Τμημα Επιστημης και Τεχνολογιας Υλικων
Πανεπιστημιο Κρητης

Εισαγωγή στη γλώσσα προγραμματισμού C++

Σταματης Σταματιαδης

Η στοιχειοθεσία έγινε από το συγγραφέα με τη χρήση του LaTeX2ε.
Χρησιμοποιήθηκε η σειρά χαρακτήρων 'Κέρκης' (© Τμήμα Μαθηματικών, Πανεπιστήμιο Αιγαίου).

Τελευταία τροποποίηση του κειμένου έγινε στις 10 Ιουνίου 2015. Η πιο πρόσφατη έκδοση βρίσκεται στο
http://www.materials.uoc.gr/el/undergrad/courses/ETY215

Περιεχόμενα

Πρόλογος

Στις παρούσες σημειώσεις γίνεται μια προσπάθεια να παρουσιαστεί συνοπτικά ένα σημαντικό τμήμα της γλώσσας προγραμματισμού ISO C++, με έμφαση σε ό,τι χρειάζεται για την ανάπτυξη κωδίκων στις φυσικές επιστήμες.

Η γλώσσα C++ είναι, κατά γενική ομολογία, ιδιαίτερα πλούσια στις δυνατότητες έκφρασης που παρέχει στον προγραμματιστή αλλά ταυτόχρονα σημαντικά πιο δύσκολη στην κατανόησή της, σε σχέση με απλές γλώσσες όπως η Fortran και η C. Οι παρούσες σημειώσεις δεν έχουν στόχο να υποκαταστήσουν την ήδη υπάρχουσα σχετική βιβλιογραφία (υπερτερούν ωστόσο έναντι ορισμένων σχετικών εγχειριδίων) ούτε και να καλύψουν τη γλώσσα προγραμματισμού C++ σε όλες τις επιμέρους δυνατότητές της.

Φιλοδοξία μου είναι να αποδοθεί μια όσο το δυνατόν πιο άρτια και πλήρης περιγραφή-περίληψη υποσυνόλου της C++, όπως αυτή διαμορφώθηκε με το Standard του 1998, και ταυτόχρονα να δημιουργηθεί μια στέρεη βάση πάνω στην οποία ο κάθε αναγνώστης, με δική του πλέον πρωτοβουλία, θα μπορέσει να αναπτύξει περαιτέρω τη 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, το κεφάλαιο για την STL προηγείται αυτού για τις κλάσεις, κ.α. Ίσως αυτή η προσέγγιση βοηθήσει στο να ανατραπεί η εικόνα που έχουν πολλοί για τη C++ ως γλώσσα αποκλειστικά για *αντικειμενοστρεφή*[1] προγραμματισμό ("κάτι προχωρημένο που δε μας χρειάζεται").

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

Κεφάλαιο 1

Εισαγωγή

Ένας ηλεκτρονικός υπολογιστής έχει τη δυνατότητα να *προγραμματιστεί* ώστε να εκτελέσει μια συγκεκριμένη διαδικασία. Προγραμματισμός είναι η λεπτομερής περιγραφή, σε κάποια γλώσσα προγραμματισμού, των βημάτων που πρέπει να ακολουθήσει ώστε να ολοκληρώσει την επιθυμητή διεργασία. Το σύνολο των βημάτων, το *πρόγραμμα* δηλαδή, συνήθως απαιτεί *δεδομένα* που πρόκειται να επεξεργαστεί ώστε να παράγει κάποιο αποτέλεσμα· σχεδιάζεται, όμως, ανεξάρτητα από συγκεκριμένες *τιμές* των δεδομένων αυτών.

Οι πιο διαδεδομένες γλώσσες προγραμματισμού στις εφαρμοσμένες επιστήμες (Fortran, C, C++) χρησιμοποιούν *σταθερές* και *μεταβλητές* ποσότητες, δηλαδή, θέσεις στη μνήμη του υπολογιστή, για την αποθήκευση των ποσοτήτων (δεδομένων και αποτελεσμάτων) του προγράμματος. Ο υπολογιστής επιδρά στις τιμές αυτών των ποσοτήτων ακολουθώντας διαδοχικά τις εντολές που περιλαμβάνονται στο πρόγραμμα. Υπάρχει η δυνατότητα *ανάθεσης* τιμής στις μεταβλητές, επιλογής της εντολής που θα εκτελεστεί στο επόμενο βήμα, ανάλογα με κάποια συνθήκη, καθώς και η δυνατότητα επανάληψης μιας ή περισσότερων εντολών. Η βασική δομή και ο τρόπος λειτουργίας ενός προγράμματος στις προαναφερθείσες γλώσσες δε διαφέρει ουσιαστικά από τη μία στην άλλη. Αυτό δεν σημαίνει ότι κάποιες γλώσσες προγραμματισμού δεν είναι πιο εξελιγμένες από άλλες—παρέχουν, δηλαδή, περισσότερες δυνατότητες—ή είναι πιο κατάλληλες για συγκεκριμένες εφαρμογές.

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 έναν πραγματικό αριθμό και να τυπώνει το τετράγωνό του. Ένα πλήρες πρόγραμμα C++ που εκτελεί την παραπάνω

εργασία και μας δίνει την ευκαιρία να δούμε στοιχεία της δομής του κώδικα, είναι το ακόλουθο:

```
#include <iostream>

/*
   main:
         Takes no arguments.
         Prompts for a real number and prints its square.
         Returns 0.
*/
int
main() {
    double a;  // Declare a real variable

    // Print text on screen
    std::cout << "Give␣a␣number:␣";

    std::cin >> a; // Get value from keyboard

    // Print text on screen
    std::cout << "Square␣of␣input␣is␣";

    std::cout << a*a;   // Print result

    std::cout << '\n';  // Change line

    return 0; // Successful exit from program.
}
```

Ας το αναλύσουμε:

Η γλώσσα C++ έχει, σχετικά με άλλες γλώσσες προγραμματισμού, λίγες ενσωματωμένες εντολές. Μία πληθώρα άλλων εντολών και δυνατοτήτων παρέχεται από τη Standard Library (STL) (Κεφάλαιο 5), τμήματα της οποίας μπορούμε να συμπεριλάβουμε με "οδηγίες" (directives) **#include** προς τον προεπεξεργαστή. Η οδηγία στην πρώτη γραμμή του παραδείγματος, **#include** `<iostream>`, 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++.

Κείμενα μεταξύ `/*` και `*/` ή από `//` μέχρι το τέλος της γραμμής είναι σχόλια (§2.1.1).

Η δήλωση **int** `main() {...}` ορίζει τη βασική συνάρτηση (§4.5) σε κάθε πρόγραμμα C++: το όνομά της είναι `main`, επιστρέφει ένα ακέραιο αριθμό (**int**, §2.2.4), 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 το 0).[1] Η επιστροφή της τιμής 0 από τη `main()` υποδηλώνει 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.

Η εντολή `double a;` ορίζει μία πραγματική μεταβλητή διπλής ακρίβειας—δηλαδή έχει (συνήθως) 15 σημαντικά ψηφία σωστά—με το όνομα a (§2.2.5). Στη C++ όλες οι μεταβλητές πρέπει να δηλωθούν, δηλαδή να παρουσιαστεί ο τύπος και το όνομά τους στον compiler, πριν χρησιμοποιηθούν.

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. Ο τελεστής (<<), όταν χρησιμοποιείται για εκτύπωση ("έξοδο") δεδομένων, "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 αποδίδει στην ποσότητα που τον ακολουθεί. Η χρήση των παραπάνω προϋποθέτει, όπως αναφέρθηκε, τη συμπερίληψη του standard header `<iostream>` (§2.9). Παρατηρήστε ότι, σε αντίθεση με παλαιότερες γλώσσες προγραμματισμού, στη C++ δεν καθορίζουμε format για είσοδο/έξοδο δεδομένων. Η πληροφορία για τη διαμόρφωσή τους 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 (§2.9.4).

Ένα σύνολο χαρακτήρων μεταξύ διπλών εισαγωγικών (") αποτελεί μια σταθερά χαρακτήρων, ένα C-style string. Σε μία τέτοια σειρά μπορούν να υπάρχουν ειδικοί χαρακτήρες όπως ο '\n' (που εκφράζει την αλλαγή γραμμής), οι οποίοι δεν εκτυπώνονται αλλά εκτελούν συγκεκριμένες λειτουργίες. Μεταξύ απλών εισαγωγικών (') μπορεί να περιλαμβάνεται ένας μόνο χαρακτήρας (που μπορεί να είναι ειδικός, δηλαδή να εισάγεται με (\))· αυτός αποτελεί μια σταθερή ποσότητα τύπου χαρακτήρα (character literal, §2.2.2).

Ο τελεστής (*) μεταξύ πραγματικών αριθμών αντιπροσωπεύει τη γνωστή πράξη του πολλαπλασιασμού.

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 (γραμμές με πρώτο χαρακτήρα τον '#') δεν έχουν τελικό ';'.

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

Η C++ δεν επιβάλλει 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 μεταβλητών. Οι θέσεις αυτών είναι ελεύθερες (δείτε πόσο ακραίες διαμορφώσεις μπορείτε να συναντήσετε στο Παράρτημα Α'). Κενά και αλλαγές γραμμής δεν μπορούν, όμως, να διαχωρίζουν τα σύμβολα που αποτελούν σύνθετους τελεστές (`+=`, `==`, `<<`, `/*`, `//`,...). Επιπλέον, σταθερές χαρακτήρων, σχόλια που αρχίζουν με `//` και εντολές προς τον προεπεξεργαστή δεν επιτρέπεται να εκτείνονται σε περισσότερες από μία γραμμές. Ειδικά οι τελευταίες πρέπει να βρίσκονται μόνες τους στη γραμμή (ή να ακολουθούνται μόνο από σχόλια) και ο πρώτος μη κενός χαρακτήρας τους να είναι ο '#'.

1.2 Ασκήσεις

1. Γράψτε, μεταγλωττίστε και εκτελέστε τον κώδικα του παραδείγματος.[2]

2. Τροποποιήστε τον κώδικα του παραδείγματος ώστε να τυπώνει αποτελέσματα άλλων εκφράσεων. Χρησιμοποιήστε τους τελεστές `+, -, *, /` (συμβολίζουν τις γνωστές πράξεις της αριθμητικής). Θυμηθείτε να διορθώσετε τα σχόλια και τα μηνύματα που τυπώνονται ώστε να ανταποκρίνονται στις αλλαγές.

3. Τι λάθη εντοπίζετε στον παρακάτω κώδικα C++ ;

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

4. Ποιο είναι το πιο σύντομο σωστό πρόγραμμα C++ ;

5. Τι λάθη εντοπίζετε στον παρακάτω κώδικα C++ ;

```
include iostream.h

Main();
{
 Double x,y,z;
 cout < "Enter␣two␣numbers␣";
 cin >> a >> b
 cout << 'The␣numbers␣in␣reverse␣order␣are' << b,a;
}
```

[2] 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

2.1.1 Σχόλια

Η C++ υποστηρίζει δύο ειδών σχόλια. Από τον compiler αγνοείται κείμενο που

- ξεκινά με `//` και τελειώνει με αλλαγή γραμμής,
- περιλαμβάνεται μεταξύ των `/*` και `*/` ανεξάρτητα από το πλήθος γραμμών που καταλαμβάνει.

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

Παράδειγμα:

```
// This is a comment
int j;  // Here is a declaration
/* Let us make
     an ugly multi-line
                        comment.
       */
```

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.

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
```

Προσέξτε ότι ο απαλοιφόμενος κώδικας πρέπει να είναι πλήρης, τουλάχιστον όσον αφορά τις παρενθέσεις (πρέπει να εμφανίζονται ανά ζεύγη). Επίσης, ο χαρακτήρας '#' πρέπει να είναι ο πρώτος μη κενός στις γραμμές εντολών προς τον προεπεξεργαστή.

2.1.2 Ονόματα

Τα ονόματα μεταβλητών, συναρτήσεων, τύπων, κλπ. στη C++ αποτελούνται από λατινικά γράμματα (a–z,A–Z), αριθμητικά ψηφία (0–9), και underscore (_). *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 (_) είναι πιθανό να χρησιμοποιούνται από τον compiler οπότε καλό είναι να αποφεύγονται από τον προγραμματιστή. Επίσης, δεν επιτρέπεται η χρήση των προκαθορισμένων λέξεων της C++ (Keywords, Πίνακας 2.1) για ονόματα μεταβλητών, συναρτήσεων, τύπων που ορίζονται από τον προγραμματιστή, κλπ.

Παράδειγμα:

- Μη αποδεκτά ονόματα:

  ena lathos onoma, pali_latho$, 1234qwer, delete, .onoma+

- Αποδεκτά ονόματα:

  timi, value12, ena_onoma_me_poly_megalo_mikos, sqrt, Delete

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

Πίνακας 2.1: Προκαθορισμένες λέξεις της C++.

2.2 Θεμελιώδεις Τύποι

Η C++ παρέχει ένα σύνολο θεμελιωδών τύπων που αντιστοιχούν στους πιο συνηθισμένους τρόπους αποθήκευσης δεδομένων:

2.2.1 Λογικός τύπος

Μεταβλητή λογικού τύπου (**bool**) είναι κατάλληλη για την αναπαράσταση ποσοτήτων που μπορούν να πάρουν δύο τιμές (π.χ. ναι/όχι, αληθές/ψευδές,...). Η δήλωση τέτοιας μεταβλητής, 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 ορίζεται έξω από κάθε συνάρτηση (§4.4), κλάση (Κεφάλαιο 6) και **namespace** (§2.8), οπότε είναι καθολική (global) και γίνεται **false**.

Δήλωση με ταυτόχρονη απόδοση συγκεκριμένης αρχικής τιμής είναι η παρακάτω:

```
bool a = true;
```

ή, ισοδύναμα,

```
bool a(true);
```

Ποσότητες τύπου **bool** και ακεραίων τύπων που θα δούμε παρακάτω, μπορούν να μετατραπούν η μία στην άλλη και, επομένως, να αναμιχθούν σε αριθμητικές και λογικές εκφράσεις. Όποτε χρειάζεται, μια λογική μεταβλητή με τιμή **true** ισοδυναμεί με 1 και με τιμή **false** ισοδυναμεί με 0. Αντίστροφα, μη μηδενικός ακέραιος μετατρέπεται σε **true** ενώ ακέραιος με τιμή 0 ισοδυναμεί με **false**. Επίσης, ένας μη μηδενικός δείκτης (§4.3) μπορεί να μετατραπεί σε **true** ενώ ο μηδενικός γίνεται **false**.

2.2.2 Τύπος χαρακτήρα

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

```
char c;
```

Οι τιμές που μπορεί να πάρει είναι ένας χαρακτήρας από το σύνολο χαρακτήρων της υλοποίησης· αυτό είναι σχεδόν πάντοτε, αλλά όχι υποχρεωτικά, το σύνολο ASCII.

Η δήλωση

```
char c = 'a';
```

ή, ισοδύναμα,

```
char c('a');
```

ορίζει μεταβλητή τύπου χαρακτήρα με όνομα c και με συγκεκριμένη αρχική τιμή, το σταθερό χαρακτήρα 'a'. Παρατηρήστε ότι ο τελευταίος περικλείεται σε απόστροφους (')· σε εισαγωγικά (") αποτελεί C-style string[1] που δεν μπορεί να αποδοθεί σε μεταβλητή τύπου **char**.

Κάποιοι από τους χαρακτήρες του συστήματος χρειάζονται ειδικό συμβολισμό για να αναπαρασταθούν. Παρουσιάζονται στον Πίνακα 2.2, μαζί με τους γενικούς τρόπους προσδιορισμού (σε δεκαεξαδικό και οκταδικό σύστημα) οποιουδήποτε χαρακτήρα. Π.χ.

```
char newline('\n');
char bell('\a');
char alpha('\141'); // alpha = 'a' in ASCII
char Alpha('\x61'); // Alpha = 'a' in ASCII
```

Οι ειδικοί χαρακτήρες μπορούν βεβαίως να περιλαμβάνονται και σε C-style string.

Παράδειγμα:

Με την εντολή

```
std::cout <<
  "This\nis\na\ntest\n\nShe␣said,␣\"How␣are␣you?\"\n";
```

εμφανίζεται στην οθόνη

```
This
is
a
test

She said, "How are you?"
```

Μεταβλητή τύπου **char** που ορίζεται έξω από κάθε συνάρτηση, **namespace** και κλάση, παίρνει αυτόματα ως αρχική τιμή το μηδενικό χαρακτήρα, ενώ αν είναι τοπική μεταβλητή έχει ακαθόριστη αρχική τιμή. Προσέξτε ότι άλλος χαρακτήρας είναι ο μηδενικός ('\0', ο χαρακτήρας με τιμή 0 στο σύνολο ASCII), και άλλος ο '0' (ο χαρακτήρας με δεκαδική τιμή 48 στο σύνολο ASCII).

Χαρακτήρες που συμμετέχουν σε εκφράσεις με άλλους ακέραιους τύπους, μετατρέπονται σε **int** (την τιμή τους στο σύνολο χαρακτήρων) ώστε να υπολογιστεί το αποτέλεσμα.

Ας αναφερθεί, χωρίς να υπεισέλθουμε σε λεπτομέρειες, ότι ένας **char** μπορεί να δηλωθεί ότι είναι **signed** ή **unsigned**.

[1] τύπου array of const char.

Ειδικοί Χαρακτήρες	Περιγραφή
\?	Ερωτηματικό
\’	Απόστροφος
\"	Εισαγωγικά
\\	Ανάποδη κάθετος
\a	Κουδούνι
\b	Διαγραφή χαρακτήρα
\f	Αλλαγή σελίδας
\n	Αλλαγή γραμμής
\r	Carriage return
\t	Οριζόντιο tab
\v	Κατακόρυφο tab
\xhhh	Χαρακτήρας με δεκαεξαδική αναπαράσταση hhh
\ooo	Χαρακτήρας με οκταδική αναπαράσταση ooo
\0	Μηδενικός χαρακτήρας

Πίνακας 2.2: Ειδικοί Χαρακτήρες.

2.2.3 Εκτεταμένος τύπος χαρακτήρα

Η C++ παρέχει τον τύπο **wchar_t** και κατάλληλες δομές και συναρτήσεις για την αποθήκευση και χειρισμό χαρακτήρων σε αλφάβητο με περισσότερους από 256 χαρακτήρες. Τέτοια αλφάβητα είναι π.χ. τα σύνολα χαρακτήρων των ασιατικών γλωσσών που δεν μπορούν να αναπαρασταθούν στις 256 θέσεις ενός συνόλου όπως π.χ. το ASCII.

2.2.4 Ακέραιοι τύποι

Μια ακέραια μεταβλητή (π.χ. τύπου **int**) δηλώνεται ως εξής:

```
int i;
```

Αν η παραπάνω δήλωση γίνει έξω από κάθε συνάρτηση, κλάση και **namespace**, η μεταβλητή i παίρνει την τιμή 0, αλλιώς η τιμή της είναι απροσδιόριστη. Αρχική τιμή (π.χ. 10) δίνεται με την εντολή

```
int i = 10;
```

ή, ισοδύναμα, με την

```
int i(10);
```

Οι τιμές που μπορεί να λάβει μια ακέραια μεταβλητή καθορίζονται από την υλοποίηση. Το μέγεθος του τύπου **int** απαιτείται να είναι τουλάχιστο 16 bits, επομένως μπορεί να αναπαραστήσει αριθμούς στο διάστημα $\left[-2^{15}, 2^{15}\right) = [-32768, 32768)$ 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>`:

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

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

  return 0;
}
```

Στη C++ υπάρχουν τρία είδη ακεραίων, **short int**, **int** και **long int**, με ελάχιστα μεγέθη τα 16, 16, 32 bits αντίστοιχα. Επιπλέον, το μέγεθος του **short int** είναι υποχρεωτικά μικρότερο ή ίσο με το μέγεθος του **int** και αυτό με τη σειρά του είναι μικρότερο ή ίσο από το μέγεθος του **long int**. Το ακριβές μέγεθος, *σε πολλαπλάσια του μεγέθους του* ***char***, δίνεται από τον τελεστή **sizeof**() με όρισμα τον κάθε τύπο, §2.7.1. Καθένας από τους τύπους ακεραίου μπορεί να ορίζεται ως **signed** (προεπιλεγμένο) ή **unsigned**.

Αν δεν υπάρχει κάποιος ειδικός λόγος για το αντίθετο, καλό είναι να χρησιμοποιείται ο απλός τύπος **int**. Στη C++ προβλέπεται η μετατροπή σε **int** όλων των ακέραιων ποσοτήτων που εμφανίζονται σε μία έκφραση με τύπο "μικρότερο" από **int** προτού εκτελεστούν οι πράξεις και υπολογιστεί η τιμή της έκφρασης.

Ακέραιες Σταθερές

Μία σειρά αριθμητικών ψηφίων χωρίς κενά ή άλλα σύμβολα, *που δεν αρχίζει από* 0, αποτελεί μία ακέραια σταθερά στο δεκαδικό σύστημα. Επιτρέπεται να αρχίζει με το πρόσημο, '+' ή '−'. Αν το πρώτο ψηφίο είναι 0 (και δεν ακολουθείται από 'x' ή 'X') θεωρείται οκταδικός αριθμός και πρέπει να περιλαμβάνει μόνο 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).

Αριθμός τύπου **long int** υποδηλώνεται με το χαρακτήρα L ή l αμέσως μετά τη σειρά των ψηφίων:

12L, 0xBABEL, -665l

Αν ο αριθμός ακολουθείται από U ή u είναι τύπου **unsigned int**, ενώ ο συνδυασμός των δύο χαρακτήρων στο τέλος του αριθμού είναι επιτρεπτός και υποδηλώνει τύπο **unsigned long int**.

2.2.5 Πραγματικοί τύποι

Στη C++ ορίζονται τρεις τύποι πραγματικών αριθμών: απλής ακρίβειας (**float**), διπλής ακρίβειας (**double**) και εκτεταμένης ακρίβειας (**long double**). 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**. Το πλήθος των σημαντικών ψηφίων και τα ακριβή όρια του **float** μπορούν να βρεθούν αν τυπώσουμε τον ακέραιο `std::numeric_limits<float>::digits10` και τις επιστρεφόμενες τιμές των συναρτήσεων

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

και

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

που ορίζονται στο header `<limits>`. Αντίστοιχα ισχύουν και για τους άλλους τύπους. Προσέξτε ότι η εξειδίκευση της `std::numeric_limits<>::min()` για τους πραγματικούς τύπους μας επιστρέφει το μικρότερο *θετικό* αριθμό.

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.

Πραγματικές Σταθερές

Μία σειρά αριθμητικών ψηφίων χωρίς κενά, που περιλαμβάνει τελεία (στη θέση της υποδιαστολής) συμβολίζει πραγματική σταθερά τύπου **double**. Πριν ή μετά την υποδιαστολή μπορεί να μην υπάρχουν ψηφία. Ο χαρακτήρας e ή E, αν υπάρχει, ακολουθείται από τον ακέραιο εκθέτη του 10 με τη δύναμη του οποίου πολλαπλασιάζεται ο αμέσως προηγούμενος του e/E αριθμός:

2.034, 0.23, .44, 23., 2e−4 (≡ 0.0002), 2.3E2 (≡ 230.0) .

Αν ο αριθμός τελειώνει σε F ή f είναι τύπου **float**· αν τελειώνει σε L ή l είναι τύπου **long double**.

2.2.6 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 αυτόν τον εναλλακτικό τρόπο την

κενή λίστα ορισμάτων. Η μόνη άλλη χρήση του είναι στον τύπο **void*** (δείκτης σε **void**) ως δείκτης σε αντικείμενο άγνωστου τύπου. Με αυτή τη μορφή χρησιμοποιείται ως τύπος ορίσματος ή επιστρεφόμενης τιμής γενικευμένης συνάρτησης.

2.2.7 Enumeration

Enumeration (αρίθμηση) είναι ένας ακέραιος τύπος οι επιτρεπτές τιμές του οποίου καθορίζονται από τον προγραμματιστή. Π.χ.

```
enum 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 είναι η ακόλουθη:

```
Color c(RED);
```

Μία αρίθμηση είναι χρήσιμη για να συγκεντρώνει τις τιμές ενός **switch** (§3.1.3), δίνοντας τη δυνατότητα στον compiler να μας ειδοποιεί αν παραλείψουμε κάποια. Επίσης, είναι χρήσιμη ως τύπος επιστροφής μιας συνάρτησης (§4.4).

Οι τιμές μιας αρίθμησης παίρνουν ακέραια τιμή ανάλογα με τη θέση τους στη λίστα της: η κάθε μία είναι κατά ένα μεγαλύτερη από την προηγούμενή της 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 αρίθμησης μπορεί να οριστεί άλλη ακέραια τιμή· σε αυτήν την περίπτωση, οι ποσότητες για τις οποίες δεν έχει οριστεί ρητά συγκεκριμένη ακέραια τιμή είναι πάλι κατά 1 μεγαλύτερες από τις αμέσως προηγούμενές τους:

```
enum Color {RED, GREEN=5, BLUE};
// RED = 0, GREEN = 5, BLUE = 6
```

Αυτές είναι οι τιμές με τις οποίες συμμετέχουν σε αριθμητικές εκφράσεις. Το αντίστροφο, δηλαδή αυτόματη μετατροπή ακεραίου σε **enum** δεν ισχύει. Η ακόλουθη εντολή είναι λάθος:

```
Color d(2);     // Error
```

2.3 Μιγαδικός τύπος

Παρόλο που η υποστήριξη των μιγαδικών αριθμών δεν είναι εγγενής στη C++ αλλά παρέχεται μέσω template class (§6.3), παρουσιάζεται σε αυτό το σημείο λόγω της μεγάλης χρησιμότητας των αριθμών αυτών σε επιστημονικούς κώδικες.

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

```
#include <complex>
```

προς τον προεπεξεργαστή.

Δήλωση μεταβλητής (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 τύπο (**int**, **float**, **long double**,...). Αν δεν καθοριστεί αρχική τιμή για τη μεταβλητή, δίνεται αυτόματα η τιμή 0, *ανεξάρτητα από το πού εμφανίζεται η δήλωση*, όπως ισχύει για όλα τα αντικείμενα τύπων που παρέχονται από την STL.

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

```
std::complex<double> z1(3.41);  // or z1 = 3.41;
// z1 = 3.41 + 0.0i

std::complex<double> z2(3.0,2.0);
// z2 = 3.0 + 2.0i

std::complex<double> z3(z2);    // or z3 = z2;
// z3 = z2

std::complex<double> z4(std::polar(2.0));
// z4 = 2.0 exp(0.0i)

std::complex<double> z5(std::polar(2.0,0.75));
// z5 = 2.0 exp(0.75i)
```

Η συνάρτηση `std::polar()` επιστρέφει μιγαδικό αριθμό με μέτρο (magnitude) το πρώτο όρισμα και φάση (phase angle) (σε rad) το δεύτερο.

Οι αριθμητικοί τελεστές `+, -, *, /` και οι συντμήσεις `+=, -=, *=, /=` (§2.7) μεταξύ ακεραίων, πραγματικών και μιγαδικών εκτελούν τις αναμενόμενες και γνωστές πράξεις από τα μαθηματικά. Επίσης, οι μαθηματικές συναρτήσεις της C++ (§4.1) δέχονται μιγαδικά ορίσματα, επιστρέφοντας το αντίστοιχο μιγαδικό αποτέλεσμα. Να διευκρινίσουμε ότι η συνάρτηση `std::abs()` με μιγαδικό όρισμα επιστρέφει το μέτρο του. Επιπλέον, η συνάρτηση `std::norm()` παρέχει το τετράγωνο του μέτρου του ορίσματός της ενώ η `std::arg()` τη φάση του· αν $z = a + \mathrm{i}\beta$ τότε

`std::abs(z)` = $\sqrt{a^2 + \beta^2}$,
`std::norm(z)` = $zz^* \equiv a^2 + \beta^2$,
`std::arg(z)` = $\arctan(\beta/a)$.

Ακόμη, η συνάρτηση `std::conj()` επιστρέφει το συζυγή του μιγαδικού ορίσματός της. Οποιαδήποτε έκφραση περιλαμβάνει τα παραπάνω επιστρέφει μιγαδικό και, επομένως, μπορεί να χρησιμοποιηθεί για την απόδοση αρχικής τιμής.

Το πραγματικό και το φανταστικό μέρος ενός μιγαδικού αριθμού z μπορεί να *αναγνωστεί μόνο*, είτε με τις συναρτήσεις `std::real(z)` και `std::imag(z)`, είτε με τις συναρτήσεις-μέλη της κλάσης `std::complex<>`, `z.real()` και `z.imag()`.

Για να αποδοθεί νέα τιμή στο πραγματικό ή φανταστικό μέρος ενός μιγαδικού πρέπει να γίνει κατασκευή και ανάθεση νέου μιγαδικού, είτε ρητά είτε αυτόματα από τον compiler:

```
std::complex<double> z(3.0,1.0);
// z =  3.0 + 1.0i

z = std::complex<double>(z.real(), 3.7);
// z =  3.0 + 3.7i

z = std::complex<double>(2.3, std::imag(z));
// z =  2.3 + 3.7i

z = 5.0; // or z = std::complex<double>(5.0);
// z =  5.0 + 0.0i

z = std::complex<double>(-1.0, 2.0);
// z = -1.0 + 2.0i
```

Η εγγραφή μιγαδικών σε αρχείο με τον τελεστή << γίνεται με τη μορφή

(πραγματικό,φανταστικό)

Η ανάγνωση μιγαδικών από αρχείο με τον τελεστή >> γίνεται με μία από τις παρακάτω διαμορφώσεις:

(πραγματικό,φανταστικό)
(πραγματικό)
πραγματικό

Η χρήση του μιγαδικού τύπου, μεταξύ άλλων, διευκολύνεται με την κατάλληλη χρήση της εντολής **`typedef`** (§2.5.1).

2.4 Σχεσιακοί και Λογικοί Τελεστές

Η C++ υποστηρίζει τη σύγκριση ποσοτήτων με τη βοήθεια των *σχεσιακών* τελεστών (Πίνακας 2.3). Το αποτέλεσμα της σύγκρισης είναι λογική ποσότη-

==	ίσο	!=	άνισο
>	μεγαλύτερο	<	μικρότερο
>=	μεγαλύτερο ή ίσο	<=	μικρότερο ή ίσο

Πίνακας 2.3: Σχεσιακοί τελεστές στη C++.

τα και επομένως έχει τιμή **`true`** ή **`false`**. Π.χ. το `3.0>2.0` είναι **`true`** ενώ

το `2!=1+1` είναι **`false`**. Οι αριθμητικοί τελεστές έχουν μεγαλύτερη προτεραιότητα από τους σχεσιακούς. τελεστές για μιγαδικούς αριθμούς ορίζονται, όπως είναι αναμενόμενο, μόνο οι (==, ισότητα) και (!=, ανισότητα). Οι πραγματικοί αριθμοί που τυχόν συμμετέχουν, μετατρέπονται στους αντίστοιχους μιγαδικούς πριν τη σύγκριση.

Για τη σύνδεση λογικών εκφράσεων η C++ παρέχει τους λογικούς τελεστές `!` (NOT), `&&` (AND), `||` (OR). Ισοδύναμες με αυτούς τους τελεστές, αλλά λιγότερο χρησιμοποιούμενες, είναι οι δεσμευμένες λέξεις **`not`**, **`and`** και **`or`** αντίστοιχα.

Οι τελεστές `&&` και `||` δρουν μεταξύ δύο λογικών ποσοτήτων ή εκφράσεων και σχηματίζουν μια νέα λογική ποσότητα ενώ ο τελεστής `!` δρα σε μία:

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

  Το `! (4 > 3)` είναι **`false`**.
  Το `! (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`**:

  Το `(4 > 3) && (3.0 > 2.0)` είναι **`true`**
  Το `(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`**:

  Το `(4 > 3) || (3.0 < 2.0)` είναι **`true`**
  Το `(4 < 3) || (3.0 < 2.0)` είναι **`false`**

Μια έκφραση με σχεσιακούς ή λογικούς τελεστές της C++, μπορεί να ανατεθεί σε μεταβλητές τύπου **`bool`**:

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

Ας αναφέρουμε εδώ ένα σημαντικό χαρακτηριστικό της C++: ο υπολογισμός των λογικών εκφράσεων με θεμελιώδεις τύπους εκτελείται από αριστερά προς τα δεξιά και σταματά όταν έχει προσδιοριστεί η τελική τιμή (short-circuit evaluation). Π.χ. στην έκφραση

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

αν ισχύει `i < 0` τότε η συνολική έκφραση είναι **`true`** ανεξάρτητα από τη δεύτερη συνθήκη, η οποία δεν υπολογίζεται. Ανάλογα, στην έκφραση

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

αν `i >= 0` τότε 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 κλήση συνάρτησης με μεγάλες απαιτήσεις σε χρόνο εκτέλεσης ή μνήμη.

2.5 Γενικές Παρατηρήσεις

2.5.1 typedef

Ένας τύπος με ιδιαίτερα μεγάλο όνομα είναι δύσχρηστος, ειδικά όταν επαναλαμβάνεται. Μπορούμε να ορίσουμε μια άλλη, πιο σύντομη, ονομασία για τέτοιο τύπο με τη βοήθεια του **`typedef`**. Έτσι, ο τύπος π.χ. `std::complex<`**`double`**`>` μπορεί να χρησιμοποιείται με το πιο σύντομο όνομα `complex` αν προηγηθεί η εντολή:

```
typedef std::complex<double> complex;
```

Μια δήλωση μιγαδικής μεταβλητής μπορεί κατόπιν να γίνει ως εξής:

```
complex z;
```

Μια άλλη χρήση του **`typedef`** είναι για να "εντοπιστεί" η δήλωση ενός τύπου ώστε να μπορεί να αλλάξει πολύ εύκολα: έστω ότι ορίζουμε μία συνάρτηση που χειρίζεται πραγματικούς αριθμούς διπλής ακρίβειας. Ο τύπος τους θα είναι **`double`**. Αν κατόπιν θελήσουμε να να φτιάξουμε μια όμοια συνάρτηση που να χειρίζεται πραγματικούς αριθμούς απλής ακρίβειας, πρέπει να αλλάξουμε κάθε εμφάνιση του τύπου **`double`** με τον τύπο **`float`**. Εναλλακτικά, στην αρχική ρουτίνα μπορούμε να δηλώσουμε όλους τους πραγματικούς ως `real` έχοντας ορίσει πιο πριν ότι

```
typedef double real;
```

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

```
typedef float real;
```

Η τελευταία εφαρμογή της **`typedef`** μπορεί να γίνει στη C++ πιο αποτελεσματικά με τη βοήθεια των template functions (§4.7).

2.5.2 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 τέτοιας ποσότητας γίνεται χρησιμοποιώντας την προκαθορισμένη λέξη **`const`** και συνοδεύεται υποχρεωτικά με απόδοση της αρχικής (και μόνιμης) τιμής:

```
double const pi(3.141592653589793);
int    const maximum(100);
```

Σε ποσότητες που έχουν δηλωθεί ως **const** αυτονόητο είναι ότι δεν μπορεί να γίνει ανάθεση τιμής.

Καλό είναι να χρησιμοποιούνται συμβολικές σταθερές για να αποφεύγεται η χρήση "μαγικών αριθμών" στον κώδικα. Αν μία ποσότητα που είναι σταθερή (π.χ. πλήθος στοιχείων σε πίνακα, 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.

2.5.3 Δηλώσεις και απόδοση αρχικής τιμής

Οι δηλώσεις μεταβλητών και σταθερών πρέπει να είναι μοναδικές στο block που ορίζονται και βέβαια σε ένα block δεν επιτρέπεται να χρησιμοποιείται το ίδιο όνομα για διαφορετικές ποσότητες. Μπορούν να εμφανίζονται σε σχεδόν οποιοδήποτε σημείο του κώδικα. Οι ποσότητες είναι καλό να δηλώνονται στο σημείο που χρειάζονται (ή αμέσως πριν) και, όποτε είναι δυνατό, να τους αποδίδεται ταυτόχρονα και η αρχική τους τιμή. Όπως αναφέρθηκε, οι μεταβλητές θεμελιωδών τύπων που ορίζονται έξω από κάθε συνάρτηση, κλάση και **namespace** (εκτός του global) χωρίς αρχική τιμή, παίρνουν αυτόματα την τιμή 0 (μετατρεπόμενη στον κατάλληλο τύπο). Αν είναι τοπικές μεταβλητές, και δε δηλώνονται ως **static** (§4.9), έχουν αυθαίρετη αρχική τιμή και δε θα πρέπει να βασιζόμαστε στο ότι κάποιος compiler μπορεί να επιλέξει να τις μηδενίσει.

Για την απόδοση αρχικής τιμής στους ενσωματωμένους τύπους της C++ παρουσιάστηκε αρχικά ένας ιδιαίτερα διαδεδομένος τρόπος, όπως κληρονομήθηκε από τη C:

```
int a = 3;
```

Όπως είδαμε στο §2.3, η συγκεκριμένη σύνταξη δεν μπορεί να επεκταθεί για την απόδοση αρχικής τιμής σε αντικείμενα κλάσεων (όπως είναι οι μιγαδικοί στη C++) όταν χρειάζονται περισσότερες από μία ποσότητες για τον καθορισμό της αρχικής κατάστασης. Η C++ εισήγαγε διαφορετική σύνταξη για αυτήν την πράξη,

```
std::complex<double> z(1.0,2.0); // z = 1.0 + 2.0 i
```

Η συγκεκριμένη σύνταξη μπορεί να χρησιμοποιηθεί και στους ενσωματωμένους τύπους· οι παρακάτω δηλώσεις με απόδοση αρχικής τιμής είναι ισοδύναμες:

```
int b = 4;
int b(4);
```

Εναλλακτικά, για ποσότητες είτε ενσωματωμένου τύπου (π.χ. **int**) είτε κατασκευασμένου από το χρήστη (π.χ. `std::complex<double>`), μπορεί ισοδύναμα να χρησιμοποιηθεί η ακόλουθη σύνταξη για απόδοση αρχικής τιμής:

```
int c = int(3);   // c = 3

std::complex<double> z = std::complex<double>(3.0, 2.0);
// z = 3.0 + 2.0i

int d = int();
// d gets default integer value, 0.

std::complex<double> y = std::complex<double>();
//  y gets the default value specified
//  in complex<> class, 0.0 + 0.0i.
```

Η ομοιομορφία στη σύνταξη κατά την απόδοση αρχικής τιμής σε ενσωματωμένους τύπους και τύπους καθοριζόμενους από το χρήστη είναι απαραίτητη για την υλοποίηση των templates (§4.7,§6.3).

Η γλώσσα παρέχει τη δυνατότητα να συνδυάζονται δηλώσεις μεταβλητών ίδιου τύπου σε μία. Έτσι οι δηλώσεις

```
int a;
int b;
```

μπορούν ισοδύναμα να γίνουν

```
int a,b;
```

Ο κώδικας είναι ευκρινέστερος αν η κάθε δήλωση γίνεται σε ξεχωριστή γραμμή καθώς μας διευκολύνει να παραθέτουμε σχόλια για την ποσότητα που ορίζεται και ελαχιστοποιεί την πιθανότητα λάθους δήλωσης κάποιας μεταβλητής. Π.χ. η δήλωση

```
int * a,b;
```

ορίζει ένα δείκτη σε ακέραιο (όπως θα δούμε παρακάτω), τον `a`, και ένα ακέραιο, το `b`. Το πιο πιθανό είναι ότι ο προγραμματιστής επιθυμούσε δήλωση δύο δεικτών.

2.5.4 Εμβέλεια

Όλες οι μεταβλητές, σταθερές, συναρτήσεις, τύποι μπορούν να χρησιμοποιηθούν από το σημείο της δήλωσής τους[2] έως το καταληκτικό } του block στο οποίο ανήκουν. Οι μεταβλητές και οι σταθερές ποσότητες χάνουν την τιμή τους μετά από αυτό το σημείο και λέμε ότι έχουν καταστραφεί.

Οι καθολικές (global) ποσότητες, αυτές δηλαδή που ορίζονται έξω από κάθε συνάρτηση (§4.4), κλάση (Κεφάλαιο 6) και **namespace** (§2.8), έχουν εμβέλεια μέχρι το τέλος του αρχείου στο οποίο γίνεται η δήλωση (και σε όσα αρχεία συμπεριλαμβάνεται αυτό με **#include**). Καθώς αυτές οι ποσότητες "φαίνονται" από μεγάλα τμήματα του κώδικα, οι αλλαγές τους είναι δύσκολο να εντοπιστούν. Για το λόγο αυτό, η χρήση τους θα πρέπει να είναι εξαιρετικά

[2]για τις συναρτήσεις, η δήλωση και ο ορισμός δεν ταυτίζονται απαραίτητα.

σπάνια, μόνο για τις περιπτώσεις που δε γίνεται να την αποφύγουμε. Οι δομές που θα συναντήσουμε στο Κεφάλαιο 3 καθώς και οι συναρτήσεις αποτελούν ξεχωριστό χώρο ονομάτων με δική τους εμβέλεια.

Προσοχή θέλει η περίπτωση που χρησιμοποιείται το ίδιο όνομα για διαφορετικές ποσότητες σε δύο διαφορετικά block το ένα μέσα στο άλλο. Π.χ.

```
#include <iostream>

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

  return 0;
}                                       // end   block A
```

Η μεταβλητή x στο εσωτερικό block "κρύβει" σε αυτό τη x του εξωτερικού block· οποιαδήποτε ανάθεση ή χρήση τιμής στο x αναφέρεται εκεί 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.

2.6 Σύνθετοι Τύποι

Χρησιμοποιώντας τους παραπάνω θεμελιώδεις τύπους μπορούμε να ορίσουμε άλλους, σύνθετους, με κατάλληλο τρόπο ώστε να αναπαριστούν έννοιες του προγράμματός μας. Έτσι για ομάδα σχετιζόμενων ποσοτήτων ίδιου τύπου χρησιμοποιούμε πίνακα (array) ενώ για σύνολο μεταβλητών διαφορετικού (ή και ίδιου) τύπου κατάλληλες είναι η δομή (**struct**) και η επέκτασή της, η κλάση (**class**).

2.6.1 Πίνακες

Ένα σύνολο σχετιζόμενων πραγματικών αριθμών που αντιπροσωπεύουν π.χ. τη θερμοκρασία το μεσημέρι σε μία πόλη κατά τη διάρκεια του έτους, είναι επιθυμητό να αποθηκεύεται σε ένα μονοδιάστατο *πίνακα (array)* πραγματικών αριθμών με 365 στοιχεία παρά σε ισάριθμες ανεξάρτητες μεταβλητές. Η χρήση

πίνακα διευκολύνει πολύ την αναζήτηση, ταξινόμηση και, γενικά, διαχείριση σχετιζόμενων ποσοτήτων.

Η δήλωση στη C++ ενός μονοδιάστατου πίνακα γίνεται συμβολικά ως εξής:

τύπος όνομα[πλήθος_στοιχείων];

Το *πλήθος_στοιχείων* πρέπει να είναι μία *ακέραιη σταθερά* ή μία *ακέραιη σταθερή ποσότητα, γνωστή κατά τη μεταγλώττιση του κώδικα.* Για άγνωστο πλήθος στοιχείων μπορούμε να χρησιμοποιούμε containers της STL, Κεφάλαιο 5.

Οι παρακάτω δηλώσεις ορίζουν τον πραγματικό πίνακα temperature με 365 στοιχεία και ένα πίνακα των τετραψήφιων εσωτερικών τηλεφώνων μιας εταιρίας (που χωρούν σε **int**):

```
double temperature[365];

int const N(155);
int telephone[N];
```

Η δήλωση του telephone έγινε με "παράμετρο" το πλήθος των στοιχείων ώστε μία αλλαγή του να μην απαιτεί εκτεταμένες τροποποιήσεις στον κώδικα.

Πρόσβαση στα στοιχεία ενός πίνακα, π.χ. του temperature, γίνεται βάζοντας σε αγκύλες μετά το όνομα του πίνακα ένα ακέραιο *μεταξύ 0 και D−1* όπου D η διάσταση. Το πρώτο, δηλαδή, στοιχείο του πίνακα είναι στη θέση 0. Π.χ.

```
temperature[0] = 14.0;
temperature[1] = 14.5;
temperature[2] = 15.5;
temperature[3] = 13.0;
temperature[4] = 15.0;
//.........

int i(3);

std::cout << "The␣Temperature␣on␣";
std::cout << i+1;                // Notice the increment
std::cout << "␣day␣was:␣";
std::cout << temperature[i];
std::cout << '\n';
```

Προσέξτε ότι αν δώσετε δείκτη (index) εκτός των ορίων του πίνακα, δηλαδή κάτω από το 0 ή πάνω από D−1 *δε* θα διαγνωστεί ως λάθος από τον compiler (δοκιμάστε το!).

Μπορούμε να δώσουμε αρχικές τιμές στα στοιχεία ενός πίνακα εάν κατά τον ορισμό του παραθέσουμε λίστα τιμών με ίδιο (ή μετατρέψιμο) τύπο με τα στοιχεία. Κατά τη δήλωση με αρχική τιμή μπορούμε να παραλείψουμε τη διάσταση του πίνακα οπότε υπολογίζεται από τον compiler με βάση το πλήθος των παρατιθέμενων τιμών. Αν υπάρχει και η διάσταση δεν επιτρέπεται να δίνονται περισσότερες αρχικές τιμές, ενώ, αν παρατίθενται λιγότερες, οι υπόλοιπες θεωρούνται 0 (μετατρεπόμενο στον αντίστοιχο τύπο):

```
int primes[5] = {1,2,3,5,7};

int digits[] = {0,1,2,3,4,5,6,7,8,9};   // size is 10

char alphabet[5] = {'a', 'b', 'c', 'd', 'e', 'f'};   // Error

int a[5] = {12, 5, 4};      // a == {12,5,4,0,0}
```

Οι πίνακες πραγματικών αριθμών είναι η βασική (και ουσιαστικά η μόνη) δομή που παρέχουν γλώσσες προγραμματισμού όπως οι C και Fortran για επιστημονικούς υπολογισμούς. Ας σημειωθεί ότι οι διδιάστατοι (και γενικά πολυδιάστατοι) πίνακες στη C και C++ παρουσιάζουν μειονεκτήματα στη χρήση τους σε τέτοιους υπολογισμούς και (θα έπρεπε να) αποφεύγονται.

Ένας διδιάστατος πίνακας στη C++ ορίζεται συμβολικά ως εξής:

τύπος όνομα[πλήθος_στοιχείων_1][πλήθος_στοιχείων_2];

Πίνακες περισσότερων διαστάσεων ορίζονται ανάλογα. Επομένως, ένας ακέραιος 6×8 διδιάστατος πίνακας είναι ο

```
int a[6][8];
```

Το στοιχείο π.χ. (3,2) του πίνακα αυτού είναι προσπελάσιμο ως `a[3][2]`.

Απόδοση αρχικών τιμών γίνεται ανά γραμμή, ως εξής:

```
int b[2][3] = { {0, 1, 2}, {3, 4, 5} };
```

Πίνακες πολυδιάστατοι, *με γνωστές διαστάσεις κατά τη μεταγλώττιση του κώδικα*, μπορούν να ορίζονται με τον παραπάνω τρόπο. Σε πίνακα με τέτοιο ορισμό, είναι καλό, όταν τον διατρέχουμε, να μεταβάλλεται πιο γρήγορα ο τελευταίος δείκτης· τα στοιχεία αποθηκεύονται *κατά γραμμές* (row-major order). *Προσέξτε ότι στη Fortran η αποθήκευση γίνεται κατά στήλες* (column-major order). Συνεπώς, ένας πολυδιάστατος πίνακας της C ή C++ αντιμετωπίζεται ως ο ανάστροφός του από ρουτίνες Fortran.

Εναλλακτικά, είναι προτιμότερη για διάφορους λόγους, αν και πιο δύσχρηστη, η αντιμετώπισή τους ως μονοδιάστατους σε column-major order με την εξής μορφή:

- Ένας μαθηματικός διδιάστατος πίνακας $[A_{ij}]$ με διαστάσεις $N1 \times N2$, ορίζεται στη C++ ως μονοδιάστατος με πλήθος στοιχείων $N1 * N2$. Το στοιχείο $A_{i,j}$ αντιστοιχεί στο `a[i+ N1*j]`.

- Ένας μαθηματικός τριδιάστατος πίνακας $[A_{ijk}]$ με διαστάσεις $N1 \times N2 \times N3$, ορίζεται ως μονοδιάστατος με πλήθος στοιχείων $N1 * N2 * N3$. Το στοιχείο $A_{i,j,k}$ αντιστοιχεί στο `a[i + N1 * (j + N2 * k)]`.

- Αντίστοιχα ισχύουν για περισσότερες διαστάσεις.

"Τεχνάσματα" που χρησιμοποιούνται από προγραμματιστές της C για να ορίσουν με πιο εύχρηστο τρόπο τους πολυδιάστατους πίνακες, τους καθιστούν συνήθως αργούς και ασύμβατους με τους πίνακες αντίστοιχης διάστασης της Fortran και, συνεπώς, ακατάλληλους για τις εκτεταμένες συλλογές μαθηματικών ρουτινών που έχουν γραφεί στη γλώσσα αυτή.

Όπως θα δούμε, η C++ παρέχει το μηχανισμό για να απλοποιείται σημαντικά η χρήση πολυδιάστατων πινάκων. Επιπλέον, εισήγαγε μέσω της Standard Library (STL) νέες δομές προς αντικατάσταση των μονοδιάστατων πινάκων με αρκετά πλεονεκτήματα· θα αναφερθούμε σε αυτές αναλυτικά στο αντίστοιχο κεφάλαιο. Εδώ μπορούμε να κάνουμε μια απλή αναφορά στην ιδιαίτερα χρήσιμη κλάση `std::vector<>` από το header `<vector>`: ορισμός μονοδιάστατου πίνακα με όνομα v, 30 στοιχείων τύπου π.χ. **double** γίνεται ως εξής

```
std::vector<double> v(30);
```

Προσέξτε τη χρήση παρενθέσεων στον ορισμό. Η διάσταση *δεν είναι απαραίτητα γνωστή κατά τη μεταγλώττιση*, μπορεί να δοθεί εξωτερικά ή να υπολογιστεί κατά την εκτέλεση του προγράμματος:

Παράδειγμα:

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

int main() {
   int N;

   std::cin >> N; // get dimension
   std::vector<double> v(N);

   // v[0], v[1], ..., v[N-1]
}
```

Κατά τα λοιπά, η χρήση του πίνακα v είναι ακριβώς όμοια με τους ενσωματωμένους πίνακες της C++. Τις επιπλέον δυνατότητες που μας παρέχει η κλάση `std::vector<>` θα τις αναλύσουμε στο Κεφάλαιο 5.

2.6.2 Δομή (struct)

Μία άλλου είδους σύνθετη δομή είναι επίσης κατάλληλη για την περιγραφή σε κώδικα π.χ. ενός χημικού στοιχείου. 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 αναπαριστώνται καλύτερα ως ένα σύνολο που συνδυάζει σειρές χαρακτήρων, ακέραιους και πραγματικούς αριθμούς. Θα δούμε αργότερα ότι ο κατάλληλος τύπος για την αναπαράσταση σειράς χαρακτήρων στη C++ είναι ο `std::string` από τον header `<string>`. Με αυτό υπόψη ο νέος τύπος ορίζεται, σε οποιοδήποτε σημείο επιτρέπεται μια δήλωση, ως εξής:

```
struct ChemicalElement {
   double mass;
   int Z; // atomic number
   std::string name;
   std::string symbol;
};
```

Παρατηρείστε το (;) που ακολουθεί το καταληκτικό }. Η δήλωση δομής (ή κλάσης) είναι ένα από τα λίγα σημεία της C++ που εμφανίζεται ο συνδυασμός };.[3]

Ο νέος τύπος με το όνομα ChemicalElement εισάγεται με την προκαθορισμένη λέξη **struct** ακολουθούμενο από το όνομα και, εντός αγκίστρων, δηλώσεις ποσοτήτων (χωρίς συγκεκριμένη σειρά) θεμελιωδών ή άλλων σύνθετων τύπων. Οι ποσότητες αυτές είναι τα *μέλη* της δομής και η εμβέλειά τους περιορίζεται στο σώμα της δομής. Μια μεταβλητή τύπου ChemicalElement ορίζεται όπως οποιαδήποτε ποσότητα θεμελιώδους τύπου:

```
ChemicalElement oxygen;
```

Απόδοση αρχικής τιμής μπορεί να γίνει με τον τρόπο που είδαμε στους πίνακες· 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"};
```

Πρόσβαση ατομικά στα μέλη μιας δομής γίνεται με τον τελεστή (.)· το όνομα της δομής ακολουθείται από (.) και το όνομα του μέλους:

```
oxygen.name = "Oxygen";
oxygen.mass = 15.99494;
oxygen.Z = 8;
oxygen.symbol = "O";

std::cout << "The␣mass␣of␣element␣";
std::cout << oxygen.name;
std::cout << "␣is␣";
std::cout << oxygen.mass;
std::cout << '\n';
```

Η δομή (**struct**) που υπάρχει στη C αποτέλεσε τη βάση για την ανάπτυξη των κλάσεων στη C++ όπως θα δούμε στο Κεφάλαιο 6. Οι κλάσεις επιτρέπουν επιπλέον τη δήλωση συναρτήσεων ως μέλη σε μία δομή.

2.7 Αριθμητικοί Τελεστές

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

[3] Τον συναντήσαμε ήδη στην ανάθεση αρχικών τιμών σε πίνακες και αριθμήσεις.

- Μοναδιαίοι `+`, `-`: Δρώντας σε ένα αριθμό a μας δίνουν τον ίδιο αριθμό ή τον αντίθετό του αντίστοιχα.
- Δυαδικοί `+`, `-`, `*`: Δρώντας μεταξύ δύο αριθμών a,b μας δίνουν το άθροισμα, τη διαφορά και το γινόμενό τους αντίστοιχα.
- Δυαδικός `/` μεταξύ πραγματικών a,b: δίνει το λόγο a/b.
- Δυαδικοί `/`, `%` μεταξύ ακεραίων δίνουν *το πηλίκο και το υπόλοιπο* της διαίρεσης αντίστοιχα.

Προσέξτε ότι δεν υπάρχει τελεστής για ύψωση σε δύναμη. Αντ' αυτού χρησιμοποιείται η συνάρτηση `std::pow()` που περιλαμβάνεται στον header `<cmath>` (Πίνακας 4.1).

Ένας ιδιωματισμός της C++ είναι οι συντμήσεις των παραπάνω τελεστών με το (=): η πράξη `a = a + b;` γράφεται συνήθως ως `a+=b;`. Αντίστοιχα ισχύουν και για τους άλλους τελεστές (`+=`, `-=`, `*=`, `/=`, `%=`) *χωρίς κενά μεταξύ του τελεστή και του* `=`. Οι δύο εκφράσεις παραπάνω είναι ισοδύναμες, εκτός από την περίπτωση που ο υπολογισμός της μεταβλητής a παρουσιάζει "παρενέργειες" σαν και αυτή που θα δούμε αμέσως παρακάτω.

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 στο a και η νέα τιμή χρησιμοποιείται για να υπολογιστεί η έκφραση. Αν δρουν μετά, π.χ. όπως στην έκφραση `b = a++ + c;` τότε πρώτα υπολογίζεται η έκφραση και μετά αυξάνεται κατά 1 το a και αποθηκεύεται. 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.

Στον Πίνακα 2.4 παρατίθενται οι σχετικές προτεραιότητες κάποιων τελεστών. Για τελεστές ίδιας προτεραιότητας, οι πράξεις εκτελούνται από αριστερά

προς τα δεξιά. Εξαίρεση αποτελούν οι τελεστές ανάθεσης και οι μοναδιαίοι. 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 μπορούν να επιβάλουν διαφορετική σειρά εκτέλεσης των πράξεων.

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 του **int** σε **double** και μετά η αντίστοιχη πράξη μεταξύ ποσοτήτων διπλής ακρίβειας. Ας σημειωθεί ότι ποσότητες τύπου μικρότερου από **int** (όπως **bool** και **char**) μετατρέπονται σε **int** και κατόπιν εκτελείται η πράξη, *ακόμα και όταν είναι ίδιες.* 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 ανάθεση σε ακέραιο ενός πραγματικού αριθμού. Σε τέτοια περίπτωση γίνεται στρογγυλοποίηση του πραγματικού στον αμέσως μικρότερο ακέραιο και κατόπιν γίνεται η ανάθεση. Επιπλέον, είναι δυνατόν η στρογγυλοποιημένη τιμή να μην είναι μέσα στα όρια τιμών της προς ανάθεση μεταβλητής οπότε η συμπεριφορά του προγράμματος (και όχι μόνο το αποτέλεσμα) είναι απροσδιόριστη.[5] Έτσι

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

Υπάρχουν περιπτώσεις που ο προγραμματιστής θέλει να καθορίζει συγκεκριμένη μετατροπή. Όπως αναφέρθηκε, ο τελεστής (/) εκτελεί κάτι διαφορετικό μεταξύ πραγματικών αριθμών από ότι αν είναι μεταξύ ακεραίων. Στον παρακάτω κώδικα που υπολογίζει την (πραγματική) μέση τιμή κάποιων ακεραίων είναι απαραίτητο να επιλεχθεί η δράση του /. Αυτό επιτυγχάνεται με τη ρητή μετατροπή των ακεραίων ορισμάτων του σε πραγματικούς με την εντολή **static_cast**:

```
int sum = 2 + 3 + 5;
int N = 3;

// Wrong value
double mean1 = sum / N;

// Correct value
```

[5] Ένας καλός compiler αναγνωρίζει και προειδοποιεί σε τέτοιες περιπτώσεις.

Σχετικές Προτεραιότητες Τελεστών της C++	
επιλογή μέλους κλάσης	`.`
επιλογή μέλους δείκτη σε κλάση	`->`
κλήση συνάρτησης	`()`
εξαγωγή τιμής από πίνακα	`[]`
αύξηση/μείωση (μετά τη μεταβλητή)	`++`, `--`
μέγεθος αντικειμένου	**`sizeof`**
μέγεθος αντικειμένου ή τύπου	**`sizeof`**`()`
αύξηση/μείωση (πριν τη μεταβλητή)	`++`, `--`
bitwise NOT	`~`
εξαγωγή διεύθυνσης	`&`
εξαγωγή τιμής από δείκτη	`*`
λογικό NOT	`!`
μοναδιαίο συν/πλην	`+`, `-`
πολλαπλασιασμός	`*`
διαίρεση (ή πηλίκο)	`/`
υπόλοιπο	`%`
άθροισμα	`+`
διαφορά	`-`
μετατόπιση δεξιά, αριστερά	`>>`, `<<`
μικρότερο	`<`
μικρότερο ή ίσο	`<=`
μεγαλύτερο	`>`
μεγαλύτερο ή ίσο	`>=`
ίσο	`==`
άνισο	`!=`
bitwise AND	`&`
bitwise XOR	`^`
bitwise OR	`|`
λογικό AND	`&&`
λογικό OR	`||`
τελεστής συνθήκης[4]	`?:`
απλή ανάθεση	`=`
πολλαπλασιασμός και ανάθεση	`*=`
διαίρεση και ανάθεση	`/=`
υπόλοιπο και ανάθεση	`%=`
άθροισμα και ανάθεση	`+=`
διαφορά και ανάθεση	`-=`
μετατόπιση αριστερά με ανάθεση	`<<=`
μετατόπιση δεξιά με ανάθεση	`>>=`
bitwise AND με ανάθεση	`&=`
bitwise XOR με ανάθεση	`^=`
bitwise OR με ανάθεση	`|=`
τελεστής κόμμα	`,`

Πίνακας 2.4: Σχετικές προτεραιότητες (κατά φθίνουσα σειρά) κάποιων τελεστών. Τελεστές στην ίδια θέση του Πίνακα έχουν ίδια προτεραιότητα.

[4] Για την προτεραιότητα του (`?:`) έναντι του (`=`) δείτε τη §3.1.2.

```
double mean2 = static_cast<double>(sum) / N;
```

Η σύνταξη του **static_cast** είναι:

```
static_cast<newtype>(variable);
```

Σε παλαιότερους κώδικες μπορείτε να δείτε τέτοιες μετατροπές με τη σύνταξη:

```
(newtype) variable.
```

Καλό είναι να μη χρησιμοποιείτε αυτή τη μορφή.

Μια άλλη περίπτωση που χρειάζεται ρητή μετατροπή σε συγκεκριμένο τύπο εμφανίζεται κατά την κλήση overloaded συνάρτησης, όταν η επιλογή της κατάλληλης υλοποίησης δεν είναι μονοσήμαντη. Π.χ. η συνάρτηση της τετραγωνικής ρίζας, `std::sqrt()` από το `<cmath>` δέχεται όρισμα τύπου **float**, **double**, **long double** αλλά όχι **int** με αποτέλεσμα να χρειάζεται μετατροπή όταν ζητούμε την ρίζα ακέραιου. Όπως θα δούμε στο §4.6, η μετατροπή που κάνει ο compiler σύμφωνα με τους κανόνες της C++ δεν είναι μονοσήμαντη, οπότε πρέπει να γίνει ρητά από τον προγραμματιστή:

```
#include <cmath>

int
main() {
  int p = 8;

  double riza = std::sqrt(p);
  // Error, ambiguous

  double r = std::sqrt(static_cast<double>(p));
  // Correct. Calls sqrt(double).
}
```

Ένα χαρακτηριστικό της C++ είναι ότι μια εντολή ανάθεσης ή μια εντολή σύγκρισης έχει η ίδια κάποια τιμή που μπορεί να ανατεθεί σε κάποια ποσότητα ή να μετέχει σε λογικές συνθήκες, π.χ.

```
int a;
int b;

b = a = 3;
//First a = 3; then b = 3;

bool cond;

cond = b < 5;
// First check b < 5; true. Then cond = true.
```

2.7.1 Άλλοι τελεστές

Τελεστής sizeof

Ο τελεστής **sizeof** δέχεται ως όρισμα μια ποσότητα ή ένα τύπο και επιστρέφει το μέγεθός τους σε bytes.[6] 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.

Η δράση του **sizeof** σε πίνακα (array) επιστρέφει το μέγεθος σε bytes ολόκληρου του πίνακα, 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 πίνακα.

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

Ο επιστρεφόμενος από το **sizeof** τύπος είναι ο `std::size_t`, ένας απρόσημος ακέραιος τύπος που ορίζεται στο `<cstddef>`. Ποσότητες τέτοιου τύπου είναι ιδιαίτερα κατάλληλες για διαστάσεις πινάκων καθώς και ως δείκτες για να προσπελαύνουμε τα στοιχεία τους.

Τελεστής κόμμα (,)

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

Τελεστές bit

Υπάρχουν τελεστές που αντιμετωπίζουν τα ορίσματά τους ως σύνολο bits σε σειρά, δηλαδή ως ακολουθίες από 0 ή 1. Η δράση τους ελέγχει ή θέτει την τιμή

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

του κάθε bit χωριστά. Τα ορίσματά τους είναι μεταβλητές με τύπο ακεραίου (**short int**, **int**, **long int**, **bool**, **char**) και **enum**, **signed** ή **unsigned**. Οι τελεστές παρουσιάζονται στον Πίνακα 2.5.

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 ανάθεση	`expr1 <<= expr2`
>>=	μετατόπιση δεξιά με ανάθεση	`expr1 >>= expr2`
&=	bitwise AND με ανάθεση	`expr1&=expr2`
^=	bitwise XOR με ανάθεση	`expr1 ^= expr2`
|=	bitwise OR με ανάθεση	`expr1 |= expr2`

Πίνακας 2.5: Τελεστές bit της C++.

Ο τελεστής ~ δρώντας σε ένα ακέραιο, επιστρέφει νέο ακέραιο έχοντας μετατρέψει τα 0 του αρχικού σε 1 και αντίστροφα.

Οι τελεστές <<, >> μετατοπίζουν τα bits του αριστερού τους ορίσματος κατά τόσες θέσεις όσες ορίζει το δεξί τους όρισμα. Τα επιπλέον bits χάνονται. Ο τελεστής << γεμίζει τις κενές θέσεις με 0 ενώ ο >> κάνει το ίδιο αν το αριστερό όρισμα είναι **unsigned**.

Οι τελεστές &, ^, | επιστρέφουν ακέραιο με bit pattern που προκύπτει αν εκτελεστεί το AND, XOR, OR αντίστοιχα στα ζεύγη bit των ορισμάτων τους.

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 `bitset<>` και την εξειδίκευση της κλάσης `std::vector<>` για **bool** που διευκολύνουν πολύ αυτό το σκοπό, και βέβαια είναι προτιμότερες.

Η προφανής εναλλακτική λύση ενός πίνακα με ποσότητες τύπου **bool**, παρόλο που είναι πιο εύχρηστη, κάνει πολύ μεγάλη σπατάλη μνήμης καθώς για την αποθήκευση ενός bit δεσμεύει τουλάχιστον ένα byte, (§2.2.4).

2.8 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 μία συνάρτηση που να επιλύει ένα γραμμικό σύστημα εξισώσεων με το πολύ φυσικό όνομα solve, όμως όταν αργότερα θελήσουμε να επεκτείνουμε τη συλλογή συ-

ναρτήσεών μας δε θα μπορέσουμε να χρησιμοποιήσουμε αυτό το όνομα για τη συνάρτηση 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;
......
}
```

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: `onoma::a`. Στο εσωτερικό του συγκεκριμένου **namespace** που ορίστηκε χρησιμοποιούμε απλά το όνομά της (a). Με αυτόν 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 **namespace** ακολουθεί τους κανόνες ονοματοδοσίας της C++ (§2.1.2).

