικών $N = 512$ ισαπέχουσες τιμές στο διάστημα $[-1, 1]$ (τα άκρα περιλαμβάνονται): x_i, $i = 0, \ldots, N - 1$.

(β′) Επιλέξτε στον άξονα των φανταστικών $M = 512$ ισαπέχουσες τιμές στο διάστημα $[-1, 1]$ (τα άκρα περιλαμβάνονται): y_j, $j = 0, \ldots, M - 1$.

(γ′) Σχηματίστε τον μιγαδικό αριθμό $z = x_i + \mathrm{i}y_j$ και βρείτε το «χρώμα» του με τη διαδικασία που περιγράφηκε παραπάνω.

(δ′) Αποθηκεύστε τα pixels (i, j) με το αντίστοιχο χρώμα τους σε αρχείο με όνομα *newton.pppm*. Χρησιμοποιήστε τη διαμόρφωση plain ppm (δείτε την άσκηση 10 στη σελίδα 119). Η εικόνα στο *newton.pppm* είναι ένα Newton fractal.

Κεφάλαιο 8

Χειρισμός σφαλμάτων

8.1 Εισαγωγή

Για να εξασφαλίσουμε την ορθή λειτουργία ενός προγράμματος δεν αρκεί να μεταγλωττίζεται χωρίς λάθη. Οι μεταβλητές και οι τύποι που χρησιμοποιούμε προϋποθέτουν να ισχύουν κάποιες συνθήκες (π.χ. οι τιμές των μεταβλητών να μπορούν να αναπαρασταθούν στους τύπους που επιλέξαμε, το πλήθος των στοιχείων ενός διανύσματος ή κάποιου container να μην ξεπερνά κάποια τιμή). Επίσης, οι συναρτήσεις που έχει ενσωματωμένη η γλώσσα ή ορίζουμε εμείς δέχονται ορίσματα με συγκεκριμένα πεδία ορισμού· αν τα παραβούμε, θα έχουμε λάθος αποτελέσματα.

Υπάρχουν δύο κατηγορίες λαθών: αυτά που μπορούν να εντοπιστούν κατά τη μεταγλώττιση (επομένως και κατά το γράψιμο του κώδικα) και αυτά που διαπιστώνονται κατά την εκτέλεση του προγράμματος. Η C++ παρέχει την εντολή `static_assert()` για τον εντοπισμό σφαλμάτων κατά τη μεταγλώττιση ενώ για τη δεύτερη κατηγορία παρέχει τη συνάρτηση `assert()` και το μηχανισμό `errno`, τα οποία κληρονόμησε από τη C, καθώς και τις *εξαιρέσεις (exceptions)* που τις αντικατέστησαν.

8.2 static_assert()

Η εντολή `static_assert()` καλείται ως εξής

`static_assert`(λογική_έκφραση, σταθερή_σειρά_χαρακτήρων);

Η «λογική_έκφραση» πρέπει να είναι κάποια σύγκριση, απλή ή σύνθετη, ή γενικότερα, κάποια ποσότητα που μπορεί να μετατραπεί σε λογική τιμή. Επιτρέπεται να αποτελείται μόνο από *σταθερές εκφράσεις* ώστε να μπορούν να υπολογιστούν από τον compiler κατά τη μεταγλώττιση. Αν η λογική_έκφραση είναι

αληθής, η μεταγλώττιση συνεχίζεται με την επόμενη εντολή. Αλλιώς, θα τυπωθεί ως σφάλμα από τον μεταγλωττιστή το (σταθερό) μήνυμα που περιέχεται στη σταθερή_σειρά_χαρακτήρων.

Παράδειγμα

Αναφέραμε στο §2.5.1 ότι ο μεγαλύτερος ακέραιος που μπορεί να αποθηκευτεί σε `int` είναι τουλάχιστον ο 32767. Επίσης, ξέρουμε ότι το άνω όριό του είναι η τιμή της ποσότητας `std::numeric_limits<int>::max()`.

Αν θέλουμε να εξασφαλίσουμε ότι ο τύπος `int` μπορεί να αναπαραστήσει ακέραιους μέχρι το 1000000 μπορούμε να γράψουμε

```
static_assert(std::numeric_limits<int>::max() > 1000000,
              "int␣is␣not␣sufficient");
```

8.3 assert()

Μία ιδιότυπη συνάρτηση που βοηθά στην ορθή λειτουργία ενός προγράμματος παρέχεται στη C++ με τη συμπερίληψη του header `<cassert>`· πρόκειται για τη macro[1] συνάρτηση `assert()`. Η συνάρτηση εξασφαλίζει ότι ικανοποιούνται κάποιες προϋποθέσεις που επιλέγει ο προγραμματιστής κατά την εκτέλεση του κώδικα. Η κλήση της γίνεται ως εξής:

```
assert(έκφραση);
```

Αν η τιμή της «έκφρασης» είναι 0, η εκτέλεση του προγράμματος διακόπτεται, τυπώνεται στο standard error του προγράμματος το αρχείο και η γραμμή στην οποία βρίσκεται η κλήση τής `assert()`, καθώς και το όρισμά της (η «έκφραση»).

Η κλήση της `assert()` αγνοείται και συνεπώς δεν μπορεί να προκαλέσει διακοπή της εκτέλεσης αν έχει οριστεί στον προεπεξεργαστή το όνομα `NDEBUG` *πριν* τη συμπερίληψη του `<cassert>`, αν δηλαδή υπάρχει πριν η εντολή `#define NDEBUG`, ή δοθεί αντίστοιχη εντολή κατά τη μεταγλώττιση.

Συνήθως, η `assert()` καλείται κατά τη διαδικασία του debugging, με όρισμα κάποια λογική συνθήκη η οποία μετατρέπεται σε 0 όταν είναι `false` (σύμφωνα με τους γνωστούς κανόνες, §2.5.3), προκαλώντας διακοπή της εκτέλεσης. Έτσι π.χ. αν ο κώδικας περιλαμβάνει την εντολή `assert(N<10);`, το πρόγραμμα σταματά με κατάλληλο πληροφοριακό μήνυμα αν δεν ισχύει το `(N<10)`.

Η συγκεκριμένη συνάρτηση καλείται *χωρίς* το πρόθεμα `std::` (δεν ανήκει στο χώρο ονομάτων std) καθώς είναι macro συνάρτηση και όχι μέρος της γλώσσας.

[1] έκφραση που παράγεται από τον προεπεξεργαστή της C++ και δεν είναι ενσωματωμένη στη γλώσσα.

8.4 Σφάλματα μαθηματικών συναρτήσεων

Οι ενσωματωμένες μαθηματικές συναρτήσεις της C++, Πίνακας 7.1, αλλά και οι πράξεις μεταξύ αριθμών, χρησιμοποιούν δύο μηχανισμούς για να ενημερώσουν για τη μη ορθή εκτέλεσή τους. Αν η ποσότητα `math_errhandling` έχει την τιμή `MATH_ERRNO` υποστηρίζεται η αλλαγή της τιμής της καθολικής μεταβλητής `errno`· αν έχει την τιμή `MATH_ERREXCEPT` υποστηρίζονται οι εξαιρέσεις πραγματικών αριθμών (Floating-point Exceptions, FE), και αν, όπως συνήθως, έχει το bitwise OR των δύο, δηλαδή `(MATH_ERRNO | MATH_ERREXCEPT)`, παρέχονται και οι δύο μηχανισμοί. Οι σταθερές `math_errhandling`, `MATH_ERRNO` και `MATH_ERREXCEPT` παρέχονται από το `<cmath>`.

Ο μηχανισμός των εξαιρέσεων προϋποθέτει ότι έχει δοθεί, είτε αυτόματα από τον compiler είτε ρητά από τον προγραμματιστή, η εντολή

```
#pragma STDC FENV_ACCESS on
```

προς τον προεπεξεργαστή. Η συγκεκριμένη εντολή εμποδίζει ορισμένες βελτιστοποιήσεις στις πράξεις και χρησιμοποιείται κατά την εύρεση σφαλμάτων (debugging). Όταν δεν χρειαζόμαστε πλέον το μηχανισμό των εξαιρέσεων πραγματικών αριθμών, μπορούμε να δώσουμε την εντολή

```
#pragma STDC FENV_ACCESS off
```

προς τον προεπεξεργαστή.

Στην περίπτωση που δοθεί όρισμα εκτός των επιτρεπόμενων τιμών σε μία μαθηματική συνάρτηση, η ποσότητα `errno` από το `<cerrno>` αποκτά την τιμή `EDOM` και εγείρεται η εξαίρεση `FE_INVALID`, η οποία ορίζεται στο `<cfenv>`. Οι περισσότερες συναρτήσεις σε αυτή την περίπτωση επιστρέφουν `NAN` (Not-A-Number)· ο έλεγχος αν μία ποσότητα είναι `NAN` μπορεί να γίνει με τη συνάρτηση `std::isnan()` του `<cmath>`. Αυτή δέχεται ως όρισμα την ποσότητα και επιστρέφει λογική τιμή.

Αν το αποτέλεσμα της μαθηματικής συνάρτησης είναι μεγαλύτερο από τα όριο που μπορεί να αναπαραστήσει ο επιστρεφόμενος τύπος της (overflow), η `errno` γίνεται `ERANGE` και εγείρεται η εξαίρεση `FE_OVERFLOW`. Η συνάρτηση επιστρέφει την τιμή (με πιθανό πρόσημο) `HUGE_VAL` ή `HUGE_VALF` ή `HUGE_VALL`, (ανάλογα αν το αποτέλεσμα είναι **`double`**, **`float`**, **`long double`**).

Αν το αποτέλεσμα είναι άπειρο, θετικό ή αρνητικό, η `errno` γίνεται `ERANGE`, εγείρεται η `FE_DIVBYZERO` και επιστρέφεται $\pm$`INFINITY`. Ο έλεγχος αν μία ποσότητα είναι `INFINITY` μπορεί να γίνει με τη συνάρτηση `std::isinf()` του `<cmath>`. Αυτή δέχεται ως όρισμα την ποσότητα και επιστρέφει λογική τιμή.

Στην περίπτωση που το αποτέλεσμα μιας μαθηματικής συνάρτησης είναι πολύ μικρό για να αναπαρασταθεί στον επιστρεφόμενο τύπο, μπορεί η `errno` να γίνει `ERANGE` και να εγερθεί η εξαίρεση `FE_UNDERFLOW`. Η επιστρεφόμενη ποσότητα μπορεί να γίνει 0.0 ή κάποια μη κανονική τιμή κοντά στο 0. Ο έλεγχος αν μία ποσότητα είναι μη κανονική μπορεί να γίνει με τη συνάρτηση `std::isnormal()` του `<cmath>`. Αυτή δέχεται ως όρισμα την ποσότητα και επιστρέφει λογική τιμή.

Αν επιθυμούμε να ελέγξουμε την ορθότητα της εκτέλεσης μιας μαθηματικής συνάρτησης, εκχωρούμε πριν την κλήση της, το 0 στην ποσότητα `errno` και ακυρώνουμε όλες τις εξαιρέσεις πραγματικών αριθμών με την εντολή

`std::feclearexcept(FE_ALL_EXCEPT);`

Μετά από την κλήση, ελέγχουμε την τιμή που έχει πλέον η `errno` ή εξετάζουμε ποια εξαίρεση έχει εγερθεί, με τη συνάρτηση `std::fetestexcept()` του `<cfenv>`. Η συγκεκριμένη συνάρτηση δέχεται ως όρισμα την εξαίρεση που επιθυμούμε να ελέγξουμε και επιστρέφει λογική τιμή **`true/false`** αν έχει εγερθεί ή όχι.

Παράδειγμα

```
#include <cerrno>
#include <cfenv>
#include <cmath>
#include <limits>
#include <iostream>

#pragma STDC FENV_ACCESS on

int
main()
{
  if (MATH_ERRNO != 1 && MATH_ERREXCEPT != 2) {
    std::cerr << "not␣supported\n";
    return -1;
  }

  errno = 0; // clear error. No error code is 0.
  std::feclearexcept(FE_ALL_EXCEPT); // clear exceptions

  std::sqrt(-1.0);
  // errno becomes EDOM, FE_INVALID is raised

  if (errno == EDOM || std::fetestexcept(FE_INVALID)) {
    std::cerr << "argument␣out␣of␣domain␣of␣function.\n";
  }

  errno = 0;
  std::feclearexcept(FE_ALL_EXCEPT);

  std::pow(std::numeric_limits<double>::max(), 2);
  // errno becomes ERANGE, FE_OVERFLOW is raised
```

```
  if (errno == ERANGE || std::fetestexcept(FE_OVERFLOW)) {
    std::cerr << "Math␣result␣not␣representable.\n";
  }

  errno = 0;
  std::feclearexcept(FE_ALL_EXCEPT);

  auto y = 1.0/0.0;
  // errno becomes ERANGE, FE_DIVBYZERO is raised

  if (errno == ERANGE || std::fetestexcept(FE_DIVBYZERO)) {
    std::cerr << "Division␣by␣zero.\n";
  }
}
```

Για την εκτύπωση των μηνυμάτων σφάλματος μπορεί να χρησιμοποιηθεί η συνάρτηση `std::strerror()` από το header `<cstring>`. Αυτή δέχεται ως μοναδικό όρισμα το `errno` και επιστρέφει `char *` με κατάλληλο πληροφοριακό μήνυμα, το οποίο μπορεί να τυπωθεί. Η γλώσσα των μηνυμάτων μπορεί να αλλάξει· δεν θα αναφερθούμε στο πώς.

Παράδειγμα

```
#include <cerrno>
#include <cmath>
#include <cstring>
#include <iostream>

int
main()
{
  errno = 0; // clear error
  std::cerr << std::strerror(errno) << '\n';

  std::sqrt(-1.0);   // errno becomes EDOM.
  std::cerr << std::strerror(errno) << '\n';
}
```

8.5 Εξαιρέσεις (exceptions)

(Υπό επεξεργασία.)

Μέρος II

Standard Library

Κεφάλαιο 9

Βασικές έννοιες της Standard Library

9.1 Εισαγωγή

Ένα ιδιαίτερα σημαντικό χαρακτηριστικό της C++ έναντι άλλων γλωσσών, είναι ότι παρέχει πλήθος δομικών στοιχείων για την ανάπτυξη κώδικα σε υψηλότερο επίπεδο, πιο απομακρυσμένο από το επίπεδο της μηχανής. Οι επεκτάσεις της βασικής γλώσσας βασίζονται στο μηχανισμό των κλάσεων (Κεφάλαιο 14) και των υποδειγμάτων (templates, §7.11, §14.9), και αποτελούν μέρος της Standard Library (SL).

Η SL έχει τρεις βασικές συνιστώσες:

- τους **containers**, δομές με κατάλληλα χαρακτηριστικά για την αποθήκευση και διαχείριση δεδομένων οποιουδήποτε τύπου, η κάθε μία με διαφορετικές ιδιότητες. Υποκαθιστούν τα ενσωματωμένα διανύσματα και επεκτείνουν σημαντικά τις περιορισμένες δυνατότητες που έχουν αυτά. Μεταξύ άλλων περιλαμβάνονται containers που παρέχουν τη δυνατότητα μεταβολής του πλήθους των στοιχείων τους, κάνουν αυτόματη ταξινόμηση (π.χ. `std::set<>`, `std::map<>`) και ταχύτατη ανάκτηση δεδομένων είτε με ακέραιο αριθμητικό δείκτη (π.χ. `std::vector<>`, `std::array<>`, `std::deque<>`) είτε με δείκτη οποιουδήποτε τύπου (π.χ. `std::map<>`). Έχουμε αναφέρει και χρησιμοποιήσει ήδη στο Κεφάλαιο 5 δύο containers, τους `std::array<>` και `std::vector<>`.

- τους **iterators**, ένα είδος δείκτη σε θέσεις μιας ακολουθίας στοιχείων, όπως π.χ. ενός container 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. Μπορούμε να προσπελάσουμε ένα στοιχείο ενός

container ή μιας ροής ανεξάρτητα από το πώς γίνεται σε χαμηλό επίπεδο η οργάνωση των δεδομένων σε αυτά.

- τους **αλγόριθμους**, που υλοποιούν με πολύ αποτελεσματικό τρόπο τμήματα κώδικα που χρειάζονται συχνά: ταξινόμηση στοιχείων μιας ακολουθίας, αναζήτηση ή αντικατάσταση στοιχείου με συγκεκριμένη τιμή σε αυτή κλπ. Οι αλγόριθμοι είναι σε μεγάλο βαθμό ανεξάρτητοι από τον τύπο του container που χρησιμοποιείται για την αποθήκευση των στοιχείων. Η ανεξαρτησία αυτή εξασφαλίζεται με τη χρήση των iterators.

Επιπλέον, η SL περιλαμβάνει τα *αντικείμενα-συναρτήσεις* (function objects) και τους *προσαρμογείς* αυτών (adapters), στα οποία θα αναφερθούμε παρακάτω. Οι *προσαρμογείς των containers* (container adapters) και το bitset, μέρη και αυτά της SL, δε θα παρουσιαστούν.

Μέχρι τώρα έχουμε χρησιμοποιήσει διάφορα τμήματα της Standard Library, καθώς κάθε τι που παρέχεται από headers (π.χ. είσοδος/έξοδος δεδομένων, μαθηματικές συναρτήσεις, όρια αριθμών, μιγαδικός τύπος, κλπ.) περιλαμβάνεται σε αυτή. Κάποια από αυτά υπάρχουν και στη C, αυτούσια ή παρόμοια. Σε αυτό το μέρος του βιβλίου θα δούμε κυρίως τα νέα χαρακτηριστικά που προσθέτει η SL.

Στο τρέχον κεφάλαιο θα παρουσιάσουμε ορισμένες βοηθητικές δομές και σχετικές έννοιες της γλώσσας. Στα επόμενα θα αναφερθούμε στους iterators, στους containers και στους αλγορίθμους που παρέχονται από την SL για τη διαχείριση των containers. Για εμβάθυνση στις παραπάνω έννοιες συμβουλευτείτε τη βιβλιογραφία ([2] και [3]).

9.2 Βοηθητικές Δομές και Συναρτήσεις

9.2.1 Ζεύγος (pair)

Η SL παρέχει containers που αποθηκεύουν ζεύγη τιμών και συναρτήσεις που χρειάζεται να επιστρέψουν δύο ποσότητες. Για την υποστήριξη αυτών, ο header `<utility>` περιλαμβάνει, ανάμεσα σε άλλα, την κλάση `std::pair<T1,T2>`. Είναι class template και περιέχει δύο μέλη με τύπους T1,T2 καθώς και τις κατάλληλες συναρτήσεις για το χειρισμό τους. Τα δύο βασικά μέλη έχουν ονόματα first και second. Ορισμός ενός ζεύγους (π.χ. για T1≡**int** και T2≡**double**) με απόδοση της προκαθορισμένης για κάθε τύπο τιμής ή ρητής αρχικής τιμής γίνεται ως εξής:

```
std::pair<int, double> p1;          // p1 = (0, 0.0)
std::pair<int, double> p2{3, 2.0}; // p2 = (3, 2.0)
```

Η πρόσβαση στα μέλη του `std::pair<>` είναι άμεση, με τη χρήση του ονόματός τους:

```
std::pair<int, double> p{3, 2.0};
std::cout << "first␣element␣is␣"  << p.first << '␣'
          << "second␣element␣is␣" << p.second <<'\n';
```

Εναλλακτικά, αντί για τα ονόματα των μελών μπορεί να χρησιμοποιηθεί το υπόδειγμα συνάρτησης `std::get()`. Η παράμετρος του template μπορούν να είναι οι ακέραιες σταθερές 0 ή 1, εξάγοντας αντίστοιχα το πρώτο ή το δεύτερο μέλος:

```
std::pair<int, double> p{3, 2.0};
std::cout << "first␣element␣is␣"  << std::get<0>(p) << '␣'
          << "second␣element␣is␣" << std::get<1>(p) << '\n';

std::get<0>(p) = 5; // p = (5,2.0)
```

Εναλλακτικά, μπορούμε να χρησιμοποιήσουμε ως παράμετρο του `std::get()` τον τύπο ενός μέλους του ζεύγους αρκεί βέβαια αυτός να είναι μοναδικός (ώστε να ξέρει ο μεταγλωττιστής ποιο στοιχείο να επιλέξει):

```
std::pair<int, double> p{3, 2.0};
std::cout << "first␣element␣is␣"  << std::get<int>(p) << '␣'
          << "second␣element␣is␣" << std::get<double>(p) << '\n';
```

Κατασκευή ενός `std::pair<>` μπορεί να γίνει χρησιμοποιώντας τη συνάρτηση `std::make_pair()`:

```
template <typename T1, typename T2>
std::pair<T1, T2>
make_pair(T1 const & f, T2 const & s);
```

του `<utility>` ως εξής:

```
std::pair<int, double> p; // p.first = 0, p.second = 0.0
p = std::make_pair(4, 3.0);//p.first = 4, p.second = 3.0
```

Ο τελεστής εκχώρησης μεταξύ δύο ζευγών, `p=q;`, αποδίδει την τιμή `q.first` στο `p.first` και την τιμή `q.second` στο `p.second`, κάνοντας μετατροπές τύπου, αν χρειάζονται. Μεταξύ ζευγών ίδιου τύπου ορίζονται οι γνωστοί τελεστές σύγκρισης (Πίνακας 3.1), αρκεί να έχουν νόημα για τους τύπους T1,T2. Για τον προσδιορισμό της σχέσης δύο ζευγών γίνεται πρώτα σύγκριση των μελών first. Αν δεν είναι ίσα, το αποτέλεσμα της σύγκρισής τους καθορίζει και τη σχέση των ζευγών. Αλλιώς, η σύγκριση των second είναι αυτή που καθορίζει αν τα ζεύγη είναι ίσα ή ποιο είναι μικρότερο και ποιο μεγαλύτερο.

9.2.2 Tuple

Η κλάση `std::tuple<T1,T2,T3,...>` γενικεύει το `std::pair<>` για οποιοδήποτε πλήθος στοιχείων. Παρέχεται από το header `<tuple>`. Ένα αντικείμενο αυτής της κλάσης αποθηκεύει μία n-άδα ποσοτήτων (με $n = 0, 1, \ldots$). Ορισμός ενός tuple (πλειάδας) (π.χ. για T1≡`int`, T2≡`double`, T3≡`int`) με απόδοση της προκαθορισμένης για κάθε τύπο τιμής ή ρητής αρχικής τιμής γίνεται ως εξής:

```
std::tuple<int, double, int> t1; // {0, 0.0, 0}
std::tuple<int, double, int> t2{3,4.1,-2};
```

Για την εισαγωγή στοιχείων σε μια πλειάδα μπορεί να χρησιμοποιηθεί η συνάρτηση `std::make_tuple()`:

```
std::tuple<int, double, int> t;
t = std::make_tuple(9,1.2,3);
```

Η συνάρτηση `std::get()` πρέπει να χρησιμοποιηθεί για την προσπέλαση των μελών ενός `std::tuple<>`. Η παράμετρος του template πρέπει να είναι σταθερός ακέραιος αριθμός από 0 έως $n-1$, όπου n το πλήθος των στοιχείων που αποθηκεύει η πλειάδα:

```
std::tuple<int, double, int> t{3,4.1,-2};
std::cout << std::get<0>(t) << '␣'
          << std::get<1>(t) << '␣'
          << std::get<2>(t) << '\n';
```

Εναλλακτικά, μπορούμε να χρησιμοποιήσουμε ως παράμετρο του `std::get()` τον τύπο ενός στοιχείου της πλειάδας, αρκεί βέβαια αυτός να είναι μοναδικός (ώστε να ξέρει ο μεταγλωττιστής ποιο μέλος να επιλέξει):

```
std::tuple<int, double, int> t{3,4.1,-2};
std::cout << std::get<double>(t) << '\n'; // 4.1
```

Οι τελεστές εκχώρησης και σύγκρισης γενικεύουν τους αντίστοιχους της δομής `std::pair<>`.

Ένα `std::tuple<>` δύο στοιχείων μπορεί να πάρει τιμή από ένα `std::pair<>`, αφού γίνουν αυτόματα οι πιθανές μετατροπές στον τύπο των στοιχείων.

9.2.3 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 τιμών, ως templates. Στην απλή τους μορφή, οι δύο πρώτες δέχονται δύο ποσότητες ή μια λίστα ποσοτήτων *ίδιου τύπου*[1] 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 `std::pair<>` με πρώτο στοιχείο το (πρώτο) ελάχιστο και δεύτερο το (τελευταίο) μέγιστο από αυτά. Η σύγκριση γίνεται με τον τελεστή '<':

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

[1] αν διαφέρουν οι τύποι των ορισμάτων, πρέπει να καλέσουμε τις συναρτήσεις και συγχρόνως να προσδιορίσουμε ρητά τον επιθυμητό τύπο ως παράμετρο του template. Σε αυτό τον τύπο θα μετατραπούν τα ορίσματα και αυτού του τύπου θα είναι το αποτέλεσμα. Έτσι στον κώδικα `auto a = std::min<double>(3,4.5);` το a είναι πραγματικός με τιμή 3.0.

Οι παραπάνω συναρτήσεις μπορούν να χρησιμοποιηθούν με την απλή μορφή τους αν για τον κοινό τύπο των ορισμάτων ορίζεται ο τελεστής '<'. Αυτό ισχύει για όλους τους ενσωματωμένους τύπους. Θα δούμε στο Κεφάλαιο 14 πώς ορίζονται νέοι τύποι από τον προγραμματιστή και πώς καθορίζεται η δράση των τελεστών σε αυτούς. Στην περίπτωση που για τα ορίσματα δεν ορίζεται ο συγκεκριμένος τελεστής ή επιθυμούμε να τα συγκρίνουμε με άλλο τρόπο, μπορούμε να προσδιορίσουμε το κριτήριο με το οποίο θα γίνει η σύγκριση των ορισμάτων στις παραπάνω συναρτήσεις ως εξής: δίνουμε ως τελευταίο όρισμα μία συνάρτηση ή ένα αντικείμενο–συνάρτηση (ή συνάρτηση λάμδα) που δέχεται δύο ορίσματα και επιστρέφει τη λογική τιμή της σύγκρισής τους (ανάλογα με το κριτήριο που έχουμε θέσει).

Παράδειγμα

Έστω ότι έχουμε δύο πραγματικούς αριθμούς a,b και θέλουμε να βρούμε τον μικρότερο *κατ' απόλυτη τιμή*. Προσέξτε ότι ο κώδικας

```
auto c = std::min(std::abs(a), std::abs(b));
```

δεν βρίσκει αυτό που ζητούμε.

Μπορούμε να χρησιμοποιήσουμε τη δεύτερη μορφή της `std::min()` αφού ορίσουμε μια κατάλληλη συνάρτηση σύγκρισης

```
bool
absless(double a, double b)
{
  return std::abs(a) < std::abs(b);
}
```

Τη συνάρτηση αυτή θα περάσουμε ως τελευταίο όρισμα στη `std::min()`:

```
auto c = std::min(a,b,absless);
```

9.2.4 Συνάρτηση εναλλαγής

Ακόμα μια βοηθητική συνάρτηση παρέχεται στο `<utility>` και είναι η συνάρτηση εναλλαγής, `std::swap()`. Η συγκεκριμένη δέχεται δύο ποσότητες ίδιου τύπου και εναλλάσσει τις τιμές τους, με *μετακίνηση*[2]:

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

Όπως θα δούμε παρακάτω, οι μεταβλητές στη `std::swap<>` μπορούν να είναι containers ή άλλοι σύνθετοι τύποι (π.χ. `std::complex<>`).

[2] δείτε την §2.17.1 για τη σχετική συζήτηση.

Η `std::swap()` μπορεί εναλλακτικά να δεχτεί δύο ενσωματωμένα διανύσματα, ίδιου πλήθους στοιχείων, και να εναλλάξει (μόνο) τους δείκτες στα πρώτα στοιχεία τους. Με αυτό τον τρόπο εκτελεί ουσιαστικά μια πολύ γρήγορη εναλλαγή των αντίστοιχων στοιχείων των διανυσμάτων:

```
int a[100];
int b[100];
 ...
a[15] = 9;
b[15] = 4;
 ...
std::swap(a,b);  // a[15] = 4, b[15] = 9
```

9.3 Αντικείμενο–Συνάρτηση

Θα συναντήσουμε πολλές φορές στον ορισμό των containers και ιδιαίτερα στους αλγόριθμους, την έννοια ενός αντικειμένου που όταν ακολουθείται από ζεύγος παρενθέσεων με κανένα, ένα ή περισσότερα ορίσματα, επιστρέφει κάποια τιμή· συμπεριφέρεται δηλαδή ως συνάρτηση. Η ποσότητα αυτή χαρακτηρίζεται ως αντικείμενο–συνάρτηση (function object ή functor). Είναι αντικείμενο μιας κλάσης για την οποία ορίζεται ο τελεστής '`()`'· θα δούμε πώς στο Κεφάλαιο 14. Εναλλακτικά, μπορεί να είναι κάποια *συνάρτηση λάμδα* (§9.3.1).

Με την συμπερίληψη του header `<functional>`, η C++ παρέχει στο χώρο ονομάτων `std` ένα αριθμό από προκαθορισμένα αντικείμενα–συναρτήσεις. Είναι όλα class templates και δέχονται ως μοναδική παράμετρο τον τύπο του ενός ή των δύο ορισμάτων που θα τους «περάσει» ο αλγόριθμος που θα τα χρησιμοποιήσει. Τα προκαθορισμένα αντικείμενα–συναρτήσεις δίνονται στον Πίνακα 9.1 μαζί με την πράξη που εκτελούν.

Ας εξηγήσουμε τον τρόπο χρήσης και λειτουργίας τους έχοντας ως παράδειγμα ένα από αυτά, το `std::plus<T>`. Με τη δήλωση

```
std::plus<int> a;
```

ορίζουμε ένα αντικείμενο, μια ποσότητα δηλαδή, αυτού του τύπου, με προκαθορισμένη τιμή. Το αντικείμενο a έχει την ιδιότητα, όταν ακολουθείται από δύο ακέραιους σε παρένθεση (ορίσματα), να έχει ως τιμή το άθροισμά τους. Προσέξτε πόσο μοιάζει η συγκεκριμένη πράξη με την κλήση μιας συνάρτησης[3]:

```
auto b = a(3,5);
```

Το b είναι `int` με τιμή 8.

[3] στην πραγματικότητα, γίνεται κλήση του τελεστή '`()`' που ορίζεται στην συγκεκριμένη κλάση, με δύο ορίσματα. Η κλήση αυτή προσδιορίζεται από τη συνάρτηση–μέλος της κλάσης, `operator()`, και η επιστρεφόμενη τιμή από αυτή είναι η τιμή της έκφρασης.

Γενικότερα, η έκφραση `std::plus<int>{}` δημιουργεί ένα ανώνυμο αντικείμενο τύπου `std::plus<int>` με μία προκαθορισμένη τιμή, όπως ακριβώς η έκφραση `double()` ή `double{}` δημιουργεί έναν (ανώνυμο) πραγματικό με τιμή 0.0.

Η χρησιμότητα και ο τρόπος χρήσης των αντικειμένων–συναρτήσεων σε αλγόριθμους και containers θα παρουσιαστούν στα επόμενα κεφάλαια.

Πίνακας 9.1: Προκαθορισμένα αντικείμενα–συναρτήσεις της C++

Αντικείμενο–Συνάρτηση	**Τιμή τελεστή** `'()'`
`negate<T>{}`	−όρισμα
`plus<T>{}`	όρισμα1 + όρισμα2
`minus<T>{}`	όρισμα1 − όρισμα2
`multiplies<T>{}`	όρισμα1 ∗ όρισμα2
`divides<T>{}`	όρισμα1 / όρισμα2
`modulus<T>{}`	όρισμα1 % όρισμα2
`equal_to<T>{}`	όρισμα1 == όρισμα2
`not_equal_to<T>{}`	όρισμα1 != όρισμα2
`less<T>{}`	όρισμα1 < όρισμα2
`greater<T>{}`	όρισμα1 > όρισμα2
`less_equal<T>{}`	όρισμα1 <= όρισμα2
`greater_equal<T>{}`	όρισμα1 >= όρισμα2
`logical_not<T>{}`	!όρισμα
`logical_and<T>{}`	όρισμα1 && όρισμα2
`logical_or<T>{}`	όρισμα1 || όρισμα2
`bit_and<T>{}`	όρισμα1 & όρισμα2
`bit_or<T>{}`	όρισμα1 | όρισμα2
`bit_xor<T>{}`	όρισμα1 ^ όρισμα2

9.3.1 Συναρτήσεις λάμδα

Μια συνάρτηση λάμδα είναι ένας εύχρηστος μηχανισμός για να ορίσουμε ένα αντικείμενο–συνάρτηση, εναλλακτικός της κλάσης. Η συνάρτηση λάμδα μπορεί να παρουσιαστεί σε οποιοδήποτε σημείο του κώδικα θα μπορούσε να εμφανιστεί μια δήλωση ποσότητας και συντάσσεται ως εξής:

1. Ο ορισμός της ξεκινά με τις μη στατικές, §2.2, ποσότητες που χρειάζεται η συνάρτηση από το περιβάλλον της, γραμμένες εντός αγκυλών, '`[]`', και χωρισμένες με κόμματα. Οι ποσότητες αυτές καθορίζουν την κατάστασή της. Σε αυτό το σημείο προσδιορίζουμε και αν δέχεται αναφορά σε αυτές τις ποσότητες, οπότε τις χρησιμοποιεί απευθείας, ή χρειάζεται μόνο τις τιμές τους, οπότε οι ποσότητες αντιγράφονται. Π.χ. αν η συνάρτηση λάμδα χρειάζεται να τροποποιήσει την ποσότητα a και να διαβάσει την τιμή της ποσότητας b, μπορούμε να την εισαγάγουμε με `[&a,b]`.

Η λίστα μεταξύ των αγκυλών μπορεί να είναι κενή. Επίσης, μπορεί να έχει μόνο το σύμβολο '`&`' οπότε έχει πρόσβαση με αναφορά σε όλες τις ποσότητες του περιβάλλοντός της. Αν υπάρχει το σύμβολο '`=`' μπορεί να χρησιμοποιήσει τα αντίγραφα όλων των ποσοτήτων.

2. Ακολουθεί μια λίστα ορισμάτων εντός παρενθέσεων. Σε αυτή προσδιορίζουμε, όπως ακριβώς και σε μία συνήθη συνάρτηση, τις ποσότητες που δέχεται κατά την κλήση της. Υπάρχει διαφορά της λίστας των ορισμάτων από την προηγούμενη λίστα των ποσοτήτων από το περιβάλλον: οι ποσότητες εντός αγκυλών περνούν μία φορά στη συνάρτηση λάμδα, όταν την προσδιορίζουμε ως όρισμα σε κάποιο αλγόριθμο· αντίθετα, οι ποσότητες εντός παρενθέσεων περνούν κάθε φορά που καλείται η συνάρτηση λάμδα. Η λίστα ορισμάτων μπορεί να είναι κενή και τότε μπορούμε να παραλείψουμε τις παρενθέσεις. Τα ορίσματα επιτρέπεται να έχουν προεπιλεγμένες τιμές (§7.6).

3. Ακολουθεί προαιρετικά η προκαθορισμένη λέξη **`mutable`**, αν επιθυμούμε να μπορούμε να τροποποιούμε την κατάσταση της συνάρτησης λάμδα. Η αλλαγή της εσωτερικής κατάστασης σημαίνει να αλλάζουν οι ποσότητες εντός αγκυλών που έχουν περάσει με *αντιγραφή* (δηλαδή να αλλάζουν τα αντίγραφά τους). Η πιθανή αλλαγή των ποσοτήτων αυτών σημαίνει ότι οι κλήσεις της συνάρτησης λάμδα δεν είναι ανεξάρτητες μεταξύ τους.

4. Ακολουθεί προαιρετικά η προκαθορισμένη λέξη **`noexcept`**. Ενημερώνει ότι η συνάρτηση λάμδα δεν έχει εξαιρέσεις (exceptions).

5. Ακολουθεί προαιρετικά ο προσδιορισμός του τύπου της επιστρεφόμενης ποσότητας στη μορφή `->type`. Αν δεν προσδιορίζεται ρητά, είναι δυνατό να προκύψει αυτόματα από το σώμα της συνάρτησης: αν δεν εμφανίζεται κανένα **`return`** ο τύπος είναι **`void`** ενώ αν υπάρχει *μία* εντολή **`return`** ο τύπος προκύπτει από την έκφραση που προσδιορίζεται σε αυτή. Διαφορετικά, δεν γίνεται αυτόματη αναγνώριση του τύπου.

6. Ακολουθεί το σώμα της συνάρτησης λάμδα εντός αγκίστρων. Το σώμα συνήθως είναι πολύ απλό, ίσως μόνο μια εντολή **`return`** που επιστρέφει κατάλληλη ποσότητα.

Παράδειγμα

Μία 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);
}
```

Η συνάρτηση λάμδα είναι ανώνυμη, δεν μπορεί να καλέσει τον εαυτό της αναδρομικά, και στη συγκεκριμένη μορφή μπορεί να χρησιμοποιηθεί μόνο ως όρισμα κάποιου αλγόριθμου ή προσαρμογέα, όπως θα δούμε παρακάτω. Εναλλακτικά, μπορούμε να δώσουμε όνομα σε μια συνάρτηση λάμδα, ως εξής:

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

Προσέξτε το καταληκτικό ‘;’ που ολοκληρώνει τη δήλωση της αναφοράς fun στην ανώνυμη συνάρτηση λάμδα. Δείτε επίσης την §2.17.1. Η χρήση της συνάρτησης λάμδα μπορεί πλέον να γίνεται και αυτόνομα, χωρίς να χρειάζεται να είναι όρισμα:

```
auto x = fun(-5,3); // x = 3
```

Παρατήρηση: Μια συνάρτηση λάμδα δεν μπορεί να είναι template. Όμως, μπορεί να συμπεριφερθεί ως template αν, ως τύπος ενός ή περισσότερων ορισμάτων προσδιοριστεί η προκαθορισμένη λέξη `auto`:

Παράδειγμα

Η συνάρτηση λάμδα

```
auto && g = [](auto x, auto y) {return x+y;};
```

αθροίζει τα ορίσματά της, όποιου τύπου κι αν είναι, ακόμα και διαφορετικού. Η εντολή

```
std::cout << g(2,3) << '␣' << g(2.4,3) << '␣'
          << g(2.4,3.4) <<'\n';
```

τυπώνει

```
5 5.4 5.8
```

9.3.2 Προσαρμογείς (adapters)

Τα αντικείμενα–συναρτήσεις, είτε είναι από τα προκαθορισμένα είτε όχι, οι συναρτήσεις λάμδα ή ακόμα και οι συνήθεις συναρτήσεις, μπορούν να τροποποιηθούν με τη βοήθεια των *προσαρμογέων* (adapters). Οι προσαρμογείς παρέχονται στο χώρο ονομάτων `std` από το header `<functional>`.

Ακολουθεί η περιγραφή των δύο πιο χρήσιμων προσαρμογέων για αντικείμενα–συναρτήσεις.

`std::bind()`

Ο πιο σημαντικός από τους προσαρμογείς είναι ο `std::bind()`. Με κατάλληλη χρήση του μπορούμε να τροποποιήσουμε μία συνάρτηση ή ένα αντικείμενο–συνάρτηση ώστε ένα ή περισσότερα από τα ορίσματα αυτών να έχουν συγκεκριμένες τιμές. Παράγουμε έτσι μια νέα συνάρτηση ή ένα νέο αντικείμενο–συνάρτηση

με ίσα ή λιγότερα ορίσματα από τα αρχικά και με συγκεκριμένες τιμές για όσα λείπουν.

Παράδειγμα

```
std::minus<int> a;

auto && b = std::bind(a, 10, 2);
std::cout << b() << '\n'; // 10-2 -> 8

auto && c = std::bind(a, std::placeholders::_1, 10);
std::cout << c(3) << '\n'; // 3-10 -> -7

auto && d = std::bind(a, 10, std::placeholders::_1);
std::cout << c(3) << '\n'; // 10-3 -> 7

auto && e = std::bind(a, std::placeholders::_2,
                      std::placeholders::_1);
std::cout << e(3,5) << '\n'; // 5-3 -> 2
```

Κατά τη δήλωση του b, το `std::bind()` δέχεται ως πρώτο όρισμα το a, ένα αντικείμενο–συνάρτηση (που μπορεί να είναι και ανώνυμο), και ως επόμενα, συγκεκριμένες τιμές για τα ορίσματα του a. Το νέο αντικείμενο–συνάρτηση, το b, καλείται χωρίς τιμές εντός παρενθέσεων και δίνει πάντα την ίδια τιμή.

Στη δήλωση της ποσότητας c, το `std::bind()` δέχεται ως πρώτο όρισμα ένα αντικείμενο–συνάρτηση, ως δεύτερο την ποσότητα _1 που ορίζεται στο χώρο ονομάτων `std::placeholders`, και ως τρίτο μία τιμή. Το νέο αντικείμενο–συνάρτηση που παράγεται από το αρχικό, το c, θα παίρνει ως πρώτη τιμή αυτή που θα προσδιορίζεται πρώτη (και μοναδική) κατά την κλήση του τελεστή '`()`' στο c και ως δεύτερη το 10.

Στη δημιουργία του d, το νέο αντικείμενο–συνάρτηση έχει ως πρώτη τιμή το 10 και ως δεύτερη, την πρώτη (και μοναδική) τιμή που θα προσδιορίζεται κατά την κλήση του τελεστή '`()`'.

Στη δήλωση του e, η δεύτερη τιμή που θα προσδιοριστεί κατά την κλήση του τελεστή '`()`' θα περάσει ως πρώτο όρισμα του αρχικού αντικειμένου–συνάρτηση ενώ η πρώτη τιμή θα αποτελέσει το δεύτερο όρισμα.

Ο προσαρμογέας `std::bind()` μπορεί επίσης να χρησιμοποιηθεί για να καλέσουμε συναρτήσεις–μέλη κάποιου αντικειμένου. Ως πρώτο όρισμά του προσδιορίζουμε τη *διεύθυνση* μιας συνάρτησης–μέλους, ή γενικότερα, ενός μέλους μιας κλάσης. Ως δεύτερο όρισμα ορίζουμε ένα αντικείμενο ή δείκτη σε αντικείμενο ή συνηθέστερα, την τιμή `std::placeholders::_1` αν θέλουμε να πάρει τιμή από το πρώτο όρισμα κατά την κλήση της προκύπτουσας συνάρτησης. Κατόπιν, ακολουθούν τιμές για τα ορίσματα της συνάρτησης–μέλους, αν προβλέπονται, πιθανώς με τη χρήση των ονομάτων `std::placeholders::_2`, `std::placeholders::_3`, κλπ.

Παράδειγμα

```
using cntr = std::array<int,10>;
cntr a, b;
auto && sizea = std::bind(&cntr::size, a);
std::cout << "the␣size␣of␣a␣is␣" << sizea() << '\n';
auto && size = std::bind(&cntr::size, std::placeholders::_1);
std::cout << "the␣size␣of␣b␣is␣" << size(b) << '\n';
```

`std::mem_fn()`

Ο δεύτερος σημαντικός προσαρμογέας είναι ο `std::mem_fn()`. Μπορεί να υποκαταστήσει το `std::bind()` στη συγκεκριμένη εφαρμογή που είδαμε παραπάνω. Ο `std::mem_fn()` δέχεται ως μόνο όρισμα τη διεύθυνση[4] μιας συνάρτησης–μέλους και επιστρέφει συνήθη συνάρτηση με τα ίδια ορίσματα της συνάρτησης–μέλους, συμπληρωμένα όμως με ένα ακόμα όρισμα, το πρώτο. Όταν κληθεί αυτή η νέα συνάρτηση, πρέπει να προσδιορίσουμε ως πρώτο όρισμα το αντικείμενο στο οποίο θα δράσει η συνάρτηση–μέλος και κατόπιν τα ορίσματά της:

```
std::array<int,10> a;
auto && size = std::mem_fn(&std::array<int,10>::size);
std::cout << "the␣size␣of␣a␣is␣" << size(a) << '\n';
```

9.4 Βοηθητικές έννοιες

Σε επόμενα κεφάλαια θα χρειαστούμε τις έννοιες της λεξικογραφικής σύγκρισης δύο ακολουθιών και της γνήσιας ασθενούς διάταξης. Θα τις παρουσιάσουμε εδώ.

9.4.1 Λεξικογραφική σύγκριση

Η έννοια της λεξικογραφικά «μικρότερης» ακολουθίας ως προς κάποια άλλη, προσδιορίζεται ως εξής:

Συγκρίνονται μεταξύ τους τα στοιχεία που βρίσκονται στις ίδιες θέσεις στις δύο ακολουθίες, ξεκινώντας από την πρώτη θέση και προχωρώντας μέχρι την τελευταία. Αν βρεθεί ζεύγος άνισων στοιχείων, το αποτέλεσμα της σύγκρισής τους είναι η τιμή της σύγκρισης των ακολουθιών. Αν όλα τα αντίστοιχα στοιχεία είναι ίσα μέχρι να τελειώσει η μία από τις δύο ακολουθίες, τότε η ακολουθία με τα λιγότερα στοιχεία (αυτή που εξαντλήθηκε πρώτη) είναι η μικρότερη. Αν εξαντληθούν ταυτόχρονα οι ακολουθίες, είναι ίσες.

[4] η οποία βρίσκεται με τη δράση του τελεστή '`&`', §2.18.

9.4.2 Γνήσια ασθενής διάταξη

Συχνά θα συναντήσουμε την έννοια μίας συνάρτησης δύο ορισμάτων ίδιου τύπου που επιστρέφει λογική τιμή, `true/false`, αν το πρώτο είναι «μικρότερο» (ό,τι κι αν σημαίνει αυτό) από το δεύτερο. Λέμε ότι η συνάρτηση αυτή, έστω `comp()`, καθορίζει *γνήσια ασθενή διάταξη* αν ικανοποιούνται τα παρακάτω κριτήρια:

- `comp(a,a)==false` για οποιοδήποτε a, δηλαδή κανένα στοιχείο δεν είναι «μικρότερο» από τον εαυτό του.
- Αν ισχύει ότι `comp(a,b)==true` και `comp(b,c)==true` τότε ισχύει και ότι `comp(a,c)==true` για οποιαδήποτε a,b,c. Δηλαδή, ισχύει ότι το a είναι «μικρότερο» από το c όταν το a είναι «μικρότερο» από το b και το b είναι «μικρότερο» από το c.
- Μπορούμε να ορίσουμε την ισοδυναμία δύο στοιχείων a,b όταν ισχύουν ταυτόχρονα τα `comp(a,b)==false` και `comp(b,a)==false`, δηλαδή όταν κανένα δεν είναι «μικρότερο» από το άλλο. Τότε, αν το a είναι ισοδύναμο του b και το b είναι ισοδύναμο του c, υποχρεωτικά το a είναι ισοδύναμο του c, για οποιαδήποτε a,b,c.

9.5 Ασκήσεις

1. Δημιουργήστε ένα αντικείμενο–συνάρτηση με όνομα lt0. Αυτό θα δέχεται ένα πραγματικό όρισμα και θα επιστρέφει λογική τιμή, `true` ή `false`, αν είναι αρνητικό ή όχι. Να ξεκινήσετε από ένα προκαθορισμένο αντικείμενο–συνάρτηση και να χρησιμοποιήσετε τον προσαρμογέα `std::bind()`.

2. Γράψτε μία συνάρτηση λάμδα που θα δέχεται ένα ακέραιο όρισμα και θα επιστρέφει λογική τιμή, `true` ή `false`, αν είναι θετικό ή όχι.

3. Προσαρμόστε ένα προκαθορισμένο αντικείμενο–συνάρτηση ώστε να ελέγχει αν το ακέραιο όρισμά του είναι άρτιος.

4. Γράψτε μία συνάρτηση λάμδα που θα δέχεται ένα πραγματικό όρισμα και θα επιστρέφει λογική τιμή, `true` ή `false`, αν είναι ίσο ή όχι με μια πραγματική μεταβλητή του περιβάλλοντός της με όνομα a. Μπορείτε να το κάνετε προσαρμόζοντας ένα προκαθορισμένο αντικείμενο–συνάρτηση;

5. Με συνδυασμό προκαθορισμένων αντικειμένων–συναρτήσεων, αφού τα προσαρμόσετε κατάλληλα, δημιουργήστε ένα αντικείμενο–συνάρτηση που θα δέχεται ένα πραγματικό όρισμα και θα ελέγχει αν αυτό ανήκει στο διάστημα $[-2.5, 4.5)$. Μπορείτε να το κάνετε με συνάρτηση λάμδα;

6. Δημιουργήστε με τη χρήση του `std::mem_fn()` ένα αντικείμενο–συνάρτηση που θα επιστρέφει το φανταστικό μέρος του μιγαδικού ορίσματός του. Θα καλεί τη συνάρτηση–μέλος `imag()` της κλάσης `std::complex<>`.

Κεφάλαιο 10

Iterators

10.1 Εισαγωγή

Οι iterators είναι το βασικό «εργαλείο» για την προσπέλαση στοιχείων σε ένα container. Το πιο σημαντικό χαρακτηριστικό τους είναι ότι αποτελούν τη βάση για τον ομοιόμορφο τρόπο χειρισμού οποιουδήποτε container.

Ένας iterator συμπεριφέρεται σε μεγάλο βαθμό ως δείκτης σε στοιχείο ενός container παρόλο που διαφέρει ως έννοια από το είδος του δείκτη που παρουσιάσαμε στο §2.18. Σε αντίθεση με τους συνήθεις δείκτες, ένας iterator έχει νόημα μόνο για τις θέσεις αποθήκευσης σε ένα συγκεκριμένο container (ή `std::string` ή ροή ή ενσωματωμένο διάνυσμα) καθώς και σε μία θέση μετά το τελευταίο στοιχείο. Δεν μπορεί να εξαχθεί με τη δράση του τελεστή '`&`' στο όνομα ενός στοιχείου ή μιας θέσης σε container[1]. Ένας iterator προσδιορίζεται από την επιστρεφόμενη τιμή αλγόριθμων της Standard Library ή συναρτήσεων–μελών των containers ή παράγεται από άλλο iterator.

10.2 Δήλωση

Μια ποσότητα με όνομα, π.χ., it, μπορεί να οριστεί ως iterator για ένα container (π.χ. `std::vector<double>`) ως εξής:

```
std::vector<double>::iterator it;
```

Παρατηρήστε ότι ενώ η *έννοια* του iterator είναι κοινή για όλους, ο *τύπος* του iterator είναι άμεσα συνδεδεμένος με τον container στον οποίο αναφέρεται και στον οποίο μπορεί να χρησιμοποιηθεί.

[1] η δράση του συγκεκριμένου τελεστή παράγει ένα συνήθη δείκτη.

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. Επίσης, η `c.end()` επιστρέφει iterator που «δείχνει» στην *επόμενη θέση μετά την τελευταία* θέση αποθήκευσης του c. Επομένως, οι δηλώσεις δύο iterators με ονόματα b, e, στην αρχή και *σε μία θέση μετά το τέλος* ενός container τύπου `std::vector<double>`, με όνομα v, είναι

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

Παρατηρήστε ότι ο τύπος των μεταβλητών b,e είναι αρκετά σύνθετος και μακρύς. Η λέξη `auto` που προκαλεί αυτόματη αναγνώριση του τύπου σε μια δήλωση με αρχική τιμή (§2.2.1) είναι ιδιαίτερα χρήσιμη. Με αυτή, οι παραπάνω δηλώσεις απλοποιούνται σε

```
auto b = v.begin();
auto e = v.end();
```

Αντί να χρησιμοποιήσουμε συνάρτηση–μέλος για την παραγωγή iterator στην αρχή ή σε μία θέση μετά το τέλος ενός container, μπορούμε να καλέσουμε τις συνήθεις συναρτήσεις `std::begin()` και `std::end()` με όρισμα το όνομα του container:

```
auto b = std::begin(v); // b ≡ v.begin()
auto e = std::end(v);   // e ≡ v.end()
```

Οι συναρτήσεις `std::begin()`, `std::end()` παρέχονται από οποιοδήποτε header παρέχει ένα container· δηλώνονται επίσης και στον `<iterator>`.

Καλό είναι να δίνουμε τη δυνατότητα στον compiler να βελτιστοποιεί τον κώδικά μας και να μας ενημερώνει αν σε κάποιο σημείο κατά λάθος προσπαθήσουμε να μεταβάλουμε στοιχεία ενός container, ενώ δεν θα έπρεπε να το κάνουμε. Γι' αυτό το σκοπό ορίζονται οι iterators σε σταθερές ποσότητες. Iterator σε ένα π.χ. `std::vector<double>`, μέσω του οποίου δεν μπορεί να αλλάξει η τιμή στη θέση που «δείχνει», ορίζεται ως εξής:

```
std::vector<double>::const_iterator it;
```

Κάθε container παρέχει iterators που δεν μπορούν να μεταβάλουν τα στοιχεία στα οποία «δείχνουν», μέσω των συναρτήσεων–μελών `cbegin()` και `cend()`, κατ' αναλογία των `begin()` και `end()`.

Ένας `iterator` μπορεί να μετατραπεί αυτόματα σε `const_iterator` ώστε να αποτελέσει αρχική τιμή ή να εκχωρηθεί ή να συμμετέχει σε κάποια σύγκριση με `const_iterator`. Το αντίστροφο δεν ισχύει. Αυτό σημαίνει ότι στις παρακάτω δηλώσεις

```
std::vector<int> v(100);
std::vector<int>::const_iterator it1{v.begin()};//correct
std::vector<int>::iterator it2{v.cbegin()}; // error
```

η τελευταία είναι λάθος.

Ας υπενθυμίσουμε στο σημείο αυτό ότι όταν δηλώνεται ένας iterator, όπως και οποιαδήποτε ποσότητα, μπορεί να προσδιοριστεί ως `const`. Τότε είναι απαραίτητη η απόδοση αρχικής (και μόνιμης) «τιμής» του, όπως ισχύει για οποιαδήποτε ποσότητα. Π.χ.

```
std::vector<double> v(100);
auto const it1 = v.begin();
auto const it2 = v.cbegin();
```

Παρατηρήστε ότι οι it1, it2, για όλη τη διάρκεια της ζωής τους θα δείχνουν σε συγκεκριμένη θέση στον v (στην αρχή του) και δεν μπορούν να μετακινηθούν. Ακόμα, μέσω του it1 μπορούμε να αλλάξουμε την τιμή του πρώτου στοιχείου του v (θα δούμε παρακάτω πώς) ενώ δεν μπορούμε να το κάνουμε μέσω του it2.

10.2.1 Iterator σε παράμετρο template

Προσέξτε ότι υπάρχει περίπτωση ο τύπος των στοιχείων ενός container ή ο ίδιος ο container να είναι άγνωστος, να ορίζεται, δηλαδή, ως παράμετρος σε template. Έτσι π.χ., μπορούμε να έχουμε

```
template<typename C, typename T>
void
f(C & a)
{
   std::vector<T> v;
   C b;
   ...
}
```

Μέσα στη συνάρτηση f() ορίζουμε ένα `std::vector<>` για αποθήκευση στοιχείων τύπου T και μια ποσότητα τύπου C (που μπορεί να είναι κάποιος container, π.χ. `std::vector<int>`). Αν θελήσουμε να ορίσουμε iterators για αυτές τις ποσότητες δεν αρκεί να γράψουμε

```
std::vector<T>::iterator itv;   // error
C::iterator itb;  // error
```

Πρέπει να ενημερώσουμε το μεταγλωττιστή ότι οι εκφράσεις `C::iterator` και `std::vector<T>::iterator` αποτελούν *τύπους* (και όχι μέλη των κλάσεων, όπως θεωρεί από μόνος του). Αυτό γίνεται αν συμπληρώσουμε τις δηλώσεις των iterators με τη λέξη `typename`:

```
typename std::vector<T>::iterator itv;   // correct
typename C::iterator itb;  // correct
```

10.3 Χρήση

Αναφέραμε ότι ο iterator και ο δείκτης συμπεριφέρονται με τον ίδιο τρόπο. Αυτό ισχύει καθώς:

- Η δράση, από τα αριστερά, του τελεστή '`*`' στο όνομα ενός iterator, μας δίνει πρόσβαση στην ποσότητα που αποθηκεύεται στη θέση του container στην οποία «δείχνει» ο iterator. Δηλαδή, *η ποσότητα* `*it` *είναι το στοιχείο που βρίσκεται, σε ένα container, στη θέση που «δείχνει» ο iterator* `it`. Η δράση του τελεστή '`*`' σε iterator όπως ο `end()`, που δεν «δείχνει» σε στοιχείο ενός container δεν έχει νόημα (και είναι λάθος).

- Η δράση του τελεστή '`->`' δίνει πρόσβαση σε μέλος του στοιχείου στο οποίο «δείχνει» ένας iterator. Δηλαδή, αν το στοιχείο στη θέση που δείχνει ο iterator `it` έχει μέλος με όνομα `member`, η έκφραση `(*it).member` ισοδυναμεί με `it->member`. Στην πρώτη έκφραση οι παρενθέσεις είναι αναγκαίες λόγω της χαμηλότερης προτεραιότητας του τελεστή πρόσβασης σε τιμή, '`*`', ως προς τον τελεστή επιλογής μέλους, '`.`'.

- Η δράση σε ένα iterator, είτε από τα αριστερά του είτε από τα δεξιά, του τελεστή '`++`', και σε κάποιες κατηγορίες iterators, του τελεστή '`--`', έχει ως αποτέλεσμα να *μετακινείται* ο iterator στην επόμενη ή την προηγούμενη θέση, αντίστοιχα. Σε αντίθεση με τους δείκτες, δεν επιτρέπεται να μετακινήσουμε ένα iterator πριν την αρχική ή μετά την τελική του επιτρεπτή θέση (π.χ. το `begin()` και το `end()`, αντίστοιχα, ενός container).

- Στους iterators τυχαίας προσπέλασης, μπορούμε επιπλέον να προσθέσουμε ή να αφαιρέσουμε ένα ακέραιο αριθμό `n` και να έχουμε ως αποτέλεσμα ένα άλλο iterator που «δείχνει» `n` θέσεις μετά (προς το τέλος) ή πριν (προς την αρχή του container). Σε αυτή την κατηγορία iterators έχουν επίσης νόημα οι σύνθετοι τελεστές '`+=`' και '`-=`', που προκαλούν μετακίνηση του iterator που βρίσκεται στο αριστερό τους μέλος κατά όσες θέσεις προσδιορίζει ο ακέραιος στο δεξί τους μέλος.

- Στα περισσότερα είδη iterators, δύο iterators *ίδιου τύπου, που δείχνουν στον ίδιο container,* μπορούν να συγκριθούν μεταξύ τους με τους τελεστές '`==`', '`!=`', ώστε να διαπιστώνουμε αν είναι ίσοι—δηλαδή, «δείχνουν» στην ίδια θέση—ή όχι. Στους iterators τυχαίας προσπέλασης έχουν νόημα και οι υπόλοιποι τελεστές σύγκρισης· έτσι, η έκφραση `it1 < it2` είναι αληθής όταν ο `it1` «δείχνει» πριν τον `it2` (εννοείται στον ίδιο container).

10.3.1 Παραδείγματα

Ας δούμε τη χρήση των iterators με παραδείγματα:

Παράδειγμα

Έστω ότι θέλουμε να δώσουμε την τιμή 3.5 στα στοιχεία ενός container που περιέχει πραγματικούς αριθμούς και έχει όνομα v. Μπορούμε να το κάνουμε με τον κώδικα

```
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;
}
```

Για την ακρίβεια, ο compiler μεταφράζει εσωτερικά κάθε `range for` στην αντίστοιχη έκφραση με iterators.

Παράδειγμα

Έστω ότι θέλουμε να τυπώσουμε στην οθόνη τα στοιχεία ενός container τύπου `std::vector<double>` με όνομα v. Μπορούμε να το κάνουμε με τον κώδικα

```
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 `const_iterator` σε `std::vector<double>`. Δεν χρειαζόμαστε (και, για ασφάλεια, με τη συγκεκριμένη δήλωση δεν επιτρέπουμε) την τροποποίηση των στοιχείων που «δείχνει» ο συγκεκριμένος iterator.