Όλες οι ποσότητες που παρέχονται από την Standard Library (STL) ορίζονται στο **namespace** `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 **namespace** 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>

// "std::" needed here
typedef std::complex<double> complex;

int
main() {
  using namespace std;

  // "std::" not needed here
  complex a(2.0,3.0);
  complex<double> b(1.0);

  // In the following cout "std::" is not needed
  cout << a;
  cout << '\n';
  cout << b;
  cout << '\n';
```

```
  return 0;
}
```

Εναλλακτικά (και καλύτερα), μπορούμε να ορίσουμε συγκεκριμένο αντικείμενο για το οποίο δεν είναι απαραίτητο το όνομα του **namespace** με εντολή σαν κι αυτή:

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

Χρήσιμος επίσης είναι και ο ανώνυμος χώρος ονομάτων: **namespace** `{...}`. Οι ποσότητες που ορίζονται σε αυτόν χρησιμοποιούνται απ' ευθείας με το όνομά τους μόνο στο αρχείο που ορίζεται ο ανώνυμος **namespace**. Δεν μπορούν να χρησιμοποιηθούν σε άλλο αρχείο και, επομένως, να "συγκρουστούν" με ποσότητες ορισμένες εκεί.

Η συνάρτηση `main()` δεν επιτρέπεται να ανήκει σε κανένα άλλο **namespace** εκτός από το global.

2.9 Ροές (streams)

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 streams, τα `std::cerr` και `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 του κώδικα. Τα παραπάνω streams ορίζονται στο header `<iostream>`.

2.9.1 Ροές Αρχείων

Εκτός των προκαθορισμένων streams μπορούμε να ορίσουμε streams συνδεδεμένα με αρχεία. Στο header `<fstream>` ορίζονται οι τύποι `std::ifstream` και `std::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*.

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. 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");`.

2.9.2 Ροές Strings

Θα αναφερθούμε τώρα σε ροές (streams) συνδεδεμένες με σειρές χαρακτήρων, λόγω της χρησιμότητάς τους σε συγκεκριμένη εφαρμογή. Καθώς δεν έχουμε αναφέρει για C++ strings, θα τις περιγράψουμε περιληπτικά.

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
```

Θα δούμε παρακάτω ότι η μετατροπή του C++ string σε C string γίνεται καλώντας τη συνάρτηση-μέλος `c_str()`. Επομένως, για να ανοίξουμε ένα αρχείο με όνομα "filename_3.dat", π.χ. για έξοδο, χρησιμοποιώντας το `ostringstream os`, δίνουμε την παρακάτω εντολή:

```
std::ofstream outstr(os.str().c_str());
```

Αντικείμενο τύπου `std::istringstream` χρησιμοποιείται για το "διάβασμα" τιμών από το `string` με το οποίο συνδέεται όπως ακριβώς θα γινόταν από αρχείο:

```
#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'
}
```

2.9.3 Είσοδος-έξοδος δεδομένων

Η εκτύπωση των θεμελιωδών τύπων, καθώς και όσων τύπων παρέχονται από την STL, σε ροή (stream) συνδεδεμένη με τα standard output, standard error, με αρχείο ή με `string`, γίνεται με τον τελεστή `<<`:

```
std::cout << 'a';
std::cerr << "Wrong␣value␣of␣b";
```

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;
std::cout << "␣einai:␣" << a+b << '\n';
```

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 (§2.2.1) 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 manipulator `std::boolalpha` που ορίζεται στο header `<ios>`:

```
std::cout << std::boolalpha << (3==2);
```

Η μετατροπή σε ακέραιο επανέρχεται αφού "τυπώσουμε" το `std::noboolalpha`.

Το "διάβασμα" από το πληκτρολόγιο ή από αρχείο, μιας ποσότητας τύπου **bool** γίνεται μόνο με τις ακέραιες τιμές.

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 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`**). Διάβασμα και απόρριψη μιας γραμμής στη ροή εισόδου is μπορεί επομένως να γίνει με τον κώδικα (δες §3.2.1)

  ```
  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);`.

2.9.4 Διαμορφώσεις

Έχουμε ήδη δει δύο διαμορφωτές (manipulators) που παρέχει η C++, τους `boolalpha` και `noboolalpha`, που καθορίζουν τη μορφή εκτύπωσης ποσοτήτων ενός συγκεκριμένου τύπου, του τύπου **`bool`**. Στο header `<ios>` υπάρχουν επίσης οι

- (no)skipws: (δεν) αγνοούνται οι κενοί χαρακτήρες στην ανάγνωση.
- (no)showpos: (δεν) τυπώνεται το πρόσημο + σε θετικούς αριθμούς.
- (no)showpoint: (δεν) τυπώνονται 0 στο τέλος ενός δεκαδικού αριθμού.
- scientific: οι πραγματικοί τυπώνονται με τη μορφή *d.ddddddDdd*
- fixed: οι πραγματικοί τυπώνονται με τη μορφή *dddd.dd*
- left/right: προκαλεί αριστερή/δεξιά στοίχιση στην εκτύπωση.

Στο header `<iomanip>` υπάρχουν επίσης

- ο `setprecision()` που ως όρισμα δέχεται την ακρίβεια (το πλήθος των σημαντικών ψηφίων) με την οποία θέλουμε να τυπώνονται οι πραγματικοί αριθμοί. Η προεπιλεγμένη τιμή είναι 6.
- ο `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 `setfill()` που ως όρισμα δέχεται τον χαρακτήρα με τον οποίο θα γεμίσουν οι κενές θέσεις αν το `setw()` όρισε περισσότερες από την ακρίβεια. Ο προεπιλεγμένος χαρακτήρας είναι ο `' '`.

Όλοι οι manipulators ανήκουν στο **namespace** `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';
}
```

Ενναλακτικά, μπορούμε να χρησιμοποιήσουμε τις *συναρτήσεις-μέλη* κάθε ροής, `precision`, `width` και `fill`, αντί για τους διαμορφωτές `setprecision`, `setw` και `setfill` αντίστοιχα:

```
#include <iostream>

int
main() {
    std::cout.precision(9);
    std::cout << 256.123456789987 << '\n';
}
```

2.10 Ασκήσεις

1. Γράψτε, μεταγλωττίστε και εκτελέστε τους κώδικες των παραδειγμάτων. Όσοι δεν αποτελούν πλήρη προγράμματα C++ συμπληρώστε τους ώστε να γίνουν.

2. Ποια από τα παρακάτω είναι έγκυρα ονόματα ποσοτήτων στη C++ ;

   (α') const

   (β') john's

   (γ') y+z12

   (δ') 1stclass

   (ε') xyz123

   (ϛ') George

   (ζ) ThisIsALongOne

   (η') To_onoma_mou

   (θ') two-way

   (ι') proto

   (ια') _posotita

3. Ένα πρόγραμμα C++ μπορεί να διαβάσει τιμές από το stream

   - `std::cin`
   - `std::cout`
   - `std::cerr`
   - `std::clog`

4. Μια γραμμή κώδικα στη C++ είναι σχόλιο αν αρχίζει με

   - `//`
   - `{`
   - **`;`**
   - **`#`**

5. Τι λάθη υπάρχουν στις ακόλουθες δηλώσεις;

```
int a = 2, b = 3, c = 4;
double x = y = 5;
bool equal = b = 3;
int d = 'd';
int e = "e";
```

6. Ποια είναι τα όρια των ακεραίων αριθμών στο δικό σας compiler; Ποια είναι τα όρια και η ακρίβεια των πραγματικών;

7. Ποιο είναι το αποτέλεσμα του παρακάτω προγράμματος στο δικό σας compiler; Ποιο θα αναμένατε να είναι;

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

int
main() {
  int a = std::numeric_limits<int>::min() - 10;
  int b = std::numeric_limits<int>::max() + 10;
  std::cout << a << "␣␣␣" << b << '\n';

  return 0;
}
```

8. Γράψτε δηλώσεις σταθερών ποσοτήτων κατάλληλου τύπου και τιμής για να παραστήσουν

   (α΄) Τη σταθερά του Euler.

   (β΄) Το πλήθος των ημερών της εβδομάδας.

   (γ΄) Τα ονόματα των μηνών του έτους.

   (δ΄) Τον πληθυσμό του Τόκιο.

   (ε΄) Τον πληθυσμό της Γης.

9. Υπολογίστε όσες από τις παρακάτω εκφράσεις της C++ είναι σωστές και εξηγήστε γιατί οι υπόλοιπες είναι λάθος:

   (α΄) `37/(5*2)`

   (β΄) `37/5/2`

   (γ΄) `37(5/2)`

   (δ΄) `37%(5%2)`

   (ε΄) `37%5%2`

   (ϛ΄) `37-5-2`

   (ζ΄) `(37-5)2`

10. Υπολογίστε τις παρακάτω εκφράσεις όταν m = 24 και n = 8:

    (α΄) `m-8-n`

    (β΄) `m==n-3`

    (γ΄) `m%n`

    (δ΄) `m%n++`

    (ε΄) `m%++n`

    (ϛ΄) `++m=n--`

    (ζ΄) `m=n=3`

    (η΄) `m+=n-=2`

(θ') `m*=++n`

(ι') `m+=--n==m--`

11. Υπολογίστε τις τιμές των παρακάτω εκφράσεων:

(α') `(50 - 5 * 4) / 10 - 7`

(β') `12 + -5 * 2 + 6 / 3`

(γ') `1 + 2 + 3 + 4 + 5 + 1 * 2 * 3 / 4 * 5`

(δ') `3 > 7`

(ε') `(3 > 7) || (4 < 9)`

(ϛ') `135 == 100 + 35`

(ζ') `(true && false) || (true || false)`

(η') `(3 > 7) || ((10 < 9) == (3 == 8))`

(θ') `false || !(3 > 7)`

(ι') `3!=6`

(ια') `6 / 4 * 5 == 6 * 5 / 4`

12. Εκτιμήστε το αποτέλεσμα και τον τύπο του στις παρακάτω εκφράσεις

(α') `7 / 2`

(β') `7.0f / 2`

(γ') `((7 / 2) == (7.0 / 2))`

(δ') `((6 * 3) / 80) / (36 - 7 * 5)`

13. Γράψτε τέσσερις διαφορετικές εντολές της C++ με τις οποίες μειώνεται κατά 1 μία μεταβλητή n.

14. Γράψτε *μία* έκφραση που αφαιρεί το άθροισμα των μεταβλητών a, b από την c και κατόπιν αυξάνει την a.

15. Τι κάνουν οι εκφράσεις

(α') `++x+=+a++-+b++`

(β') `++x-=-a+b++`

(γ') `-x---a---b--`

16. Τι συμβαίνει όταν χρησιμοποιήσουμε μεταβλητή που δεν της έχει δοθεί τιμή; Δοκιμάστε να τυπώσετε μια τέτοια.

17. Γράψτε εκφράσεις της C++ για τον υπολογισμό των παρακάτω ποσοτήτων:

(α') Το εμβαδόν κύκλου με διάμετρο d.

(β') Η περιφέρεια κύκλου με ακτίνα r.

(γ΄) Ο όγκος κυλίνδρου με διάμετρο d και ύψος h.

(δ΄) Το μήκος της υποτείνουσας c ορθογώνιου τριγώνου συναρτήσει των μηκών a,b των άλλων πλευρών.

18. Δηλώστε με παρενθέσεις τη σειρά εκτέλεσης των πράξεων στις παρακάτω εκφράσεις:

(α΄) `a==b||a==c&&c<5`

(β΄) `c=x!=0`

(γ΄) `0<=i<6`

(δ΄) `a=b==c++`

19. Τρεις ακέραιοι αριθμοί a, b, c που ικανοποιούν τη σχέση $a^2 + b^2 = c^2$ αποτελούν μία Πυθαγόρεια τριάδα. Από δύο τυχαίους ακεραίους m, n με $m > n$, μπορούμε να παράγουμε μια τέτοια τριάδα σχηματίζοντας τους αριθμούς $m^2 - n^2$, $2mn$, $m^2 + n^2$.[7] Γράψτε ένα πρόγραμμα C++ που να διαβάζει δύο ακεραίους και να τυπώνει την αντίστοιχη Πυθαγόρεια τριάδα.

20. 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.
  (ϛ΄) $r_b = 2r_2 + 4r_3 + 6r_4$.
  (ζ΄) r_5 = υπόλοιπο διαίρεσης του r_b με το 7.
  (η΄) $r_c = r_4 + r_5$.
- Το rc είναι το πλήθος των ημερών μετά την 3η Απριλίου του συγκεκριμένου έτους, που μεσολαβούν μέχρι το Πάσχα.

Γράψτε σε κώδικα C++ τον παραπάνω αλγόριθμο. Φροντίστε να τυπώνει κατάλληλο μήνυμα και τον αριθμό rc αφού τον υπολογίσει.

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

21. 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 sec`.

Κεφάλαιο 3

Εντολές Ελέγχου–Βρόχοι

3.1 Εντολές Ελέγχου

3.1.1 if

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

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

Εάν η συνθήκη (condition) είναι αληθής ή μπορεί να μετατραπεί σε αληθή, εκτελείται το block των εντολών που περικλείεται μεταξύ των πρώτων {} (block A). Αν η συνθήκη είναι ψευδής, τότε εκτελείται το block των εντολών μετά το **else** (block B).

Το κάθε block μπορεί να αποτελείται από καμμία, μία, ή περισσότερες εντολές. Στην περίπτωση που περιλαμβάνει μία μόνο εντολή μπορούν να παραληφθούν τα άγκιστρα ({}) που την περικλείουν. Π.χ.

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

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

```
if (condition) {
  ...
}
```

Το κάθε block μπορεί να περιλαμβάνει οποιεσδήποτε άλλες εντολές και βέβαια άλλο **if**. Σημειώστε ότι η δομή **if** `(condition) {...}` **else** `{...}` θεωρείται μία εντολή. Παρατηρήστε ότι η στοίχιση των εντολών σε κάθε block υποδηλώνει ότι βρίσκονται στο "εσωτερικό" μιας (σύνθετης) εντολής. Δεν είναι υποχρεωτική αλλά διευκολύνει τον προγραμματιστή 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 το i είναι 0 και δεν ισχύει το (`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;
```

3.1.2 ?:

Ο τελεστής (`?:`) είναι ένας ιδιαίτερα διαδεδομένος ιδιωματισμός της C++. Η εντολή

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

ισοδυναμεί με

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

Γενικότερα, η έκφραση

(condition ? έκφρασηΑ : έκφρασηΒ)

έχει την τιμή "έκφρασηΑ" όταν η συνθήκη condition είναι αληθής, ενώ έχει την τιμή "έκφρασηΒ" όταν η συνθήκη είναι ψευδής.

Οι παρενθέσεις που περιβάλουν τον τελεστή (`?:`) με τα ορίσματά του στο συγκεκριμένο παράδειγμα δεν είναι απαραίτητες, βοηθούν όμως στην κατανόηση του κώδικα. Καθώς για το συγκεκριμένο τελεστή *δεν μπορεί να καθοριστεί μονοσήμαντα* η προτεραιότητά του σε σχέση με τον (=), πρέπει να τις χρησιμοποιούμε για να εκτελείται η επιδιωκόμενη πράξη. Εναλλακτικά, μπορούμε να εφαρμόζουμε τον εξής κανόνα: 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)
```

3.1.3 switch

Η δομή **switch** αποτελεί μια πιο κομψή και 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 (**char**, **int**, (**short int**, **long int**) με τις **signed** και **unsigned** παραλλαγές τους) ή **enum**. Αυτή η τιμή μπορεί να προέρχεται από μεταβλητή τέτοιου τύπου ή να είναι η τιμή συνάρτησης που επιστρέφει τέτοιο τύπο ή κάποια μεταβλητή ή συνάρτηση τύπου που μπορεί να μετατραπεί απ' ευθείας σε ακέραιο τύπο.

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

Κατά την εκτέλεση, η τιμή ελέγχου i συγκρίνεται με κάθε μία 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 διακοπεί με* ***break*** *(ή άλλες εντολές αλλαγής της ροής π.χ.* ***goto***, ***return***, ***throw***). Μετά το **break** η εκτέλεση συνεχίζει με την πρώτη εντολή μετά το καταληκτικό άγκιστρο της δομής **switch**. Αν δεν βρεθεί η τιμή της στις value1, value2,. . . ,valueN εκτελείται το block των εντολών του **default**, αν υπάρχει. Αλλιώς, η εκτέλεση συνεχίζει μετά το } 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 διακόπτεται η εκτέλεση του προγράμματος. Μπορούμε να γράψουμε το εξής

```
#include <iostream>
int
main() {
    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';
}
```

3.1.4 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.

```
labelname : statement;
```

Από άλλη θέση στο σώμα της *ίδιας* συνάρτησης μπορούμε να συνεχίσουμε την εκτέλεση με αυτήν την εντολή με τη χρήση της **goto**:

```
goto labelname;
```

Με το **goto** δεν επιτρέπεται να "υπερπηδήσουμε" έναν ορισμό μεταβλητής καθώς δε θα μπορεί να χρησιμοποιηθεί η συγκεκριμένη μεταβλητή στο σημείο του κώδικα που θα μεταβούμε.

Η χρήση της **goto** πρέπει να αποφεύγεται. Η C++ παρέχει τις κατάλληλες εντολές ελέγχου και επαναληπτικές δομές (§3.2) καθιστώντας την **goto** περιττή. Μοναδική περίπτωση που είναι προτιμότερη από τις εναλλακτικές λύσεις, εμφανίζεται όταν επιθυμούμε έξοδο από πολλαπλούς βρόχους, ο ένας μέσα στον άλλο. Ακόμη και τότε, η "μετακίνηση" με την **goto** προς προηγούμενο σημείο του κώδικα πρέπει να αποφεύγεται.

3.1.5 assert()

Μία ιδιότυπη συνάρτηση ελέγχου παρέχεται στη C++ με τη συμπερίληψη του header `<cassert>`· πρόκειται για τη macro[1] συνάρτηση `assert()`. Η κλήση της γίνεται ως εξής:

```
assert(integer_number);
```

Αν ο `integer_number` είναι 0, η εκτέλεση του προγράμματος διακόπτεται, εκτός εάν έχει οριστεί στον προεπεξεργαστή το όνομα `NDEBUG` πριν τη συμπερίληψη του header. Όταν διακοπεί το πρόγραμμα, τυπώνεται το αρχείο και η γραμμή στην οποία βρίσκεται η κλήση τής `assert()` καθώς και το όρισμά της.

Συνήθως, η `assert()` καλείται κατά τη διαδικασία του debugging, με όρισμα κάποια λογική συνθήκη η οποία μετατρέπεται σε 0 αν είναι **false** (σύμφωνα με τους γνωστούς κανόνες, §2.2.1), προκαλώντας διακοπή της εκτέλεσης. Έτσι π.χ. αν ο κώδικας περιλαμβάνει την εντολή `assert(N<10);`, το πρόγραμμα σταματά αν δεν ισχύει το `(N<10)`. Η παραπάνω συμπεριφορά δεν εμφανίζεται αν έχει δοθεί προς τον προεπεξεργαστή η εντολή **#define** `NDEBUG` ή δοθεί αντίστοιχη εντολή κατά τη μεταγλώττιση.

Η συγκεκριμένη συνάρτηση καλείται *χωρίς* το πρόθεμα `std::` (δεν ανήκει στο χώρο ονομάτων std) καθώς είναι macro συνάρτηση.

[1] έκφραση που παράγεται από τον προεπεξεργαστή της C++ και δεν είναι ενσωματωμένη στη γλώσσα.

3.2 Βρόχοι

3.2.1 while

Η δομή **while** είναι η πιο απλή υλοποίηση βρόχου, δηλαδή επαναληπτικής διαδικασίας. Συντάσσεται ως εξής:

```
while (condition) {
  ...
}
```

Κατά την εκτέλεση της δομής **while**:

1. Ελέγχεται η συνθήκη `condition`·
   - Αν είναι ψευδής, η ροή συνεχίζει με την πρώτη εντολή μετά τη δομή.
   - Αν είναι αληθής εκτελείται το block εντολών μεταξύ των αγκίστρων {}.
2. Αν εκτελέστηκε το block χωρίς αλλαγή ροής (με **break**, **return**, **goto**, **throw**, ...), επαναλαμβάνεται το βήμα 1 (έλεγχος της συνθήκης). Η τιμή της συνθήκης μπορεί να έχει μεταβληθεί στο προηγούμενο βήμα.

3.2.2 do while

Η δομή **do while** είναι μια παραλλαγή του **while** (§3.2.1) στην οποία το σώμα του βρόχου εκτελείται τουλάχιστον μία φορά. Η σύνταξή της είναι:

```
do {
  ...
} while (condition);
```

Κατά την εκτέλεση της δομής **do while**:

1. Εκτελείται το block εντολών μεταξύ των αγκίστρων {}.
2. Αν δεν υπήρξε αλλαγή ροής (με **break**, **return**, **goto**, **throw**,...), ελέγχεται η συνθήκη `condition`.
   - Αν είναι ψευδής, η ροή συνεχίζει με την πρώτη εντολή μετά τη δομή.
   - Αν είναι αληθής επαναλαμβάνεται το βήμα 1 (εκτέλεση του block).

3.2.3 for

Στις περισσότερες γλώσσες προγραμματισμού οι αντίστοιχες με τη δομή **for** χρησιμοποιούνται στην υλοποίηση βρόχων με εκ των προτέρων γνωστό το πλήθος των επαναλήψεων. Στη C++ το **for** είναι πολύ πιο γενικό. Η σύνταξή του είναι:

```
for (initial_statement ; condition ; final_statement) {
  ...
}
```

Η δομή **for** εκτελείται ως εξής:

1. Εκτελείται η αρχική εντολή `initial_statement`. Αυτή η εντολή μπορεί να είναι και δήλωση μεταβλητής (ή ακόμα και σύνολο εντολών χωριζόμενων με τον τελεστή κόμμα (,), §2.7.1).

2. Ελέγχεται η συνθήκη `condition`.

   - Αν είναι ψευδής, η ροή συνεχίζει με την πρώτη εντολή μετά τη δομή **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 το `final_statement`.

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.

Ένα ή περισσότερα από τα initial_statement, condition, final_statement μπορεί να λείπει. Αν λείπει η συνθήκη (condition) οι έλεγχοί της στην εκτέλεση του **for** θεωρούνται αληθείς.

Προσέξτε ότι αν το initial_statement περιλαμβάνει δήλωση μεταβλητής, η εμβέλεια αυτής περιορίζεται στη δομή **for**, μέχρι, δηλαδή, το καταληκτικό }.

Παράδειγμα:
Η ανάθεση στα 10 πρώτα στοιχεία του πίνακα a των τιμών $0, 10, 20, \ldots 90$ μπορεί να γίνει ως εξής:

```
for (std::size_t i = 0; i < 10; ++i)
    a[i] = i*10;
```

Η μεταβλητή i δεν μπορεί να χρησιμοποιηθεί μετά το βρόχο. Αν είναι επιθυμητό κάτι τέτοιο, η δήλωση `std::size_t i;` πρέπει να γίνει πριν το **for**:

```
std::size_t i;
for (i = 0; i < 10; ++i)
    a[i] = i*10;

// here i is 10
```

3.2.4 continue

Η εντολή **continue** μπορεί να εμφανίζεται μόνο στο σώμα βρόχου **for**, **while**, **do while**. Η εκτέλεσή της μεταφέρει τη ροή του προγράμματος στο καταληκτικό άγκιστρο του βρόχου, από όπου συνεχίζει η εκτέλεσή του.

3.2.5 break

Η εντολή **break** μπορεί να εμφανίζεται μόνο στο σώμα βρόχου **while**, **do while**, **for**, ή εντολής ελέγχου **switch**. Η εκτέλεσή της προκαλεί έξοδο από τη περικλείουσα δομή, μεταφέροντας τη ροή στην αμέσως επόμενη εντολή από αυτή.

3.3 Γενικές Παρατηρήσεις

Κάθε block μπορεί να αποτελείται από καμμία, μία, ή περισσότερες εντολές. Στην περίπτωση που περιλαμβάνει μία μόνο εντολή μπορούν να παραληφθούν τα άγκιστρα ({}) που την περικλείουν.

Η συνθήκη στα **if** και **while** μπορεί να είναι ταυτόχρονα και δήλωση μεταβλητής με αρχική τιμή (ή σταθερής ποσότητας) αρκεί η τιμή της να μετατρέπεται σε λογική τιμή. Η εμβέλειά της περιορίζεται στη δομή.

Παράδειγμα:

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

Η δήλωση του j έχει εμβέλεια μόνο στη δομή **if**. Η τιμή της συνθήκης είναι μη μηδενική (3) άρα είναι **true**.

3.4 Ασκήσεις

1. (α΄) Πώς εκτελείται το

   ```
   for (;;) {
     ...
   }
   ```

   (β΄) Γράψτε κώδικα που να ισοδυναμεί με το **for** χρησιμοποιώντας τη δομή **while**. Μπορείτε να το κάνετε με το **do while**;

2. Γράψτε το παρακάτω πρόγραμμα:

   ```
   #include <iostream>

   int
   main() {
     int j = 3;

     if (j = 4) {
       std::cout << "j␣is␣" << j << '\n';
       std::cout << "Should␣not␣print␣this!!!\n";
     }
   }
   ```

   Τι κάνει; Έχει λάθος; Μεταγλωττίστε το και εκτελέστε το. Μπορείτε να εξηγήσετε τι συμβαίνει;

3. Γράψτε προγράμματα C++ ώστε

   (α΄) να υπολογίσετε το παραγοντικό (n!) ενός ακεραίου.

   (β΄) να τυπώσετε τους N πρώτους όρους της *ακολουθίας Fibonacci*[2].

   $$f(n+2) = f(n+1) + f(n), \quad n \geq 0, \qquad f(0) = 0, f(1) = 1 \ .$$

   (γ΄) να βρείτε το μέγιστο κοινό διαιρέτη δύο ακεραίων αριθμών. Χρησιμοποιήστε τον *αλγόριθμο του Ευκλείδη*[3].

   (δ΄) να επιβεβαιώσετε το *Θεώρημα των τεσσάρων τετραγώνων του Lagrange*[4]: κάθε θετικός ακέραιος μπορεί να γραφεί ως άθροισμα τεσσάρων (ή λιγότερων) τετραγώνων ακεραίων αριθμών.

   Υπόδειξη: Για δεδομένο ακέραιο n, δοκιμάστε όλους τους συνδυασμούς των ακεραίων a,b,c,d με $0 \leq a, b, c, d \leq \sqrt{n}$. Τυπώστε στην οθόνη τις τετράδες που ικανοποιούν τη σχέση $n = a^2 + b^2 + c^2 + d^2$.

4. Γράψτε κώδικες C++ που να υπολογίζουν

   - το e^x εφαρμόζοντας τη σχέση

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

   - 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^{-12}.

5. Γράψτε κώδικα που να τυπώνει στην οθόνη 8 τυχαίους αριθμούς. Η συνάρτηση `std::rand()` στο `<cstdlib>` καλούμενη χωρίς όρισμα επιστρέφει ένα ψευδοτυχαίο *ακέραιο* αριθμό από 0 έως και `RAND_MAX`. Πόσο είναι η προκαθορισμένη σταθερή `RAND_MAX` στο δικό σας compiler; Βρείτε τι κάνει η συνάρτηση `std::srand()`.[5]

[2] `http://oeis.org/A000045`

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

[4] `http://mathworld.wolfram.com/LagrangesFour-SquareTheorem.html`

[5] σε συστήματα UNIX η εντολή man srand θα σας βοηθήσει.

6. (α΄) Δημιουργήστε ένα αρχείο με 1000 τυχαίους *ακεραίους* στο διάστημα $[a, b]$ με $a = -20$, $b = 20$. Χρησιμοποιείστε ανάλογο κώδικα με τον παρακάτω (προσπαθήστε να τον καταλάβετε!)

   ```
   int r = a + (b-a+1) * (std::rand()/(RAND_MAX+1.0));
   ```

   Υπόδειξη: Υπολογίστε τα όρια τιμών που μπορεί να πάρει ο ακέραιος r στον τύπο, ξεκινώντας από τη σχέση 0 ≤`std::rand()`≤`RAND_MAX`. Γιατί διαιρούμε με `RAND_MAX`+1.0 και όχι με `RAND_MAX`+1;

   (β΄) Γράψτε ένα πρόγραμμα C++ που να διαβάζει το αρχείο και να τυπώνει στην οθόνη πόσους θετικούς, αρνητικούς και ίσους με το 0 αριθμούς περιέχει.

7. 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 $n \le m \le k$).

   Δοκιμάστε το για τους αριθμούς 16 ([0, 1, 5] ή [0, 3, 4] ή ...), 104 ([6, 7, 10] ή ...), και 111 ([1, 10, 10] ή [0, 9, 11] ή ...).

8. Γράψτε πρόγραμμα C++ που

   (α΄) να ελέγχει αν ένας ακέραιος αριθμός είναι πρώτος.

   (β΄) να τυπώνει όλους τους πρώτους αριθμούς μέχρι έναν ακέραιο N. ("κόσκινο του Ερατοσθένη": διαγράφονται τα πολλαπλάσια των 2,3,5,...).

   (γ΄) 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.

9. Γράψτε κώδικα που να υλοποιεί τον *δυαδικό αλγόριθμο* για τον πολλαπλασιασμό δύο ακεραίων. Σύμφωνα με αυτόν τον αλγόριθμο, σχηματίζουμε δύο στήλες με επικεφαλής τους δύο αριθμούς. Οι αριθμοί της πρώτης στήλης συμπληρώνονται υπολογίζοντας το ακέραιο μέρος (πηλίκο) της διαίρεσης του αμέσως προηγούμενου με το 2. Οι αριθμοί της δεύτερης στήλης προκύπτουν από το διπλασιασμό του αμέσως προηγούμενου. Οι διαιρέσεις/πολλαπλασιασμοί στις στήλες σταματούν όταν στην πρώτη εμφανιστεί ο αριθμός 1. Το γινόμενο των δύο αρχικών αριθμών είναι το άθροισμα των αριθμών της δεύτερης στήλης που αντιστοιχούν σε περιττό αριθμό στην πρώτη στήλη.

[6] http://oeis.org/A047948

10. Γράψτε κώδικα που να ζητά από το χρήστη τρεις πραγματικούς αριθμούς a, b, c και να τυπώνει με 12 σημαντικά ψηφία τις ρίζες του πολυωνύμου $ax^2 + bx + c$. Χρησιμοποιείστε τον τύπο

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} .$$

Η συνάρτηση της τετραγωνικής ρίζας στη C++ είναι η `std::sqrt()` στο `<cmath>`. Δοκιμάστε τον για διάφορες τριάδες αριθμών. Προσέξτε να κάνετε διερεύνηση για διάφορες τιμές των συντελεστών και να τυπώνετε τις ρίζες ή κατάλληλο μήνυμα (ποιές είναι οι ρίζες όταν $a = 0$ ή όταν $a = b = 0$;). Λάβετε υπόψη την περίπτωση που έχετε μιγαδικά αποτελέσματα (όταν $b^2 < 4ac$).

11. Γράψτε ένα πρόγραμμα που να υπολογίζει τις ρίζες ενός γενικού πολυωνύμου 4ου βαθμού (με, εν γένει, μιγαδικούς συντελεστές). Εφαρμόστε τον *αλγόριθμο του Ferrari*[7].

12. Δημιουργήστε ένα πίνακα $M \times N$ στον οποίο K στοιχεία θα έχουν την τιμή 1 και τα υπόλοιπα θα είναι 0. Τα στοιχεία με τιμή 1 θα είναι επιλεγμένα με τυχαίο τρόπο.

    (α') Τυπώστε στην οθόνη (δοκιμάστε με $M = 20, N = 60, K = 400$) τον πίνακα αυτόν κατά σειρές, βάζοντας `'x'` για τα μη μηδενικά στοιχεία και `'o'` για τα υπόλοιπα.

    (β') Τυπώστε τον σε αρχείο με κατάληξη .ppbm με την εξής διαμόρφωση (δοκιμάστε με $M = N = 512, K = 150000$):

    - η πρώτη γραμμή του αρχείου να γράφει: P1.
    - η δεύτερη να γράφει τις διαστάσεις: πλάτος ύψος (δηλ. τους δύο αριθμούς με κενό μεταξύ τους).
    - να ακολουθούν *σε ξεχωριστές σειρές* οι χαρακτήρες '1' ή '0'· αυτοί αντιπροσωπεύουν τα στοιχεία του πίνακα *κατά γραμμές* (μη μηδενικά και μηδενικά αντίστοιχα).

    Η διαμόρφωση αυτή αποτελεί ένα αρχείο τύπου plain pbm (portable bitmap) που μπορούμε να το δούμε με προγράμματα απεικόνισης.

13. Γράψτε κώδικα που να υλοποιεί τον πολλαπλασιασμό δύο γενικών πινάκων A ($M \times N$) και B ($N \times P$). Θεωρήστε ότι $M = 10$, $N = 20$, $P = 30$ και ότι τα στοιχεία των πινάκων A, B βρίσκονται στα αρχεία "matA.dat" και "matB.dat". Αν θέλετε, μπορείτε να δημιουργήσετε αυτά τα αρχεία ώστε να περιέχουν τυχαίους αριθμούς. Να τυπώσετε στο αρχείο "matC.dat" το γινόμενο $C = A \cdot B$.

[7] `http://en.wikipedia.org/wiki/Quartic_function#Ferrari.27s_solution`

14. Γράψτε ένα πρόγραμμα C++ το οποίο να επιλύει ένα σύστημα γραμμικών εξισώσεων $Ax = B$ με τη *μέθοδο απαλοιφής του Gauss*[8]. Το διδιάστατο τετραγωνικό πίνακα A και το διάνυσμα B να τα διαβάζει από αρχείο.

15. Γράψτε ένα πρόγραμμα C++ το οποίο να υπολογίζει και να τυπώνει τον αντίστροφο ενός τετραγωνικού πίνακα A με τη *μέθοδο απαλοιφής Gauss-Jordan*[9].

16. (α') Γράψτε κώδικα που να τυπώνει σε αρχείο 30 ισαπέχουσες τιμές της μαθηματικής συνάρτησης $f(x) = x(x^2 + 5\sin(x))$ στο διάστημα $[-5 : 5]$. Η συνάρτηση του ημιτόνου στη C++ είναι η `std::sin()` στο `<cmath>`.

    (β') Γράψτε πρόγραμμα C++ το οποίο να διαβάζει το αρχείο που προκύπτει και να υπολογίζει το ολοκλήρωμα της $f(x)$ στο διάστημα $[-5 : 5]$ με κατάλληλη εφαρμογή

    - του κανόνα τραπεζίου: για δύο διαδοχικά σημεία x_1, x_2 ισχύει
    $$\int_{x_1}^{x_2} f(x)\mathrm{d}x \approx \frac{1}{2}\delta x\,(f_1 + f_2) \ .$$
    - του κανόνα Simpson: για τρία ισαπέχοντα διαδοχικά σημεία x_1, x_2, x_3 ισχύει
    $$\int_{x_1}^{x_3} f(x)\mathrm{d}x \approx \frac{1}{3}\delta x\,(f_1 + 4f_2 + f_3) \ .$$
    - του κανόνα Boole: για πέντε ισαπέχοντα διαδοχικά σημεία x_1, x_2, x_3, x_4, x_5 ισχύει
    $$\int_{x_1}^{x_5} f(x)\mathrm{d}x \approx \frac{2}{45}\delta x\,(7f_1 + 32f_2 + 12f_3 + 32f_4 + 7f_5) \ .$$
    - του κανόνα Durand: για n ισαπέχοντα διαδοχικά σημεία x_1, x_2, ..., x_n ισχύει
    $$\int_{x_1}^{x_n} f(x)\mathrm{d}x \approx \delta x\left(\frac{2}{5}f_1 + \frac{11}{10}f_2 + f_3 + \cdots + f_{n-2} + \frac{11}{10}f_{n-1} + \frac{2}{5}f_n\right) \ .$$

    Στους παραπάνω τύπους $f_i \equiv f(x_i)$ και δx η απόσταση διαδοχικών σημείων. Συγκρίνετε τα αποτελέσματα με την ακριβή τιμή $[10\sin(5) - 50\cos(5)]$. Αλλάξτε το πλήθος των σημείων και δείτε πώς μεταβάλεται η ακρίβεια.

[8] `http://mathworld.wolfram.com/GaussianElimination.html`
[9] `http://mathworld.wolfram.com/Gauss-JordanElimination.html`

17. **Spline Fit**. Έστω ένα σύνολο n σημείων (x_i, y_i). Ανά δύο σημεία περνούμε πολυώνυμα τρίτου βαθμού και κατασκευάζουμε συνολικά $2(n-1)$ εξισώσεις για τους συντελεστές τους. Επιπλέον, ζητούμε οι πρώτες και δεύτερες παράγωγοι στα σημεία "επαφής" των πολυωνύμων να είναι ίδιες (άλλες $2(n-2)$ εξισώσεις). Αν η καμπύλη είναι ανοιχτή απαιτούμε οι δεύτερες παράγωγοι στα άκρα να είναι 0 (2 εξισώσεις), ενώ αν είναι κλειστή ζητούμε να υπάρχει συνέχεια στο "άκρο", δηλ. οι πρώτες και δεύτερες παράγωγοι να είναι ίσες (2 εξισώσεις). Συνολικά έχουμε $4n-4$ εξισώσεις για ισάριθμους αγνώστους. Με την επίλυση του συστήματος γνωρίζουμε πλήρως μια προσεγγιστική συνεχή καμπύλη που περνά από τα δεδομένα σημεία.

    (α') Να γράψετε πρόγραμμα που να κάνει προσέγγιση με spline μιας άγνωστης συνάρτησης που δίνεται ως σύνολο σημείων και να υπολογίζει την τιμή της σε οποιοδήποτε ενδιάμεσο σημείο.

    (β') Τροποποιήστε τον κώδικα ώστε να υπολογίζει προσεγγιστικά το ολοκλήρωμα της άγνωστης συνάρτησης.

    (γ') Τροποποιήστε τον κώδικα ώστε να υπολογίζει προσεγγιστικά την *πρώτη παράγωγο* της άγνωστης συνάρτησης σε οποιοδήποτε ενδιάμεσο σημείο.

18. Γράψτε ένα πρόγραμμα C++ που να υλοποιεί το *Game of Life*[10] του Dr. J. Conway. Αυτό προσομοιώνει την εξέλιξη ζωντανών οργανισμών βασιζόμενο σε συγκεκριμένους κανόνες:

    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 άσκηση 12β') ώστε να μπορείτε να τις δείτε όλες μαζί διαδοχικά[11].

[10] `http://www.math.com/students/wonders/life/life.html`

[11] Σε συστήματα UNIX, με εγκατεστημένο το πρόγραμμα imagemagick, η εντολή είναι animate *.ppbm

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).

19. 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.

20. Μία 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: 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 κάθε χρώματος. Τυπική τιμή είναι το 255.
    - να ακολουθούν *σε ξεχωριστές σειρές* τα pixels της εικόνας *κατά γραμμές*. Σε κάθε σειρά του αρχείου θα αναγράφεται 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 αναπαριστάται 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. Σε κάθε συνιστώσα RGB μπορούμε γενικά να έχουμε οποιαδήποτε τιμή μεταξύ 0 και K ώστε να παράγουμε όλα τα χρώματα.

    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, χρησιμοποιώντας την παραπάνω διαμόρφωση.

21. 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 $p(z)$. 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+i0 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 στο αρχείο "newton.pppm" χρησιμοποιώντας τη διαμόρφωση plain ppm (δείτε την άσκηση 20).

Η εικόνα στο "newton.pppm" είναι ένα Newton fractal.

Κεφάλαιο 4

Συναρτήσεις

4.1 Εισαγωγή

Στα προηγούμενα κεφάλαια έχουν παρουσιαστεί κάποιες από τις βασικές εντολές–δομές 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, σε μεγάλο βαθμό, είναι 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 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++.

Προτού παρουσιάσουμε λεπτομέρειες για τη σύνταξη και χρήση συναρτήσεων, θα δούμε δύο χαρακτηριστικά της C++ που βρίσκουν ιδιαίτερη εφαρμογή στους ορισμούς τους.

4.2 Αναφορά

Η αναφορά (reference) είναι 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 όνομα (π.χ. r) με το a στη μεταβλητή αυτή:

```
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.

Στο συγκεκριμένο παράδειγμα, οποιαδήποτε αλλαγή της τιμής του a εμφανίζεται αυτόματα και στο r και το αντίστροφο:

```
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

int d;
int & r = d;    // Error
```

Σημειώστε ότι αν μια μεταβλητή έχει οριστεί ως **const**, πρέπει και οι αναφορές σε αυτή να ορίζονται αντίστοιχα:

```
int const a = 5;

int & r1 = a;   // Error

int const & r2 = a;
// Correct. Value of a cannot change through r2.
```

Αν η δήλωση του r1 ήταν αποδεκτή, θα μπορούσαμε να μεταβάλουμε την τιμή του a, μιας ποσότητας **const**, μέσω του r1.

Η χρήση μιας αναφοράς, εκτός του ορισμού συναρτήσεων, 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. Παραδείγματος χάριν, είδαμε ότι στο <limits> ορίζεται η ποσότητα std::numeric_limits<**double**>::digits10. Ένα πιο εύχρηστο όνομα για αυτή ορίζεται και χρησιμοποιείται στο παρακάτω πρόγραμμα:

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

int
main() {
  int const & digits = std::numeric_limits<double>::digits10;

  std::cout << digits << '\n';
}
```

Ας δούμε μία λίγο διαφορετική σύνταξη της αναφοράς, ιδιαίτερα χρήσιμη κατά την κλήση συναρτήσεων. Μπορεί να οριστεί αναφορά σε αριθμητική σταθερά:

```
double const & r = 4.0;
```

Μία τέτοια δήλωση ισοδυναμεί με τον ορισμό μιας προσωρινής σταθερής ποσότητας με αρχική τιμή την αριθμητική σταθερά· κατόπιν, η αναφορά ορίζεται σε σχέση με αυτή την ποσότητα.

Η χρήση αναφορών για τα ορίσματα και την επιστρεφόμενη τιμή από συναρτήσεις θα παρουσιαστεί παρακάτω.

4.3 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. Αυτός ο αριθμός μπορεί να ανατεθεί σε ένα *δείκτη* σε τύπο ίδιο (ή ισοδύναμο) 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 ακέραιο, με όνομα p, και με ταυτόχρονη ανάθεση 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. 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 ανατεθεί σε άλλο δείκτη *ίδιου τύπου* ή σε δείκτη σε **void**[1]:

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

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

Η τελευταία εντολή από τις παρακάτω

```
int a = 5;
int * p = &a;
void * t = p;
```

ορίζει ένα δείκτη σε **void** και του αποδίδει ως 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.* Γενικά, 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.

Προσέξτε ότι ένας δείκτης που δεν του αποδοθεί αρχική τιμή—είτε άλλος δείκτης είτε διεύθυνση—δείχνει σε τυχαία περιοχή μνήμης. Η δράση του (*) δεν είναι λάθος αλλά θα δώσει μία *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.*

[1] Ανάθεση σε **void** * δεν επιτρέπεται για δείκτη σε συνάρτηση ή σε μέλος κλάσης.

Σε ένα οποιοδήποτε δείκτη μπορεί να γίνει απόδοση της τιμής 0. Το 0 δεν αποτελεί αριθμό θέσης μνήμης και, επομένως, η ανάθεση αυτή υποδηλώνει ότι ο δείκτης δε δείχνει σε κανένα αντικείμενο. Σε τέτοιο δείκτη (null pointer) η δράση του (*) δεν επιτρέπεται (προκαλεί λάθος κατά την εκτέλεση αλλά όχι κατά τη μεταγλώττιση του προγράμματος). Η σύγκριση ενός άγνωστου δείκτη (π.χ. όρισμα συνάρτησης) με το 0 (ή η μετατροπή του σε λογική τιμή, §2.2.1) πρέπει να προηγείται οποιασδήποτε απόπειρας προσπέλασης της θέσης μνήμης στην οποία θεωρούμε ότι δείχνει. Προσέξτε ότι σε αυτό το σημείο υπάρχει μία βασική διαφορά με την αναφορά: ένας δείκτης μπορεί να μη συνδέεται με κανένα αντικείμενο ενώ, αντίθετα, μία αναφορά είναι οπωσδήποτε συνδεδεμένη με κάποια ποσότητα.

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;
```

- 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 μεταβλητή στην οποία δείχνει.

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

Παράδειγμα:

```
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
```

Αν p είναι ένας δείκτης σε ποσότητα κάποιου τύπου, είναι προφανές ότι όλοι οι τελεστές που η δράση τους έχει νόημα για αυτόν 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 **sizeof**`(T)` μετά ή πριν την αρχική του τιμή. 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)`. Προφανώς, πρόσθεση δεικτών δεν έχει νόημα, ενώ αντίθετα, η διαφορά δεικτών ίδιου τύπου (μόνο!) δίνει το πλήθος των θέσεων μνήμης που μεσολαβεί.[2] Οι μόνες πράξεις που μπορούμε να κάνουμε σε δείκτη σε **void** είναι η ανάθεση δείκτη ίδιου ή άλλου τύπου, η ρητή μετατροπή σε άλλο τύπο, ο έλεγχος για ισότητα και ανισότητα (με άλλο **void** `*`).

Η αριθμητική δεικτών είναι χρήσιμη στην περίπτωση που αναθέσουμε τη διεύθυνση ενός στοιχείου πίνακα σε ένα δείκτη. Τότε, η μετακίνηση κατά πολλαπλάσια του μεγέθους του τύπου μας μεταφέρει σε επόμενο ή προηγούμενο στοιχείο του πίνακα:

```
int a[10];

int * p = &a[3];

int * q = p + 2;    // q == &a[5]
```

Μάλιστα, αν ισχύει

```
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 στον πίνακα· αυτό αποτελεί ένα πολύ συνηθισμένο λάθος για αρχάριους προγραμματιστές.

Σημειώστε ότι το όνομα ενός πίνακα 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 οποιου-

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

δήποτε στοιχείου ενός πίνακα *καθώς και τη διεύθυνση του πρώτου στοιχείου μετά το τέλος του.*

Παράδειγμα:
Τι κάνει ο παρακάτω κώδικας:

```
double b[10];

double * p = b;

for (int i = 0; i < 10; ++i)
   *p++ = 1.0;
```

Ισοδύναμος με τον παραπάνω κώδικα είναι ο

```
double b[10];

for (double * p = b; p != b+10; ++p)
   *p = 1.0;
```

Αναφέραμε στο §2.6.2 ότι η προσπέλαση ενός μέλους, π.χ. member, μιας δομής (**struct**) και, κατ' επέκταση, κλάσης, π.χ. a, γίνεται ως εξής:

```
a.member
```

Στην περίπτωση που έχουμε δείκτη στην κλάση, pa, η προσπέλαση γίνεται [λαμβάνοντας υπόψη τις σχετικές προτεραιότητες των (*) και (.), Πίνακας 2.4]

```
(*pa).member
```

Τέτοια έκφραση χρησιμοποιείται πολύ συχνά στη C++ και γι' αυτό έχει εισαχθεί ειδικός συμβολισμός, τελείως ισοδύναμος με τον παραπάνω:

```
pa->member
```

4.4 Ορισμός και κλήση συνάρτησης

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 §4.7, 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.

4.4.1 Ορισμός και δήλωση

Ο ορισμός μιας συνάρτησης έχει την ακόλουθη γενική μορφή:

```
τύπος_επιστρεφόμενης_ποσότητας
όνομα(τύπος_ορίσματος_Α όρισμαΑ, τύπος_ορίσματος_Β όρισμαΒ,. . . )
{
// κώδικας
}
```

Ο *τύπος επιστρεφόμενης ποσότητας* μπορεί να είναι **void**· υποδηλώνεται έτσι ότι δεν επιστρέφεται τιμή. Εναλλακτικά, μπορεί να είναι οποιοσδήποτε απλός ή σύνθετος τύπος εκτός από πίνακας και συνάρτηση (επιτρέπεται, όμως, να είναι δείκτης σε πίνακα ή συνάρτηση). 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 μίας συνάρτησης έναν μονοδιάστατο πίνακα, χρησιμοποιούμε την ακόλουθη μορφή:

```
τύπος_στοιχείων όνομα_πίνακα [ ]
```

Προσέξτε ότι μεταξύ των αγκυλών έχουμε κενό. Στην πραγματικότητα, σε μια τέτοια δήλωση ορίσματος, "περνά" ως όρισμα ένας δείκτης στο αρχικό του στοιχείο του πίνακα. 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 πίνακα οπότε, αν χρειάζεται, πρέπει να δοθεί με ξεχωριστό όρισμα. Οι containers της STL που θα δούμε στο Κεφάλαιο 5 δεν έχουν τέτοιο πρόβλημα.

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 αριθμό τους) και τον τύπο επιστρεφόμενης ποσότητας ώστε να ελέγξει αν γίνεται σωστά η κλήση. Τα στοιχεία αυτά τα λαμβάνει από τη *δήλωση της συνάρτησης*. Η δήλωση που αντιστοιχεί στον παραπάνω γενικό ορισμό είναι ακριβώς η ίδια 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,. . . );
```

Στην παραπάνω δήλωση μπορούν να παραληφθούν τα ονόματα των ορισμάτων.

Προσέξτε ότι ο ορισμός μίας συνάρτησης δεν μπορεί να γίνει στο σώμα άλλης συνάρτησης. Η δήλωσή της όμως επιτρέπεται να εμφανίζεται οπουδήποτε μπορούμε να ορίσουμε μια μεταβλητή.

Τέλος, ένα ιδιαίτερα χρήσιμο χαρακτηριστικό 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 μία 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.

4.4.2 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()` η οποία δε χρειάζεται **return**: αν παραλείπεται, θεωρείται ότι δόθηκε ως τελευταία εκτελέσιμη γραμμή η εντολή **return** `0;`.

Μια συνάρτηση που δεν επιστρέφει τιμή (δηλαδή "επιστρέφει" **void**) μπορεί 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 αποθηκεύουμε το αποτέλεσμα της συνάρτησης σε οποιοδήποτε σημείο παραχθεί· κατόπιν, μπορούμε να την "επιστρέψουμε" από ένα σημείο, στο τέλος του σώματος της συνάρτησης.

4.4.3 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 για κάποια από αυτά, οπότε μπορούν να είναι λιγότερες.

Μία συνάρτηση που επιστρέφει τιμή μπορεί να χρησιμοποιηθεί όπου θα χρησιμοποιούσαμε σταθερή ποσότητα του ίδιου τύπου με την επιστρεφόμενη τιμή, π.χ. σε ανάθεση, σύνθετη έκφραση, εκτύπωση κλπ. Αν δεν επιθυμούμε να χρησιμοποιήσουμε το αποτέλεσμα μιας συνάρτησης έχουμε τη δυνατότητα να μην το κάνουμε. Μπορούμε να έχουμε ως αυτόνομη εντολή την κλήση οποιασδήποτε συνάρτησης· προσέξτε την κλήση της `read` στο επόμενο παράδειγμα:

Παράδειγμα:

```
#include <iostream>
#include <fstream>

double
func(double a, double b);
// declaration. The definition is elsewhere.

int
read(double & a, char const fname[]) { // definition
  std::ifstream file(fname);
  file >> a;

  return 0; // All ok
}

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). 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. Οι νέες μεταβλητές έχουν 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) {  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 εξάγουμε 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 όρισμα x είναι συνώνυμο του z· οποιαδήποτε αλλαγή στην τιμή της 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` παραμένει και μετά την επιστροφή της συνάρτησης.

Το παραπάνω σημαίνει ότι, αν το όρισμα είναι πίνακας ή ισοδύναμα, δείκτης σε πίνακα, δεν μπορούμε να το αλλάξουμε· όμως, τα στοιχεία του πίνακα μπορούν να μεταβληθούν. Επομένως, 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[], std::size_t N) {
  for (std::size_t i(0); i < N; ++i)
      std::cout << a[i] << '\n';
}
```

δηλώνουμε ότι τα στοιχεία του πίνακα a, μέσα στο σώμα της συνάρτησης, είναι σταθερά. Αν τυχόν προσπαθούσαμε να τροποποιήσουμε κάποιο από αυτά, η μεταγλώττιση θα σταματούσε.

Παρατηρήστε ότι είναι περιττό να ορίσουμε 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 §4.3.

Συνοψίζοντας, έχουμε τις ακόλουθες περιπτώσεις για τη δήλωση ορισμάτων, ως προς τη δυνατότητα να εξάγουμε αλλαγές στην τιμή τους:

- 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 πίνακα `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 πίνακα `a` δεν μπορεί να μεταβληθεί, αλλά ούτε και τα στοιχεία του

```
void f(double const a[]);
// argument cannot change, a[i] cannot change
```

4.4.4 Οργάνωση κώδικα

Συνήθως, οι δηλώσεις των συναρτήσεων που καλεί ένα τμήμα κώδικα συγκεντρώνονται σε ένα ή περισσότερους headers, αρχεία με συνήθη κατάληξη .h,[3] 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 ("). 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,

[3] εξαρτώμενη από τον compiler.

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 όρισμαB
> **max**: αν όρισμαA < όρισμαB επίστρεψε το όρισμαB αλλιώς επίστρεψε το όρισμαA

Για την καλύτερη οργάνωση του κώδικα μπορούμε να έχουμε στο αρχείο utilities.h τις δηλώσεις τους (π.χ. για ορίσματα τύπου **double**)

```
// utilities.h

// declarations
double
dmin(double a, double b);

double
dmax(double a, double b);
```

και στο αρχείο utilities.cc τους ορισμούς τους

```
// utilities.cc
#include "utilities.h"  // Not necessary but good practice

// definitions
double
dmin(double a, double b) { return a<b ? a : b; }

double
dmax(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␣" << dmax(a,b) << '\n';
  std::cout << "Min␣is␣" << dmin(a,b) << '\n';
}
```

- κάνουμε ξεχωριστό compile στο utilities.cc 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.

Αναδρομική (recursive) κλήση

Στη C++ επιτρέπεται σε μια συνάρτηση να καλεί τον εαυτό της. Βέβαια, αυτή η κλήση πρέπει να γίνεται υπό κάποια συνθήκη, αλλιώς δεν θα επιστραφεί ποτέ τιμή.

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 $n-1$). Ο ορισμός που δόθηκε παραπάνω για το παραγοντικό εκφράζεται σε συνάρτηση C++ ως εξής

```
std::size_t
factorial(std::size_t n) {
  std::size_t result;

  if (n > 0)
  result = n * factorial(n-1);

  if (n == 0)
  result = 1;

  return result;
}
```

Στη συνάρτηση αυτή έχουμε αγνοήσει 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 τελεστή (`?:`), §3.1.2:

```
std::size_t
factorial(std::size_t n) {
  return (n > 0 ? n * factorial(n-1) : 1);
}
```

4.4.5 Δείκτης σε συνάρτηση

Ένας χρήσιμος τύπος, ειδικά για όρισμα συνάρτησης, είναι ο δείκτης σε συνάρτηση.

Ας υποθέσουμε ότι θέλουμε να γράψουμε κώδικα με τον οποίο να σχεδιάζεται μία 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), Πίνακας 2.4. Μετά από τέτοια δήλωση, η μεταβλητή όνομα_δείκτη μπορεί να πάρει "τιμή" με ανάθεση 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;

double y = f(x);

double 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 **typedef** 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

typedef int (*ftype)(double x, double y);   // type

ftype g = func;   // declaration with assignment
```

4.5 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 τη στιγμή της κλήσης της*:

```
int main(int argc, char* argv[]) {........}
```

Ισοδύναμος με αυτόν τον τρόπο δήλωσης (δες §4.4.3) 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 πίνακας 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 **namespace** `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

  double x = std::atof(argv[4]);
  // x gets the value of the fourth argument

  std::ifstream filein(argv[2]);
  // open input file. Name is given in argv[2].

  std::ofstream fileout(argv[3]);
  // open output file. Name is given in argv[3].

  // ..........
}
```

4.6 overloading

Ας εξετάσουμε την περίπτωση που θέλουμε να γράψουμε συναρτήσεις που να εκτελούν πολλαπλασιασμό αριθμού με μονοδιάστατο πίνακα (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.

4.7 Συναρτήσεις template

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,[4] είτε είναι ενσωματωμένοι (**int**, **float**,...), είτε πρόκειται για τύπους που ορίζει ο προγραμματιστής (κλάσεις, Κεφάλαιο 6). 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) {
  float const temp = b;
  b = a;
  a = temp;
}

void
swap(double & a, double & b) {
  double const temp = b;
  b = a;
  a = temp;
}
```

[4] έχουμε ήδη, την `std::swap()` στο `<algorithm>`.

```
...
...
```

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 έναν τύπο. Με αυτόν 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):

```
template <typename X, typename Y, ... >.
```
[5]

Η κλήση μιας συνάρτησης template 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);
```

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)`.

Εκτός από τύπος, μια παράμετρος ενός template μπορεί να είναι σταθερή ακέραια ποσότητα (**int**, **char**, **bool**,...) γνωστή κατά τη μεταγλώττιση, ή

[5] Από αβλεψία το Standard του 1998 για τη C++ δεν επιτρέπει προεπιλεγμένες "τιμές" παραμέτρων σε template συνάρτησης, όπως, π.χ. το **template** <**typename** X,**typename** Y = **double**>. Σε επόμενη αναθεώρηση θα επιτρέπεται και μάλιστα δε θα είναι απαραίτητο οι προεπιλεγμένες παράμετροι να βρίσκονται στο τέλος της λίστας.

enum.[6]

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 στην περίπτωση που περιλαμβάνεται μια συνάρτηση template παρουσιάζει ιδιαιτερότητα. Πρέπει να περιλαμβάνεται στο header όχι μόνο η δήλωση αλλά και ο ορισμός της συνάρτησης template.

4.7.1 Εξειδίκευση

Στην περίπτωση που ο γενικός αλγόριθμος που προκύπτει από ένα template δε μας ικανοποιεί (π.χ. ως προς την ταχύτητα ή τον αλγόριθμο) για κάποιο συγκεκριμένο σύνολο παραμέτρων, μπορούμε να δηλώσουμε προς τον compiler ότι πρέπει να χρησιμοποιεί άλλη ρουτίνα όποτε χρειαστεί να παράγει 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 με το υπόδειγμα για τη `swap()` με τον εξής κώδικα:

```
template<>
void
swap(int & x, int & 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.

[6] ή δείκτης σε συνάρτηση και αναφορά ή δείκτης σε σταθερή ποσότητα με εξωτερική σύνδεση.

4.8 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 απ' ευθείας 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 επιστροφής. Παραδείγματος χάριν, μια συνάρτηση που βρίσκει το μεγαλύτερο δύο ακεραίων 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()`.

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 τη δήλωσή της.

4.9 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 την 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). 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 (ελευθερώνεται η αντίστοιχη μνήμη) μόνο όταν ολοκληρωθεί το πρόγραμμα[7].

4.10 Μαθηματικές συναρτήσεις της C++

Κάθε υλοποίηση της C++ παρέχει ορισμένες μαθηματικές συναρτήσεις. Οι δηλώσεις των παρακάτω βρίσκονται στο header `<cmath>` και ορίζονται με το ίδιο όνομα για αριθμούς τύπου **float**, **double**, **long double**. Στον Πίνακα 4.1 παρατίθενται οι δηλώσεις για **double**. Για ιστορικούς λόγους, κάποιες μαθηματικές συναρτήσεις και οι συνοδευτικές τους δομές δηλώνονται στο `<cstdlib>`. Ορισμένες συναρτήσεις της C που έχουν αντικατασταθεί από ισοδύναμες της C++ δεν παρουσιάζονται. Όλες οι αναφερόμενες συναρτήσεις και δομές δηλώνονται στο **namespace** `std`.

Προσέξτε ότι οι περισσότερες μαθηματικές συναρτήσεις δεν ορίζονται για ακέραιο. Αυτόματη επιλογή από τον compiler δε γίνεται γιατί οι μετατροπές ακεραίου σε καθένα από τους τρεις πραγματικούς τύπους είναι ισοδύναμες. Έτσι, πρέπει να γίνεται ρητή μετατροπή σε ένα από τους πραγματικούς τύπους από τον προγραμματιστή κατά την κλήση, αν το όρισμα είναι ακέραιος.

Στην περίπτωση που δοθεί όρισμα εκτός των επιτρεπόμενων τιμών στις παραπάνω μαθηματικές συναρτήσεις, η ποσότητα `errno` από το `<cerrno>` αποκτά την τιμή `EDOM`. Αν το αποτέλεσμα είναι εκτός των ορίων, η `errno` γίνεται `ERANGE`:

```
#include <cerrno>
#include <cmath>
#include <limits>
#include <iostream>

int main() {
  errno = 0; // clear error. No error code is 0.

  std::sqrt(-1.0);   // here errno becomes EDOM.

  if (errno == EDOM)
```

[7] αρκεί να μην διακοπεί το πρόγραμμα με την `std::abort()`.

Συνάρτηση	Επιστρεφόμενη τιμή	Παρατηρήσεις
	`<cmath>`	
`double abs(double)`	Απόλυτη τιμή του ορίσματος.	
`double ceil(double)`	Η αμέσως μεγαλύτερη από το όρισμα ακέραια τιμή ως πραγματικός.	
`double floor(double)`	Η αμέσως μικρότερη από το όρισμα ακέραια τιμή ως πραγματικός.	
`double sqrt(double)`	Η τετραγωνική ρίζα του ορίσματος.	Το όρισμα πρέπει να είναι μη αρνητικό.
`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 pow(double x, int i)`	Ύψωση σε ακέραια δύναμη, x^i.	Πρέπει να ισχύει `i>0` αν `x=0`.
`double cos(double)`	Συνημίτονο.	Το όρισμα σε rad.
`double sin(double)`	Ημίτονο.	Το όρισμα σε rad.
`double tan(double)`	Εφαπτομένη.	Το όρισμα σε rad.
`double acos(double)`	Τόξο συνημιτόνου.	Το όρισμα στο $[-1:1]$, το αποτέλεσμα στο $[0:\pi]$ σε rad.
`double asin(double)`	Τόξο ημιτόνου.	Το όρισμα στο $[-1:1]$, το αποτέλεσμα στο $[-\pi/2:\pi/2]$ σε rad.
`double atan(double)`	Τόξο εφαπτομένης.	Το αποτέλεσμα στο $[-\pi/2:\pi/2]$ σε rad.
`double atan2(double x, double y)`	Τόξο εφαπτομένης $\arctan(x/y)$.	Τα πρόσημα των ορισμάτων καθορίζουν το τεταρτημόριο. Το αποτέλεσμα στο $[-\pi:\pi]$ σε rad.
`double cosh(double)`	Υπερβολικό συνημίτονο.	
`double sinh(double)`	Υπερβολικό ημίτονο.	
`double tanh(double)`	Υπερβολική εφαπτομένη.	
`double exp(double)`	Εκθετικό.	
`double log(double)`	Φυσικός λογάριθμος (ln).	Πρέπει το όρισμα να είναι θετικό.
`double log10(double)`	Δεκαδικός λογάριθμος (log).	Πρέπει το όρισμα να είναι θετικό.
`double modf(double d, double *p)`	Το δεκαδικό μέρος του d.	Το ακέραιο μέρος στο `*p`.
`double frexp(double d, int *p)`	Βρίσκει x στο $[0.5, 1)$ και y ώστε $d = x2^y$. Επιστρέφει το x.	Θέτει το y στο `*p`.
`double fmod(double d, double m)`	Υπόλοιπο της διαίρεσης πραγματικών d/m.	Η επιστρεφόμενη τιμή έχει το ίδιο πρόσημο με το d.
`double ldexp(double d, int i)`	$d2^i$.	
	`<cstdlib>`	
`int abs(int)`	Απόλυτη τιμή.	Είναι overloaded και για **`long int`**.
`div_t div(int n, int d)`	Πηλίκο και υπόλοιπο της διαίρεσης `n/d`.	Επιστρέφονται στα μέλη `quot`, `rem` ποσότητας τύπου `div_t`.
`ldiv_t div(long int n, long int d)`	Πηλίκο και υπόλοιπο της διαίρεσης `n/d`.	Επιστρέφονται στα μέλη `quot`, `rem` ποσότητας τύπου `ldiv_t`.