10.4 Κατηγορίες

Οι βασικές κατηγορίες iterators παρατίθενται παρακάτω. Όπως θα εξηγήσουμε στο §13.3, οι ιεραρχίες {input, forward, bidirectional, random} και {output, mutable forward, mutable bidirectional, mutable random} των τύπων για τους iterators, στην οποία κάθε τύπος έχει όλες τις ιδιότητες του προηγούμενου, συμπεριφέρεται σαν αυτόν, και έχει κάποιες επιπλέον ιδιότητες, είναι παραδείγματα *κληρονομικότητας*.

10.4.1 Input iterators

Οι iterators εισόδου μπορούν να χρησιμοποιηθούν για να *διαβάσουμε* την τιμή στη θέση που «δείχνουν», και μάλιστα *μόνο μία φορά*. Επομένως, αν ο it είναι τέτοιος iterator, δεν επιτρέπεται η χρήση του `*it` στο αριστερό μέλος εντολής εκχώρησης. Επιτρέπονται

- Η δράση του τελεστή '`->`' για ανάγνωση μέλους στοιχείου.
- η μετακίνηση του iterator μόνο κατά μία θέση και μόνο προς τα εμπρός, με την εντολή `++it` ή την `it++`,
- η σύγκριση για ισότητα ή μη, με iterator που «δείχνει» σε μία θέση μετά το τέλος μιας ακολουθίας στοιχείων. Η δυνατότητα σύγκρισης μπορεί να μην ορίζεται για iterators σε άλλες θέσεις.

Η δυνατότητα ενός τέτοιου iterator να διαβάσει τιμή μόνο μία φορά σημαίνει ότι ένας iterator εισόδου και ένα αντίγραφό του, αν μετακινηθούν προς τα εμπρός, είναι πιθανό να δείχνουν σε διαφορετικές θέσεις. Επομένως, αν p είναι ένας iterator εισόδου, στον κώδικα

```
auto q = p;
++p;
++q;
bool eq{p==q};
```

το eq δεν είναι απαραιτήτως `true`, ίσως και να μην ορίζεται. Γι' αυτό το λόγο, iterators εισόδου μπορούν να χρησιμοποιηθούν για να διατρέξουμε μόνο μία φορά ένα διάστημα· ένας τέτοιος iterator δεν μπορεί να περάσει από το ίδιο σημείο δύο φορές. Κάθε ανάγνωση τιμής πρέπει να ακολουθείται από μετατόπιση (και όχι νέα ανάγνωση τιμής).

Iterators εισόδου μπορούν να συνδεθούν με ροές εισόδου (`istream_iterators` (§10.9.2)).

10.4.2 Output iterators

Οι iterators εξόδου μπορούν να χρησιμοποιηθούν μόνο για να δώσουμε τιμή στη θέση που «δείχνουν» και όχι για να «διαβάσουν» το στοιχείο. Δηλαδή, αν it είναι τέτοιος iterator, επιτρέπεται η εντολή `*it = ...` και δεν υποστηρίζεται άλλη χρήση του `*it`. Επίσης, επιτρέπεται μόνο η μετακίνηση του iterator κατά μία θέση μετά, με την εντολή `++it` ή την `it++`, και μάλιστα πρέπει να γίνεται μετά από κάθε εκχώρηση τιμής. Αυτό σημαίνει ότι ένας iterator εξόδου μπορεί να χρησιμοποιηθεί προς μία κατεύθυνση (εμπρός).

Iterators εξόδου μπορούν να χρησιμοποιηθούν για την προσπέλαση ροής εξόδου (`ostream_iterators` (§10.9.2)).

10.4.3 Forward iterators

Οι iterators μονής κατεύθυνσης (από την αρχή προς το τέλος του container ή της ακολουθίας εισόδου)

- δίνουν πρόσβαση για ανάγνωση τιμής στη θέση που δείχνουν, με τη δράση του τελεστή '`*`' ή σε μέλος του στοιχείου με τον τελεστή '`->`',
- μπορούν να μετακινηθούν κατά μία θέση, μόνο προς τα εμπρός,
- μπορούν να συγκριθούν, μόνο για ισότητα ή μη, με άλλο iterator αυτής της κατηγορίας.

Συνεπώς, οι iterators μονής κατεύθυνσης έχουν τουλάχιστον όλες τις ιδιότητες των iterators εισόδου και μπορούν να συμπεριφερθούν ως τέτοιοι. Επιπλέον, μπορούν να χρησιμοποιηθούν για να διατρέξουν ένα διάστημα πολλές φορές.

Οι const_iterators για τους containers

- `std::forward_list<>`,
- `std::unordered_set<>`,
- `std::unordered_multiset<>`,
- `std::unordered_map<>`,
- `std::unordered_multimap<>`,

είναι αυτού του είδους.

Mutable forward iterator

Ένας iterator αυτής της κατηγορίας που είναι και εξόδου, δηλαδή μπορεί να γράψει στη θέση που δείχνει, είναι *mutable forward iterator* (τροποποιήσιμος iterator μονής κατεύθυνσης). Τέτοιοι είναι οι απλοί iterators των containers που αναφέρθηκαν πιο πάνω.

10.4.4 Bidirectional iterators

Οι iterators δύο κατευθύνσεων έχουν όλες τις ιδιότητες των iterators μονής κατεύθυνσης και μπορούν να συμπεριφερθούν ως τέτοιοι. Επιπλέον, επιτρέπεται να μετακινηθούν κατά μία θέση προς τα πίσω, με τον τελεστή '`--`'.

Οι const_iterators για τους containers

- `std::list<>`,
- `std::set<>`,
- `std::multiset<>`,

- `std::map<>`,
- `std::multimap<>`,

είναι αυτού του είδους.

Mutable bidirectional iterator

Ένας iterator αυτής της κατηγορίας που είναι και εξόδου, δηλαδή μπορεί να γράψει στη θέση που δείχνει, είναι *mutable bidirectional iterator*. Τέτοιοι είναι οι απλοί iterators των containers που αναφέρθηκαν στην προηγούμενη παράγραφο.

10.4.5 Random iterators

Οι iterators τυχαίας προσπέλασης έχουν όλες τις ιδιότητες των iterators δύο κατευθύνσεων και μπορούν να συμπεριφερθούν ως τέτοιοι. Επιπλέον, όπως αναφέραμε στο §10.3, μπορούμε

- να τους προσθέσουμε ή αφαιρέσουμε ένα ακέραιο αριθμό και να παραγάγουμε έτσι νέο iterator, μετατοπισμένο σε επόμενη ή προηγούμενη θέση,
- να μετακινήσουμε τους ίδιους με τους σύνθετους τελεστές '`+=`' και '`-=`',
- να αφαιρέσουμε ένα iterator τυχαίας προσπέλασης από άλλον ώστε να υπολογίσουμε την απόστασή τους,
- να τους συμπληρώσουμε με ακέραιο δείκτη εντός αγκυλών· αν `it` είναι τέτοιος iterator, η έκφραση `it[n]` ισοδυναμεί με `*(it+n)`,
- να τους συγκρίνουμε με όλους τους τελεστές σύγκρισης.

Σε αυτή την κατηγορία ανήκουν

- οι const_iterators των containers
  - `std::array<>`,
  - `std::vector<>`,
  - `std::deque<>`,
- οι const_iterators του `std::string` (που ενώ δεν είναι container συμπεριφέρεται ως τέτοιος σε κάποιες περιπτώσεις),
- οι δείκτες σε σταθερό στοιχείο (`T const *`, όπου `T` ο τύπος των στοιχείων) σε ενσωματωμένο διάνυσμα (§10.8).

Mutable random iterator

Ένας iterator αυτής της κατηγορίας που είναι και εξόδου, δηλαδή μπορεί να γράψει στη θέση που δείχνει, λέγεται *mutable random iterator*.

10.5 Βοηθητικές συναρτήσεις και κλάσεις

Για να μπορούμε να γράφουμε κώδικα γενικό, που να ισχύει για διάφορες κατηγορίες iterators, η Standard Library παρέχει με το `<iterator>` στο χώρο ονομάτων std, κάποιες βοηθητικές συναρτήσεις και κλάσεις.

10.5.1 `advance()`

Η συνάρτηση

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

δέχεται έναν iterator εισόδου, it, και μια ποσότητα ακέραιου τύπου, n. Αν ο it είναι στην πραγματικότητα

τυχαίας προσπέλασης, η κλήση της ισοδυναμεί με `it += n;`.

δύο κατευθύνσεων, η κλήση της ισοδυναμεί με n διαδοχικές κλήσεις της εντολής `++it` (αν $n > 0$) ή `--it` (αν $n < 0$).

μονής κατεύθυνσης ή εισόδου, η κλήση της έχει νόημα μόνο αν το n δεν είναι αρνητικό και ισοδυναμεί με n διαδοχικές κλήσεις του `++it`.

10.5.2 `next()`

Η συνάρτηση

```
template<typename ForwardIterator>
ForwardIterator
next(ForwardIterator it, Dist n = 1);
```

ουσιαστικά καλεί την `std::advance()` και επιστρέφει iterator, η θέσεις μετά τον it (χωρίς να τροποποιεί τον it). Αν το n είναι αρνητικό, ο iterator που επιστρέφεται είναι πριν τον it, ο οποίος πρέπει να είναι δύο κατευθύνσεων. Αν δεν προσδιοριστεί τιμή για το n, αυτό παίρνει την τιμή 1.

Προσέξτε ότι ο επιστρεφόμενος iterator μπορεί να «δείχνει» έξω από τα όρια του container (οπότε δεν μπορεί να χρησιμοποιηθεί για προσπέλαση στοιχείου).

Ο τύπος Dist είναι ακέραιος[2].

10.5.3 `prev()`

Η συνάρτηση

[2] `typename std::iterator_traits<ForwardIterator>::difference_type`

```
template<typename BidirectionalIterator>
BidirectionalIterator
prev(BidirectionalIterator it, Dist n = 1);
```

ουσιαστικά καλεί την `std::advance()` και επιστρέφει iterator, n θέσεις πριν τον it (χωρίς να τροποποιεί τον it). Αν το n είναι αρνητικό, ο iterator που επιστρέφεται είναι μετά τον it. Αν δεν προσδιοριστεί τιμή για το n, αυτό παίρνει την τιμή 1.

Προσέξτε ότι ο επιστρεφόμενος iterator μπορεί να «δείχνει» έξω από τα όρια του container.

Ο τύπος Dist είναι ακέραιος[3].

10.5.4 `distance()`

Η συνάρτηση

```
template<typename InputIterator>
Dist
distance(InputIterator it1, InputIterator it2);
```

δέχεται δύο iterators it1 και it2, ίδιου τύπου, που «δείχνουν» στον ίδιο container. Αν οι iterators είναι τυχαίας προσπέλασης, η συνάρτηση επιστρέφει το `it2-it1` ενώ σε άλλη περίπτωση αυξάνει τον it1 έως ότου γίνει ίσος με it2 και επιστρέφει το πλήθος των αυξήσεων. Προφανώς, πρέπει στην τελευταία περίπτωση ο it1 να μη «δείχνει» μετά τον it2.

Ο τύπος Dist είναι ακέραιος[4].

10.5.5 `iterator_traits<>`

Συχνά χρειάζεται να γράψουμε συνάρτηση template με παράμετρο τον τύπο ενός iterator. Ο τύπος που θα περάσει ως παράμετρος όταν θα κληθεί η συνάρτηση, μεταφέρει πληροφορίες, μεταξύ άλλων, σχετικά με την κατηγορία στην οποία ανήκει, τον τύπο των στοιχείων στα οποία δείχνει και τον κατάλληλο τύπο για «αποστάσεις» (διαφορές δύο iterators). Μπορούμε να εξαγάγουμε αυτή την πληροφορία με τη βοήθεια της κλάσης `iterator_traits<>` ως εξής: αν T είναι ο τύπος του iterator,

- η κατηγορία στην οποία ανήκει ο iterator είναι η

  `typename std::iterator_traits<T>::iterator_category`

  Αυτός ο τύπος είναι ένας από τους

  - `std::output_iterator_tag`,

[3] `typename std::iterator_traits<BidirectionalIterator>::difference_type`

[4] `typename std::iterator_traits<InputIterator>::difference_type`

  - `std::input_iterator_tag`,
  - `std::forward_iterator_tag`,
  - `std::bidirectional_iterator_tag`,
  - `std::random_access_iterator_tag`.

  Θα δούμε παρακάτω παράδειγμα χρήσης του.

- ο τύπος των στοιχείων στα οποία δείχνει είναι

  ```
  typename std::iterator_traits<T>::value_type
  ```

- ο τύπος για αποστάσεις είναι

  ```
  typename std::iterator_traits<T>::difference_type
  ```

10.6 Παράδειγμα

Έστω ότι θέλουμε να γράψουμε μια συνάρτηση που θα αντιγράφει το πρώτο, τρίτο, πέμπτο κλπ. στοιχείο σε ένα διάστημα κάποιου container, σε διαδοχικές θέσεις κάποιου άλλου. Ας την ονομάσουμε copyodd. Τα διαστήματα θα προσδιορίζονται από iterators.

Αν επιθυμούμε να αντιγράψουμε κάθε δεύτερο στοιχείο του container a στον container b, μετά την όγδοη θέση του, θα πρέπει να μπορούμε να γράψουμε

```
copyodd(a.cbegin(), a.cend(), std::next(b.begin(), 8));
```

Ας γράψουμε τη συνάρτηση. Θα δέχεται τρεις iterators:

- οι δύο πρώτοι θα ορίζουν ένα διάστημα στον αρχικό container: η θέση που δείχνει ο πρώτος θεωρείται ως αρχή ενώ ο δεύτερος δείχνει σε μία θέση μετά το τέλος του διαστήματος που μας ενδιαφέρει. Οι iterators αυτοί δεν θα μπορούν να τροποποιήσουν τα στοιχεία του container στον οποίο δείχνουν. Ας τους ονομάσουμε beg1, end1.

- ο τρίτος iterator, έστω beg2, θα δείχνει σε άλλο container, σε θέση που θα θεωρείται ως η αρχή. Στον δεύτερο container θα γίνεται η αντιγραφή των στοιχείων. Δεν χρειάζεται να ορίσουμε το τέλος του διαστήματος· ξέρουμε από τους δύο πρώτους iterators ακριβώς πόσα στοιχεία θα αντιγράψουμε. Θεωρούμε βέβαια ότι επαρκούν οι θέσεις που ακολουθούν το beg2 για όσες τιμές θα αντιγράψουμε.

Καθώς η συνάρτηση πρέπει να εφαρμόζεται για iterators δύο πιθανώς διαφορετικών containers, πρέπει να γραφεί ως template με δύο παραμέτρους για τους τύπους των iterators: οι δύο πρώτοι iterators θα έχουν κοινό τύπο και ο τρίτος κάποιον άλλο. Η συνάρτηση δεν χρειάζεται να επιστρέφει τίποτε.

Μια πρώτη απόπειρα να γράψουμε τη συνάρτηση είναι

```
template<typename Iterator1, typename Iterator2>
void
copyodd(Iterator1 beg1, Iterator1 end1, Iterator2 beg2)
{
  while (beg1 < end1) {
    *beg2 = *beg1;
    if (beg1 == end1 - 1) {
      break;
    }
    beg1+=2;
    ++beg2;
  }
}
```

Παρατηρήστε ότι δεν ορίζεται iterator *μετά* το `end()` ενός container οπότε πρέπει να μην μετακινήσουμε τον beg1 σε τέτοια θέση. Προσέξτε ότι η επιλογή της μετακίνησης του iterator beg1 με τον τελεστή '+=', η σύγκρισή του με τον end1 χρησιμοποιώντας τον τελεστή '<' και η αφαίρεση ακέραιου από τον end1 μας υποχρεώνει να θεωρούμε ότι ο τύπος των beg1, end1 είναι iterator τυχαίας προσπέλασης.

Ο beg2 απαιτείται να είναι τουλάχιστον iterator εξόδου, καθώς χρησιμοποιούμε μόνο την εκχώρηση τιμής μέσω αυτού και, αμέσως μετά, την προώθησή του κατά μία θέση.

Ένας άλλος τρόπος να γράψουμε τη συνάρτηση είναι ο εξής

```
#include <iterator>

template<typename Iterator1, typename Iterator2>
void
copyodd(Iterator1 beg1, Iterator1 end1, Iterator2 beg2)
{
  while (beg1 != end1) {
    *beg2 = *beg1;
    if (std::next(beg1) == end1) {
      break;
    }
    std::advance(beg1, 2);
    ++beg2;
  }
}
```

Παρατηρήστε ότι η προώθηση του iterator beg1 κατά δύο θέσεις γίνεται χρησιμοποιώντας τη συνάρτηση `std::advance()`. Επίσης, ο έλεγχος αν ο ίδιος iterator είναι εντός του διαστήματος γίνεται `beg1 != end1`. Με αυτές τις επιλογές αρκεί να είναι iterator εισόδου ο beg1.

Προσέξτε ότι λόγω της συγκεκριμένης σύγκρισης του beg1 με το τέλος του δια-

στήματος, πρέπει να ελέγχουμε μήπως αυτός ξεπεράσει το τέλος, πριν τον μετατοπίσουμε κατά δύο θέσεις. Αν είναι μία θέση πριν το τέλος πρέπει να διακόψουμε την επανάληψη. Έχουμε δύο δυνατότητες για να το κάνουμε αυτό:

- να ελέγχουμε αν `beg1 == std::prev(end1)`.
- να ελέγχουμε αν `std::next(beg1) == end1`.

Η χρήση της `std::prev()` απαιτεί το end1 (άρα και το beg1) να είναι iterator δύο κατευθύνσεων. Επιλέχθηκε η `std::next()` καθώς αρκείται σε iterator μονής κατεύθυνσης.

Συνοψίζοντας, η συνάρτηση στην τωρινή της εκδοχή απαιτεί ο τύπος Iterator1 να είναι iterator μονής κατεύθυνσης τουλάχιστον, και ο τύπος Iterator2 να είναι iterator εξόδου ή επόμενος στην ιεραρχία.

Εναλλακτικά, μπορούμε να γράψουμε τη συνάρτηση ως εξής

```
template<typename Iterator1, typename Iterator2>
void
copyodd(Iterator1 beg1, Iterator1 end1, Iterator2 beg2)
{
  while (beg1 != end1) {
    *beg2 = *beg1;
    ++beg1;
    if (beg1 == end1) {
      break;
    }
    ++beg1;
    ++beg2;
  }
}
```

Με τη νέα μορφή, η απαίτηση για τον τύπο των beg1, end1 έχει χαλαρώσει: αρκεί να είναι iterator εισόδου.

10.7 Επιλογή συνάρτησης με βάση την κατηγορία iterator

Οι συγκεκριμένες εκδοχές της συνάρτησης που παρουσιάστηκαν στο παράδειγμα τυχαίνει να μην διαφέρουν σε ταχύτητα εκτέλεσης ή σε απαιτήσεις μνήμης. Γενικά όμως, οι διάφορες κατηγορίες iterators επιβάλλουν ή επιτρέπουν τη χρήση αλγόριθμων με διαφορετικά χαρακτηριστικά και επιδόσεις. Ανάλογα με την κατηγορία των iterators που θα χρησιμοποιήσουμε κατά την κλήση της συνάρτησης, θέλουμε να εκμεταλλευόμαστε την αντίστοιχη εκδοχή της. Αυτό γίνεται ως εξής:

Ως πρώτο βήμα, συμπληρώνουμε τα ορίσματα της συνάρτησης σε κάθε εκδοχή με ένα ακόμα:

- η εκδοχή της συνάρτησης για iterators τυχαίας προσπέλασης θα δέχεται μια ποσότητα τύπου `std::random_access_iterator_tag`,
- η εκδοχή της συνάρτησης για iterators μονής κατεύθυνσης θα δέχεται μια ποσότητα τύπου `std::forward_iterator_tag`,
- η βασική εκδοχή της συνάρτησης για iterators εισόδου θα δέχεται μια ποσότητα τύπου `std::input_iterator_tag`.

Παραθέτουμε όλες τις εκδοχές με την τροποποίηση αυτή παρακάτω:

```
#include <iterator>

template<typename Iterator1, typename Iterator2>
void
copyodd(Iterator1 beg1, Iterator1 end1, Iterator2 beg2,
        std::random_access_iterator_tag name)
{
  while (beg1 < end1) {
    *beg2 = *beg1;
    if (beg1 == end1 - 1) {
      break;
    }
    beg1+=2;
    ++beg2;
  }
}

template<typename Iterator1, typename Iterator2>
void
copyodd(Iterator1 beg1, Iterator1 end1, Iterator2 beg2,
        std::forward_iterator_tag name)
{
  while (beg1 != end1) {
    *beg2 = *beg1;
    if (std::next(beg1) == end1) {
      break;
    }
    std::advance(beg1, 2);
    ++beg2;
  }
}

template<typename Iterator1, typename Iterator2>
void
```

```
copyodd(Iterator1 beg1, Iterator1 end1, Iterator2 beg2,
        std::input_iterator_tag name)
{
  while (beg1 != end1) {
    *beg2 = *beg1;
    ++beg1;
    if (beg1 == end1) {
      break;
    }
    ++beg1;
    ++beg2;
  }
}
```

Οι παραπάνω συναρτήσεις είναι εξειδικεύσεις (§7.11.1) ενός γενικότερου template για την copyodd (που δεν έχει γραφεί καθώς δεν θα χρησιμοποιηθεί ποτέ). Όπως αναφέραμε στο §7.2, μπορούμε να παραλείψουμε το όνομα του επιπλέον ορίσματος, καθώς αυτό δεν χρησιμοποιείται στο σώμα των συναρτήσεων.

Ως δεύτερο βήμα, συμπληρώνουμε τον κώδικα με συνάρτηση που έχει ως ορίσματα τους τρεις iterators. Καθώς διαφέρει στο πλήθος των ορισμάτων κρατά το ίδιο όνομα με το template που γράψαμε. Σε αυτή τη συνάρτηση γίνεται η επιλογή της κατάλληλης εξειδίκευσης:

```
#include <iterator>

template<typename Iterator1, typename Iterator2>
void
copyodd(Iterator1 beg1, Iterator1 end1, Iterator2 beg2)
{
  typename std::iterator_traits<Iterator1>::iterator_category a;
  return copyodd(beg1,end1,beg2,a);
}
```

Πλέον, η κλήση της `copyodd()` με τρεις iterators επιλέγει την καταλληλότερη εκδοχή[5], που θεωρητικά είναι βελτιστοποιημένη για συγκεκριμένη κατηγορία iterator.

10.8 Iterator σε ενσωματωμένο διάνυσμα

Ας αναφερθεί εδώ ότι για τα ενσωματωμένα διανύσματα—παρόλο που δεν έχουν όλα τα χαρακτηριστικά των containers—μπορούμε να χρησιμοποιήσουμε τους συνήθεις δείκτες για να ορίσουμε «διαστήματα» iterators, όπου χρειάζονται τέτοια.

[5]Μπορείτε να το ελέγξετε αν κάθε εκδοχή τυπώνει κατάλληλο μήνυμα στην οθόνη κατά την κλήση της.

Έστω ότι έχουμε ένα ενσωματωμένο διάνυσμα 5 στοιχείων με όνομα a. Θυμηθείτε (§2.18) ότι το όνομα ενός διανύσματος είναι και δείκτης στο πρώτο στοιχείο του ενώ η πρόσθεση ενός ακεραίου n σε αυτό το όνομα μας μεταφέρει n θέσεις μετά. Έτσι, το a είναι δείκτης στο `a[0]` ενώ το `a+5` δείχνει *σε μία θέση μετά* το τελευταίο στοιχείο (που είναι το `a[4]`).

Οι δείκτες a, `a+5` μπορούν να χρησιμοποιηθούν όπου χρειάζονται δύο iterators που να «δείχνουν» στην αρχή και σε μία θέση μετά το τέλος στο ενσωματωμένο διάνυσμα a. Οι συγκεκριμένοι iterators είναι τυχαίας προσπέλασης. Εναλλακτικά, για την παραγωγή των συγκεκριμένων iterators μπορούμε να χρησιμοποιήσουμε τις συναρτήσεις `std::begin()` και `std::end()` με όρισμα το όνομα του διανύσματος. Αυτές παρέχονται από το header `<iterator>` (και από πολλούς άλλους). Είναι προτιμότερη η χρήση τους έναντι των δεικτών αρχής και τέλους.

10.9 Προσαρμογείς για iterators

10.9.1 Ανάστροφοι iterators

Υπάρχει περίπτωση να χρειαστούμε ένα iterator που να μπορεί να διατρέξει ένα container *ανάστροφα*, από το τελευταίο στοιχείο προς το πρώτο, με την ίδια ευχρηστία που έχουν οι iterators «ορθής φοράς» που έχουμε αναφέρει. Οι ανάστροφοι iterators προκύπτουν με προσαρμογή των απλών iterators και, όπως αυτοί, διακρίνονται ανάλογα με το αν επιτρέπουν ή όχι την τροποποίηση των στοιχείων στα οποία «δείχνουν». Οι τύποι τέτοιων iterators, π.χ. για ένα `std::vector<double>`, είναι

```
std::vector<double>::reverse_iterator
```

και

```
std::vector<double>::const_reverse_iterator
```

αντίστοιχα.

Κάθε container που έχει iterators διπλής κατεύθυνσης ή πιο γενικούς παρέχει και ανάστροφους iterators. Τέτοιοι containers παρέχουν συναρτήσεις–μέλη για τον προσδιορισμό αρχικού και τελικού ανάστροφου iterator:

- Οι συναρτήσεις–μέλη `rbegin()` και `rend()` επιστρέφουν ανάστροφο iterator (`reverse_iterator`) στο *τελευταίο* στοιχείο και *σε μία θέση πριν το πρώτο στοιχείο* αντίστοιχα.
- Οι `crbegin()` και `crend()` επιστρέφουν ανάστροφο iterator σε σταθερό στοιχείο (`const_reverse_iterator`) στο *τελευταίο* στοιχείο και *σε μία θέση πριν το πρώτο στοιχείο* αντίστοιχα.

Προσέξτε ότι οι τελεστές '`++`', '`--`', όταν δρουν σε ανάστροφο iterator (σε σταθερό στοιχείο ή όχι), τον μετακινούν σε μία θέση *προς την αρχή* ή *προς το τέλος* του container αντίστοιχα.

Παράδειγμα

Έστω ότι θέλουμε να τυπώσουμε στην οθόνη τα στοιχεία ενός container τύπου `std::vector<double>` με όνομα v, με *αντίστροφη σειρά*. Μπορούμε να το κάνουμε με τον κώδικα

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

Ο it είναι τύπου `std::vector<double>::` `const_reverse_iterator` .

10.9.2 Iterators ροής

Μια ροή εισόδου που έχει συνδεθεί με αρχείο ή με το πληκτρολόγιο (δηλαδή με το standard input, `std::cin`, του προγράμματος) μπορεί να προσαρμοστεί σε iterator εισόδου ώστε να μπορούμε να τη χειριστούμε όπως κάθε ακολουθία τιμών, π.χ. με ένα αλγόριθμο της Standard Library. Καθώς οποιοσδήποτε iterator «δείχνει» σε στοιχεία συγκεκριμένου τύπου θα πρέπει η ροή να περιέχει τον ίδιο τύπο τιμών.

Ένας iterator εισόδου που συνδέεται με το `std::cin` και μπορεί να διαβάσει ακέραιες τιμές από αυτό, παράγεται ως εξής:

```
std::istream_iterator<int> a{std::cin};
```

Ο a είναι iterator εισόδου, στην «αρχή» του `std::cin`. Η μετακίνησή του στην επόμενη θέση[6] γίνεται με τον τελεστή '`++`' και η απόδοση τιμής με τον τελεστή '`*`'.

Όπως θα δούμε στο Κεφάλαιο 12, σε ένα αλγόριθμο της Standard Library χρειαζόμαστε iterator αρχής και τέλους για το χειρισμό μιας ακολουθίας τιμών. Ο iterator που αντιπροσωπεύει το «τέλος» (επόμενη θέση από την τελευταία τιμή) οποιασδήποτε ροής εισόδου ακεραίων τιμών δηλώνεται ως εξής:

```
std::istream_iterator<int> b{};
```

Παράδειγμα

Έστω ότι θέλουμε να βρούμε το άθροισμα των πραγματικών αριθμών που περιέχονται στο αρχείο με όνομα *data*. Μπορούμε να γράψουμε τον κώδικα

```
std::ifstream in{"data"};
std::istream_iterator<double> beg{in}, end{};
double s{0.0};
for (auto it = beg; it != end; ++it) {
  s += *it;
}
```

[6] και η ταυτόχρονη ανάγνωση τιμής από τη ροή

Σε αντιστοιχία με τους iterators σε ροή εισόδου, μπορούμε να ορίσουμε iterators συνδεδεμένους με αρχείο ή το `std::cout`. Έτσι, αν το αντικείμενο out έχει συνδεθεί με αρχείο πραγματικών αριθμών, η δήλωση

```
std::ostream_iterator<double> beg{out, "␣"};
```

δημιουργεί ένα iterator εξόδου με όνομα beg που μπορεί να γράψει πραγματικές τιμές στη ροή out. Οποιαδήποτε εκχώρηση τιμής στο `*beg` (ή και στο ίδιο το beg) μεταφέρει την τιμή στο αρχείο. Ακολούθως εκτυπώνει τη σειρά χαρακτήρων (δηλαδή, χαρακτήρες εντός διπλών εισαγωγικών) που προσδιορίσαμε κατά τη δημιουργία του beg, και μετακινείται στην επόμενη θέση. Η δράση του τελεστή '++' σε iterator εξόδου είναι επιτρεπτή αλλά δεν προκαλεί μετακίνηση· στην πραγματικότητα δεν κάνει τίποτε.

Μπορούμε να παραλείψουμε να προσδιορίσουμε κατά τη δημιουργία του iterator τη σειρά χαρακτήρων που θα «συμπληρώνει» τις τιμές που θα τυπώνονται. Σε τέτοια περίπτωση δεν θα διαχωρίζονται διαδοχικές εκτυπώσεις.

Οι προσαρμογείς `std::istream_iterator<>` και `std::ostream_iterator<>` παρέχονται από το header `<iterator>`.

10.9.3 Iterators εισαγωγής

Αναφέραμε ότι η δράση του τελεστή '`*`' σε ένα iterator a δίνει πρόσβαση στη θέση που δείχνει αυτός. Αν αυτός είναι iterator εξόδου ή επόμενος στην ιεραρχία, μπορούμε να αλλάξουμε την τιμή της θέσης με εκχώρηση της νέας τιμής στο `*a`. Υπάρχει περίπτωση, ειδικά κατά τη χρήση των αλγόριθμων που θα δούμε στο Κεφάλαιο 12, να επιθυμούμε *εισαγωγή νέου στοιχείου* και όχι τροποποίηση υπάρχοντος κατά την εκχώρηση τιμής μέσω iterator. Γι' αυτό το σκοπό η Standard Library παρέχει στο `<iterator>` το `std::insert_iterator<>`, ένα υπόδειγμα iterator εξόδου που συνδέεται με συγκεκριμένο container και συγκεκριμένη θέση σε αυτόν. Η εκχώρηση τιμής μέσω αυτού του iterator προκαλεί εισαγωγή στοιχείου στον container. Η δήλωση και η χρήση του φαίνεται στον ακόλουθο κώδικα:

```
std::vector<int> v; // empty vector
v.reserve(10);

std::insert_iterator<std::vector<int>> iend{v,v.end()};
*iend = 1;  // v is {1}
++iend;
*iend = 2;  // v is {1, 2}
++iend;
*iend = 3;  // v is {1, 2, 3}
++iend;

std::insert_iterator<std::vector<int>> ibeg{v,v.begin()};
*ibeg = -1; // v is {-1, 1, 2, 3}
```

```
++ibeg;
*ibeg = -2; // v is {-1, -2, 1, 2, 3}
++ibeg;
```

Παρατηρήστε ότι η πρώτη εισαγωγή στοιχείου με τη χρήση του ibeg γίνεται στην αρχή του v. Η εισαγωγή του επόμενου στοιχείου μέσω του ibeg γίνεται στην αμέσως επόμενη θέση και όχι πριν την αρχή, όπως θα ανέμενε κανείς. Το ibeg έχει συνδεθεί με συγκεκριμένη θέση μνήμης (αυτή που έδειχνε το `v.begin()` κατά τον ορισμό του ibeg) και έχει προχωρήσει με την πρώτη χρήση του στην επόμενη θέση από αυτή.

Όπως θα αναφέρουμε στο Κεφάλαιο 11, η εισαγωγή στοιχείου σε container μπορεί να προκαλέσει τη μετακίνηση κάποιων ή όλων των στοιχείων του. Σε τέτοια περίπτωση, οι iterators στα στοιχεία ακυρώνονται, παύουν να δείχνουν στις θέσεις με τις οποίες είχαν συνδεθεί και συνεπώς δεν επιτρέπεται να χρησιμοποιηθούν. Στο συγκεκριμένο παράδειγμα έγινε χρήση της συνάρτησης–μέλους `reserve()` η οποία δεσμεύει κάποιες συνεχόμενες θέσεις μνήμης για την επέκταση του v. Με αυτό τον τρόπο εξασφαλίσαμε ότι οι εισαγωγές των λίγων στοιχείων του παραδείγματος δεν θα προκαλέσουν μετακίνηση του vector σε άλλο τμήμα μνήμης.

Η συνάρτηση `std::inserter()` του `<iterator>` παρέχει ένα εύκολο τρόπο για να δημιουργήσουμε ένα insert_iterator ώστε να αναφέρεται σε συγκεκριμένη θέση ενός container. Δέχεται ως ορίσματα ένα container και ένα iterator σε αυτόν και επιστρέφει ένα κατάλληλο insert_iterator:

```
std::vector<int> v;
auto it = std::inserter(v,v.begin());

std::vector<int> c;
it = std::inserter(c, c.end());
```

Ο `std::insert_iterator<>` και η συνάρτηση `std::inserter()` μπορούν να χρησιμοποιηθούν για όλους τους containers που παρέχουν τη συνάρτηση–μέλος `insert()`. Όπως θα δούμε στο Κεφάλαιο 11, η `insert()` παρέχεται από όλους του containers εκτός από το `std::array<>` και το `std::forward_list<>`.

Η Standard Library παρέχει δύο ακόμα insert_iterators με τις βοηθητικές συναρτήσεις τους στο `<iterator>`:

- Ο `std::back_insert_iterator<>` δηλώνεται όπως στον παρακάτω κώδικα

  ```
  std::vector<int> v;
  std::back_insert_iterator<std::vector<int>> b{v};
  ```

  Η εκχώρηση τιμής μέσω αυτού, εισάγει νέο στοιχείο στο τέλος του container στον οποίο αναφέρεται, με την κλήση της συνάρτησης–μέλους `push_back()` του container. Μπορεί να χρησιμοποιηθεί για όσους containers παρέχουν τέτοια συνάρτηση–μέλος, δηλαδή τους `std::vector<>`, `std::deque<>` και `std::list<>`.

Η συνάρτηση `std::back_inserter()` με όρισμα ένα από τους παραπάνω containers δημιουργεί ένα κατάλληλο back_insert_iterator:

```
std::vector<int> v;
auto it = std::back_inserter(v);
```

- Ο `std::front_insert_iterator<>` δηλώνεται όπως στον παρακάτω κώδικα

```
std::list<int> c;
std::front_insert_iterator<std::list<int>> b{c};
```

Η εκχώρηση τιμής μέσω αυτού εισάγει νέο στοιχείο στην αρχή του container στον οποίο αναφέρεται, με την κλήση της συνάρτησης–μέλους `push_front()` του container. Μπορεί να χρησιμοποιηθεί για όσους containers παρέχουν τη συγκεκριμένη συνάρτηση–μέλος, δηλαδή τους `std::deque<>`, `std::list<>` και `std::forward_list<>`.

Η συνάρτηση `std::front_inserter()` με όρισμα ένα από τους παραπάνω containers δημιουργεί ένα κατάλληλο front_insert_iterator:

```
std::list<int> c;
auto it = std::front_inserter(c);
```

Οι iterators εισαγωγής παρουσιάζουν ιδιαίτερη χρησιμότητα στους αλγόριθμους που παρέχει η Standard Library (Κεφάλαιο 12).

10.9.4 Iterators μετακίνησης

Ο `std::move_iterator` του `<iterator>` είναι προσαρμογέας για οποιοδήποτε iterator εισόδου ή επόμενου στην ιεραρχία. Συμπεριφέρεται ακριβώς όπως ο iterator που προσαρμόζει με μόνη διαφορά ότι η ανάγνωση τιμής από αυτόν *μετακινεί* (και δεν αντιγράφει) την τιμή στην οποία δείχνει.

Στον παρακάτω κώδικα

```
#include <vector>
#include <iterator>
#include <iostream>
int
main()
{
    using cntr = std::vector<std::vector<int>>;
    cntr a(5, {1,2,3});
    cntr b(5);
    std::cout << "␣sizes␣of␣a␣before:␣";
    for (auto & x: a) {
        std::cout << x.size() << '␣';
    }
```

```
    std::cout << "\b\n";
    std::cout << "␣sizes␣of␣b␣before:␣";
    for (auto & x: b) {
        std::cout << x.size() << '␣';
    }
    std::cout << "\b\n";
    std::move_iterator<cntr::iterator> beg{a.begin()};
    std::move_iterator<cntr::iterator> end{a.end()};
    auto bit = b.begin();
    for (auto ait = beg; ait != end; ++ait) {
        *bit = *ait;
        ++bit;
    }
    std::cout << "␣sizes␣of␣a␣after:␣";
    for (auto & x: a) {
        std::cout << x.size() << '␣';
    }
    std::cout << "\b\n";
    std::cout << "␣sizes␣of␣b␣after:␣";
    for (auto & x: b) {
        std::cout << x.size() << '␣';
    }
    std::cout << "\b\n";
}
```

το διάνυσμα a έχει 5 θέσεις. Η κάθε μια αποτελείται από ένα διάνυσμα ακεραίων με αρχική εκχώρηση τριών αριθμών. Το ίδιο ισχύει και γα το διάνυσμα b μόνο που σε αυτό δεν έχουμε εκχώρηση τιμών. Τα επιβεβαιώνουμε με εκτύπωση του πλήθους των στοιχείων κάθε θέσης στα a,b. Κατόπιν, προσαρμόζουμε τους iterators αρχής και τέλους του a δημιουργώντας iterators μετακίνησης. Κατά την επακόλουθη εκχώρηση τιμών από το διάνυσμα a στο διάνυσμα b μέσω αυτών, γίνεται η μετακίνηση των στοιχείων του a στο b. Το επιβεβαιώνουμε με την εκτύπωση του πλήθους των στοιχείων κάθε θέσης στα διανύσματα a,b. Το a πλέον μπορεί μόνο να καταστραφεί ή να του εκχωρηθούν κατάλληλες τιμές με μετακίνηση σε αυτό.

Σχετική βοηθητική συνάρτηση είναι η `std::make_move_iterator()` που παρέχεται από το `<iterator>`. Αυτή δέχεται ένα iterator και επιστρέφει τον κατάλληλο `move_iterator`. Έτσι, στον προηγούμενο κώδικα, η δημιουργία των beg, end θα μπορούσε να γίνει με τις εντολές

```
auto beg = std::make_move_iterator(a.begin());
auto end = std::make_move_iterator(a.end());
```

Και οι iterators μετακίνησης παρουσιάζουν ιδιαίτερη χρησιμότητα στους αλγόριθμους που παρέχει η Standard Library (Κεφάλαιο 12).

10.10 Ασκήσεις

1. Γράψτε συνάρτηση που θα δέχεται δύο iterators beg, end και θα επιστρέφει iterator στο μικρότερο στοιχείο (ή στο τελευταίο από τα μικρότερα στοιχεία) στο διάστημα που ορίζουν αυτοί: από το beg έως μία θέση πριν το end. Οι iterators να είναι όσο πιο θεμελιώδους τύπου μπορείτε.

2. Γράψτε συνάρτηση που θα δέχεται δύο iterators μονής κατεύθυνσης, beg και end, και θα ελέγχει αν τα στοιχεία στο διάστημα που ορίζουν αυτοί (από το beg έως μία θέση πριν το end) είναι ταξινομημένα από το μικρότερο στο μεγαλύτερο. Θα επιστρέφει λογική τιμή.

3. Γράψτε συνάρτηση που θα δέχεται δύο iterators διπλής κατεύθυνσης, beg και end, και θα αναστρέφει τη σειρά των στοιχείων του διαστήματος [beg,end).

4. Γράψτε συνάρτηση που θα δέχεται τρεις iterators: οι δύο πρώτοι, beg1, end1 θα ορίζουν μια ακολουθία αριθμών και ο τρίτος, beg2, θα προσδιορίζει την αρχή άλλης ακολουθίας αριθμών. Η συνάρτηση να επιστρέφει το εσωτερικό γινόμενο των δύο ακολουθιών (δηλαδή το $\sum_i \alpha_i \beta_i$).

5. Γράψτε συνάρτηση που θα δέχεται δύο iterators διπλής κατεύθυνσης, beg και end, και θα ταξινομεί από το μικρότερο στο μεγαλύτερο τα στοιχεία του διαστήματος [beg,end). Να χρησιμοποιήσετε τον αλγόριθμο ταξινόμησης bubble sort (§B′.2.1).

6. Γράψτε συνάρτηση που να δέχεται τρεις iterators:

   - δύο ίδιου τύπου με ονόματα beg1, end1 που θα ορίζουν ένα διάστημα [beg1,end1) και
   - ένας τρίτος με όνομα beg2, πιθανώς διαφορετικού τύπου από τους beg1, end1. Αυτός θα δείχνει στην αρχή κάποιου διαστήματος.

   Η συνάρτηση να ελέγχει αν τα στοιχεία στο διάστημα [beg1,end1) είναι ίσα με τα στοιχεία στο διάστημα που αρχίζει από το beg2. Αν όλα είναι ίσα, να επιστρέφει `true`, αλλιώς να επιστρέφει `false`. Θεωρήστε ότι υπάρχουν μετά το beg2 τουλάχιστον όσα στοιχεία στο [beg1,end1).

7. Γράψτε συνάρτηση που να δέχεται τρεις iterators:

   - δύο ίδιου τύπου με ονόματα beg1, end1 που θα ορίζουν ένα διάστημα [beg1,end1) και
   - ένας τρίτος με όνομα beg2, πιθανώς διαφορετικού τύπου από τους beg1, end1. Αυτός θα δείχνει στην αρχή κάποιου διαστήματος.

   Η συνάρτηση θα αντιγράφει τα στοιχεία του διαστήματος [beg1,end1) στο διάστημα που ξεκινά με το beg2.

8. Υλοποιήστε τον αλγόριθμο quicksort (§Β΄.2.3), ώστε να ταξινομεί από το μικρότερο στο μεγαλύτερο τα στοιχεία μιας ακολουθίας μεταξύ δυο iterators beg, end.

9. Γράψτε συνάρτηση που θα δέχεται δύο iterators beg, end σε μία ακολουθία ακέραιων αριθμών. Να ελέγχει αν τα μη μηδενικά στοιχεία του διαστήματος [beg,end) εμφανίζονται το καθένα μία μόνο φορά.

10. Θέλουμε να γράψουμε συνάρτηση που να δέχεται

    - δύο iterators beg1, end1 που ορίζουν διάστημα σε ένα container,
    - ένα άλλο iterator beg2 σε άλλο container, πιθανώς διαφορετικού τύπου.

    Η συγκεκριμένη συνάρτηση θα αντιγράφει όλα τα στοιχεία του διαστήματος [beg1,end1) σε διαδοχικές θέσεις από το beg2 και μετά. Αν εντοπίσει ομάδες *ίσων και διαδοχικών* στοιχείων θα παραλείπει κατά την αντιγραφή όλα τα στοιχεία της ομάδας εκτός από το πρώτο. Επιστρέφει iterator στο νέο container, στην πρώτη θέση μετά το τελευταίο στοιχείο που δεν έχει γραφτεί. Θεωρήστε ότι επαρκούν οι θέσεις μετά το beg2 για όσα στοιχεία αντιγραφούν. Επομένως, συμπληρώστε τον κώδικα

    ```
    template<typename Iterator1, typename Iterator2>
    Iterator2
    uniq(Iterator1 beg1, Iterator1 end1, Iterator2 beg2)
    {
    }
    ```

    Κατόπιν, δημιουργήστε ένα `std::vector<int>` και αποθηκεύστε σε αυτό τους αριθμούς $1, 6, 6, 5, 2, 2, 2, 8, 9, 9, 8, 9, 7, 0, 1, 1$. Καλέστε τη συνάρτηση που γράψατε για να αντιγράψετε τα μοναδικά στοιχεία του σε άλλο container. Ο νέος container πρέπει να έχει τις τιμές $1, 6, 5, 2, 8, 9, 8, 9, 7, 0, 1$.

    Μπορείτε να τροποποιήσετε το πρόγραμμά σας ώστε να προστίθενται τα στοιχεία σε νέες θέσεις μετά το beg2;

11. Γράψτε συνάρτηση που θα δέχεται τέσσερις iterators. Οι δύο πρώτοι, beg1, end1, θα έχουν ίδιο τύπο και θα ορίζουν ένα διάστημα [beg1,end1) σε μια ακολουθία εισόδου. Οι άλλοι δύο, beg2, end2, θα έχουν ίδιο τύπο, ίσως διαφορετικό από τον τύπο του πρώτου ζεύγους, και θα ορίζουν ένα διάστημα [beg2,end2) σε άλλη ακολουθία εισόδου. Οι iterators να είναι τουλάχιστον μονής κατεύθυνσης. Η συνάρτηση θα επιστρέφει το πλήθος των κοινών στοιχείων στις δύο ακολουθίες. Προσέξτε να λαμβάνετε υπόψη *μία φορά* τα στοιχεία που επαναλαμβάνονται.

    Χρησιμοποιήστε τη συνάρτηση που γράψατε για να βρείτε πόσοι αριθμοί είναι κοινοί στις ακολουθίες $\{2, 2, 3, 4, 4, 2, 5, 6, 7\}$ και $\{9, 8, 7, 6, 4, 2, 5, 4, 3, 2, 1, 2, 4\}$, [Απάντηση: είναι 6, οι $\{2, 3, 4, 5, 6, 7\}$.]

Κεφάλαιο 11

Containers

11.1 Εισαγωγή

Οι containers χρησιμοποιούνται για την αποθήκευση και διαχείριση συλλογών από αντικείμενα οποιουδήποτε τύπου, κοινού όμως για όλα. Κάθε container έχει πλεονεκτήματα και μειονεκτήματα, συγκρινόμενοι μεταξύ τους. Ο κατάλληλος container, ωστόσο, υπερέχει σε σύγκριση με την μοναδική αντίστοιχη δομή που παρέχει η C (και κληρονομήθηκε στη C++), το ενσωματωμένο στατικό διάνυσμα ή το μηχανισμό δημιουργίας δυναμικών διανυσμάτων.

Όπως αναφέραμε στο Κεφάλαιο 5, το ενσωματωμένο διάνυσμα, όπως κληρονομήθηκε από τη C, μπορεί να έχει διάσταση είτε γνωστή κατά τη μεταγλώττιση είτε προσδιοριζόμενη κατά την εκτέλεση του προγράμματος. Μετά τη δημιουργία του δεν έχουμε ουσιαστικά ιδιαίτερες ευκολίες στη διαχείρισή του (π.χ. δεν μπορούμε να προσθέσουμε στοιχεία ανάμεσα στα ήδη υπάρχοντα με εύκολο τρόπο ή να κάνουμε γρήγορη αναζήτηση). Αντίθετα, ένα σημαντικό χαρακτηριστικό των containers (εκτός ενός) είναι ότι κάνουν *δυναμική* διαχείριση μνήμης (δέσμευση/αποδέσμευση), και μάλιστα *αυτόματα*. Αυτό σημαίνει πως χωρίς την παρέμβαση του προγραμματιστή, οι θέσεις μνήμης που καταλαμβάνουν μπορούν να αυξάνουν ή να μειώνονται ώστε να χωρούν τα στοιχεία που αποτελούν κάθε στιγμή τη συλλογή. Κάποιοι containers επιτρέπουν με πολύ απλό μηχανισμό για τον προγραμματιστή, την εισαγωγή ή διαγραφή στοιχείων σε οποιοδήποτε σημείο τους. Άλλοι αποθηκεύουν τα στοιχεία τους με τέτοιο τρόπο ώστε η αναζήτηση σε αυτά να είναι ταχύτατη. Αυτές και άλλες δυνατότητές τους μας διευκολύνουν στο να προσαρμόσουμε τους containers της C++ στις ανάγκες του προγράμματός μας.

11.1.1 Κατηγορίες container

Οι containers διακρίνονται σε τρεις γενικές κατηγορίες:

- Οι *sequence containers* (σειριακές συλλογές) είναι συλλογές ομοειδών στοιχείων στις οποίες κάθε ένα έχει συγκεκριμένη θέση ως προς τα υπόλοιπα. Αυτή η σχετική θέση προσδιορίζεται κατά την εισαγωγή κάθε στοιχείου και είναι ανεξάρτητη από την τιμή του. Π.χ. αν συγκεντρώσουμε έναν αριθμό στοιχείων σε έναν τέτοιο container προσθέτοντάς τα διαδοχικά στο τέλος του, θα αποθηκευθούν στον container με τη συγκεκριμένη σειρά που εισήχθησαν. Η Standard Library περιλαμβάνει πέντε κλάσεις που είναι sequence containers: `array<>`, `vector<>`, `deque<>`, `forward_list<>` και `list<>`. Επιπλέον, το `std::string` και το ενσωματωμένο διάνυσμα μπορούν να θεωρηθούν ότι έχουν παρόμοια χαρακτηριστικά με containers τέτοιου τύπου και να χρησιμοποιηθούν με παραπλήσιο τρόπο.

- Οι *associative containers* (συσχετιστικές συλλογές) 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. Προσέξτε πως η *αυτόματη* ταξινόμηση που γίνεται κατά την εισαγωγή των στοιχείων δε σημαίνει ότι οι containers αυτής της κατηγορίας είναι ειδικά σχεδιασμένοι ή οι μόνοι ικανοί για ταξινόμηση. Το σημαντικό πλεονέκτημα έναντι των sequence containers είναι η ταχύτητα στην εύρεση συγκεκριμένου στοιχείου (με δυαδική αναζήτηση) καθώς η ακολουθία σε αυτούς είναι ήδη ταξινομημένη. Η Standard Library παρέχει τέσσερις κλάσεις που είναι associative containers: `set<>`, `multiset<>`, `map<>` και `multimap<>`.

- Οι *unordered associative containers* (συσχετιστικές συλλογές χωρίς σειρά) αποτελούν ένα σύνολο στοιχείων χωρίς καθορισμένη σειρά. Η θέση κάθε στοιχείου στη συλλογή είναι μεν συγκεκριμένη αλλά μπορεί να αλλάξει κατά την εκτέλεση του προγράμματος (π.χ. μετά από εισαγωγή ή διαγραφή στοιχείων). Ουσιαστικά, έχει νόημα να ελέγχουμε μόνο αν ένα στοιχείο υπάρχει ή όχι σε ένα τέτοιο container. Ο έλεγχος αυτός είναι γενικά πιο γρήγορος από την αναζήτηση στοιχείου στις άλλες κατηγορίες container. Κατ' αντιστοιχία των associative containers, η Standard Library παρέχει τέσσερις κλάσεις αυτής της κατηγορίας:

  - `unordered_set<>`,
  - `unordered_multiset<>`,
  - `unordered_map<>`, και
  - `unordered_multimap<>`.

11.2 Δήλωση

Οι containers είναι υλοποιημένοι ως υποδείγματα κλάσης (class template, (§14.9)). Δέχονται ως πρώτη παράμετρο τον τύπο των ποσοτήτων που θα αποθηκευθούν ή, στην περίπτωση των `std::map<>`, `std::multimap<>`, και στις παραλλαγές τους χωρίς σειρά, `std::unordered_map<>`, `std::unordered_multimap<>`, τους τύπους *δύο* ποσοτήτων, καθώς αυτοί αποθηκεύουν *ζεύγη* τιμών.

Επιπλέον, οι containers δέχονται και άλλες παραμέτρους:

- Στον `std::array<>` πρέπει να προσδιορίσουμε εκτός από τον τύπο *και* το πλήθος των στοιχείων που θα αποθηκεύει, ως δεύτερη παράμετρο. Το πλήθος πρέπει να είναι σταθερή (`constexpr`) έκφραση, δηλαδή γνωστή κατά τη μεταγλώττιση.
- Στους associative containers η επόμενη παράμετρος μετά τον ή τους τύπους των αποθηκευόμενων τιμών καθορίζει το κριτήριο με το οποίο γίνεται η ταξινόμηση.
- Στους unordered containers οι επόμενες δύο παράμετροι προσδιορίζουν τη συνάρτηση hash και το κριτήριο ισότητας στοιχείων.

Η τελευταία παράμετρος στους containers σχετίζεται με τη διαχείριση της μνήμης και έχει προκαθορισμένη τιμή. Τη δέχονται όλοι οι containers εκτός από τον `std::array<>`, καθώς σε αυτόν η δέσμευση μνήμης γίνεται κατά τη μεταγλώττιση. Δε θα αναφερθούμε περισσότερο σε αυτή.

Για να γίνει διαθέσιμος ένας container σε κάποιο κώδικα πρέπει να γίνει η συμπερίληψη του αντίστοιχου header:

- `<array>` για το `array<>`,
- `<vector>` για το `vector<>`,
- `<deque>` για το `deque<>`,
- `<forward_list>` για το `forward_list<>`,
- `<list>` για το `list<>`,
- `<set>` για τα `set<>` *και* `multiset<>`,
- `<map>` για τα `map<>` *και* `multimap<>`,
- `<unordered_set>` για τα `unordered_set<>` *και* `unordered_multiset<>`,
- `<unordered_map>` για τα `unordered_map<>` *και* `unordered_multimap<>`.

Ας επαναλάβουμε ότι οι containers, όπως όλη η Standard Library, ορίζονται στο χώρο ονομάτων `std`.

11.2.1 Τρόποι ορισμού

Ας παραθέσουμε κάποιους γενικούς τρόπους ορισμού που είναι κοινοί για όλους τους containers, εκτός από τον `std::array<>`. Τους τρόπους ορισμού ενός array θα τους παρουσιάσουμε στην §11.5.1.

Έστω cntr ένας οποιοσδήποτε τύπος τέτοιου container. Στους παρακάτω ορισμούς το `<T>` αντιπροσωπεύει συλλογικά τις κατάλληλες παραμέτρους του template κάθε container (τύπος στοιχείων, κριτήριο ταξινόμησης, κλπ.).

- Η δήλωση

  ```
  cntr<T> c1;
  ```

  ορίζει το c1 ως ένα κενό (χωρίς στοιχεία) cntr.

- Ο κώδικας

  ```
  cntr<T> c1;
  // fill c1 ....
  cntr<T> c2{c1};
  ```

  ορίζει το c1 ως ένα αρχικά κενό cntr, εισάγει στοιχεία σε αυτό (θα δούμε παρακάτω πώς) και δημιουργεί το c2 ως *αντίγραφο* του c1, στοιχείο προς στοιχείο. Τα c1, c2 πρέπει φυσικά να είναι ίδιου τύπου. Προσέξτε ότι ο τύπος περιλαμβάνει εκτός από το είδος του container, *και τις παραμέτρους του template*. Ισοδύναμες με την τελευταία εντολή είναι οι παρακάτω εντολές

  ```
  cntr<T> c2=c1;
  cntr<T> c2(c1);
  ```

- Ο κώδικας

  ```
  cntr<T> c1;
  // fill c1 ....
  cntr<T> c2{std::move(c1)};
  ```

  ορίζει το c1 ως ένα αρχικά κενό cntr, εισάγει στοιχεία σε αυτό και, σύμφωνα με όσα αναφέραμε στη §2.17.1, δημιουργεί το c2 με *μετακίνηση* των στοιχείων του c1. Όπως έχουμε αναφέρει, η μετακίνηση στοιχείων είναι πιο γρήγορη από την αντιγραφή. Μετά τη μετακίνηση, το c1 υπάρχει αλλά είναι κενό. Μπορεί να χρησιμοποιηθεί μόνο για μετακίνηση άλλου container τύπου `cntr<T>` σε αυτόν (ή να καταστραφεί).

  Ισοδύναμες με την τελευταία εντολή είναι οι

  ```
  cntr<T> c2=std::move(c1);
  cntr<T> c2(std::move(c1));
  ```

- Μπορούμε να δημιουργήσουμε ένα container με συγκεκριμένες τιμές v1, v2, v3, ..., για τα στοιχεία του, παραθέτοντας τη λίστα τιμών κατά τη δήλωσή του:

  ```
  T v1, v2, v3, ...;
  cntr<T> c1{v1, v2, v3, ... };
  ```

  Ισοδύναμα θα μπορούσαμε να γράψουμε μία από τις εντολές

  ```
  cntr<T> c1 = {v1, v2, v3, ... };
  cntr<T> c1({v1, v2, v3, ... });
  ```

  Εννοείται ότι οι τιμές της λίστας πρέπει να είναι του ίδιου τύπου με τα στοιχεία του container ή να μπορούν να μετατραπούν αυτόματα σε αυτόν.

- Αν beg και end είναι δύο input iterators *στην ίδια ακολουθία τιμών, με τον beg να μη «δείχνει» μετά τον end*, μπορούμε να δημιουργήσουμε ένα container αντιγράφοντας σε αυτόν το τμήμα των στοιχείων στο διάστημα [beg,end). Η σχετική εντολή είναι

  ```
  cntr<T> c1{beg,end};
  ```

  Ο τύπος των στοιχείων στο διάστημα [beg,end) δεν είναι απαραίτητα ίδιος με τον τύπο των στοιχείων του container· αρκεί να μπορεί να μετατραπεί αυτόματα σε αυτόν ή γενικότερα, οι τιμές των στοιχείων να μπορούν να παραγάγουν μέσω κατάλληλου constructor (§14.5.1) τα στοιχεία του container. Έτσι αν π.χ. ο container αποθηκεύει `double`, τα στοιχεία του διαστήματος μπορούν να είναι `int` (καθώς ορίζεται η μετατροπή `int` σε `double`) αλλά όχι `std::complex<double>`.

Επιπλέον των παραπάνω, υπάρχουν και άλλοι τρόποι για δήλωση με ταυτόχρονη απόδοση αρχικών τιμών, συγκεκριμένοι για κάθε container.

Παράδειγμα

Στον παρακάτω κώδικα ορίζεται μια μεταβλητή v1 ως vector ακεραίων, με ταυτόχρονο ορισμό των αρχικών τιμών των στοιχείων του. Το v1 αντιγράφεται κατά τη δημιουργία ενός όμοιου vector με όνομα v2. Κατόπιν, δημιουργείται και ένα set s1 με αντίγραφα των στοιχείων του v1, χρησιμοποιώντας κατάλληλους const_iterators. Με την επόμενη εντολή, τα στοιχεία του s1 μετακινούνται σε νέο set που δημιουργείται σε αυτό το σημείο:

```
#include <vector>
#include <set>

std::vector<int> v1{0,2,-2,4,-4,6,-6};
std::vector<int> v2{v1};
std::set<int> s1{v1.cbegin(), v1.cend()};
```

```
std::set<int> s2{std::move(s1)};
```

Όπως όλες οι μεταβλητές ενσωματωμένων τύπων, μια μεταβλητή τύπου container καταστρέφεται όταν η ροή εκτέλεσης βγει εκτός της εμβέλειάς της. Η καταστροφή της ελευθερώνει τη μνήμη που καταλαμβάνει αυτή, *αυτόματα*.

11.3 Τροποποίηση container

Σε οποιοδήποτε container, εκτός του `std::array<>` που δημιουργείται με συγκεκριμένο σταθερό πλήθος στοιχείων, εισαγωγή ή τροποποίηση στοιχείων γίνεται ως εξής:

- Με εκχώρηση άλλου container, ίδιου τύπου:

  ```
  cntr<T> c1,c2;
  ...
  c1 = c2;
  ```

  Με την παραπάνω εκχώρηση, τα αρχικά στοιχεία του c1 σβήνονται και αντιγράφονται στη θέση τους τα στοιχεία του c2. Το μέγεθος του c1 προσαρμόζεται ώστε να χωρέσει ακριβώς τα στοιχεία του c2.

  Αν δεν επιθυμούμε να διατηρήσουμε τον c2 μπορούμε να μετακινήσουμε (και όχι να αντιγράψουμε) τα στοιχεία του στον c1:

  ```
  c1 = std::move(c2);
  ```

  Παρατηρήστε τη χρήση της `std::move()` του `<utility>` ώστε ο c2 να μετατραπεί σε ποσότητα που μπορεί να μετακινηθεί.

- Με εκχώρηση λίστας στοιχείων:

  ```
  cntr<T> c;
  c = {v1,v2,v3, ...};
  ```

  Με αυτή την εντολή τα στοιχεία του c αντικαθιστώνται ή καταστρέφονται και και αντιγράφονται στη θέση τους οι τιμές της λίστας, αλλάζοντας πιθανόν και το μέγεθος του c. Οι τιμές της λίστας πρέπει να μπορούν να μετατραπούν αυτόματα στον τύπο των στοιχείων του container.

- Με εναλλαγή στοιχείων με container ίδιου τύπου, χρησιμοποιώντας τη συνάρτηση `std::swap()` του `<utility>`:

  ```
  cntr<T> c1,c2;
  std::swap(c1,c2);
  ```

  ή, ισοδύναμα, τη συνάρτηση–μέλος `swap()`:

```
cntr<T> c1,c2;
c1.swap(c2);
```

Η συνάρτηση `swap()` μετακινεί, δεν αντιγράφει, τα στοιχεία του ενός container στον άλλο και αντίστροφα, με συνέπεια να είναι ταχύτατη.

- Με τη συνάρτηση–μέλος `insert()` κάθε container, εκτός από `array<>` και `forward_list<>`. Ο πρώτος container δεν υποστηρίζει εισαγωγή στοιχείων. Ο δεύτερος ονομάζει τη συνάρτηση `insert_after()`. Δέχεται ως ορίσματα ένα iterator σε σταθερό στοιχείο (const_iterator) στον container για τον οποίο καλείται, και μία τιμή κατάλληλη για αποθήκευση σε αυτόν, με αντιγραφή ή μετακίνηση:

  ```
  cntr<T> c;
  c.insert(pos, elem);
  ```

  Με την παραπάνω κλήση, γίνεται εισαγωγή αντιγράφου του elem ή μετακίνηση του elem—αν έχει προσαρμοστεί με τη `std::move()`. Η θέση εισαγωγής καθορίζεται (για sequence containers) ή προτείνεται (για associative containers) από τον const_iterator pos. Θυμηθείτε ότι σε associative containers η θέση του στοιχείου καθορίζεται μόνο από την τιμή του σε σχέση με τα ήδη υπάρχοντα στοιχεία και, επομένως, μπορούμε μόνο να υποδείξουμε την πιθανή θέση για να γίνει πιο γρήγορα η εισαγωγή. Η επιστρεφόμενη τιμή είναι iterator στη θέση του νέου στοιχείου[1]. Παρατηρήστε ότι με τη συνάρτηση–μέλος `insert()` γίνεται *εισαγωγή* νέου στοιχείου και όχι *αντικατάσταση* κάποιου υπάρχοντος.

- Με το μηχανισμό των αλγορίθμων που θα δούμε στο Κεφάλαιο 12.

Επίσης, μεταβολή της τιμής ενός στοιχείου μπορούμε να κάνουμε με το μηχανισμό των iterators που περιγράψαμε στο Κεφάλαιο 10 ή των δεικτών (§2.18).

Διαγραφή στοιχείων γίνεται:

- Με τη συνάρτηση–μέλος `clear()`. Αυτή διαγράφει όλα τα στοιχεία αφήνοντας κενό τον container για τον οποίο καλείται. Δεν επιστρέφει τίποτε («επιστρέφει» `void`) και βέβαια ακυρώνει όλες τις αναφορές, τους δείκτες και τους iterators σε στοιχεία του container:

  ```
  c.clear();
  ```

- Με τη χρήση της συνάρτησης–μέλους `erase()` κάθε container εκτός από `array<>` και `forward_list<>`, σε μία από τις παρακάτω μορφές:

  ```
  c.erase(pos);
  c.erase(beg,end);
  ```

[1] Σε set και unordered_set, map και unordered_map, αν το στοιχείο (ή το κλειδί του) υπάρχει ήδη, επιστρέφεται iterator στο υπάρχον στοιχείο.

Με την πρώτη κλήση διαγράφεται το στοιχείο που δείχνει ο const_iterator pos· με τη δεύτερη διαγράφονται τα στοιχεία με const_iterators στο διάστημα [beg,end) (απαιτείται, βέβαια, το διάστημα να μην είναι κενό και να είναι διάστημα στο c). Αν ο c είναι sequence container η συνάρτηση επιστρέφει iterator στην επόμενη θέση από το τελευταίο στοιχείο που διαγράφηκε ενώ σε associative containers δεν επιστρέφει τίποτε.

- Με κατάλληλους αλγόριθμους.

Πέρα από αυτούς, κάθε container παρέχει και άλλους μηχανισμούς για προσπέλαση και μεταβολή των στοιχείων του.

11.4 Κοινά μέλη των containers

Κάθε container (αλλά και το `std::string`) ορίζει «εσωτερικά», συγκεκριμένα ονόματα τύπων. Έχουμε ήδη παρουσιάσει στο Κεφάλαιο 10 τους τύπους για iterators σε στοιχείο σταθερό ή μη, `const_iterator` και `iterator` αντίστοιχα. Κάποιοι containers παρέχουν και τους ανάστροφους iterators, σε στοιχείο σταθερό, `const_reverse_iterator` , ή μη, `reverse_iterator` . Υπάρχουν επιπλέον και άλλοι τύποι. Μεταξύ άλλων, κάθε container ορίζει (με **`typedef`** (§2.15.1)) τους τύπους:

- `value_type`, που είναι ο τύπος των στοιχείων που αποθηκεύει,
- `reference`, που είναι ο τύπος της αναφοράς στα στοιχεία,
- `const_reference` , που είναι ο τύπος της αναφοράς σε σταθερά στοιχεία,
- `pointer`, που είναι ο τύπος του δείκτη στα στοιχεία,
- `const_pointer` , που είναι ο τύπος του δείκτη σε σταθερά στοιχεία,
- `difference_type` , που είναι εμπρόσημος τύπος για διαφορές θέσεων των στοιχείων,
- `size_type`, που είναι απρόσημος ακέραιος τύπος για την αρίθμηση (και το πλήθος) των θέσεων αποθήκευσης.

Οι τύποι αυτοί μπορούν να χρησιμοποιηθούν από τον προγραμματιστή για τη δήλωση σχετικών ποσοτήτων. Στη δήλωση γράφουμε τον τύπο του container, κατόπιν τον τελεστή εμβέλειας '::' και μετά, το όνομα του τύπου. Έτσι, π.χ., ο `std::vector<double>` παρέχει τον τύπο `std::vector<double>::value_type`. Δήλωση ποσότητας κατάλληλης για να αποθηκεύσει στοιχεία τέτοιου container με όνομα v είναι η ακόλουθη

```
std::vector<double>::value_type a{v[0]};
```

Η μεταβλητή a είναι τύπου `double`.

Αυτός ο έμμεσος, φαινομενικά περιττός προσδιορισμός του τύπου των στοιχείων ή των δεικτών και αναφορών σε αυτά, έχει ιδιαίτερη χρησιμότητα όταν ο τύπος του container είναι παράμετρος template. Ακολουθώντας όσα αναφέραμε στο §10.2.1, αν σε template με παράμετρο ένα τύπο container θέλουμε να χρησιμοποιήσουμε τον τύπο των στοιχείων που αποθηκεύει αυτός, πρέπει να γράψουμε κάτι σαν

```
template<typename C>
void
f(C c)
{
   typename C::value_type a;
...
}
```

Η μεταβλητή a έχει κατάλληλο τύπο για την αποθήκευση των στοιχείων του c.

Οι containers παρέχουν, επιπλέον, ορισμένες κοινές συναρτήσεις–μέλη για την παραγωγή iterators, τον έλεγχο του μεγέθους τους, τη σύγκρισή τους, την εκχώρηση τιμών, κλπ. Παρατίθενται παρακάτω.

Θυμηθείτε ότι μια συνάρτηση–μέλος της κλάσης X, με όνομα member, καλείται για ένα αντικείμενο αυτής της κλάσης με όνομα a, κάνοντας χρήση του τελεστή επιλογής μέλους, '.' μεταξύ του ονόματος τού container και της συνάρτησης (με τα ορίσματά της):

```
a.member();
```

Εντός παρενθέσεων ακολουθούν τα ορίσματα της συνάρτησης, αν υπάρχουν. Φυσικά, αν η συνάρτηση–μέλος επιστρέφει κάποια τιμή που τη χρειαζόμαστε, πρέπει να την εκχωρήσουμε σε ποσότητα κατάλληλου τύπου.

11.4.1 Iterators αρχής και τέλους

Έχουμε παρουσιάσει στο Κεφάλαιο 10 τις συναρτήσεις–μέλη που παράγουν iterators αρχής και τέλους σε οποιονδήποτε container. Τις παραθέτουμε συγκεντρωτικά στον Πίνακα 11.1.

Οι ανάστροφοι iterators δεν παρέχονται από το `forward_list<>` και τους unordered containers. Ο `forward_list<>` παρέχει τις συναρτήσεις `before_begin()` και `cbefore_begin()` που επιστρέφουν αντίστοιχα iterator και const_iterator σε θέση πριν το πρώτο στοιχείο.

Υπενθυμίζουμε ότι το «τέλος» για τους iterators και τους ανάστροφους iterators ενός container είναι μία θέση *μετά την τελευταία* ή *πριν την πρώτη* αντίστοιχα. Καθώς εκεί δεν υπάρχει στοιχείο, δεν επιτρέπεται η απόπειρα προσπέλασής του με τη δράση του τελεστή '*' στον iterator. Προσέξτε επίσης ότι ο reverse iterator `rbegin()` θεωρείται πως είναι *πριν* τον reverse iterator `rend()`· ο συνδυασμός των δύο χρησιμεύει στο να διατρέχουμε ένα container ανάστροφα. Αντίστοιχα ισχύουν για τους ανάστροφους iterators σε σταθερό στοιχείο, `crbegin()`, `crend()`.

Πίνακας 11.1: Συναρτήσεις–μέλη για iterators σε containers της Standard Library

Συνάρτηση	**Επιστρεφόμενος iterator**
`begin()`	Iterator στη θέση του πρώτου στοιχείου.
`end()`	Iterator σε *μία θέση μετά* το τελευταίο στοιχείο.
`cbegin()`	Iterator στη θέση του πρώτου στοιχείου (χωρίς δυνατότητα τροποποίησης).
`cend()`	Iterator σε *μία θέση μετά* το τελευταίο στοιχείο (χωρίς δυνατότητα τροποποίησης).
`rbegin()`	Ανάστροφος iterator στη θέση του τελευταίου στοιχείου.
`rend()`	Ανάστροφος iterator σε *μία θέση πριν* το πρώτο στοιχείο.
`crbegin()`	Ανάστροφος iterator στη θέση του τελευταίου στοιχείου (χωρίς δυνατότητα τροποποίησης).
`crend()`	Ανάστροφος iterator σε *μία θέση πριν* το πρώτο στοιχείο (χωρίς δυνατότητα τροποποίησης).

11.4.2 Έλεγχος μεγέθους

Όλοι (σχεδόν) οι containers παρέχουν συναρτήσεις–μέλη που επιστρέφουν ποσότητες σχετικές με το μέγεθός τους. Είναι οι ακόλουθες:

size() Επιστρέφει το πλήθος των στοιχείων κατά τη στιγμή της κλήσης.

Η συγκεκριμένη συνάρτηση–μέλος δεν ορίζεται για `forward_list<>` καθώς δεν είναι ιδιαίτερα γρήγορη η υλοποίησή της για τέτοιο container.

max_size() Επιστρέφει το μέγιστο δυνατό πλήθος στοιχείων (καθοριζόμενο από την υλοποίηση). Για ένα `array` επιστρέφει ό,τι και η `size()`.

empty() Επιστρέφει λογική τιμή, **`false/true`**, αν ο container είναι κενός ή όχι. Είναι ισοδύναμη με `size()==0` αλλά πιθανόν πιο γρήγορη.

Οι συναρτήσεις `size()` και `max_size()` επιστρέφουν τιμή με τύπο το `size_type` που ορίζεται σε κάθε container. Έτσι, ένας `std::vector<double>` με όνομα v έχει πλήθος στοιχείων που μπορεί να εκχωρηθεί σε μεταβλητή όπως στην παρακάτω εντολή

```
std::vector<double>::size_type n{v.size()};
```

Η χρήση του **`auto`** για την αυτόματη αναγνώριση τύπου απλοποιεί την προηγούμενη εντολή:

```
auto n = v.size();
```

11.4.3 Σύγκριση containers

Για δύο ποσότητες *ίδιου τύπου* container ορίζονται οι τελεστές σύγκρισης του Πίνακα 3.1 με τη γνωστή ερμηνεία τους. Η ισότητα δύο containers σημαίνει ότι έχουν το ίδιο πλήθος στοιχείων, με την ίδια σειρά και τιμή. Η έννοια του «μικρότερου» ή «μεγαλύτερου» καθορίζεται *λεξικογραφικά* (§9.4.1).

Για τους unordered containers δεν έχει νόημα (και δεν παρέχεται) η δυνατότητα σύγκρισης για μικρότερο και μεγαλύτερο αλλά μόνο για ισότητα και ανισότητα.

Η σύγκριση δύο containers διαφορετικού τύπου μπορεί να γίνει με κατάλληλους αλγόριθμους. Θα τους περιγράψουμε στο Κεφάλαιο 12.

11.5 Sequence Containers

11.5.1 array

Ο container `std::array<>` διαφέρει σε πολλά από τους υπόλοιπους containers της Standard Library. Περιβάλλει ως «κέλυφος» το ενσωματωμένο διάνυσμα με πλήθος στοιχείων γνωστό κατά τη μεταγλώττιση και έχει εισαχθεί στη C++ ώστε να καταστήσει το στατικό διάνυσμα της C διαχειρίσιμο με παρόμοιο τρόπο με τους άλλους containers. Ένα array αποθηκεύει τα στοιχεία του σε διαδοχικές θέσεις μνήμης.

Ο συγκεκριμένος container είναι ιδιαίτερος καθώς η δημιουργία του γίνεται κατά τη διάρκεια της μεταγλώττισης και το πλήθος στοιχείων του δεν μπορεί να μεταβληθεί κατά την εκτέλεση του προγράμματος. Έτσι, κάποιοι από τους γενικούς τρόπους ορισμού δεν έχουν εφαρμογή σε αυτόν.

Ο τύπος `std::array<>` παρέχει όλους τους κοινούς τύπους, τις συναρτήσεις–μέλη και τους τελεστές σύγκρισης που παρουσιάσαμε στο §11.4. Η χρήση του προϋποθέτει τη συμπερίληψη του header `<array>`. Οι iterators που παρέχει ο τύπος αυτός, ορθής και ανάστροφης φοράς, είναι τυχαίας προσπέλασης (random iterators).

Ορισμός

Ορισμό ενός `array` έχουμε με τους τρόπους που ακολουθούν. Σε όλους, το πλήθος των στοιχείων, N, είναι μια ακέραιη σταθερή ποσότητα, που μπορεί να προκύπτει από κάποια έκφραση ή κλήση συνάρτησης `constexpr`. Σημασία έχει, το πλήθος να μπορεί να υπολογιστεί κατά τη μεταγλώττιση.

- Η δήλωση

  ```
  std::array<T,N> a;
  ```

  ορίζει το a ως ένα array με N θέσεις για στοιχεία τύπου T. Οι τιμές των στοιχείων είναι απροσδιόριστες, αν ο τύπος T είναι από τους θεμελιώδεις ή αν δεν υπάρχει προκαθορισμένος constructor (§14.5.4) στην κλάση T.

- Ο κώδικας

```
std::array<T,N> a1;
// assign values to a1 ...
std::array<T,N> a2{a1};
```

ορίζει το array a1, του εκχωρεί τιμές (θα δούμε παρακάτω πώς) και δημιουργεί το ίδιου τύπου array a2 ως *αντίγραφο* του a1. Ισοδύναμες με την τελευταία εντολή είναι και οι παρακάτω εντολές

```
std::array<T,N> a2 = {a1};
std::array<T,N> a2 = a1;
std::array<T,N> a2(a1);
```

Αν επιθυμούμε να ορίσουμε ένα array και η αρχική του τιμή να προέλθει με *μετακίνηση* των στοιχείων άλλου array, πρέπει να μετατρέψουμε το αρχικό array σε ποσότητα που μπορεί να μετακινηθεί με τη συνάρτηση `std::move()` και να γράψουμε

```
std::array<T,N> a1;
// assign values to a1 ...
std::array<T,N> a2{std::move(a1)};
```

ή τις ισοδύναμες εκφράσεις. Η μετακίνηση είναι γενικά πιο γρήγορη από την αντιγραφή αλλά αφήνει τα στοιχεία του αρχικού array σε απροσδιόριστη κατάσταση.

- Ο κώδικας

```
T v1,v2,..., vN;
std::array<T,N> a{v1, v2, ..., vN};
```

δημιουργεί το array a με N στοιχεία. Οι αρχικές τιμές αυτών παρατίθενται εντός αγκίστρων. Εάν κάποιες (ή όλες) παραλείπονται, η λίστα τιμών συμπληρώνεται στο τέλος της με την προκαθορισμένη τιμή `T{}` για το συγκεκριμένο τύπο των στοιχείων, (η τιμή είναι συνήθως το 0 αφού μετατραπεί στον τύπο T). Είναι λάθος αν η λίστα περιλαμβάνει περισσότερες τιμές από τα στοιχεία του array. Ισοδύναμα με την παραπάνω αρχικοποίηση θα μπορούσαμε να γράψουμε τις εντολές

```
std::array<T,N> a = {v1, v2, ..., vN};
std::array<T,N> a({v1, v2, ..., vN});
std::array<T,N> a{{v1, v2, ..., vN}};
```

Συνεπώς, στον κώδικα

```
std::array<int,10> a;
std::array<int,10> b{};
```

τα διανύσματα a,b έχουν 10 ακέραια στοιχεία· τα στοιχεία του a δεν έχουν συγκεκριμένη τιμή ενώ του b έχουν την τιμή 0.

Προσπέλαση στοιχείων

Εκχώρηση τιμής σε όλα τα στοιχεία ενός array μετά τη δημιουργία του, μπορεί να γίνει

- με τη συνάρτηση–μέλος `fill()`. Στον κώδικα

  ```
  std::array<int,10> a;
  a.fill(20);
  ```

  δημιουργείται ένα array 10 θέσεων με αρχικά απροσδιόριστες τιμές. Κατόπιν, όλα τα στοιχεία αποκτούν την τιμή 20.

- Με εκχώρηση άλλου array, ίδιου τύπου. Προσέξτε ότι ο τύπος περιλαμβάνει και το πλήθος στοιχείων:

  ```
  std::array<int,10> a, b;
  a.fill(20);
  b = a;
  ```

  Με την τελευταία εντολή γίνεται αντιγραφή των στοιχείων του a στα στοιχεία του b. Αν δεν επιθυμούμε να διατηρήσουμε τον a μπορούμε να μετακινήσουμε (και όχι να αντιγράψουμε) τα στοιχεία του στο b:

  ```
  b = std::move(a);
  ```

- Με εκχώρηση λίστας στοιχείων:

  ```
  std::array<int,3> a;
  a = {-1,-2,-3};
  ```

  Εννοείται ότι οι τιμές της λίστας πρέπει να μπορούν να μετατραπούν αυτόματα στον τύπο των στοιχείων του container και δεν θα πρέπει να είναι περισσότερες από τις θέσεις του. Εάν κάποιες (ή όλες) παραλείπονται, η λίστα τιμών συμπληρώνεται στο τέλος της με την προκαθορισμένη τιμή για το συγκεκριμένο τύπο των στοιχείων.

- Με εναλλαγή στοιχείων με άλλο array ίδιου τύπου, χρησιμοποιώντας τη συνάρτηση `std::swap()` του `<utility>`:

  ```
  std::array<int,10> a, b;
  a.fill(20);
  std::swap(a,b);
  ```

  ή, ισοδύναμα, τη συνάρτηση–μέλος `swap()`:

  ```
  b.swap(a);
  ```

  Η συνάρτηση `swap()` μετακινεί τα στοιχεία του ενός array στον άλλο και αντίστροφα.

Προσπέλαση και επομένως, δυνατότητα μεταβολής των μεμονωμένων στοιχείων ενός array γίνεται ως εξής:

- Με τη χρήση ακέραιου δείκτη μεταξύ των αγκυλών '`[]`':

```
std::array<int,5> a{2,4,6,8,10};
std::cout << a[0] << '\n'; // a[0] = 2
a[3] = a[2]+a[1]; // a[3] = 10
```

- Με τη χρήση της συνάρτησης–μέλους `at()` με ακέραιο όρισμα. Το πρώτο, δεύτερο, τρίτο,...στοιχείο ενός `std::array<>` με όνομα a είναι το `a.at[0]`, `a.at[1]`, `a.at[2]`,.... Προσέξτε πως η διαφορά από την προηγούμενη περίπτωση είναι ότι αν το όρισμα δεν αντιστοιχεί σε θέση του array εγείρεται εξαίρεση του τύπου `std::out_of_range`. Στην περίπτωση που αυτή δεν συλληφθεί, διακόπτεται η εκτέλεση του προγράμματος. Ο απαιτούμενος έλεγχος στην τιμή του δείκτη έχει ως αποτέλεσμα να είναι πιο αργή η πρόσβαση στο στοιχείο απ' ό,τι με τις αγκύλες.

  Στον κώδικα

```
std::array<int,5> a{2,4,6,8,10};
std::cout << a.at(0) << '\n';
std::cout << a.at(10) << '\n';
```

  τυπώνεται το πρώτο στοιχείο του a και διακόπτεται με μήνυμα λάθους η εκτέλεση κατά την απόπειρα προσπέλασης του ενδέκατου στοιχείου του a (καθώς αυτό δεν υπάρχει).

- Με το υπόδειγμα συνάρτησης `std::get()`. Δέχεται ως όρισμα ένα array και έχει ως παράμετρο του template μια ακέραιη σταθερά –όχι μεταβλητή, μικρότερη από το πλήθος των στοιχείων του array. Στον κώδικα

```
std::array<int,5> a{2,4,6,8,10};
std::cout << std::get<3>(a) << '\n'; // a[3] = 8
std::get<0>(a) = 5; // a[0] = 5
```

  εκτυπώνεται το τέταρτο στοιχείο του array και αλλάζει η τιμή του πρώτου μέσω της `std::get()`. Προσέξτε ότι ο έλεγχος της παραμέτρου του template (να είναι μη αρνητική και μικρότερη από το πλήθος των στοιχείων) γίνεται κατά τη *μεταγλώττιση*. Αυτή διακόπτεται αν η παράμετρος έχει τιμή εκτός των ορίων.

- Με τις συναρτήσεις–μέλη `front()` και `back()`. Αυτές επιστρέφουν αναφορά στο πρώτο και στο τελευταίο στοιχείο αντίστοιχα. Εννοείται ότι το array θα έχει τουλάχιστον ένα στοιχείο αλλιώς έχουν απροσδιόριστη επιστρεφόμενη τιμή:

```
std::array<int,5> a{2,4,6,8,10};
std::cout << "first␣element␣is␣" << a.front()
          << "␣last␣element␣is␣" << a.back() << '\n';
a.front() = 11; // a[0] = 11
a.back()  = 55; // a[4] = 55
```

- Με τη δράση του τελεστή '∗' σε όνομα iterator. Φυσικά, αν ο iterator είναι σε σταθερό στοιχείο (π.χ. `const_iterator`) δεν μπορούμε να μεταβάλουμε την τιμή που «δείχνει» αυτός αλλά μόνο να τη διαβάσουμε.
- Με τη δράση του τελεστή '∗' σε δείκτη σε κάποιο στοιχείο του array. Επιτρέπεται η μετακίνηση του δείκτη σε άλλη θέση.

  Στον κώδικα

  ```
  std::array<int,5> a{2,4,6,8,10};
  int * p{&a[1]};
  *p=30;
  --p;     // p ≡ &a[0]
  *p=1;
  ```

  το δεύτερο στοιχείο του array παίρνει την τιμή 30 και το πρώτο την τιμή 1.

  Εναλλακτικά, μπορούμε να χρησιμοποιήσουμε τη συνάρτηση–μέλος `data()` που επιστρέφει δείκτη στο πρώτο στοιχείο του array για το οποίο καλείται. Έτσι, στον κώδικα

  ```
  std::array<int,5> a{2,4,6,8,10};
  int * p{a.data()}; // p ≡ &a[0]
  p+=2;              // p ≡ &a[2]
  *p=-1;
  ```

  το τρίτο στοιχείο τού a γίνεται -1.

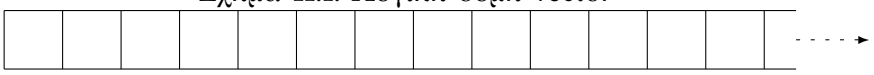
11.5.2 vector

Ποσότητα τύπου `std::vector<>` αποθηκεύει ένα σύνολο στοιχείων ίδιου τύπου σε διαδοχικές θέσεις μνήμης, επιτρέποντας την τυχαία προσπέλαση σε αυτά. Όπως και σε κάθε sequence container, τα στοιχεία αποθηκεύονται με τη σειρά εισαγωγής τους.

Ένα vector δημιουργείται κατά την εκτέλεση του προγράμματος και όχι κατά τη μεταγλώττιση όπως συμβαίνει στο array. Έχει τη δυνατότητα μεταβολής του πλήθους των στοιχείων του με προσθήκη ή αφαίρεση στοιχείων σε οποιοδήποτε σημείο του. Η μεταβολή του πλήθους είναι ιδιαίτερα γρήγορη αν γίνει στο *τέλος* του container, Σχήμα 11.1, (εκτός αν απαιτηθεί μετακίνηση σε μεγαλύτερο τμήμα μνήμης, όπως θα εξηγήσουμε). Σε οποιοδήποτε άλλο σημείο του η προσθήκη ή αφαίρεση στοιχείου είναι αργή, λίγο ή πολύ. Για να εξασφαλιστεί ότι διαδοχικά

στοιχεία βρίσκονται σε διαδοχικές θέσεις μνήμης, η εισαγωγή ή διαγραφή στοιχείου σε κάποια θέση απαιτεί τη μετακίνηση όλων των επόμενων στοιχείων· η διαδικασία χρειάζεται χρόνο ανάλογο του πλήθους των μετακινούμενων στοιχείων. Η μετακίνηση κάποιων στοιχείων ακυρώνει τους δείκτες, τους iterators και τις αναφορές που έχουμε ορίσει σε αυτά. Επιπλέον, αν το τμήμα της μνήμης που διατέθηκε από το λειτουργικό σύστημα σε ένα vector δεν επαρκεί για την εισαγωγή στοιχείου οπουδήποτε σε αυτό, γίνεται μετακίνηση *όλων* των στοιχείων, μαζί με το προστιθέμενο, σε νέο, μεγαλύτερο τμήμα μνήμης. Σε αυτή την περίπτωση, όλοι οι δείκτες, iterators και αναφορές που έχουμε ορίσει σε στοιχεία του αρχικού vector παύουν να ισχύουν καθώς συνδέονται με το παλαιό τμήμα μνήμης.

Σχήμα 11.1: Λογική δομή vector



Ας σημειωθεί εδώ ότι μπορούμε να προσομοιάσουμε τη συμπεριφορά ενός vector αν χρησιμοποιήσουμε δυναμικό διάνυσμα (§5.2.4), τις ενσωματωμένες συναρτήσεις `std::malloc()`, `std::realloc()`, `std::free()` για δέσμευση, αντιγραφή ή ελευθέρωση μνήμης, και δικές μας συναρτήσεις για εισαγωγή στοιχείου με μετακίνηση άλλων. Όλα τα παραπάνω όμως, πρέπει να προσδιοριστούν ρητά από τον προγραμματιστή. Η αυτόματη διαχείριση της προσθήκης ή διαγραφής στοιχείων και η όμοια συμπεριφορά με τους λοιπούς containers δίνουν στο vector σημαντικό πλεονέκτημα έναντι του δυναμικού διανύσματος και κανένα μειονέκτημα σε ταχύτητα ή ευχρηστία. Επομένως, το `std::vector<>` είναι ο προτιμητέος container όταν χρειαζόμαστε δυναμική διαχείριση μνήμης, χωρίς ειδικές απαιτήσεις, οι οποίες θα ικανοποιούνται καλύτερα από άλλο container.

Ο τύπος `std::vector<>` παρέχει όλους τους κοινούς τύπους, τις συναρτήσεις–μέλη και τους τελεστές σύγκρισης που παρουσιάσαμε στο §11.4. Η χρήση του προϋποθέτει τη συμπερίληψη του `<vector>`. Οι iterators που παρέχει ένα vector, ορθής και ανάστροφης φοράς, είναι τυχαίας προσπέλασης (random iterators).

Ορισμός

Η κλάση `std::vector<>` παρέχει διάφορους μηχανισμούς για τον ορισμό ποσοτήτων με πιθανή ταυτόχρονη απόδοση αρχικής τιμής. Έχουμε ήδη δει κάποιους από αυτούς στο §11.2.1· για πληρότητα, θα τους επαναλάβουμε με συντομία και θα παρουσιάσουμε τους επιπλέον μηχανισμούς για τη δημιουργία vector.

- Η εντολή

  ```
  std::vector<T> v;
  ```

  δημιουργεί ένα κενό vector για αποθήκευση στοιχείων τύπου T.

- Ο κώδικας

```
std::vector<T> v1;
std::vector<T> v2{v1};
std::vector<T> v3{std::move(v1)};
```

δημιουργεί το vector v2 με *αντιγραφή* του v1 και το vector v3 με *μετακίνηση* του v1. Η δημιουργία με μετακίνηση είναι ταχύτατη καθώς μεταφέρεται μόνο η εσωτερική αναπαράσταση του v1. Αυτή είναι συνήθως ένας δείκτης στο πρώτο από τα αποθηκευμένα στοιχεία και ένας ακέραιος για το πλήθος τους.

- Ο κώδικας

```
T a1, a2, a3, ...;
std::vector<T> v{a1, a2, a3, ... };
```

δημιουργεί ένα vector με πλήθος και τιμές στοιχείων που προσδιορίζονται από τη λίστα που ακολουθεί το όνομά του.

- Αν beg και end είναι δύο input iterators *στον ίδιο container ή ροή εισόδου, με τον beg να μη «δείχνει» μετά τον end*, μπορούμε να δημιουργήσουμε ένα vector κατασκευάζοντας τα στοιχεία του από τα στοιχεία στο διάστημα [beg,end) με την εντολή

```
std::vector<T> v{beg,end};
```

Εκτός από τους κοινούς μηχανισμούς ορισμού, ένα vector μπορεί να δημιουργηθεί ως εξής:

- Η εντολή

```
std::vector<T> v(N);
```

ορίζει το v ως ένα vector με N θέσεις για αντικείμενα τύπου T. Το πλήθος N πρέπει να είναι ακέραια ποσότητα, σταθερή ή μεταβλητή. Όλα τα στοιχεία παίρνουν αρχική τιμή `T{}`, την προκαθορισμένη για τον τύπο T[2]. Αν ο T είναι ενσωματωμένος τύπος, η προκαθορισμένη τιμή είναι το 0 με κατάλληλη μετατροπή.

- Η εντολή

```
std::vector<T> v(N, elem);
```

ορίζει το v ως ένα vector με N αντίγραφα του αντικειμένου elem. Είναι απαραίτητο βέβαια, το elem να είναι τύπου T ή να μπορεί να μετατραπεί σε αυτό τον τύπο[3].

[2] αρκεί να υπάρχουν και να είναι προσβάσιμοι ο default (§14.5.4) και ο copy (§14.5.2) constructor, αλλιώς ο ορισμός του vector είναι λάθος.

[3] και να είναι προσβάσιμος ο copy constructor (§14.5.2) του τύπου T.

Διαχείριση μνήμης

Ο container `std::vector<>` περιλαμβάνει ως μέλη τρεις συναρτήσεις σχετικές με το μέγεθος ενός αντικειμένου του, επιπλέον των κοινών `size()`, `empty()` και `max_size()` που περιγράψαμε ήδη:

- Η `capacity()` δεν δέχεται όρισμα· επιστρέφει το μέγιστο δυνατό πλήθος στοιχείων που μπορεί να αποθηκευθεί στο vector για το οποίο καλείται, χωρίς να χρειαστεί μετακίνησή του σε άλλο τμήμα μνήμης· μας δίνει, δηλαδή, το διαθέσιμο χώρο τη στιγμή της κλήσης. Η ποσότητα που επιστρέφεται είναι τύπου `size_type`, όπως αυτός ορίζεται στο συγκεκριμένο τύπο vector.

- Η `reserve()` δεσμεύει τόσες συνεχόμενες θέσεις όσες καθορίζει το όρισμά της (ακέραιος τύπου `size_type`). Αν αυτό είναι μεγαλύτερο από τον τρέχοντα διαθέσιμο χώρο προκαλεί τη μεταφορά του vector σε κατάλληλο τμήμα μνήμης, αλλιώς δεν κάνει τίποτε. Η συγκεκριμένη συνάρτηση δε δίνει αρχική τιμή στις δεσμευόμενες θέσεις και ούτε αλλάζει το μέγεθος του vector. Όταν, επομένως, γνωρίζουμε το πλήθος N των στοιχείων τύπου T που θα έχει ένα νέο vector αλλά όχι ακόμα τις τιμές τους, είναι προτιμότερο να το δηλώσουμε ως εξής

  ```
  std::vector<T> v;
  v.reserve(N);
  ```

  και να ακολουθήσουν N κλήσεις της `push_back()` (που θα συναντήσουμε αμέσως παρακάτω), παρά ως

  ```
  std::vector<T> v(N);
  ```

  και να κάνουμε εκχώρηση στα στοιχεία `v[0]`, `v[1]`, ..., `v[N-1]`. Με τον προτεινόμενο τρόπο αποφεύγουμε την αυτόματη απόδοση αρχικής τιμής στα στοιχεία.

  Όπως αναφέραμε, η μεταφορά ενός vector σε νέο τμήμα μνήμης, η οποία μπορεί να προκληθεί με κλήση της `reserve()`, ακυρώνει όλους τους δείκτες, iterators και αναφορές στο vector.

- Η `shrink_to_fit()` ζητά (αλλά δεν εξασφαλίζει) τη μείωση της δεσμευμένης μνήμης ώστε να χωρά ακριβώς τα τρέχοντα στοιχεία του vector για το οποίο καλείται. Ζητά δηλαδή από τον compiler την εξίσωση του `capacity()` με το `size()` για το vector για το οποίο καλείται. Αν τελικά γίνει η εξίσωση, ακυρώνονται οι δείκτες, iterators και αναφορές στο vector. Η συνάρτηση δεν δέχεται ορίσματα και δεν επιστρέφει τίποτε.

Προσθήκη στοιχείων

Περιγράψαμε ήδη στο §11.3 τους κοινούς μηχανισμούς με τους οποίους προσθέτουμε στοιχεία σε ένα οποιοδήποτε container μετά τον ορισμό του. Θα τους επαναλάβουμε με συντομία εδώ και θα τους συμπληρώσουμε για vector.

- Μπορούμε να εισαγάγουμε στοιχεία σε ένα vector διαγράφοντας τα υπάρχοντα με αντιγραφή ή μετακίνηση άλλου vector ίδιου τύπου. Στον κώδικα

```
std::vector<T> v1, v2, v3;
 ...
v2 = v1;
v3 = std::move(v1);
```

  τα στοιχεία του v1 *αντιγράφονται* στο v2 καταστρέφοντας όσα υπήρχαν σε αυτόν. Κατόπιν το v1 *μετακινείται* στο v3.

- Εισαγωγή και αντικατάσταση των παλαιών στοιχείων, και πιθανή μεταβολή του πλήθους τους, γίνεται και με εκχώρηση λίστας τιμών:

```
std::vector<T> v;
T a1, a2, a3, ...;
v = {a1,a2,a3, ...};
```

- Η εναλλαγή στοιχείων με άλλο vector ίδιου τύπου, με κλήση είτε της συνάρτησης–μέλους `swap()` είτε της `std::swap()`, τροποποιεί τα στοιχεία ενός vector:

```
std::vector<T> v1, v2, v3;
 ...
v1.swap(v2);
std::swap(v3,v1);
```

- Με την κλήση της συνάρτησης–μέλους `insert()` για ένα vector προκαλούμε εισαγωγή σε επιλεγμένη θέση, ενός νέου στοιχείου, με αντιγραφή ή μετακίνηση:

```
v.insert(pos, elem);
```

  Η συγκεκριμένη κλήση αντιγράφει ή μετακινεί το elem ως νέο στοιχείο στο vector v, *πριν* τη θέση που «δείχνει» ο const_iterator pos. Επιστρέφει iterator στη θέση του νέου στοιχείου.

- Προσθήκη ή διαγραφή στοιχείων γίνεται και με κατάλληλους αλγόριθμους.

Επιπλέον των παραπάνω μηχανισμών:

- Τρεις παραλλαγές της συνάρτησης–μέλους `insert()` προκαλούν την ταυτόχρονη εισαγωγή ενός πλήθους νέων στοιχείων:

```
v.insert(pos, N, elem);
v.insert(pos, beg, end);
v.insert(pos, {a1, a2, a3, ...});
```

  Η πρώτη μορφή εισάγει N αντίγραφα του elem, *πριν* τη θέση που «δείχνει» ο const_iterator pos. Στη δεύτερη μορφή, εισάγονται στο vector v πριν τη θέση

pos, αντίγραφα των στοιχείων στο διάστημα των input iterators [beg,end). Στην τρίτη μορφή, εισάγονται στο vector v πριν τη θέση pos, αντίγραφα των στοιχείων της λίστας στο δεύτερο όρισμα. Η συνάρτηση και στις τρεις μορφές επιστρέφει iterator στο πρώτο νέο στοιχείο ή το pos, αν το πλήθος των εισαγόμενων στοιχείων είναι 0 (αν `N==0` ή `beg==end` ή η λίστα είναι κενή).

Η εισαγωγή πολλών νέων στοιχείων γίνεται πιο γρήγορα με τις συγκεκριμένες παραλλαγές παρά με πολλαπλές κλήσεις της `insert()` για ένα στοιχείο.

- Η εντολή

  ```
  v.push_back(elem);
  ```

  είναι ο πιο αποτελεσματικός και συνηθέστερα χρησιμοποιούμενος τρόπος για την εισαγωγή ενός στοιχείου στο τέλος του vector. Αντιγράφει ή μετακινεί στη νέα θέση το elem. Δεν επιστρέφει τιμή. Η εντολή `v.push_back(elem);` πρακτικά ισοδυναμεί με την `v.insert(v.cend(),elem);`.

- Με τη συνάρτηση–μέλος `emplace()` δημιουργείται προσωρινά ένα αντικείμενο κατάλληλου τύπου, το οποίο κατόπιν εισάγεται με μετακίνηση στο vector για το οποίο καλείται. Η συνάρτηση δέχεται ως πρώτο όρισμα ένα const_iterator· η εισαγωγή θα γίνει πριν από τη θέση που δείχνει αυτός. Ως δεύτερο, τρίτο, ... όρισμα δέχεται μία ή περισσότερες ποσότητες που τις χρησιμοποιεί για να δημιουργήσει το αντικείμενο προς εισαγωγή. Έτσι, στον κώδικα

  ```
  std::vector<std::complex<double>> v;
  v.emplace(v.cend(), 3.1, 2.1);
  ```

  τοποθετείται ως πρώτο στοιχείο στο vector v ο μιγαδικός $3.1 + \mathrm{i}2.1$.

  Η συνάρτηση επιστρέφει iterator στη θέση του νέου στοιχείου.

- Η συνάρτηση–μέλος `emplace_back()` δημιουργεί και εισάγει στοιχείο στο τέλος του vector για το οποίο καλείται. Η κλήση `v.emplace_back(a,b,c,...);` ισοδυναμεί πρακτικά με `v.emplace(v.cend(), a,b,c,...);`. Η συνάρτηση δεν επιστρέφει τίποτε.

- Καταστροφή όλων των στοιχείων ενός vector και αντικατάστασή τους από νέα μπορεί επίσης να γίνει με τη συνάρτηση–μέλος `assign()`. Η συγκεκριμένη έχει τρεις παραλλαγές ως προς τα ορίσματά της:

  ```
  v.assign(N,elem);
  v.assign(beg, end);
  v.assign({a1,a2,a3,...});
  ```

  Η πρώτη μορφή εισάγει N αντίγραφα του elem στο vector v. Το πλήθος N είναι φυσικά ακέραιο. Στη δεύτερη μορφή, αντικαθιστά τα στοιχεία του v με αντίγραφα των στοιχείων στο διάστημα [beg,end). Στην τρίτη μορφή, η

κλήση εισάγει τις τιμές της λίστας. Όταν χρειάζεται γίνεται μετατροπή των εισαγόμενων στοιχείων στον τύπο των στοιχείων του vector.

Η συνάρτηση δεν επιστρέφει τίποτε και στις τρεις μορφές της.

- Η κλήση της συνάρτησης–μέλους `resize()` προκαλεί εισαγωγή ή διαγραφή στοιχείων. Δεν επιστρέφει τίποτε. Η εντολή

  ```
  v.resize(N);
  ```

  αλλάζει το πλήθος των στοιχείων του v σε N, διαγράφοντας από το τέλος ή προσθέτοντας εκεί στοιχεία. Στην τελευταία περίπτωση τα νέα στοιχεία που εισάγονται έχουν την προκαθορισμένη τιμή για τον τύπο τους. Με δεύτερο όρισμα, δηλαδή με την εντολή

  ```
  v.resize(N,elem);
  ```

  τα τυχόν νέα στοιχεία είναι αντίγραφα του elem.

Προσέξτε ότι κάθε εισαγωγή στοιχείου ακυρώνει τις αναφορές, τους δείκτες και τους iterators σε επόμενα στοιχεία (και τους iterators τέλους). Αν η εισαγωγή εξαντλήσει το διαθέσιμο χώρο για ένα vector, προκαλείται μετακίνηση όλων των στοιχείων σε μεγαλύτερο τμήμα μνήμης και όλες οι αναφορές, δείκτες και iterators ακυρώνονται.

Παρατηρήστε τη συνέπεια της ακύρωσης των iterators που δείχνουν στις επόμενες θέσεις από ένα νεοεισαχθέν στοιχείο στον παρακάτω κώδικα. Η κλήση της `reserve()` γίνεται για να εξασφαλιστεί ότι δεν θα ακυρώνονται οι iterators λόγω συνολικής μετακίνησης.

```
std::vector<int> v; // empty vector
v.reserve(4);

v.insert(v.cend(),1); // v is {1}
v.insert(v.cend(),2); // v is {1,2}

auto end = v.cend();

v.insert(end, 3);   // v is {1,2,3}
v.insert(end, 4);   // v is ? v may be {1,2,4,3}
```

Ο const_iterator end δεν δείχνει πάντα στο τέλος του container.

Διαγραφή στοιχείων

Διαγραφή στοιχείων ενός vector γίνεται:

- Με την κοινή συνάρτηση–μέλος `clear()`:

  ```
  v.clear();
  ```

Η συγκεκριμένη εντολή καταστρέφει όλα τα στοιχεία του v.

- Με την κοινή συνάρτηση–μέλος `erase()`. Όπως ήδη περιγράψαμε, κατά την κλήση της μπορεί να έχει είτε ως μόνο όρισμα ένα const_iterator είτε δύο const_iterators ως ορίσματα:

  ```
  v.erase(pos);
  v.erase(beg,end);
  ```

  Στην πρώτη μορφή διαγράφεται το στοιχείο στη θέση pos και στη δεύτερη τα στοιχεία στο διάστημα [beg,end). Και στις δύο περιπτώσεις η επιστρεφόμενη τιμή είναι iterator στην επόμενη θέση από το τελευταίο διαγραμμένο στοιχείο.

- Με τη χρήση της `resize()` που παρουσιάστηκε παραπάνω.

- Σε αντιστοιχία με την `push_back()`, η συνάρτηση–μέλος `pop_back()`, χωρίς όρισμα και επιστρεφόμενη τιμή, διαγράφει το τελευταίο στοιχείο (το οποίο πρέπει να υπάρχει, δηλαδή το vector να μην είναι κενό):

  ```
  v.pop_back();
  ```

  Η συγκεκριμένη κλήση είναι ο πιο γρήγορος τρόπος διαγραφής στοιχείου και ισοδυναμεί πρακτικά με την `v.erase(v.cend()-1);`.

Προσέξτε ότι κάθε διαγραφή στοιχείου ακυρώνει όλες τις αναφορές, τους δείκτες και τους iterators σε επόμενες θέσεις.

Προσπέλαση στοιχείων

Προσπέλαση και επομένως, δυνατότητα μεταβολής των μεμονωμένων στοιχείων ενός `vector<>` γίνεται με τους ίδιους μηχανισμούς που περιγράψαμε στο `array<>` (εκτός από το `std::get()`), ως εξής:

- Με τη χρήση ακέραιου δείκτη μεταξύ των αγκυλών '`[]`'.

- Με τη χρήση της συνάρτησης–μέλους `at()` με ακέραιο όρισμα. Το πρώτο, δεύτερο, τρίτο,...στοιχείο ενός `std::vector<>` με όνομα v είναι το `v.at[0]`, `v.at[1]`, `v.at[2]`,.... Προσέξτε πως η διαφορά από την προηγούμενη περίπτωση είναι ότι αν το όρισμα είναι έξω από το διάστημα `[0:v.size())` εγείρεται εξαίρεση του τύπου `std::out_of_range`. Στην περίπτωση που αυτή δεν συλληφθεί, διακόπτεται η εκτέλεση του προγράμματος. Ο απαιτούμενος έλεγχος στην τιμή του δείκτη έχει ως αποτέλεσμα να είναι πιο αργή η πρόσβαση στο στοιχείο απ' ό,τι με τις αγκύλες.

- Με τις συναρτήσεις–μέλη `front()` και `back()`. Αυτές επιστρέφουν αναφορά στο πρώτο και στο τελευταίο στοιχείο αντίστοιχα, του vector για το οποίο καλούνται. Εννοείται ότι το vector δεν θα είναι κενό.

- Με τη δράση του τελεστή '`*`' σε όνομα iterator. Φυσικά, αν ο iterator είναι σε σταθερό στοιχείο (π.χ. `const_iterator`) δεν μπορούμε να μεταβάλουμε την τιμή που «δείχνει» αυτός αλλά μόνο να τη διαβάσουμε.

- Με τη δράση του τελεστή '`*`' σε δείκτη σε κάποιο στοιχείο του vector. Επιτρέπεται η μετακίνηση του δείκτη σε άλλη θέση. Εναλλακτικά, μπορούμε να χρησιμοποιήσουμε τη συνάρτηση–μέλος `data()` που επιστρέφει δείκτη στο πρώτο στοιχείο του vector για το οποίο καλείται.

Παράδειγμα

Δημιουργία, αντιγραφή και προσπέλαση στοιχείων ενός `std::vector<>` μπορεί να γίνει ως ακολούθως:

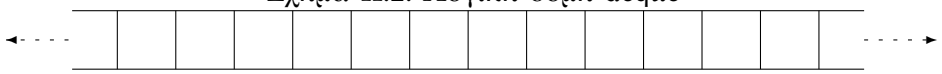
```
#include <iostream>
#include <vector>

int
main()
{
    std::vector<double> v(9);
    // v = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0}

    // assign some values
    for (int i{0}; i < v.size(); ++i) {
        v[i] = 4.0 * i*i;
    }

    auto v2 = v; // v2 is a copy of v

    // append more values to v2
    auto siz = v2.size();
    for (auto i = siz; i < 2*siz; ++i) {
        v2.push_back(4.0*i*i);
    }

    // print v2
    std::cout << "v2␣is\t";
    for (auto const & x : v2) {
        std::cout << x << '␣';
    }
    std::cout << '\n';
}
```

Παρατήρηση: Η εξειδίκευση `std::vector<bool>` έχει αρκετούς περιορισμούς στη χρήση της (δεν έχει τα στοιχεία της συνεχόμενα στη μνήμη, δεν είναι container και δεν περιέχει `bool`!). Προτιμήστε το `std::deque<bool>` αν χρειαστείτε να αποθηκεύσετε `bool` ή συμβουλευτείτε τη βιβλιογραφία (π.χ. [2] §7.3.6, [4] σελ. 41–45, [3] item 18) αν τη χρειαστείτε.

11.5.3 deque

Ποσότητα τύπου `std::deque<>` (double-ended queue) αποθηκεύει ένα σύνολο στοιχείων ίδιου τύπου με τη σειρά που τοποθετούνται κατά την εισαγωγή τους και επιτρέπει την τυχαία προσπέλαση σε αυτά. Ο τύπος `std::deque<>` είναι σε μεγάλο βαθμό όμοιος στις δυνατότητες με το `std::vector<>`. Όμως, σε αντίθεση με το vector, η αποθήκευση των στοιχείων δεν γίνεται σε διαδοχικές θέσεις μνήμης: εσωτερικά, τα στοιχεία οργανώνονται συνήθως σε ανεξάρτητα τμήματα που συνδέονται μεταξύ τους. Αυτό έχει ως συνέπεια το deque να προσφέρει πιο αργή πρόσβαση στα στοιχεία απ᾽ ό,τι ένα vector αλλά και το πλεονέκτημα να μη χρειάζεται μετακίνηση πολλών στοιχείων κατά την εισαγωγή μακριά από τα άκρα. Έτσι, ενώ το vector επιτρέπει ταχύτατη εισαγωγή ή διαγραφή στοιχείων μόνο στο τέλος του, ένα deque παρέχει αυτή τη δυνατότητα και στις δύο άκρες του. Η εισαγωγή σε σημείο μακριά από τα άκρα είναι αργή, λίγο ή πολύ. Επομένως, ένα deque είναι

Σχήμα 11.2: Λογική δομή deque



προτιμότερο ενός vector όταν χρειαζόμαστε δυναμική διαχείριση μνήμης με γρήγορη εισαγωγή ή διαγραφή στοιχείων στα δύο άκρα.

Η κατακερματισμένη μνήμη σε ένα `std::deque<>` δεν επιτρέπει στο χρήστη του να έχει τον έλεγχο που έχει σε ένα vector. Η κλάση δεν παρέχει τις συναρτήσεις–μέλη `capacity()` και `reserve()` που συναντήσαμε στο vector. Σημειώστε ότι ο κατακερματισμός της μνήμης είναι εσωτερικός· κατά το χειρισμό του μέσω iterators το `deque` συμπεριφέρεται σαν να αποθηκεύει τα στοιχεία διαδοχικά στη μνήμη. Επομένως, ανεξάρτητα από τις λεπτομέρειες της υλοποίησης του deque, η αύξηση ενός iterator κατά ένα μας μεταφέρει στο αμέσως επόμενο στοιχείο, όπου και να βρίσκεται αυτό στη μνήμη.

Καθώς τα στοιχεία ενός deque δεν αποθηκεύονται σε διαδοχικές θέσεις μνήμης, η προσπέλασή τους με δείκτες που μετακινούνται με αριθμητική δεικτών (§2.18.2) δεν είναι εφικτή. Γι᾽ αυτό το λόγο δεν παρέχεται η συνάρτηση–μέλος `data()` που έχει ο vector.

Με την εξαίρεση των `capacity()`, `reserve()`, `data()` και του μηχανισμού προσπέλασης με δείκτες, όλες οι συναρτήσεις–μέλη, οι τρόποι δημιουργίας και οι μηχανισμοί προσθήκης, διαγραφής ή προσπέλασης στοιχείων που παρουσιάστηκαν στο vector παρέχονται και από το deque. Επιπλέον, υπάρχει η αναμενόμενη συμπλήρωση με τις συναρτήσεις–μέλη

- `push_front()` (εισαγωγή στοιχείου στην αρχή με μετακίνηση ή αντιγραφή),
- `pop_front()` (διαγραφή του πρώτου στοιχείου),
- `emplace_front()` (δημιουργία νέου στοιχείου στην αρχή),

σε πλήρη αντιστοιχία με τις `push_back()`, `pop_back()`, `emplace_back()`.

Προσέξτε ότι καθώς δεν μπορούμε να γνωρίζουμε πότε θα χρειαστεί μετακίνηση στοιχείων (και ποιων), θα πρέπει να θεωρούμε ότι η εισαγωγή ή διαγραφή σε οποιοδήποτε σημείο ενός deque εκτός από την αρχή ή το τέλος του, ακυρώνει τις αναφορές και τους iterators σε όλα τα στοιχεία του. Η εισαγωγή στην αρχή ή το τέλος, διατηρεί όλες τις αναφορές αλλά όχι τους iterators. Η διαγραφή του πρώτου ή του τελευταίου στοιχείου ακυρώνει τις αναφορές και τους iterators σε αυτό. Επιπλέον, η διαγραφή οποιουδήποτε στοιχείου εκτός από το πρώτο ακυρώνει και τους iterators τέλους.

Ο τύπος `std::deque<>` παρέχει όλους τους κοινούς τύπους, τις συναρτήσεις–μέλη και τους τελεστές σύγκρισης που παρουσιάσαμε στο §11.4. Η χρήση του προϋποθέτει τη συμπερίληψη του `<deque>`. Οι iterators που παρέχει ένα deque, ορθής και ανάστροφης φοράς, είναι τυχαίας προσπέλασης (random iterators).