Πίνακας 4.1: Συναρτήσεις των `<cmath>` και `<cstdlib>`.

```
    std::cerr <<
    "argument␣out␣of␣domain␣of␣function.\n";

    std::pow(std::numeric_limits<double>::max(), 2.0);
    // here errno becomes ERANGE.

    if (errno == ERANGE)
    std::cerr << "Math␣result␣not␣representable.\n";
  }
```

Εναλλακτικά, 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.

4.11 Ασκήσεις

4.11.1 Αλγόριθμοι Αναζήτησης–Ταξινόμησης

Αναζήτηση

Γραμμική Ο αλγόριθμος της γραμμικής αναζήτησης στοιχείου σε ένα πίνακα ή, γενικότερα, μιας συλλογής στοιχείων, είναι ο προφανής: διατρέχουμε ένα-ένα τα στοιχεία έως ότου συναντήσουμε το ζητούμενο ή εξαντληθεί ο πίνακας.

Δυαδική Η δυαδική αναζήτηση προϋποθέτει ότι ο πίνακας *είναι ήδη ταξινομημένος*. Στον αλγόριθμο αυτό γίνεται αρχικά η επιλογή του μεσαίου στοιχείου (ή του πλησιέστερου στη μέση) και η σύγκρισή του με το ζητούμενο. Το αποτέλεσμα της σύγκρισης καθορίζει σε ποιο "μισό" τμήμα του πίνακα μπορεί να βρίσκεται το επιθυμητό στοιχείο. Επαναλαμβάνουμε τη διαδικασία για το συγκεκριμένο τμήμα (δηλαδή συγκρίνουμε το μεσαίο του νέου τμήματος με το ζητούμενο) έτσι ώστε διαδοχικά να περιορίζουμε το διάστημα που είναι πιθανό να υπάρχει το στοιχείο που αναζητούμε. Η διαδικασία σταματά όταν το διάστημα αυτό συρρικνωθεί τόσο ώστε να γνωρίζουμε με βεβαιότητα την ύπαρξη ή μη του ζητούμενου.

Ταξινόμηση

Bubble sort

Ο πιο απλός αλγόριθμος ταξινόμησης. Είναι τάξης $O(n^2)$ και έχει ως εξής:

1. Ξεκινώντας από το πρώτο στοιχείο της λίστας εισόδου, συγκρίνουμε διαδοχικά στοιχεία ανά δύο μέχρι το τελευταίο. Αν το πρώτο είναι μεγαλύτερο από το δεύτερο, αλλάζουμε τις θέσεις τους μεταξύ τους.

2. Επαναλαμβάνουμε τη διαδικασία μέχρι το προτελευταίο στοιχείο. Το τελευταίο είναι, από το προηγούμενο βήμα, το μεγαλύτερο.

3. Επαναλαμβάνουμε τη διαδικασία αγνοώντας κάθε φορά ένα λιγότερο στοιχείο από το τέλος έως ότου δεν υπάρχουν στοιχεία για σύγκριση.

Στο τέλος, 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!

Insertion sort

Είναι παρόμοιος αλγόριθμος με τον προηγούμενο. 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.

Quicksort

Είναι, υπό κατάλληλες συνθήκες, από τους πιο γρήγορους αλγόριθμους. Είναι τάξης $O(n^2)$ (στη χειρότερη περίπτωση της ήδη ταξινομημένης λίστας) έως (συνήθως) $O(n \log n)$. Υλοποιείται πολύ εύκολα με αναδρομική συνάρτηση, σύμφωνα με ένα από τους παρακάτω αλγόριθμους:

Quicksort με βοηθητικό πίνακα

1. Αν η λίστα στοιχείων δεν έχει κανένα ή έχει μόνο ένα στοιχείο, επιστρέφουμε (δεν χρειάζεται ταξινόμηση).

2. Επιλέγουμε τυχαία ένα στοιχείο της λίστας. Το αντιγράφουμε σε νέα λίστα.

3. Διατρέχουμε όλα τα υπόλοιπα στοιχεία της αρχικής λίστας. Όσα είναι μικρότερα από το επιλεγμένο, τα αντιγράφουμε στη *αρχή* της νέας λίστας. Όσα είναι μεγαλύτερα τα αντιγράφουμε στο *τέλος* της νέας λίστας.

4. Αντιγράφουμε τη νέα λίστα στην αρχική.

5. Εφαρμόζουμε την ίδια διαδικασία στις υπο-λίστες πριν και μετά το επιλεγμένο στοιχείο.

Quicksort χωρίς βοηθητικό πίνακα

1. Αν η λίστα στοιχείων δεν έχει κανένα ή έχει μόνο ένα στοιχείο, επιστρέφουμε (δεν χρειάζεται ταξινόμηση).

2. Επιλέγουμε τυχαία ένα στοιχείο της λίστας, ας υποθέσουμε το πρώτο.

3. Επιδιώκουμε να μεταφέρουμε στην αρχή της λίστας τα στοιχεία που είναι μικρότερα από το επιλεγμένο και στο τέλος της λίστας όσα είναι μεγαλύτερα ή ίσα από το επιλεγμένο.

   Διατρέχουμε τη λίστα με δύο δείκτες: ο ένας ξεκινά από την αρχή παραλείποντας το επιλεγμένο στοιχείο 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 του επιλεγμένου.

4. Αν ο πρώτος δείκτης δεν ξεπεράσει τον δεύτερο, εναλλάσσουμε τα στοιχεία στα οποία δείχνουν οι δύο δείκτες και συνεχίζουμε τη μετακίνηση των δεικτών.

5. Αν ο πρώτος δείκτης ξεπεράσει τον δεύτερο, έχουμε βρει τη θέση που πρέπει να μεταφερθεί το επιλεγμένο στοιχείο (είναι η θέση του δεύτερου δείκτη). Εναλλάσσουμε το επιλεγμένο στοιχείο με αυτό που δείχνει ο δεύτερος δείκτης.

6. Εφαρμόζουμε την ίδια διαδικασία στις υπο-λίστες πριν και μετά το επιλεγμένο στοιχείο (στη νέα του θέση).

Mergesort

Είναι από τους πιο γρήγορους αλγόριθμους. Η τάξη του είναι $O(n \log n)$. Σύμφωνα με τον mergesort:

1. Αν η λίστα εισόδου έχει 0 ή 1 στοιχείο, είναι ταξινομημένη. Αλλιώς:

2. Χωρίζουμε τη λίστα σε δύο περίπου ίσα μέρη.

3. Ταξινομούμε κάθε νέο τμήμα με ξεχωριστή εφαρμογή (κλήση) της mergesort.

4. Συγχωνεύουμε τις δύο ταξινομημένες λίστες με τέτοιο τρόπο ώστε η τελική να είναι επίσης ταξινομημένη.

4.11.2 Ασκήσεις

1. Γράψτε συνάρτηση που

   (α΄) να υπολογίζει το παραγοντικό ενός μικρού ακεραίου.

(β΄) να ελέγχει αν το (ακέραιο) όρισμά της είναι πρώτος. Τι επιλέξατε για τον τύπο της επιστρεφόμενης ποσότητας;

(γ΄) να εναλλάσσει τα πραγματικά ορίσματά της.

(δ΄) να υπολογίζει το μέσο όρο των στοιχείων ενός μονοδιάστατου πίνακα ακεραίων.

(ε΄) να βρίσκει το μέγιστο στοιχείο ενός μονοδιάστατου πίνακα ακεραίων.

(ϛ΄) να βρίσκει τη *θέση* του μέγιστου στοιχείου ενός μονοδιάστατου πίνακα ακεραίων.

(ζ΄) να επιστρέφει ένα τυχαίο ακέραιο σε διάστημα $[a, b]$ που θα προσδιορίζεται από τα ορίσματά της.

(η΄) να υπολογίζει το εσωτερικό γινόμενο δύο πραγματικών μονοδιάστατων πινάκων με ίδιο πλήθος στοιχείων, δηλαδή το $\sum_i a_i b_i$.

2. Γράψτε συνάρτηση που να υπολογίζει

(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 μια συγκεκριμένη τιμή που δίνεται ως όρισμα στη συνάρτηση.

3. Γράψτε συνάρτηση της C++ που να υπολογίζει την τιμή των πολυωνύμων Legendre, $P_\ell(x)$. Τα $P_\ell(x)$ με $x \in [-1, 1]$, αποτελούν ένα σύνολο πολυωνύμων με συγκεκριμένες ιδιότητες και πολλές εφαρμογές. Τα δύο πρώτα είναι $P_0(x) = 1$ και $P_1(x) = x$, ενώ για μεγαλύτερες τιμές του ℓ υπολογίζονται από την αναδρομική σχέση:

$$\ell P_\ell(x) = (2\ell - 1)xP_{\ell-1}(x) - (\ell - 1)P_{\ell-2}(x) \ .$$

4. Υλοποιείστε τη *γεννήτρια ψευδοτυχαίων αριθμών του Cliff Pickover*[8]:

$$X_{n+1} = |100(\ln(X_n) \pmod 1)|$$

με $X_0 = 0.1$.

5. Γράψτε αναδρομική συνάρτηση που να υπολογίζει την ορίζουσα ενός τετραγωνικού πίνακα A διάστασης 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 j στήλης.

6. Γράψτε κώδικα που να προσδιορίζει τη λύση γραμμικού συστήματος χρησιμοποιώντας τη *μέθοδο του Cramer*[10].

7. Η κβαντομηχανική αντιμετώπιση του ατόμου του Υδρογόνου καταλήγει στις ιδιοσυναρτήσεις (σε σφαιρικές συντεταγμένες)

$$\psi_{n\ell m}(r, \vartheta, \varphi) = R_{n\ell}(r) Y_{\ell m}(\vartheta, \varphi) \ .$$

Το γωνιακό τμήμα τους είναι οι *σφαιρικές αρμονικές*,

$$Y_{\ell m}(\vartheta, \varphi) = \sqrt{\frac{2\ell + 1}{4\pi} \frac{(\ell - m)!}{(\ell + m)!}} \, P_\ell^m(\cos\vartheta) e^{im\varphi} \ .$$

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] http://mathworld.wolfram.com/CliffRandomNumberGenerator.html
[9] http://mathworld.wolfram.com/DeterminantExpansionbyMinors.html
[10] 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, $Y_{\ell m}(\vartheta, \varphi)$.
- Δημιουργήστε ένα καρτεσιανό πλέγμα 50×100 σημείων στο επίπεδο $\vartheta - \varphi$ και υπολογίστε σε καθένα από αυτά τις τιμές των $Y_{\ell m}(\vartheta, \varphi)$. Τυπώστε στο αρχείο "ylm_data" τις τιμές $\sin\vartheta\cos\varphi$, $\sin\vartheta\sin\varphi$, $\cos\vartheta$, $Y_{\ell m}(\vartheta, \varphi)Y^*_{\ell m}(\vartheta, \varphi)$ (δηλαδή, ουσιαστικά, τα $x, y, z, \psi\psi^*$) για κάθε σημείο, με $\ell = 2$, $m = 0$ (δηλαδή, ένα από τα d-τροχιακά).

8. Γράψτε συνάρτηση που να υπολογίζει την τιμή των πολυωνύμων Hermite, $H_n(x)$, για κάποια τιμή του x. Τα πολυώνυμα αυτά ικανοποιούν την αναδρομική σχέση:

$$H_n(x) - 2xH_{n-1}(x) + 2(n-1)H_{n-2}(x) = 0 \ , \qquad n \geq 2 \ ,$$

με $H_0(x) = 1$, $H_1(x) = 2x$.

Η κβαντομηχανική αντιμετώπιση του μονοδιάστατου αρμονικού ταλαντωτή (μάζα m σε δυναμικό $V = kx^2/2$) καταλήγει στις ιδιοσυναρτήσεις (χωρικό τμήμα)

$$\psi_n(y) = \sqrt{\frac{1}{2^n n!\sqrt{\pi}}} H_n(y) \mathrm{e}^{-y^2/2} \ , \tag{4.1}$$

όπου $y = x\sqrt{\sqrt{km}/\hbar}$. Χρησιμοποιείστε τη συνάρτηση που γράψατε για τα πολυώνυμα Hermite για να υπολογίσετε την πυκνότητα πιθανότητας ($\psi\psi^*$) της κυματοσυνάρτησης (4.1). Θα γράψετε μια νέα συνάρτηση για αυτό 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 $\psi\psi^*$).

9. Γράψτε μια συνάρτηση C++ που να βρίσκει (αν υπάρχει) μία λύση της εξίσωσης $f(x) = 0$ στο διάστημα $[x_a, x_b]$ για οποιαδήποτε συνεχή συνάρτηση $f(x)$. Να χρησιμοποιήσετε:

(α') τη *μέθοδο διχοτόμησης*[11]. Με διαδοχικές διχοτομήσεις του διαστήματος που περικλείει τη ρίζα, τα άκρα πλησιάζουν σε απόσταση μικρότερη από την επιθυμητή ακρίβεια.

[11] http://mathworld.wolfram.com/Bisection.html

(β΄) τη *μέθοδο ψευδούς θέσης*[12]. Σε αυτή, η συνάρτηση προσεγγίζεται με πολυώνυμο πρώτου βαθμού, η ρίζα του οποίου αποτελεί το ένα άκρο του νέου διαστήματος ενώ το άλλο επιλέγεται από τα προηγούμενα άκρα ώστε το διάστημα να περικλείει τη ρίζα της $f(x)$. Διαδοχικές επαναλήψεις προσδιορίζουν τη ρίζα με τη ζητούμενη ακρίβεια.

(γ΄) τη *μέθοδο Brent*[13]. Σε αυτή, η συνάρτηση προσεγγίζεται με πολυώνυμο δεύτερου βαθμού που περνά από τα άκρα του διαστήματος και ένα τρίτο, αρχικά αυθαίρετο, σημείο. Η μία ρίζα του πολυωνύμου αποτελεί το ένα άκρο του νέου διαστήματος.

10. Γράψτε ένα πρόγραμμα C++ 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. 'x' ή 'o') 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, καταρχήν, 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|
```

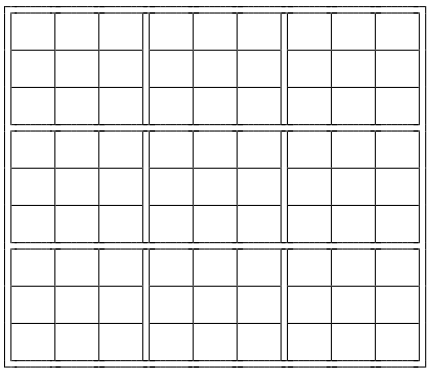
    Φροντίστε, επιπλέον, να περιγράφετε επαρκώς με σχόλια (τι κάνουν) τις ομάδες εντολών που χρησιμοποιείτε.

11. Γράψτε ένα πρόγραμμα C++ 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.

[12] http://mathworld.wolfram.com/MethodofFalsePosition.html
[13] http://mathworld.wolfram.com/BrentsMethod.html

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.

12. **Sudoku**. Γράψτε ένα πρόγραμμα C++ που να λύνει sudoku. Σε αυτή τη δραστηριότητα ο σκοπός είναι να γεμίσει ένα 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

13. **Το πρόβλημα των N βασιλισσών**. Σε μία σκακιέρα $N \times N$ θέλουμε να τοποθετήσουμε N βασίλισσες σε τέτοιες θέσεις ώστε να μην βρίσκονται ανά δύο στην ίδια γραμμή, στήλη ή διαγώνιο. Γράψτε πρόγραμμα που να υπολογίζει και να τυπώνει στην οθόνη μία 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 πίνακα ακεραίων με διάσταση N 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 υπολογίζει 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 στους δύο πίνακες 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.

- Όταν υπολογιστούν οι θέσεις όλων των βασιλισσών, τυπώστε στην οθόνη τη σκακιέρα (N αριθμοί σε N γραμμές, όπου υπάρχει βασίλισσα να έχει τιμή 1 αλλιώς να έχει τιμή 0.).

14. Έστω ένα πλέγμα 5×5 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.

15. 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.

16. 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

$$\begin{aligned} q &= \frac{x_2 - x_1}{x_1 - x_0} , \\ A &= q(f(x_2) - f(x_1)) - q^2 (f(x_1) - f(x_0)) , \\ B &= (q+1)(f(x_2) - f(x_1)) + A , \\ C &= (q+1) f(x_2) . \end{aligned}$$

(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 $B + \sqrt{B^2 - 4AC}$, $B - \sqrt{B^2 - 4AC}$.

(δ΄) Αν η νέα προσέγγιση είναι ικανοποιητική πηγαίνουμε στο βήμα 16ϛ΄.

(ε΄) Θέτουμε $x_0 \leftarrow x_1$, $x_1 \leftarrow x_2$, $x_2 \leftarrow x_3$. Επαναλαμβάνουμε τη διαδικασία από το βήμα 16β΄.

(ϛ΄) 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*.

Χρησιμοποιώντας τον αλγόριθμο Müller, βρείτε μία ρίζα της συνάρτησης $f(x) = x^3 - x + 1$.

17. Γράψτε σε ανεξάρτητες συναρτήσεις της C++ τους αλγόριθμους αναζήτησης που αναφέρθηκαν.

18. Γράψτε σε ανεξάρτητες συναρτήσεις της C++ τους αλγορίθμους ταξινόμησης που αναφέρθηκαν και συγκρίνετε τις επιδόσεις τους. Οι συναρτήσεις να δέχονται ως όρισμα ένα μονοδιάστατο πίνακα πραγματικών αριθμών. Φροντίστε όλες οι συναρτήσεις να δέχονται το ίδιο πλήθος, σειρά και τύπο ορισμάτων ώστε να μπορούν να χρησιμοποιηθούν εύκολα η μία στη θέση της άλλης.

19. Βελτιώστε τους κώδικες αναζήτησης και ταξινόμησης χρησιμοποιώντας τις συναρτήσεις `std::swap()` και `std::min()`, `std::max()` της STL, §5.1.3.

20. Τροποποιήστε τις συναρτήσεις αναζήτησης και ταξινόμησης ώστε να μπορούν να δεχθούν και μεταβλητή τύπου `std::vector<`**`double`**`>` (από το `<vector>`) ή, ακόμα, και οποιαδήποτε συλλογή αντικειμένων για την οποία ορίζονται οι τελεστές (<, μικρότερο) και (=, ανάθεση) καθώς και η αναφορά σε μέλος της συλλογής με δείκτη (`[]`, subscripting).

21. Χρησιμοποιήστε τη συνάρτηση `std::sort()` της STL από το `<algorithm>`.

22. Χρησιμοποιήστε τη συνάρτηση `dlasrt()` της LAPACK[14].

[14] Δείτε πώς στο Παράρτημα Β΄

Κεφάλαιο 5

Standard Library

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 6) και των υποδειγμάτων (templates, §4.7, §6.3), και αποτελούν τη Standard Library (STL).

Η STL έχει τρεις βασικές συνιστώσες:

- τους **containers**, δομές με κατάλληλα χαρακτηριστικά για την αποθήκευση και διαχείριση δεδομένων, η κάθε μία με διαφορετικές ιδιότητες. Υποκαθιστούν τους ενσωματωμένους πίνακες και επεκτείνουν σημαντικά τις περιορισμένες δυνατότητες που έχουν αυτοί. Μεταξύ άλλων περιλαμβάνονται containers που παρέχουν αυτόματη ταξινόμηση (π.χ. `set`, `map`) και ταχύτατη ανάκτηση δεδομένων είτε με ακέραιο αριθμητικό δείκτη (π.χ. `vector`, `deque`) είτε με δείκτη οποιουδήποτε τύπου (π.χ. `map`).

- τους **iterators**, ένα είδος δείκτη σε θέσεις στοιχείων ενός container. Οι iterators έχουν την ίδια μορφή για όλους τους containers με αποτέλεσμα να παρέχουν συγκεκριμένο, ενιαίο τρόπο για τη διαχείρισή τους. Μπορούμε να διατρέξουμε ένα container ή να προσπελάσουμε ένα στοιχείο του ανεξάρτητα από το πώς γίνεται σε χαμηλό επίπεδο η οργάνωση των δεδομένων σε αυτόν.

- τους **αλγορίθμους**, που υλοποιούν με πολύ αποτελεσματικό τρόπο συνήθη τμήματα κώδικα όπως ταξινόμηση και εύρεση ή αντικατάσταση στοιχείου με συγκεκριμένη τιμή, ανεξάρτητα από τον container που χρησιμοποιείται για την αποθήκευση. Η ανεξαρτησία αυτή εξασφαλίζεται με τη χρήση των iterators.

Επιπλέον, η STL περιλαμβάνει τους *προσαρμογείς* (adapters) και τα *αντικείμενα-συναρτήσεις* (function objects), στα οποία θα αναφερθούμε παρακάτω. Οι

προσαρμογείς των containers (container adapters) και το bitset, μέρη και αυτά της STL, δε θα παρουσιαστούν.

Μέχρι τώρα έχουμε συναντήσει και χρησιμοποιήσει αρκετά τμήματα της STL καθώς κάθε τι που παρέχεται από headers (π.χ. είσοδος/έξοδος δεδομένων, μαθηματικές συναρτήσεις, όρια αριθμών, μιγαδικός τύπος, κ.λ.π.) περιλαμβάνεται σε αυτή. Κάποια από αυτά υπάρχουν και στη C, αυτούσια ή παρόμοια. Σε αυτό το κεφάλαιο θα δούμε κυρίως τα νέα χαρακτηριστικά που προσθέτει η STL.

Αρχικά θα παρουσιάσουμε ορισμένες βοηθητικές δομές και σχετικές έννοιες της γλώσσας, και ακολούθως θα αναφερθούμε στους containers και στους αλγορίθμους που παρέχονται για τη διαχείρισή τους.

5.1 Βοηθητικές Δομές και Συναρτήσεις

5.1.1 Ζεύγος (Pair)

Η STL 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);
```

Η πρόσβαση στα μέλη του `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';
```

Κατασκευή ενός `pair` μπορεί να γίνει με 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);
```

Μεταξύ ζευγών ίδιου τύπου ορίζονται οι γνωστοί σχεσιακοί τελεστές (Πίνακας 2.3), 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.

5.1.2 Αντικείμενο–Συνάρτηση

Θα συναντήσουμε πολλές φορές στον ορισμό των containers και, ιδιαίτερα, στους αλγορίθμους, την έννοια ενός αντικειμένου που όταν ακολουθείται από ζεύγος παρενθέσεων με κανένα, ένα ή περισσότερα ορίσματα, επιστρέφει κάποια τιμή· συμπεριφέρεται δηλαδή ως συνάρτηση. Αυτό χαρακτηρίζεται ως αντικείμενο–συνάρτηση (function object ή functor). Στην απλή περίπτωση, είναι απλά το όνομα μιας συνάρτησης. Γενικότερα, μπορεί να είναι αντικείμενο μιας κλάσης για την οποία ορίζεται ο τελεστής `()`· θα δούμε πώς στο Κεφάλαιο 6.

Με την συμπερίληψη του header `<functional>`, η C++ παρέχει στο χώρο ονομάτων `std` ένα αριθμό από προκαθορισμένα αντικείμενα–συναρτήσεις. Είναι όλα 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 αντικείμενα–συναρτήσεις δίνονται στον Πίνακα 5.1 μαζί με την πράξη που εκτελούν. Ο τρόπος χρήσης τους θα παρουσιαστεί σε επόμενα εδάφια.

Αντικείμενο–Συνάρτηση	**Επιστρεφόμενη τιμή**
`negate<T>()`	−παράμετρος1
`plus<T>()`	παράμετρος1 + παράμετρος2
`minus<T>()`	παράμετρος1 − παράμετρος2
`multiplies<T>()`	παράμετρος1 * παράμετρος2
`divides<T>()`	παράμετρος1 / παράμετρος2
`modulus<T>()`	παράμετρος1 % παράμετρος2
`equal_to<T>()`	παράμετρος1 == παράμετρος2
`not_equal_to<T>()`	παράμετρος1 != παράμετρος2
`less<T>()`	παράμετρος1 < παράμετρος2
`greater<T>()`	παράμετρος1 > παράμετρος2
`less_equal<T>()`	παράμετρος1 <= παράμετρος2
`greater_equal<T>()`	παράμετρος1 >= παράμετρος2
`logical_not<T>()`	!παράμετρος1
`logical_and<T>()`	παράμετρος1 && παράμετρος2
`logical_or<T>()`	παράμετρος1 || παράμετρος2

Πίνακας 5.1: Προκαθορισμένα αντικείμενα–συναρτήσεις.

Τα προκαθορισμένα αντικείμενα–συναρτήσεις μπορούν να τροποποιηθούν με τη βοήθεια των *προσαρμογέων* (adapters), Πίνακας 5.2. Π.χ. το `plus<T>()` κανονικά όταν του δοθούν δύο ορίσματα επιστρέφει το άθροισμά τους. Μπορούμε να το τροποποιήσουμε ώστε το πρώτο του όρισμα να είναι σταθερό, π.χ.

Προσαρμογέας	**Επιστρεφόμενη τιμή**
`bind1st(functor,value)`	`functor(value,parameter)`
`bind2nd(functor,value)`	`functor(parameter,value)`
`not1(functor)`	`!functor(parameter)`
`not2(functor)`	`!functor(parameter1, parameter2)`
`ptr_fun(function)`	η συνάρτηση function γίνεται functor

Πίνακας 5.2: Προσαρμογείς για αντικείμενα-συναρτήσεις.

constval—και έτσι να δέχεται μόνο ένα όρισμα—αν το χρησιμοποιήσουμε μέσα από το `std::bind1st()` ως εξής:

```
std::bind1st(plus<T>(), constval)
```

Γενικότερα, μπορούμε να καθορίσουμε σταθερή τιμή για το πρώτο ή το δεύτερο όρισμα (αν υπάρχει) για όλα τα προκαθορισμένα αντικείμενα-συναρτήσεις ή να δράσουμε τον τελεστή (`!`) στο αποτέλεσμά τους. Μια συνήθης συνάρτηση με ένα ή δύο ορίσματα μπορεί να μετατραπεί με τη χρήση του `ptr_fun()`, σε αντικείμενο-συνάρτηση, ώστε να μπορεί να περάσει σε προσαρμογέα ή σε αλγόριθμο, §5.3.

5.1.3 Συναρτήσεις ελαχίστου, μεγίστου και εναλλαγής

Στο header `<algorithm>` ορίζονται οι συναρτήσεις `std::min<>()`, `std::max<>()` και `std::swap<>()` ως templates. Οι δηλώσεις τους είναι οι εξής:

```
namespace std {
  // minimum of a, b
  template <typename T>
  T const & min(T const & a, T const & b);

  // maximum of a, b
  template <typename T>
  T const & max(T const & a, T const & b);

  //  b <-> a
  template <typename T>
  void swap(T& a, T& b);
}
```

Οι ορισμοί που αντιστοιχούν στις παραπάνω δηλώσεις για τις `min/max` συγκρίνουν τα ορίσματα των συναρτήσεων με τον τελεστή (`<`) για να προσδιορίσουν το μικρότερο. Μπορούν, επομένως, να χρησιμοποιηθούν με αυτήν την απλή μορφή αν ο τύπος `T` έχει ορίσει τον παραπάνω τελεστή—κάτι που ισχύει για όλους τους ενσωματωμένους τύπους. Θα δούμε στο Κεφάλαιο 6 πώς ορίζονται νέοι τύποι από τον προγραμματιστή και πώς καθορίζεται η δράση των τελεστών.

Στην περίπτωση που η κλάση `T` δεν περιλαμβάνει τον τελεστή (<) μπορούμε να καθορίσουμε το κριτήριο με το οποίο θα γίνει η σύγκριση των ορισμάτων στις `min/max` ως εξής: δίνουμε ως τρίτο όρισμα ένα αντικείμενο-συνάρτηση το οποίο δέχεται δύο ορίσματα και επιστρέφει τη λογική τιμή της σύγκρισής τους (ανάλογα με το κριτήριο που έχουμε θέσει). Οι δηλώσεις των `min/max` γίνονται:

```
namespace std {
  // minimum of a, b based on cmp(a,b) ordering
  template <typename T, typename Compare>
  T const & min(T const & a, T const & b, Compare cmp);

  // maximum of a, b based on cmp(a,b) ordering
  template <typename T, typename Compare>
  T const & max(T const & a, T const & b, Compare cmp);
}
```

Για να γίνουν κατανοητά τα παραπάνω, ας δούμε μια πιθανή υλοποίηση της `std::min<>`, σύμφωνη με τη δήλωση που προηγήθηκε:

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

Ο ορισμός βασίζεται στο ότι η ποσότητα `cmp(a,b)` πρέπει να έχει νόημα και να επιστρέφει **`true`** ή **`false`** αν το πρώτο όρισμα είναι ή δεν είναι "μικρότερο" (ό,τι κι αν σημαίνει αυτό) από το δεύτερο. Η απαίτηση αυτή ικανοποιείται αν το `cmp` είναι κατάλληλα ορισμένη συνάρτηση με δήλωση

```
template<typename T>
bool cmp(T const & a, T const & b);
```

Εναλλακτικά, το `cmp` μπορεί να είναι αντικείμενο κλάσης στην οποία έχουμε ορίσει κατάλληλα τη δράση του τελεστή (`()`), όπως θα δούμε στο Κεφάλαιο 6.

5.2 Συλλογές (containers)

5.2.1 Εισαγωγή

Οι containers χρησιμοποιούνται για την αποθήκευση και διαχείριση συλλογών από αντικείμενα συγκεκριμένου τύπου. 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 διάνυσμα (array). Να αναφέρουμε εδώ ότι το διάνυσμα στη C++ μπορεί να έχει διάσταση είτε γνωστή κατά τη μεταγλώττιση είτε προσδιοριζόμενη κατά την εκτέλεση του προγράμματος (ως όρισμα στις συναρτήσεις δέσμευσης μνήμης `malloc`, `calloc`, `realloc`,

ή στον τελεστή **new**). Δεν θα επεκταθούμε περισσότερο στη δημιουργία και καταστροφή του διανύσματος, αλλά θα επικεντρωθούμε στους containers.

Οι containers διακρίνονται σε δύο γενικές κατηγορίες:

- Οι *sequence containers* είναι συλλογές στις οποίες κάθε στοιχείο έχει συγκεκριμένη θέση ως προς τα υπόλοιπα. Αυτή η σχετική θέση προσδιορίζεται κατά την εισαγωγή του 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 STL περιλαμβάνει τρεις κλάσεις που είναι sequence containers: `vector`, `deque` και `list`. Επιπλέον, το `string` και οι συνήθεις πίνακες μπορούν να θεωρηθούν ότι έχουν παρόμοια χαρακτηριστικά με τέτοιου τύπου containers και να χρησιμοποιηθούν με παραπλήσιο τρόπο.

- Οι *associative containers* είναι *ταξινομημένες* συλλογές στις οποίες η θέση κάθε στοιχείου εξαρτάται μόνο από την τιμή του και προσδιορίζεται σύμφωνα με κάποιο κριτήριο ταξινόμησης. Η STL παρέχει τέσσερις κλάσεις που είναι associative containers: `set`, `multiset`, `map` και `multimap`.

Προσέξτε πως η *αυτόματη* ταξινόμηση που γίνεται κατά την εισαγωγή των στοιχείων σε ένα associative container δε σημαίνει ότι αυτοί οι containers είναι ειδικά σχεδιασμένοι ή οι μόνοι ικανοί για ταξινόμηση. Το σημαντικό πλεονέκτημα έναντι των sequence containers είναι η ταχύτητα στην εύρεση συγκεκριμένου στοιχείου καθώς η ακολουθία σε αυτούς είναι ήδη ταξινομημένη.