Παράδειγμα

Ένα παράδειγμα ορισμού και χρήσης του deque είναι το ακόλουθο. Δημιουργούμε ένα deque, δίνουμε τιμές σε αυτό και κατόπιν διαγράφουμε όσες είναι πολλαπλάσια του 3. Στο τέλος, τυπώνουμε τις τιμές που απομένουν.

```
#include <deque>
#include <iostream>

int
main()
{
    std::deque<int> d{0}; //deque with one element, 0.
    //fill with values:
    for (int i{1}; i < 100; ++i) {
        d.push_back(i);
        d.push_front(-i);
    }
    // d is {-99,-98, ..., -1, 0, 1, ..., 98, 99}

    // erase all multiples of 3:
    for (auto it = d.begin(); it != d.end(); ) {
        if (*it % 3 == 0) {
            d.erase(it);   // 'it' undefined
            it = d.begin();  // 'it' defined; restart.
```

```
        } else {
            ++it;
        }
    }

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

Προσέξτε τον κώδικα που χρησιμοποιήθηκε για τον εντοπισμό και διαγραφή των επιθυμητών τιμών. Δεν είναι ο πιο γρήγορος αλλά αποφεύγει να χρησιμοποιήσει ακυρωμένο iterator. Αντίθετα, ο παρακάτω κώδικας

```
for (auto it = d.begin(); it != d.end(); ++it) {
    if (*it % 3 == 0) {
         d.erase(it);   // it undefined
    }
}
```

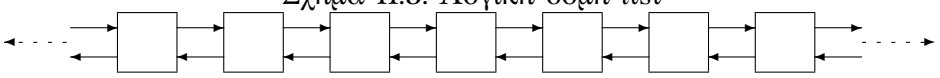
που πιθανόν θα ήταν η πρώτη απόπειρά μας, είναι λάθος καθώς αυξάνει ένα iterator που μετά τη διαγραφή δε δείχνει στον container.

Η χρήση κατάλληλων αλγορίθμων που θα παρουσιάσουμε στο Κεφάλαιο 12, είναι προτιμότερη από δικό μας κώδικα για συνήθεις διαδικασίες όπως η διαγραφή συγκεκριμένων στοιχείων.

11.5.4 list

Ο `std::list<>` είναι ακόμα ένας sequence container που παρέχει η Standard Library. Και αυτός αποθηκεύει στοιχεία ίδιου τύπου με τη σειρά που εισάγονται σε αυτόν. Όμως, τα στοιχεία δεν βρίσκονται σε διαδοχικές θέσεις μνήμης. Η υλοποίηση του `list<>` είναι διαφορετική απ᾽ ό,τι των `vector<>` και `deque<>`. Στο `list<>` τα στοιχεία αποθηκεύονται σε ξεχωριστά τμήματα μνήμης το καθένα, μαζί με την πληροφορία (πιθανόν σε μορφή δεικτών) για τη θέση του επόμενου και του προηγούμενου στοιχείου, Σχήμα 11.3.

Σχήμα 11.3: Λογική δομή list

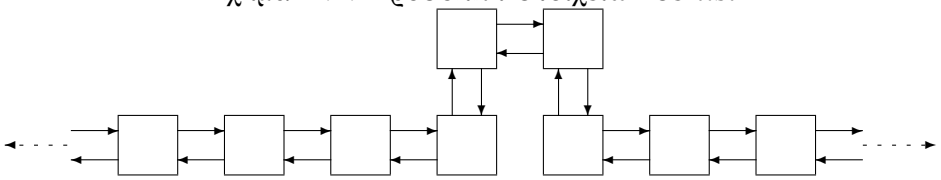


Το `list<>`, έχοντας τη συγκεκριμένη εσωτερική δομή, παρουσιάζει το μειονέκτημα έναντι των `vector<>` και `deque<>` ότι δεν επιτρέπει την τυχαία προσπέλαση των στοιχείων του, δηλαδή τη μετακίνηση στον ίδιο χρόνο σε οποιοδήποτε

από αυτά. Αντίθετα, πρέπει να μετακινούμαστε, ξεκινώντας από μία θέση, διαδοχικά από το ένα στοιχείο στο άλλο έως ότου φτάσουμε στη ζητούμενη θέση, χρησιμοποιώντας την πληροφορία για τη θέση του επόμενου στοιχείου στην κατεύθυνση κίνησης. Η διαδικασία αυτή βέβαια απαιτεί χρόνο ανάλογο του πλήθους των στοιχείων που διατρέχουμε.

Η δομή του `list<>` του δίνει ένα σημαντικό πλεονέκτημα έναντι άλλων sequence containers: η προσθήκη ή διαγραφή στοιχείων σε οποιοδήποτε σημείο του list γίνεται σε ίδιο χρόνο, καθώς απαιτεί αλλαγές δεικτών (Σχήμα 11.4) μόνο, της πληροφορίας δηλαδή του επόμενου και προηγούμενου στοιχείου. Το ότι δε γίνεται

Σχήμα 11.4: Προσθήκη στοιχείων σε list



μετακίνηση στοιχείων έχει ως συνέπεια πως οι αναφορές, οι δείκτες και οι iterators σε άλλα στοιχεία του list δε χάνονται κατά την εισαγωγή ή διαγραφή ενός στοιχείου.

Ο τύπος `std::list<>` παρέχει όλους τους κοινούς τύπους, τις συναρτήσεις–μέλη και τους τελεστές σύγκρισης που παρουσιάσαμε στο §11.4. Η χρήση του προϋποθέτει τη συμπερίληψη του header `<list>`. Οι iterators που παρέχει ένα list, ορθής και ανάστροφης φοράς, είναι δύο κατευθύνσεων (bidirectional iterators).

Ορισμός

Για τον ορισμό με πιθανή απόδοση αρχικής τιμής ενός `std::list<>` ισχύουν οι μηχανισμοί που είδαμε στους `vector<>` και `deque<>`. Συγκεκριμένα, μπορούμε να χρησιμοποιήσουμε τους κοινούς μηχανισμούς για οποιοδήποτε container καθώς και τη δημιουργία με αντίγραφα ποσότητας (ή της προκαθορισμένης τιμής). Καθώς οι μηχανισμοί παρουσιάστηκαν για το vector, τους επαναλαμβάνουμε για list συνοπτικά:

- Η εντολή

  ```
  std::list<T> c;
  ```

  δημιουργεί ένα κενό list για αποθήκευση στοιχείων τύπου T.

- Ο κώδικας

  ```
  std::list<T> c1;
  std::list<T> c2{c1};
  std::list<T> c3{std::move(c1)};
  ```

δημιουργεί το list c2 με *αντιγραφή* του c1 και το list c3 με *μετακίνηση* του c1. Η δημιουργία με μετακίνηση είναι ταχύτατη καθώς μεταφέρεται μόνο η εσωτερική αναπαράσταση του c1 και όχι ένα-ένα τα στοιχεία.

- Ο κώδικας

  ```
  T a1, a2, a3, ...;
  std::list<T> c{a1, a2, a3, ... };
  ```

  δημιουργεί ένα list με πλήθος και τιμές στοιχείων που προσδιορίζονται από τη λίστα που ακολουθεί το όνομά του.

- Αν beg και end είναι δύο input iterators *στον ίδιο container ή ροή εισόδου, με τον beg να μη «δείχνει» μετά τον end*, μπορούμε να δημιουργήσουμε ένα list με αντίγραφα των στοιχείων στο διάστημα [beg,end) με την εντολή

  ```
  std::list<T> c{beg,end};
  ```

- Η εντολή

  ```
  std::list<T> c(N);
  ```

  ορίζει το c ως ένα list με N θέσεις για αντικείμενα τύπου T.

- Η εντολή

  ```
  std::list<T> c(N, elem);
  ```

  ορίζει το c ως ένα list με N αντίγραφα του αντικειμένου elem, αφού αυτό μετατραπεί στον τύπο T.

Διαχείριση μνήμης

Ένα `list<>` παρέχει τις κοινές συναρτήσεις–μέλη `size()`, `empty()`, `max_size()` που περιγράψαμε προηγουμένως (§11.4).

Προσθήκη στοιχείων

Από τον container `list<>` παρέχονται οι μηχανισμοί που ισχύουν και στο `deque<>` για την προσθήκη στοιχείων μετά τον ορισμό του:

- Μπορούμε να εισαγάγουμε στοιχεία σε ένα list διαγράφοντας τα υπάρχοντα με αντιγραφή ή μετακίνηση άλλου list ίδιου τύπου:

  ```
  std::list<T> c1, c2, c3;
   ...
  c2 = c1;
  c3 = std::move(c1);
  ```

- Εισαγωγή και αντικατάσταση των παλαιών στοιχείων σε list, και πιθανή μεταβολή του πλήθους τους, γίνεται και με εκχώρηση λίστας τιμών:

```
std::list<T> c;
T a1, a2, a3, ...;
c = {a1,a2,a3, ...};
```

- Η εναλλαγή στοιχείων με άλλο list ίδιου τύπου, με κλήση είτε της συνάρτησης–μέλους `swap()` είτε της `std::swap()`, τροποποιεί τα στοιχεία ενός list:

```
std::list<T> c1, c2, c3;
 ...
c1.swap(c2);
std::swap(c3,c1);
```

- Με την κλήση της συνάρτησης–μέλους `insert()` για ένα list προκαλούμε εισαγωγή σε επιλεγμένη θέση, ενός ή περισσότερων νέων στοιχείων, με αντιγραφή ή μετακίνηση:

```
c.insert(pos, elem);
c.insert(pos, N, elem);
c.insert(pos, beg, end);
c.insert(pos, {a1, a2, a3, ...});
```

  Το πρώτο όρισμα είναι ένας const_iterator. Η εισαγωγή στοιχείου γίνεται πριν τη θέση που δείχνει αυτός. Η πρώτη εντολή αντιγράφει ή μετακινεί το elem ως νέο στοιχείο στο list c. Οι επόμενες εισάγουν N αντίγραφα του elem, τα στοιχεία στο διάστημα των input iterators [beg,end) και τα στοιχεία μιας λίστας, αντίστοιχα.

  Η συνάρτηση–μέλος `insert()` επιστρέφει iterator στο πρώτο νέο στοιχείο ή το pos, αν το πλήθος των εισαγόμενων στοιχείων είναι 0 (αν `N==0` ή `beg==end` ή η λίστα είναι κενή).

- Καταστροφή όλων των στοιχείων ενός list και αντικατάστασή τους από νέα γίνεται με τη συνάρτηση–μέλος `assign()`:

```
c.assign(N,elem);
c.assign(beg, end);
c.assign({a1,a2,a3,...});
```

  Η πρώτη μορφή εισάγει N αντίγραφα του elem στο list c. Στη δεύτερη μορφή, η κλήση αντικαθιστά τα στοιχεία του c με αντίγραφα των στοιχείων στο διάστημα των input_iterators [beg,end). Στην τρίτη μορφή, η συνάρτηση εισάγει τις τιμές της λίστας. Η συνάρτηση δεν επιστρέφει τίποτε και στις τρεις μορφές της.

- Οι εντολές

```
c.push_front(elem);
c.push_back(elem);
```

αντιγράφουν ή μετακινούν το στοιχείο elem στην αρχή ή στο τέλος του list c. Δεν επιστρέφουν τιμή.

- Με τη συνάρτηση–μέλος `emplace()` δημιουργείται προσωρινά ένα αντικείμενο κατάλληλου τύπου, το οποίο κατόπιν εισάγεται με μετακίνηση στο list για το οποίο καλείται. Η συνάρτηση δέχεται ως πρώτο όρισμα ένα const_iterator· η εισαγωγή θα γίνει πριν από τη θέση που δείχνει αυτός. Ως δεύτερο, τρίτο, ... όρισμα δέχεται μία ή περισσότερες ποσότητες που τις χρησιμοποιεί για να δημιουργήσει το αντικείμενο προς εισαγωγή. Η συνάρτηση επιστρέφει iterator στη θέση του νέου στοιχείου.

- Οι συναρτήσεις–μέλη `emplace_front()` και `emplace_back()` δημιουργούν και εισάγουν στοιχείο στην αρχή ή στο τέλος του list για το οποίο καλούνται. Δεν επιστρέφουν τίποτε.

- Η κλήση της συνάρτησης–μέλους `resize()` προκαλεί εισαγωγή ή διαγραφή στοιχείων. Δεν επιστρέφει τίποτε. Η εντολή

  ```
  v.resize(N);
  ```

  αλλάζει το πλήθος των στοιχείων του v σε N, διαγράφοντας από το τέλος ή προσθέτοντας εκεί στοιχεία. Στην τελευταία περίπτωση τα νέα στοιχεία που εισάγονται έχουν την προκαθορισμένη τιμή για τον τύπο τους. Με δεύτερο όρισμα, δηλαδή με την εντολή

  ```
  v.resize(N,elem);
  ```

  τα τυχόν νέα στοιχεία είναι αντίγραφα του elem.

- Προσθήκη ή διαγραφή στοιχείων γίνεται και με κατάλληλους αλγόριθμους.

Εκτός από τους παραπάνω μηχανισμούς, το list παρέχει και άλλες συναρτήσεις–μέλη που τροποποιούν τα στοιχεία του:

- Η συνάρτηση `splice()` μετακινεί στοιχεία από ένα list σε άλλο. Έχει τρεις παραλλαγές. Σε όλες δέχεται ως πρώτο όρισμα ένα const_iterator· η εισαγωγή θα γίνει πριν από τη θέση που δείχνει αυτός. Ως δεύτερο, δέχεται το list από το οποίο θα γίνει η μετακίνηση, είτε ως αναφορά είτε ως προσωρινή ποσότητα. Οι παραλλαγές διαφοροποιούνται στα επόμενα ορίσματα.

  Η πρώτη μορφή δεν δέχεται τρίτο όρισμα. Μετακινεί όλα τα στοιχεία του δεύτερου ορίσματος στο list για το οποίο κλήθηκε. Με τον κώδικα

  ```
  std::list<T> c1, c2;
  std::list<T>::const_iterator pos; // iterator to c1
   ...
  c1.splice(pos, c2);
  ```

όλα τα στοιχεία του c2 μεταφέρονται στο c1, πριν τη θέση pos, αφήνοντας το c2 κενό. Προφανώς, οι c1, c2 δεν πρέπει να είναι το ίδιο list.

Η δεύτερη μορφή δέχεται ως τρίτο όρισμα ένα iterator σε σταθερό στοιχείο του δεύτερου ορίσματος. Μετακινεί το στοιχείο που δείχνει αυτός. Με τον κώδικα

```
std::list<T> c1, c2;
std::list<T>::const_iterator c1pos; // iterator to c1
std::list<T>::const_iterator c2pos; // iterator to c2
 ...
c1.splice(c1pos, c2, c2pos);
```

το στοιχείο στη θέση c2pos του c2 μετακινείται στο c1, πριν το c1pos, και διαγράφεται από το c2. Οι c1, c2 μπορούν να είναι ο ίδιος list.

Η τρίτη μορφή δέχεται ως τρίτο και τέταρτο όρισμα, δύο const_iterators στο δεύτερο όρισμα. Μετακινεί τα στοιχεία μεταξύ αυτών. Με τον κώδικα

```
std::list<T> c1, c2;
std::list<T>::const_iterator c1pos; // iterator to c1
std::list<T>::const_iterator c2beg; // iterator to c2
std::list<T>::const_iterator c2end; // iterator to c2
 ...
c1.splice(c1pos, c2, c2beg, c2end);
```

μετακινούνται, πριν τη θέση με iterator c1pos, τα στοιχεία του c2 στο διάστημα [c2beg, c2end). Οι c1, c2 μπορούν να είναι ο ίδιος list.

- Η συνάρτηση `merge()` μετακινεί στοιχεία από ένα *ταξινομημένο* list σε άλλο list, επίσης *ταξινομημένο*. Το list από το οποίο μετακινούνται τα στοιχεία απομένει κενό. Έχει δύο παραλλαγές:

  Ο κώδικας

  ```
  std::list<T> c1, c2;
   ...
  c1.merge(c2);
  ```

  μετακινεί τα στοιχεία της (ταξινομημένης) c2 στην (ταξινομημένη) c1 με τέτοιο τρόπο ώστε, μετά τη συγχώνευση, η c1 να είναι πάλι ταξινομημένη.

  Ο κώδικας

  ```
  std::list<T> c1, c2;
   ...
  c1.merge(c2, comp);
  ```

  κάνει το ίδιο με τον προηγούμενο, όμως, θεωρεί ότι η ταξινόμηση κάθε list έγινε με το κριτήριο `comp()` και με αυτό το κριτήριο γίνεται η συγχώνευση

της c2 στη c1, ώστε να προκύψει η c1 ταξινομημένη. Το `comp()` είναι ένα αντικείμενο–συνάρτηση (§9.3) ή μία συνάρτηση λάμδα (§9.3.1) ή μία συνήθης συνάρτηση. Δέχεται δύο ορίσματα με τύπο τον τύπο των στοιχείων του list, και επιστρέφει λογική τιμή, **`true`** ή **`false`**, αν το πρώτο όρισμά του είναι «μικρότερο» (ό,τι κι αν σημαίνει αυτό) από το δεύτερο ή όχι.

Διαγραφή στοιχείων

Διαγραφή στοιχείων από ένα list c, γίνεται με τις συναρτήσεις που έχει και το deque, ως εξής:

- Με την κοινή συνάρτηση–μέλος `clear()`:

  ```
  c.clear();
  ```

  Η συγκεκριμένη εντολή καταστρέφει όλα τα στοιχεία του c.

- Με την κοινή συνάρτηση–μέλος `erase()`, είτε με ένα είτε με δύο ορίσματα τύπου const_iterator:

  ```
  c.erase(pos);
  c.erase(beg,end);
  ```

  Στην πρώτη μορφή διαγράφεται το στοιχείο στη θέση pos και στη δεύτερη τα στοιχεία στο διάστημα [beg,end). Και στις δύο περιπτώσεις η επιστρεφόμενη τιμή είναι iterator στην επόμενη θέση από το τελευταίο διαγραμμένο στοιχείο.

- Με τη χρήση της `resize()` που παρουσιάστηκε παραπάνω.

- Με τις συναρτήσεις–μέλη `pop_front()` και `pop_back()`. Αυτές διαγράφουν το πρώτο ή το τελευταίο στοιχείο αντίστοιχα, ενός μη κενού list. Δεν επιστρέφουν τίποτε.

Επιπλέον, το list παρέχει και άλλους μηχανισμούς διαγραφής:

- Η συνάρτηση–μέλος `remove()` έχει δύο παραλλαγές:

  ```
  c.remove(a);
  c.remove_if(op);
  ```

  Στην πρώτη μορφή, η κλήση διαγράφει όλα τα στοιχεία του list c που είναι ίσα με a. Στη δεύτερη μορφή της, η `remove_if()` δέχεται ένα αντικείμενο–συνάρτηση (§9.3) ή μία συνάρτηση λάμδα (§9.3.1) ή μία συνήθη συνάρτηση, `op()`. Το `op()` δέχεται ένα όρισμα με τύπο τον τύπο των στοιχείων του list, και επιστρέφει λογική τιμή, **`true`** ή **`false`**. Η `remove_if()` δρα το `op()` σε όλα τα στοιχεία του list. Διαγράφει όλα τα στοιχεία για τα οποία το `op()` επιστρέφει **`true`**.

  Η συνάρτηση δεν επιστρέφει τιμή.

Παρατηρήστε τη διαφορά τής `remove()` από τη συνάρτηση–μέλος `erase()`: Η `erase()` διαγράφει το στοιχείο που δείχνει ένας iterator ή τα στοιχεία ενός διαστήματος iterators. Η `remove()` διαγράφει στοιχεία με συγκεκριμένη τιμή ή που ικανοποιούν συγκεκριμένο κριτήριο.

- Η συνάρτηση–μέλος `unique()` έχει δύο μορφές

```
c.unique();
c.unique(comp);
```

Στην πρώτη εντολή, η συνάρτηση εντοπίζει στο list c, ομάδες *διαδοχικών* στοιχείων που έχουν ίδια τιμή. Διαγράφει όλα τα στοιχεία εκτός από το πρώτο σε κάθε ομάδα.

Στη δεύτερη μορφή, η συνάρτηση δέχεται ένα αντικείμενο–συνάρτηση (§9.3) ή μία συνάρτηση λάμδα (§9.3.1) ή μία συνήθη συνάρτηση, `comp()`. Το `comp()` δέχεται δύο ορίσματα με τύπο τον τύπο των στοιχείων του list, και επιστρέφει λογική τιμή.

Η `unique()` δρα το `comp()` σε όλα τα διαδοχικά ζεύγη στοιχείων του list. Σε όσα το `comp()` επιστρέφει **`true`**, διαγράφει το δεύτερο στοιχείο.

Η συνάρτηση δεν επιστρέφει τιμή.

Προσπέλαση στοιχείων

Προσπέλαση στοιχείων ενός list γίνεται ως εξής:

- Με τις συναρτήσεις–μέλη `front()` και `back()`. Αυτές επιστρέφουν αναφορά στο πρώτο και τελευταίο στοιχείο της μη κενού list για το οποίο καλούνται.
- Με τη δράση του τελεστή '`*`' σε όνομα iterator. Εννοείται ότι αν ο iterator δείχνει σε σταθερό στοιχείο, δεν μπορούμε να μεταβάλουμε μέσω αυτού την τιμή του στοιχείου.
- Με τη δράση του τελεστή '`*`' σε δείκτη σε κάποιο στοιχείο του list. Δεν μπορούμε όμως να μετακινήσουμε το δείκτη με αριθμητική δεικτών.

Επιπλέον συναρτήσεις–μέλη

Επιπλέον των παραπάνω, το list παρέχει συναρτήσεις–μέλη που τροποποιούν τη σειρά των στοιχείων του:

- Η συνάρτηση–μέλος `reverse()` αναστρέφει τη σειρά των στοιχείων στο list για το οποίο καλείται. Δεν δέχεται ορίσματα και δεν επιστρέφει τιμή.
- Η συνάρτηση–μέλος `sort()` ταξινομεί ένα list. Αν N είναι το πλήθος των στοιχείων του, χρειάζεται αριθμό πράξεων της τάξης του $N \log N$. Η συνάρτηση έχει δύο παραλλαγές:

```
c.sort();
c.sort(comp);
```

Στην πρώτη, δεν δέχεται όρισμα. Η ταξινόμηση γίνεται από το «μικρότερο» στο «μεγαλύτερο» (ό,τι κι αν σημαίνει αυτό), συγκρίνοντας τα στοιχεία με τον τελεστή '<'.

Στη δεύτερη μορφή της, η συνάρτηση δέχεται ένα αντικείμενο–συνάρτηση (§9.3) ή μία συνάρτηση λάμδα (§9.3.1) ή μία συνήθη συνάρτηση, `comp()`. Το `comp()` δέχεται δύο ορίσματα με τύπο τον τύπο των στοιχείων του list, και επιστρέφει λογική τιμή, **`true`** ή **`false`**, αν το πρώτο όρισμά του είναι «μικρότερο» (ό,τι κι αν σημαίνει αυτό) από το δεύτερο ή όχι. Η ταξινόμηση γίνεται συγκρίνοντας τα στοιχεία με το `comp()`.

Η συνάρτηση και στις δύο μορφές ταξινομεί έτσι ώστε ισοδύναμα στοιχεία να μην αλλάξουν τη σχετική τους θέση. Δεν επιστρέφει τίποτε.

Παράδειγμα

```
#include <list>
#include <iostream>

bool
lessthan10(double a)
{
  return a < 10.0;
}

int
main()
{
  std::list<double> c;  // creates empty list

  for (int i{0}; i != 10; ++i) {
      c.push_back(2.0 * i);
  }
  // c: {0.0, 2.0, 4.0, ..., 18.0}

  c.remove(16.0);
  // any element with value equal to 16.0 is erased.

  c.remove_if(lessthan10);
  // all elements with value less than 10 are erased.
```

```
  std::cout << "List␣is␣:\t";

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

Παρατηρήστε ότι η εντολή

```
c.remove_if(lessthan10);
```

μπορεί να αντικατασταθεί από την

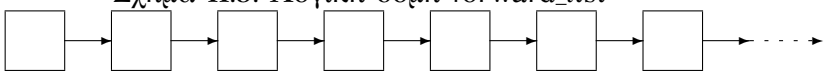
```
c.remove_if(std::bind(std::less<double>{},
                      std:placeholders::_1,10.0));
```

αρκεί να συμπεριλάβουμε το `<functional>`. Με αυτή την τροποποίηση αποφεύγουμε να ορίσουμε δική μας συνάρτηση, την `lessthan10()`, και έχουμε μεγαλύτερη ευελιξία καθώς η κλήση της με άλλη σταθερή τιμή δεν απαιτεί νέα συνάρτηση.

11.5.5 forward_list

Ένα forward_list, δηλαδή ένα αντικείμενο του container `std::forward_list<>`, αποθηκεύει τα στοιχεία του με τη σειρά εισαγωγής σε ξεχωριστά τμήματα μνήμης, όπως κάνει και ένα list. Σε αντίθεση με αυτό, όμως, κάθε θέση στοιχείου γνωρίζει τη θέση μόνο του επόμενου στοιχείου και όχι του προηγούμενου. Η εσωτερική δομή του forward_list δίνεται σχηματικά στο Σχήμα 11.5.

Σχήμα 11.5: Λογική δομή forward_list



Η συγκεκριμένη εσωτερική δομή μοιάζει πολύ με αυτή του list που είδαμε προηγουμένως. Γι' αυτό, το forward_list παρέχει πολλές από τις δυνατότητες του list. Μάλιστα, είναι σχεδιασμένο να μην υστερεί σε ταχύτητα ή μνήμη από οποιαδήποτε λίστα μονής κατεύθυνσης θα μπορούσαμε να κατασκευάσουμε σε κώδικα.

Υπάρχει βέβαια μια βασική διαφορά του forward_list από το list: από την κατασκευή του δεν μπορούμε να διατρέξουμε ένα forward_list ανάστροφα, σε σταθερό χρόνο ανεξάρτητα από τη θέση που ξεκινούμε. Έτσι, απουσιάζουν από το `forward_list<>` οι τύποι των ανάστροφων iterators και οι σχετικές συναρτήσεις που παράγουν τέτοιους iterators. Τις τελευταίες τις αντικαθιστούν δύο άλλες συναρτήσεις, οι `before_begin()` και `cbefore_begin()` με επιστρεφόμενη τιμή iterator και const_iterator, αντίστοιχα, στη θέση πριν το πρώτο στοιχείο ενός forward_list. Επιπλέον, οι iterators που παρέχονται είναι μονής κατεύθυνσης (forward iterators),

από την αρχή προς το τέλος του container.

Συναντήσαμε επίσης στο `list<>`, αλλά και σε άλλους containers που έχουμε δει μέχρι τώρα, κάποιες συναρτήσεις–μέλη για εισαγωγή ή διαγραφή στοιχείων που δέχονται ως όρισμα const_iterator στον container για τον οποίο καλούνται. Ο iterator καθορίζει τη θέση *πριν* από την οποία γίνεται η προσθήκη ή αφαίρεση του στοιχείου. Παρόμοιες συναρτήσεις παρέχονται και από το forward_list, αλλά με ονόματα που καταλήγουν σε `_after`, καθώς ο const_iterator που δέχονται προσδιορίζει τη θέση *μετά* από την οποία γίνεται η δράση της συνάρτησης. Η εισαγωγή ή διαγραφή στοιχείου στην αρχή του forward_list, με τέτοιες συναρτήσεις, προϋποθέτει την ύπαρξη iterator στην προηγούμενη θέση· γι' αυτό ακριβώς παρέχονται οι `before_begin()` και `cbefore_begin()`.

Οι συγκεκριμένες απαιτήσεις σε ταχύτητα και μνήμη που καθορίζουν τη σχεδίαση του `forward_list<>`, σε συνδυασμό με τη μη δυνατότητα αναστροφής σε αυτό, έχουν ορισμένες «επιπλοκές». Δεν παρέχονται συναρτήσεις–μέλη που χειρίζονται το τελευταίο στοιχείο· απουσιάζουν οι `back()`, `push_back()` και `pop_back()`, που παρέχει το list. Δεν υπάρχει iterator στο τελευταίο στοιχείο, για λόγους εξοικονόμησης μνήμης, και δεν μπορούμε να μετακινηθούμε από το `end()` σε αυτό καθώς είναι ανάστροφη κίνηση. Επίσης, απουσιάζει η συνάρτηση `size()`, καθώς η υλοποίησή της απαιτεί επιπλέον πληροφορία από την απολύτως απαραίτητη (μια επιπλέον μεταβλητή) ή χρόνο υπολογισμού ανάλογο του πλήθους των στοιχείων.

Κατά τα λοιπά, ο τύπος `std::forward_list<>` παρέχει τους κοινούς τύπους, τις συναρτήσεις–μέλη και τους τελεστές σύγκρισης που παρουσιάσαμε στο §11.4. Η χρήση του προϋποθέτει τη συμπερίληψη του header `<forward_list>`. Διαχειρίζεται τα στοιχεία του πιο γρήγορα και με λιγότερες απαιτήσεις μνήμης από ένα list. Συνεπώς, ο forward_list είναι ο προτιμητέος container αν χρειαζόμαστε αποθήκευση σε λίστα και δεν μας δυσχεραίνουν οι περιορισμοί του.

Έχοντας υπόψη τα παραπάνω, ας δούμε συνοπτικά τους μηχανισμούς δημιουργίας, προσθήκης/διαγραφής στοιχείων και τις λοιπές συναρτήσεις–μέλη που παρέχει ο `std::forward_list<>`. Δείτε για επεξηγήσεις την παρουσίαση του `std::list<>` που προηγήθηκε.

Ορισμός

Ορισμό με πιθανή απόδοση αρχικής τιμής ενός `std::forward_list<>` μπορούμε να κάνουμε ως εξής:

- Η εντολή

  ```
  std::forward_list<T> c;
  ```

  δημιουργεί ένα κενό forward_list για αποθήκευση στοιχείων τύπου T.

- Ο κώδικας

  ```
  std::forward_list<T> c1;
  std::forward_list<T> c2{c1};
  ```

```
std::forward_list<T> c3{std::move(c1)};
```

δημιουργεί το forward_list c2 με *αντιγραφή* του c1 και το forward_list c3 με *μετακίνηση* του c1.

- Ο κώδικας

```
T a1, a2, a3, ...;
std::forward_list<T> c{a1, a2, a3, ... };
```

δημιουργεί ένα forward_list με πλήθος και τιμές στοιχείων που προσδιορίζονται από τη λίστα που ακολουθεί το όνομά του.

- Αν beg και end είναι δύο input iterators, η εντολή

```
std::forward_list<T> c{beg,end};
```

δημιουργεί forward_list με αντίγραφα των στοιχείων στο διάστημα [beg,end).

- Η εντολή

```
std::forward_list<T> c(N);
```

ορίζει το c ως ένα forward_list με N θέσεις για αντικείμενα τύπου T.

- Η εντολή

```
std::forward_list<T> c(N, elem);
```

ορίζει ένα forward_list με N αντίγραφα του αντικειμένου elem, με πιθανή μετατροπή του στον τύπο T.

Διαχείριση μνήμης

Ένα `forward_list<>` παρέχει τις συναρτήσεις–μέλη `empty()` και `max_size()` που περιγράψαμε προηγουμένως. Δεν παρέχει τη `size()` για λόγους που αναφέραμε. Όποτε χρειαζόμαστε το πλήθος στοιχείων ενός `forward_list<>` μπορούμε να το υπολογίσουμε από την απόσταση τελικού και αρχικού iterator σε αυτόν (§10.5.4):

```
#include <iterator>
#include <forward_list>

std::forward_list<T> c;
 ...
auto size = std::distance(c.cbegin(), c.cend());
```

Προσθήκη στοιχείων

Προσθήκη στοιχείων σε αντικείμενο του `forward_list<>` μπορούμε να κάνουμε ως εξής:

- Με αντιγραφή ή μετακίνηση άλλου forward_list ίδιου τύπου, και ταυτόχρονη καταστροφή των παλαιών στοιχείων:

```
std::forward_list<T> c1, c2, c3;
 ...
c2 = c1;
c3 = std::move(c1);
```

- Με εκχώρηση λίστας τιμών:

```
std::forward_list<T> c;
T a1, a2, a3, ...;
c = {a1,a2,a3, ...};
```

- Με εναλλαγή στοιχείων με άλλο forward_list ίδιου τύπου, με κλήση είτε της συνάρτησης–μέλους `swap()` είτε της `std::swap()`:

```
std::forward_list<T> c1, c2, c3;
 ...
c1.swap(c2);
std::swap(c3,c1);
```

- Με την κλήση της συνάρτησης–μέλους `insert_after()` για ένα forward_list προκαλούμε εισαγωγή σε επιλεγμένη θέση, ενός ή περισσότερων νέων στοιχείων, με αντιγραφή ή μετακίνηση:

```
c.insert_after(pos, elem);
c.insert_after(pos, N, elem);
c.insert_after(pos, beg, end);
c.insert_after(pos, {a1, a2, a3, ...});
```

  Το πρώτο όρισμα είναι ένας const_iterator. *Μετά* τη θέση που δείχνει αυτός, γίνεται η εισαγωγή. Η πρώτη εντολή αντιγράφει ή μετακινεί το elem ως νέο στοιχείο στο forward_list c. Οι επόμενες εισάγουν N αντίγραφα του elem, τα στοιχεία στο διάστημα των input iterators [beg,end) και τα στοιχεία μιας λίστας.

  Η συνάρτηση–μέλος `insert_after()` επιστρέφει iterator στο *τελευταίο* νέο στοιχείο ή το pos, αν το πλήθος των εισαγόμενων στοιχείων είναι 0 (αν `N==0` ή `beg==end` ή η λίστα είναι κενή).

- Καταστροφή όλων των στοιχείων ενός forward_list και αντικατάστασή τους από νέα γίνεται με τη συνάρτηση–μέλος `assign()`:

```
c.assign(N,elem);
c.assign(beg, end);
c.assign({a1,a2,a3,...});
```

Η πρώτη μορφή εισάγει N αντίγραφα του elem στο forward_list c. Στη δεύτερη μορφή, η κλήση αντικαθιστά τα στοιχεία του c με αντίγραφα των στοιχείων στο διάστημα των input_iterators [beg,end). Στην τρίτη μορφή, η συνάρτηση εισάγει τις τιμές της λίστας. Η συνάρτηση δεν επιστρέφει τίποτε και στις τρεις μορφές της.

- Η εντολή

  ```
  c.push_front(elem);
  ```

  αντιγράφει ή μετακινεί το στοιχείο elem στην αρχή του forward_list c. Δεν επιστρέφει τιμή.

- Η συνάρτηση–μέλος `emplace_after()` δημιουργεί προσωρινά ένα αντικείμενο κατάλληλου τύπου, το οποίο κατόπιν εισάγεται με μετακίνηση στο forward_list για το οποίο καλείται. Η συνάρτηση δέχεται ως πρώτο όρισμα ένα const_iterator· η εισαγωγή θα γίνει *μετά* τη θέση που δείχνει αυτός. Τα επόμενα ορίσματα στη συνάρτηση χρησιμοποιούνται για να δημιουργηθεί το αντικείμενο προς εισαγωγή. Η συνάρτηση επιστρέφει iterator στη θέση του νέου στοιχείου.

- Η συνάρτηση–μέλος `emplace_front()` δημιουργεί και εισάγει στοιχείο στην αρχή του forward_list για το οποίο καλείται. Δεν επιστρέφει τίποτε.

- Η κλήση της συνάρτησης–μέλους `resize()` προκαλεί εισαγωγή ή διαγραφή στοιχείων. Δεν επιστρέφει τίποτε. Η εντολή

  ```
  v.resize(N);
  ```

  αλλάζει το πλήθος των στοιχείων του v σε N, διαγράφοντας από το τέλος ή προσθέτοντας εκεί στοιχεία. Στην τελευταία περίπτωση τα νέα στοιχεία που εισάγονται έχουν την προκαθορισμένη τιμή για τον τύπο τους. Με δεύτερο όρισμα, δηλαδή με την εντολή

  ```
  v.resize(N,elem);
  ```

  τα τυχόν νέα στοιχεία είναι αντίγραφα του elem.

- Προσθήκη ή διαγραφή στοιχείων γίνεται και με κατάλληλους αλγόριθμους.

- Η συνάρτηση–μέλος `splice_after()` μετακινεί στοιχεία από ένα forward_list σε άλλο. Έχει τρεις παραλλαγές. Σε όλες δέχεται ως πρώτο όρισμα ένα const_iterator· η εισαγωγή θα γίνει μετά τη θέση που δείχνει αυτός. Ως δεύτερο, δέχεται το forward_list από το οποίο θα γίνει η μετακίνηση, είτε ως αναφορά

είτε ως προσωρινή ποσότητα. Οι παραλλαγές διαφοροποιούνται στα επόμενα ορίσματα.

Η πρώτη μορφή δεν δέχεται τρίτο όρισμα. Μετακινεί όλα τα στοιχεία του δεύτερου ορίσματος στο forward_list για το οποίο κλήθηκε. Η δεύτερη μορφή δέχεται ως τρίτο όρισμα ένα iterator σε σταθερό στοιχείο του δεύτερου ορίσματος. Μετακινεί το στοιχείο που δείχνει αυτός. Η τρίτη μορφή δέχεται ως τρίτο και τέταρτο όρισμα, δύο const_iterators στο δεύτερο όρισμα. Μετακινεί τα στοιχεία μεταξύ αυτών.

- Η συνάρτηση `merge()` μετακινεί στοιχεία από ένα *ταξινομημένο* forward_list σε άλλο forward_list, επίσης *ταξινομημένο*. Το forward_list από το οποίο μετακινούνται τα στοιχεία απομένει κενό. Έχει δύο παραλλαγές:

  Ο κώδικας

  ```
  std::forward_list<T> c1, c2;
   ...
  c1.merge(c2);
  ```

  μετακινεί τα στοιχεία της (ταξινομημένης) c2 στην (ταξινομημένη) c1 με τέτοιο τρόπο ώστε, μετά τη συγχώνευση, η c1 να είναι πάλι ταξινομημένη.

  Ο κώδικας

  ```
  std::forward_list<T> c1, c2;
   ...
  c1.merge(c2, comp);
  ```

  κάνει το ίδιο με τον προηγούμενο, όμως, θεωρεί ότι κάθε forward_list έχει ταξινομηθεί με το κριτήριο `comp()`. Με αυτό το κριτήριο γίνεται η συγχώνευση της c2 στη c1, ώστε να προκύψει η c1 ταξινομημένη.

Διαγραφή στοιχείων

Διαγραφή στοιχείων από ένα forward_list c, γίνεται ως εξής:

- Με την κοινή συνάρτηση–μέλος `clear()`.
- Με τη συνάρτηση–μέλος `erase_after()`, είτε με ένα είτε με δύο ορίσματα τύπου const_iterator:

  ```
  c.erase_after(pos);
  c.erase_after(beg,end);
  ```

  Στην πρώτη μορφή διαγράφεται το στοιχείο *μετά* τη θέση pos και στη δεύτερη τα στοιχεία στο διάστημα (beg,end). Παρατηρήστε ότι στη δεύτερη παραλείπει το στοιχείο στην θέση beg. Και στις δύο περιπτώσεις η επιστρεφόμενη τιμή είναι iterator στην επόμενη θέση από το τελευταίο διαγραμμένο στοιχείο.

- Με τη χρήση της `resize()`.
- Με τη συνάρτηση–μέλος `pop_front()`. Διαγράφεται το πρώτο στοιχείο ενός μη κενού forward_list. Δεν επιστρέφει τίποτε.
- Με τη συνάρτηση–μέλος `remove()`:

  ```
  c.remove(a);
  c.remove_if(op);
  ```

  Στην πρώτη μορφή, η κλήση διαγράφει όλα τα στοιχεία του forward_list c που είναι ίσα με a. Στη δεύτερη μορφή της, η `remove_if()` διαγράφει όλα τα στοιχεία για τα οποία το `op()` επιστρέφει **`true`**. Η συνάρτηση δεν επιστρέφει τιμή.
- Με τη συνάρτηση–μέλος `unique()`:

  ```
  c.unique();
  c.unique(comp);
  ```

  Στην πρώτη εντολή, η συνάρτηση εντοπίζει στο forward_list c, ομάδες *διαδοχικών* στοιχείων που έχουν ίδια τιμή. Διαγράφει όλα τα στοιχεία εκτός από το πρώτο σε κάθε ομάδα.

  Στη δεύτερη μορφή, η `unique()` δρα το `comp()` σε όλα τα διαδοχικά ζεύγη στοιχείων του forward_list. Σε όσα το `comp()` επιστρέφει **`true`**, διαγράφει το δεύτερο στοιχείο. Η συνάρτηση δεν επιστρέφει τιμή.

Προσπέλαση στοιχείων

Προσπέλαση στοιχείων ενός forward_list γίνεται ως εξής:

- Με τη συνάρτηση–μέλος `front()`.
- Με τη δράση του τελεστή '`*`' σε όνομα iterator.
- Με τη δράση του τελεστή '`*`' σε δείκτη σε κάποιο στοιχείο του forward_list.

Επιπλέον συναρτήσεις–μέλη

Επιπλέον των παραπάνω, το forward_list παρέχει συναρτήσεις–μέλη που τροποποιούν τη σειρά των στοιχείων του:

- Η συνάρτηση–μέλος `reverse()` αναστρέφει τη σειρά των στοιχείων του αντικειμένου για το οποίο καλείται. Δεν δέχεται ορίσματα και δεν επιστρέφει τιμή.
- Η συνάρτηση–μέλος `sort()` ταξινομεί ένα forward_list. Αν N είναι το πλήθος των στοιχείων του, χρειάζεται αριθμό πράξεων της τάξης του $N \log N$. Η συνάρτηση έχει δύο παραλλαγές:

```
c.sort();
c.sort(comp);
```

Στην πρώτη, δεν δέχεται όρισμα. Η ταξινόμηση γίνεται από το «μικρότερο» στο «μεγαλύτερο» (ό,τι κι αν σημαίνει αυτό), συγκρίνοντας τα στοιχεία με τον τελεστή '<'. Στη δεύτερη μορφή της, η ταξινόμηση γίνεται συγκρίνοντας τα στοιχεία με το `comp()`. Η συνάρτηση και στις δύο μορφές ταξινομεί έτσι ώστε ισοδύναμα στοιχεία να μην αλλάξουν τη σχετική τους θέση. Δεν επιστρέφει τίποτε.

11.6 Associative containers

Οι containers αυτής της κατηγορίας αποθηκεύουν τα στοιχεία τους όχι με τη σειρά εισαγωγής αλλά με σειρά που καθορίζεται από κάποιο κριτήριο. Η ταξινόμηση είναι αυτόματη και ο προγραμματιστής δε χρειάζεται να κάνει κάτι το ιδιαίτερο, πέρα από το να καθορίσει αυτό το κριτήριο, αν δεν τον ικανοποιεί το προκαθορισμένο.

Οι associative containers παρουσιάζουν ένα βασικό πλεονέκτημα έναντι των sequence containers: Δεν είναι τόσο η αυτόματη ταξινόμηση που κάνουν όσο η ταχύτητα που συνεπάγεται αυτή κατά την αναζήτηση στοιχείου με συγκεκριμένη τιμή ή ιδιότητα. Η δυαδική αναζήτηση (binary search (§Β′.1.2)) που μπορεί να χρησιμοποιηθεί σε αντικείμενο τέτοιου τύπου, με πλήθος στοιχείων N, χρειάζεται πλήθος συγκρίσεων της τάξης $O(\log N)$ ενώ η γραμμική αναζήτηση (§Β′.1.1) που πρέπει να εφαρμοστεί στους (αταξινόμητους) sequence containers, απαιτεί συγκρίσεις με πλήθος $O(N)$. Από την άλλη, ο συγκεκριμένος τρόπος αποθήκευσης των στοιχείων έχει ως συνέπεια να μην επιτρέπεται να αλλάξουμε τιμή σε ένα στοιχείο καθώς θα αλλοιώσουμε τη σειρά ταξινόμησης. Έτσι, στους associative containers δεν παρέχονται συναρτήσεις για την άμεση πρόσβαση ή τροποποίηση στοιχείων. Πρόσβαση γίνεται μόνο μέσω iterator και μάλιστα η τιμή που «δείχνει» αυτός μπορεί μόνο να διαβαστεί αλλά όχι να αλλάξει. Στην περίπτωση που θέλουμε να τροποποιήσουμε κάποιο στοιχείο πρέπει να το διαγράψουμε και κατόπιν να το εισαγάγουμε με τη νέα τιμή.

Στην κατηγορία των associative containers ανήκουν οι κλάσεις `std::set<>`, `std::multiset<>`, `std::map<>`, `std::multimap<>`.

11.6.1 set και multiset

Τα `std::set<>` και `std::multiset<>` είναι containers που αποθηκεύουν τιμές με σειρά που καθορίζεται από κάποιο κριτήριο. Η διαφορά των δύο είναι ότι το multiset μπορεί να δεχτεί περισσότερα από ένα στοιχεία με ίδια τιμή ενώ το set αγνοεί τυχόν προσπάθειες να εισαγάγουμε στοιχείο που ήδη υπάρχει στο σύνολο. Με άλλα λόγια, στο set τα στοιχεία είναι μοναδικά.

Οι τύποι `std::set<>` και `std::multiset<>` παρέχουν όλους τους κοινούς τύπους, τις συναρτήσεις–μέλη και τους τελεστές σύγκρισης που παρουσιάσαμε στο §11.4. Η χρήση τους προϋποθέτει τη συμπερίληψη του `<set>`. Οι iterators που παρέχει ένα set ή multiset, ορθής και ανάστροφης φοράς, είναι δύο κατευθύνσεων (bidirectional iterators).

Ό,τι θα αναφέρουμε παρακάτω για set ισχύει και για multiset, εκτός αν διευκρινίζεται διαφορετικά.

Ορισμός

Δήλωση μεταβλητής τύπου set ή multiset με στοιχεία τύπου T γίνεται με ένα από τους παρακάτω τρόπους. Καθώς τους έχουμε ήδη αναλύσει στο §11.2.1 θα τους επαναλάβουμε περιληπτικά. Στα επόμενα, ο τύπος Set συμβολίζει οποιοδήποτε από τα `std::set<T>`, `std::multiset<T>`.

- Η εντολή

  ```
  Set c;
  ```

  δημιουργεί κενό Set.

- Οι εντολές

  ```
  Set c1;
  Set c2{c1};
  Set c3{std::move(c1)};
  ```

  δημιουργούν το Set c2 ως αντίγραφο του c1 και το Set c3 με μετακίνηση του c1.

- Ο κώδικας

  ```
  T a1, a2, a3, ...;
  Set c{a1, a2, a3, ... };
  ```

  δημιουργεί Set με τιμές στοιχείων που προσδιορίζονται από τη λίστα που ακολουθεί το όνομά του.

- Η εντολή

  ```
  Set c{beg,end};
  ```

  δημιουργεί ένα Set αντιγράφοντας στοιχεία από κάποιο άλλο container, πιθανώς διαφορετικού τύπου, με iterators που βρίσκονται στο διάστημα [beg,end).

Στους παραπάνω ορισμούς η ταξινόμηση γίνεται με το προκαθορισμένο κριτήριο, το `std::less<T>`, το προκαθορισμένο δηλαδή αντικείμενο–συνάρτηση του `<functional>` (§9.3). Αυτό συγκρίνει τα στοιχεία με τον τελεστή '<' (αύξουσα σειρά). Γενικά, μπορούμε να περάσουμε ως δεύτερη παράμετρο του template *τον τύπο* ενός

αντικειμένου–συνάρτηση ή μια συνάρτηση λάμδα, που θα δέχεται δύο ορίσματα και θα επιστρέφει λογική τιμή, `true/false`, ανάλογα αν το πρώτο είναι «μικρότερο» ή όχι από το δεύτερο. Π.χ. αν θέλουμε να ορίσουμε ένα set που ταξινομεί με φθίνουσα σειρά χρησιμοποιούμε την εντολή

```
std::set<T, std::greater<T>> c;
```

Τονίζουμε ότι στο template πρέπει να δοθεί ως δεύτερη παράμετρος ένας *τύπος*· αυτό αποκλείει την απλή συνάρτηση. Το κριτήριο ταξινόμησης πρέπει να ικανοποιεί τις συνθήκες της *γνήσιας ασθενούς διάταξης* (§9.4.2). Η ισοδυναμία δύο στοιχείων προσδιορίζεται με τη βοήθεια αυτού του κριτηρίου: δύο στοιχεία είναι ισοδύναμα όταν κανένα δεν είναι «μικρότερο» του άλλου.

Αν επιθυμούμε, μπορούμε να προσδιορίσουμε το κριτήριο ταξινόμησης κατά τη διάρκεια της εκτέλεσης του προγράμματος, περνώντας το ως όρισμα στους constructors. Δεν θα αναφερθούμε περισσότερο σε αυτή τη δυνατότητα. Επιπλέον, μπορούμε να εξαγάγουμε το κριτήριο ταξινόμησης ενός Set με τη συνάρτηση–μέλος `key_comp()`.

Διαχείριση μνήμης

Οι containers `std::set<>` και `std::multiset<>` παρέχουν τις συναρτήσεις `size()`, `empty()` και `max_size()` που περιγράψαμε προηγουμένως (§11.4).

Προσθήκη στοιχείων

Εισαγωγή στοιχείων σε ένα Set μετά τη δημιουργία του γίνεται με τους ακόλουθους τρόπους:

- με εκχώρηση άλλου Set ίδιου τύπου ή προσωρινής ποσότητας ίδιου τύπου:

```
Set c1, c2, c3;
 ...
c2 = c1;
c3 = std::move(c1);
```

- με εκχώρηση λίστας τιμών:

```
Set c;
T a1, a2, a3, ...;
c = {a1,a2,a3, ...};
```

- Με εναλλαγή στοιχείων με άλλο Set ίδιου τύπου, με κλήση είτε της συνάρτησης–μέλους `swap()` είτε της `std::swap()`:

```
Set c1, c2, c3;
 ...
c1.swap(c2);
std::swap(c3,c1);
```

- Με την κλήση της συνάρτησης–μέλους `insert()`, με όρισμα μία τιμή κατάλληλη για αποθήκευση στο Set για το οποίο καλείται, αντιγράφεται ή μετακινείται το όρισμα:

  ```
  Set c;
  T elem;
  ...
  c.insert(elem);
  ```

  Αν το c είναι multiset, η συνάρτηση επιστρέφει iterator στη θέση του νέου στοιχείου. Το νέο στοιχείο τοποθετείται μετά από όλα τα ισοδύναμά του στοιχεία. Αν το c είναι set, η `insert()` επιστρέφει ζεύγος (std::pair<> (§9.2.1)), το πρώτο μέλος του οποίου είναι iterator στη θέση του νέου ή του ήδη υπάρχοντος στοιχείου ενώ το δεύτερο είναι λογική τιμή που δείχνει αν η εισαγωγή ήταν επιτυχής ή όχι (στην περίπτωση που το στοιχείο υπήρχε ήδη).

- Με τον κώδικα

  ```
  Set c;
  T elem;
  Set::const_iterator pos;
  ...
  c.insert(pos,elem);
  ```

  επιχειρείται η εισαγωγή με αντιγραφή ή μετακίνηση του elem, λαμβάνοντας υπόψη την *υπόδειξη* για την πιθανή θέση μέσω του const_iterator pos. Η συνάρτηση επιστρέφει iterator: αν το c είναι set, δείχνει στο νέο ή υπάρχον στοιχείο. Αν το c είναι multiset, δείχνει στο νέο στοιχείο (το οποίο τοποθετείται μετά από όλα τα ισοδύναμά του στοιχεία).

- Οι εντολές

  ```
  c.insert(beg, end);
  c.insert({a1, a2, a3, ...});
  ```

  εισάγουν στο Set c αντίγραφα των στοιχείων στο διάστημα [beg,end) (η πρώτη) και τη λίστα τιμών που δίνεται ως όρισμα στη δεύτερη. Οι συναρτήσεις δεν επιστρέφουν τίποτε.

- Με τη συνάρτηση–μέλος `emplace()` δημιουργείται προσωρινά ένα αντικείμενο κατάλληλου τύπου, από τα ορίσματά της. Το αντικείμενο κατόπιν εισάγεται με μετακίνηση στο Set για το οποίο καλείται η συνάρτηση. Π.χ.

  ```
  std::set<std::string> c;
  c.emplace("one");   // c is { "one" }
  c.emplace("two");   // c is { "one", "two" }
  c.emplace("three"); // c is { "one", "three", "two" }
  ```

Αν το c είναι multiset, η συνάρτηση επιστρέφει iterator στη θέση του νέου στοιχείου (το στοιχείο τοποθετείται μετά από όλα τα ισοδύναμά του στοιχεία). Αν το c είναι set, επιστρέφει ζεύγος, το πρώτο μέλος του οποίου είναι iterator στη θέση του νέου ή του ήδη υπάρχοντος στοιχείου ενώ το δεύτερο είναι λογική τιμή που δείχνει αν η εισαγωγή ήταν επιτυχής ή όχι.

- Με τη συνάρτηση–μέλος `emplace_hint()` δημιουργείται προσωρινά ένα αντικείμενο κατάλληλου τύπου, το οποίο κατόπιν εισάγεται με μετακίνηση στο set για το οποίο καλείται. Η συνάρτηση δέχεται ως πρώτο όρισμα ένα const_iterator· *υποδεικνύεται* έτσι η πιθανή θέση εισαγωγής. Ως δεύτερο, τρίτο, ... όρισμα δέχεται μία ή περισσότερες ποσότητες που τις χρησιμοποιεί για να δημιουργήσει το αντικείμενο προς εισαγωγή.

  Η συνάρτηση επιστρέφει iterator: αν το c είναι set, δείχνει στο νέο ή σε υπάρχον στοιχείο. Αν το c είναι multiset, δείχνει στο νέο στοιχείο (το οποίο τοποθετείται μετά από όλα τα ισοδύναμά του στοιχεία).

Διαγραφή στοιχείων

Διαγραφή στοιχείων από ένα Set c γίνεται με τους ακόλουθους τρόπους:

- Με την κοινή συνάρτηση–μέλος `clear()`:

  ```
  c.clear();
  ```

  Η κλήση της διαγράφει όλα τα στοιχεία του Set c.

- Με τη συνάρτηση–μέλος `erase()` στις τρεις παραλλαγές της:

  ```
  Set c;
  Set::value_type elem;
  Set::const_iterator pos,beg,end;

  c.erase(elem);
  c.erase(pos);
  c.erase(beg,end);
  ```

  Στην πρώτη, δέχεται ως όρισμα μια τιμή. Διαγράφει όλα τα στοιχεία που είναι «ισοδύναμα» (ίσα) με αυτή την τιμή στο Set για το οποίο καλείται. Επιστρέφει το πλήθος των στοιχείων που διαγράφηκαν (για set είναι το πολύ 1, για multiset μπορεί να είναι μεγαλύτερο). Ο τύπος της επιστρεφόμενης ποσότητας είναι το `size_type`, όπως αυτό ορίζεται από το συγκεκριμένο set ή multiset.

  Στη δεύτερη, δέχεται ως όρισμα ένα const_iterator στο Set για το οποίο καλείται. Διαγράφει το στοιχείο στη θέση που δείχνει ο συγκεκριμένος. Επιστρέφει iterator στην επόμενη θέση από το στοιχείο που διαγράφηκε.

  Στην τρίτη, δέχεται ως όρισμα δυο const_iterators στο Set για το οποίο καλείται. Διαγράφει τα στοιχεία με θέσεις στο διάστημα [beg,end). Επιστρέφει iterator στην επόμενη θέση από το στοιχείο που διαγράφηκε.

Προσπέλαση στοιχείων

Όπως αναφέραμε, στο set και multiset δεν παρέχονται συναρτήσεις για την άμεση πρόσβαση ή τροποποίηση στοιχείων. Υπάρχει βέβαια η δυνατότητα προσπέλασης (αλλά όχι τροποποίησης) μέσω iterator.

Επιπλέον συναρτήσεις–μέλη

Καθώς οι `set<>` και `multiset<>` είναι βελτιστοποιημένοι για γρήγορη αναζήτηση στοιχείων, παρέχουν ως μέλη, συναρτήσεις που εκτελούν λειτουργίες κάποιων γενικών αλγορίθμων που θα δούμε παρακάτω, πολύ πιο γρήγορα (ο αριθμός των απαιτούμενων πράξεων είναι τάξης $O(\log N)$ έναντι $O(N)$ των γενικών). Είναι οι εξής:

- Η συνάρτηση–μέλος `count()` με όρισμα μια τιμή επιστρέφει το πλήθος των στοιχείων στο Set για το οποίο καλείται, που είναι «ισοδύναμα» (ίσα) με αυτή την τιμή. Ο τύπος της επιστρεφόμενης ποσότητας είναι το `size_type`, όπως αυτό ορίζεται από το συγκεκριμένο set ή multiset.
- Η συνάρτηση–μέλος `find()` με όρισμα μια τιμή εντοπίζει το στοιχείο που είναι «ισοδύναμο» με αυτή την τιμή ή, για multiset, ένα από τα ισοδύναμα στοιχεία. Επιστρέφει iterator στη θέση του, ή αν δεν υπάρχει αυτό, `end()`. Στην περίπτωση που η συνάρτηση κληθεί για σταθερό set ή multiset επιστρέφει const_iterator στη θέση του στοιχείου (ή το `cend()` αν δεν το βρει).
- Η συνάρτηση–μέλος `lower_bound()`, με όρισμα μία τιμή, επιστρέφει σε iterator τη θέση του πρώτου στοιχείου από την αρχή που δεν είναι «μικρότερο» από το όρισμά της.
- Η συνάρτηση–μέλος `upper_bound()`, με όρισμα μία τιμή, επιστρέφει σε iterator τη θέση του πρώτου στοιχείου από την αρχή που είναι «μεγαλύτερο» από το όρισμά της (ισοδύναμα, το πρώτο στοιχείο από το οποίο το όρισμα είναι «μικρότερο»).
- Η συνάρτηση–μέλος `equal_range()`, με όρισμα μία τιμή, επιστρέφει σε ζεύγος (pair), τους iterators που προσδιορίζουν το διάστημα στο οποίο τα στοιχεία είναι ισοδύναμα με το όρισμα.

Παράδειγμα

```
#include <set>
#include <iostream>
#include <functional>
#include <iterator>
```

```
int
main()
{
    using crit = std::greater<int>;
    std::cout << "Set:\n";
    std::set<int,crit> c;  // define empty set

    // insert values in random order.
    c = {5, 12, 3, 6, 7, 1, 9};

    // print set
    std::cout << "Number␣of␣elements:␣" << c.size() << '\n';
    for (auto it = c.cbegin(); it != c.cend(); ++it) {
        std::cout << *it << '␣';
    }
    std::cout << '\n';

    // remove elements with value 4 and 6 and print information
    auto m = c.erase(4);
    std::cout << "There␣were␣" << m << "␣elements␣with␣value␣4\n";

    m = c.erase(6);
    std::cout << "There␣were␣" << m << "␣elements␣with␣value␣6\n";
    std::cout << '\n';

    std::cout << "Multiset:\n";

    // create multiset from c
    std::multiset<int, crit> mc{c.cbegin(), c.cend()};

    //insert element 12 three times
    mc.insert(12);
    mc.insert(12);
    mc.insert(12);

    auto p = mc.equal_range(12);
    std::cout <<"First␣12␣is␣in␣position␣"
              << std::distance(mc.cbegin(), p.first) << '\n';
    std::cout <<"Last␣12␣is␣in␣position␣"
              << std::distance(mc.cbegin(), p.second)-1 << '\n';

    // print multiset
```

```
    std::cout << "Number␣of␣elements:␣" << mc.size() << '\n';

    for (auto const & x: mc) {
        std::cout << x << '␣';
    }
    std::cout << '\n';
}
```

11.6.2 map και multimap

Οι `std::map<>` και `std::multimap<>` είναι containers που αποθηκεύουν ζεύγη (§9.2.1) ποσοτήτων στα οποία το πρώτο μέλος (first) έχει το ρόλο του «κλειδιού» και το δεύτερο (second) είναι η τιμή που αντιστοιχεί σε αυτό. Τα ζεύγη διατάσσονται αυτόματα με βάση την τιμή του «κλειδιού» τους, σύμφωνα με το κριτήριο ταξινόμησης που ορίζουμε κατά τη δημιουργία των map και multimap. Η διαφορά των δύο containers είναι ότι το multimap μπορεί να δεχτεί περισσότερα από ένα ζεύγη με το ίδιο «κλειδί» ενώ το map αγνοεί τυχόν προσπάθειες να εισαγάγουμε στοιχείο με «κλειδί» που ήδη υπάρχει στη συλλογή.

Οι τύποι `std::map<>` και `std::multimap<>` παρέχουν όλους τους κοινούς τύπους, τις συναρτήσεις–μέλη και τους τελεστές σύγκρισης που παρουσιάσαμε στο §11.4. Η χρήση τους προϋποθέτει τη συμπερίληψη του `<map>`. Οι iterators που παρέχει ένα map ή multimap, ορθής και ανάστροφης φοράς, είναι δύο κατευθύνσεων (bidirectional iterators).

Ο container `map<>` μπορεί να θεωρηθεί ως γενίκευση του `set<>` με τη διαφορά ότι κάθε στοιχείο σε ένα map συνοδεύεται από μια δεύτερη ποσότητα η οποία δεν παίζει ρόλο στην ταξινόμηση. Ανάλογη ομοιότητα παρουσιάζει και ο `multimap<>` με το `multiset<>`).

Ό,τι θα αναφέρουμε παρακάτω για map ισχύει και για multimap, εκτός αν διευκρινίζεται διαφορετικά.

Ορισμός

Ένα map ή multimap, ένα αντικείμενο δηλαδή των containers `std::map<>` ή `std::multimap<>` αντίστοιχα, μπορεί να οριστεί με τους γνωστούς τρόπους. Θα τους επαναλάβουμε συνοπτικά.

Προσέξτε ότι κατά τη δήλωση του αντικειμένου πρέπει να προσδιορίσουμε υποχρεωτικά τουλάχιστον δύο παραμέτρους στο template. Η πρώτη, K, καθορίζει τον τύπο του «κλειδιού» των στοιχείων που θα αποθηκευθούν ενώ η δεύτερη, T, είναι ο τύπος της συνοδεύουσας ποσότητας. Στα επόμενα, ο τύπος Map συμβολίζει οποιοδήποτε από τα `std::map<K,T>`, `std::multimap<K,T>`.

- Η εντολή

  ```
  Map c;
  ```

δημιουργεί κενό Map.

- Οι εντολές

```
Map c1;
Map c2{c1};
Map c3{std::move(c1)};
```

δημιουργούν το Map c2 ως αντίγραφο του c1 και το Map c3 με μετακίνηση του c1.

- Ο κώδικας

```
K k1, k2, k3, ...;
T a1, a2, a3, ...;

std::pair<K,T> p1{k1,a1};
std::pair<K,T> p2{k2,a2};
std::pair<K,T> p3{k3,a3};
...

Map c1{ p1, p2, p3, ... };
Map c2{ {k1, a1}, {k2, a2}, {k3, a3}, ... };
```

δημιουργεί το Map c1 με στοιχεία τα ζεύγη που προσδιορίζονται στη λίστα αρχικοποίησής του και το Map c2 με τα ίδια ακριβώς στοιχεία, που παράγονται όμως από τα δυάδες στοιχείων της λίστας που ακολουθεί το όνομά του.

- Η εντολή

```
Map c{beg,end};
```

δημιουργεί ένα Map κατασκευάζοντας τα στοιχεία του από τα στοιχεία με iterators στο διάστημα [beg,end).

Στους παραπάνω ορισμούς η ταξινόμηση γίνεται με το προκαθορισμένο κριτήριο, το `std::less<K>` του `<functional>`. Αυτό συγκρίνει τα «κλειδιά» των στοιχείων με τον τελεστή '`<`' (αύξουσα σειρά). Γενικά, μπορούμε να περάσουμε ως τρίτη παράμετρο του template *τον τύπο* ενός αντικειμένου–συνάρτηση ή μια συνάρτηση λάμδα, που θα δέχεται δύο ορίσματα και θα επιστρέφει λογική τιμή, **`true/false`**, ανάλογα αν το πρώτο είναι «μικρότερο» ή όχι από το δεύτερο. Τονίζουμε ότι στο template πρέπει να δοθεί ως τρίτη παράμετρος ένας *τύπος*· αυτό αποκλείει την απλή συνάρτηση. Το κριτήριο ταξινόμησης πρέπει να ικανοποιεί τις συνθήκες της *γνήσιας ασθενούς διάταξης* (§9.4.2). Η ισοδυναμία δύο στοιχείων προσδιορίζεται με τη βοήθεια αυτού του κριτηρίου.

Αν επιθυμούμε, μπορούμε να προσδιορίσουμε το κριτήριο ταξινόμησης κατά τη διάρκεια της εκτέλεσης του προγράμματος, περνώντας το ως όρισμα στους

constructors. Δεν θα αναφερθούμε περισσότερο σε αυτή τη δυνατότητα. Επιπλέον, μπορούμε να εξαγάγουμε το κριτήριο ταξινόμησης ενός Map με τη συνάρτηση–μέλος `key_comp()`.

Διαχείριση μνήμης

Οι containers `std::map<>` και `std::multimap<>` παρέχουν τις συναρτήσεις `size()`, `empty()` και `max_size()` που περιγράψαμε προηγουμένως (§11.4).

Προσθήκη στοιχείων

Εισαγωγή στοιχείων σε ένα Map γίνεται με τους ακόλουθους τρόπους:

- με εκχώρηση άλλου Map ίδιου τύπου ή προσωρινής ποσότητας ίδιου τύπου:

```
Map c1, c2, c3;
 ...
c2 = c1;
c3 = std::move(c1);
```

- με εκχώρηση λίστας τιμών:

```
Map c;

K k1, k2, k3, ...;
T a1, a2, a3, ...;

std::pair<K,T> p1{k1,a1};
std::pair<K,T> p2{k2,a2};
std::pair<K,T> p3{k3,a3};
...
c = { p1, p2, p3, ... };
```

- Με εναλλαγή στοιχείων με άλλο Map ίδιου τύπου, με κλήση είτε της συνάρτησης–μέλους `swap()` είτε της `std::swap()`:

```
Map c1, c2, c3;
 ...
c1.swap(c2);
std::swap(c3,c1);
```

- Η εντολή

```
c.insert(elem);
```

εισάγει στο Map ένα νέο στοιχείο με αντιγραφή ή μετακίνηση του elem. Θυμηθείτε ότι το elem είναι τύπου `std::pair<K,T>`· πρέπει, επομένως, να κατασκευαστεί κατάλληλα (§9.2.1).

Αν το c είναι multimap, η συνάρτηση επιστρέφει iterator στη θέση του νέου στοιχείου. Το νέο στοιχείο τοποθετείται μετά από όλα τα ισοδύναμά του στοιχεία. Αν το c είναι map, επιστρέφει ζεύγος, το πρώτο στοιχείο του οποίου είναι iterator στη θέση του νέου ή του ήδη υπάρχοντος στοιχείου ενώ το δεύτερο είναι λογική τιμή που δείχνει αν η εισαγωγή ήταν επιτυχής ή όχι (στην περίπτωση που υπάρχει στον container ίδιο στοιχείο).

- Ο κώδικας

```
Map c;
std::pair<K,T> elem;
Map:const_iterator pos;

c.insert(pos,elem);
```

επιχειρεί να εισαγάγει νέο στοιχείο με αντιγραφή ή μετακίνηση του elem, λαμβάνοντας υπόψη την *υπόδειξη* μέσω του const_iterator pos. Η συνάρτηση επιστρέφει iterator: αν το c είναι map, δείχνει στο νέο ή υπάρχον στοιχείο. Αν το c είναι multimap, δείχνει στο νέο στοιχείο (το οποίο τοποθετείται μετά από όλα τα ισοδύναμά του στοιχεία).

- Οι εντολές

```
c.insert(beg, end);
c.insert({p1, p2, p3, ...});
```

εισάγουν στο Map c αντίγραφα των στοιχείων στο διάστημα [beg,end) (η πρώτη) και τη λίστα τιμών που δίνεται ως όρισμα στη δεύτερη. Οι συναρτήσεις δεν επιστρέφουν τίποτε.

- Με τη συνάρτηση–μέλος `emplace()` δημιουργείται προσωρινά ένα αντικείμενο κατάλληλου τύπου, από τα ορίσματά της. Το αντικείμενο κατόπιν εισάγεται με μετακίνηση στο Map για το οποίο καλείται η συνάρτηση. Αν το c είναι multimap, η συνάρτηση επιστρέφει iterator στη θέση του νέου στοιχείου (το στοιχείο τοποθετείται μετά από όλα τα ισοδύναμά του στοιχεία). Αν το c είναι map, επιστρέφει ζεύγος (std::pair<> (§9.2.1)), το πρώτο μέλος του οποίου είναι iterator στη θέση του νέου ή του ήδη υπάρχοντος στοιχείου ενώ το δεύτερο είναι λογική τιμή που δείχνει αν η εισαγωγή ήταν επιτυχής ή όχι.

- Με τη συνάρτηση–μέλος `emplace_hint()` δημιουργείται προσωρινά ένα αντικείμενο κατάλληλου τύπου, το οποίο κατόπιν εισάγεται με μετακίνηση στο Map για το οποίο καλείται αυτή. Η συνάρτηση δέχεται ως πρώτο όρισμα ένα const_iterator· *υποδεικνύεται* έτσι η πιθανή θέση εισαγωγής. Ως δεύτερο,

τρίτο, ... όρισμα δέχεται μία ή περισσότερες ποσότητες που τις χρησιμοποιεί για να δημιουργήσει το αντικείμενο προς εισαγωγή.

Η συνάρτηση επιστρέφει iterator: αν το c είναι map, δείχνει στο νέο ή σε υπάρχον ζεύγος. Αν το c είναι multimap, δείχνει στο νέο ζεύγος (το οποίο τοποθετείται μετά από όλα τα ισοδύναμά του ζεύγη).

Διαγραφή στοιχείων

Διαγραφή στοιχείων από ένα Map c γίνεται με τους ακόλουθους τρόπους:

- Με την κοινή συνάρτηση–μέλος `clear()`:

  ```
  c.clear();
  ```

  Η κλήση της καταστρέφει όλα τα ζεύγη του Map c.

- Με τη συνάρτηση–μέλος `erase()` στις τρεις παραλλαγές της:

  ```
  Map c;
  K key;
  Map::const_iterator pos,beg,end;

  c.erase(key);
  c.erase(pos);
  c.erase(beg,end);
  ```

  Στην πρώτη, δέχεται ως όρισμα μια τιμή για «κλειδί». Διαγράφει όλα τα ζεύγη που έχουν τιμή «κλειδιού» «ισοδύναμη» (ίση) με το όρισμά της. Επιστρέφει το πλήθος των ζευγών που διαγράφηκαν. Ο τύπος της επιστρεφόμενης ποσότητας είναι το `size_type`, όπως αυτό ορίζεται από το συγκεκριμένο map ή multimap.

  Στη δεύτερη, δέχεται ως όρισμα ένα const_iterator στο Map για το οποίο καλείται. Διαγράφει το ζεύγος στη θέση που δείχνει ο συγκεκριμένος. Επιστρέφει iterator στην επόμενη θέση από το ζεύγος που διαγράφηκε.

  Στην τρίτη, δέχεται ως όρισμα δυο const_iterators στο Map για το οποίο καλείται. Διαγράφει τα ζεύγη με θέσεις στο διάστημα [beg,end). Επιστρέφει iterator στην επόμενη θέση από το ζεύγος που διαγράφηκε.

Προσπέλαση στοιχείων

Ο τρόπος αποθήκευσης σε ένα map καθορίζεται αποκλειστικά από τις σχετικές τιμές των «κλειδιών» των στοιχείων του. Αυτό έχει ως συνέπεια να μην επιτρέπεται να αλλάξουμε τιμή στο «κλειδί» ενός στοιχείου καθώς θα αλλοιώσουμε τη σειρά ταξινόμησης. Αντίθετα, δεν απαγορεύεται η τροποποίηση της συνοδεύουσας ποσότητας.

Πρόσβαση στα στοιχεία μπορεί να γίνει μέσω iterator. Θυμηθείτε ότι αν it είναι ένας iterator σε Map τότε το `(*it).first` (ή, ισοδύναμα, το `it->first`) είναι το «κλειδί» και το `(*it).second` (ή, ισοδύναμα, το `it->second`) είναι η συνοδεύουσα ποσότητα.

Στην περίπτωση που θέλουμε να τροποποιήσουμε το «κλειδί» κάποιου στοιχείου πρέπει να το διαγράψουμε και κατόπιν να το εισαγάγουμε με νέα τιμή.

Μια σημαντική διαφορά του `std::map<>` (*μόνο, και όχι του multimap*) από το `std::set<>` είναι ότι παρέχεται άμεση πρόσβαση στις συνοδεύουσες ποσότητες και μάλιστα με μηχανισμούς (τις αγκύλες, '`[]`' και τη συνάρτηση–μέλος `at()`) που θυμίζει το vector. Όμως, σε αντίθεση με το vector, ο δείκτης που τοποθετείται μεταξύ των αγκυλών ή ως όρισμα του `at()` δεν είναι ποσότητα ακέραιου τύπου αλλά το «κλειδί». Πιο αναλυτικά:

- αν c είναι ένα map που περιέχει στοιχείο με «κλειδί» key, τότε το `c[key]` ή το `c.at(key)` είναι *αναφορά* σε αυτό το στοιχείο.
- αν c είναι ένα map που *δεν* περιέχει στοιχείο με «κλειδί» key, η έκφραση

  ```
  c[key] = value;
  ```

  εισάγει στο c το στοιχείο `std::make_pair(key,value)`. Αν, κατά λάθος, χρησιμοποιηθεί η παραπάνω έκφραση χωρίς τιμή, π.χ. στην εντολή

  ```
  std::cout << c[key];
  ```

  τότε, πάλι θα εισαχθεί στοιχείο με πρώτο μέλος το «κλειδί» key και δεύτερο την προκαθορισμένη τιμή για τις συνοδεύουσες ποσότητες του συγκεκριμένου map. Αυτή θα είναι και η τιμή που θα τυπωθεί στο παραπάνω παράδειγμα, καθώς επιστρέφεται αναφορά στο νέο στοιχείο.
- αν c είναι ένα map που *δεν* περιέχει στοιχείο με «κλειδί» key, η έκφραση `c.at(key)` προκαλεί εξαίρεση του τύπου `std::out_of_range`. Στην περίπτωση που αυτή δεν συλληφθεί, διακόπτεται η εκτέλεση του προγράμματος.

Επιπλέον συναρτήσεις–μέλη

Οι κλάσεις `map` και `multimap` είναι βελτιστοποιημένες για γρήγορη αναζήτηση στοιχείων παρέχουν τις ίδιες εξειδικευμένες συναρτήσεις που παρέχουν και οι `std::set<>` και `std::multiset<>`:

- `count(key)`: Επιστρέφει το πλήθος των στοιχείων με «κλειδί» key. Ο τύπος της επιστρεφόμενης ποσότητας είναι το `size_type`, όπως αυτό ορίζεται από το συγκεκριμένο map ή multimap.
- `find(key)`: Επιστρέφει iterator στη θέση του στοιχείου (ή, για multimap, σε ένα από τα ισοδύναμα στοιχεία) με «κλειδί» key. Αν δεν υπάρχει στοιχείο με τέτοιο «κλειδί», επιστρέφει `end()`.

- `lower_bound(key)`: Επιστρέφει τη θέση του πρώτου στοιχείου από την αρχή, του οποίου το «κλειδί» δεν είναι μικρότερο από το key.

- `upper_bound(key)`: Εντοπίζει και επιστρέφει τη θέση του πρώτου στοιχείου που έχει «κλειδί» μεγαλύτερο από το key.

- `equal_range(key)`: Επιστρέφει, σε ζεύγος, τους iterators που προσδιορίζουν το διάστημα όπου τα στοιχεία έχουν «κλειδιά» ισοδύναμα με key.

Παράδειγμα

Παράδειγμα ορισμού, εισαγωγής και προσπέλασης στοιχείων ενός `std::map<>` είναι το ακόλουθο:

```
#include <iostream>
#include <string>
#include <map>
#include <utility>

int
main()
{
    using Map = std::map<std::string, int>;

    Map birthyear; // Empty map.
    // insert a few pairs in different ways:
    birthyear.insert(std::make_pair("John", 1940));
    birthyear.insert(Map::value_type("Paul", 1942));
    birthyear.emplace("George", 1943);

    std::cout << "John␣was␣born␣in␣"
              << birthyear["John"] << "\n\n";

    // insert new pair
    birthyear["Ringo"] = 1941; // wrong value, change it
    birthyear.at("Ringo") = 1940;

    // print all pairs
    for (auto & x : birthyear) {
        std::cout << x.first  << "␣was␣born␣in␣"
                  << x.second << '\n';
    }
}
```

11.7 Unordered associative containers

Οι containers αυτής της κατηγορίας περιλαμβάνουν τους

- `std::unordered_set<>`,
- `std::unordered_multiset<>`,
- `std::unordered_map<>`, και
- `std::unordered_multimap<>`.

Οι συγκεκριμένοι αποθηκεύουν σε πίνακα κατακερματισμού (δείτε το §Β΄.1.3), τιμές οποιουδήποτε τύπου (οι δύο πρώτοι) ή ζεύγη τιμών από «κλειδί» και συνοδεύουσα τιμή (οι δύο επόμενοι). Όπως συμβαίνει και στους αντίστοιχους associative containers, οι `std::unordered_multiset<>` και `std::unordered_multimap<>` επιτρέπουν πολλαπλές ίδιες τιμές (ή με ίδιο «κλειδί»), ενώ οι `std::unordered_set<>` και `std::unordered_map<>` όχι.

Οι unordered containers αποθηκεύουν στοιχεία με τυχαία σειρά, που μπορεί να αλλάξει μετά από εισαγωγή ή διαγραφή στοιχείου. Το σημαντικό τους χαρακτηριστικό είναι ότι προσφέρουν αναζήτηση τιμής μεταξύ των στοιχείων που αποθηκεύουν, με βάση το hash. Είναι πιο γρήγοροι σε αυτή τη διαδικασία σε σύγκριση με τους απλούς associative containers, που εφαρμόζουν μια παραλλαγή του δυαδικού αλγόριθμου, και βέβαια με τους (μη ταξινομημένους) sequence containers που χρειάζονται τον αλγόριθμο γραμμικής αναζήτησης.

Κατά τη δήλωση αντικειμένου ενός container αυτής της κατηγορίας πρέπει να προσδιορίσουμε τον τύπο T των στοιχείων που θα αποθηκευτούν (για unordered set και unordered multiset) ή δύο τύπους K, T που αντιστοιχούν στον τύπο του «κλειδιού» και τις συνοδεύουσας τιμής (για unordered map και unordered multimap). Η επόμενη, προαιρετική παράμετρος καθορίζει τη συνάρτηση hash που θα χρησιμοποιηθεί για την οργάνωση των στοιχείων. Αν δεν ορίσουμε κάποια εμείς, έχει την προκαθορισμένη τιμή `std::hash<>` του `<functional>`, εξειδικευμένη για τον τύπο T ή τον τύπο K. Η συγκεκριμένη συνάρτηση ορίζεται για όλους τους τύπους που παρέχει η C++, είτε ενσωματωμένους είτε της Standard Library. Δεν θα αναφερθούμε στο πώς ορίζουμε συνάρτηση hash για δικές μας κλάσεις. Η τρίτη/τέταρτη παράμετρος προσδιορίζει το κριτήριο ισότητας δύο στοιχείων. Αυτό είναι ο *τύπος* ενός αντικειμένου–συνάρτηση δύο ορισμάτων που επιστρέφει λογική τιμή, **`true`** ή **`false`**, αν το πρώτο όρισμα είναι «ίσο» ή όχι με το δεύτερο. Η συγκεκριμένη παράμετρος έχει την προκαθορισμένη τιμή `std::equal_to<T>` ή `std::equal_to<K>` (§9.3) ώστε στη σύγκριση να χρησιμοποιείται ο τελεστής '`==`' (που θα πρέπει να ορίζεται). Η τελευταία παράμετρος σχετίζεται με το μηχανισμό διαχείρισης μνήμης και έχει προκαθορισμένη τιμή.

Οι unordered associative containers παρέχουν τα κοινά **`typedef`** και τις γνωστές συναρτήσεις–μέλη για το μέγεθός τους. Από τους τελεστές σύγκρισης, παρέχουν μόνο αυτούς που έχουν νόημα για τις συγκεκριμένες συλλογές στοιχείων: τους

τελεστές ισότητας και ανισότητας. Δύο unordered containers είναι ίσοι όταν έχουν όλα τα στοιχεία τους ίσα, με οποιαδήποτε σειρά· αλλιώς είναι άνισοι.

Οι iterators τους είναι μίας κατεύθυνσης (forward iterators). Επομένως, δεν παρέχονται οι τύποι των ανάστροφων iterators και οι σχετικές συναρτήσεις που τους παράγουν. Λάβετε υπόψη ότι σε περίπτωση ανακατανομής των στοιχείων είτε με αυτόματο (μετά από κάποια εισαγωγή στοιχείου) είτε με ρητό rehash, οι iterators (αλλά όχι οι αναφορές) ακυρώνονται.

Όπως είδαμε και στους associative containers, δεν μπορούμε να τροποποιήσουμε απευθείας τα στοιχεία των unordered containers. Αν χρειάζεται κάποια αλλαγή, πρέπει να αφαιρέσουμε το στοιχείο και να το εισαγάγουμε με άλλη τιμή.

Η χρήση των `std::unordered_set<>` και `std::unordered_multiset<>`, προϋποθέτει τη συμπερίληψη του `<unordered_set>`. Αντίστοιχα, ο `<unordered_map>` παρέχει τους `std::unordered_map<>`, `std::unordered_multimap<>`.

Ορισμός

Στα παρακάτω ο τύπος Unord συμβολίζει οποιοδήποτε από τους containers

- `std::unordered_set<T>`,
- `std::unordered_multiset<T>`,
- `std::unordered_map<K,T>`,
- `std::unordered_multimap<K,T>`.

Στις παραμέτρους των templates θεωρούμε ότι περιλαμβάνονται και οι τύποι των συναρτήσεων hash και ισότητας.

Ορισμό ενός Unord έχουμε με τις παρακάτω εντολές:

- Η εντολή

  ```
  Unord c;
  ```

  δημιουργεί κενό Unord.

- Οι εντολές

  ```
  Unord c1;
  Unord c2{c1};
  Unord c3{std::move(c1)};
  ```

  δημιουργούν το Unord c2 ως αντίγραφο του c1 και το Unord c3 με μετακίνηση του c1.

- Ο κώδικας

  ```
  Unord::value_type a1, a2, a3, ...;
  Unord c{a1, a2, a3, ... };
  ```

δημιουργεί Unord με τιμές στοιχείων που προσδιορίζονται από τη λίστα που ακολουθεί το όνομά του.

- Η εντολή

  ```
  Unord c{beg,end};
  ```

  δημιουργεί ένα Unord αντιγράφοντας στοιχεία από κάποιο άλλο container, πιθανώς διαφορετικού τύπου, με iterators του διαστήματος [beg,end).

Οι γνωστοί τρόποι δήλωσης συμπληρώνονται με μηχανισμούς που προσδιορίζουν ως όρισμα των constructors τη συνάρτηση hash, το κριτήριο ισότητας και το αρχικό πλήθος ομάδων οργάνωσης των στοιχείων. Δεν θα αναφερθούμε σε αυτούς περισσότερο.

Διαχείριση μνήμης

Οι unordered containers παρέχουν τις συναρτήσεις–μέλη `size()`, `empty()` και `max_size()` που περιγράψαμε προηγουμένως (§11.4).

Προσθήκη στοιχείων

Εισαγωγή στοιχείων σε ένα Unord γίνεται με τους ακόλουθους τρόπους:

- με εκχώρηση άλλου Unord ίδιου τύπου ή προσωρινής ποσότητας ίδιου τύπου:

  ```
  Unord c1, c2, c3;
   ...
  c2 = c1;
  c3 = std::move(c1);
  ```

- με εκχώρηση λίστας τιμών:

  ```
  Unord c;
  Unord::value_type p1,p2,p3,...;
  ...
  c = {p1, p2, p3, ... };
  ```

- Με εναλλαγή στοιχείων με άλλο Unord ίδιου τύπου, με κλήση είτε της συνάρτησης–μέλους `swap()` είτε της `std::swap()`:

  ```
  Unord c1, c2, c3;
   ...
  c1.swap(c2);
  std::swap(c3,c1);
  ```

- Η εντολή

```
Unord::value_type elem;
...
c.insert(elem);
```

εισάγει στο c ένα νέο στοιχείο με αντιγραφή ή μετακίνηση του elem.

Αν το c είναι unordered_set ή unordered_map, η συνάρτηση επιστρέφει ζεύγος, το πρώτο στοιχείο του οποίου είναι iterator στη θέση του νέου ή του ήδη υπάρχοντος στοιχείου ή ζεύγους, ενώ το δεύτερο είναι λογική τιμή που δείχνει αν η εισαγωγή ήταν επιτυχής ή όχι (στην περίπτωση που υπάρχει στον container ίδιο στοιχείο ή ζεύγος).

Αν το c είναι unordered_multiset ή unordered_multimap, η συνάρτηση επιστρέφει iterator στη θέση του νέου στοιχείου ή ζεύγους.

- Ο κώδικας

```
Unord::value_type elem;
Unord:const_iterator pos;
c.insert(pos,elem);
```

επιχειρεί να εισαγάγει νέο στοιχείο ή ζεύγος στο c με αντιγραφή ή μετακίνηση του elem, λαμβάνοντας υπόψη την *υπόδειξη* μέσω του const_iterator pos. Η συνάρτηση επιστρέφει iterator: αν το c είναι unordered set ή unordered map, δείχνει στο νέο ή υπάρχον στοιχείο. Αν το c είναι unordered multiset ή unordered multimap, δείχνει στο νέο στοιχείο.

- Οι εντολές

```
c.insert(beg, end);
c.insert({p1, p2, p3, ...});
```

εισάγουν στο Unord c, αντίγραφα των στοιχείων στο διάστημα [beg,end) (η πρώτη) και τη λίστα τιμών που δίνεται ως όρισμα στη δεύτερη. Οι συναρτήσεις δεν επιστρέφουν τίποτε.

- Με τη συνάρτηση–μέλος `emplace()` δημιουργείται προσωρινά ένα αντικείμενο κατάλληλου τύπου, από τα ορίσματά της. Το αντικείμενο κατόπιν εισάγεται με μετακίνηση στο Unord για το οποίο καλείται η συνάρτηση. Αν το c είναι unordered multiset ή unordered multimap, η συνάρτηση επιστρέφει iterator στη θέση του νέου στοιχείου ή ζεύγους. Αν το c είναι unordered set ή unordered map, επιστρέφει ζεύγος (std::pair<> (§9.2.1)), το πρώτο μέλος του οποίου είναι iterator στη θέση του νέου ή του ήδη υπάρχοντος στοιχείου ενώ το δεύτερο είναι λογική τιμή που δείχνει αν η εισαγωγή ήταν επιτυχής ή όχι.

- Με τη συνάρτηση–μέλος `emplace_hint()` δημιουργείται προσωρινά ένα αντικείμενο κατάλληλου τύπου, το οποίο κατόπιν εισάγεται με μετακίνηση στο Unord για το οποίο καλείται αυτή. Η συνάρτηση δέχεται ως πρώτο όρισμα

ένα const_iterator· *υποδεικνύεται* έτσι η πιθανή θέση εισαγωγής. Ως δεύτερο, τρίτο, ... όρισμα δέχεται μία ή περισσότερες ποσότητες που τις χρησιμοποιεί για να δημιουργήσει το αντικείμενο προς εισαγωγή.

Η συνάρτηση επιστρέφει iterator: αν το c είναι unordered set ή unordered map, δείχνει στο νέο ή σε υπάρχον στοιχείο. Αν το c είναι unordered multiset ή unordered multimap, δείχνει στο νέο στοιχείο.

Διαγραφή στοιχείων

Διαγραφή στοιχείων από ένα Unord c γίνεται με τους ακόλουθους τρόπους:

- Με την κοινή συνάρτηση–μέλος `clear()`:

  ```
  c.clear();
  ```

  Η κλήση της καταστρέφει όλα τα στοιχεία του c.

- Με τη συνάρτηση–μέλος `erase()`:

  ```
  Unord::const_iterator pos,beg,end;
  c.erase(pos);
  c.erase(beg,end);
  ```

  Στην πρώτη, δέχεται ως όρισμα ένα const_iterator στο Unord για το οποίο καλείται. Διαγράφει το στοιχείο ή ζεύγος στη θέση που δείχνει ο συγκεκριμένος. Επιστρέφει iterator στην επόμενη θέση από το στοιχείο ή ζεύγος που διαγράφηκε. Στη δεύτερη, δέχεται ως όρισμα δυο const_iterators στο Unord για το οποίο καλείται. Διαγράφει τα στοιχεία με θέσεις στο διάστημα [beg,end). Επιστρέφει iterator στην επόμενη θέση από το στοιχείο που διαγράφηκε.

  Με μια παραλλαγή της συνάρτησης `erase()` μπορούμε να διαγράψουμε

  - όλα τα στοιχεία με τιμή «ισοδύναμη» του elem, στην περίπτωση των unordered set και unordered multiset:

    ```
    c.erase(elem);
    ```

  - όλα τα ζεύγη με «κλειδί» key, στην περίπτωση των unordered map και unordered multimap:

    ```
    c.erase(key);
    ```

  Η συνάρτηση επιστρέφει το πλήθος των στοιχείων που διαγράφηκαν. Ο τύπος της επιστρεφόμενης ποσότητας είναι το `size_type`, όπως αυτό ορίζεται από το συγκεκριμένο Unord.

Προσπέλαση στοιχείων

Όπως αναφέραμε, δεν παρέχονται συναρτήσεις για την άμεση πρόσβαση ή τροποποίηση στοιχείων, με την εξαίρεση της `at()` και των αγκυλών '`[]`' που παρέχονται μόνο για unordered map. Δείτε τη σχετική ανάλυση στο map (§11.6.2).

Παρέχεται βέβαια η δυνατότητα προσπέλασης (αλλά όχι τροποποίησης) μέσω iterator.

Επιπλέον συναρτήσεις–μέλη

Οι unordered containers παρέχουν τις ακόλουθες συναρτήσεις για την αναζήτηση ή καταμέτρηση στοιχείων:

- Η συνάρτηση–μέλος `count()` με όρισμα μια τιμή επιστρέφει το πλήθος των στοιχείων στο Unord για το οποίο καλείται, που είναι «ισοδύναμα» (ίσα) με αυτή την τιμή (για unordered set και unordered multiset) ή έχουν αυτό το «κλειδί» (για unordered map και unordered multimap). Ο τύπος της επιστρεφόμενης ποσότητας είναι το `size_type`, όπως αυτό ορίζεται από το συγκεκριμένο Unord.
- Η συνάρτηση–μέλος `find()` με όρισμα μια τιμή εντοπίζει ένα στοιχείο που είναι «ισοδύναμο» με αυτή την τιμή ή να έχει αυτό το «κλεδί». Επιστρέφει iterator στη θέση του στοιχείου, ή αν δεν υπάρχει αυτό, `end()`. Στην περίπτωση που η συνάρτηση κληθεί για σταθερό Unord επιστρέφει const_iterator στη θέση του στοιχείου (ή το `cend()` αν δεν το βρει).
- Η συνάρτηση–μέλος `equal_range()`, με όρισμα μία τιμή, επιστρέφει σε ζεύγος (pair), τους iterators που προσδιορίζουν το διάστημα στο οποίο τα στοιχεία είναι ισοδύναμα με το όρισμα ή έχουν αυτό το «κλειδί».

Επιπλέον, οι unordered containers παρέχουν συναρτήσεις–μέλη για την εξαγωγή της συνάρτησης hash (`hash_function()`), του κριτηρίου ισότητας (`key_eq()`), διαφόρων χαρακτηριστικών της οργάνωσης των στοιχείων:

- `bucket_count()`,
- `max_bucket_count()`,
- `bucket_size()`,
- `bucket()`,
- `load_factor()`,
- `max_load_factor()`,
- `rehash()`,
- `reserve()`.

Δεν θα αναφερθούμε περισσότερο σε αυτές.

11.8 Ασκήσεις

1. Γράψτε συναρτήσεις που να υλοποιούν τις ακόλουθες συναρτήσεις–μέλη της κλάσης `list`: `unique()`, `splice()`, `merge()`, `reverse()`. Οι δικές σας συναρτήσεις θα δέχονται ως πρώτο όρισμα μία αναφορά σε `std::list<T>` με T οποιοδήποτε τύπο (άρα υλοποιήστε τις ως templates). Κατόπιν θα ακολουθούν τα ορίσματα που έχει η αντίστοιχη συνάρτηση–μέλος.

2. Συμπληρώστε τον κώδικα της παρακάτω συνάρτησης

```
template<typename container>
void
split(container const & c, container & odd, container & even)
{
...
}
```

   Η συνάρτηση αυτή θέλουμε να διαβάζει τα στοιχεία ενός container c και να αντιγράφει το πρώτο, τρίτο, πέμπτο,..., στοιχείο στο τέλος του container odd και το δεύτερο, τέταρτο, έκτο,..., στοιχείο στο τέλος του container even.

   Αφού γράψτε τον κώδικα που λείπει, γράψτε ένα πρόγραμμα που θα αποθηκεύει σε `std::list<int>` τους αριθμούς $\{3, 5, -1, 9, -7, 88, 3, -6, -4\}$ και θα τους ξεχωρίζει σε δύο νέες λίστες a, b καλώντας τη συνάρτηση `split()`.

   Κατόπιν, τα στοιχεία κάθε νέας λίστας να τα γράψετε σε αρχείο σε ξεχωριστή γραμμή το καθένα. Τα στοιχεία της λίστας a γράψτε τα στο αρχείο με όνομα *odd.dat* ενώ τα στοιχεία της λίστας b στο αρχείο με όνομα *even.dat*. Να χρησιμοποιήσετε τη συνάρτηση που γράψατε στην άσκηση 7 της σελίδας 210.

3. Γράψτε πρόγραμμα που να αθροίζει δύο ακέραιους με οσαδήποτε ψηφία και να τυπώνει το αποτέλεσμα στην οθόνη. Γι' αυτό το σκοπό:

   - Να γράψετε συνάρτηση που να αναλύει ένα μη αρνητικό ακέραιο (α΄ όρισμα) στα ψηφία του και να τα αποθηκεύει σε container της επιλογής σας (β' όρισμα) με ακέραια στοιχεία. Ο container θα θεωρείται αρχικά κενός.
   - Να γράψετε συνάρτηση που να δέχεται δύο ακολουθίες αριθμών. Αυτές θα ορίζονται από τέσσερις iterators: τους beg1, end1, με ίδιο τύπο, και τους beg2, end2, με ίδιο ή άλλο τύπο. Οι ακολουθίες είναι τα στοιχεία στα διαστήματα [beg1,end1) και [beg2,end2) και αντιπροσωπεύουν τα ψηφία δύο ακεραίων, με πρώτο το ψηφίο των μονάδων. Η συνάρτηση θα υπολογίζει το άθροισμα των δύο ακεραίων και θα αποθηκεύει τα ψηφία του αποτελέσματος σε άλλη ακολουθία που θα ξεκινά από έναν πέμπτο iterator beg3.

Το πρόγραμμα να χρησιμοποιεί τις προηγούμενες συναρτήσεις για να αναλύσει και να προσθέσει τους αριθμούς 1958723584 και 60945983.

4. Γράψτε πρόγραμμα που να πολλαπλασιάζει δύο ακέραιους με οσαδήποτε ψηφία, αποθηκευμένους σε containers της επιλογής σας. Χρησιμοποιήστε τις συναρτήσεις που γράψατε στην προηγούμενη άσκηση.

5. Υλοποιήστε ένα αγγλοελληνικό λεξικό: Ο χρήστης να μπορεί να αναζητά τη μετάφραση οποιασδήποτε λέξης (αγγλικής ή ελληνικής) καθώς και να εισάγει νέες (οι οποίες, βεβαίως, πρέπει να είναι διαθέσιμες σε κάθε νέα εκτέλεση του προγράμματος).

6. Υλοποιήστε έναν τηλεφωνικό κατάλογο: κάθε εγγραφή θα περιλαμβάνει το όνομα, το επώνυμο, τη διεύθυνση (οδός και αριθμός), τον ταχυδρομικό κώδικα, την πόλη και το τηλέφωνο ενός προσώπου. Να παρέχεται η δυνατότητα αναζήτησης και ανάκτησης με βάση το επώνυμο ή το τηλέφωνο, καθώς και η δυνατότητα προσθήκης νέας εγγραφής από το χρήστη.

Κεφάλαιο 12

Αλγόριθμοι της Standard Library

12.1 Εισαγωγή

Η Standard Library παρέχει ένα μεγάλο πλήθος αλγόριθμων για την επεξεργασία συλλογών οποιουδήποτε είδους στοιχείων. Η επεξεργασία συνίσταται σε συνήθεις, θεμελιώδεις πράξεις: αντιγραφή ή τροποποίηση συλλογών, αναζήτηση σε αυτές στοιχείων με συγκεκριμένη ιδιότητα, ταξινόμηση, αναδιάταξη στοιχείων, κλπ. Κάθε αλγόριθμος έχει συγκεκριμένες προδιαγραφές ως προς την ταχύτητά του, απαιτούμενη μνήμη, κλπ. και δεν θα ήταν εύκολο να γράψουμε δικό μας κώδικα που να είναι ταχύτερος ή με λιγότερες απαιτήσεις στη μνήμη.

Οι αλγόριθμοι είναι ανεξάρτητοι από τους containers. Δρουν σε συλλογές στοιχείων που υποδεικνύονται από iterators· έτσι, μπορούμε, χωρίς καμία αλλαγή, να εναλλάσσουμε τους containers που χρησιμοποιούμε για την αποθήκευση των στοιχείων. Βέβαια, αυτό δε σημαίνει ότι θα έχουμε την ίδια απόδοση: ο κάθε container έχει ειδικά χαρακτηριστικά, όπως είδαμε, και πρέπει να επιλέγεται εξ αρχής με βάση τις ανάγκες μας. Θα πρέπει να διευκρινίσουμε ότι η δυνατότητα εναλλαγής των containers δεν είναι απόλυτη: υπάρχουν αλγόριθμοι, π.χ. για ταξινόμηση, που χρειάζονται iterators τυχαίας προσπέλασης (§10.4). Θυμηθείτε ότι οι περισσότεροι containers δεν παρέχουν iterators τέτοιου είδους. Έχουν όμως υπάρχουν συγκεκριμένες συναρτήσεις–μέλη που εκτελούν την απαιτούμενη λειτουργία.

Η πλειοψηφία των αλγορίθμων παρέχονται από το header `<algorithm>`. Οι λίγοι που περιλαμβάνονται στο `<numeric>` θα επισημαίνονται. Όλοι ορίζονται στο χώρο ονομάτων `std`.

Όλοι οι αλγόριθμοι δέχονται ως ορίσματα δύο iterators που καθορίζουν το διάστημα σε ένα container ή γενικότερα σε μια ακολουθία εισόδου, στο οποίο θα δράσουν. Προσέξτε ότι θα πρέπει ο πρώτος iterator να «δείχνει» πριν ή, το πολύ, στην ίδια θέση με το δεύτερο. Η αρχή του διαστήματος προσδιορίζεται από τον

πρώτο iterator ενώ το τέλος του είναι *μία θέση πριν* τη θέση στην οποία «δείχνει» ο δεύτερος. Ανάλογα με τη λειτουργία κάθε αλγόριθμου μπορεί να χρειάζεται να προσδιοριστεί και δεύτερο διάστημα σε ένα container. Σε τέτοια περίπτωση περνά μόνο ο iterator της αρχής (ή του τέλους, ανάλογα με τον αλγόριθμο) και ο προγραμματιστής πρέπει να έχει φροντίσει να ακολουθούν (ή να προηγούνται) αρκετές θέσεις στο δεύτερο container ή ακολουθία ώστε να χωρούν όσα στοιχεία θα γράψει εκεί ο αλγόριθμος. Εναλλακτικά, θα πρέπει να χρησιμοποιηθεί κατάλληλος insert_iterator (§10.9.3) ώστε τα στοιχεία που εγγράφονται να προστίθενται στον container και όχι να αντικαθιστούν τα υπάρχοντα.

Η λειτουργία πολλών αλγορίθμων μπορεί να τροποποιηθεί καθώς έχουν παραλλαγές που δέχονται αντικείμενα–συναρτήσεις είτε με ένα όρισμα (ο τύπος τους θα συμβολίζεται στην περιγραφή τους με το UnaryFunctor) είτε με δύο (ο τύπος τους θα είναι BinaryFunctor). Αυτά τα αντικείμενα–συναρτήσεις επιστρέφουν λογική τιμή (`true/false`). Τα ορίσματά τους πρέπει να είναι ίδιου τύπου με τα στοιχεία του διαστήματος στο οποίο δρα ο αλγόριθμος. Αντί για αντικείμενα–συναρτήσεις μπορούμε να χρησιμοποιήσουμε συναρτήσεις λάμδα ή συνήθεις συναρτήσεις. Προσέξτε ότι αν η συνάρτηση που επιθυμούμε να χρησιμοποιήσουμε είναι template, πρέπει να προσδιορίσουμε μια εκδοχή της· δεν μπορούμε να παραλείψουμε τον προσδιορισμό των παραμέτρων του template καθώς δεν μπορούν να εξαχθούν από τα (ανύπαρκτα) ορίσματα. Καθώς δεν μπορούμε να υποθέσουμε τίποτε για την υλοποίηση των συναρτήσεων που παρέχει η C++ (μπορεί και να μην είναι καν συναρτήσεις αλλά macros) πρέπει να τις χρησιμοποιούμε μέσω δικών μας συναρτήσεων ή συναρτήσεων λάμδα.

Αν δεν αναφερθεί κάτι διαφορετικό ρητά, τα αντικείμενα–συναρτήσεις δεν επιτρέπεται να τροποποιούν τα ορίσματά τους ή την εσωτερική τους κατάσταση.

Ακολουθεί η παρουσίαση των αλγόριθμων της Standard Library. Όλοι είναι υλοποιημένοι ως templates· τα ονόματα των παραμέτρων που χρησιμοποιούνται ως τύποι για iterators υποδηλώνουν τους πιο θεμελιώδεις τύπους iterator που μπορούν να χρησιμοποιηθούν.

12.2 Αριθμητικοί αλγόριθμοι

Οι αλγόριθμοι της συγκεκριμένης κατηγορίας παρέχονται από το `<numeric>`.

12.2.1 `accumulate()`

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

template<typename InputIterator, typename Type,
         typename BinaryFunctor>
```

```
Type
accumulate(InputIterator beg, InputIterator end, Type value,
           BinaryFunctor op);
```

Η πρώτη μορφή του αλγόριθμου επιστρέφει το άθροισμα της τιμής value και των στοιχείων του διαστήματος [beg,end). Η άθροιση γίνεται με τον τελεστή '+'. Αν {a1, a2, a3, ...} είναι τα στοιχεία του διαστήματος, ο αλγόριθμος τροποποιεί και επιστρέφει τελικά το *αντίγραφο* του ορίσματος value, με διαδοχικές προσθέσεις των στοιχείων σε αυτόν:

```
value += a1;
value += a2;
...
```

Στη δεύτερη μορφή, ο αλγόριθμος δέχεται ως επιπλέον τελευταίο όρισμα ένα αντικείμενο–συνάρτηση `op()` που παίρνει δύο ορίσματα. Αυτό προσδιορίζει την εκτελούμενη πράξη μεταξύ των στοιχείων. Ο αλγόριθμος υπολογίζει διαδοχικά τα

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

και επιστρέφει το τελικό value.

Ο αλγόριθμος και στις δύο μορφές επιστρέφει την τιμή value αν το διάστημα είναι κενό, δηλαδή αν `beg==end`.

Παράδειγμα

Ο κώδικας

```
auto sum  = std::accumulate(v.cbegin(), v.cend(), 0.0};
auto prod = std::accumulate(v.cbegin(), v.cend(), 0.0,
                            std::multiplies<double>{}};
```

υπολογίζει το άθροισμα και το γινόμενο των πραγματικών στοιχείων ενός container v.

12.2.2 `inner_product()`

```
template<typename InputIterator, typename Type>
Type
inner_product(InputIterator beg1, InputIterator end1,
              InputIterator beg2, Type value);

template<typename InputIterator, typename Type,
         typename BinaryFunctor>
Type
inner_product(InputIterator beg1, InputIterator end1,
```

```
                InputIterator beg2, Type value,
                BinaryFunctor op1, BinaryFunctor op2);
```

Η πρώτη μορφή του αλγόριθμου επιστρέφει το άθροισμα της value και του εσωτερικού γινομένου των στοιχείων στο [beg1,end1) με τα αντίστοιχά τους στο διάστημα με αρχή το beg2. Η άθροιση γίνεται με τον τελεστή '+' και ο πολλαπλασιασμός με τον τελεστή '*'. Αν {a1, a2, a3, ...} είναι τα στοιχεία του διαστήματος [beg1,end1), και {b1, b2, b3, ...} τα στοιχεία στο διάστημα που αρχίζει με το beg2, ο αλγόριθμος τροποποιεί και επιστρέφει τελικά το *αντίγραφο* του ορίσματος value, με διαδοχικές προσθέσεις του γινομένου των αντίστοιχων στοιχείων:

```
value += a1*b1;
value += a2*b2;
...
```

Στη δεύτερη μορφή, ο αλγόριθμος δέχεται δύο επιπλέον ορίσματα· είναι αντικείμενα–συναρτήσεις δύο ορισμάτων. Το πρώτο αντικείμενο–συνάρτηση προσδιορίζει την εκτελούμενη πράξη αντί για την πρόσθεση και το δεύτερο προσδιορίζει την πράξη στη θέση του πολλαπλασιασμού. Ο αλγόριθμος υπολογίζει διαδοχικά τα

```
value = op1(value,op2(a1,b1));
value = op1(value,op2(a2,b2));
...
```

και επιστρέφει το τελικό value.

Ο αλγόριθμος επιστρέφει την τιμή value αν το πρώτο διάστημα είναι κενό, δηλαδή αν `beg1==end1`.

Παράδειγμα

Ο κώδικας

```
auto s = std::inner_product(v.cbegin(), v.cend(), v.cbegin(),
                            0.0);
```

υπολογίζει το άθροισμα των τετραγώνων των πραγματικών στοιχείων του container με όνομα v.

12.2.3 `partial_sum()`

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

template<typename InputIterator, typename OutputIterator,
         typename BinaryFunctor>
OutputIterator
```

```
partial_sum(InputIterator beg1, InputIterator end1,
            OutputIterator beg2, BinaryFunctor op);
```

Ο συγκεκριμένος αλγόριθμος υπολογίζει το *μερικό άθροισμα* (στην πρώτη μορφή) ή το *γενικευμένο μερικό άθροισμα* (στη δεύτερη). Η πρώτη μορφή εκχωρεί:

- στο στοιχείο που «δείχνει» ο beg2 την τιμή στο beg1,
- στο στοιχείο που «δείχνει» ο beg2+1 το άθροισμα των τιμών στα beg2 και beg1+1,
- στο στοιχείο που «δείχνει» ο beg2+2 το άθροισμα των τιμών στα beg2+1 και beg1+2, κοκ.

Η διαδικασία επαναλαμβάνεται μέχρι να εξαντληθούν τα στοιχεία στο διάστημα [beg1,end1). Ο beg2 μπορεί να ταυτίζεται με τον beg1.

Η δεύτερη μορφή του αλγόριθμου δέχεται ως επιπλέον όρισμα ένα αντικείμενο–συνάρτηση δύο ορισμάτων. Ο αλγόριθμος, αντί για το *άθροισμα* των σχετικών στοιχείων, εκχωρεί την τιμή που επιστρέφεται από το `op()`, δηλαδή, στο στοιχείο της θέσης `beg2+i+1` εκχωρείται η τιμή `op(*(beg2+i), *(beg1+i+1))`.

Και οι δύο μορφές επιστρέφουν iterator στην ακολουθία εξόδου, στην επόμενη θέση από την τελευταία που τροποποιήθηκε.

12.2.4 `adjacent_difference ()`

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

template<typename InputIterator, typename OutputIterator,
         typename BinaryFunctor>
OutputIterator
adjacent_difference(InputIterator beg1, InputIterator end1,
                    OutputIterator beg2, BinaryFunctor op);
```

Ο αλγόριθμος στην πρώτη του μορφή υπολογίζει τη διαφορά κάθε στοιχείου στο διάστημα (beg1, end1) από το προηγούμενό του και εκχωρεί αυτή στο διάστημα που ξεκινά από την επόμενη θέση μετά το beg2. Στη θέση που «δείχνει» ο beg2 εκχωρεί την τιμή στη θέση beg1. Ο beg2 μπορεί να ταυτίζεται με τον beg1.

Η δεύτερη μορφή του αλγόριθμου δέχεται ως επιπλέον όρισμα ένα αντικείμενο–συνάρτηση δύο ορισμάτων. Ο αλγόριθμος, αντί για τη *διαφορά* των διαδοχικών στοιχείων του πρώτου διαστήματος, εκχωρεί την τιμή που επιστρέφεται από το `op()`, δηλαδή, η τιμή `op(*(beg1+i), *(beg1+i+1))` εκχωρείται στο στοιχείο της θέσης `beg2+i+1`.

Και οι δύο μορφές επιστρέφουν iterator στην ακολουθία εξόδου, στην επόμενη θέση από την τελευταία που τροποποιήθηκε.

Ο συγκεκριμένος αλγόριθμος είναι ουσιαστικά ο «αντίστροφος» αυτού που είδαμε στην §12.2.3, του `std::partial_sum()`.

12.3 Αλγόριθμοι ελάχιστου/μέγιστου στοιχείου

Οι παρακάτω αλγόριθμοι παρέχονται από την Standard Library για την εύρεση ελάχιστου ή/και μέγιστου στοιχείου σε μια ακολουθία στοιχείων.

Έχουν δύο μορφές: η σύγκριση των στοιχείων στην πρώτη μορφή γίνεται με τον τελεστή '<'. Στη δεύτερη γίνεται με βάση το αντικείμενο–συνάρτηση `comp()`, το οποίο δέχεται δύο ορίσματα και επιστρέφει **`true`** ή **`false`** αν το πρώτο όρισμά του είναι «μικρότερο» (ό,τι κι αν σημαίνει αυτό) από το δεύτερο ή όχι.

Το `comp()` πρέπει να ικανοποιεί τα κριτήρια της *γνήσιας ασθενούς διάταξης* (§9.4.2).

12.3.1 `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 και στις δύο μορφές 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.

Παράδειγμα

Ο κώδικας

```
auto && f = [](double x, double y) -> double
{ return (std::abs(x) < std::abs(y) ? x : y); }

auto it = std::min_element(v.cbegin(), v.cend(), f);
```

βρίσκει το (πρώτο) στοιχείο με τη μικρότερη απόλυτη τιμή σε ένα container πραγματικών αριθμών με όνομα v. Το στοιχείο «δείχνεται» από τον iterator it και η τιμή του βέβαια είναι `*it`.

12.3.2 `max_element()`

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

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

Η συνάρτηση και στις δύο μορφές επιστρέφει iterator στη θέση του μεγαλύτερου στοιχείου (ή του πρώτου από όλα τα μεγαλύτερα) στο διάστημα [beg,end). Αν το διάστημα είναι κενό (δηλαδή, αν `beg==end`), επιστρέφει το beg.

12.3.3 `minmax_element()`

```
template<typename ForwardIterator>
std::pair<ForwardIterator,ForwardIterator>
minmax_element(ForwardIterator beg, ForwardIterator end);

template<typename ForwardIterator, typename BinaryFunctor>
std::pair<ForwardIterator,ForwardIterator>
minmax_element(ForwardIterator beg, ForwardIterator end,
               BinaryFunctor comp);
```

Η συνάρτηση και στις δύο μορφές επιστρέφει ζεύγος (`std::pair<>`) από iterators. Ο πρώτος είναι στη θέση του μικρότερου στοιχείου (ή του πρώτου από όλα τα μικρότερα) στο διάστημα [beg,end) και ο δεύτερος στο μεγαλύτερο στοιχείο (ή στο τελευταίο από όλα τα μεγαλύτερα). Αν το διάστημα είναι κενό (αν δηλαδή `beg==end`), και οι δύο iterators στο ζεύγος αποκτούν την τιμή beg.

Παράδειγμα

Ο κώδικας

```
auto && f = [](double x, double y) -> double
{ return (std::abs(x) < std::abs(y) ? x : y); }
auto p = std::minmax_element(v.cbegin(), v.cend(), f);
```

βρίσκει το πρώτο στοιχείο με τη μικρότερη απόλυτη τιμή και το τελευταίο στοιχείο με τη μεγαλύτερη απόλυτη τιμή σε ένα container πραγματικών αριθμών με όνομα v. Το στοιχείο με τη μικρότερη απόλυτη τιμή «δείχνεται» από τον iterator `p.first` ενώ το στοιχείο με τη μεγαλύτερη απόλυτη τιμή «δείχνεται» από τον iterator `p.second`.

12.4 Αλγόριθμοι αντιγραφής/μετακίνησης

12.4.1 `copy()`

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

template<typename InputIterator, typename OutputIterator,
         typename UnaryFunctor>
OutputIterator
copy_if(InputIterator beg1, InputIterator end1,
        OutputIterator beg2, UnaryFunctor op);

template<typename InputIterator, typename Size,
         typename OutputIterator>
OutputIterator
copy_n(InputIterator beg1, Size n, OutputIterator beg2);
```

Σε όλες τις μορφές του ο αλγόριθμος *αντιγράφει* τα στοιχεία του διαστήματος [beg1, end1) ή (στην τρίτη μορφή) n διαδοχικά στοιχεία από τη θέση beg1 και μετά, στο διάστημα που ξεκινά με το beg2. Αν το beg1 δείχνει στην ίδια θέση με το end1 ή σε επόμενη θέση (για τις δύο πρώτες μορφές) ή αν το n δεν είναι θετικό για την τρίτη μορφή, 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) (ή στο [beg1, beg1+n) για την τρίτη μορφή). Και οι τρεις μορφές επιστρέφουν iterator στην ακολουθία εξόδου, στην επόμενη θέση από την τελευταία στην οποία έγινε εγγραφή.

Η δεύτερη μορφή του αλγόριθμου δέχεται ένα αντικείμενο–συνάρτηση (ή συνήθη συνάρτηση), `op()`, ενός ορίσματος, το οποίο δεν πρέπει να τροποποιείται από αυτό. Ο αλγόριθμος αντιγράφει στο beg2 και πέρα τα στοιχεία του διαστήματος [beg1, end1) για τα οποία το `op()` επιστρέφει **`true`**.

Στην περίπτωση που τα στοιχεία στο [beg1, end1) δεν πρόκειται να χρησιμοποιηθούν μετά την αντιγραφή με τον αλγόριθμο `std::copy()`, είναι προτιμότερο να γίνει η *μετακίνησή τους* με τον αλγόριθμο `std::move()` (§12.4.2).

12.4.2 `move()`

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

Ο αλγόριθμος *μετακινεί* τα στοιχεία του διαστήματος [beg1, end1) στο διάστημα που ξεκινά με το beg2. Ο iterator beg2 μπορεί να «δείχνει» στον ίδιο container με το

beg1 αλλά δεν επιτρέπεται να ανήκει στο διάστημα [beg1, end1). Επιστρέφει iterator στην ακολουθία εξόδου, στην επόμενη θέση από την τελευταία στην οποία έγινε εγγραφή. Μετά την κλήση του αλγόριθμου, τα στοιχεία στο διάστημα [beg1, end1) έχουν απροσδιόριστη κατάσταση και δεν μπορούν να χρησιμοποιηθούν παρά μόνο για μετακίνηση σε αυτά άλλων στοιχείων. Δείτε για περισσότερα την §2.17.1.

12.4.3 `copy_backward()`

```
template<typename BidirectionalIterator1,
         typename BidirectionalIterator2>
BidirectionalIterator2
copy_backward(BidirectionalIterator1 beg1,
              BidirectionalIterator1 end1,
              BidirectionalIterator2 end2);
```

Ο αλγόριθμος *αντιγράφει* τα στοιχεία του διαστήματος [beg1, end1) στο διάστημα που «τελειώνει» με το end2. Η σειρά τους διατηρείται. Επομένως, το στοιχείο στη θέση `end1-i` αντιγράφεται στη θέση `end2-i`, με $i = 1, 2, \ldots$, έως ότου αντιγραφεί και το στοιχείο στη θέση beg1. Το end2 μπορεί να «δείχνει» στον ίδιο container αλλά δεν πρέπει να ανήκει στο διάστημα (beg1, end1]. Επιστρέφει iterator στην ακολουθία εξόδου, στο τελευταίο στοιχείο στο οποίο έγινε εγγραφή.

Στην περίπτωση που τα στοιχεία στο [beg1, end1) δεν πρόκειται να χρησιμοποιηθούν μετά την αντιγραφή, είναι προτιμότερο να γίνει η *μετακίνησή τους* με τον αλγόριθμο `std::move_backward()` (§12.4.4).

12.4.4 `move_backward()`

```
template<typename BidirectionalIterator1,
         typename BidirectionalIterator2>
BidirectionalIterator2
move_backward(BidirectionalIterator1 beg1,
              BidirectionalIterator1 end1,
              BidirectionalIterator2 end2);
```

Ο αλγόριθμος *μετακινεί* τα στοιχεία του διαστήματος [beg1, end1) στο διάστημα που «τελειώνει» με το end2. Η σειρά τους διατηρείται. Επομένως, το στοιχείο στη θέση `end1-i` μετακινείται στη θέση `end2-i`, με $i = 1, 2, \ldots$, έως ότου μετακινηθεί και το στοιχείο στη θέση beg1. Το end2 μπορεί να «δείχνει» στον ίδιο container αλλά δεν πρέπει να ανήκει στο διάστημα (beg1, end1]. Επιστρέφει iterator στην ακολουθία εξόδου, στο τελευταίο στοιχείο στο οποίο έγινε εγγραφή. Μετά την κλήση του αλγόριθμου, τα στοιχεία στο διάστημα [beg1, end1) έχουν απροσδιόριστη κατάσταση και δεν μπορούν να χρησιμοποιηθούν παρά μόνο για μετακίνηση σε αυτά άλλων στοιχείων. Δείτε για περισσότερα την §2.17.1.

12.5 Αλγόριθμοι περιστροφής

12.5.1 `rotate()`

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

Ο αλγόριθμος περιστρέφει τα στοιχεία στο διάστημα [beg,end) ώστε το στοιχείο στο οποίο δείχνει ο iterator mid να έρθει πρώτο. Επιστρέφει iterator στη θέση που βρίσκεται πλέον το προηγούμενο πρώτο στοιχείο.

12.5.2 `rotate_copy()`

```
template<typename ForwardIterator, typename OutputIterator>
OutputIterator
rotate_copy(ForwardIterator beg, ForwardIterator mid,
            ForwardIterator end, OutputIterator beg2);
```

Ο αλγόριθμος αντιγράφει σε διαδοχικές θέσεις, από το beg2 και μετά, τα στοιχεία του διαστήματος [beg,end), αφού πρώτα περιστρέψει τη σειρά τους ώστε το στοιχείο στο οποίο δείχνει ο iterator mid να έρθει πρώτο.

Επιστρέφει iterator στην ακολουθία εξόδου, στη θέση μετά το τελευταίο στοιχείο που έχει γραφεί.

12.6 Αλγόριθμοι αντικατάστασης

12.6.1 `replace()`

```
template<typename ForwardIterator, typename Type>
void
replace(ForwardIterator beg, ForwardIterator end,
        Type const & oldvalue, Type const & newvalue);

template<typename ForwardIterator, typename Type,
         typename UnaryFunctor>
void
replace_if(ForwardIterator beg, ForwardIterator end,
           UnaryFunctor op, Type const & newvalue);
```

Η πρώτη μορφή του αλγόριθμου εκχωρεί την τιμή newvalue σε κάθε στοιχείο στο διάστημα [beg,end) που είναι ίσο με oldvalue. Η σύγκριση των τιμών γίνεται με τον τελεστή '=='.

Στη δεύτερη μορφή, τα στοιχεία που αντικαθίστανται είναι αυτά για τα οποία το αντικείμενο–συνάρτηση `op()`, ενός ορίσματος, επιστρέφει **`true`**.

Παράδειγμα

Ο κώδικας

```
using type = decltype(v)::value_type;
type zero{0};
std::less<type> f;
auto && lt0 = std::bind(f, std::placeholders::_1, zero);
std::replace_if(v.begin(), v.end(), lt0, zero);
```

αντικαθιστά τα αρνητικά στοιχεία ενός container v με το 0.

Παρατηρήστε ότι χρησιμοποιήθηκε ένας *προσαρμογέας* (§9.3.2) για να τροποποιήσει ένα αντικείμενο–συνάρτηση με δύο ορίσματα ώστε να δέχεται ένα όρισμα: το προκαθορισμένο αντικείμενο–συνάρτηση f με τύπο `std::less<>` που δέχεται δύο ορίσματα, αρχικά δημιουργήθηκε με παράμετρο τον κατάλληλο τύπο και κατόπιν προσαρμόστηκε ώστε να συμπεριφέρεται ως αντικείμενο–συνάρτηση ενός ορίσματος· το δεύτερο όρισμά του απέκτησε την τιμή 0 με τη δράση του `std::bind()`.

Εναλλακτικά, θα μπορούσαμε να χρησιμοποιήσουμε ως τρίτο όρισμα του αλγόριθμου τη συνάρτηση λάμδα

```
auto && lt0 = [zero] (decltype(zero) x) -> bool
        {return x < zero;};
```

12.6.2 `replace_copy()`

```
template<typename InputIterator, typename OutputIterator,
         typename Type>
OutputIterator
replace_copy(InputIterator beg1, InputIterator end1,
             OutputIterator beg2, Type const & oldvalue,
             Type const & newvalue);

template<typename InputIterator, typename OutputIterator,
          typename UnaryFunctor, typename Type>
OutputIterator
replace_copy_if(InputIterator beg1, InputIterator end1,
                OutputIterator beg2, UnaryFunctor op,
                Type const & newvalue);
```

Ο συγκεκριμένος αλγόριθμος συνδυάζει αντιγραφή και αντικατάσταση.

Η πρώτη μορφή του αλγόριθμου αντιγράφει τα στοιχεία του διαστήματος [beg1,

end1) στο διάστημα που ξεκινά με το beg2 αντικαθιστώντας με newvalue όσα στοιχεία είναι ίσα με oldvalue. Η σύγκριση γίνεται με τον τελεστή '=='.