Οι containers είναι υλοποιημένοι ως class template (§6.3). Και οι δύο κατηγορίες container δέχονται ως πρώτη παράμετρο τον τύπο των ποσοτήτων που θα αποθηκευθούν. Εξαίρεση αποτελούν οι `map` και `multimap`· σε αυτούς πρέπει να προσδιορίσουμε τους τύπους *δύο* ποσοτήτων καθώς αποθηκεύουν *ζεύγη* τιμών. Στους associative containers η επόμενη παράμετρος καθορίζει το κριτήριο με το οποίο γίνεται η ταξινόμηση. Είναι ο *τύπος* ενός αντικειμένου–συνάρτηση που ακολουθούμενο από δύο ορίσματα εντός παρενθέσεων επιστρέφει λογική τιμή (**true**/**false**) αν το πρώτο όρισμα είναι "μικρότερο" ή όχι από το δεύτερο. Η συγκεκριμένη παράμετρος έχει την προκαθορισμένη τιμή `std::less<T>` (§5.1.2) ώστε στη σύγκριση να χρησιμοποιείται ο τελεστής (<). Μια τελευταία παράμετρος, στην οποία δε θα αναφερθούμε, σχετίζεται με τη διαχείριση της μνήμης και έχει προκαθορισμένη τιμή.

Για να γίνει διαθέσιμος ένας container σε κάποιο πρόγραμμα πρέπει να γίνει η συμπερίληψη του αντίστοιχου header: `<vector>`, `<deque>`, `<list>`, `<set>` (*για* `set` *και* `multiset`) και `<map>` (*για* `map` *και* `multimap`). Ας επαναλάβουμε ότι οι containers, όπως όλη η STL, ορίζονται στο χώρο ονομάτων `std`.

Ένα σημαντικό χαρακτηριστικό των containers είναι ότι κάνουν *δυναμική* διαχείριση μνήμης, *αυτόματα*. Αυτό σημαίνει πως, χωρίς την παρέμβαση του

προγραμματιστή, οι θέσεις μνήμης που καταλαμβάνουν μπορούν να αυξάνουν ή, σε ορισμένες περιπτώσεις, να μειώνονται ώστε να χωρούν τα στοιχεία που αποτελούν κάθε στιγμή τη συλλογή.

Ορισμός container

Παράδειγμα δηλώσεων συγκεκριμένων containers είναι ο ακόλουθος κώδικας. Σε αυτόν ορίζεται μια μεταβλητή c ως (αρχικά κενό) `set` ακεραίων και μία μεταβλητή v ως κενό `vector` μιγαδικών αριθμών:

```
#include <set>
#include <vector>
#include <complex>

std::set<int> c;

std::vector<std::complex<double> > v;
// use c, v ........
```

Προσέξτε το κενό μεταξύ των > στον ορισμό του v· είναι αναγκαίο για να μην εμφανιστεί ο τελεστής >>.

Ας παραθέσουμε κάποιους γενικούς τρόπους ορισμού, κοινούς για όλους τους containers:

Έστω cntr ένας οποιοσδήποτε τύπος container (`vector`, `list`, `map`,...). 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 cntr.

- Ο κώδικας

  ```
  cntr<T> c2;

  // fill c2 ....

  cntr<T> c3(c2);
  ```

  ορίζει το c2 ως ένα αρχικά κενό cntr, εισάγει στοιχεία σε αυτό (θα δούμε παρακάτω πώς) και δημιουργεί το c3 ως *αντίγραφο* του c2, στοιχείο προς στοιχείο. Τα c2, c3 πρέπει φυσικά να είναι ίδιου τύπου. Προσέξτε ότι ο τύπος περιλαμβάνει και τις παραμέτρους του template.

- Ένας τρίτος τρόπος ορισμού χρησιμοποιεί την έννοια του iterator. Έτσι, αν π.χ. beg, end είναι δύο iterators ("δείκτες") *στον ίδιο container με τον beg να μη "δείχνει" μετά τον end*, μπορούμε να δημιουργήσουμε

έναν άλλο container (όχι απαραίτητα του ίδιου τύπου), αντιγράφοντας σε αυτόν το τμήμα των στοιχείων του αρχικού μεταξύ των θέσεων [beg,end):

```
cntr<T1> c4;
// fill c4 ...
// set beg, end on c4

cntr<T2> c5(beg,end);
// creates c5 by copying the elements of c4
// from 'beg' to (one before) 'end'.
```

Στο παράδειγμα αυτό τα T1, T2 δεν είναι απαραίτητα ισοδύναμοι τύποι· αρκεί να μπορούν αντικείμενα τύπου T1 να μετατραπούν σε T2. Έτσι αν π.χ. T2≡**double**, μπορούμε να έχουμε T1≡**int** (καθώς ορίζεται η μετατροπή **int** σε **double**) αλλά όχι T1≡`std::complex<`**double**>.

Ας αναφερθεί εδώ ότι για τους ενσωματωμένους πίνακες—παρόλο που δεν έχουν όλα τα χαρακτηριστικά των containers—μπορούμε να χρησιμοποιήσουμε τους συνήθεις δείκτες για να ορίσουμε "διαστήματα" iterators όπου χρειάζονται αυτά. Έτσι μπορούμε να δημιουργήσουμε π.χ. `vector` από ένα πίνακα χρησιμοποιώντας τον τελευταίο μηχανισμό ως εξής:

```
#include <vector>

double a[5] = {0.1, 0.2, 0.5, 0.3, 7.2};
std::vector<double> v(a, a+5);
```

Θυμηθείτε (§4.3) ότι το όνομα ενός πίνακα 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]`).

Επιπλέον των παραπάνω, υπάρχουν και άλλοι τρόποι για δήλωση με ταυτόχρονη απόδοση αρχικών τιμών, συγκεκριμένοι για κάθε container.

Κοινές συναρτήσεις–μέλη των containers

Όλοι οι containers παρέχουν ορισμένες κοινές συναρτήσεις–μέλη. Ανάμεσά τους είναι αυτές που παρουσιάζονται στον Πίνακα 5.3. Προσέξτε ότι ο reverse iterator `rbegin()` θεωρείται πως είναι *πριν* τον reverse iterator `rend()`· ο συνδυασμός των δύο χρησιμεύει στο να διατρέχουμε ένα container ανάστροφα.

Η κλήση των συναρτήσεων–μελών για ένα συγκεκριμένο container γίνεται με τον τελεστή (.) μεταξύ του ονόματος του container και της συνάρτησης (με τα ορίσματά της). Έτσι π.χ., αν έχουμε ορίσει ένα `vector` για πραγματικούς αριθμούς με όνομα c,

```
std::vector<double> c;
```

τότε, σε οποιοδήποτε επόμενο σημείο του κώδικά μας, μπορούμε να εξάγουμε το πλήθος των στοιχείων του όπως στο παρακάτω παράδειγμα:

Συνάρτηση	Επιστρεφόμενη τιμή — Δράση
`size()`	Το πλήθος των στοιχείων.
`empty()`	**`true`**/**`false`** αν ο container είναι κενός ή όχι. (ισοδύναμη με `size()==0` αλλά πιθανόν πιο γρήγορη)
`max_size()`	Το μέγιστο δυνατό πλήθος στοιχείων (καθοριζόμενο από την υλοποίηση).
`swap()`	Εναλλάσσει τα στοιχεία του container με τα στοιχεία του container που δίνεται ως όρισμα.
`clear()`	Διαγράφει όλα τα στοιχεία (αφήνει τον container κενό).
`erase()`	Διαγράφει το στοιχείο στον iterator που δίνεται ως όρισμα ή τα στοιχεία μεταξύ των iterators που δίνονται ως όρισμα.
`insert()`	Εισαγωγή στοιχείου.
`begin()`	iterator στη θέση του πρώτου στοιχείου.
`end()`	iterator σε *μία θέση μετά* το τελευταίο στοιχείο.
`rbegin()`	reverse iterator στη θέση του τελευταίου στοιχείου.
`rend()`	reverse iterator σε *μία θέση πριν* το πρώτο στοιχείο.

Πίνακας 5.3: Κοινές συναρτήσεις–μέλη των containers της STL.

```
std::cout << "Size␣of␣argument␣is␣" << c.size() << '\n';
```

Για δύο μεταβλητές *ίδιου τύπου* container ορίζονται οι σχεσιακοί τελεστές ==, !=, <, >, <=, >= 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*:

1. η σύγκριση γίνεται στοιχείο με στοιχείο έως ότου βρεθούν αντίστοιχα στοιχεία άνισα. Το αποτέλεσμα της σύγκρισής τους είναι η τιμή της σύγκρισης των containers.
2. αν όλα τα στοιχεία είναι ίσα, τότε ο container με τα λιγότερα είναι "μικρότερος".
3. αλλιώς οι containers είναι ίσοι.

Τροποποίηση container

Εισαγωγή ή τροποποίηση στοιχείων σε οποιοδήποτε container γίνεται ως εξής:

- Με ανάθεση από άλλο container, ίδιου τύπου:

  ```
  c1 = c2;
  ```

  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. Η διαδικασία αυτή μπορεί να είναι αρκετά χρονοβόρα σε σχέση με τον επόμενο μηχανισμό.

- Με εναλλαγή στοιχείων με container ίδιου τύπου χρησιμοποιώντας τη συνάρτηση `std::swap()` του `<algorithm>`:

  ```
  std::swap(c1,c2);
  ```

  ή, ισοδύναμα, τη συνάρτηση–μέλος `swap()`

  ```
  c1.swap(c2);
  ```

- Με τη συνάρτηση-μέλος `insert()`:

  ```
  c.insert(pos, elem);
  ```

  Με αυτή γίνεται εισαγωγή αντιγράφου του elem σε θέση που καθορίζεται (για sequence containers) ή προτείνεται (για associative containers) από τον 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 να υποδείξουμε μόνο την πιθανή θέση για να γίνει πιο γρήγορα η εισαγωγή. Η επιστρεφόμενη τιμή είναι iterator στη θέση του νέου στοιχείου. Παρατηρείστε ότι με τη συνάρτηση `insert()` γίνεται *εισαγωγή και όχι αντικατάσταση* στοιχείου.

Διαγραφή στοιχείων γίνεται:

- Με τη χρήση της συνάρτησης–μέλους `erase()` σε μία από τις παρακάτω μορφές:

  ```
  c.erase(pos);
  c.erase(beg,end);
  ```

  Με την πρώτη κλήση διαγράφεται το στοιχείο με iterator pos· με τη δεύτερη διαγράφονται τα στοιχεία με iterators στο "διάστημα" [beg,end) (απαιτείται, βέβαια, το "διάστημα" να μην είναι κενό και να είναι "διάστημα" του c). Αν ο c είναι sequence container επιστρέφει iterator στο επόμενο στοιχείο ενώ για associative containers δεν επιστρέφει τίποτε.

- Με τη συνάρτηση–μέλος `clear()`. Αυτή αφαιρεί όλα τα στοιχεία αφήνοντας κενό τον container. Δεν επιστρέφει τίποτε ("επιστρέφει" **`void`**):

  ```
  c.clear();
  ```

Πέρα από αυτούς, κάθε container παρέχει και άλλους μηχανισμούς για προσπέλαση και μεταβολή των στοιχείων του.

5.2.2 Iterators

Όπως γίνεται κατανοητό από τα παραπάνω, μια εισαγωγική περιγραφή των iterators είναι απαραίτητη για την κατανόηση πολλών θεμάτων που σχετίζονται με τους containers.

Έχουμε αναφέρει πως ο iterator συμπεριφέρεται σε μεγάλο βαθμό ως δείκτης παρόλο που διαφέρει ως έννοια. Η αναλογία iterator και δείκτη που συναντήσαμε στην απόδοση αρχικών τιμών από πίνακα, ενισχύεται από το ότι η δράση του τελεστή (`*`) στο όνομα ενός iterator μας δίνει πρόσβαση στην ποσότητα στην οποία αυτός "δείχνει". Έτσι, *η ποσότητα `*it` είναι η τιμή του στοιχείου που βρίσκεται, σε ένα container, στη θέση που "δείχνει" ο iterator it*.

Σε αντίθεση με τους δείκτες, ένας iterator δεν μπορεί να εξαχθεί με τη δράση του (`&`) στο όνομα μιας μεταβλητής. Προσδιορίζεται από την επιστρεφόμενη τιμή αλγορίθμων της STL ή συναρτήσεων–μελών των containers. Μία ποσότητα, π.χ. it, μπορεί να οριστεί ως iterator για ένα container (π.χ. `vector`) ως εξής:

```
std::vector<double>::iterator it;
```

Κατ' αντιστοιχία με τους pointers σε σταθερές τιμές ορίζονται οι iterators σε σταθερές ποσότητες. Iterator μέσω του οποίου δεν μπορεί να αλλάξει η τιμή στη θέση που "δείχνει" ορίζεται ως εξής:

```
std::vector<double>::const_iterator it;
```

Και στις δύο παραπάνω δηλώσεις, ο it μπορεί επιπλέον να προσδιοριστεί ως **const** και να του αποδοθεί αρχική (και μόνιμη) "τιμή".

Αναφέραμε παραπάνω τις συναρτήσεις–μέλη `rbegin()` και `rend()` κάθε container. Αυτές επιστρέφουν *ανάστροφο* (reverse) iterator. Ο τύπος αυτός (που στην πραγματικότητα είναι προσαρμογέας) παρέχεται από κάθε container και χρησιμοποιείται για να διατρέξουμε ανάστροφα τους containers. Μπορεί να χρησιμοποιηθεί όπου χρειάζεται iterator. Δήλωση, με ανάθεση αρχικής τιμής, τέτοιου iterator, π.χ. για `vector`, γίνεται ως εξής:

```
std::vector<double> v(10);

std::vector<double>::reverse_iterator rit = v.rbegin();
```

Επιπλέον, υπάρχει και ο τύπος `const_reverse_iterator` για ανάστροφους iterators που δεν μπορούν να μεταβάλουν την τιμή στη θέση στην οποία "δείχνουν".

Παράδειγμα:

Αν θέλουμε να τυπώσουμε τα στοιχεία ενός container, π.χ. ενός `std::vector<double>` με όνομα v, μπορούμε να χρησιμοποιήσουμε την ακόλουθη εντολή (δείτε τον Πίνακα 5.3 για τις συναρτήσεις `begin()`, `end()`):

```
for (std::vector<double>::const_iterator cit = v.begin();
     cit != v.end();
     ++cit)
    std::cout << *cit << '\n';
```

Η εκτύπωση με την αντίστροφη φορά μπορεί να γίνει με την εντολή

```
for (std::vector<double>::const_reverse_iterator crit = v.rbegin();
     crit != v.rend();
```

```
     ++crit)
    std::cout << *crit << '\n';
```

Η απόδοση μίας τιμής, π.χ. −3, στα στοιχεία ενός `std::vector<int>` με όνομα v μπορεί να γίνει ως εξής

```
for (std::vector<int>::iterator it = v.begin(); it != v.end();
     ++it)
     *it = -3;
```

Δύο βασικές κατηγορίες iterators μας ενδιαφέρουν:

Οι random (τυχαίας προσπέλασης**) iterators.** Αυτοί συμπεριφέρονται ακριβώς όπως οι δείκτες· ό,τι γνωρίζουμε για λογικές σχέσεις και αριθμητική δεικτών ισχύει και για αυτούς: μπορούμε να τους συγκρίνουμε με τους γνωστούς σχεσιακούς τελεστές (Πίνακας 2.3), η έκφραση `++it` προωθεί τον it στην επόμενη θέση, η πρόσθεση ενός αριθμού σε ένα iterator αυτής της κατηγορίας μας μεταφέρει τόσες θέσεις παρακάτω, η διαφορά δύο random iterators που "δείχνουν" στον ίδιο container είναι το πλήθος των ενδιάμεσων στοιχείων, κοκ. Οι iterators των `vector` και `deque` ανήκουν σε αυτή την κατηγορία.

Οι bidirectional (δύο κατευθύνσεων**) iterators.** Οι iterators αυτοί έχουν ορισμένους περιορισμούς:

1. μπορούν να συγκριθούν μεταξύ τους μόνο για ισότητα ή ανισότητα με τους τελεστές (==) και (!=).
2. μπορούν να προωθηθούν ή να υποχωρήσουν μόνο κατά ένα βήμα, επιτρέπεται δηλαδή το `++it` ή το `--it` (και, βέβαια, τα `it++` και `it--`). Δεν μπορούμε να αφαιρέσουμε ένα bidirectional iterator από άλλον, ούτε να του προσθέσουμε αριθμό.

Οι iterators της `list` και των associative containers ανήκουν σε αυτή την κατηγορία.

Για να μπορούμε να γράφουμε κώδικα γενικό, που να ισχύει για διάφορους containers, η STL παρέχει στο `<iterator>`

- τη συνάρτηση

  ```
  void advance(Iterator & it, Difference_type n);
  ```

  όπου n ποσότητα ακέραιου τύπου.[1] Αν ο iterator it είναι random, η κλήση της ισοδυναμεί με `it += n;`. Αν είναι bidirectional ισοδυναμεί με n διαδοχικές κλήσεις του `++it` (αν n>0) ή `--it` (αν n<0).

- τη συνάρτηση

[1] `std::iterator_traits<InputIterator>::difference_type`.

```
Difference_type distance(Iterator it1, Iterator it2);
```

Οι it1, it2 είναι iterators ίδιου τύπου που "δείχνουν" στον ίδιο container. Αν είναι random επιστρέφει το `it2-it1` ενώ αν είναι bidirectional 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.

Παρατήρηση 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 T, typename C>
void
f(C & a) {
std::vector<T> v;
C b;
...
}
```

Μέσα στην f ορίζουμε ένα `std::vector` για αποθήκευση στοιχείων τύπου T και μια ποσότητα τύπου C (που μπορεί να είναι κάποιος σύνθετος τύπος, π.χ. `std::list<int>`). Αν θελήσουμε να ορίσουμε iterators για αυτές τις ποσότητες δεν αρκεί να γράψουμε

```
std::vector<T>::iterator itv;   // error
C::iterator itl;  // error
```

αλλά πρέπει να δηλώσουμε στο μεταγλωττιστή ότι οι εκφράσεις `C::iterator` και `std::vector<T>::iterator` αποτελούν *τύπους*. Αυτό γίνεται αν συμπληρώσουμε τις δηλώσεις των iterators με τη λέξη **typename**:

```
typename std::vector<T>::iterator itv;   // correct
typename C::iterator itl;  // correct
```

5.2.3 vector

Ποσότητα τύπου `vector` επιτρέπει την τυχαία προσπέλαση των αντικειμένων που περιέχει. Όπως και σε κάθε sequence container, τα στοιχεία αποθηκεύονται με τη σειρά εισαγωγής τους. Ένα `vector` είναι ιδιαίτερα κατάλληλο για την προσθήκη στοιχείων στο τέλος του. Αντίθετα, η εισαγωγή σε οποιοδήποτε άλλο σημείο του, επιτρέπεται μεν αλλά είναι πιο αργή, λιγότερο ή περισσότερο. Αυτό συμβαίνει για τον εξής λόγο:

Το μέγεθος της μνήμης που καταλαμβάνει ένα `vector` αυξάνει αυτόματα κατά την προσθήκη στοιχείων όπως και σε κάθε άλλον container. Όμως, τα στοιχεία του είναι απαραίτητο να βρίσκονται *διαδοχικά* στη μνήμη του συστήματος. Πρώτη συνέπεια αυτού είναι ότι η απόπειρα εισαγωγής στοιχείου

στο "εσωτερικό" ή στην αρχή του `vector` προκαλεί μετακίνηση με αντιγραφή λιγότερων ή περισσότερων στοιχείων ώστε να δημιουργηθεί χώρος. Δεύτερη συνέπεια είναι πως αν το τμήμα της μνήμης που διατέθηκε από το λειτουργικό σύστημα σε ένα `vector` δεν επαρκεί για την εισαγωγή στοιχείου οπουδήποτε, γίνεται αντιγραφή *όλων* των στοιχείων, μαζί με το προστιθέμενο, σε νέο, μεγαλύτερο. Σε αυτήν την περίπτωση, όλοι οι δείκτες, iterators και αναφορές παύουν να ισχύουν καθώς συνδέονται με το παλαιό τμήμα.

Ας σημειωθεί εδώ ότι μπορούμε να προσομοιάσουμε τη συμπεριφορά ενός `vector` αν χρησιμοποιήσουμε ενσωματωμένο πίνακα (array), κατάλληλες ενσωματωμένες συναρτήσεις (που δεν παρουσιάσαμε) για δέσμευση, αντιγραφή ή ελευθέρωση μνήμης, και δικές μας συναρτήσεις για εισαγωγή στοιχείου με μετακίνηση/αντιγραφή άλλων. Όλα τα παραπάνω, όμως, πρέπει να προσδιορίζονται ρητά από τον προγραμματιστή. Η αυτόματη διαχείριση της προσθήκης ή διαγραφής στοιχείων δίνει στο `vector` σημαντικό πλεονέκτημα έναντι του ενσωματωμένου πίνακα χωρίς να υστερεί καθόλου σε ταχύτητα.

Η χρήση ενός `std::vector` προϋποθέτει τη συμπερίληψη του header `<vector>`.

Ορισμός

Η κλάση `vector` παρέχει διάφορους μηχανισμούς για τον ορισμό ποσοτήτων με ταυτόχρονη απόδοση αρχικής τιμής. Έχουμε ήδη δει κάποιους από αυτούς· για πληρότητα θα τους επαναλάβουμε με συντομία.

- Η εντολή

  ```
  std::vector<T> v;
  ```

  δημιουργεί ένα κενό `vector`.

- Η εντολή

  ```
  std::vector<T> v(N);
  ```

  ορίζει το v ως ένα `vector` με N θέσεις για αντικείμενα τύπου T. *Όλα παίρνουν αρχική τιμή `T()`, την προκαθορισμένη για τον τύπο T.*[2] Αν ο T είναι ενσωματωμένος τύπος, η προκαθορισμένη τιμή είναι 0 (§2.2).

- 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 αυτόν τον τύπο.

- Η εντολή

[2] αρκεί να είναι προσβάσιμος ο συγκεκριμένος constructor, §6.2.2.

```
std::vector<T> v1(v2);
```

δημιουργεί το v1 ως αντίγραφο του v2.

- Η εντολή

```
std::vector<T> v(beg,end);
```

κατασκευάζει το v αντιγράφοντας στοιχεία από το τμήμα μεταξύ των iterators beg και end.

Η κλάση `vector` περιλαμβάνει ως μέλη δύο συναρτήσεις σχετικές με το μέγεθος μιας ποσότητας τύπου `vector` επιπλέον των κοινών `size()`, `empty()` και `max_size()` που περιγράψαμε ήδη:

- Η `capacity()` επιστρέφει το μέγιστο δυνατό πλήθος στοιχείων που μπορεί να αποθηκευθεί στη μεταβλητή για την οποία καλείται χωρίς να χρειαστεί μετακίνηση του `vector` σε άλλο τμήμα μνήμης· μας δίνει, δηλαδή, το διαθέσιμο χώρο τη στιγμή της κλήσης.

- Η `reserve()` δεσμεύει τόσες συνεχόμενες θέσεις όσες καθορίζει το όρισμά της. Αν αυτό είναι μεγαλύτερο από τον τρέχοντα διαθέσιμο χώρο προκαλεί τη μεταφορά του `vector` σε κατάλληλο τμήμα μνήμης. 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 ανάθεση στα στοιχεία `v[0]`, `v[1]`, ..., `v[N-1]`.

Προσθήκη στοιχείων

Περιγράψαμε ήδη 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`.

- Η εντολή

```
v1 = v2;
```

αντιγράφει όλα τα στοιχεία του v2 στο v1 καταστρέφοντας τα αρχικά.

- Η εντολή

  ```
  v.assign(N,elem);
  ```

  καταστρέφει τα στοιχεία του v και εισάγει N αντίγραφα του elem μετατρέποντάς τα, αν χρειάζεται, στον τύπο των στοιχείων του v.

- Η εντολή

  ```
  v.assign(beg, end);
  ```

  καταστρέφει τα στοιχεία του v και εισάγει τα στοιχεία ενός άλλου container (ή και ενσωματωμένου πίνακα) με iterators μεταξύ του beg και μιας θέσης πριν το end. Να επαναλάβουμε ότι προϋπόθεση είναι οι beg και end να αναφέρονται στον ίδιο container, τα στοιχεία του οποίου να μπορούν να μετατραπούν στον τύπο των στοιχείων του v και ο beg να μη "δείχνει" μετά τον end.

- Οι εντολές

  ```
  v1.swap(v2);
  ```

  ή

  ```
  std::swap(v1,v2);
  ```

  εναλλάσσουν τα στοιχεία των v1, v2.

- Η εντολή

  ```
  v.insert(pos, elem);
  ```

  τοποθετεί πριν τη θέση που "δείχνει" ο iterator pos, ένα αντίγραφο του elem και επιστρέφει iterator στη θέση του νέου στοιχείου.

- Η εντολή

  ```
  v.insert(pos, N, elem);
  ```

  τοποθετεί, πριν τη θέση που "δείχνει" ο iterator pos, N αντίγραφα του elem, ενώ δεν επιστρέφει τίποτε. Η εισαγωγή πολλών αντιγράφων ενός στοιχείου γίνεται πιο γρήγορα με τη συγκεκριμένη συνάρτηση-μέλος παρά με πολλαπλές κλήσεις της προηγούμενης.

- Πολλαπλή εισαγωγή στοιχείων επιτυγχάνεται επίσης με τη κλήση της `insert()` με ορίσματα iterators:

  ```
  v.insert(pos, beg, end);
  ```

  Με αυτή, τοποθετούνται πριν τη θέση iterator pos, στο `vector` v, τα στοιχεία στο "διάστημα" μεταξύ των iterators beg (περιλαμβανομένου), και end (μη περιλαμβανομένου).

- Η εντολή

  ```
  v.push_back(elem);
  ```

  και είναι ο πιο αποτελεσματικός και συνηθέστερα χρησιμοποιούμενος τρόπος για εισαγωγή ενός στοιχείου elem καθώς το τοποθετεί στο τέλος του `vector`. Δεν επιστρέφει τιμή.

- 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.

Διαγραφή στοιχείων

Διαγραφή στοιχείων ενός `vector` γίνεται:

- Με τη χρήση της `resize()` που παρουσιάστηκε παραπάνω.
- Με τη κοινή συνάρτηση-μέλος `clear()`.
- Με τη συνάρτηση-μέλος `erase()`. Όπως ισχύει για κάθε container και ήδη περιγράψαμε, η κλήση της μπορεί να γίνει με "διάστημα" καθοριζόμενο από iterators

  ```
  c.erase(beg,end);
  ```

  Εναλλακτικά για sequence containers, μπορούμε να διαγράψουμε το στοιχείο στη θέση που προσδιορίζει ένας iterator, έστω ο pos:

  ```
  v.erase(pos);
  ```

  Και στις δύο περιπτώσεις η επιστρεφόμενη τιμή είναι iterator στο επόμενο στοιχείο.

- Σε αντιστοιχία με την `push_back()` η συνάρτηση-μέλος `pop_back()`, χωρίς όρισμα και επιστρεφόμενη τιμή, διαγράφει το τελευταίο στοιχείο:

  ```
  v.pop_back();
  ```

Προσέξτε ότι κάθε εισαγωγή ή διαγραφή στοιχείου ακυρώνει όλες τις αναφορές, τους δείκτες και τους iterators σε στοιχεία *μετά* τη θέση που έγινε η τροποποίηση. Επιπλέον, αν η εισαγωγή προκάλεσε τη μετακίνηση του `vector` σε μεγαλύτερο τμήμα μνήμης, ακυρώνονται *όλοι* οι δείκτες, αναφορές και iterators.

Προσπέλαση στοιχείων

Προσπέλαση και επομένως, δυνατότητα μεταβολής των μεμονωμένων στοιχείων ενός `vector` γίνεται ως εξής:

- Με τη χρήση ακέραιου δείκτη μεταξύ των αγκυλών (`[]`) όπως ακριβώς στους ενσωματωμένους πίνακες:

```
std::vector<double> v(3);

v[0] = 1.0;
v[1] = 3.0;
v[2] = v[1] + 5.0;
```

- Με τη χρήση της συνάρτησης-μέλους `at()` με ακέραιο όρισμα. Το πρώτο, δεύτερο, τρίτο,... στοιχείο του `vector` 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()-1]` διακόπτεται η εκτέλεση του προγράμματος.[3] Ο απαιτούμενος έλεγχος στην τιμή του δείκτη έχει ως αποτέλεσμα να είναι πιο αργή η πρόσβαση απ' ότι με τις αγκύλες.

- Με τις συναρτήσεις-μέλη `front()` και `back()`. Αυτές επιστρέφουν αναφορά στο πρώτο και τελευταίο στοιχείο αντίστοιχα χωρίς, όμως, να ελέγχεται η ύπαρξή τους.

- Με τη δράση του τελεστή (`*`) σε όνομα iterator. Προσέξτε ότι αν είναι τύπου `const_iterator` δεν μπορούμε να μεταβάλουμε την τιμή που "δείχνει" αλλά μόνο να τη διαβάσουμε.

Παράδειγμα:

Δημιουργία, αντιγραφή και προσπέλαση στοιχείων ενός `vector` μπορεί να γίνει ως ακολούθως:

```
#include <iostream>
#include <vector>
#include <cstddef>

int
main() {
    std::vector<double> v(10);
    // v = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0}

    // assign values
    for (std::size_t i = 0; i < v.size(); ++i)
        v[i] = 4.0 * i*i;
```

[3] Αυτό συμβαίνει αν δεν "συλλάβουμε" την "εξαίρεση" που προκαλεί η συνάρτηση `at()`.

```
    std::vector<double> v2(v);
    // v2 is a copy of v

    // append more values to v2
    std::size_t const size = v2.size();
    for (std::size_t i = size; i < 2*size; ++i)
        v2.push_back(4.0*i*i);

    // print v2
    std::cout << "v2␣is\t";

    for (std::vector<double>::const_iterator it = v2.begin();
         it != v2.end();
         ++it)
        std::cout << *it << '␣';

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

Παρατήρηση Η εξειδίκευση `std::vector<bool>` έχει αρκετούς περιορισμούς στη χρήση της (δεν είναι container και δεν περιέχει **bool**!). Προτιμήστε το `std::deque<bool>` αν χρειαστείτε να αποθηκεύσετε **bool** ή συμβουλευτείτε τη βιβλιογραφία (π.χ. [1], σελ. 158, [2], σελ. 41–45, [3], item 18) αν τη χρειαστείτε.

5.2.4 deque

Ο τύπος `deque` (double-ended queue) είναι σε μεγάλο βαθμό όμοιος στις δυνατότητες με το `vector`. Είναι sequence container και, όπως ένα `vector`, επιτρέπει την τυχαία προσπέλαση των αντικειμένων που περιέχει. Η διαφορά τους είναι ότι, σε αντίθεση με το `vector` που επιτρέπει ταχύτατη εισαγωγή ή διαγραφή στοιχείων μόνο στο τέλος του, η `deque` παρέχει αυτή τη δυνατότητα και στις δύο άκρες της. Η εισαγωγή σε σημείο μακριά από τα άκρα είναι αργή, λιγότερο ή περισσότερο.

Η χρήση μιας `std::deque` προϋποθέτει τη συμπερίληψη του header `<deque>`.

Η `deque` συνήθως δεν αποθηκεύει τα στοιχεία σε ένα ενιαίο τμήμα μνήμης αλλά σε πολλά κομμάτια. Αυτό έχει ως συνέπεια να παρουσιάζει πιο αργή πρόσβαση στα στοιχεία απ' ό,τι ένα `vector` αλλά και το πλεονέκτημα να μη χρειάζεται μετακίνηση και αντιγραφή πολλών στοιχείων κατά την εισαγωγή μακριά από τα άκρα. Επιπλέον, η συγκεκριμένη οργάνωση της μνήμης επιτρέπει όχι μόνο να αυξάνει ο απαιτούμενος χώρος αυτόματα, όπως γίνεται σε ένα `vector`, αλλά και να μειώνεται. Σημειώστε ότι αυτός ο κατακερματισμός είναι εσωτερικός· κατά τη χρήση της η `deque` συμπεριφέρεται σαν να αποθηκεύει τα στοιχεία διαδοχικά στη μνήμη. Επομένως, ανεξάρτητα από τις

λεπτομέρειες της υλοποίησής της, η αύξηση ενός iterator κατά ένα μας μεταφέρει στο αμέσως επόμενο στοιχείο, όπου και να βρίσκεται αυτό στη μνήμη.