Η δεύτερη μορφή κάνει την αντιγραφή αντικαθιστώντας με newvalue τα στοιχεία για τα οποία το αντικείμενο–συνάρτηση `op()`, ενός ορίσματος, επιστρέφει **`true`**.

Και στις δύο μορφές, ο iterator beg2 δεν επιτρέπεται να «δείχνει» σε θέση του διαστήματος [beg1, end1). Επιστρέφουν iterator στην επόμενη θέση από την τελευταία που γράφτηκε στην ακολουθία εξόδου, αυτή δηλαδή που ξεκινά με το beg2.

Παράδειγμα

Ο κώδικας

```
using type = decltype(a)::value_type;
type constexpr c{1};
auto && g = [c] (type const & x) -> bool
       {return std::abs(x) >= c;};
std::replace_copy_if(a.cbegin(), a.cend(), b.begin(), g, c);
```

αντιγράφει στον container b τα στοιχεία του a. Όσα στοιχεία του a έχουν απόλυτη τιμή (ή μέτρο) μεγαλύτερη ή ίση με 1 αντιγράφονται με τιμή 1.

12.7 Αλγόριθμοι διαγραφής

12.7.1 `remove()`

```
template<typename ForwardIterator, typename Type>
ForwardIterator
remove(ForwardIterator beg, ForwardIterator end, Type const & value);

template<typename ForwardIterator, typename UnaryFunctor>
ForwardIterator
remove_if(ForwardIterator beg, ForwardIterator end, UnaryFunctor op);
```

Η πρώτη μορφή του αλγόριθμου διαγράφει τα στοιχεία του διαστήματος [beg,end) που είναι ίσα με value. Η σύγκριση γίνεται με τον τελεστή '=='.

Στη δεύτερη μορφή του, ο αλγόριθμος δέχεται ένα αντικείμενο–συνάρτηση `op()`, ενός ορίσματος, που επιστρέφει λογική τιμή. Διαγράφει τα στοιχεία για τα οποία το `op()` επιστρέφει **`true`**. Το αντικείμενο–συνάρτηση δεν πρέπει να μεταβάλλει την εσωτερική του κατάσταση κατά την κλήση.

Η διαγραφή των στοιχείων δεν σημαίνει διαγραφή των θέσεών τους. Όταν διαγράφεται ένα στοιχείο μετακινούνται (και δεν αντιγράφονται) τα επόμενα στοιχεία, χωρίς να αλλάξει η σειρά τους. Δείτε την §2.17.1 για τη διαφορά μετακίνησης και αντιγραφής.

Η επιστρεφόμενη τιμή και στις δύο μορφές είναι iterator στην πρώτη θέση μετά το τελευταίο στοιχείο που δεν έχει διαγραφεί. Συνεπώς, το διάστημα από beg μέχρι τον επιστρεφόμενο iterator περιλαμβάνει τα μη διαγραμμένα στοιχεία.

Ο συγκεκριμένος αλγόριθμος μεταβάλλει τα στοιχεία της ακολουθίας εισόδου και επομένως δεν μπορεί να χρησιμοποιηθεί σε associative ή unordered containers. Παρατηρήστε ότι αυτοί παρέχουν παρόμοια συνάρτηση–μέλος (`erase()`).

Θυμηθείτε ότι οι containers `std::list<>` και `std::forward_list<>` παρέχουν τη συνάρτηση–μέλος `remove()`. Είναι αντίστοιχη με τον αλγόριθμο `std::remove()` αλλά πιο γρήγορη, καθώς η μετακίνηση στοιχείων γίνεται αλλάζοντας απλώς δείκτες.

Παράδειγμα

Ο κώδικας

```
using type = decltype(v)::value_type;
auto && ge0 = std::bind(std::greater_equal<type>{},
                        std::placeholders::_1, type{0});
auto end = std::remove_if(v.begin(), v.end(), ge0);
```

τροποποιεί τον container με όνομα v ώστε στο διάστημα `[v.begin(),end)` να υπάρχουν μόνο τα αρνητικά στοιχεία του.

Παρατηρήστε ότι χρησιμοποιήθηκε ένας *προσαρμογέας* (§9.3.2) για να τροποποιήσει ένα αντικείμενο–συνάρτηση με δύο ορίσματα ώστε να δέχεται ένα όρισμα[α΄]. Συγκεκριμένα, το προκαθορισμένο αντικείμενο–συνάρτηση δύο ορισμάτων με τύπο `std::greater_equal<>`, δημιουργήθηκε αρχικά με παράμετρο τον κατάλληλο τύπο και κατόπιν προσαρμόστηκε ώστε να συμπεριφέρεται ως αντικείμενο–συνάρτηση ενός ορίσματος· το δεύτερο όρισμά του απέκτησε την τιμή 0 με τη δράση του `std::bind()`.

[α΄] για την ακρίβεια, για να παράξει ένα άλλο, ανώνυμο αντικείμενο–συνάρτηση, ενός ορίσματος.

12.7.2 `remove_copy()`

```
template<typename InputIterator, typename OutputIterator,
         typename Type>
OutputIterator
remove_copy(InputIterator beg1, InputIterator end1,
            OutputIterator beg2, Type const & value);

template<typename InputIterator, typename OutputIterator,
         typename UnaryFunctor>
OutputIterator
remove_copy_if(InputIterator beg1, InputIterator end1,
               OutputIterator beg2, UnaryFunctor op);
```

Ο συγκεκριμένος αλγόριθμος συνδυάζει αντιγραφή και διαγραφή.

Η πρώτη μορφή του αλγόριθμου αντιγράφει στο διάστημα που αρχίζει με beg2, όσα στοιχεία του διαστήματος [beg,end) δεν είναι ίσα με value. Η σύγκριση γίνεται με τον τελεστή '=='.

Ο αλγόριθμος στη δεύτερη μορφή του, δέχεται ένα αντικείμενο–συνάρτηση `op()`, ενός ορίσματος, που επιστρέφει λογική τιμή. Κατά την αντιγραφή, παραλείπει τα στοιχεία για τα οποία το `op()` επιστρέφει **`true`**.

Και στις δύο μορφές, ο iterator beg2 δεν επιτρέπεται να «δείχνει» σε θέση του διαστήματος [beg1, end1). Η επιστρεφόμενη τιμή και στις δύο μορφές είναι iterator στην ακολουθία εξόδου, στην πρώτη θέση μετά το τελευταίο στοιχείο που δεν έχει αντιγραφεί.

12.7.3 `unique()`

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

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

Ο αλγόριθμος δρα στα στοιχεία ενός διαστήματος [beg, end). Εντοπίζει ομάδες «ίσων» και διαδοχικών στοιχείων στις οποίες «διαγράφει» όλα τα στοιχεία εκτός από το πρώτο σε κάθε ομάδα. Ο έλεγχος για ισότητα στην πρώτη μορφή γίνεται με τον τελεστή '==' ενώ στη δεύτερη γίνεται με το αντικείμενο–συνάρτηση (ή τη συνήθη συνάρτηση) `comp()`. Το `comp()` δέχεται δύο ορίσματα και επιστρέφει **`true`** ή **`false`** αν το πρώτο όρισμά του είναι «ίσο» (ό,τι κι αν σημαίνει αυτό) με το δεύτερο ή όχι. Πρέπει να ικανοποιεί τα κριτήρια στο §9.4.2.

Η διαγραφή στοιχείων δεν σημαίνει ότι αλλάζει το μέγεθος του container ή γενικότερα, της ακολουθίας εισόδου: τα στοιχεία που απομένουν μετακινούνται πάνω σε αυτά που διαγράφονται.

Ο αλγόριθμος επιστρέφει iterator στο νέο τέλος της ακολουθίας, στην πρώτη θέση μετά το τελευταίο στοιχείο που δεν έχει διαγραφεί.

Ο συγκεκριμένος αλγόριθμος μεταβάλλει τα στοιχεία της ακολουθίας εισόδου, και επομένως δεν μπορεί να χρησιμοποιηθεί σε associative ή unordered containers.

Θυμηθείτε ότι οι κλάσεις `std::list<>` και `std::forward_list<>` παρέχουν τη συνάρτηση–μέλος `unique()`. Είναι αντίστοιχη με τον αλγόριθμο `std::unique()` αλλά πιο γρήγορη, καθώς η διαγραφή στοιχείων γίνεται αλλάζοντας απλώς δείκτες.

12.7.4 `unique_copy()`

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

template<typename InputIterator, typename OutputIterator,
         typename BinaryFunctor>
OutputIterator
unique_copy(InputIterator beg1, InputIterator end1,
            OutputIterator beg2, BinaryFunctor comp);
```

Ο αλγόριθμος είναι παρόμοιος με τον `std::unique()`. Αντιγράφει στην ακολουθία εξόδου, από το beg2 και μετά, τα στοιχεία της ακολουθίας εισόδου, μεταξύ beg1 και end1, παραλείποντας τα «ίσα», διαδοχικά στοιχεία εκτός από το πρώτο κάθε φορά.

Ο έλεγχος για ισότητα στην πρώτη μορφή γίνεται με τον τελεστή '==' ενώ στη δεύτερη γίνεται με το αντικείμενο–συνάρτηση `comp()`.

Ο αλγόριθμος επιστρέφει iterator στην ακολουθία εξόδου, στην επόμενη θέση από την τελευταία που έχει γραφεί.

12.8 Αλγόριθμοι αναστροφής

12.8.1 `reverse()`

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

Αναστρέφει τη σειρά των στοιχείων του διαστήματος [beg,end).

Προσέξτε ότι οι `std::list<>` και `std::forward_list<>` παρέχουν γρηγορότερη συνάρτηση–μέλος.

12.8.2 `reverse_copy()`

```
template<typename BidirectionalIterator>
OutputIterator
reverse_copy(BidirectionalIterator beg1, BidirectionalIterator end1,
             OutputIterator beg2);
```

Αναστρέφει τη σειρά των στοιχείων του διαστήματος [beg1,end1) αντιγράφοντάς τα στο διάστημα που ξεκινά με το beg2. Επιστρέφει iterator στην ακολουθία εξόδου, στη θέση μετά το τελευταίο στοιχείο που έχει γραφεί.

12.9 Αλγόριθμοι τυχαίας αναδιάταξης

12.9.1 `random_shuffle()`

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

template<typename RandomIterator, typename RNG>
void
random_shuffle(RandomIterator beg, RandomIterator end, RNG && rng);
```

Αναδιατάσσει με τυχαίο τρόπο τα στοιχεία στο διάστημα [beg,end).

Στην πρώτη μορφή χρησιμοποιεί εσωτερική γεννήτρια τυχαίων αριθμών. Στη δεύτερη, δέχεται ως τρίτο όρισμα το `rng()`, ένα αντικείμενο–συνάρτηση (ή μια συνήθη συνάρτηση) ενός ορίσματος. Αυτό πρέπει να επιστρέφει τυχαίο μη αρνητικό αριθμό, μικρότερο από το όρισμά του. Το όρισμα και η ποσότητα που επιστρέφεται πρέπει να είναι ακέραιου τύπου[1]. Το `rng()` χρησιμοποιείται από τον αλγόριθμο για την αναδιάταξη.

Καλό είναι να μη χρησιμοποιείται ο συγκεκριμένος αλγόριθμος. Είναι προτιμότερο να αναδιατάσσουμε με τυχαίο τρόπο μια ακολουθία με τον επόμενο αλγόριθμο, `std::shuffle()`.

12.9.2 `shuffle()`

```
template<typename RandomIterator, typename URNG>
void
shuffle(RandomIterator beg, RandomIterator end, URNG && g);
```

Ο αλγόριθμος αναδιατάσσει με τυχαίο τρόπο τα στοιχεία στο διάστημα [beg,end). Η επιλογή της σειράς των στοιχείων γίνεται χρησιμοποιώντας γεννήτρια τυχαίων αριθμών ως τρίτο όρισμα. Η γεννήτρια πρέπει να είναι από αυτές που παρέχονται από το header `<random>`.

12.10 Αλγόριθμοι διαμοίρασης

Οι αλγόριθμοι αυτής της κατηγορίας διαμοιράζουν ή ελέγχουν τη διαμοίραση μιας συλλογής στοιχείων. Δέχονται ως τρίτο όρισμα ένα αντικείμενο–συνάρτηση (ή μία συνήθη συνάρτηση ή συνάρτηση λάμδα) `op()`. Το `op()` δέχεται ένα όρισμα που δεν επιτρέπεται να το τροποποιεί και επιστρέφει λογική τιμή, **`true`** ή **`false`**.

[1] `typename std::iterator_traits<RandomIterator>::difference_type`

12.10.1 `partition()`

```
template<typename ForwardIterator, typename UnaryFunctor>
ForwardIterator
partition(ForwardIterator beg, ForwardIterator end, UnaryFunctor op);
```

Ο αλγόριθμος αναδιατάσσει τα στοιχεία στο διάστημα [beg,end) ώστε αυτά για τα οποία το `op()` επιστρέφει **`true`** να βρίσκονται πριν από ώστε αυτά για τα οποία το `op()` επιστρέφει **`false`**. Η σχετική θέση όσων στοιχείων ανήκουν στις δύο ομάδες δεν διατηρείται απαραίτητα.

Επιστρέφει iterator στο πρώτο στοιχείο για το οποίο το `op()` έχει τιμή **`false`** ή end αν δεν βρεθεί τέτοιο στοιχείο.

Παράδειγμα

Ο κώδικας

```
using type = decltype(v)::value_type;
type zero{0};
auto && gt0 = std::bind(std::greater<type>{},
                        std::placeholders::_1, zero);
auto mid = std::partition(v.begin(), v.end(), gt0);
```

διαχωρίζει τα στοιχεία του container v σε θετικά, στο [v.begin(),mid), και μη θετικά, στο [mid,v.end()).

Παρατηρήστε ότι χρησιμοποιήθηκε ένας *προσαρμογέας* (§9.3.2) για να τροποποιήσει ένα αντικείμενο–συνάρτηση με δύο ορίσματα ώστε να δέχεται ένα όρισμα: αρχικά δημιουργήθηκε το προκαθορισμένο αντικείμενο–συνάρτηση δύο ορισμάτων με τύπο `std::greater<>`, έχοντας ως παράμετρο τον κατάλληλο τύπο, και κατόπιν προσαρμόστηκε ώστε να συμπεριφέρεται ως αντικείμενο–συνάρτηση ενός ορίσματος· το δεύτερο όρισμά του απέκτησε την τιμή 0 με τη δράση του `std::bind()`.

12.10.2 `stable_partition()`

```
template<typename BidirectionalIterator, typename UnaryFunctor>
BidirectionalIterator
stable_partition(BidirectionalIterator beg,
                 BidirectionalIterator end, UnaryFunctor op);
```

Ο αλγόριθμος είναι όμοιος με τον `std::partition()` (§12.10.1) αλλά διατηρεί τις σχετικές θέσεις των στοιχείων που, ως ορίσματα του `op()`, αντιστοιχούν σε **`true`** και σε **`false`**.

12.10.3 `is_partitioned()`

```
template<typename InputIterator, typename UnaryFunctor>
bool
is_partitioned(InputIterator beg, InputIterator end,
               UnaryFunctor op);
```

Ο αλγόριθμος ελέγχει αν τα στοιχεία στο διάστημα [beg,end) είναι διαμοιρασμένα ανάλογα με το αποτέλεσμα της δράσης του `op()`. Αν όλα τα στοιχεία για τα οποία η δράση του `op()` δίνει **`true`** βρίσκονται πριν από αυτά με αποτέλεσμα **`false`**, ο αλγόριθμος επιστρέφει **`true`**. Δεν διαφορετική περίπτωση επιστρέφει **`false`**. Αν το διάστημα είναι κενό, δηλαδή αν `beg==end`, επιστρέφει **`true`**.

12.10.4 `partition_point()`

```
template<typename ForwardIterator, typename UnaryFunctor>
ForwardIterator
partition_point(ForwardIterator beg, ForwardIterator end,
                UnaryFunctor op);
```

Ο αλγόριθμος δέχεται μια ακολουθία στοιχείων στο διάστημα [beg,end). Η ακολουθία θεωρείται ως ήδη διαμοιρασμένη με τη δράση του `op()`, δηλαδή, θα μπορούσε να έχει προκύψει από την κλήση

```
std::partition(beg,end,op);
```

Επιστρέφει iterator στο πρώτο στοιχείο για το οποίο το `op()` επιστρέφει **`false`**. Αν το διάστημα είναι κενό, επιστρέφει το end.

12.10.5 `partition_copy()`

```
template<typename InputIterator, typename OutputIterator1,
         typename OutputIterator2, typename UnaryFunctor>
std::pair<OutputIterator1,OutputIterator2>
partition_copy(InputIterator beg1, InputIterator end1,
               OutputIterator1 beg2, OutputIterator2 beg3,
               UnaryFunctor op);
```

Ο αλγόριθμος δρα στα στοιχεία του διαστήματος [beg1,end1). Αντιγράφει αυτά για τα οποία το `op()` δίνει **`true`**, στις θέσεις της πρώτης ακολουθίας εξόδου, από το beg2 και μετά. Τα στοιχεία για τα οποία το `op()` επιστρέφει **`false`** αντιγράφονται στη δεύτερη ακολουθία εξόδου, από το beg3 και μετά.

Ο αλγόριθμος επιστρέφει ζεύγος (`std::pair<>`) με δύο iterators, ένα σε κάθε μια ακολουθία εξόδου. Κάθε iterator «δείχνει» στην επόμενη θέση από αυτή που έχει γραφτεί τελευταία.

12.11 Αλγόριθμοι ταξινόμησης

Οι παρακάτω αλγόριθμοι ταξινομούν ή ελέγχουν αν είναι ταξινομημένη, μια συλλογή στοιχείων, ολικά ή μερικά.

Έχουν δύο μορφές: η σύγκριση των στοιχείων στην πρώτη μορφή γίνεται με τον τελεστή '<'. Στη δεύτερη γίνεται με βάση το αντικείμενο–συνάρτηση `comp()`, το οποίο δέχεται δύο ορίσματα και επιστρέφει **`true`** ή **`false`** αν το πρώτο όρισμά του είναι «μικρότερο» (ό,τι κι αν σημαίνει αυτό) από το δεύτερο ή όχι. Το `comp()` πρέπει να ικανοποιεί τα κριτήρια στο §9.4.2.

12.11.1 `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 σχετική θέση ισοδύναμων στοιχείων δεν διατηρείται απαραίτητα.

Προσέξτε ότι ο αλγόριθμος χρειάζεται random iterators οπότε δεν μπορεί να εφαρμοστεί σε `std::list<>` ή `std::forward_list<>`. Θυμηθείτε ότι υπάρχει κατάλληλη συνάρτηση–μέλος αυτών των containers που εκτελεί ταξινόμηση.

12.11.2 `stable_sort()`

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

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

Ο συγκεκριμένος αλγόριθμος είναι ίδιος με τον `std::sort()` με τη διαφορά ότι η σχετική θέση ισοδύναμων στοιχείων διατηρείται.

12.11.3 `nth_element()`

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

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

Ο αλγόριθμος δέχεται μια συλλογή στοιχείων στο διάστημα [beg, end) και τα μετακινεί έτσι ώστε στον iterator nth (που πρέπει να είναι μεταξύ beg, end) να μεταφερθεί το στοιχείο που θα βρισκόταν εκεί αν η ακολουθία στο [beg,end) ήταν ταξινομημένη. Επιπλέον, τα στοιχεία στο [beg,nth) είναι μικρότερα ή ίσα από το `*nth`.

12.11.4 `partial_sort()`

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

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

Ο συγκεκριμένος αλγόριθμος είναι ίδιος με τον `std::sort()` αλλά σταματά την ταξινόμηση όταν τοποθετηθούν στο [beg,sortEnd) σωστά ταξινομημένα τα στοιχεία.

12.11.5 `partial_sort_copy()`

```
template<typename InputIterator, typename RandomIterator>
RandomIterator
partial_sort_copy(InputIterator beg1, InputIterator end1,
                  RandomIterator beg2, RandomIterator end2);

template<typename InputIterator, typename RandomIterator,
         typename BinaryFunctor>
RandomIterator
partial_sort_copy(InputIterator beg1, InputIterator end1,
                  RandomIterator beg2, RandomIterator end2,
                  BinaryFunctor comp);
```

Ο αλγόριθμος αντιγράφει ταξινομημένα τα στοιχεία του διαστήματος [beg1, end1) στο διάστημα [beg2,end2). Αν το [beg2,end2) έχει λιγότερες θέσεις από το [beg1,end1) αντιγράφονται όσα χωρούν, ξεκινώντας από το «μικρότερο».

Επιστρέφει iterator στη ακολουθία εξόδου, επόμενη θέση από την τελευταία που έχει γραφεί.

12.11.6 `is_sorted()`

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

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

Και οι δύο μορφές ελέγχουν αν τα στοιχεία στο διάστημα [beg,end) είναι ταξινομημένα. Αν είναι ταξινομημένα ή είναι λιγότερα από δύο, επιστρέφουν `true`, αλλιώς επιστρέφουν `false`.

12.11.7 `is_sorted_until()`

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

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

Και οι δύο μορφές ελέγχουν αν τα στοιχεία στο διάστημα [beg,end) είναι ταξινομημένα. Επιστρέφουν iterator στο πρώτο στοιχείο που δεν ακολουθεί την ταξινόμηση ή end αν όλα είναι στη σειρά (ή λιγότερα από δύο).

12.12 Αλγόριθμοι μετάθεσης

Ένα σύνολο N στοιχείων μπορεί να διαταχθεί με $N!$ διαφορετικούς τρόπους. Μπορούμε να ορίσουμε μια σειρά σε αυτές τις διαφορετικές διατάξεις, ανάλογα με το πώς συγκρίνονται μεταξύ τους *λεξικογραφικά* (§9.4.1). Η συγκεκριμένη σύγκριση υλοποιείται από τον αλγόριθμο `std::lexicographical_compare()`.

Από τις $N!$ διαφορετικές διατάξεις, μπορούμε να θεωρήσουμε ως πρώτη, «μικρότερη» διάταξη αυτή που έχει τα στοιχεία ταξινομημένα από το «μικρότερο» στο «μεγαλύτερο». Ως τελευταία, «μεγαλύτερη» διάταξη αυτή που τα έχει ταξινομημένα αντίστροφα. Η παραγωγή μιας ενδιάμεσης διάταξης μπορεί να γίνει τροποποιώντας την αμέσως προηγούμενη διάταξή της, σύμφωνα με τον ακόλουθο αλγόριθμο[2]:

1. Έστω $p_1 p_2 \dots p_n$ η τρέχουσα διάταξη.

2. Εντοπίζουμε το μεγαλύτερο δείκτη k ώστε $p_k < p_{k+1}$. Αν δεν υπάρχει τέτοιος, η τρέχουσα διάταξη είναι η μεγαλύτερη.

3. Εντοπίζουμε το μεγαλύτερο δείκτη $i > k$ τέτοιον ώστε $p_k < p_i$.

4. Εναλλάσσουμε τα p_k, p_i.

5. Αναστρέφουμε την ακολουθία των στοιχείων από το p_{k+1} ως το p_n.

Ο αλγόριθμος αυτός υλοποιείται στη σχετική συνάρτηση που παρέχει η Standard Library, την `std::next_permutation()`, που παρουσιάζεται παρακάτω.

12.12.1 `lexicographical_compare()`

```
template<typename InputIterator1, typename InputIterator2>
bool
lexicographical_compare(InputIterator1 beg1, InputIterator1 end1,
                        InputIterator2 beg2, InputIterator2 end2);

template<typename InputIterator1, typename InputIterator2,
         typename BinaryFunctor>
bool
lexicographical_compare(InputIterator1 beg1, InputIterator1 end1,
                        InputIterator2 beg2, InputIterator2 end2,
                        BinaryFunctor comp);
```

Ο αλγόριθμος συγκρίνει *λεξικογραφικά* δύο ακολουθίες στοιχείων στα διαστήματα [beg1,end1) και [beg2, end2). Όταν η πρώτη ακολουθία είναι «μικρότερη» από τη δεύτερη επιστρέφει `true`. Σε άλλη περίπτωση επιστρέφει `false`.

Η σύγκριση στοιχείων δεν γίνεται με τον τελεστή '`==`' μεταξύ δύο στοιχείων a,b. Η ισότητα ή μη, ελέγχεται με χρήση του τελεστή '`<`', στην πρώτη μορφή του αλγόριθμου, με τον κώδικα `!(a<b) && !(b<a)`. Στη δεύτερη, η σύγκριση γίνεται με το αντικείμενο–συνάρτηση `comp()`, με τον κώδικα `!comp(a,b) && !comp(b,a)`.

[2] Algorithm L, [5]

12.12.2 `next_permutation()`

```
template<typename BidirectionalIterator>
bool
next_permutation(BidirectionalIterator beg,
                 BidirectionalIterator end);

template<typename BidirectionalIterator, typename BinaryFunctor>
bool
next_permutation(BidirectionalIterator beg,
                 BidirectionalIterator end, BinaryFunctor comp);
```

Ο αλγόριθμος δέχεται μια ακολουθία στοιχείων στο διάστημα [beg,end). Αν τα στοιχεία είναι στη λεξικογραφικά μεγαλύτερη μετάθεση, τα αναδιατάσσει στη λεξικογραφικά μικρότερη μετάθεση και επιστρέφει **false**. Αλλιώς, τα αναδιατάσσει στην επόμενη, λεξικογραφικά αμέσως μεγαλύτερη, μετάθεση και επιστρέφει **true**.

Η σύγκριση των στοιχείων στην πρώτη μορφή γίνεται με τον τελεστή '<'. Στη δεύτερη γίνεται με βάση το αντικείμενο–συνάρτηση `comp()`, το οποίο δέχεται δύο ορίσματα και επιστρέφει **true** ή **false** αν το πρώτο όρισμά του είναι «μικρότερο» (ό,τι κι αν σημαίνει αυτό) από το δεύτερο ή όχι. Το `comp()` πρέπει να ικανοποιεί τα κριτήρια στο §9.4.2.

12.12.3 `prev_permutation()`

```
template<typename BidirectionalIterator>
bool
prev_permutation(BidirectionalIterator beg,
                 BidirectionalIterator end);

template<typename BidirectionalIterator, typename BinaryFunctor>
bool
prev_permutation(BidirectionalIterator beg,
                 BidirectionalIterator end, BinaryFunctor comp);
```

Ο αλγόριθμος δέχεται μια ακολουθία στοιχείων στο διάστημα [beg,end). Αν τα στοιχεία είναι στη λεξικογραφικά μικρότερη μετάθεση, τα αναδιατάσσει στη λεξικογραφικά μεγαλύτερη μετάθεση και επιστρέφει **false**. Αλλιώς, τα αναδιατάσσει στην προηγούμενη, λεξικογραφικά αμέσως μικρότερη, μετάθεση και επιστρέφει **true**.

Η σύγκριση των στοιχείων στην πρώτη μορφή γίνεται με τον τελεστή '<'. Στη δεύτερη γίνεται με βάση το αντικείμενο–συνάρτηση `comp()`, το οποίο δέχεται δύο ορίσματα και επιστρέφει **true** ή **false** αν το πρώτο όρισμά του είναι «μικρότερο» (ό,τι κι αν σημαίνει αυτό) από το δεύτερο ή όχι. Το `comp()` πρέπει να ικανοποιεί τα κριτήρια στο §9.4.2.

12.12.4 `is_permutation()`

```
template<typename ForwardIterator1, typename ForwardIterator2>
bool
is_permutation(ForwardIterator1 beg1, ForwardIterator1 end1,
               ForwardIterator2 beg2);

template<typename ForwardIterator1, typename ForwardIterator2,
         typename BinaryFunctor>
bool
is_permutation(ForwardIterator1 beg1, ForwardIterator1 end1,
               ForwardIterator2 beg2, BinaryFunctor comp);

template<typename ForwardIterator1, typename ForwardIterator2>
bool
is_permutation(ForwardIterator1 beg1, ForwardIterator1 end1,
               ForwardIterator2 beg2, ForwardIterator2 end2);

template<typename ForwardIterator1, typename ForwardIterator2,
         typename BinaryFunctor>
bool
is_permutation(ForwardIterator1 beg1, ForwardIterator1 end1,
               ForwardIterator2 beg2, ForwardIterator2 end2,
               BinaryFunctor comp);
```

Ο αλγόριθμος ελέγχει αν όλα τα στοιχεία του διαστήματος [beg1,end1) περιέχονται με οποιαδήποτε σειρά, σε ισάριθμες θέσεις από beg2 και μετά (στην πρώτη και δεύτερη μορφή) ή στο διάστημα [beg2,end2) (στην τρίτη και τέταρτη μορφή). Επιστρέφει λογική τιμή.

Στην πρώτη και τρίτη μορφή η σύγκριση των στοιχείων γίνεται με τον τελεστή '==' ενώ στη δεύτερη και τέταρτη μορφή γίνεται με το αντικείμενο–συνάρτηση `comp()`. Το `comp()` δέχεται δύο ορίσματα και επιστρέφει **`true`** ή **`false`** αν το πρώτο όρισμά του είναι «ίσο» (ό,τι κι αν σημαίνει αυτό) με το δεύτερο ή όχι.

12.13 Αλγόριθμοι αναζήτησης

Η C++ παρέχει πληθώρα αλγορίθμων αναζήτησης. Όλοι τους έχουν δύο μορφές. Αυτοί που δρουν σε *ταξινομημένο ή διαμοιρασμένο διάστημα*, θεωρούν, στην πρώτη τους μορφή, ότι η ταξινόμηση έχει γίνει με τον τελεστή '<'. Στη δεύτερη, ο αλγόριθμος δέχεται ως όρισμα ένα αντικείμενο–συνάρτηση `comp()` που προσδιορίζει το κριτήριο με το οποίο έγινε η ταξινόμηση. Το `comp()` δέχεται δύο ορίσματα και επιστρέφει **`true`** ή **`false`** αν το πρώτο όρισμά του είναι «μικρότερο» (ό,τι κι αν σημαίνει αυτό) από το δεύτερο ή όχι.

Οι αλγόριθμοι που δρουν σε *μη ταξινομημένο διάστημα* κάνουν αναζήτηση με τον τελεστή '==' στην πρώτη τους μορφή. Στη δεύτερη μορφή οι περισσότεροι δέχονται ένα αντικείμενο–συνάρτηση `comp()` δύο ορισμάτων. Το `comp()` επιστρέφει **`true`** ή **`false`** αν το πρώτο όρισμά του είναι «ίσο» (ό,τι κι αν σημαίνει αυτό) με το δεύτερο ή όχι.

12.13.1 `find()`

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

template<typename InputIterator, typename UnaryFunctor>
InputIterator
find_if(InputIterator beg, InputIterator end, UnaryFunctor op);

template<typename InputIterator, typename UnaryFunctor>
InputIterator
find_if_not(InputIterator beg, InputIterator end, UnaryFunctor op);
```

Η πρώτη μορφή της συνάρτησης επιστρέφει iterator στη θέση του πρώτου στοιχείου στο [beg,end) που είναι ίσο με value. Η σύγκριση γίνεται με τον τελεστή '==. Θυμηθείτε ότι οι associative containers παρέχουν ανάλογη συνάρτηση–μέλος που είναι πιο γρήγορη.

Η δεύτερη και η τρίτη μορφή δέχονται ένα ένα αντικείμενο–συνάρτηση `op()`, ενός ορίσματος. Η `std::find_if()` επιστρέφει iterator στη θέση του πρώτου στοιχείου για το οποίο η δράση του αντικειμένου–συνάρτηση `op()` επιστρέφει **`true`**. Η `std::find_if_not()` επιστρέφει iterator στο πρώτο στοιχείο για το οποίο η `op()` επιστρέφει **`false`**.

Και στις τρεις μορφές επιστρέφεται 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. Δείτε παρακάτω τις συναρτήσεις `std::upper_bound()`, `std::lower_bound()`, `std::equal_range()`.

Οι associative και unordered containers παρέχουν γρηγορότερη συνάρτηση–μέλος, τη `find()`, με ίδια λειτουργικότητα.

Παράδειγμα

Ο κώδικας

```
auto && lta = std::bind(std::less<decltype(a)>{},
                        std::placeholders::_1, a);
auto it = std::find_if(v.cbegin(), v.cend(), lta);
```

εντοπίζει το πρώτο στοιχείο ενός container v, που είναι μικρότερο από μια τιμή a. Αν δεν υπάρχει τέτοιο, ο iterator it παίρνει την τιμή `v.cend()`.

12.13.2 `find_first_of()`

```
template<typename InputIterator, typename ForwardIterator>
InputIterator
find_first_of(InputIterator beg1, InputIterator end1,
              ForwardIterator beg2, ForwardIterator end2);

template<typename InputIterator, typename ForwardIterator,
         typename BinaryFunctor>
InputIterator
find_first_of(InputIterator beg1, InputIterator end1,
              ForwardIterator beg2, ForwardIterator end2,
              BinaryFunctor comp);
```

Ο αλγόριθμος αναζητά οποιοδήποτε στοιχείο του διαστήματος [beg2,end2) στο διάστημα [beg1,end1). Επιστρέφει iterator στην ακολουθία [beg1,end1), στη θέση του πρώτου στοιχείου από τα αναζητούμενα. Αν δεν βρεθεί κανένα στοιχείο επιστρέφει end1.

Στην πρώτη μορφή του αλγόριθμου, η σύγκριση των στοιχείων γίνεται με τον τελεστή '=='. Στη δεύτερη, γίνεται με το αντικείμενο–συνάρτηση `comp()`.

12.13.3 `search()`

```
template<typename ForwardIterator1, typename ForwardIterator2>
ForwardIterator1
search(ForwardIterator1 beg1, ForwardIterator1 end1,
       ForwardIterator2 beg2, ForwardIterator2 end2);

template<typename ForwardIterator1, typename ForwardIterator2,
         typename BinaryFunctor>
ForwardIterator1
search(ForwardIterator1 beg1, ForwardIterator1 end1,
       ForwardIterator2 beg2, ForwardIterator2 end2,
       BinaryFunctor comp);
```

Ο αλγόριθμος αναζητά στο διάστημα [beg1,end1) την πρώτη εμφάνιση της ακολουθίας στοιχείων του διαστήματος [beg2,end2) (με την ίδια ακριβώς σειρά).

Στην πρώτη μορφή του αλγόριθμου η σύγκριση των στοιχείων γίνεται με τον τελεστή '=='. Στη δεύτερη, γίνεται με το αντικείμενο–συνάρτηση `comp()`.

Ο αλγόριθμος επιστρέφει iterator μεταξύ beg1 και end1, στην πρώτη θέση που εντοπίστηκε η ζητούμενη ακολουθία, στο αρχικό στοιχείο. Αν η ακολουθία δεν βρεθεί, επιστρέφει end1. Αν η ακολουθία [beg2,end2) είναι κενή, δηλαδή αν `beg2==end2`, ο αλγόριθμος επιστρέφει τον iterator beg1.

12.13.4 `find_end()`

```
template<typename ForwardIterator1, typename ForwardIterator2>
ForwardIterator1
find_end(ForwardIterator1 beg1, ForwardIterator1 end1,
         ForwardIterator2 beg2, ForwardIterator2 end2);

template<typename ForwardIterator1, typename ForwardIterator2,
         typename BinaryFunctor>
ForwardIterator1
find_end(ForwardIterator1 beg1, ForwardIterator1 end1,
         ForwardIterator2 beg2, ForwardIterator2 end2,
         BinaryFunctor comp);
```

Ο αλγόριθμος είναι παρόμοιος με τον `std::search()`: αναζητά στο διάστημα [beg1, end1) την *τελευταία* εμφάνιση της ακολουθίας στοιχείων του διαστήματος [beg2, end2) (με την ίδια ακριβώς σειρά). Όπως και στον αλγόριθμο `std::search()`, η σύγκριση των στοιχείων γίνεται με τον τελεστή '==' ή το αντικείμενο–συνάρτηση `comp()`.

Ο αλγόριθμος επιστρέφει iterator μεταξύ beg1 και end1, στην τελευταία θέση που εντοπίστηκε η ζητούμενη ακολουθία, στο αρχικό στοιχείο. Αν η ακολουθία δεν βρεθεί, επιστρέφει end1. Αν η ακολουθία [beg2,end2) είναι κενή, δηλαδή αν `beg2==end2`, ο αλγόριθμος επιστρέφει τον iterator end1.

12.13.5 `adjacent_find()`

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

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

Ο αλγόριθμος αναζητά στο διάστημα [beg,end) την πρώτη δυάδα διαδοχικών και ίσων στοιχείων. Επιστρέφει iterator στη θέση του πρώτου στοιχείου από τη δυάδα των ίσων, διαδοχικών στοιχείων ή το end αν δεν βρέθηκε τέτοια.

12.13.6 `search_n()`

```
template<typename ForwardIterator, typename Size, typename Type>
ForwardIterator
search_n(ForwardIterator beg, ForwardIterator end, Size n,
         Type const & value);

template<typename ForwardIterator, typename Size, typename Type,
         typename BinaryFunctor>
ForwardIterator
search_n(ForwardIterator beg, ForwardIterator end,
         Size n, Type const & value, BinaryFunctor comp);
```

Ο αλγόριθμος αναζητά στο διάστημα [beg,end) την πρώτη ομάδα n διαδοχικών στοιχείων που είναι ίσα με value. Επιστρέφει iterator στη θέση του πρώτου στοιχείου από την ομάδα των n διαδοχικών στοιχείων ή το end αν δεν βρέθηκε τέτοια ομάδα.

12.13.7 `binary_search()`

```
template<typename ForwardIterator, typename Type>
bool
binary_search(ForwardIterator beg, ForwardIterator end,
              Type const & value);

template<typename ForwardIterator, typename Type,
         typename BinaryFunctor>
bool
binary_search(ForwardIterator beg, ForwardIterator end,
              Type const & value, BinaryFunctor comp);
```

Η συνάρτηση και στις δύο μορφές επιστρέφει `true` ή `false` αν τα στοιχεία στο διάστημα [beg,end) περιλαμβάνουν τιμή ίση με value ή όχι. Τα στοιχεία του διαστήματος *θεωρούνται ταξινομημένα* από το «μικρότερο» στο «μεγαλύτερο», είτε με τον τελεστή '<' (στην πρώτη μορφή) είτε με το αντικείμενο–συνάρτηση `comp()` (στη δεύτερη). Το `comp()` πρέπει να ικανοποιεί τα κριτήρια στο §9.4.2.

12.13.8 `upper_bound()`

```
template<typename ForwardIterator, typename Type>
ForwardIterator
upper_bound(ForwardIterator beg, ForwardIterator end,
            Type const & value);

template<typename ForwardIterator, typename Type,
```

```
         typename BinaryFunctor>
ForwardIterator
upper_bound(ForwardIterator beg, ForwardIterator end,
            Type const & value, BinaryFunctor comp);
```

Ο αλγόριθμος και στις δύο μορφές του δέχεται ένα διάστημα, [beg,end), και μια τιμή, value. Το διάστημα θεωρείται ότι είναι ταξινομημένο ή έστω διαμοιρασμένο ως προς την value, είτε με τον τελεστή '<', στην πρώτη μορφή, είτε με το `comp()` στη δεύτερη. Το `comp()` πρέπει να ικανοποιεί τα κριτήρια στο §9.4.2.

Ο αλγόριθμος εντοπίζει το πρώτο στοιχείο στο διάστημα [beg,end) που είναι μεγαλύτερο από το value και επιστρέφει iterator στη θέση του. Αν δεν υπάρχει τέτοιο στοιχείο επιστρέφει end.

12.13.9 `lower_bound()`

```
template<typename ForwardIterator, typename Type>
ForwardIterator
lower_bound(ForwardIterator beg, ForwardIterator end,
            Type const & value);

template<typename ForwardIterator, typename Type,
         typename BinaryFunctor>
ForwardIterator
lower_bound(ForwardIterator beg, ForwardIterator end,
            Type const & value, BinaryFunctor comp);
```

Ο αλγόριθμος και στις δύο μορφές του δέχεται ένα διάστημα, [beg,end), και μια τιμή, value. Το διάστημα θεωρείται ότι είναι ταξινομημένο ή έστω διαμοιρασμένο ως προς την value, είτε με τον τελεστή '<', στην πρώτη μορφή, είτε με το `comp()` στη δεύτερη. Το `comp()` πρέπει να ικανοποιεί τα κριτήρια στο §9.4.2.

Ο αλγόριθμος εντοπίζει το πρώτο στοιχείο στο διάστημα [beg,end) που δεν είναι μικρότερο από το value και επιστρέφει iterator στη θέση του. Αν δεν υπάρχει τέτοιο στοιχείο επιστρέφει end.

12.13.10 `equal_range()`

```
template<typename ForwardIterator, typename Type>
std::pair<ForwardIterator,ForwardIterator>
equal_range(ForwardIterator beg, ForwardIterator end,
            Type const & value);

template<typename ForwardIterator, typename Type,
         typename BinaryFunctor>
std::pair<ForwardIterator,ForwardIterator>
```

```
equal_range(ForwardIterator beg, ForwardIterator end,
            Type const & value, BinaryFunctor comp);
```

Ο αλγόριθμος δέχεται ένα διάστημα, [beg,end), και μια τιμή, value. Το διάστημα θεωρείται ότι είναι ταξινομημένο ή έστω διαμοιρασμένο ως προς την value, είτε με τον τελεστή '<', στην πρώτη μορφή, είτε με το `comp()` στη δεύτερη. Το `comp()` πρέπει να ικανοποιεί τα κριτήρια στο §9.4.2.

Ο αλγόριθμος εντοπίζει και επιστρέφει σε ζεύγος, τους iterators αρχής και τέλους των θέσεων με τιμή value. Αν η τιμή δεν υπάρχει, οι δύο iterators που επιστρέφονται είναι ίσοι, είτε στην πλησιέστερη θέση με τιμή μεγαλύτερη από value είτε στο end, αν όλα τα στοιχεία είναι μικρότερα από value.

Στο συγκεκριμένο αλγόριθμο, «ίσα» είναι δύο στοιχεία που κανένα δεν είναι μικρότερο από το άλλο. Δεν χρησιμοποιείται ο τελεστής '==' για τη σύγκριση δύο στοιχείων a,b αλλά η ισότητά τους ελέγχεται με τον κώδικα

```
!(a<b) && !(b<a)
```

στην πρώτη μορφή και με το

```
!comp(a,b) && !comp(b,a)
```

στη δεύτερη.

12.14 Αλγόριθμοι για πράξεις συνόλων

Οι παρακάτω αλγόριθμοι παρέχονται για συγχώνευση, εύρεση ένωσης, τομής και διαφοράς ταξινομημένων συνόλων.

Έχουν δύο μορφές: στην πρώτη, η ταξινόμηση και η σύγκριση των στοιχείων θεωρείται ότι έγινε με τον τελεστή '<'. Στη δεύτερη, θεωρείται ότι έγινε με βάση το αντικείμενο–συνάρτηση `comp()`, το οποίο δέχεται δύο ορίσματα και επιστρέφει **`true`** ή **`false`** αν το πρώτο όρισμά του είναι «μικρότερο» (ό,τι κι αν σημαίνει αυτό) από το δεύτερο ή όχι. Το `comp()` πρέπει να ικανοποιεί τα κριτήρια στο §9.4.2.

Για τα παρακάτω, «ίσα» είναι δύο στοιχεία που κανένα δεν είναι μικρότερο από το άλλο. Δεν χρησιμοποιείται ο τελεστής '==' για τη σύγκριση δύο στοιχείων a,b· η ισότητα ελέγχεται με τον κώδικα `!(a<b) && !(b<a)` ή, στη δεύτερη μορφή, με το `!comp(a,b) && !comp(b,a)`.

12.14.1 `merge()`

```
template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator>
OutputIterator
merge(InputIterator1 beg1, InputIterator1 end1,
      InputIterator2 beg2, InputIterator2 end2, OutputIterator beg3);
```

```
template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator, typename BinaryFunctor>
OutputIterator
merge(InputIterator1 beg1, InputIterator1 end1,
      InputIterator2 beg2, InputIterator2 end2,
      OutputIterator beg3, BinaryFunctor comp);
```

Ο αλγόριθμος συγχωνεύει τα στοιχεία των *ήδη ταξινομημένων* διαστημάτων [beg1, end1) και [beg2,end2) στο διάστημα που ξεκινά με beg3, πάλι *ταξινομημένα*.

Επιστρέφει iterator στη θέση μετά το τελευταίο στοιχείο που γράφτηκε στην ακολουθία εξόδου.

Προσέξτε ότι οι `std::list<>` και `std::forward_list<>` παρέχουν γρηγορότερη συνάρτηση–μέλος.

12.14.2 `inplace_merge()`

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

template<typename BidirectionalIterator, typename BinaryFunctor>
void
inplace_merge(BidirectionalIterator beg, BidirectionalIterator mid,
              BidirectionalIterator end, BinaryFunctor comp);
```

Ο αλγόριθμος συγχωνεύει δύο ταξινομημένες, διαδοχικές ακολουθίες μεταξύ [beg, mid) και [mid,end) σε μία ταξινομημένη στο διάστημα [beg,end). Στην περίπτωση κοινών («ίσων») στοιχείων, διατηρείται η σειρά τους.

12.14.3 `set_union()`

```
template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator>
OutputIterator
set_union(InputIterator1 beg1, InputIterator1 end1,
          InputIterator2 beg2, InputIterator2 end2,
          OutputIterator beg3);

template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator, typename BinaryFunctor>
OutputIterator
set_union(InputIterator1 beg1, InputIterator1 end1,
          InputIterator2 beg2, InputIterator2 end2,
```

```
          OutputIterator beg3, BinaryFunctor comp);
```

Ο αλγόριθμος εκτελεί παρόμοια λειτουργία με την `std::merge()`. Αντιγράφει σε θέσεις από το beg3 και μετά, τα στοιχεία των ταξινομημένων συλλογών στα διαστήματα [beg1,end1) και [beg2,end2), αφού τα ταξινομήσει. Αν κάποιο στοιχείο εμφανίζεται m_1 φορές στο πρώτο διάστημα και m_2 στο δεύτερο, αντιγράφονται τα m_1 στοιχεία από το πρώτο διάστημα και μετά τα υπόλοιπα m_2-m_1 από το δεύτερο διάστημα (αν $m_2>m_1$).

Επιστρέφει iterator στην ακολουθία εξόδου, στην επόμενη θέση από την τελευταία που έχει γραφεί.

12.14.4 `set_intersection()`

```
template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator>
OutputIterator
set_intersection(InputIterator1 beg1, InputIterator1 end1,
                 InputIterator2 beg2, InputIterator2 end2,
                 OutputIterator beg3);

template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator, typename BinaryFunctor>
OutputIterator
set_intersection(InputIterator1 beg1, InputIterator1 end1,
                 InputIterator2 beg2, InputIterator2 end2,
                 OutputIterator beg3, BinaryFunctor comp);
```

Ο αλγόριθμος αντιγράφει σε θέσεις από το beg3 και μετά, τα κοινά στοιχεία των ταξινομημένων συλλογών στα διαστήματα [beg1,end1) και [beg2,end2), με τη σειρά που τα συναντά. Αν κάποιο στοιχείο εμφανίζεται m_1 φορές στο πρώτο διάστημα και m_2 στο δεύτερο, αντιγράφονται τα πρώτα $\min\{m_1,m_2\}$ από αυτά.

Επιστρέφει iterator στην ακολουθία εξόδου, στην επόμενη θέση από την τελευταία που έχει γραφεί.

12.14.5 `set_difference()`

```
template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator>
OutputIterator
set_difference(InputIterator1 beg1, InputIterator1 end1,
               InputIterator2 beg2, InputIterator2 end2,
               OutputIterator beg3);

template<typename InputIterator1, typename InputIterator2,
```

```
        typename OutputIterator, typename BinaryFunctor>
OutputIterator
set_difference(InputIterator1 beg1, InputIterator1 end1,
               InputIterator2 beg2, InputIterator2 end2,
               OutputIterator beg3, BinaryFunctor comp);
```

Ο αλγόριθμος και στις δύο μορφές, δέχεται δύο ταξινομημένες συλλογές στα διαστήματα [beg1,end1) και [beg2,end2). Αντιγράφει ταξινομημένα, σε διαδοχικές θέσεις από το beg3 και μετά, τα στοιχεία της πρώτης ακολουθίας που δεν περιέχονται στη δεύτερη. Αν κάποιο στοιχείο εμφανίζεται m_1 φορές στο πρώτο διάστημα και m_2 στο δεύτερο, αντιγράφονται τα τελευταία $\max\{m_1-m_2,0\}$ από το πρώτο διάστημα.

Ο αλγόριθμος επιστρέφει iterator στην ακολουθία εξόδου, στην επόμενη θέση από την τελευταία που έχει γραφεί.

12.14.6 `set_symmetric_difference ()`

```
template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator>
OutputIterator
set_symmetric_difference(InputIterator1 beg1, InputIterator1 end1,
                         InputIterator2 beg2, InputIterator2 end2,
                         OutputIterator beg3);

template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator, typename BinaryFunctor>
OutputIterator
set_symmetric_difference(InputIterator1 beg1, InputIterator1 end1,
                         InputIterator2 beg2, InputIterator2 end2,
                         OutputIterator beg3, BinaryFunctor comp);
```

Ο αλγόριθμος και στις δύο μορφές, δέχεται δύο ταξινομημένες συλλογές στα διαστήματα [beg1,end1) και [beg2,end2). Αντιγράφει ταξινομημένα, σε διαδοχικές θέσεις από το beg3 και μετά, τα στοιχεία που περιέχονται στην πρώτη ή στη δεύτερη ακολουθία αλλά όχι και στις δύο. Αν κάποιο στοιχείο εμφανίζεται m_1 φορές στο πρώτο διάστημα και m_2 στο δεύτερο, αντιγράφονται τα τελευταία m_1-m_2 από το πρώτο διάστημα (αν $m_1>m_2$) ή τα τελευταία m_2-m_1 από το δεύτερο διάστημα (αν $m_2>m_1$) ή κανένα (αν $m_1=m_2$).

12.14.7 `includes ()`

```
template<typename InputIterator1, typename InputIterator2>
bool
includes(InputIterator1 beg1, InputIterator1 end1,
         InputIterator2 beg2, InputIterator2 end2);
```

```
template<typename InputIterator1, typename InputIterator2,
         typename BinaryFunctor>
bool
includes(InputIterator1 beg1, InputIterator1 end1,
         InputIterator2 beg2, InputIterator2 end2,
         BinaryFunctor comp);
```

Ο αλγόριθμος ελέγχει αν η ακολουθία στο διάστημα [beg1,end1), η οποία θεωρείται ήδη ταξινομημένη, περιέχει όλα τα στοιχεία της επίσης ταξινομημένης ακολουθίας στο διάστημα [beg2,end2).

12.15 Αλγόριθμοι χειρισμού heap

Η δομή δεδομένων heap είναι ένας τρόπος οργάνωσης μιας συλλογής στοιχείων που επιτρέπει τη γρήγορη ανάκληση του στοιχείου με τη «μεγαλύτερη» τιμή (για max heap) ή «μικρότερη» τιμή (για min heap) και τη γρήγορη εισαγωγή νέων στοιχείων.

Οι παρακάτω αλγόριθμοι παρέχονται για το χειρισμό max heap. Έχουν δύο μορφές: η σύγκριση των στοιχείων στην πρώτη μορφή γίνεται με τον τελεστή '<'. Στη δεύτερη γίνεται με βάση το αντικείμενο–συνάρτηση `comp()`, το οποίο δέχεται δύο ορίσματα και επιστρέφει **`true`** ή **`false`** αν το πρώτο όρισμά του είναι «μικρότερο» (ό,τι κι αν σημαίνει αυτό) από το δεύτερο ή όχι. Το `comp()` πρέπει να ικανοποιεί τα κριτήρια στο §9.4.2.

12.15.1 `make_heap()`

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

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

Ο αλγόριθμος δέχεται δύο iterators που ορίζουν ένα διάστημα [beg,end). Αναδιατάσσει τα στοιχεία ώστε να οργανωθούν σε max heap. Το στοιχείο με τη «μεγαλύτερη» τιμή τοποθετείται στη θέση beg και τα υπόλοιπα ακολουθούν.

12.15.2 `is_heap()`

```
template<typename RandomIterator>
bool
is_heap(RandomIterator beg, RandomIterator end);

template<typename RandomIterator, typename BinaryFunctor>
bool
is_heap(RandomIterator beg, RandomIterator end, BinaryFunctor comp);
```

Ο αλγόριθμος ελέγχει αν τα στοιχεία στο διάστημα [beg,end) είναι οργανωμένα σε max heap. Επιστρέφει λογική τιμή: `true` αν είναι, `false` αν δεν είναι. Αν το διάστημα περιέχει λιγότερα από δύο στοιχεία, ο αλγόριθμος επιστρέφει `true`.

12.15.3 `is_heap_until()`

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

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

Ο αλγόριθμος ελέγχει αν τα στοιχεία στο διάστημα [beg,end) είναι οργανωμένα σε heap. Αν βρει στοιχείο που δεν είναι στη σωστή θέση ως προς τα άλλα, επιστρέφει iterator σε αυτό. Αν όλα τα στοιχεία είναι σε heap ή αν το πλήθος των στοιχείων στο [beg,end) είναι λιγότερο από δύο, επιστρέφει end. Σε κάθε περίπτωση, το διάστημα από beg έως τον επιστρεφόμενο iterator είναι οργανωμένο σε heap.

12.15.4 `pop_heap()`

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

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

Ο αλγόριθμος δέχεται δύο iterators που ορίζουν ένα διάστημα [beg,end) σε max heap. Αναδιατάσσει τα στοιχεία ώστε αυτό με τη μεγαλύτερη τιμή, δηλαδή το στοιχείο στη θέση beg, να μεταφερθεί στη θέση (end-1) και αφήνει το διάστημα [beg,end-1) ως νέο heap.

12.15.5 `push_heap()`

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

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

Ο αλγόριθμος δέχεται δύο iterators που ορίζουν ένα διάστημα [beg, end). Το στοιχεία στο διάστημα [beg,end-1) είναι οργανωμένα σε max heap. Ο αλγόριθμος αναδιατάσσει τα στοιχεία ώστε το στοιχείο στη θέση (end-1) να τοποθετηθεί στη σωστή του θέση ως προς τα υπόλοιπα ώστε το διάστημα [beg,end) να είναι heap.

12.15.6 `sort_heap()`

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

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

Ο αλγόριθμος δέχεται δύο iterators που ορίζουν ένα διάστημα [beg, end). Το στοιχεία σε αυτό είναι οργανωμένα σε max heap. Τα αναδιατάσσει ώστε να ταξινομηθούν από το μικρότερο στο μεγαλύτερο.

12.16 Μη τροποποιητικοί αλγόριθμοι

12.16.1 `all_of()`

```
template<typename InputIterator, typename UnaryFunctor>
bool
all_of(InputIterator beg, InputIterator end, UnaryFunctor op);
```

Στο συγκεκριμένο αλγόριθμο, το αντικείμενο–συνάρτηση `op()` δέχεται ως μοναδικό όρισμα κάθε στοιχείο του διαστήματος [beg,end). Αν για όλα τα στοιχεία το `op()` επιστρέψει **`true`** ή το διάστημα είναι κενό, ο αλγόριθμος επιστρέφει **`true`**, αλλιώς επιστρέφει **`false`**.

12.16.2 `none_of()`

```
template<typename InputIterator, typename UnaryFunctor>
bool
none_of(InputIterator beg, InputIterator end, UnaryFunctor op);
```

Στο συγκεκριμένο αλγόριθμο, το αντικείμενο–συνάρτηση `op()` δέχεται ως μοναδικό όρισμα κάθε στοιχείο του διαστήματος [beg,end). Αν για όλα τα στοιχεία το `op()` επιστρέψει **`false`** ή το διάστημα είναι κενό, ο αλγόριθμος επιστρέφει **`true`**, αλλιώς επιστρέφει **`false`**.

12.16.3 `any_of()`

```
template<typename InputIterator, typename UnaryFunctor>
bool
any_of(InputIterator beg, InputIterator end, UnaryFunctor op);
```

Στο συγκεκριμένο αλγόριθμο το αντικείμενο–συνάρτηση `op()` δέχεται ως μοναδικό όρισμα όσα στοιχεία του διαστήματος [beg,end) χρειαστεί για να προσδιορίσει την επιστρεφόμενη τιμή. Αν υπάρχει τουλάχιστον ένα στοιχείο για το οποίο το `op()` επιστρέψει **`true`**, ο αλγόριθμος επιστρέφει **`true`**. Σε άλλη περίπτωση ή αν το διάστημα είναι κενό, επιστρέφει **`false`**.

12.16.4 `count()`

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

template<typename InputIterator, typename UnaryFunctor>
Diff
count_if(InputIterator beg, InputIterator end, UnaryFunctor op);
```

Η πρώτη μορφή του αλγόριθμου επιστρέφει τον αριθμό των στοιχείων στο διάστημα [beg,end) που είναι ίσα με value. Θυμηθείτε ότι οι associative containers παρέχουν ανάλογη συνάρτηση–μέλος.

Η δεύτερη επιστρέφει το πλήθος των στοιχείων για τα οποία η δράση του αντικειμένου–συνάρτηση `op()`, ενός ορίσματος, επιστρέφει **`true`**. Είναι κατάλληλη για πιο σύνθετες «μετρήσεις».

Ο τύπος Diff είναι ακέραιος[3].

[3] `typename std::iterator_traits<InputIterator>::difference_type`

Παράδειγμα

Ο κώδικας

```
auto cnt = std::count(v.cbegin(), v.cend(), a);
auto && lta = std::bind(std::less<decltype(a)>{},
                        std::placeholders::_1, a);
auto cnt2 = std::count_if(v.cbegin(), v.cend(), lta);
```

μετρά στη μεταβλητή cnt το πλήθος των στοιχείων ενός container v που είναι ίσα με μια τιμή a και στη μεταβλητή cnt2 το πλήθος των στοιχείων του v που είναι μικρότερα από αυτή την τιμή.

Παρατηρήστε ότι χρησιμοποιήθηκε ένας *προσαρμογέας* (§9.3.2) για να τροποποιήσει ένα αντικείμενο–συνάρτηση με δύο ορίσματα ώστε να δέχεται ένα όρισμα. Εναλλακτικά, θα μπορούσαμε να χρησιμοποιήσουμε ως τρίτο όρισμα του `std::count_if()` τη συνάρτηση λάμδα

```
auto && lta = [a] (decltype(a) x) -> bool {return x < a;};
```

12.16.5 `equal()`

```
template<typename InputIterator1, typename InputIterator2>
bool
equal(InputIterator1 beg1, InputIterator1 end1, InputIterator2 beg2);

template<typename InputIterator1, typename InputIterator2,
         typename BinaryFunctor>
bool
equal(InputIterator1 beg1, InputIterator1 end1, InputIterator2 beg2,
      BinaryFunctor comp);

template<typename InputIterator1, typename InputIterator2>
bool
equal(InputIterator1 beg1, InputIterator1 end1,
      InputIterator2 beg2, InputIterator2 end2);

template<typename InputIterator1, typename InputIterator2,
         typename BinaryFunctor>
bool
equal(InputIterator1 beg1, InputIterator1 end1, InputIterator2 beg2,
      InputIterator2 end2, BinaryFunctor comp);
```

Ο αλγόριθμος στην πρώτη και δεύτερη μορφή συγκρίνει όλα τα στοιχεία στο διάστημα [beg1,end1) με τα αντίστοιχα στο ίσου μήκους διάστημα που ξεκινά με το

beg2. Επιστρέφει `true` αν όλα τα ζεύγη έχουν «ίσα» στοιχεία. Αν κάποια στοιχεία διαφέρουν, επιστρέφει `false`.

Στην τρίτη και τέταρτη μορφή ο αλγόριθμος επιστρέφει `true` αν τα διαστήματα [beg1,end1) και [beg2,end2) έχουν (i) ίσο πλήθος στοιχείων και (ii) τα αντίστοιχα στοιχεία είναι «ίσα». Σε αντίθετη περίπτωση επιστρέφει `false`.

Στην πρώτη και στην τρίτη μορφή, η σύγκριση των στοιχείων γίνεται με τον τελεστή '=='. Στη δεύτερη και στην τέταρτη, η σύγκριση γίνεται με το αντικείμενο–συνάρτηση `comp()`. Αυτό δέχεται δύο ορίσματα και επιστρέφει λογική τιμή, χαρακτηρίζοντάς τα έτσι ως «ίσα» ή όχι.

Παράδειγμα

Ο κώδικας

```
auto && f = std::less<decltype(a)::value_type>{};
bool g = std::equal(a.cbegin(), a.cend(), b.cbegin(), f);
```

ελέγχει αν τα στοιχεία του container a είναι όλα μικρότερα από τα αντίστοιχα στοιχεία του container b.

12.16.6 `mismatch()`

```
template<typename InputIterator1, typename InputIterator2>
std::pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 beg1, InputIterator1 end1,
         InputIterator2 beg2);

template<typename InputIterator1, typename InputIterator2,
         typename BinaryFunctor>
std::pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 beg1, InputIterator1 end1,
         InputIterator2 beg2, BinaryFunctor comp);

template<typename InputIterator1, typename InputIterator2>
std::pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 beg1, InputIterator1 end1,
         InputIterator2 beg2, InputIterator2 end2);

template<typename InputIterator1, typename InputIterator2,
         typename BinaryFunctor>
std::pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 beg1, InputIterator1 end1,
         InputIterator2 beg2, InputIterator2 end2,
         BinaryFunctor comp);
```

Ο αλγόριθμος στην πρώτη και δεύτερη μορφή συγκρίνει όλα τα στοιχεία στο διάστημα [beg1,end1) με τα αντίστοιχα στο ίσου μήκους διάστημα που ξεκινά με το beg2 και εντοπίζει το πρώτο ζεύγος άνισων στοιχείων. Στην τρίτη και τέταρτη μορφή συγκρίνει τα αντίστοιχα στοιχεία στα διαστήματα [beg1,end1) και [beg2,end2) και εντοπίζει το πρώτο ζεύγος άνισων στοιχείων.

Στην πρώτη και στην τρίτη μορφή, η σύγκριση των στοιχείων γίνεται με τον τελεστή '=='. Στη δεύτερη και στην τέταρτη, η σύγκριση γίνεται με το αντικείμενο–συνάρτηση `comp()`. Αυτό δέχεται δύο ορίσματα που δεν επιτρέπεται να τα τροποποιεί και επιστρέφει **`true`** ή **`false`** αν το πρώτο όρισμά του είναι «ίσο» (ό,τι κι αν σημαίνει αυτό) με το δεύτερο ή όχι.

Ο αλγόριθμος επιστρέφει ζεύγος (`std::pair<>`) με δύο iterators, ένα σε κάθε μια ακολουθία εισόδου. Κάθε iterator «δείχνει» στο στοιχείο που διαφέρει από το αντίστοιχό του. Εάν όλα τα αντίστοιχα στοιχεία είναι ίσα, το ζεύγος από iterators που επιστρέφεται στην πρώτη και δεύτερη μορφή έχει πρώτο μέλος το end1 και δεύτερο μέλος τον iterator σε end1−beg1 θέσεις μετά το beg2 (που υποθέτουμε ότι υπάρχει). Στην τρίτη και τέταρτη μορφή, αν το πρώτο διάστημα έχει λιγότερα ή ίσαριθμα στοιχεία από το δεύτερο, το ζεύγος από iterators που επιστρέφεται έχει μέλη το end1 και τον αντίστοιχο iterator από το δεύτερο διάστημα (αυτόν που βρίσκεται end1−beg1 θέσεις μετά το beg2). Αν το δεύτερο διάστημα είναι μικρότερο, επιστρέφεται το ζεύγος με πρώτο μέλος τον iterator που βρίσκεται end2−beg2 θέσεις μετά το beg1 και ως δεύτερο το end2.

12.17 Τροποποιητικοί αλγόριθμοι

Οι παρακάτω αλγόριθμοι τροποποιούν τα στοιχεία της ακολουθίας εισόδου. Επομένως, δεν μπορούν να χρησιμοποιηθούν σε associative ή unordered containers.

12.17.1 `iota()`

```
template<typename ForwardIterator, typename Type>
void
iota(ForwardIterator beg, ForwardIterator end, Type value);
```

Η συνάρτηση αποδίδει στα στοιχεία του διαστήματος [beg,end) διαδοχικά τις τιμές value, `value+1`, `value+2`, ... Η διαδοχικές τιμές από value και μετά, παράγονται με διαδοχικές εφαρμογές του τελεστή '++'.

Παρέχεται από το `<numeric>`.

Παράδειγμα

Ο κώδικας

```
std::iota(a.begin(), a.end(), 0);
```

εκχωρεί σε διαδοχικά στοιχεία ενός container ακεραίων με όνομα a, τις τιμές 0, 1, 2, ..., ξεκινώντας από την αρχική θέση.

12.17.2 `for_each()`

```
template<typename InputIterator, typename UnaryFunctor>
UnaryFunctor
for_each(InputIterator beg, InputIterator end, UnaryFunctor op);
```

Καλεί το αντικείμενο–συνάρτηση `op()` με όρισμα αναφορά σε κάθε στοιχείο του διαστήματος [beg,end). Το `op()` μπορεί να τροποποιήσει το όρισμα. Τυχόν επιστρεφόμενη τιμή του αγνοείται.

Ο αλγόριθμος επιστρέφει το αντικείμενο–συνάρτηση που έχει δημιουργηθεί με την κλήση του `op()`, σε κατάλληλη μορφή για *μετακίνηση*, δηλαδή, σαν να έχει εκτελεστεί ως τελευταία εντολή του αλγόριθμου το `return std::move(op);`. Το αντικείμενο αυτό μπορεί να έχει τροποποιηθεί, να έχει αλλάξει η εσωτερική του κατάσταση.

12.17.3 `swap_ranges()`

```
template<typename ForwardIterator1, typename ForwardIterator2>
ForwardIterator2
swap_ranges(ForwardIterator1 beg1, ForwardIterator1 end1,
            ForwardIterator2 beg2);
```

Ο αλγόριθμος εναλλάσσει κάθε στοιχείο στο διάστημα [beg1, end1) με το αντίστοιχό του στο διάστημα που ξεκινά από το beg2 (δηλαδή το [beg2, `beg2+(end1-beg1)`)). Επιστρέφει iterator στην ακολουθία εξόδου, στο τελευταίο στοιχείο που άλλαξε.

12.17.4 `transform()`

```
template<typename InputIterator, typename OutputIterator,
         typename UnaryFunctor>
OutputIterator
transform(InputIterator beg1, InputIterator end1,
          OutputIterator beg2, UnaryFunctor op);

template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator, typename UnaryFunctor>
OutputIterator
transform(InputIterator1 beg1, InputIterator1 end1,
          InputIterator1 beg2, OutputIterator beg3,
          BinaryFunctor op);
```

Η πρώτη μορφή καλεί το αντικείμενο–συνάρτηση ενός ορίσματος `op()` για κάθε στοιχείο στο διάστημα [beg1,end1) και γράφει το αποτέλεσμά του στο διάστημα που ξεκινά με το beg2. Τα beg1, beg2 μπορεί να είναι ίδια. Δεν επιτρέπεται να περιλαμβάνεται το beg2 στο υπόλοιπο διάστημα, (beg1,end1).

Η δεύτερη μορφή καλεί το αντικείμενο–συνάρτηση δύο ορισμάτων `op()` για τα αντίστοιχα στοιχεία του διαστήματος [beg1,end1) και του διαστήματος που ξεκινά με το beg2 και γράφει το αποτέλεσμά του στο διάστημα που ξεκινά με το beg3. Τα beg1, beg2, beg3 μπορεί να είναι ίδια.

Και στις δύο μορφές επιστρέφεται iterator στο επόμενο στοιχείο από το τελευταίο στο οποίο έγινε εγγραφή.

12.17.5 `fill()`

```
template<typename ForwardIterator, typename Type>
void
fill(ForwardIterator beg, ForwardIterator end, Type const & value);

template<typename OutputIterator, typename Size, typename Type>
OutputIterator
fill_n(OutputIterator beg, Size n, Type const & value);
```

Η πρώτη μορφή εκχωρεί την τιμή value στα στοιχεία του διαστήματος [beg,end), με μετατροπή τύπου αν χρειάζεται.

Η δεύτερη εκχωρεί την τιμή value σε n διαδοχικά στοιχεία, ξεκινώντας από το beg. Αν το n δεν είναι θετικό, δεν κάνει τίποτε. Επιστρέφει iterator στην επόμενη θέση από την τελευταία τροποποιημένη (δηλαδή το `beg+n`). Αν το n είναι αρνητικό ή μηδέν, επιστρέφει τον iterator beg.

Παράδειγμα

Ο κώδικας

```
auto it = std::fill_n(a.begin(), 10, 1);
std::fill(it, a.end(), 2);
```

εκχωρεί στα 10 πρώτα στοιχεία ενός container ακεραίων με όνομα a την τιμή 1 και στα υπόλοιπα την τιμή 2.

12.17.6 `generate()`

```
template<typename ForwardIterator, typename Functor>
void
generate(ForwardIterator beg, ForwardIterator end, Functor op);

template<typename OutputIterator, typename Size, typename Functor>
```

```
OutputIterator
generate_n(OutputIterator beg, Size n, Functor op);
```

Η πρώτη μορφή του αλγόριθμου εκχωρεί στα στοιχεία του διαστήματος [beg,end) την τιμή που επιστρέφει το αντικείμενο–συνάρτηση `op()`, το οποίο δε δέχεται ορίσματα. Το `op()` μπορεί να τροποποιεί την εσωτερική του κατάσταση.

Η δεύτερη μορφή αναθέτει την επιστρεφόμενη τιμή του `op()` σε n διαδοχικά στοιχεία, ξεκινώντας από το beg. Αν το n είναι αρνητικό ή μηδέν, δεν κάνει τίποτε. Επιστρέφει iterator στην επόμενη θέση από την τελευταία τροποποιημένη (δηλαδή το `beg+n`). Αν το n δεν είναι θετικό, επιστρέφει τον iterator beg.

12.18 Ασκήσεις

1. Γράψτε πρόγραμμα που να δημιουργεί ένα αρχείο με 1000000 τυχαίους ακέραιους αριθμούς στο διάστημα $[-1000000, 1000000]$. Σε άλλο πρόγραμμα, διαβάστε το αρχείο σε container της επιλογής σας. Κατόπιν, με κατάλληλους αλγόριθμους,

   (α΄) να προσδιορίσετε το μικρότερο και το μεγαλύτερο στοιχείο.

   (β΄) Αναζητήστε το στοιχείο -1234. Σε ποια θέση είναι; Αν δεν υπάρχει τυπώστε κατάλληλο μήνυμα.

   (γ΄) Υπολογίστε το άθροισμα και το μέσο όρο όλων των στοιχείων.

   (δ΄) Τυπώστε το πλήθος των στοιχείων που είναι ίσα με 0.

   (ε΄) Τυπώστε το πλήθος των στοιχείων που είναι θετικά.

   (στ΄) Διαγράψτε όλα τα στοιχεία που είναι ίσα με 0.

   (ζ΄) Αντικαταστήστε όλα τα στοιχεία που είναι μεγαλύτερα από το 10000 με το 10000.

   (η΄) Βρείτε τα 100 μικρότερα στοιχεία.

   Μπορείτε να μετατρέψετε το πρόγραμμά σας σε συνάρτηση template με παράμετρο τον τύπο του container; Μπορείτε να τη χρησιμοποιήσετε για τους containers `vector`, `deque`, `list`, `set`; Χρονομετρήστε τη συνάρτηση με διαφορετικούς containers (χρησιμοποιήστε την `std::clock()`[4] από το `<ctime>`). Τι συμπεραίνετε;

2. Γράψτε συναρτήσεις που να υλοποιούν τους αλγόριθμους της Standard Library. Σε όσες μπορείτε, κρατήστε το ίδιο interface, να δέχονται δηλαδή iterators. Χρονομετρήστε αυτές και τους αντίστοιχους αλγορίθμους. Τι συμπεραίνετε;

3. (α΄) Υλοποιήστε τον αλγόριθμο `std::equal()` σε δική σας συνάρτηση με όνομα equal. Συμπληρώστε, δηλαδή, τον κώδικα

   ```
   template<typename Iterator>
   bool
   equal(Iterator beg1, Iterator end1, Iterator beg2)
   {
   .....
   }
   ```

   (β΄) Χρησιμοποιώντας το `std::list<>`, δημιουργήστε δύο λίστες ακέραιων αριθμών και αποθηκεύστε τις ακολουθίες $1, 3, 5, 88, 92, 4, 91$ στην πρώτη και $1, 3, 5, 88, 92, 4, 2, 91$ στη δεύτερη. Καλέστε τη δική σας συνάρτηση

[4] Σε συστήματα Unix η εντολή man 3 clock θα σας βοηθήσει.

`equal()` για να ελέγξετε αν είναι ίσες οι δύο ακολουθίες (προφανώς πρέπει να σας επιστρέψει `false`).

4. Τα αρχεία που βρίσκονται στις διευθύνσεις http://bit.ly/2bDuQxB και http://bit.ly/2bIIhtv περιέχουν άγνωστο πλήθος ακέραιων αριθμών, ένα σε κάθε γραμμή τους. Να γράψετε πρόγραμμα που να «διαβάζει» τους αριθμούς του πρώτου αρχείου σε ένα container a και τους αριθμούς του δεύτερου σε container b, και να γράφει στο αρχείο *fileC.txt* τους αριθμούς του a που δεν περιέχονται στο b, σε ξεχωριστή γραμμή τον καθένα.

   Να το κάνετε *χωρίς τη χρήση* των εντολών βρόχου (`for`, `while`, `do ... while`, `goto`). Χρησιμοποιήστε κατάλληλο αλγόριθμο.

Μέρος ΙΙΙ

Αντικειμενοστρεφής Προγραμματισμός

Κεφάλαιο 13

Γενικές έννοιες αντικειμενοστρεφούς προγραμματισμού

Στο κεφάλαιο αυτό θα παρουσιαστούν οι βασικές έννοιες του «προγραμματισμού βασιζόμενου σε αντικείμενα» (object based) και του «προγραμματισμού προσανατολισμένου σε αντικείμενα» (αντικειμενοστρεφούς προγραμματισμού, object oriented programming). Με τη βοήθεια ενός παραδείγματος, θα περιγράψουμε στο τρέχον και στο επόμενο κεφάλαιο τη βασική σχεδίαση και ανάπτυξη ενός προγράμματος που προσομοιώνει όσο πιο πιστά γίνεται το εκάστοτε πρόβλημα, ακολουθώντας τον αντικειμενοστρεφή προγραμματισμό.

13.1 Εισαγωγή

Ας υποθέσουμε ότι επιθυμούμε να περιγράψουμε σε πρόγραμμα για υπολογιστή, τη δανειστική λειτουργία μιας πανεπιστημιακής Βιβλιοθήκης· θέλουμε να καταγράψουμε ηλεκτρονικά τα βιβλία που υπάρχουν, να γνωρίζουμε την κατάστασή τους (δανεισμένα ή όχι και από ποιον), να γνωρίζουμε σε ποια υπάρχει καθυστέρηση στην επιστροφή. Επίσης, πρέπει να γνωρίζουμε όσους έχουν δικαίωμα δανεισμού (φοιτητές, καθηγητές, κλπ.). Το πρόβλημά μας επομένως περιλαμβάνει

- ένα σύνολο δικαιούχων δανεισμού (που χαρακτηρίζονται από το όνομά τους, το Τμήμα στο οποίο ανήκουν, και αν είναι φοιτητές, τον αριθμό μητρώου στη Σχολή τους, το έτος εισαγωγής τους, κλπ.),

- ένα πλήθος βιβλίων (που χαρακτηρίζονται από τον τίτλο τους, το συγγραφέα, τον εκδοτικό οίκο, το έτος έκδοσης, κλπ.), που μπορούν να δοθούν προς δανεισμό.

Η απλοϊκή υλοποίηση του προγράμματός μας θα περιλαμβάνει δηλώσεις ξεχωριστών διανυσμάτων από σειρές χαρακτήρων και ακέραιους για την αναπαράσταση καθενός από τα χαρακτηριστικά που αναφέρθηκαν. Παρατηρήστε ότι στη συγκεκριμένη σχεδίαση, τίποτε δεν εκφράζει τη σχέση που έχουν (αν έχουν) τα στοιχεία διαφορετικών διανυσμάτων μεταξύ τους· τίποτε, εκτός πιθανόν από το όνομά τους, δεν υποδηλώνει ότι κάποια περιγράφουν χαρακτηριστικά της έννοιας «φοιτητής» ενώ κάποια άλλα της έννοιας «Βιβλίο». Προσέξτε ότι ένα σωστά επιλεγμένο όνομα μεταβλητής διευκολύνει αρκετά τον προγραμματιστή ή τον αναγνώστη στην κατανόηση του κώδικα, αλλά για το μεταγλωττιστή δεν έχει κάποιο ιδιαίτερο νόημα. Επιπλέον, όχι μόνο δε γίνεται η ομαδοποίηση των χαρακτηριστικών στο επίπεδο των βασικών εννοιών του προβλήματός μας (όνομα φοιτητή, αριθμός μητρώου, κλπ. μαζί και, ξεχωριστά, αλλά πάλι συγκεντρωμένα, ο τίτλος του βιβλίου, το όνομα του συγγραφέα, κλπ.) αλλά, χειρότερα, συνδέονται σε διανύσματα όλα τα ονόματα φοιτητών μαζί, όλοι οι αριθμοί μητρώου, κλπ. Παρόμοιο πρόβλημα συναντούμε σε συναρτήσεις που θα έχει ο κώδικάς μας. Έτσι, θα έχουμε εντολές που θέλουμε να εκτελούνται για τα χαρακτηριστικά της έννοιας «Βιβλίο», τίποτε όμως δεν θα εξασφαλίζει ότι μια συνάρτηση που δέχεται ως ορίσματα ένα αριθμό από σειρές χαρακτήρων και ακέραιους σχετίζεται με το «Βιβλίο» και επιτρέπεται να δρα μόνο σε αυτό. Η συγκεκριμένη οργάνωση των δεδομένων μας, συνεπώς και του υπόλοιπου κώδικά μας, σε ανεξάρτητες ποσότητες, είναι αρκετά διαφορετική από αυτή που υπαγορεύει το πρόβλημα που προσομοιώνουμε. Η μεθοδολογία αυτή δεν είναι λάθος, δυσκολεύει όμως την ανάπτυξη και διόρθωση μεγάλων και πολύπλοκων προγραμμάτων.

Μια καλύτερη, πιο «φυσική» προσέγγιση είναι να χρησιμοποιήσουμε τύπους ποσοτήτων που θα μπορούν να περιγράψουν *συνολικά* τα χαρακτηριστικά κάθε έννοιας. Οι ενσωματωμένοι τύποι μιας γενικής γλώσσας προγραμματισμού δεν έχουν αυτή τη δυνατότητα: η έννοια «Φοιτητής» δεν είναι (μόνο) ένας ακέραιος ή μια σειρά χαρακτήρων.

Είναι προφανές ότι μια γενική γλώσσα προγραμματισμού δεν μπορεί να παρέχει έτοιμους τύπους για οποιαδήποτε έννοια χρειαστεί κάποιο πρόγραμμα. Οι σύγχρονες γλώσσες παρέχουν στον προγραμματιστή τη δυνατότητα να δημιουργήσει νέους τύπους. Θα πρέπει να προσδιορίσει

- την αναπαράστασή τους στη μνήμη, δηλαδή τον τύπο και το όνομα των ποσοτήτων που αποτελούν το νέο τύπο. Οι ποόστητες αυτές χαρακτηρίζονται ως «μέλη» της νέας δομής.

- τους τρόπους δημιουργίας (τι συμβαίνει, δηλαδή, κατά τη δήλωση μιας ποσότητας του νέου τύπου), καταστροφής (τι συμβαίνει όταν η ροή του προγράμματος ξεπεράσει την εμβέλεια μιας ποσότητας του τύπου), αντιγραφής (πώς

δημιουργείται μια ποσότητα από μια άλλη ή πώς περνά ως όρισμα συνάρτησης) κλπ.

- τις πράξεις που έχουν νόημα για ποσότητες των νέων τύπων,
- την αλληλεπίδρασή τους με άλλες ποσότητες.

Αφού υλοποιηθεί ο νέος τύπος, μπορεί να χρησιμοποιηθεί σε περισσότερα του ενός προγράμματα και να τα απλοποιήσει σε μεγάλο βαθμό. Σκεφτείτε ότι και ο ενσωματωμένος ακέραιος τύπος που παρέχει κάθε γλώσσα προγραμματισμού στην πραγματικότητα είναι ένα σύνολο bits. Ο μεταγλωττιστής γνωρίζει πώς το δημιουργεί και του δίνει τιμές, ξέρει πώς να το προσθέτει ή να το πολλαπλασιάζει με άλλο. Ο προγραμματιστής μπορεί να χρησιμοποιήσει τον ακέραιο τύπο χωρίς να απασχολείται με τέτοιες λεπτομέρειες. Αυτή τη διευκόλυνση επιδιώκουμε να έχουμε, ορίζοντας νέους τύπους, τις *κλάσεις*, που να ανταποκρίνονται στις έννοιες του προβλήματός μας. Μια ποσότητα που δημιουργείται ώστε να έχει ένα τέτοιο νέο τύπο λέγεται «αντικείμενο» αυτής της κλάσης.

13.2 Ενθυλάκωση (encapsulation)

Ο τύπος που αναπαριστά την έννοια «Βιβλίο» στο πρόγραμμά μας θα πρέπει να περιλαμβάνει κάποια ποσότητα για την αποθήκευση του ονόματος του συγγραφέα (ή περισσότερες αν το βιβλίο έχει πολλούς συγγραφείς). Αυτή προφανώς θα είναι μια ή περισσότερες σειρές χαρακτήρων ανάλογα αν επιλέξουμε να διατηρήσουμε ενιαίο το όνομα και το επώνυμο ή χωριστά (π.χ. για λόγους εύκολης ταξινόμησης). Η πρόσβαση σε αυτές για ανάγνωση ή τροποποίηση θα είναι αναγκαία σε διάφορα τμήματα του προγράμματός μας (εισαγωγή στοιχείων ενός βιβλίου, εμφάνιση στην οθόνη, ομαδοποίηση έργου ενός συγγραφέα, κλπ.). Δεν είναι λάθος αυτή να γίνεται απευθείας στις σχετικές μεταβλητές. Σκεφτείτε όμως να υπάρξει ανάγκη μεταβολής τους, απλής (π.χ. των ονομάτων τους) ή ουσιαστικής (π.χ. των τύπων τους ώστε να υποστηρίζουν μη λατινικούς χαρακτήρες). Θα πρέπει να αναζητήσουμε σε όλο τον κώδικά μας τα σημεία στα οποία γίνεται αναφορά σε αυτές τις ποσότητες ώστε να εξετάσουμε αν χρειάζονται τροποποίηση. Είναι πολύ πιο εύκολο κατά τη δημιουργία του νέου τύπου να «εμποδίσουμε» την πρόσβαση στις συγκεκριμένες μεταβλητές από οποιοδήποτε κώδικά μας εκτός από μία ή περισσότερες «προνομιούχες» συναρτήσεις. Οπουδήποτε αλλού χρειαζόμαστε πρόσβαση στην τιμή των ποσοτήτων αυτή θα γίνεται μέσω των συγκεκριμένων συναρτήσεων. Με αυτό τον τρόπο θα έχουμε εντοπισμένες τις αναφορές στις επίμαχες μεταβλητές ώστε το όνομα, ο τύπος ή ακόμα και η ίδια η ύπαρξή τους να μπορεί να τροποποιηθεί σε ελάχιστα σημεία και αυτή η μεταβολή να διαδίδεται αυτόματα σε όλο τον κώδικα.

Η δυνατότητα να διαχωρίζουμε τον τρόπο *υλοποίησης* (την εσωτερική αναπαράσταση) από τον τρόπο *χρήσης* μιας σύνθετης έννοιας ονομάζεται *ενθυλάκωση (encapsulation)*. Αποτελεί ένα βασικό στοιχείο του *προγραμματισμού προσανατολισμένου σε αντικείμενα*. Είναι ιδιαίτερα σημαντικό στην ανάπτυξη μεγάλων κωδίκων

το να μπορούμε να περιορίζουμε την πρόσβαση στα μέλη μιας δομής μόνο σε συγκεκριμένες συναρτήσεις. Αυτές οι συναρτήσεις θα είναι οι μόνες που επιτρέπεται να καλούμε όταν θέλουμε να τροποποιήσουμε ένα αντικείμενο ή να δούμε την εσωτερική του κατάσταση. Διευκολύνεται η αναγνώριση και διόρθωση των σφαλμάτων καθώς και η δυνατότητα τροποποίησης της εσωτερικής αναπαράστασης.

13.3 Κληρονομικότητα – Πολυμορφισμός

Ας υποθέσουμε ότι έχουμε αναπτύξει στον κώδικά μας τις κλάσεις που αντιπροσωπεύουν τις έννοιες «Βιβλίο» και «Φοιτητής». Επιθυμούμε σε δεύτερη φάση να επεκτείνουμε το πρόγραμμα και στα άλλα μέλη της πανεπιστημιακής κοινότητας· οποιοσδήποτε ανήκει στο προσωπικό του Πανεπιστημίου μπορεί να δανειστεί βιβλία, με διαφορετικές προϋποθέσεις, ανάλογα με το αν είναι φοιτητής, καθηγητής, ή εργαζόμενος. Θα μπορούσαμε να ορίσουμε κατάλληλες κλάσεις για κάθε κατηγορία. Σε αυτήν την προσέγγιση όμως, θα διαπιστώναμε ότι θα είχαμε μεγάλα τμήματα κώδικα να επαναλαμβάνονται, ουσιαστικά αυτούσια, καθώς πολλές λεπτομέρειες της κάθε κατηγορίας δεν παίζουν ιδιαίτερο ρόλο. Π.χ., ο κώδικας που τυπώνει τα βιβλία που έχει δανειστεί ένας χρήστης είναι ανεξάρτητος από την κατηγορία του χρήστη (φοιτητής, καθηγητής, κλπ.)· όμως, θα πρέπει να επαναληφθεί για κάθε κατηγορία (κλάση) που θα ορίσουμε. Είναι προτιμότερο να υλοποιήσουμε στο πρόγραμμά μας τη γενική έννοια του «προσωπικού του Πανεπιστημίου» και να αναπτύξουμε τον κώδικα με αυτήν ως βάση.

Βασικό στοιχείο του αντικειμενοστρεφούς προγραμματισμού είναι η δυνατότητα να παράγουμε ειδικές έννοιες από πιο γενικές. Μπορούμε, επομένως, να δημιουργήσουμε μια *ιεραρχία* από κλάσεις· οι έννοιες «Φοιτητής», «Καθηγητής», «Εργαζόμενος» είναι εξειδικεύσεις της έννοιας «Προσωπικό» και έχουν όλες τις ιδιότητές της. Το χαρακτηριστικό αυτό λέγεται *κληρονομικότητα (inheritance)*.

Η εξειδίκευση δεν εμποδίζει τις παραγόμενες έννοιες να έχουν, πιθανόν, επιπλέον χαρακτηριστικές ιδιότητες ή να τροποποιούν τη συμπεριφορά που υπαγορεύει η βασική έννοια. Όπως είναι φυσικό, και οι ειδικές έννοιες μπορούν να αποτελέσουν βάση για άλλες· έτσι, ο «Μεταπτυχιακός Φοιτητής» έχει όλες τις ιδιότητες του «Φοιτητή» (και μερικές ακόμα), και βέβαια, όλα τα χαρακτηριστικά του «Προσωπικού» (της πιο γενικής κλάσης).

Κάθε παραγόμενος τύπος εμπεριέχει τη βασική του κλάση και μπορεί να χρησιμοποιηθεί και να συμπεριφερθεί όπως αυτή. Ένα θεμελιώδες χαρακτηριστικό της ιεραρχίας είναι ότι οι ιδιότητες της παραγόμενης έννοιας δε χάνονται όταν αυτή χρησιμοποιείται (με κατάλληλο τρόπο) στη θέση της βασικής έννοιας· έτσι, ο ίδιος ακριβώς κώδικας, γραμμένος για τη βασική έννοια, μπορεί να έχει διαφορετικό αποτέλεσμα κατά την εκτέλεση, ανάλογα με την ειδική έννοια για την οποία θα κληθεί. Αυτό το χαρακτηριστικό λέγεται *πολυμορφισμός (polymorphism)*.

Όπως ίσως αντιλαμβάνεστε, η σχεδίαση και ανάπτυξη ενός προγράμματος προσανατολισμένου σε αντικείμενα είναι θεμελιωδώς διαφορετικές από τη μεθοδολογία που ακολουθούμε στο διαδικαστικό προγραμματισμό. Στο νέο τρόπο που παρου-

σιάζουμε, οργανώνουμε το πρόβλημά μας σε έννοιες, προσδιορίζουμε τις ιδιότητές τους και τις πράξεις που μπορούμε να εκτελέσουμε σε αυτές. Επιδιώκουμε να δημιουργήσουμε κατάλληλη ιεραρχία, εξάγοντας κοινές ιδιότητες εννοιών του προβλήματός μας (ή και παρόμοιων προβλημάτων που θα συναντήσουμε στο μέλλον) σε όσο πιο γενικές κλάσεις γίνεται και να αναπτύξουμε τον κώδικα βάσει αυτών και των αλληλεπιδράσεών τους.

Σύγχρονες γλώσσες προγραμματισμού όπως η C++, η Fortran 2003, η Java και η python υποστηρίζουν μεταξύ άλλων τρόπων οργάνωσης κώδικα, τη μεθοδολογία του αντικειμενοστρεφούς προγραμματισμού. Στο επόμενο κεφάλαιο θα παρουσιαστεί με λεπτομέρειες ο μηχανισμός που παρέχει η C++ για την υλοποίηση όσων αναπτύχθηκαν παραπάνω.

Κεφάλαιο 14

Ορισμός Κλάσης

14.1 Εισαγωγή

Στο προηγούμενο κεφάλαιο περιγράψαμε σε γενικές γραμμές ένα τρόπο οργάνωσης των δεδομένων κάποιου προγράμματος που υποστηρίζει τη δανειστική λειτουργία μιας πανεπιστημιακής Βιβλιοθήκης. Η μεθοδολογία που ακολουθήσαμε είναι αυτή του αντικειμενοστρεφούς προγραμματισμού: δημιουργούμε νέους τύπους (κλάσεις) για τις βασικές έννοιες του προβλήματός μας, το «Βιβλίο», το «Φοιτητή», τον «Καθηγητή», το «Προσωπικό του Πανεπιστημίου», κλπ. Οι δομές αυτές περιλαμβάνουν όλες τις ποσότητες που αντιστοιχούν στις συγκεκριμένες έννοιες, συνδέονται με συγκεκριμένες ειδικές συναρτήσεις που έχουν δικαίωμα πρόσβασης στην εσωτερική αναπαράσταση και συσχετίζονται σε ιεραρχία κλάσεων ώστε να αποφεύγουμε την επανάληψη κώδικα. Παρακάτω θα παρουσιάσουμε λεπτομερώς πώς υλοποιεί η C++ τη συγκεκριμένη οργάνωση του κώδικα.

14.2 Κατασκευή τύπου

Από την ανάλυση του προβλήματος που κάναμε στο προηγούμενο κεφάλαιο, διαπιστώσαμε ότι χρειαζόμαστε δύο νέους τύπους στον κώδικά μας (σε πρώτη φάση τουλάχιστον), σε πλήρη αντιστοιχία με τις έννοιες του προβλήματός μας: τον τύπο του «Φοιτητή» και τον τύπο του «Βιβλίου». Ας επικεντρωθούμε στη δημιουργία του πρώτου, ως παράδειγμα. Αφού ορίσουμε αυτόν τον τύπο, θα μπορούμε να δηλώσουμε «αντικείμενά» του (ποσότητες αυτού του τύπου).

Θυμηθείτε όσα αναφέραμε για την εμβέλεια των μεταβλητών, §2.8: υπάρχουν οι διαδικασίες δημιουργίας και καταστροφής που εκτελούνται αυτόματα όταν ορίζουμε μια μεταβλητή (δημιουργία) και όποτε τελειώνει η «ζωή» της (καταστροφή).

Αυτές οι διαδικασίες πρέπει να προσδιοριστούν για τον νέο τύπο. Επιπλέον, πρέπει να προσδιορίσουμε πώς «συμπεριφέρεται» όταν συμμετέχει σε εκφράσεις και τι νόημα έχει (αν έχει) η δράση διάφορων τελεστών σε μεταβλητή τέτοιου τύπου. Πριν απ' όλα όμως, πρέπει να ορίσουμε πώς αποθηκεύεται η πληροφορία στον τύπο.

Συνοπτικά, η κατασκευή ενός νέου τύπου περιλαμβάνει

- τη δήλωση των ποσοτήτων που χρειάζονται για την αποθήκευση των πληροφοριών, και γενικά 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 αντιγραφούν 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.
- τον ορισμό συναρτήσεων για την προσπέλαση ή τροποποίηση της εσωτερικής αναπαράστασης.

Πιθανόν να χρειάζεται, αν έχουν νόημα, να ορίσουμε

- τελεστές που δρουν σε αντικείμενα του συγκεκριμένου τύπου και
- πώς γίνεται η μετατροπή του συγκεκριμένου τύπου σε άλλο.

14.3 Εσωτερική αναπαράσταση – συναρτήσεις πρόσβασης

Ας δημιουργήσουμε σταδιακά την κλάση με το όνομα Student, η οποία θα περιγράψει τελικά την έννοια «Φοιτητής». Το πρώτο στάδιο του ορισμού της είναι ο κώδικας

```
class Student {
...
};
```

Ο ορισμός της κλάσης (του νέου τύπου, δηλαδή) εισάγεται με μία από τις λέξεις `class` ή `struct` (θα αναφερθούμε παρακάτω στη διαφορά τους) ακολουθούμενης

από το όνομα του τύπου που δημιουργούμε.[1] Μέσα στα άγκιστρα '{}' θα παραθέσουμε τα μέλη του· αυτά υλοποιούν χαρακτηριστικά και ιδιότητές του. Προσέξτε το απαραίτητο καταληκτικό ';' που ολοκληρώνει τον ορισμό. Η παράθεση του συγκεκριμένου κώδικα μπορεί να γίνει σε οποιοδήποτε σημείο του κώδικά μας, αφού λάβουμε υπόψη την επιθυμητή εμβέλεια που θέλουμε να έχει.

Για να αναπαραστήσουμε ένα Student χρειαζόμαστε τουλάχιστον πέντε ποσότητες: μια σειρά χαρακτήρων για να αποθηκεύσουμε το όνομα, άλλη μια για να αποθηκεύσουμε το Τμήμα, ένα ακέραιο για τον αριθμό μητρώου, ένα άλλο ακέραιο για το έτος εισαγωγής και μια λογική ποσότητα για να αποθηκεύσουμε το αν ο φοιτητής είναι ενεργός ή όχι. Για μια ρεαλιστική περιγραφή χρειάζονται και άλλα μέλη αλλά για τις ανάγκες του παρόντος οι πέντε που αναφέραμε αρκούν. Ας τις ονομάσουμε name, department, am, year, active. Όπως αναφέραμε στο προηγούμενο κεφάλαιο, είναι επιθυμητό να περιορίσουμε την άμεση πρόσβαση σε αυτές. Θα επιτρέπεται σε συγκεκριμένες συναρτήσεις η ανάγνωση ή η μεταβολή της τιμής τους.

Η δήλωσή τους ως μέλη γίνεται ως εξής

```
class Student {

private:
    std::string name;
    std::string department;
    int am;
    int year;
    bool active;
};
```

Της παράθεσης των συγκεκριμένων μελών προηγείται η ετικέτα `private:` με την οποία προσδιορίζεται ότι η πρόσβαση στα συγκεκριμένα μέλη επιτρέπεται μόνο από άλλα μέλη της κλάσης. Εννοείται βέβαια, ότι πρέπει να συμπεριλάβουμε με οδηγίες `#include` τους αναγκαίους headers (μέχρι στιγμής, το `<string>`).

Όπως αναμένουμε, στο επόμενο στάδιο χρειάζεται να συμπληρώσουμε τον ορισμό της κλάσης με τις συναρτήσεις–μέλη που θα πρέπει να χρησιμοποιήσει το υπόλοιπο πρόγραμμά μας για να έχει πρόσβαση στα «εσωτερικά» μέλη:

```
class Student {
public:
    std::string const & getName() const {return name;}
    std::string const & getDept() const {return department;}
    int const & getAm() const {return am;}
    int const & getYear() const {return year;}
    bool const & getActive() const {return active;}

    void setActive(bool a) {active = a;}
```

[1] Ειδικές κλάσεις μπορούν να οριστούν με το `union`· δεν θα αναφερθούμε περισσότερο σε αυτές.

```
private:
    std::string name;
    std::string department;
    int am;
    int year;
    bool active;
};
```

Ο ορισμός των νέων μελών έπεται της ετικέτας `public:`. Με αυτό τον τρόπο υποδηλώνεται ότι στα συγκεκριμένα μέλη η πρόσβαση και χρήση είναι επιτρεπτή από οποιοδήποτε σημείο του κώδικά μας. Σημειώστε ότι οι ετικέτες μπορούν να επαναλαμβάνονται στο σώμα της κλάσης, δεν έχουν προκαθορισμένη σειρά και δεν είναι απαραίτητο να ακολουθούνται από δηλώσεις μελών[2]. Η πλησιέστερη, προς τα επάνω, ετικέτα είναι αυτή που καθορίζει την εμβέλεια των μελών που ακολουθούν. 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`. Η επιθυμητή συμπεριφορά είναι σχεδόν πάντοτε η αυτόματη ενθυλάκωση των μελών εκτός από συγκεκριμένα που ρητά εξαιρούνται· επομένως, ο ορισμός νέου τύπου στην πράξη γίνεται συνήθως με το `class`. Οι σχετικές θέσεις των μελών μπορούν να είναι οποιεσδήποτε. Είναι θέμα προσωπικής προτίμησης η παράθεση στο τέλος των αναγκαίων μελών για την εσωτερική αναπαράσταση.

Παρατηρήστε ότι έχουμε μία συνάρτηση–μέλος που μεταβάλλει την τιμή άλλου μέλους, την `setActive()`. Ο τρόπος ορισμού της δε διαφέρει από τις κοινές συναρτήσεις.[3] Η πρόσβαση στο τροποποιούμενο μέλος γίνεται χρησιμοποιώντας απευθείας το όνομά του. Προσέξτε ότι οποιαδήποτε συνάρτηση–μέλος μπορεί να αναφέρεται σε μέλη που ορίζονται παρακάτω στην κλάση καθώς οποιαδήποτε δήλωση οπουδήποτε εντός κλάσης θεωρείται γνωστή για τα υπόλοιπα μέλη. Το σκεπτικό για την ύπαρξη μόνο μίας συνάρτησης μεταβολής είναι ότι στον κώδικά μας θα δημιουργούμε αντικείμενα τύπου Student με συγκεκριμένες τιμές για τα μέλη της εσωτερικής αναπαράστασης (θα δούμε παρακάτω πώς). Μετά τη δημιουργία δεν έχει νόημα η μεταβολή του ονόματος, του τμήματος, του έτους εισαγωγής κλπ. οπότε δεν παρέχονται σχετικές συναρτήσεις–μέλη. Αντίθετα, το αν ένας φοιτητής είναι ενεργός ή όχι μπορεί να αλλάξει κατά τη διάρκεια φοίτησης. Σε άλλο πρόβλημα, θα μπορούσαμε να παρέχουμε συναρτήσεις αλλαγής και άλλων μελών.

Οι συναρτήσεις μεταβολής της εσωτερικής αναπαράστασης μιας κλάσης, όπως είναι φυσικό, δεν μπορούν να κληθούν για αντικείμενα που έχουν δηλωθεί ως `const` καθώς αυτά δεν μπορούν να τροποποιηθούν. Αντίθετα, οι συναρτήσεις–μέλη `getName()`, `getDept()`, `getAm()`, `getYear()`, `getActive()`, οι οποίες δεν μεταβάλλουν την κατάσταση του αντικειμένου για το οποίο γίνεται η κλήση τους, η λέξη `const` ακολουθεί τη λίστα ορισμάτων τους και αποτελεί μέρος της δήλωσής τους. Οι συγκεκριμένοι ορισμοί (με τη λέξη `const`) επιτρέπουν στις συναρτήσεις να κληθούν

[2] Την τρίτη ετικέτα που επιτρέπεται, την `protected:`, θα τη συναντήσουμε αργότερα.

[3] Είναι επιτρεπτό αλλά όχι αναγκαίο να ακολουθεί τον ορισμό το ελληνικό ερωτηματικό, ';', όπως γίνεται υποχρεωτικά για τα μέλη που δεν είναι συναρτήσεις.

και για σταθερά αντικείμενα. Γνωρίζουμε ήδη ότι η κλήση μιας συνάρτησης–μέλους για κάποιο αντικείμενο κλάσης γίνεται γράφοντας το όνομα του αντικειμένου, το σύμβολο '.' και το όνομα της συνάρτησης (με όλα τα ορίσματα που μπορεί να έχει αυτή). Επομένως, αν το s είναι Student, στον παρακάτω κώδικα

```
Student & s1{s};
Student const & s2{s};

s1.setActive(false);  // correct
s2.setActive(false);  // error
auto y1 = s1.getYear();  // correct
auto y2 = s2.getYear();  // correct
```

η κλήση της `setYear()` για το s1 είναι αποδεκτή ενώ για το σταθερό s2 δεν επιτρέπεται. Αντίθετα η `getYear()` μπορεί να κληθεί και για σταθερά και για μεταβλητά αντικείμενα.

Η συμπλήρωση της δήλωσης συγκεκριμένων και κατάλληλων συναρτήσεων–μελών με τη λέξη `const` δεν είναι αυστηρά απαραίτητη, είναι όμως αναγκαία για να μπορούν να κληθούν αυτές για αντικείμενα που έχουν δηλωθεί ή με οποιοδήποτε τρόπο είναι `const`.

Όπως αναφέραμε, τα μέλη που αποτελούν την υλοποίηση μιας κλάσης, δηλαδή οι ποσότητες των θεμελιωδών τύπων ή άλλων κλάσεων που προσδιορίζουν την οργάνωσή της στη μνήμη, είναι καλό να ορίζονται ως `private` και να παρέχονται, αν χρειάζεται, συναρτήσεις–μέλη δηλούμενες ως `public` για το χειρισμό τους εκτός κλάσης.

14.4 Οργάνωση κώδικα κλάσης

Όπως καταλαβαίνετε, ο χρήστης μιας κλάσης, σε αντίθεση με τον προγραμματιστή της, ενδιαφέρεται μόνο για τις δηλώσεις των μελών (ειδικά, αυτών που είναι `public`) και όχι για τους ορισμούς τους. Είναι επιθυμητό για λόγους ευκρίνειας, να μετακινήσουμε την υλοποίηση των συναρτήσεων–μελών εκτός του σώματος μιας κλάσης. Για παράδειγμα, στη Student μπορούμε να ορίσουμε τις συναρτήσεις–μέλη που γράψαμε μέχρι τώρα, έξω από το κυρίως σώμα της ως εξής:

```
class Student {
public:
    std::string const & getName() const;
    std::string const & getDept() const;
    int const & getAm() const;
    int const & getYear() const;
    bool const & getActive() const;

    void setActive(bool a);
```

```
private:
    std::string name;
    std::string department;
    int am;
    int year;
    bool active;
};

std::string const &
Student::getName() const
{
  return name;
}

std::string const &
Student::getDept() const
{
  return department;
}

int const &
Student::getAm() const
{
  return am;
}

int const &
Student::getYear() const
{
  return year;
}

bool const &
Student::getActive() const
{
  return active;
}

void
Student::setActive(bool a)
{
  active = a;
}
```

Παρατηρήστε ότι στο σώμα *δηλώνονται* τα μέλη, ενώ εκτός αυτού *ορίζονται*. Μία διαφορά από τον προηγούμενο τρόπο είναι ότι τα ονόματα των μελών στους ορισμούς έξω από το σώμα της κλάσης πρέπει να συμπληρωθούν με το όνομα της κλάσης στην οποία ανήκουν (το τμήμα `Student::`). Η άλλη διαφορά είναι ότι οι συναρτήσεις που ορίζονται εντός της κλάσης θεωρούνται ότι είναι `inline` (§7.13)· αντίθετα, οι ορισμοί εκτός πρέπει να συμπληρωθούν με το `inline` για όσες το κρίνουμε σημαντικό.

Ο διαχωρισμός του ορισμού μιας κλάσης σε δύο τμήματα—κυρίως σώμα (δήλωση) και ορισμοί των μελών—μας επιτρέπει να συγκεντρώνουμε σε αρχείο header (π.χ. *Student.h*) το πρώτο τμήμα (αλλά *και όσες συναρτήσεις-μέλη είναι* `inline` *ή* `template`, δείτε τις §7.13 και §7.11), και σε άλλο αρχείο (π.χ. *Student.cpp*) την υλοποίησή της. Κατά τη μεταγλώττιση, ο κώδικας που χρησιμοποιεί την κλάση αρκεί να περιλαμβάνει (με οδηγία `#include "Student.h"`) το αρχείο header ενώ η υλοποίησή της μπορεί να μεταγλωττιστεί ανεξάρτητα.

14.5 Συναρτήσεις Δημιουργίας

Η επόμενη συμπλήρωση της κλάσης είναι ο προσδιορισμός του τρόπου *δημιουργίας* ενός αντικειμένου της. Δημιουργία έχουμε όταν, μεταξύ άλλων,

- δηλώνουμε μια μεταβλητή συγκεκριμένου τύπου,
- περνάμε μια ποσότητα ως όρισμα σε συνάρτηση που δέχεται τέτοιο τύπο,
- επιστρέφουμε ποσότητα από συνάρτηση.

Ας εξετάσουμε τις περιπτώσεις αναλυτικά:

14.5.1 Κατασκευαστής (constructor)

Επιθυμούμε να μπορούμε να ορίσουμε μια μεταβλητή τύπου Student ως εξής:

```
Student s{"George␣Thomson", "Physics", 123, 2010, true};
```

Με αυτή την εντολή, θέλουμε το Student s να κατασκευαστεί με συγκεκριμένες τιμές για τα μέλη του, αυτές που παραθέτουμε στη λίστα. Επίσης, θα μας διευκολύνει αν μπορούμε να κατασκευάζουμε ένα Student που να είναι ενεργός (active) χωρίς να το προσδιορίζουμε ρητά. Θέλουμε, δηλαδή, με τη δήλωση

```
Student p{"George␣Thomson", "Physics", 123, 2010};
```

να δημιουργούμε πάλι ένα αντικείμενο με τις προσδιοριζόμενες τιμές για κάποια μέλη του και την τιμή `true` για το active.

Μια συνάρτηση ειδικής μορφής περιγράφει στην κλάση τούς δύο παραπάνω τρόπους δημιουργίας: στο τμήμα public της κλάσης γράφουμε τη δήλωση

```
Student(std::string const & onoma, std::string const & dept,
        int armi, int y, bool act = true);
```

ενώ εκτός της κλάσης δίνουμε τον ορισμό

```
Student::Student(std::string const & onoma, std::string const & dept,
            int armi, int y, bool act)
            : name{onoma}, department{dept}, am{armi}, year{y},
              active{act}
{}
```

Παρατηρήστε την ιδιαίτερη μορφή της δήλωσης: η συνάρτηση

- έχει ως όνομα το όνομα της κλάσης και
- δεν έχει επιστρεφόμενο τύπο.

Ιδιαιτερότητα παρουσιάζει η συνάρτηση και στον ορισμό: γίνεται ιδιότυπη απόδοση αρχικών τιμών στα μέλη που αποτελούν την υλοποίηση του αντικειμένου, αμέσως μετά τα ορίσματα και πριν το (κενό) σώμα της συνάρτησης, με λίστα που ακολουθεί το χαρακτήρα ‘:΄. Τυχαίνει στο συγκεκριμένο παράδειγμα να μην χρειάζονται εντολές στο σώμα της συνάρτησης.

Επιλέξαμε να δηλώσουμε μία συνάρτηση με προκαθορισμένη τιμή στο τελευταίο όρισμα (§7.6) αντί να γράψουμε δύο συναρτήσεις με διαφορετικό πλήθος ορισμάτων (και να βασιστούμε στο overloading για την επιλογή). Ως τρίτη εναλλακτική υπάρχει η δυνατότητα να μην προσδιορίσουμε τιμή για κάποιο όρισμα αλλά να ορίσουμε προκαθορισμένη τιμή κατά τη δήλωση του αντίστοιχου μέλους της κλάσης, δηλαδή, στην εσωτερική αναπαράσταση να γράψουμε `bool active{true};`.

Κάθε συνάρτηση–μέλος, με όνομα το όνομα της κλάσης και χωρίς επιστρεφόμενο τύπο, είναι ένας *constructor* («κατασκευαστής») της κλάσης. Η επιλογή του constructor που θα κληθεί σε μια δήλωση, αν είναι διαθέσιμοι περισσότεροι του ενός, γίνεται με τους κανόνες του overloading (§7.10). Αν η δήλωση γίνει εκτός κλάσης, συμμετέχουν στην επιλογή μόνο οι public constructors. Αν η δήλωση είναι σε μέλος της κλάσης, συνυπολογίζονται και οι τυχόν private constructors.

Ας εξετάσουμε πώς λειτουργεί ο μοναδικός μέχρι στιγμής κατασκευαστής της κλάσης Student. Καλείται *αυτόματα* όταν γράψουμε δήλωσεις της μορφής

```
Student s{"George Thomson", "Physics", 123, 2010, true};
Student p{"George Thomson", "Physics", 123, 2010};
```

όταν, δηλαδή, θελήσουμε να δημιουργήσουμε ένα Student παραθέτοντας δύο ποσότητες με τύπους `std::string` (ή κάποιον που μπορεί να μετατραπεί αυτόματα σε αυτόν, όπως εδώ), δύο άλλες ακέραιες ποσότητες και, προαιρετικά, μία λογική τιμή. Η λίστα τιμών περιλαμβάνει τις αρχικές τιμές των μελών για το αντικείμενο που δημιουργείται. Η κλήση του συγκεκριμένου κατασκευαστή ισοδυναμεί με τις εντολές

```
std::string name{onoma};
std::string department{dept};
int am{armi};
```

```
int year{y};
bool active{act};
```

Προσέξτε ότι η απόδοση αρχικής τιμής στα μέλη γίνεται *με τη σειρά που αυτά δηλώνονται στο σώμα της κλάσης, ανεξάρτητα από τη σειρά με την οποία παρατίθενται στον constructor.*

Θα μπορούσε κανείς να θεωρήσει ότι πρέπει να γράψουμε

```
Student::Student(std::string const & onoma, std::string const & dept,
            int armi, int y, bool act)
{
  name = onoma;
  department = dept;
  am = armi;
  year = y;
  active = act;
}
```

Τυπικά δεν είναι λάθος ο παραπάνω ορισμός· με την αυτόματη κλήση του συκγεκριμένου κατασκευαστή, δημιουργείται το αντικείμενο σαν να εκτελούνται οι εντολές για τα μέλη του

```
std::string name;
name = onoma;

std::string department;
department = dept;

int am;
am = armi;

int year;
year = y;

bool active;
active = act;
```

Προσέξτε τη διαφορά: ο εναλλακτικός ορισμός δημιουργεί τις μεταβλητές name, department, am, year, active δίνοντας αρχική τιμή στα name και department το κενό `std::string` ενώ οι ακέραιες και η λογική ποσότητες είναι απροσδιόριστες. Κατόπιν, γίνεται εκχωρηση των επιθυμητών τιμών. Αντίθετα, ο αρχικός ορισμός δίνει τις επιθυμητές τιμές ως *αρχικές*. Ως εκ τούτου, μπορεί να χρησιμοποιηθεί αν υπάρχουν μέλη που έχουν δηλωθεί ως `const` ή αναφορές, κάτι που δεν μπορεί να κάνει ο εναλλακτικός ορισμός.

Συμπερασματικά, όποτε μπορούμε να αποδώσουμε αρχικές τιμές στην αναπαράσταση ενός αντικειμένου είναι καλό να χρησιμοποιούμε στον ορισμό του constructor

τη λίστα απόδοσης τιμών που εισάγεται με το ':' πριν το σώμα της συνάρτησης. Βέβαια, υπάρχει περίπτωση να χρειάζεται η εκτέλεση κάποιων εντολών ώστε να προσδιοριστεί η αρχική τιμή κάποιου μέλους. Δεν είναι πάντα εφικτό να γίνει κάτι τέτοιο στη λίστα απόδοσης τιμών και θα πρέπει αναγκαστικά να γίνει στο σώμα του κατασκευαστή.

14.5.2 Κατασκευαστής αντίγραφου (copy constructor)

Για να μπορούμε να δημιουργήσουμε ένα αντικείμενο τύπου Student με αντιγραφή από άλλο, ήδη κατασκευασμένο, Student ή για να μπορούμε να περάσουμε αντικείμενο ως όρισμα συνάρτησης τύπου Student, πρέπει να ορίσουμε τον *κατασκευαστή με αντιγραφή* (copy constructor). Στο σώμα της κλάσης πρέπει να συμπεριλάβουμε τη δήλωση

```
Student(Student const & other);
```

και να παραθέσουμε ένα ορισμό της ειδικής συνάρτησης:

```
Student::Student(Student const & other)
        : name{other.name}, department{other.department},
          am{other.am}, year{other.year}, active{other.active}
{}
```

Ο συγκεκριμένος κατασκευαστής καλείται αυτόματα στις (ισοδύναμες) δηλώσεις των t1, t2, t3:

```
Student s{"George␣Thomson", "Physics", 123, 2010, true};
Student t1{s};
Student t2(s);
Student t3 = s;
```

Ο copy constructor μιας κλάσης χρησιμοποιείται ακόμα όποτε δίνουμε ως όρισμα συνάρτησης ένα αντικείμενο αυτής της κλάσης. Έτσι, αν έχουμε τη συνάρτηση

```
void f(Student t);
```

η κλήση `f(s)` παρακάτω δημιουργεί το t με αντιγραφή από το s καλώντας τον copy constructor της Student:

```
Student s{"George␣Thomson", "Physics", 123, 2010, true};
f(s);
```

Κατά την κλήση, ουσιαστικά εκτελείται η εντολή `Student t{s};` προτού χρησιμοποιηθεί το t στο σώμα της συνάρτησης. Προσέξτε ότι αν το όρισμα της συνάρτησης είναι αναφορά, δεν γίνεται αντιγραφή, δηλαδή δημιουργία νέας ποσότητας με απόδοση αρχικής τιμής, αλλά χρησιμοποιείται απευθείας η ποσότητα που δίνεται στη συνάρτηση.

Το αντικείμενο που περνά στον copy constructor μιας κλάσης πρέπει να δοθεί ως αναφορά, σταθερή ή μη, καθώς δεν έχει οριστεί πώς γίνεται η αντιγραφή του ορίσματος πριν ολοκληρωθεί το σώμα του copy constructor.

14.5.3 Κατασκευαστής με μετακίνηση (move constructor)

Ο κατασκευαστής με μετακίνηση είναι παρόμοιος με τον κατασκευαστή με αντιγραφή. Προκαλεί τη δημιουργία αντικειμένου από άλλο, μεταφέροντας τη μνήμη που δεσμεύει αυτό. Το αρχικό αντικείμενο, αν όντως γίνει η μεταφορά μνήμης, απομένει σε απροσδιόριστη κατάσταση. Μπορούμε μόνο να του μεταφέρουμε την κατάσταση άλλου αντικειμένου (κατά την εκχώρηση τιμής με μετακίνηση που θα δούμε παρακάτω).

Αν ορίσουμε κατασκευαστή με μετακίνηση σε μια κλάση, έχει προτεραιότητα έναντι του copy constructor όποτε έχουμε δημιουργία αντικειμένου από ποσότητα που έχει προσαρμοστεί με τη συνάρτηση `std::move()` του `<utility>` (§2.17.1) ή από επιστρεφόμενη ποσότητα συνάρτησης. Επομένως, αν υποθέσουμε ότι το s είναι αντικείμενο της κλάσης Student και υπάρχουν οι συναρτήσεις `void f(Student r);` και `Student g();`, ο move constructor, αν υπάρχει, καλείται στις παρακάτω εντολές[4]:

```
Student p = std::move(s);
f(std::move(p));
Student q = f();
```

Η δήλωσή του για την κλάση Student είναι

```
Student(Student && other);
```

ενώ ο ορισμός του μπορεί να είναι

```
Student::Student(Student && other)
        : name{std::move(other.name)},
          department{std::move(other.department)},
          am{std::move(other.am)}, year{std::move(other.year)},
          active{std::move(other.active)}
{}
```

Παρατηρήστε ότι προσαρμόζουμε με το `std::move()` όλα τα μέλη της ποσότητας που θέλουμε να μετακινήσουμε. Όταν κληθεί, ο move constructor μεταφέρει τη μνήμη που καταλαμβάνουν τα μέλη του Student other στα αντίστοιχα μέλη της νέας μεταβλητής. Μετά την κλήση, τα μέλη του other δεν περιέχουν τίποτε (για την ακρίβεια, η κατάστασή τους δεν είναι γνωστή).

Η μετακίνηση, όταν γίνεται, είναι προφανώς πολύ πιο γρήγορη από την αντιγραφή, ειδικά για μέλη που δεσμεύουν αρκετά bytes μνήμης, όπως τα δύο μέλη τύπου `std::string` ενός αντικειμένου τύπου Student. Γι᾽ αυτό, καλό είναι να ορίζουμε move constructor σε κλάσεις που δεσμεύουν αρκετή μνήμη ή άλλους πόρους του συστήματος.

[4] αν δεν υπάρχει, καλείται ο copy constructor.

14.5.4 Προκαθορισμένοι κατασκευαστές

Σε μια κλάση X που *δεν έχουμε ορίσει ρητά κανένα κατασκευαστή* (εκτός των κατασκευαστών μετακίνησης και αντιγραφής), ο μεταγλωττιστής πρέπει να εξασφαλίσει ότι μπορούν να δημιουργηθούν αντικείμενα αυτής. Αυτόματα ορίζει στο τμήμα public τον κατασκευαστή `inline X(){}`, που δεν δέχεται ορίσματα και έχει κενό σώμα. Ο συγκεκριμένος χρησιμοποιείται όποτε έχουμε δήλωση αντικειμένου της κλάσης X με τις εντολές `X a;` ή `X a{};` και καλεί τους αντίστοιχους κατασκευαστές των μελών της κλάσης, με τη σειρά που παρατίθενται στον ορισμό της κλάσης. Προσέξτε ότι η δήλωση

```
X a();
```

δεν είναι δήλωση μεταβλητής (η οποία θα προκαλέσει την κλήση του default constructor) αλλά *δήλωση συνάρτησης* (που δεν δέχεται ορίσματα και επιστρέφει ποσότητα τύπου X).

Αν έχουμε ορίσει κάποιο κατασκευαστή, ο μεταγλωττιστής δεν θα προχωρήσει στον αυτόματο ορισμό του προκαθορισμένου constructor. Αν τυχόν τον χρειαζόμαστε, πρέπει ή να τον ορίσουμε ρητά ή να ζητήσουμε από τον compiler να το κάνει, με την εντολή (στο τμήμα public)

```
X() = default;
```

Αν για κάποιο λόγο δεν επιθυμούμε να οριστεί αυτόματα ο default constructor μιας κλάσης X (π.χ. γιατί δεν υπάρχουν προκαθορισμένοι κατασκευαστές για τα μέλη της) μπορούμε να δώσουμε την εντολή

```
X() = delete;
```

Μια κλάση μπορεί να έχει ως μέλη ποσότητες ενσωματωμένων τύπων· οι κατασκευαστές τους τεχνικά υπάρχουν αλλά δεν αποδίδουν αρχική τιμή στα αντικείμενα.

Αντίστοιχα, αν σε μία κλάση X *δεν ορίσουμε ρητά ένα κατασκευαστή με αντιγραφή*, ο μεταγλωττιστής, όταν προσπαθήσουμε να δημιουργήσουμε ένα αντικείμενο με αντιγραφή άλλου της ίδιας κλάσης, ορίζει αυτόματα έναν copy constructor με δήλωση `inline X(X const & other)` (ή `inline X(X & other)` ανάλογα με κάποια κριτήρια), ο οποίος δημιουργεί ένα αντικείμενο αντιγράφοντας κάθε μέλος από το όρισμα στο αντίστοιχο του νέου αντικειμένου, καλώντας τους copy constructors των μελών με τη σειρά που παρατίθενται στην κλάση. Αυτή η αντιγραφή είναι συνήθως αποδεκτή· όταν όμως ως μέλος του τύπου περιλαμβάνεται ένας δείκτης (§2.18) ή ποσότητα για την οποία δεν έχει νόημα (ή δεν έχει το αναμενόμενο νόημα) η αντιγραφή (π.χ. `std::ifstream`), ο προγραμματιστής της κλάσης πρέπει να ορίσει ρητά τι επιθυμεί να συμβεί κατά την αντιγραφή, γράφοντας την αντίστοιχη συνάρτηση.

Στο παράδειγμα της κλάσης Student ο προκαθορισμένος copy constructor είναι αποδεκτός. Μπορούμε να ζητήσουμε ρητά να παραχθεί αυτός, αντί να γράφουμε το δικό μας κατασκευαστή με αντιγραφή, με την εντολή (στο τμήμα public)

```
Student(Student const & other) = default;
```

Αν υπάρχει λόγος, μπορούμε να εμποδίσουμε το μεταγλωττιστή να παραγάγει τον υπονοούμενο copy constructor μιας κλάσης X, ακόμα και όταν δεν γράψουμε δικό μας, γράφοντας

```
X(X const & other) = delete;
```

Με την παραπάνω εντολή, αποτυγχάνουν οποιαδήποτε δήλωση θα δημιουργούσε αντικείμενο της X με αντιγραφή άλλου και οποιαδήποτε κλήση συνάρτησης με όρισμα τύπου X.

Αν σε μια κλάση X δεν ορίσουμε ρητά

- κατασκευαστές με αντιγραφή ή μετακίνηση,
- τελεστές εκχώρησης με αντιγραφή ή μετακίνηση (§14.7),
- καταστροφέα (§14.6),

ο μεταγλωττιστής παρέχει αυτόματα τον προκαθορισμένο move constructor με δήλωση `inline X(X && other)` στο τμήμα public της κλάσης. Αυτός προκαλεί μετακίνηση των μελών ενός αντικειμένου με τη σειρά που παρατίθενται στον ορισμό της κλάσης. Στο παράδειγμα του Student ο υπονοούμενος move constructor είναι αποδεκτός. Μπορούμε να ζητήσουμε ρητά να παραχθεί αυτός, αντί να γράφουμε το δικό μας κατασκευαστή με μετακίνηση, με την εντολή

```
Student(Student && other) = default;
```

Αν υπάρχει λόγος, μπορούμε να εμποδίσουμε το μεταγλωττιστή να παραγάγει τον προκαθορισμένο move constructor μιας κλάσης X γράφοντας

```
X(X const && other) = delete;
```

14.6 Συνάρτηση καταστροφής (Destructor)

Η επόμενη προσθήκη στην κλάση Student περιγράφει πώς καταστρέφεται ένα αντικείμενό της. Η συνάρτηση `~Student(){}` ονομάζεται destructor (καταστροφέας) και καλείται *αυτόματα* όποτε χρειαστεί η καταστροφή αντικειμένου της κλάσης Student. Για αντικείμενο που δεν είναι στατικό (§2.2), αυτό συμβαίνει όποτε η ροή του κώδικα συναντά το τέλος της εμβέλειάς του ή την εντολή `delete` (§5.2.4) για αυτό. Για τα στατικά αντικείμενα, η καταστροφή γίνεται στο τέλος του προγράμματος.

Ο destructor μιας κλάσης είναι μοναδικός και δεν παίρνει ορίσματα.

Και στην περίπτωση του destructor, αν σε μία κλάση X δεν τον ορίσουμε ρητά και χρειαστεί η καταστροφή ενός αντικειμένου, ο μεταγλωττιστής ορίζει αυτόματα στο τμήμα public την ειδική συνάρτηση `inline ~X(){}`. Είτε γραφεί από τον προγραμματιστή είτε από τον μεταγλωττιστή, η συγκεκριμένη μορφή του destructor

καλεί αυτόματα τους destructors (είτε ρητά ορισμένους από τον προγραμματιστή είτε αυτόματους από τον μεταγλωττιστή) κάθε μέλους της υλοποίησης του αντικειμένου, με *αντίστροφη* σειρά από αυτή που ορίζονται στην κλάση.

Ο ρητός προσδιορισμός του destructor είναι απαραίτητος όταν υπάρχει ένα τουλάχιστον μέλος της κλάσης X που είναι δείκτης, και η περιοχή μνήμης ή η ποσότητα στην οποία δείχνει χρειάζεται «ειδική« καταστροφή. Στην αντίθετη περίπτωση ο αυτόματος ορισμός από το μεταγλωττιστή είναι ικανοποιητικός. Μπορούμε να τον προκαλέσουμε με την εντολή

```
~X() = default;
```

Αν υπάρχει λόγος, μπορούμε να εμποδίσουμε το μεταγλωττιστή να παραγάγει τον προκαθορισμένο destructor μιας κλάσης X γράφοντας

```
~X() = delete;
```

14.7 Τελεστής εκχώρησης (assignment operator)

Εκτός από τους τρόπους δημιουργίας, αντιγραφής και καταστροφής ενός αντικειμένου, μπορούμε να ορίσουμε και το πώς συμπεριφέρεται σε εκφράσεις που περιλαμβάνουν τελεστές (‘=’, ‘+’, ‘<’, κλπ.). Ο τελεστής = που συμβολίζει την αντιγραφή ενός αντικειμένου σε άλλο έχει ξεχωριστή σημασία και χρειάζεται σε οποιαδήποτε κλάση. Καλείται αυτόματα όποτε χρειαστεί η εκχώρηση ενός αντικειμένου σε άλλο, *προϋπάρχον*, ίδιου τύπου. Π.χ.