Η κατακερματισμένη μνήμη σε μία `deque` δεν επιτρέπει στο χρήστη της να έχει τον έλεγχο που έχει σε ένα `vector`. Η κλάση δεν παρέχει τις συναρτήσεις-μέλη `capacity()` και `reserve()` που συναντήσαμε στο `vector`. Καθώς δεν μπορούμε να γνωρίζουμε πότε θα χρειαστεί μετακίνηση στοιχείων (και ποιων) θα πρέπει να θεωρούμε ότι η εισαγωγή ή διαγραφή σε οποιοδήποτε σημείο μιας `deque` ακυρώνει όλες τις αναφορές, τους δείκτες και τους iterators σε στοιχεία της. Εξαίρεση αποτελεί η εισαγωγή στην αρχή ή στο τέλος· αυτή διατηρεί τις αναφορές και τους δείκτες (αλλά όχι τους iterators).

Με την εξαίρεση των `capacity()` και `reserve()`, όλες οι συναρτήσεις που αναφέρθηκαν στο `vector` παρέχονται και από τη `deque`. Επιπλέον, υπάρχει η αναμενόμενη συμπλήρωση με τις `push_front()` (εισαγωγή στοιχείου στην αρχή) και `pop_front()` (διαγραφή του πρώτου στοιχείου), σε πλήρη αντιστοιχία με τις `push_back()` και `pop_back()`.

Παράδειγμα:

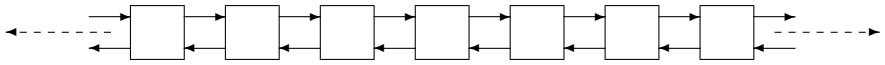
Ένα παράδειγμα ορισμού και χρήσης της `deque` είναι το ακόλουθο. Δοκιμάστε να το εκτελέσετε. Προσέξτε τη διαφορετική συμπεριφορά της αναφοράς και του iterator κατά την εισαγωγή στοιχείων.

```
#include <deque>
#include <iostream>

int
main() {
    std::deque<int> d;  // d is empty

    for (int i = 0; i != 10; ++i)
        d.push_front(i);
    // d is : {9, 8, ..., 0}


    int & r = d.front();
    std::cout << "front␣element␣is␣" << r << '\n';

    std::deque<int>::iterator beg = d.begin();
    std::cout << "front␣element␣through␣iterator␣is␣"
              << *beg << '\n';

    for (int i = 10; i != 10000; ++i)
        d.push_front(i);
    // d is : {9999, 9998, ..., 0}

    // THE SAME AS BEFORE INSERTION
    std::cout << "old␣front␣element␣is␣" << r << '\n';

    // NOT NECESSARILY THE SAME AS ABOVE
```

```
    std::cout << "old␣front␣element␣through␣iterator␣is␣"
              << *beg << '\n';
}
```

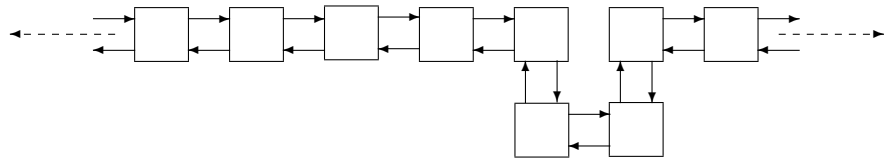
5.2.5 list

Η υλοποίηση της `list`, του τρίτου sequence container που παρέχει η STL, είναι πολύ διαφορετική απ' ό,τι των `vector` και `deque`. Στη `list` τα στοιχεία αποθηκεύονται σε πιθανώς απομονωμένα τμήματα μνήμης το καθένα, μαζί με την πληροφορία (συνήθως σε μορφή δεικτών) για τη θέση του επόμενου και του προηγούμενου στοιχείου, Σχήμα 5.1.



Σχήμα 5.1: Δομή `list`.

Η συγκεκριμένη εσωτερική δομή παρουσιάζει το μειονέκτημα, έναντι των `vector` και `deque`, ότι δεν επιτρέπει τυχαία προσπέλαση των στοιχείων. Αντίθετα, πρέπει να μετακινούμαστε διαδοχικά από το ένα στοιχείο στο άλλο έως ότου φτάσουμε στο ζητούμενο. Η διαδικασία αυτή είναι βέβαια πιο αργή, λιγότερο ή περισσότερο, από την πρόσβαση που παρέχουν οι άλλοι sequence containers (που είναι σε σταθερό χρόνο). Η δομή όμως έχει ένα σημαντικό πλεονέκτημα: η προσθήκη ή διαγραφή στοιχείων σε οποιοδήποτε σημείο μιας `list` είναι το ίδιο γρήγορη καθώς απαιτεί αλλαγές δεικτών (Σχήμα 5.2) και όχι αντιγραφή στοιχείων. Το ότι δε γίνεται μετακίνηση στοιχείων έχει ως συνέπεια πως οι αναφορές, οι δείκτες και οι iterators σε στοιχεία της δε χάνονται ποτέ.



Σχήμα 5.2: Προσθήκη στοιχείων σε `list`.

Η χρήση της `std::list` προϋποθέτει τη συμπερίληψη του header `<list>`.

Ορισμός

Ορισμός μιας `list` γίνεται με τους γνωστούς τρόπους από τους άλλους sequence containers. Παρατίθενται χωρίς ιδιαίτερο σχολιασμό.

- `std::list<T> c;`: Δημιουργεί κενή λίστα.
- `std::list<T> c1(c2);`: Δημιουργεί λίστα, αντίγραφο άλλης.

- `std::list<T> c(N);`: Δημιουργεί λίστα N στοιχείων με την προκαθορισμένη τιμή `T()`.

- `std::list<T> c(N, elem);`: Δημιουργεί λίστα από N αντίγραφα του elem.

- `std::list<T> c(beg,end);`: Δημιουργεί λίστα από τα στοιχεία στο διάστημα [beg,end).

Προσθήκη στοιχείων

Από την κλάση `list` παρέχονται οι αναμενόμενοι μηχανισμοί για την προσθήκη στοιχείων:

- `c1 = c2;`: Αντιγράφει τα στοιχεία της c2 στη c1 καταστρέφοντας τα αρχικά.

- `c.assign(N,elem);`: Καταστρέφει τα στοιχεία της c και εισάγει N αντίγραφα του elem.

- `c.assign(beg,end);`: Καταστρέφει τα στοιχεία της c και εισάγει τα στοιχεία του διαστήματος μεταξύ των iterators [beg,end).

- `c1.swap(c2);` ή `std::swap(c1,c2);`: Εναλλάσσει τα στοιχεία των c1, c2.

- `c.insert(pos,elem);`: Εισάγει πριν τη θέση που "δείχνει" ο iterator pos αντίγραφο του elem. Επιστρέφει iterator στο νέο στοιχείο.

- `c.insert(pos,N,elem);`: Εισάγει πριν τη θέση που "δείχνει" ο iterator pos N αντίγραφα του elem. Δεν επιστρέφει τίποτε.

- `c.insert(pos,beg,end);`: Εισάγει πριν τη θέση που "δείχνει" ο iterator pos τα στοιχεία στο διάστημα [beg,end). Δεν επιστρέφει τίποτε.

- `c.push_back(elem);`: Εισάγει αντίγραφο του elem στο τέλος του c.

- `c.push_front(elem);`: Εισάγει αντίγραφο του elem στην αρχή του c.

- `c.resize(N);`: Μεταβάλλει σε N το πλήθος των στοιχείων του c. Αν το αυξάνει, τα νέα στοιχεία έχουν την προκαθορισμένη τιμή για τον τύπο των στοιχείων της λίστας. Δεν επιστρέφει τίποτε.

- `c.resize(N,elem);`: Ό,τι κάνει η προηγούμενη εντολή αλλά σε περίπτωση αύξησης τα νέα στοιχεία είναι αντίγραφα του elem.

Διαγραφή στοιχείων

Διαγραφή στοιχείων από μια `list` c, γίνεται ως εξής:

- `c.pop_back();`: Διαγράφει το τελευταίο στοιχείο.
- `c.pop_front();`: Διαγράφει το πρώτο στοιχείο.
- `c.erase(pos);`: Διαγράφει το στοιχείο στη θέση με iterator pos και επιστρέφει iterator στο επόμενο.
- `c.erase(beg,end);`: Διαγράφει τα στοιχεία με θέσεις στο διάστημα [beg,end) και επιστρέφει iterator στο επόμενο.
- `c.clear();`: Διαγράφει όλα τα στοιχεία του c.
- Με τους δύο τρόπους κλήσης της `resize()`.

Προσπέλαση στοιχείων

Προσπέλαση στοιχείων μιας `list` γίνεται ως εξής:

- Με τις συναρτήσεις–μέλη `front()` και `back()`. Αυτές επιστρέφουν αναφορά στο πρώτο και τελευταίο στοιχείο αντίστοιχα χωρίς, όμως, να ελέγχουν αν αυτά υπάρχουν.
- Με τη δράση του τελεστή (`*`) σε όνομα iterator. Προσέξτε ότι αν είναι τύπου `const_iterator` δεν μπορούμε να μεταβάλουμε την τιμή που "δείχνει" αλλά μόνο να τη διαβάσουμε.

Επιπλέον συναρτήσεις–μέλη

Η `list`, όπως και όλοι οι containers, παρέχει τις `size()`, `empty()` και `max_size()` που περιγράψαμε προηγουμένως (§5.2.1), τις συναρτήσεις που επιστρέφουν iterators, ενώ λόγω του τρόπου οργάνωσης της μνήμης, δεν υπάρχουν οι `capacity()` και `reserve()`.

Η `list` παρέχει επιπλέον και ορισμένες συναρτήσεις–μέλη τις οποίες δεν έχουμε συναντήσει στους άλλους sequence containers:

- Η εντολή

  ```
  c.remove(val);
  ```

  διαγράφει όλα τα στοιχεία του c με τιμή val.

- Η εντολή

  ```
  c.remove_if(func);
  ```

παίρνει ως όρισμα ένα αντικείμενο-συνάρτηση το οποίο δέχεται ένα όρισμα και επιστρέφει λογική τιμή, **true** ή **false**. Η `remove_if()` δρα αυτό το αντικείμενο σε όλα τα στοιχεία της `list` και διαγράφει αυτά για τα οποία η `func()` δίνει **true**.

- Η συνάρτηση-μέλος `unique()` εντοπίζει ομάδες *διαδοχικών* στοιχείων με ίδια τιμή και τα διαγράφει όλα εκτός από το πρώτο.

- Η `unique(func)` κάνει το ίδιο για όλα τα διαδοχικά στοιχεία που η δράση της `func()` επιστρέφει **true**.

- Η εντολή

  ```
  c1.splice(pos, c2);
  ```

  μετακινεί, πριν τη θέση με iterator pos, όλα τα στοιχεία του c2, διαγράφοντάς τα από τον c2. Οι c1, c2 δεν πρέπει να είναι η ίδια `list`.

- Η εντολή

  ```
  c1.splice(c1pos, c2, c2pos);
  ```

  μετακινεί πριν τη θέση με iterator c1pos το στοιχείο του c2 με iterator c2pos. Οι c1, c2 μπορούν να είναι ίδιοι.

- Η εντολή

  ```
  c1.splice(c1pos, c2, c2beg, c2end);
  ```

  μετακινεί πριν τη θέση με iterator c1pos τα στοιχεία του c2 στο διάστημα [c2beg, c2end). Οι c1, c2 μπορούν να είναι ίδιοι.

- Η συνάρτηση-μέλος `sort()` ταξινομεί τη λίστα συγκρίνοντας τα στοιχεία με τον τελεστή (<). Προσέξτε ότι δεν μπορούμε να χρησιμοποιήσουμε την `std::sort()` από το `<algorithm>` καθώς αυτή χρειάζεται να έχει πρόσβαση σε τυχαίο στοιχείο, κάτι που δεν παρέχει η `list`.

- Η `sort(op)` δέχεται ως όρισμα αντικείμενο-συνάρτηση που παίρνει δυο ορίσματα και επιστρέφει **true** ή **false**. Ταξινομεί χρησιμοποιώντας αυτή τη συνάρτηση.

- Η εντολή

  ```
  c1.merge(c2);
  ```

  για ταξινομημένες λίστες c1 και c2, μετακινεί τα στοιχεία της c2 στη c1 με τέτοιο τρόπο ώστε, μετά τη συγχώνευση, η τελευταία να είναι πάλι ταξινομημένη.

- Η εντολή

```
c1.merge(c2, op);
```

κάνει το ίδιο με την προηγούμενη ταξινομώντας με βάση το αποτέλεσμα του αντικειμένου-συνάρτησης `op()`.

- Η συνάρτηση-μέλος `reverse()` αναστρέφει τη σειρά των στοιχείων στη `list` για την οποία καλείται.

Παράδειγμα:

```
#include <list>
#include <iostream>

inline 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(18.0);
  // any element with value equal to 18.0 is erased.

  c.remove_if(lessthan10);
  // all elements with value less than 10 are erased.


  std::cout << "List␣is␣:\t";

  for ( std::list<double>::const_iterator it = c.begin();
        it != c.end();
        ++it )
        std::cout << *it << "␣";

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

Παρατηρήστε ότι η εντολή

```
  c.remove_if(lessthan10);
```

μπορεί να αντικατασταθεί από την

```
  c.remove_if(std::bind2nd(std::less<double>(), 10.0));
```

αρκεί να συμπεριλάβουμε το `<functional>`. Με τη δεύτερη εκδοχή, αποφεύγουμε να ορίσουμε δική μας συνάρτηση, την `lessthan10()`, και έχουμε μεγαλύτερη ευελιξία καθώς η κλήση της με άλλη σταθερή τιμή δεν απαιτεί νέα συνάρτηση.

5.2.6 set και multiset

Τα `set` και το `multiset` είναι containers που αποθηκεύουν τα στοιχεία τους με συγκεκριμένη σειρά, ανάλογα με 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.

Η διαφορά των δύο containers είναι ότι το `multiset` μπορεί να δεχτεί περισσότερα από ένα στοιχεία με ίδια τιμή ενώ το `set` αγνοεί τυχόν προσπάθειες να εισάγουμε στοιχείο που ήδη υπάρχει στη συλλογή. Με άλλα λόγια, στο `set` τα στοιχεία είναι μοναδικά. *Ό,τι θα αναφέρουμε παρακάτω για `set` ισχύει και για `multiset`.*

Το ιδιαίτερο πλεονέκτημα αυτών των containers, όπως και των άλλων της ίδιας κατηγορίας, έναντι των sequence containers δεν είναι τόσο η αυτόματη ταξινόμηση όσο η ταχύτητα που συνεπάγεται αυτή κατά την αναζήτηση στοιχείου με συγκεκριμένη ιδιότητα. Η δυαδική αναζήτηση (binary search) που μπορεί να χρησιμοποιηθεί στους associative containers είναι τάξης $O(\log n)$ ενώ η γραμμική που πρέπει να εφαρμοστεί στους sequence containers είναι $O(n)$.

Η χρήση των `std::set` και `std::multiset` προϋποθέτει τη συμπερίληψη του `<set>`.

Ορισμός

Δήλωση μεταβλητής τύπου `set` με στοιχεία τύπου T γίνεται με ένα από τους παρακάτω τρόπους.[4] Καθώς τους έχουμε ήδη αναλύσει στο §5.2.1 θα τους επαναλάβουμε περιληπτικά:

- Η εντολή

  ```
  std::set<T> c;
  ```

  δημιουργεί κενό `set`.

- Η εντολή

  ```
  std::set<T> c1(c2);
  ```

  δημιουργεί ένα `set` αντίγραφο άλλου.

- Η εντολή

[4] Δε θα αναφερθούμε σε κάποια παραλλαγή που υπάρχει.

`std::set<T> c(beg,end);`

δημιουργεί ένα `set` με 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>` που συγκρίνει τα στοιχεία με τον τελεστή (<) (αύξουσα σειρά). Γενικά, μπορούμε να περάσουμε ως δεύτερη παράμετρο του template *του τύπο* ενός αντικειμένου-συνάρτηση 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 τη μορφή

```
std::set<T, std::greater<T> > c;
```

Προσέξτε ότι στο template πρέπει να δοθεί ως δεύτερη παράμετρος ένας *τύπος*· αυτό αποκλείει την απλή συνάρτηση.

Το κριτήριο ταξινόμησης, f(a,b), πρέπει να ικανοποιεί τις ακόλουθες συνθήκες:

- Το f(x,x) είναι **`false`**.
- Αν το f(x,y) είναι **`true`** τότε το f(y,x) είναι **`false`**.
- Αν το f(x,y) και το f(y,z) είναι **`true`**, ισχύει και ότι το f(x,z) είναι **`true`**.
- Αν x και y είναι *ισοδύναμα*, δηλαδή τα f(x,y) και f(y,x) είναι **`false`**, και y και z είναι *ισοδύναμα*, δηλαδή τα f(y,z) και f(z,y) είναι **`false`**, τότε και τα x και z είναι *ισοδύναμα*, δηλαδή τα f(x,z) και f(z,x) είναι **`false`**.

Προσθήκη στοιχείων

Εισαγωγή στοιχείων σε ένα `set` γίνεται με τους ακόλουθους τρόπους:

- `c1 = c2;`: Αντιγράφει τα στοιχεία του c2 στο c1 καταστρέφοντας τα αρχικά.
- `c1.swap(c2);` ή `std::swap(c1,c2);`: Εναλλάσσει τα στοιχεία των c1, c2.
- `c.insert(elem);`: Εισάγει στο `set` ή `multiset` c αντίγραφο του elem. Αν το c είναι `multiset` επιστρέφει iterator στη θέση του νέου στοιχείου. Αν το c είναι `set` επιστρέφει ζεύγος (pair, §5.1.1), το πρώτο στοιχείο του οποίου είναι iterator στη θέση του νέου ή του ήδη υπάρχοντος στοιχείου ενώ το δεύτερο είναι **`bool`** που δείχνει αν η εισαγωγή ήταν επιτυχής ή όχι.

- `c.insert(pos,elem);`: Επιχειρεί να εισάγει αντίγραφο του elem, λαμβάνοντας υπόψη την *υπόδειξη* του iterator pos, και επιστρέφει iterator στο νέο ή υπάρχον στοιχείο.

- `c.insert(beg,end);`: Εισάγει στο c αντίγραφα των στοιχείων στο διάστημα [beg,end). Δεν επιστρέφει τίποτε.

Διαγραφή στοιχείων

Διαγραφή στοιχείων από ένα `set` c γίνεται με τους ακόλουθους τρόπους:

- `c.erase(elem);`: Διαγράφει όλα τα στοιχεία με τιμή elem και επιστρέφει το πλήθος των διαγραμμένων

- `c.erase(pos);`: Διαγράφει το στοιχείο στη θέση pos. Δεν επιστρέφει τίποτε.

- `c.erase(beg,end);`: Διαγράφει τα στοιχεία με iterators στο διάστημα [beg,end). Δεν επιστρέφει τίποτε.

- `c.clear();`: Διαγράφει όλα τα στοιχεία του c.

Προσπέλαση στοιχείων

Ο τρόπος αποθήκευσης σε ένα `set` καθορίζεται αποκλειστικά από τις σχετικές τιμές των στοιχείων. Αυτό έχει ως συνέπεια να μην επιτρέπεται να αλλάξουμε τιμή σε ένα στοιχείο καθώς θα αλλοιώσουμε τη σειρά ταξινόμησης. Έτσι, 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 εισάγουμε με τη νέα τιμή.

Επιπλέον συναρτήσεις-μέλη

Οι κλάσεις `set` και `multiset`, όπως και όλοι οι containers, παρέχουν τις `size()`, `empty()` και `max_size()` που περιγράψαμε προηγουμένως (§5.2.1), όπως και τις συναρτήσεις που επιστρέφουν iterators. Καθώς είναι βελτιστοποιημένες για γρήγορη αναζήτηση στοιχείων, παρέχουν ως μέλη συναρτήσεις που εκτελούν λειτουργίες κάποιων γενικών αλγορίθμων που θα δούμε παρακάτω αλλά πολύ πιο γρήγορα. Είναι οι εξής:

- `count(elem)`: Επιστρέφει το πλήθος των στοιχείων με τιμή elem.

- `find(elem)`: Επιστρέφει iterator στη θέση του στοιχείου με τιμή elem ή, αν δεν υπάρχει, `end()`.

- `lower_bound(elem)`: Επιστρέφει τη θέση του πρώτου στοιχείου που δεν είναι μικρότερο από το elem.
- `upper_bound(elem)`: Επιστρέφει τη θέση του πρώτου στοιχείου που είναι μεγαλύτερο από το elem.
- `equal_range(elem)`: Επιστρέφει, σε ζεύγος, τους iterators που προσδιορίζουν το διάστημα όπου τα στοιχεία είναι ίσα με elem.

Παράδειγμα:

```
#include <set>
#include <iostream>
#include <functional>

int
main() {
    typedef std::set<int, std::greater<int> > Set;

    // define empty set.
    Set c;

    // insert values in random order.

    c.insert(5);
    c.insert(12);
    c.insert(3);
    c.insert(6);
    c.insert(7);
    c.insert(1);
    c.insert(9);

    // print set
    std::cout << "Number␣of␣elements␣in␣set:␣"
              << c.size() << '\n';

    for (Set::const_iterator it = c.begin(); it != c.end(); ++it)
        std::cout << *it << "␣";

    std::cout << '\n';

    // remove elements with value 4, 6 and print information
    int howmany = c.erase(4);
    std::cout << "There␣were␣" << howmany
              << "␣elements␣with␣value␣4\n";

    howmany = c.erase(6);
    std::cout << "There␣were␣" << howmany
              << "␣elements␣with␣value␣6\n";
```

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

    typedef std::multiset<int, std::greater<int> > MSet;

    // create multiset from c
    MSet mc(c);

    //insert element 12 three times
    mc.insert(12);
    mc.insert(12);
    mc.insert(12);

    // print multiset
    std::cout << "Number␣of␣elements␣in␣multiset:␣"
              << mc.size() << '\n';

    for (MSet::const_iterator it = mc.begin();
         it != mc.end(); ++it)
        std::cout << *it << "␣";

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

5.2.7 map και multimap

Τα `map` και `multimap` είναι containers που αποθηκεύουν ζεύγη (§5.1.1) ποσοτήτων στα οποία το πρώτο μέλος (first) έχει το ρόλο του "κλειδιού" και το δεύτερο (second) είναι η αντίστοιχη τιμή. Στα ζεύγη γίνεται αυτόματη ταξινόμηση με βάση την τιμή του "κλειδιού" τους. 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 εισάγουμε στοιχείο με "κλειδί" που ήδη υπάρχει στη συλλογή. *Ό,τι θα αναφέρουμε παρακάτω για `map` ισχύει και για `multimap`.*

Η κλάση `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.

Η χρήση των `std::map` και `std::multimap` προϋποθέτει τη συμπερίληψη του `<map>`.

Ορισμός

Μεταβλητή τύπου `map` μπορεί να οριστεί με τους γνωστούς τρόπους. Στα παρακάτω, η πρώτη παράμετρος του template, K, προσδιορίζει τον τύπο του "κλειδιού" των στοιχείων που θα αποθηκευθούν ενώ η δεύτερη, T, είναι ο τύπος της συνοδεύουσας ποσότητας.[5]

[5] Δε θα αναφερθούμε σε κάποια παραλλαγή που υπάρχει.

- Η εντολή

  ```
  std::map<K,T> c;
  ```

  δημιουργεί κενό `map`.

- Η εντολή

  ```
  std::map<K,T> c1(c2);
  ```

  δημιουργεί ένα `map` αντίγραφο άλλου, ίδιου τύπου.

- Η εντολή

  ```
  std::map<K,T> c(beg,end);
  ```

  δημιουργεί ένα `map` με στοιχεία από κάποιο άλλο map, πιθανώς διαφορετικού τύπου, με iterators που βρίσκονται στο διάστημα [beg,end).

Στους παραπάνω ορισμούς η ταξινόμηση γίνεται με το προκαθορισμένο κριτήριο, το `std::less<K>` που συγκρίνει τα "κλειδιά" των στοιχείων με τον τελεστή (<) (αύξουσα σειρά). Γενικά, μπορούμε να περάσουμε ως τρίτη παράμετρο του template *τον τύπο* ενός αντικειμένου-συνάρτηση που θα δέχεται δύο ορίσματα και θα επιστρέφει λογική τιμή, **`true`**/**`false`**, ανάλογα αν το πρώτο είναι "μικρότερο" ή όχι από το δεύτερο. Οι συνθήκες που πρέπει να ικανοποιεί αυτό δόθηκαν στην αντίστοιχη περιγραφή του `set`, §5.2.6.

Προσθήκη στοιχείων

Εισαγωγή στοιχείων σε ένα `map` γίνεται με τους ακόλουθους τρόπους:

- `c1 = c2;`: Αντιγράφει τα στοιχεία του c2 στο c1 καταστρέφοντας τα αρχικά.

- `c1.swap(c2);` ή `std::swap(c1,c2);`: Εναλλάσσει τα στοιχεία των c1, c2.

- `c.insert(elem);`: Εισάγει στο `map` ή `multimap` c αντίγραφο του elem. Θυμηθείτε ότι το elem είναι `pair`· πρέπει, επομένως, να κατασκευαστεί κατάλληλα (§5.1.1). Αν το c είναι `multimap` επιστρέφει iterator στη θέση του νέου στοιχείου. 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 **`bool`** που δείχνει αν η εισαγωγή ήταν επιτυχής ή όχι.

- `c.insert(pos,elem);`: Επιχειρεί να εισάγει αντίγραφο του elem, λαμβάνοντας υπόψη την *υπόδειξη* του iterator pos, και επιστρέφει iterator στο νέο ή υπάρχον στοιχείο.

- `c.insert(beg,end);`: Εισάγει στο c αντίγραφα των στοιχείων στο διάστημα [beg,end). Δεν επιστρέφει τίποτε.

Διαγραφή στοιχείων

Διαγραφή στοιχείων από ένα `map` c γίνεται με τους ακόλουθους τρόπους:

- `c.erase(key);`: Διαγράφει όλα τα στοιχεία με τιμή "κλειδιού" key και επιστρέφει το πλήθος των διαγραμμένων.
- `c.erase(pos);`: Διαγράφει το στοιχείο στη θέση pos. Δεν επιστρέφει τίποτε.
- `c.erase(beg,end);`: Διαγράφει τα στοιχεία με iterators στο διάστημα [beg,end). Δεν επιστρέφει τίποτε.
- `c.clear();`: Διαγράφει όλα τα στοιχεία του c.

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 `(*it).first` (ή, ισοδύναμα, `it->first`) είναι το "κλειδί" και `(*it).second` (ή, ισοδύναμα, `it->second`) είναι η συνοδεύουσα ποσότητα.

Στην περίπτωση που θέλουμε να τροποποιήσουμε το "κλειδί" κάποιου στοιχείου πρέπει να το διαγράψουμε και κατόπιν να το εισάγουμε με νέα τιμή.

Μια σημαντική διαφορά του `map` (μόνο, και όχι του `multimap`) από το `set` είναι ότι παρέχεται άμεση πρόσβαση στις συνοδεύουσες ποσότητες και μάλιστα με μηχανισμό (τις αγκύλες `[]`) που θυμίζει τους πίνακες. Όμως, σε αντίθεση με τους πίνακες, ο δείκτης που τοποθετείται μεταξύ των αγκυλών 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]` είναι *αναφορά* σε αυτό το στοιχείο.
- αν c είναι ένα `map` που *δεν* περιέχει στοιχείο με "κλειδί" key τότε η έκφραση

  ```
  c[key] = value;
  ```

  εισάγει στο c το στοιχείο `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 `map`. Αυτή θα είναι και η τιμή που θα τυπωθεί στο παραπάνω παράδειγμα.

Παράδειγμα:

Παράδειγμα ορισμού, εισαγωγής και προσπέλασης στοιχείων ενός `map` είναι το ακόλουθο.

```
#include <iostream>
#include <string>
#include <map>
#include <utility>

int
main() {
    typedef std::map<std::string, int> Map;

    Map birthyear;   // Empty map.