```
Student s{"George␣Thomson", 123, 2000};
Student t;
t = s; // operator=() is called.
```

Η εντολή `t=s;` ισοδυναμεί με την `t.operator=(s);`.

Η συνάρτηση–μέλος

```
Student & operator=(Student const & other);
```

είναι αυτή που προσδιορίζει τι ακριβώς γίνεται στην εκχώρηση. Αν δεν την ορίσουμε ρητά, παράγεται αυτόματα από το μεταγλωττιστή. Ο αυτόματος ορισμός της αντιγράφει τα μέλη του δεξιού μέρους της εκχώρησης, στο παράδειγμα s, στα αντίστοιχα μέλη του αριστερού, t. Όταν αυτό δεν είναι επιθυμητό ή δεν έχει το νόημα που θέλουμε, πρέπει να ορίσουμε τη συγκεκριμένη συνάρτηση. Προσέξτε όμως, ότι κατά την κλήση αυτής της συνάρτησης–μέλους, το αντικείμενο για το οποίο καλείται (και το οποίο θα τροποποιηθεί) υπάρχει ήδη και έχει τιμές στα μέλη του. Η συνάρτηση πρέπει να καταστρέφει κατάλληλα τις προϋπάρχουσες ποσότητες και να τις ξαναδημιουργεί ως αντίγραφα των αντίστοιχων μελών του ορίσματός της. Σε αυτή τη διαδικασία, ιδιαίτερη προσοχή πρέπει δείξουμε ώστε η επιτρεπτή εντολή `t=t;` να μας αφήνει το αντικείμενο t αναλλοίωτο. Έτσι, η συνάρτηση πρέπει γενικά να μεριμνά για να αποφύγει καταστροφή του αντικειμένου κατά την «αυτοεκχώρηση». Στο συγκεκριμένο παράδειγμα δεν υπάρχει τέτοιος κίνδυνος αλλά ας

δούμε τι πρέπει να κάνουμε γενικά. Βάσει των παραπάνω, ένας ορισμός που θα μπορούσαμε να γράψουμε για τη συνάρτηση `operator=()` του Student είναι

```
Student &
operator=(Student const & other)
{
  if (this != &other) {
     name = other.name;
     AM = other.AM;
     Year = other.Year;
  }
  return *this;
}
```

Προσέξτε τη χρήση του δείκτη `this` ο οποίος δείχνει στο αντικείμενο για το οποίο καλείται η συνάρτηση `operator=()`. Η συνθήκη μέσα στο `if` ελέγχει αν το αντικείμενο other έχει διαφορετική διεύθυνση μνήμης από αυτό. Αν ναι, εκτελεί τις αντιγραφές. Σε κάθε περίπτωση, επιστρέφει στο τέλος το αντικείμενο για το οποίο κλήθηκε, ώστε να επιτρέπεται η σύνθετη εντολή για αντικείμενα ίδιου τύπου `a=b=c=d`, όπως ισχύει για μεταβλητές θεμελιωδών τύπων.

Γενικά, όποτε χρειαστεί να ορίσουμε μία από τις συναρτήσεις–μέλη: δημιουργίας αντίγραφου, καταστροφέα και τελεστή εκχώρησης, θα πρέπει να ορίσουμε ρητά και τις υπόλοιπες δύο από την τριάδα. Οι αυτόματοι ορισμοί τους που παράγονται από το μεταγλωττιστή πιθανότατα δεν είναι κατάλληλοι.

14.8 Λοιποί τελεστές

Ο τελεστής εκχώρησης (=) που είδαμε παραπάνω, έχει νόημα για οποιαδήποτε κλάση. Γι' αυτό το λόγο, ο ορισμός του υπάρχει πάντα, είτε τον παρέχει ρητά ο προγραμματιστής είτε αυτόματα ο μεταγλωττιστής. Αντίθετα, άλλοι τελεστές θα ορίζονται μόνο αν χρειάζονται. Η συμπεριφορά του τελεστή $\otimes$ για αντικείμενα μιας κλάσης καθορίζεται από τη συνάρτηση–μέλος της κλάσης με όνομα `operator`$\otimes$. Το $\otimes$ μπορεί να είναι ένας από τους τελεστές του Πίνακα 2.3 (και ελάχιστοι ακόμα που δεν αναφέραμε) εκτός από τον τελεστή επιλογής μέλους κλάσης (`.`) και τον τελεστή για τον υπολογισμό του μεγέθους αντικειμένου (`sizeof`).

Παρατηρήστε ότι μπορούμε να παραγάγουμε όλους τους υπόλοιπους τελεστές σύγκρισης του Πίνακα 3.1 αν γνωρίζουμε τη δράση των τελεστών '`<`' και '`==`'. Επομένως, σε μία οποιαδήποτε κλάση, οι συναρτήσεις–μέλη `operator<()` και `operator==()` αρκούν για να ορίσουμε, με συγκεκριμένο τρόπο, τις υπόλοιπες των τελεστών σύγκρισης. Μπορούμε να αποφύγουμε να προσδιορίσουμε ρητά τις σχετικές συναρτήσεις–μέλη, αν συμπεριλάβουμε στον κωδικά μας τον header `<utility>` και χρησιμοποιήσουμε την εντολή

```
using namespace std::rel_ops;
```

Με αυτό τον τρόπο εισάγονται στον κώδικά μας οι συναρτήσεις σύγκρισης που είναι προκαθορισμένες από τη Standard Library και ανήκουν στο χώρο ονομάτων `std::rel_ops`.

Με βάση τα παραπάνω, μπορούμε να ορίσουμε πώς συγκρίνεται ένα αντικείμενο της κλάσης Student με άλλο, συγκρίνοντας π.χ. τα μέλη AM. Περιλαμβάνουμε στον ορισμό της σχετικής κλάσης τα παρακάτω:

```
bool
operator<(Student const & other)
{
  return AM < other.AM;
}
bool
operator==(Student const & other)
{
  return AM == other.AM;
}
using namespace std::rel_ops;
```

14.9 Υπόδειγμα κλάσης

Μια κλάση μπορεί να παραμετροποιείται με ένα ή περισσότερους *τύπους* ποσοτήτων ή ακέραιες ποσότητες. Έτσι, αν επιθυμούμε να έχουμε στην κλάση X ως παράμετρο ένα απροσδιόριστο τύπο T, συμπληρώνουμε τον ορισμό της με το `template<typename T>`:

```
template<typename T>
class X {
  T a;
...
};
```

Στο «εσωτερικό» της κλάσης (στα μέλη της, είτε ποσότητες είτε συναρτήσεις) το T συμβολίζει ένα τύπο που θα προσδιοριστεί κατά τη μεταγλώττιση. Στο συγκεκριμένο παράδειγμα, το μέλος a θα είναι ποσότητα τύπου T. Όταν θελήσουμε να δηλώσουμε ένα αντικείμενο της κλάσης X πρέπει να προσδιορίσουμε τον τύπο T όπως παρακάτω:

```
X<double> x1;
X<int>    x2;
```

Στο παράδειγμα, το `x1.a` είναι πραγματική ποσότητα ενώ το `x2.a` είναι ακέραια.

Μπορούμε να έχουμε ως παράμετρο του template μία ακέραια ποσότητα. Έτσι π.χ., η διάσταση ενός πίνακα, μέλους της κλάσης Y μπορεί να δοθεί κατά τη μεταγλώττιση, αρκεί ο ορισμός της κλάσης να είναι όπως στο ακόλουθο παράδειγμα

```
template<int N>
class Y {
  std::array<double,N> v;
};
```

Όταν χρησιμοποιήσουμε την κλάση Y πρέπει να προσδιορίσουμε την παράμετρο του template:

```
Y<10> c;
```

Το `c.v`, μετά τον ορισμό αυτόν, έχει 10 πραγματικά στοιχεία.

Αν επιθυμούμε να ορίσουμε μια συνάρτηση–μέλος της κλάσης X έξω από το σώμα της κλάσης, πρέπει να συμπληρώσουμε τον ορισμό της στην αρχή του με το `template<typename T>` ενώ στο όνομα της κλάσης που συμπληρώνει το όνομα της συνάρτησης–μέλους πρέπει να προσθέσουμε το `<T>`:

```
template<typename T>
class X {
  T a;
  ...
  void f(T v);
};

template<typename T>
void
X<T>::f(T v)
{
  ...
}
```

14.10 Ασκήσεις

1. Συμπληρώστε τον τύπο Book που αναφέρθηκε παραπάνω.
2. Υλοποιήστε ένα τύπο με όνομα array ώστε να αναπαράγει τον container `std::array<>` του header `<array>`.
3. Δημιουργήστε ένα κατάλληλο τύπο οποίος να αναπαριστά την έννοια «ημερομηνία». Το όνομά του ας είναι date και το interface του θα βρίσκεται στο αρχείο *date.h*. Επιθυμούμε να τον χρησιμοποιήσουμε στον παρακάτω κώδικα:

```
#include <iostream>
#include "date.h"

void
cmp(date const & a, date const & b)
{
    if (a > b) {
        std::cout << a << "␣is␣after␣" << b << '\n';
    }
    if (a == b) {
        std::cout << a << "␣coincides␣with␣" << b <<'\n';
    }

    if (a < b) {
        std::cout << a << "␣is␣before␣" << b << '\n';
    }
}

int
main()
{
    date a; // a = 1/1/1970
    std::cout << a << '\n';

    date b{14,5,2005};   // b = 14/5/2005
    std::cout << b << '\n';

    date c{b};           // c = b  (14/5/2005)
    std::cout << c << '\n';

    date d{c+3};  // d = 17/5/2005
    std::cout << d << '\n';

    date e{d+38}; // e = 24/6/2005
```

```
    std::cout << e << '\n';

    date f{d-1000};   // f = 21/8/2002
    std::cout << "␣the␣day␣is␣"   << f.day()
              << "␣the␣month␣is␣" << f.month()
              << "␣the␣year␣is␣"  << f.year() << '\n';

    int k{b-f};  // k = 997
    std::cout << k << '\n';

    --a; //  31/12/1969
    std::cout << a << '\n';
    ++f; // 22/8/2002
    std::cout << f << '\n';

    cmp(a,b);
    cmp(d,e);
    cmp(c,b);

    d += 4;  // d = 21/5/2005
    std::cout << d << '\n';

    e -= 3000;  // e = 7/4/1997
    std::cout << e << '\n';

    f = d;   // f = 21/5/2005
    std::cout << f << '\n';
}
```

Θα σας χρειαστεί ο αλγόριθμος της άσκησης 8 στη σελίδα 68.

4. Δημιουργήστε ένα κατάλληλο τύπο οποίος να αναπαριστά το διάνυσμα στις τρεις διαστάσεις που γνωρίζουμε από τα μαθηματικά. Να ορίσετε κατάλληλες συναρτήσεις–μέλη που να το δημιουργούν, να το αντιγράφουν, να υπολογίζουν το μέτρο του, το εσωτερικό και το εξωτερικό γινόμενο διανυσμάτων, την πρόσθεση και την αφαίρεση διανυσμάτων, τη γωνία δύο διανυσμάτων και γενικά να υλοποιούν όλες τις ιδιότητες διανυσμάτων που γνωρίζετε.

5. Χωρίς να χρησιμοποιήσετε την κλάση `std::complex<>` της Standard Library, δημιουργήστε μια δική σας κλάση που να αντιπροσωπεύει τους μιγαδικούς αριθμούς με όλες τις ιδιότητες που έχουν. Επομένως, συμπληρώστε τον κώδικα στην κλάση

```
template<typename T>
complex<T> {
```

```
}
```

6. Δημιουργήστε την κλάση sudoku στα αρχεία *sudoku.cpp* και *sudoku.h* ώστε ο παρακάτω κώδικας

```
#include <iostream>
#include "sudoku.h"

int
main(int argc, char *argv[])
{
   sudoku s{argv[1]};
   auto res = s.solve();
   if (res != 0) {
      std::cerr << "Den␣lynetai\n";
      return -1;
   }
   s.print(argv[2]);
}
```

να διαβάζει ένα ημιτελές sudoku από το αρχείο με όνομα αποθηκευμένο στο `argv[1]`, να το επιλύει και να το τυπώνει στο αρχείο με όνομα αποθηκευμένο στο `argv[2]`. Ακολουθήστε τον αλγόριθμο της άσκησης 47, στη σελίδα 162.

Μέρος IV

Παραρτήματα

Κεφάλαιο Α΄

Παραδείγματα προς ..αποφυγή!

Ας παραθέσουμε απλώς, χωρίς σχόλια, λίγα παραδείγματα κώδικα σωστής C που δείχνουν την κακομεταχείριση των κανόνων και των δυνατοτήτων της γλώσσας. Η C++ είναι πιο αυστηρή και δεν επιτρέπει πολλές από αυτές τις ακρότητες. Περισσότερα μπορείτε να βρείτε στη διεύθυνση του σχετικού διεθνούς διαγωνισμού[1]. Εδώ παρουσιάζονται οι συμμετοχές

- του Raymond Cheong το 2001, Κώδικας Α΄.1. Υπολογίζει το ακέραιο μέρος της τετραγωνικής ρίζας του ορίσματός του (ακέραιος με άρτιο αριθμό ψηφίων).

```
#include <stdio.h>
int l;int main(int o,char **O,
int I){char c,*D=O[1];if(o>0){
for(l=0;D[l              ];D[l
++]-=10){D   [l++]-=120;D[l]-=
110;while   (!main(0,O,l))D[l]
+=   20;   putchar((D[l]+1032)
/20   )   ;}putchar(10);}else{
c=o+     (D[I]+82)%10-(I>l/2)*
(D[I-l+I]+72)/10-9;D[I]+=I<0?0
:!(o=main(c/10,O,I-1))*((c+999
)%10-(D[I]+92)%10);}return o;}
```

Κώδικας Α΄.1: cheong.c

[1] http://www.ioccc.org

- του Michael Savastio το 1995, Κώδικας Α΄.2. Υπολογίζει *ακριβώς* το παραγοντικό ακεραίων μέχρι το 429539.

```
#include <stdio.h>

#define l11l 0xFFFF
#define ll1 for
#define ll111 if
#define l1l1 unsigned
#define l111 struct
#define llll1 short
#define ll11l long
#define ll1ll putchar
#define l1ll1(l) l=malloc(sizeof(l111 llll1));l->llll1=1-1;l->ll1l1=1-1;
#define l1ll1 *llll1++=l1ll%10000;l1ll/=10000;
#define l1lll ll111(!l1->llll1){l1ll1(l1->llll1);l1->llll1->ll1l1=l1;}\
llll1=(l1=l1->llll1)->lll;ll=1-1;
#define llll 1000



                                                l111 llll1 {
                                                l111 llll1 *
llll1,*ll1l1        ;l1l1                       llll1 lll [
llll];};main      (){l111 llll1                 *l1l1,*l1l,*
l1, *ll1l, *    malloc ( ) ; l1l1               ll11l l1ll ;
ll11l l11,ll  ,l;l1l1 llll1 *llll,*             llll1; ll1(l
=1-1 ;l< 14; ll1ll("\t\"8)>l\"9!.)>vl"          [l]^'L'),++l
);scanf("%d",&l);l1l1l(l1l) l1l1l(l111          ) (l1=l11)->
lll[l1l->lll[1-1]     =1]=l111;ll1(l11          =1+1;l11<=l;
++l11){l1=ll11;         llll = (llll=(          ll11=l11))->
lll; llll1 =(             l1l=l1)->lll;         ll=(l1ll=1-1
);ll1(;ll1l->              llll1||l111!=        *llll;){llll
+=l11**lll1++              ;l1ll1 ll111         (++ll>llll){
l1lll lll1=(               ll1l =llll->         llll1)->lll;
}}lll(;l1ll;               ){l1ll1 ll111        (++ll>=llll)
{ l1lll} } *               llll1=l111;}
ll1(l=(ll=1-               1);(l<llll)&&
(l1->lll[ l]               !=l111);++l);        ll1 (;l1;l1=
l1->ll1l1,l=               llll){lll(--l        ;l>=1-1;--l,
++ll)printf(               (ll)?((ll%19)        ?"%04d":(ll=
19,"\n%04d")               ):"%4d",l1->         lll[l] ) ; }
                                                l1l1l(10); }
```

Κώδικας Α΄.2: savastio.c

- του Ken Huffman το 1996, Κώδικας Α΄.3 Μετατρέπει ένα κείμενο σε κώδικα Braille και αντίστροφα.

```
#define x char
#define z else
#define w gets
#define r if
#define u int
#define s main
#define v putchar
#define y while
#define t "␣A?B?K?L?CIF?MSP?E?H?O?R?DJG?NTQ?????U?V?????X????????Z????W??Y??"
 s (   )  {   x* c  ,  b[ 5  * 72   ]; u  a, e  ,  d   [  9
*9 *9 ]  ;    y  (w  ( b) ){           r  ( 0   [ b] -7 *
5        )    {  c  =     b    ;       y  (  (*    c  -  6
* 7 )* *  c )c =  c+ 1  ;   r ((   -0 )  [ c  ] &&  w (  b   +
  8 *  5*    3         ) && w          (b   +8      *5  * 6) )
  { a       =  0; y     (                a       [     b  ]
 ) {    a [  d  ] = ! !    ( a[ b]   -   4 *8 )    ;  a  =   a +
1;  }   y (a   <  8 *  5      *  3   )  d      [   (   a ++ )
]  =      0       ; a        =       0;            y(    b  [
 a +  8  *3   *  5  ] )  {   d  [ a  ]   = a  [    d] +  !  !( b[
a+ 40  *  3     ]  -4  * 8) *  2    ;      ++  a   ;   }     a  =
 0       ;      y  (     a     [  b +      6          *  40 ]
)  {a [d ]  += !   !  ( b[ a   +    5 *  6  *    8  ] - 4   *
 8        ) *  4;   a =   a  + 1    ;} a=   0;   y  (a   <3 *8
   *5       )  {    r    (     a    [       d    ]  )    {  e
 =  1  ; y    (   e[ a  +  d] ){    *  ( d+ a+    e )   = a[ d
+e -  1] +(       d   [     a  +   e  ]  <<  (   3* e) )     ;e
 =    e           +  1  ;} a       =        a     +
e -  1     ; v( !! (  *  (  d+
  a)  %   (  64  )  - 12  *  5
  )            +  (  e
>  4   ) ?t [   e   >2 ?  2  :a    [ d   ]  ]   :6 *  8  +( t  [ d[
 a    ]/ 8     /8   ]  -4  )      %( 10 )  )    ;  r(  ! (2 [a
+  d] +  3  [d +       a  ]  )    )        v    (  4  *     8    )
 ;  }    a =   a  + 1 ; } v    (5 *  2     )  ;  }    z {c =b ;   e
=  0    ;     y  (*   c ) {   *   c +=   -  (* c    >   8 *     12
   )    *  32    ;      a         =  8       *  8       ;  r  (* c
 >=  48  &&*  c<8  *8   -6)  {*c  =(   * c  + 1
 )   %(  5    * 2  ) +  65   ;y(  --a  >0   &&*
c-  a[  t]   );   d[   (e   ++   )]   =4   ;(
 * (  d +  (e   ++)  ))=  07   ;}z  y(a  --  >1
 &&*  c-   t[a  ]);  d[   ( e  = e  +    1)  -
1]   =a   %8   ;y   (a   /=   8)   d[   (e  ++
 ) ]  =    a% 8   ; ++ c   ; *  (   e ++   +d )  =0
;} --  e   ;r    (         e >0  ) {a  =   1   ;
y          (  a<       8     )  {   c =       b  ;y
 ( c  <e +b  )   {  v  (  *     ( c  -    b + d  )&   a? 6  *7
:8       *  4     ) ;   c ++   ;} a=  a   +    a  ;    v  ( 2*
   5) ;  }  v    (  5*    2    )          ;   }} }}      }
```

Κώδικας Α΄.3: huffman.c

Κεφάλαιο Β΄

Αναζήτηση–Ταξινόμηση

Β΄.1 Αναζήτηση στοιχείου

Β΄.1.1 Γραμμική αναζήτηση

Η αναζήτηση συγκεκριμένης τιμής σε ένα γενικό διάνυσμα απαιτεί να γίνεται σύγκρισή της με κάθε στοιχείο του διανύσματος. Κατά μέσο όρο χρειάζονται $(1 + N)/2$ συγκρίσεις, όπου N το πλήθος των στοιχείων του διανύσματος.

Β΄.1.2 Δυαδική αναζήτηση

Για ένα *ταξινομημένο* διάνυσμα με N στοιχεία, το πλήθος των απαιτούμενων συγκρίσεων στην αναζήτηση μπορεί να γίνει $1+\log_2 N$, πολύ μικρότερο (για μεγάλα N) από όσο είναι στο γραμμικό αλγόριθμο, αν εφαρμόσουμε τη δυαδική αναζήτηση. Βέβαια, στις απαιτούμενες πράξεις δεν υπολογίζουμε αυτές που χρειάζονται για να ταξινομηθεί το διάνυσμα.

Ο συγκεκριμένος αλγόριθμος σε ένα ταξινομημένο, με αύξουσα σειρά, διάνυσμα έχει ως εξής:

Συγκρίνουμε το μεσαίο στοιχείο του διανύσματος (ή ένα από τα δύο πλησιέστερα στη μέση αν το διάνυσμα έχει άρτιο πλήθος στοιχείων) με τη ζητούμενη τιμή:

- Αν είναι ίσα, έχουμε βρει το ζητούμενο και επιστρέφουμε τη θέση στην οποία το βρήκαμε.
- Αν η ζητούμενη τιμή είναι μικρότερη, σημαίνει ότι, αν υπάρχει, βρίσκεται στο πρώτο μισό του διανύσματος.
- Αν η ζητούμενη τιμή είναι μεγαλύτερη, σημαίνει ότι, αν υπάρχει, βρίσκεται στο δεύτερο μισό του διανύσματος.

Επομένως, με την πρώτη σύγκριση, αν δεν βρήκαμε την ζητούμενη τιμή, περιορίζουμε στο μισό τον χώρο αναζήτησης. Επαναλαμβάνουμε τη διαδικασία για το τμήμα του διανύσματος που επιλέξαμε στο προηγούμενο βήμα έως ότου, με διαδοχικές διχοτομήσεις, περιοριστούμε σε ένα στοιχείο. Τότε, αν είναι ίσο με τη ζητούμενη τιμή επιστρέφουμε τη θέση του, αλλιώς η ζητούμενη τιμή δεν περιέχεται στο διάνυσμα.

Μπορούμε να υλοποιήσουμε τη συγκεκριμένη μέθοδο αναζήτησης και με συνάρτηση που καλεί τον εαυτό της:

- αν το πλήθος των στοιχείων είναι 0, το ζητούμενο στοιχείο δεν υπάρχει.
- αν το πλήθος των στοιχείων είναι 1, συγκρίνουμε το μοναδικό στοιχείο με το ζητούμενο. Αν είναι ίσα επιστρέφουμε τη θέση του, αλλιώς το ζητούμενο στοιχείο δεν υπάρχει.
- αν το πλήθος των στοιχείων είναι μεγαλύτερο από 1, συγκρίνουμε το μεσαίο στοιχείο (ή ένα από τα δύο στοιχεία που είναι πλησιέστερα στο μέσο του διανύσματος) με το ζητούμενο. Αν το ζητούμενο στοιχείο είναι μικρότερο, το αναζητούμε στο ίδιο διάνυσμα, στο «πρώτο μισό». Αλλιώς, το αναζητούμε στο «δεύτερο μισό».

Β΄.1.3 Αναζήτηση με hash

Τα στοιχεία στα οποία επιθυμούμε να αναζητήσουμε μια συγκεκριμένη τιμή, μπορεί να είναι οργανωμένα σε *πίνακα κατακερματισμού* (hash table), δηλαδή σε πολλές ομάδες λίγων στοιχείων με ίδια χαρακτηριστική τιμή. Η αναζήτηση σε τέτοιο πίνακα γίνεται σε δύο στάδια: Πρώτα υπολογίζουμε τη χαρακτηριστική τιμή της αναζητούμενης ποσότητας και κατόπιν συγκρίνουμε την ποσότητα μόνο με τα στοιχεία που έχουν την ίδια χαρακτηριστική τιμή. Η αναζήτηση με αυτό τον τρόπο χρειάζεται πράξεις ανεξάρτητες από το πλήθος των αποθηκευμένων τιμών, και υπό κατάλληλες συνθήκες, είναι πολύ πιο γρήγορη από την δυαδική αναζήτηση.

Για την κατάλληλη οργάνωση των στοιχείων ώστε να είναι εφικτή τέτοια αναζήτηση, είναι βασική η *συνάρτηση κατακερματισμού*, (hash function). Τέτοια συνάρτηση αντιστοιχεί καθένα από τα στοιχεία σε μία χαρακτηριστική τιμή (hash). Το hash μπορεί να πάρει τιμή σε ένα πεπερασμένο σύνολο. Έτσι, π.χ.

- Οι λέξεις ενός λεξικού ή τα λήμματα μιας εγκυκλοπαίδειας μπορούν να οργανωθούν σε διαφορετικούς τόμους (ομάδες), ανάλογα με το αρχικό γράμμα τους.
- ένας οποιοσδήποτε απρόσημος ακέραιος μήκους $16\,\mathrm{bit}$, δηλαδή στο διάστημα $[0, 65535]$, μπορεί να αντιστοιχηθεί στους ακέραιους στο διάστημα $[0, 255]$, υπολογίζοντας το υπόλοιπο της διαίρεσής του με το 256. Η συνάρτηση hash σε αυτή την περίπτωση είναι η $h(k) = k \bmod 256$.
- Ένα κείμενο, δηλαδή μια σειρά χαρακτήρων οποιουδήποτε μήκους, μπορεί να αντιστοιχηθεί σε ακέραιο στο $[0, 255]$ αν κάθε χαρακτήρας του συνδυαστεί (π.χ. με XOR, §2.11.2) σε ένα byte.

Σε μία συνάρτηση hash, γενικά, το πεδίο ορισμού της έχει πολύ περισσότερες τιμές από τις δυνατές τιμές του hash, δηλαδή του αποτελέσματος. Αυτό έχει ως συνέπεια να υπάρχει πιθανότητα περισσότερες από μία τιμές να έχουν το ίδιο hash και επομένως να κατανέμονται στην ίδια ομάδα στοιχείων· σε τέτοια περίπτωση λέμε ότι έχουμε *σύγκρουση* (collision). Μια καλή συνάρτηση hash (και συνεπώς, καλή επιλογή του πλήθους των ομάδων) μπορεί να οργανώσει ένα πλήθος στοιχείων σε ομάδες με ελάχιστες συγκρούσεις, οπότε η αναζήτηση είναι ταχύτατη. Προφανώς, δεν λαμβάνουμε υπόψη το χρόνο οργάνωσης των στοιχείων σε πίνακα κατακερματισμού και θεωρούμε ότι ο υπολογισμός του hash είναι γρήγορος.

Β΄.2 Ταξινόμηση στοιχείων

Β΄.2.1 Bubble sort

Ένας αλγόριθμος ταξινόμησης είναι ο bubble sort (αλγόριθμος φυσαλλίδας). Είναι πολύ απλός αλλά πολύ αργός καθώς το πλήθος των απαιτούμενων πράξεων για να γίνει η ταξινόμηση με αυτόν είναι ανάλογο του τετραγώνου του πλήθους των στοιχείων, N, προς ταξινόμηση. Γι᾽ αυτό χαρακτηρίζεται ως τάξης $O(N^2)$.

Έστω ότι επιθυμούμε να ταξινομήσουμε N αντικείμενα από το μικρότερο στο μεγαλύτερο. Ο αλγόριθμος έχει ως εξής:

1. Συγκρίνουμε το πρώτο με το δεύτερο στοιχείο. Αν χρειάζεται, τα εναλλάσσουμε ώστε το μικρότερο να είναι πρώτο. Επαναλαμβάνουμε διαδοχικά τη σύγκριση και πιθανή εναλλαγή για τα ζεύγη στοιχείων $(2, 3)$, $(3, 4)$, ..., $(N-1, N)$. Στο τέλος της συγκεκριμένης διαδικασίας, το μεγαλύτερο στοιχείο από όλα θα βρεθεί στη θέση N, δηλαδή στη σωστή του θέση.

2. Επαναλαμβάνουμε το προηγούμενο βήμα, διαδοχικά: μία φορά μέχρι το ζεύγος $(N-2, N-1)$, την επόμενη μέχρι το $(N-2, N-1)$, ..., την τελευταία φορά μέχρι το ζεύγος $(1, 2)$. Σε κάθε βήμα, το στοιχείο με το κατάλληλο μέγεθος έρχεται στις θέσεις $N-1$, $N-2$, ..., 1.

Με το τέλος του αλγορίθμου η λίστα των στοιχείων είναι ταξινομημένη από το μικρότερο προς το μεγαλύτερο.

Όπως θα καταλάβετε από τις συγκρίσεις, είναι εξαιρετικά αργός αλγόριθμος και δε θα πρέπει να τον χρησιμοποιείτε για οτιδήποτε σοβαρό!

Β΄.2.2 Insertion sort

Είναι παρόμοιος αλγόριθμος με τον αλγόριθμο bubble sort. Είναι τάξης $O(N^2)$ έως (στην καλύτερη περίπτωση της ήδη ταξινομημένης λίστας) $O(N)$. Έχει ως εξής:

1. Ξεκινώντας από το δεύτερο στοιχείο της λίστας, το «ταξινομούμε» σε σχέση με το πρώτο.

2. Επιλέγουμε διαδοχικά το τρίτο, τέταρτο,... στοιχείο και το τοποθετούμε στη σωστή σειρά σε σχέση με τα προηγούμενα (που έχουν ήδη ταξινομηθεί), κάνοντας και όποιες μετακινήσεις στοιχείων είναι απαραίτητες.

Β΄.2.3 Quick sort

Ο αλγόριθμος quick sort είναι (υπό προϋποθέσεις) από τους πιο γρήγορους αλγόριθμους ταξινόμησης. Είναι τάξης $O(N^2)$ (στη χειρότερη περίπτωση της ήδη ταξινομημένης λίστας) έως (συνήθως) $O(N \log N)$, όπου N το πλήθος των στοιχείων.

Έστω ότι έχουμε λίστα στοιχείων που επιθυμούμε να τα ταξινομήσουμε από το μικρότερο στο μεγαλύτερο. Ο αλγόριθμος quick sort υλοποιείται πολύ εύκολα με αναδρομική συνάρτηση.

Quick sort με βοηθητική λίστα Η απλή εκδοχή του αλγορίθμου, με βοηθητική λίστα, είναι η ακόλουθη:

1. Αν η λίστα στοιχείων δεν έχει κανένα ή έχει μόνο ένα στοιχείο, επιστρέφουμε καθώς δεν χρειάζεται ταξινόμηση.

2. Δημιουργούμε μία νέα λίστα με ίδια διάσταση με την αρχική.

3. Επιλέγουμε ένα οποιοδήποτε στοιχείο της αρχικής λίστας (π.χ. το πρώτο) και διατρέχουμε όλα τα υπόλοιπα στοιχεία της.

4. Αντιγράφουμε (ή καλύτερα, μετακινούμε) σε διαδοχικές θέσεις *από την αρχή* της νέας λίστας τα στοιχεία της αρχικής που είναι *μικρότερα* από το επιλεγμένο στοιχείο.

5. Αντιγράφουμε (ή καλύτερα, μετακινούμε) σε διαδοχικές θέσεις *από το τέλος* της νέας λίστας τα στοιχεία της αρχικής που είναι *μεγαλύτερα ή ίσα* με το επιλεγμένο στοιχείο.

6. Αντιγράφουμε (ή καλύτερα, μετακινούμε) το επιλεγμένο στοιχείο στη μοναδική κενή θέση της νέας λίστας, μεταξύ των στοιχείων της αρχής (τα «μικρότερα») και των στοιχείων του τέλους (τα «μεγαλύτερα» ή ίσα). Αυτή είναι και η τελική θέση του συγκεκριμένου στοιχείου στην ταξινομημένη λίστα.

7. Αντιγράφουμε (ή καλύτερα, μετακινούμε) τη νέα λίστα στην αρχική.

8. Επαναλαμβάνουμε όλη τη διαδικασία για τις υπο-λίστες πριν και μετά το επιλεγμένο στοιχείο, χωρίς να το περιλαμβάνουμε.

Quick sort χωρίς βοηθητική λίστα Αν δεν επιθυμούμε, για λόγους ταχύτητας ή περιορισμένης μνήμης, να χρησιμοποιήσουμε βοηθητική λίστα, μπορούμε να ακολουθήσουμε τον παρακάτω αλγόριθμο:

1. Αν η λίστα στοιχείων δεν έχει κανένα ή έχει μόνο ένα στοιχείο, επιστρέφουμε καθώς δεν χρειάζεται ταξινόμηση.

2. Επιλέγουμε ένα οποιοδήποτε στοιχείο της αρχικής λίστας (π.χ. το πρώτο) και διατρέχουμε όλα τα υπόλοιπα στοιχεία της.

3. Επιδιώκουμε να μεταφέρουμε στην αρχή της λίστας τα στοιχεία που είναι μικρότερα από το επιλεγμένο και στο τέλος της λίστας όσα είναι ίσα ή μεγαλύτερα από το επιλεγμένο.

   Επομένως, διατρέχουμε τη λίστα με δύο δείκτες· ο ένας ξεκινά από την αρχή και ο άλλος από το τέλος. Ο πρώτος θα αυξάνει όσο δείχνει σε στοιχεία μικρότερα από το επιλεγμένο ενώ ο δεύτερος θα μειώνεται όσο δείχνει σε στοιχεία μεγαλύτερα ή ίσα με το επιλεγμένο. Αν κάποιος δείκτης συναντήσει τη θέση του επιλεγμένου στοιχείου θα μετακινείται στην επόμενη θέση.

4. Όσο ο πρώτος δείκτης δεν έχει ξεπεράσει τον δεύτερο, εναλλάσσουμε τα στοιχεία στα οποία δείχνουν οι δύο δείκτες ώστε ο πρώτος να δείχνει σε στοιχεία μικρότερα του επιλεγμένου και ο δεύτερος σε μεγαλύτερα ή ίσα. Μετά από κάθε σύγκριση μετακινούμε τους δείκτες κατά μία θέση.

5. Όταν ο πρώτος δείκτης ξεπεράσει τον δεύτερο, έχουμε βρει τη θέση που πρέπει να μεταφερθεί το επιλεγμένο στοιχείο (είναι η θέση του πλησιέστερου δείκτη στην επιλεγμένη θέση). Εναλλάσσουμε το επιλεγμένο στοιχείο με αυτό που δείχνει ο πλησιέστερος δείκτης.

6. Εφαρμόζουμε την ίδια διαδικασία στις υπο-λίστες πριν και μετά το επιλεγμένο στοιχείο (στη νέα του θέση), χωρίς να το περιλαμβάνουμε.

Β΄.2.4 Merge sort

Είναι από τους πιο γρήγορους αλγόριθμους ταξινόμησης, με πλήθος πράξεων ανάλογες του $N \log N$, όπου N το πλήθος των στοιχείων. Σύμφωνα με τον αλγόριθμο merge sort:

1. Αν η λίστα στοιχείων δεν έχει κανένα ή έχει μόνο ένα στοιχείο, επιστρέφουμε καθώς δεν χρειάζεται ταξινόμηση.

2. Χωρίζουμε τη λίστα σε δύο περίπου ίσα μέρη.

3. Ταξινομούμε κάθε νέο τμήμα με ξεχωριστή εφαρμογή της τρέχουσας διαδικασίας (επομένως καλούμε τη συνάρτηση που γράφουμε και που υλοποιεί τη merge sort).

4. Συγχωνεύουμε τις δύο ταξινομημένες λίστες με τέτοιο τρόπο ώστε η τελική να είναι επίσης ταξινομημένη.

Β΄.3 Ασκήσεις

1. Να γράψετε δύο συναρτήσεις που να δέχονται από δύο ορίσματα: το πρώτο θα είναι ένα `std::vector<int>` και το δεύτερο ένας ακέραιος (η ζητούμενη τιμή). Να επιστρέφουν σε όρισμα τη θέση του διανύσματος στην οποία βρίσκεται η ζητούμενη τιμή ή -1 αν αυτή δεν βρέθηκε. Η μία συνάρτηση θα εφαρμόζει τη γραμμική αναζήτηση και η άλλη τη δυαδική (το μη αναδρομικό αλγόριθμο).

2. Γράψτε συναρτήση που να υλοποιεί τον αναδρομικό αλγόριθμο (η συνάρτηση καλεί τον εαυτό της) για τη δυαδική αναζήτηση. Θα δέχεται ως ορίσματα ένα ταξινομημένο `std::vector<>` με στοιχεία οποιουδήποτε τύπου, δύο ακέραιους που θα προσδιορίζουν το πρώτο και το τελευταίο στοιχείο του διανύσματος που θα ληφθούν υπόψη στην αναζήτηση, καθώς και τη ζητούμενη τιμή. Η συνάρτηση θα επιστρέφει τη θέση που βρήκε τη ζητούμενη τιμή ή -1 αν δεν τη βρει.

3. Γράψτε κατάλληλη συνάρτηση που να εφαρμόζει τον αλγόριθμο φυσαλλίδας. Αυτή θα δέχεται ως όρισμα ένα `std::vector<>` οποιουδήποτε τύπου και θα το τροποποιεί ώστε να είναι ταξινομημένο από το μικρότερο στο μεγαλύτερο στοιχείο.

   Χρησιμοποιήστε τη συνάρτηση για να ταξινομήσετε τους 1254 ακέραιους αριθμούς που δίνονται σε ξεχωριστή γραμμή ο καθένας στο αρχείο στη διεύθυνση `http://tinyurl.com/114nonrepeat`. Το πρόγραμμά σας να τυπώνει τους αριθμούς αυτούς, ταξινομημένους, στο αρχείο *sorted_data*.

   Υπόδειξη: Να γράψετε και να χρησιμοποιήσετε συνάρτηση για την εναλλαγή τιμών δύο μεταβλητών οποιουδήποτε τύπου.

4. Υλοποιήστε σε αναδρομική συνάρτηση τον αλγόριθμο ταξινόμησης quick sort. Η συνάρτηση θα δέχεται ως όρισμα ένα `std::vector<>` με στοιχεία οποιουδήποτε τύπου, το οποίο θα ταξινομεί από το μικρότερο στο μεγαλύτερο στοιχείο. Χρησιμοποιήστε τη συνάρτησή σας για να ταξινομήσετε τα στοιχεία του αρχείου στη διεύθυνση `http://tinyurl.com/114rndint`. Η πρώτη γραμμή του αρχείου περιέχει το πλήθος των αριθμών που ακολουθούν.

5. Γράψτε συνάρτηση που να συγχωνεύει δύο ήδη ταξινομημένα `std::vector<>` με στοιχεία οποιουδήποτε τύπου σχηματίζοντας άλλο, επίσης ταξινομημένο. Η ταξινόμηση των διανυσμάτων εισόδου θα θεωρήσετε ότι έγινε με αύξουσα σειρά. Τέτοια ταξινόμηση θα πρέπει να έχει και το διάνυσμα εξόδου.

6. Χρησιμοποιήστε τη συνάρτηση που γράψατε στην άσκηση 5 για να υλοποιήσετε τον αλγόριθμο merge sort.

7. Στο αρχείο στη διεύθυνση `http://tinyurl.com/114nonrepeat` περιέχονται 1254 ακέραιοι αριθμοί σε ξεχωριστή γραμμή ο καθένας. Να γράψετε πρόγραμμα που να τους διαβάζει σε διάνυσμα και να χρησιμοποιεί τη γραμμική

αναζήτηση για να εντοπίσει τους αριθμούς 316001 και 499160. Κατόπιν, να ταξινομεί το διάνυσμα είτε με quick sort (άσκηση 4) είτε με merge sort (άσκηση 6) και να επαναλαμβάνει την αναζήτηση των αριθμών με το δυαδικό αλγόριθμο.

Κεφάλαιο Γ΄

Διασύνδεση με κώδικες σε Fortran και C

Η C++ δίνει τη δυνατότητα να ενσωματώσουμε σε πρόγραμμά μας κώδικες γραμμένους σε άλλες γλώσσες προγραμματισμού. Η ακριβής διαδικασία εξαρτάται σε πολύ μεγάλο βαθμό από τους compilers που θα χρησιμοποιηθούν, θα προσπαθήσουμε όμως να περιγράψουμε τη γενική ιδέα. Θα αναφερθούμε στη διασύνδεση με κώδικα σε Fortran 77 και C· παρόμοια ισχύουν και σε όσες γλώσσες προγραμματισμού ακολουθούν τη σύμβαση διασύνδεσης της C. Οι compilers για κώδικες σε Fortran 90/95 *δεν ικανοποιούν αυτό το κριτήριο* και πρέπει να συμβουλευτούμε την τεκμηρίωση που τους συνοδεύει για τον ακριβή μηχανισμό διασύνδεσης. Ακόμα και σε Fortran 77 ο compiler μπορεί να αποκλίνει από όσα θα αναφέρουμε παρακάτω[1].

Στα παρακάτω, εννοείται ότι η βασική συνάρτηση του προγράμματός μας (η `main()`) μεταγλωττίζεται με τον compiler της C++, και, βέβαια, στον κώδικα που προσπαθούμε να συνδέσουμε δεν υπάρχει άλλη `main()` (για κώδικα C) ή `PROGRAM` (για κώδικα Fortran).

Γ΄.1 Κώδικας σε C

Στην περίπτωση που επιθυμούμε να χρησιμοποιήσουμε κώδικα σε C, επιχειρούμε, καταρχάς, να τον μεταγλωττίσουμε με τον compiler της C++. Υπάρχει πιθανότητα να μην χρειάζεται καμμία τροποποίηση καθώς η C, εκτός από ειδικές περιπτώσεις, είναι υποσύνολο της C++. Αν επιτύχουμε, μπορούμε να ακολουθήσουμε το μοντέλο ξεχωριστής μεταγλώττισης που αναφέραμε στο Κεφάλαιο 7.

[1] για παράδειγμα, ο Fortran compiler της NAG όταν το υποπρόγραμμα προς μεταγλώττιση έχει ποσότητα τύπου CHARACTER ως όρισμα.

Σε περίπτωση που πρέπει να χρησιμοποιήσουμε τον compiler της C στον κώδικα που έχουμε ή όταν δεν έχουμε πρόσβαση στον κώδικα αλλά μόνο σε object file ή library (που είναι ήδη μεταγλωττισμένος κώδικας) ακολουθούμε την εξής διαδικασία: απομονώνουμε τις δηλώσεις κάθε συνάρτησης στον «ξένο» κώδικα και τις συμπεριλαμβάνουμε είτε αυτούσιες είτε μέσω αρχείου header στον δικό μας κώδικα. Φροντίζουμε κατόπιν να ενημερώσουμε τον compiler της C++ ότι αυτές οι συναρτήσεις ακολουθούν το πρότυπο διασύνδεσης της C συμπληρώνοντας τις δηλώσεις τους με το `extern "C"`.

Παράδειγμα

Έστω ότι δύο συναρτήσεις f,g

```
int
f(double a)
{
.....
}

void
g(int a)
{
.....
}
```

έχουν μεταγλωττιστεί με compiler της C. Στον κώδικα της C++ συμπεριλαμβάνονται οι δηλώσεις τους με τη μορφή

```
extern "C"
int f(double a);

extern "C"
void g(int a);
```

ή, ισοδύναμα, με

```
extern "C" {
  int f(double a);
  void g(int a);
}
```

Οι παραπάνω δηλώσεις αρκούν για τη μεταγλώττιση του κώδικα C++. Στη φάση του link πρέπει να προσδιορίσουμε κατάλληλα και το αρχείο object ή τη library που περιέχει το μεταγλωτισμένο κώδικά τους. Έτσι π.χ. σε συστήματα UNIX με compilers του GNU Project, αν υποθέσουμε ότι το πρόγραμμά μας περιέχεται στο αρχείο *prog.cpp* και οι «ξένες» συναρτήσεις είναι στο *fg.c*, δίνουμε τις εντολές

```
gcc -c fg.c
g++ prog.cpp fg.o
```

Η πρώτη παράγει ένα αρχείο τύπου object, με κατάληξη .o, το οποίο συνδέεται με τον κώδικα σε C++ με τη δεύτερη εντολή για να παραχθεί εκτελέσιμο αρχείο. Ανάλογα ισχύουν και για άλλους compilers και λειτουργικά συστήματα.

Γ΄.2 Κώδικας σε Fortran

Η συμπερίληψη μεταγλωττισμένου κώδικα Fortran 77 είναι παρόμοια με την συμπερίληψη κώδικα σε C. Το σημείο που πρέπει να προσέξουμε είναι ο σχηματισμός της δήλωσης της συνάρτησης. Ας το δούμε με ένα ρεαλιστικό παράδειγμα:

Δυο συλλογές υποπρογραμμάτων σε Fortran, ελεύθερα διαθέσιμες, είναι η BLAS και η LAPACK. Η πρώτη παρέχει ταχύτατες συναρτήσεις για στοιχειώδη χειρισμό διανυσμάτων ή πινάκων, (πολλαπλασιασμός σταθεράς με διάνυσμα, πρόσθεση πινάκων, κλπ.). Η δεύτερη χρησιμοποιεί αυτές για να υλοποιήσει αλγόριθμους γραμμικής άλγεβρας (επίλυση γραμμικών συστημάτων, εύρεση ιδιοτιμών και ιδιοδιανυσμάτων κλπ.). Είναι ιδιαίτερα χρήσιμες σε υπολογιστικούς κώδικες. Ένα από τα βοηθητικά υποπρογράμματα σε αυτές είναι η `DLASRT()` η οποία ταξινομεί διάνυσμα πραγματικών αριθμών διπλής ακρίβειας. Η περιγραφή της από την τεκμηρίωση[2] της LAPACK είναι η εξής

```
SUBROUTINE DLASRT( ID, N, D, INFO )

    CHARACTER          ID
    INTEGER            INFO, N
    DOUBLE PRECISION D( * )
```

Η επεξήγηση των ορισμάτων από την περιγραφή της είναι

```
ID      (input) CHARACTER*1
        = 'I': sort D in increasing order;
        = 'D': sort D in decreasing order.

N       (input) INTEGER
        The length of the array D.

D       (input/output) DOUBLE PRECISION array, dimension (N)
        On entry, the array to be sorted.
        On exit, D has  been  sorted into increasing order
        (D(1) <= ... <= D(N) ) or into decreasing order
        (D(1) >= ... >= D(N) ), depending on ID.
```

[2] σε συστήματα UNIX η εντολή man dlasrt παρουσιάζει την περιγραφή της.

Πίνακας Γ΄.1: Αντιστοίχιση ενσωματωμένων τύπων της Fortran σε C++

Fortran	C++
`INTEGER`	`int`
`REAL`	`float`
`DOUBLE PRECISION`	`double`
`LOGICAL`	`bool`
`CHARACTER`	`char`
`COMPLEX`	`std::complex<float>`

```
INFO    (output) INTEGER
        = 0:  successful exit
        < 0:  if INFO = -i, the i-th argument had an illegal value
```

Στο σχηματισμό της δήλωσής της για τη C++ ισχύουν οι εξής αντιστοιχήσεις (ή πιο σωστά, ελπίζουμε ότι ισχύουν, καθώς εξαρτώνται από τους compilers):

Παρατηρούμε ότι η `DLASRT()` είναι **`SUBROUTINE`** και επομένως θα δηλωθεί στη C++ ως **`void`**. Σε αρχικό στάδιο η δήλωση είναι

```
void dlasrt(char id, int n, double d[], int info);
```

Το όνομα της συνάρτησης είναι με πεζούς χαρακτήρες. Τα ονόματα των ορισμάτων δεν έχουν σημασία παρά μόνον ο τύπος τους. Προσέξτε ότι αν εμφανιζόταν ως όρισμα διδιάστατος πίνακας, η δήλωση στη C++ θα ήταν ένας *διάνυσμα* σε column-major order (η αποθήκευση γίνεται κατά στήλες). Συμβουλευτείτε τη §5.2.1 για το πώς γίνεται ο ορισμός και η πρόσβαση σε τέτοιο διάνυσμα. Στην περίπτωση που χρησιμοποιήσουμε `std::vector<>` για την υλοποίησή του θα πρέπει να «περάσουμε» στη συνάρτηση *τη διεύθυνση του πρώτου στοιχείου*, δηλαδή, αν `v` είναι ένα `std::vector<>` το όρισμα θα υποκατασταθεί με το `&v[0]` ή το `v.data()`.

Όπως βλέπουμε στην περιγραφή της συνάρτησης τα δύο πρώτα ορίσματα, `id` και `n`, χρησιμοποιούνται μόνο για είσοδο δεδομένων, ενώ τα άλλα δύο τροποποιούνται από τη συνάρτηση. Καλό, αλλά όχι απαραίτητο, είναι τα δύο πρώτα ορίσματα να δηλωθούν ως **`const`**.

Προσέξτε ότι όλα τα ορίσματα σε ένα υποπρόγραμμα της Fortran μπορούν να τροποποιηθούν από αυτό. Αντίστοιχη συμπεριφορά επιτυγχάνεται στη C++ δηλώνοντάς τα ως αναφορές (§2.17) (εκτός από τα ορίσματα που είναι ενσωματωμένα διανύσματα ή άλλοι δείκτες, τα οποία τα αφήνουμε ως έχουν). Με τα παραπάνω, η δήλωση γίνεται σε δεύτερο στάδιο

```
void dlasrt(char const & id, int const & n, double d[], int & info);
```

Αν ο compiler της Fortran ακολουθεί το πρότυπο διασύνδεσης της C, η δήλωση πρέπει να συμπληρωθεί με το **`extern "C"`**. Ο compiler μπορεί να παρέχει υποστήριξη και για άλλους τρόπους διασύνδεσης.

Ειδικά για τους compilers της GNU και όσους ακολουθούν το δικό τους τρόπο διασύνδεσης, το όνομα της συνάρτησης τροποποιείται κατά τη μεταγλώττιση ως

εξής: μετατρέπεται σε πεζά και προσαρτάται στο τέλος του ονόματος ο χαρακτήρας '_', μία φορά αν το όνομα της συνάρτησης δεν τον περιέχει, και δύο αν ήδη υπάρχει σε αυτό. Βάσει των παραπάνω, η δήλωση για αυτή την κατηγορία μεταγλωττιστών γίνεται

```
extern "C"
void dlasrt_(char const & id, int const & n, double d[], int & info);
```

Με την παραπάνω δήλωση ο μεταγλωττιστής έχει όλη την πληροφορία που χρειάζεται για να προχωρήσει. Η σύνδεση των υποπρογραμμάτων (που για τις LAPACK και BLAS έρχονται σε library) γίνεται σε συστήματα UNIX με την ακόλουθη εντολή

```
g++ prog.cpp -llapack -lblas
```

Χωρίς να μπούμε σε λεπτομέρειες, ας αναφέρουμε ότι η Fortran 2003 προσδιορίζει κανόνες για τη διασύνδεση υποπρογραμμάτων της σε κώδικα C (συνεπώς και C++, σύμφωνα με όσα παρουσιάστηκαν στο §Γ΄.1).

Σε κάθε περίπτωση πρέπει να συμβουλευόμαστε την τεκμηρίωση που συνοδεύει τους compilers που θα χρησιμοποιήσουμε.

Βιβλιογραφία

[1] ISO. *ISO/IEC 14882:2014 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, fourth edition, December 2014. URL: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=64029.

[2] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison Wesley, Reading, MA, USA, second edition, March 2012. URL: http://www.cppstdlib.com.

[3] Scott Meyers. *Effective STL: 50 Specific Ways to Improve the Use of the Standard Template Library*. Professional Computing Series. Addison Wesley, Reading, MA, USA, July 2001. URL: http://www.aristeia.com.

[4] Herb Sutter. *More Exceptional C++: 40 More Engineering Puzzles, Programming Problems, and Solutions*. C++ in Depth Series. Addison Wesley, Reading, MA, USA, January 2002.

[5] D.E. Knuth. *The Art of Computer Programming: Combinatorial Algorithms, Part 1*, volume 4A. Addison Wesley, Reading, MA, USA, first edition, 2011.

Κατάλογος πινάκων

Ευρετήριο