    // insert a few pairs.
    birthyear.insert(std::make_pair("John", 1940));
    birthyear.insert(std::make_pair("Paul", 1942));
    birthyear.insert(std::make_pair("George", 1943));

    std::cout << "John␣was␣born␣in␣"
              << birthyear["John"] << "\n\n";

    // insert new data
    birthyear["Ringo"] = 1941; // wrong value, change it
    birthyear["Ringo"] = 1940;

    // print all pairs
    for (Map::const_iterator it = birthyear.begin();
         it != birthyear.end();
         ++it)
        std::cout << it->first  << "␣was␣born␣in␣"
                  << it->second << '\n';
}
```

Επιπλέον συναρτήσεις-μέλη

Οι κλάσεις `map` και `multimap`, παρέχουν τις γνωστές συναρτήσεις-μέλη που περιγράψαμε προηγουμένως (§5.2.1), `size()`, `empty()` και `max_size()` καθώς και τις συναρτήσεις που επιστρέφουν iterators.

Καθώς είναι βελτιστοποιημένες για γρήγορη αναζήτηση στοιχείων παρέχουν τις ίδιες εξειδικευμένες συναρτήσεις που παρέχουν και οι `set/multiset`:

- `count(key)`: Επιστρέφει το πλήθος των στοιχείων με "κλειδί" key.
- `find(key)`: Επιστρέφει iterator στη θέση του στοιχείου με "κλειδί" key ή, αν δεν υπάρχει, `end()`.
- `lower_bound(key)`: Επιστρέφει τη θέση του πρώτου στοιχείου του οποίου το "κλειδί" δεν είναι μικρότερο από το key.

- `upper_bound(key)`: Επιστρέφει τη θέση του πρώτου στοιχείου το "κλειδί" του οποίου είναι μεγαλύτερο από το key.
- `equal_range(key)`: Επιστρέφει, σε ζεύγος, τους iterators που προσδιορίζουν το διάστημα όπου τα στοιχεία έχουν "κλειδιά" ίσα με key.

5.3 Αλγόριθμοι (algorithms)

Η STL παρέχει ένα μεγάλο πλήθος αλγορίθμων για την επεξεργασία συλλογών οποιουδήποτε είδους στοιχείων. Η επεξεργασία συνίσταται σε συνήθεις, θεμελιώδεις πράξεις: αντιγραφή ή τροποποίηση συλλογών, αναζήτηση στοιχείων με συγκεκριμένη ιδιότητα, ταξινόμηση, αναδιάταξη στοιχείων, κλπ.

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 random iterators (§5.2.2). Θυμηθείτε ότι τέτοιοι δεν παρέχονται από πολλές κλάσεις containers. Σε τέτοιες περιπτώσεις υπάρχουν συναρτήσεις–μέλη αυτών των κλάσεων που εκτελούν την απαιτούμενη λειτουργία.

Η πλειοψηφία των αλγορίθμων περιλαμβάνονται στο header `<algorithm>`. Οι λίγοι που παρέχονται από το `<numeric>` θα επισημαίνονται. Όλοι ορίζονται στο χώρο ονομάτων `std`.

Όλοι οι αλγόριθμοι δέχονται ως πρώτα ορίσματα δύο iterators που καθορίζουν το διάστημα σε ένα container 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 που "δείχνει" ο δεύτερος. Ανάλογα με τη λειτουργία κάθε αλγορίθμου μπορεί να χρειάζεται να προσδιοριστεί και δεύτερο διάστημα σε ένα container. Σε τέτοια περίπτωση περνά μόνο ο iterator της αρχής και ο προγραμματιστής πρέπει να φροντίσει να ακολουθούν αρκετές θέσεις στο δεύτερο container ώστε να χωρά όσα θα γράψει εκεί ο αλγόριθμος. Επιπλέον, 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`)[6].

Ας περιγράψουμε τους πιο σημαντικούς αλγορίθμους:

[6]Παραδείγματα χρήσης τους θα βρείτε στις περιγραφές των `accumulate`, `count`.

• **accumulate()**

```
Type accumulate(Iterator beg, Iterator end, Type value)

Type accumulate(Iterator beg, Iterator end, Type value,
                BinaryFunctor op)
```

Η πρώτη μορφή επιστρέφει το άθροισμα των στοιχείων του διαστήματος [beg,end) και της τιμής value. Για μία σειρά στοιχείων {a1, a2, a3, ...} επιστρέφει το

```
value + a1 + a2 + a3 + ...
```

Στη δεύτερη μορφή δέχεται ως επιπλέον τελευταίο όρισμα ένα αντικείμενο-συνάρτηση με δύο ορίσματα για να προσδιορίσει άλλη εκτελούμενη πράξη.

Παρέχεται από το <numeric>.

Παράδειγμα:

Το άθροισμα των στοιχείων ενός std::vector v με πραγματικά στοιχεία μπορεί να υπολογιστεί ως εξής:

```
double sum = 0.0;
for (std::size_t i = 0; i < v.size(); ++i)
    sum += v[i];
```

Με τη χρήση της accumulate, ο παραπάνω κώδικας απλοποιείται στον

```
double sum = std::accumulate(v.begin(), v.end(), 0.0);
```

Αν επιθυμούμε να υπολογίσουμε το *γινόμενο* των στοιχείων του v, μπορούμε να χρησιμοποιήσουμε ένα βρόχο

```
double prod = 1.0;
for (std::size_t i = 0; i < v.size(); ++i)
    prod *= v[i];
```

ή, πιο απλά (και πιθανόν πιο γρήγορα) την εντολή

```
double prod = std::accumulate(v.begin(), v.end(), 1.0,
                              std::multiplies<double>());
```

Προσέξτε ότι χρησιμοποιήθηκε η δεύτερη μορφή της accumulate και το κατάλληλο αντικείμενο-συνάρτηση, §5.1.2.

• **partial_sum()**

```
Iterator
partial_sum(Iterator beg1, Iterator end1, Iterator beg2)

Iterator
partial_sum(Iterator beg1, Iterator end1, Iterator beg2,
            BinaryFunctor op)
```

Ο συγκεκριμένος αλγόριθμος υπολογίζει το *μερικό άθροισμα* (στην πρώτη μορφή) ή το *γενικευμένο μερικό άθροισμα* (στη δεύτερη). Η πρώτη μορφή αναθέτει:

- 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 [beg1,end1).

Η δεύτερη μορφή αντί για *άθροισμα* των σχετικών στοιχείων, αναθέτει την επιστρεφόμενη τιμή του αντικειμένου-συνάρτηση op(), δηλαδή, στο στοιχείο στο beg2+i+1 αναθέτει το op(*(beg2+i), *(beg1+i+1)).

Ο beg2 μπορεί να "δείχνει" στον ίδιο container που "δείχνουν" οι beg1, end1.

Παρέχεται από το <numeric>.

- **inner_product()**

```
Type
inner_product(Iterator beg1, Iterator end1,
              Iterator beg2, Type value)

Type
inner_product(Iterator beg1, Iterator end1,
              Iterator beg2, Type value,
              BinaryFunctor op1, BinaryFunctor op2)
```

Η πρώτη μορφή επιστρέφει το άθροισμα της value και του εσωτερικού γινομένου των στοιχείων στο [beg1,end1) με τα αντίστοιχά τους στο διάστημα με αρχή το beg2. Για δύο σειρές στοιχείων {a1, a2, a3, ...} και {b1, b2, b3, ...} επιστρέφει το

```
value + a1 * b1 + a2 * b2 + a3 * b3 + ...
```

Στη δεύτερη μορφή μπορεί να δεχτεί ως επιπλέον ορίσματα δύο δυαδικά αντικείμενα-συναρτήσεις, `op1(), op2()`. Τότε επιστρέφεται το

```
value op1 (a1 op2 b1) op1 (a2 op2 b2) op1 (a3 op2 b3) + ...
```

Παρέχεται από το <numeric>.

- **for_each()**

```
UnaryFunctor
for_each(Iterator beg, Iterator end, Unaryfunctor op)
```

Καλεί το αντικείμενο-συνάρτηση op() με όρισμα αναφορά σε κάθε στοιχείο του διαστήματος [beg,end). Αγνοεί τυχόν επιστρεφόμενη τιμή του op(). Επιστρέφει το αντικείμενο-συνάρτηση που έχει δημιουργηθεί με την κλήση του op(). Το αντικείμενο αυτό μπορεί να έχει τροποποιηθεί, να έχει αλλάξει η εσωτερική του κατάσταση.

- **count()**

```
Difference_type
count(Iterator beg, Iterator end, Type const & value)

Difference_type
count_if(Iterator beg, Iterator end, UnaryFunctor op)
```

Η πρώτη μορφή επιστρέφει τον αριθμό των στοιχείων στο [beg,end) που είναι ίσα με value. Θυμηθείτε ότι οι associative containers παρέχουν ανάλογη συνάρτηση-μέλος.

Η δεύτερη μετρά το πλήθος των στοιχείων για τα οποία η δράση του αντικειμένου-συνάρτηση op() επιστρέφει **true**.

Ο τύπος `Difference_type` είναι ακέραιος[1].

Παράδειγμα:

Το πλήθος των στοιχείων που είναι ίσα με μια τιμή a, ενός `std::vector` v με πραγματικά στοιχεία, μπορεί να βρεθεί ως εξής:

```
std::size_t cnt = 0;
for (std::size_t i = 0; i < v.size(); ++i)
    if (v[i] == a)
       ++cnt;
```

Εναλλακτικά (και πιο απλά και γρήγορα) μπορούμε να χρησιμοποιήσουμε τον αλγόριθμο `count` ως εξής:

```
std::size_t cnt = std::count(v.begin(), v.end(), a);
```

Η δεύτερη μορφή, το `count_if`, είναι χρήσιμη σε πιο σύνθετες "μετρήσεις". Το πλήθος των στοιχείων του v που είναι *μικρότερα* από μια τιμή a, μπορεί να υπολογιστεί ως εξής

```
std::size_t cnt =
    std::count_if(v.begin(), v.end(),
                  std::bind2nd(std::less<double>(), a));
```

Προσέξτε ότι χρησιμοποιήθηκε ένας *προσαρμογέας*, §5.2, για να τροποποιήσει ένα αντικείμενο-συνάρτηση με δύο ορίσματα ώστε να δέχεται ένα όρισμα: ο προσαρμογέας `std::less<double>()` που δέχεται δύο ορίσματα προσαρμόστηκε ώστε να συμπεριφέρεται ως αντικείμενο-συνάρτηση ενός ορίσματος· το δεύτερο απέκτησε την τιμή a με τη δράση του `std::bind2nd()`.

• **min_element()**

```
Iterator
min_element(Iterator beg, Iterator end)

Iterator
min_element(Iterator beg, Iterator end, BinaryFunctor op)
```

Επιστρέφει iterator στη θέση του μικρότερου στοιχείου (ή του πρώτου από όλα τα μικρότερα) στο διάστημα [beg,end). Στην πρώτη μορφή η σύγκριση των στοιχείων γίνεται με τον τελεστή (<) ενώ στη δεύτερη με βάση το αντικείμενο-συνάρτηση op(), το οποίο δέχεται δύο ορίσματα και επιστρέφει λογική τιμή.

• **max_element()**

```
Iterator
max_element(Iterator beg, Iterator end)

Iterator
max_element(Iterator beg, Iterator end, BinaryFunctor op)
```

Επιστρέφει iterator στη θέση του μεγαλύτερου στοιχείου (ή του πρώτου από όλα τα μεγαλύτερα) στο διάστημα [beg,end). Στην πρώτη μορφή η σύγκριση των στοιχείων γίνεται με τον τελεστή (<) ενώ στη δεύτερη με βάση το αντικείμενο-συνάρτηση op(), το οποίο δέχεται δύο ορίσματα και επιστρέφει λογική τιμή.

• **find()**

```
Iterator
find(Iterator beg, Iterator end, Type const & value)

Iterator
find_if(Iterator beg, Iterator end, UnaryFunctor op)
```

Η πρώτη μορφή επιστρέφει iterator στη θέση του πρώτου στοιχείου στο [beg,end) που είναι ίσο με value. Θυμηθείτε ότι οι associative containers παρέχουν ανάλογη συνάρτηση-μέλος που είναι πιο γρήγορη.

Η δεύτερη επιστρέφει iterator στη θέση του πρώτου στοιχείου για το οποίο η δράση του αντικειμένου-συνάρτηση op() επιστρέφει **true**.

Και στις δύο μορφές επιστρέφεται το end αν δεν υπάρχει στοιχείο που να ικανοποιεί την αντίστοιχη συνθήκη.

Αν το διάστημα [beg,end) είναι ταξινομημένο, υπάρχουν πιο γρήγορες συναρτήσεις για την εύρεση στοιχείου (`lower_bound()`, `upper_bound()`, `equal_range()`, `binary_search()`).

• **equal()**

```
bool
equal(Iterator beg1, Iterator end1, Iterator beg2)

bool
equal(Iterator beg1, Iterator end1, Iterator beg2,
      BinaryFunctor op)
```

Η πρώτη μορφή επιστρέφει **true** αν όλα τα στοιχεία στο [beg1,end1) είναι ίσα με τα αντίστοιχά τους στο ίσου μήκους διάστημα που ξεκινά από το beg2. Αν κάποια στοιχεία διαφέρουν, επιστρέφει **false**.

Στη δεύτερη μορφή η σύγκριση των στοιχείων γίνεται με το αντικείμενο-συνάρτηση op(), το οποίο δέχεται δύο ορίσματα και επιστρέφει λογική τιμή. Αν για κάποιο ζεύγος στοιχείων, η op() επιστρέψει **false**, η `equal` επιστρέφει **false** αλλιώς επιστρέφει **true**.

• **copy()**

```
Iterator
copy(Iterator beg1, Iterator end1, Iterator beg2)
```

Αντιγράφει τα στοιχεία του διαστήματος [beg1, end1) στο διάστημα που ξεκινά με το beg2. Επιστρέφει iterator στο επόμενο στοιχείο από το τελευταίο στο οποίο έγινε εγγραφή.

Το beg2 μπορεί να "δείχνει" στον ίδιο container αλλά δεν πρέπει να ανήκει στο διάστημα [beg1, end1).

• **copy_backward()**

```
Iterator
copy_backward(Iterator beg1, Iterator end1, Iterator end2)
```

Αντιγράφει τα στοιχεία του διαστήματος [beg1, end1) στο διάστημα που τελειώνει με το end2, προχωρώντας ανάστροφα. Επιστρέφει iterator στο επόμενο στοιχείο από το τελευταίο στο οποίο έγινε εγγραφή.

Το end2 μπορεί να "δείχνει" στον ίδιο container αλλά δεν πρέπει να ανήκει στο διάστημα [beg1, end1).

• **swap_ranges()**

```
Iterator
swap_ranges(Iterator beg1, Iterator end1, Iterator beg2)
```

Εναλλάσσει κάθε στοιχείο στο διάστημα [beg1, end1) με το αντίστοιχό του στο διάστημα που ξεκινά από το beg2 (δηλαδή το [beg2, beg2+(end1-beg1))).

• **transform()**

```
Iterator
transform(Iterator beg1, Iterator end1, Iterator beg2,
          UnaryFunctor op)

Iterator
transform(Iterator beg1, Iterator end1, Iterator beg2,
          Iterator 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.

• **fill()**

```
void
fill(Iterator beg, Iterator end, Type const & value)

void
fill_n(Iterator beg, Size N, Type const & value)
```

Η πρώτη μορφή αναθέτει την τιμή value στα στοιχεία του διαστήματος [beg,end). Η δεύτερη αναθέτει την τιμή value σε N διαδοχικά στοιχεία ξεκινώντας από το beg.

• **generate()**

```
void
generate(Iterator beg, Iterator end, Functor op)

void
generate_n(Iterator beg, Size N, Functor op)
```

Η πρώτη μορφή αναθέτει την επιστρεφόμενη τιμή του αντικειμένου-συνάρτηση op() (το οποίο δε δέχεται ορίσματα) στα στοιχεία του διαστήματος [beg,end). Η δεύτερη αναθέτει την επιστρεφόμενη τιμή του op() σε N διαδοχικά στοιχεία ξεκινώντας από το beg.

• **replace()**

```
void
replace(Iterator beg, Iterator end,
        Type const & oldvalue, Type const & newvalue)

void
replace_if(Iterator beg, Iterator end,
           UnaryFunctor op, Type const & newvalue)
```

Η πρώτη μορφή αναθέτει την τιμή newvalue σε κάθε στοιχείο στο [beg,end) που είναι ίσο με oldvalue. Στη δεύτερη μορφή τα επιλεγόμενα στοιχεία για αντικατάσταση είναι αυτά για τα οποία το αντικείμενο-συνάρτηση op επιστρέφει **true**.

• **replace_copy()**

```
void
replace_copy(Iterator beg1, Iterator end1, Iterator beg2,
             Type const & oldvalue, Type const & newvalue)

void
replace_copy_if(Iterator beg1, Iterator end1, Iterator beg2,
                UnaryFunctor op, Type const & newvalue)
```

Η πρώτη μορφή αντιγράφει τα στοιχεία του [beg1,end1) στο διάστημα που ξεκινά με το beg2 μετατρέποντας σε newvalue όσα είναι ίσα με oldvalue. Στη δεύτερη μορφή τα επιλεγόμενα στοιχεία για αντικατάσταση είναι αυτά για τα οποία το αντικείμενο-συνάρτηση op επιστρέφει **true**.

• **remove()**

```
Iterator
remove(Iterator beg, Iterator end, Type const & value)

Iterator
remove_if(Iterator beg, Iterator end, UnaryFunctor op)
```

Η πρώτη μορφή διαγράφει όσα στοιχεία του διαστήματος [beg,end) είναι ίσα με value. Στη δεύτερη μορφή τα επιλεγόμενα στοιχεία για διαγραφή είναι αυτά για τα οποία το αντικείμενο-συνάρτηση op επιστρέφει **true**. Τα κενά καλύπτονται με μετακίνηση των επόμενων στοιχείων.

Η επιστρεφόμενη τιμή είναι iterator στην πρώτη θέση μετά το τελευταίο στοιχείο που δεν διαγράφηκε. Συνεπώς, το διάστημα από beg μέχρι τον επιστρεφόμενο iterator περιλαμβάνει τα μη διαγραμμένα στοιχεία.

Καθώς οι συγκεκριμένοι αλγόριθμοι μεταβάλλουν στοιχεία, δεν μπορούν να χρησιμοποιηθούν σε associative containers. Αυτοί παρέχουν παρόμοια συνάρτηση-μέλος (`erase()`).

Προσέξτε ότι η `list` παρέχει αντίστοιχη συνάρτηση-μέλος (`remove()`) πιο γρήγορη.

- **`reverse()`**

```
void
reverse(Iterator beg, Iterator end)
```

Αναστρέφει τη σειρά των στοιχείων του διαστήματος [beg,end).

Προσέξτε ότι η `list` παρέχει γρηγορότερη συνάρτηση-μέλος.

- **`reverse_copy()`**

```
Iterator
reverse(Iterator beg1, Iterator end1, Iterator beg2)
```

Αναστρέφει τη σειρά των στοιχείων του διαστήματος [beg1,end1) αντιγράφοντάς τα στο διάστημα που ξεκινά με το beg2. Επιστρέφει τη θέση μετά το τελευταίο στοιχείο που γράφτηκε στο νέο container.

- **`merge()`**

```
Iterator
merge(Iterator beg1, Iterator end1,
      Iterator beg2, Iterator end2, Iterator beg3)

Iterator
merge(Iterator beg1, Iterator end1,
      Iterator beg2, Iterator end2, Iterator beg3,
      BinaryFunctor op)
```

Η πρώτη μορφή συγχωνεύει τα *ταξινομημένα* διαστήματα [beg1,end1) και [beg2,end2) στο διάστημα που ξεκινά με beg3, πάλι ταξινομημένα. Η δεύτερη χρησιμοποιεί το αντικείμενο-συνάρτηση op() για την ταξινόμηση· αυτό δέχεται δύο ορίσματα και επιστρέφει **`true`** ή **`false`** αν το πρώτο είναι ή όχι "μικρότερο" από το δεύτερο.

Και οι δύο επιστρέφουν τη θέση μετά το τελευταίο στοιχείο που γράφτηκε στο νέο container.

Προσέξτε ότι η `list` παρέχει γρηγορότερη συνάρτηση-μέλος.

- **`set_union()`**

```
Iterator
set_union(Iterator beg1, Iterator end1,
          Iterator beg2, Iterator end2, Iterator beg3)

Iterator
```

```
set_union(Iterator beg1, Iterator end1,
          Iterator beg2, Iterator end2, Iterator beg3,
          BinaryFunctor op)
```

Εκτελούν όμοιες λειτουργίες με τη `merge()` με τη διαφορά ότι από τα κοινά στοιχεία στα δύο διαστήματα αντιγράφεται μόνο ένα στον νέο διάστημα. Προσέξτε ότι αν σε κάποιο διάστημα ένα στοιχείο επαναλαμβάνεται, θα εμφανιστεί με το ίδιο πλήθος και στο τελικό διάστημα.

- **`set_intersection()`**

```
Iterator
set_intersection(Iterator beg1, Iterator end1,
                 Iterator beg2, Iterator end2,
                 Iterator beg3)

Iterator
set_intersection(Iterator beg1, Iterator end1,
                 Iterator beg2, Iterator end2,
                 Iterator beg3, BinaryFunctor op)
```

Εκτελούν όμοιες λειτουργίες με τη `merge()` με τη διαφορά ότι στο νέο διάστημα εμφανίζονται μόνο τα κοινά στοιχεία των δύο αρχικών διαστημάτων.

- **`set_difference()`**

```
Iterator
set_difference(Iterator beg1, Iterator end1,
               Iterator beg2, Iterator end2,
               Iterator beg3)

Iterator
set_difference(Iterator beg1, Iterator end1,
               Iterator beg2, Iterator end2,
               Iterator beg3, BinaryFunctor op)
```

Εκτελούν όμοιες λειτουργίες με τη `merge()` με τη διαφορά ότι στο νέο διάστημα εμφανίζονται μόνο τα στοιχεία που ανήκουν στο πρώτο διάστημα και δεν υπάρχουν στο δεύτερο.

- **`set_symmetric_difference()`**

```
Iterator
set_symmetric_difference(Iterator beg1, Iterator end1,
                         Iterator beg2, Iterator end2,
                         Iterator beg3)

Iterator
```

```
set_symmetric_difference(Iterator beg1, Iterator end1,
                         Iterator beg2, Iterator end2,
                         Iterator beg3, BinaryFunctor op)
```

Εκτελούν όμοιες λειτουργίες με τη `merge()` με τη διαφορά ότι στο νέο διάστημα εμφανίζονται μόνο τα στοιχεία που ανήκουν είτε στο πρώτο είτε στο δεύτερο διάστημα αλλά όχι και στα δύο.

- **`binary_search()`**

```
bool
binary_search(Iterator beg, Iterator end, Type const & value)

bool
binary_search(Iterator beg, Iterator end, Type const & value,
              BinaryFunctor op)
```

Η πρώτη μορφή επιστρέφει **`true`** ή **`false`** ανάλογα αν το *ταξινομημένο* διάστημα [beg,end) περιέχει στοιχείο ίσο με value.

Στη δεύτερη μορφή, το αντικείμενο-συνάρτηση op(), το οποίο δέχεται δύο ορίσματα και επιστρέφει λογική τιμή, προσδιορίζει το κριτήριο με το οποίο έγινε η ταξινόμηση.

- **`sort()`**

```
void
sort(RandomIterator beg, RandomIterator end)

void
sort(RandomIterator beg, RandomIterator end,
     BinaryFunctor op)
```

Και οι δύο μορφές ταξινομούν τα στοιχεία στο διάστημα [beg,end). Στην πρώτη, η σύγκριση των στοιχείων γίνεται με τον τελεστή (<) ενώ στη δεύτερη με βάση το αντικείμενο-συνάρτηση op(), το οποίο δέχεται δύο ορίσματα και επιστρέφει λογική τιμή.

Η σχετική θέση ίσων στοιχείων δεν διατηρείται απαραίτητα.

Προσέξτε ότι χρησιμοποιεί random iterators οπότε δεν μπορεί να εφαρμοστεί σε `list`. Θυμηθείτε ότι υπάρχει κατάλληλη συνάρτηση-μέλος της `list` που εκτελεί ταξινόμηση σε αντικείμενα τύπου `list`.

- **`stable_sort()`**

```
void
stable_sort(RandomIterator beg, RandomIterator end)

void
```

```
stable_sort(RandomIterator beg, RandomIterator end,
            BinaryFunctor op)
```

Ο συγκεκριμένος αλγόριθμος είναι ίδιος με τον `sort()` με τη διαφορά ότι η σχετική θέση ίσων στοιχείων διατηρείται.

- **`partial_sort()`**

```
void
partial_sort(RandomIterator beg, RandomIterator sortEnd,
             RandomIterator end)

void
partial_sort(RandomIterator beg, RandomIterator sortEnd,
             RandomIterator end, BinaryFunctor op)
```

Ο συγκεκριμένος αλγόριθμος είναι ίδιος με τον `sort()` αλλά σταματά την ταξινόμηση όταν τοποθετηθούν στο [beg,sortEnd) σωστά ταξινομημένα τα στοιχεία.

- **`random_shuffle()`**

```
void
random_shuffle(RandomIterator beg, RandomIterator end)
```

Αναδιατάσσει με τυχαίο τρόπο τα στοιχεία στο διάστημα [beg,end).

Προσέξτε ότι χρησιμοποιεί random iterators οπότε δεν μπορεί να εφαρμοστεί σε `list` ή associative containers.

5.4 Ασκήσεις

1. Γράψτε πρόγραμμα που να δημιουργεί ένα αρχείο με 1000000 τυχαίους ακέραιους αριθμούς στο διάστημα $[-1000000, 1000000]$.

2. Σε άλλο πρόγραμμα, διαβάστε το αρχείο σε container της επιλογής σας. Κατόπιν

   (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.

   (ϛ') 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 `clock()`[7] από το `<ctime>`). Τι συμπεραίνετε;

3. Γράψτε συναρτήσεις που να υλοποιούν τις ακόλουθες συναρτήσεις–μέλη της κλάσης `list`: `unique(), splice(), merge(), reverse()`.

4. Γράψτε συναρτήσεις που να υλοποιούν τους αλγορίθμους της STL. Σε όσες μπορείτε, κρατήστε το ίδιο interface, να δέχονται δηλαδή iterators. Χρονομετρήστε αυτές και τους αντίστοιχους αλγορίθμους. Τι συμπεραίνετε;

5. Υλοποιήστε τον αλγόριθμο quicksort, §4.11.1, ώστε να ταξινομεί τα στοιχεία ενός container μεταξύ δυο iterators. Συμπληρώστε, επομένως, τον παρακάτω κώδικα

```
template<typename Iterator>
void
quicksort(Iterator beg, Iterator end) {
...
}
```

6. Συμπληρώστε τον κώδικα της παρακάτω συνάρτησης

```
typedef std::list<int> 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 την συνάρτηση split. Τα στοιχεία κάθε νέας λίστας, αφού δημιουργηθεί, να τα γράψετε

[7] Σε συστήματα Unix η εντολή man 3 clock θα σας βοηθήσει.

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.

7. (α') Υλοποιήστε τον αλγόριθμο `std::equal()` σε δικιά 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 δικιά σας συνάρτηση `equal()` για να ελέγξετε αν είναι ίσες οι δύο ακολουθίες (προφανώς πρέπει να σας επιστρέψει **`false`**).

8. 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).

9. 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 6

Κλάσεις

6.1 Εισαγωγή

Στο κεφάλαιο αυτό θα παρουσιαστούν οι δομές που παρέχει η C++ για την υποστήριξη του "προγραμματισμού βασιζόμενου σε αντικείμενα" (object based) και του "προγραμματισμού προσανατολισμένου σε αντικείμενα" (object oriented programming). Ας δούμε όμως πρώτα, με τη βοήθεια ενός παραδείγματος, κάποιους λόγους που καθιστούν ιδιαίτερα επιθυμητές αυτές τις δομές για τη σχεδίαση και ανάπτυξη ενός προγράμματος που προσομοιώνει όσο πιο πιστά γίνεται, το εκάστοτε πρόβλημα.

6.1.1 Αρχικό στάδιο οργάνωσης

Ας υποθέσουμε ότι επιθυμούμε να περιγράψουμε, σε πρόγραμμα για υπολογιστή, τη δανειστική λειτουργία μιας πανεπιστημιακής Βιβλιοθήκης· σε πρώτη φάση, θέλουμε να καταγράψουμε τα βιβλία που υπάρχουν, 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, τον Αριθμό Μητρώου στη Σχολή τους αν είναι φοιτητές, το έτος εισαγωγής τους, κλπ.), και ένα πλήθος βιβλίων (που χαρακτηρίζονται από τον τίτλο τους, το συγγραφέα, τον εκδοτικό οίκο, το έτος έκδοσης, κλπ.), που μπορούν να δανειστούν. Η απλοϊκή (και σε ορισμένες γλώσσες προγραμματισμού η μόνη δυνατή) υλοποίηση περιλαμβάνει δηλώσεις ξεχωριστών πινάκων για την ομαδοποίηση του καθενός από τα χαρακτηριστικά που αναφέρθηκαν. Π.χ. σε C++ θα είχαμε

```
std::size_t const N = 50;  // number of students
std::string studentName[N];
int studentAM[N];
int studentYear[N];
// .......
```

```
std::size_t const M = 500;  // number of books
std::string bookTitle[M];
std::string bookAythor[M];
int bookYear[M];
// .......
```

Παρατηρήστε ότι τίποτε στους παραπάνω ορισμούς δεν εκφράζει τη σχέση που έχουν (αν έχουν) τα στοιχεία διαφορετικών πινάκων μεταξύ τους· τίποτε, εκτός πιθανόν από το όνομά τους, δεν υποδηλώνει ότι κάποια περιγράφουν χαρακτηριστικά της έννοιας "φοιτητής" και 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. Όχι μόνο δε γίνεται η ομαδοποίηση των χαρακτηριστικών στο επίπεδο των δύο 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 όλα τα ονόματα φοιτητών μαζί, όλοι οι αριθμοί μητρώου, κλπ. Η συγκεκριμένη οργάνωση των δεδομένων μας, και συνεπώς, και του υπόλοιπου κώδικά μας, είναι αρκετά διαφορετική από αυτή που υπαγορεύει το πρόβλημα που προσομοιώνουμε. Η μεθοδολογία αυτή δεν είναι λάθος, δυσκολεύει όμως την ανάπτυξη μεγάλων και πολύπλοκων προγραμμάτων.

Μια καλύτερη, πιο "φυσική" προσέγγιση είναι να ορίσουμε νέους τύπους ποσοτήτων που θα μπορούν να περιγράψουν *συνολικά* τα χαρακτηριστικά κάθε έννοιας. Είναι προφανές ότι οι ενσωματωμένοι τύποι δεν ανταποκρίνονται σε αυτή την απαίτηση: η έννοια "φοιτητής" δεν είναι ακέραιος ούτε πραγματικός, για αντικείμενο αυτής της έννοιας δεν ορίζεται η πρόσθεση ή η εξαγωγή τετραγωνικής ρίζας! Ανάμεσα στις γλώσσες προγραμματισμού που παρέχουν τη δυνατότητα ορισμού νέων τύπων είναι και η C++· βασιζόμενοι σε όσα έχουμε αναφέρει για **struct**, §2.6.2, μπορούμε να έχουμε τους ακόλουθους ορισμούς

```
struct Student {
    std::string name;
    int AM;
    int Year;
// ......
};

struct Book {
    std::string title;
    std::string author;
    int year;
// ......
};
```

Οι ορισμοί αυτοί περιλαμβάνουν, ως *μέλη* των δομών, μεταβλητές κατάλληλων τύπων και αποτελούν ένα πρώτο βήμα βελτίωσης του προγράμματος, καθώς ακολουθούν τον τρόπο οργάνωσης των δεδομένων του προβλήματός μας. Δήλωση ποσοτήτων ("*αντικειμένων*") αυτών των νέων τύπων γίνεται ως εξής:

```
std::size_t const N = 50;
std::size_t const M = 500;

Student students[N];
Book    books[M];
```

6.1.2 Ενθυλάκωση (encapsulation)

Η προσπέλαση των μελών των students, books που ορίσαμε στην προηγούμενη παράγραφο, είναι επιτρεπτή από οποιοδήποτε σημείο του κώδικα στην εμβέλεια των μεταβλητών students και books. Έτσι, τα στοιχεία κάποιου φοιτητή μπορούν να τυπωθούν με την εντολή

```
std::cout << "Name:␣" << students[10].name
          << "AM:␣"   << students[10].AM
          << "Year:␣" << students[10].year
          ......
```

Η ελεύθερη πρόσβαση στα μέλη των δομών που ορίσαμε, έχει ως συνέπεια να καθίσταται αναγκαία η εξέταση και πιθανή τροποποίηση όλων των σημείων του κώδικα στα οποία χρησιμοποιείται μια δομή —όταν προσπαθούμε να εντοπίσουμε και να διορθώσουμε κάποιο σφάλμα— ή στην περίπτωση που θελήσουμε να τροποποιήσουμε την εσωτερική αναπαράσταση της δομής (με προσθήκη ή διαγραφή μέλους, αλλαγή του ονόματος ή του τύπου κάποιου υπάρχοντος, κλπ.). Διευκολύνει πολύ την 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.

Ας προχωρήσουμε στο επόμενο στάδιο σχεδιασμού του κώδικά μας και ας αναπτύξουμε συναρτήσεις που θα παρουσιάζουν (π.χ. τυπώνοντας στη οθόνη) τα χαρακτηριστικά ενός αντικειμένου με τύπο `Student` και μιας ποσότητας τύπου `Book`. Οι δηλώσεις τους μπορούν να είναι οι

```
int printStudent(Student const & s);
int printBook(Book const & c);
```

Και πάλι συναντούμε την έλλειψη ισχυρής σύνδεσης της κάθε συνάρτησης με τον αντίστοιχο τύπο, παρόλο που υλοποιούν πράξεις αποκλειστικά σε αντικείμενα αυτών. Από την άποψη του μεταγλωττιστή είναι "ισότιμες" με οποιαδήποτε άλλη συνάρτηση του προγράμματός μας. Οι συγκεκριμένες συναρτήσεις δεν έχουν την επιθυμητή, αποκλειστική, πρόσβαση στα μέλη, έναντι

των άλλων συναρτήσεων του προγράμματός μας. Όλες μπορούν να προσπελάσουν αλλά και να τροποποιήσουν τις τιμές των "εσωτερικών" χαρακτηριστικών σε μεταβλητές τύπου `Student` ή `Book`.

Η δυνατότητα να διαχωρίζουμε τον τρόπο *υλοποίησης* (την εσωτερική αναπαράσταση) από τον τρόπο *χρήσης* μιας σύνθετης ποσότητας ονομάζεται *ενθυλάκωση (encapsulation)*. Αποτελεί ένα βασικό στοιχείο του προγραμματισμού προσανατολισμένου σε αντικείμενα.

Για την υποστήριξη της ενθυλάκωσης η C++ επεκτείνει τη δομή **struct** που κληρονόμησε από τη C και επιτρέπει να υπάρχουν συναρτήσεις ως μέλη μιας δομής, ξεχωρίζοντάς τες από τις υπόλοιπες συναρτήσεις του προγράμματος. Επιπλέον, με τη χρήση της δεσμευμένης λέξης **class** αντί για τη **struct** στη δήλωση της δομής, περιορίζει την εμβέλεια (τη δυνατότητα πρόσβασης και τροποποίησης δηλαδή) των μελών της κλάσης, μόνο στο "εσωτερικό" της: καμία "ξένη" συνάρτηση δεν έχει πρόσβαση σε αυτά, είτε είναι μεταβλητές είτε είναι συναρτήσεις, εκτός εάν ρητά δηλωθεί το αντίθετο.[1] Οι τύποι που ορίσαμε παραπάνω γίνονται πιο περιεκτικοί και ασφαλείς με τις ακόλουθες τροποποιήσεις

```
class Student {
    std::string name;
    int AM;
    int Year;
// ......

public:
    int print() {....}     // the old printStudent
    std::string getName() {....}
    int getAM() {....}
    int getYear() {....}
// ......
};


class Book {
public:
    int print() {....}  // the old printBook
    std::string getTitle() {....}
    std::string getAuthor() {....}
    int getYear() {....}
// ......
private:
    std::string title;
    std::string author;
    int year;
// ......
```

[1] Η δήλωση κλάσης με τη **struct** παρέχει ελεύθερη πρόσβαση στα μέλη εκτός αν ρητά περιοριστεί η πρόσβαση σε συγκεκριμένα.

```
};
```

Ας τις δούμε πιο αναλυτικά. Έχουμε μεταφέρει τις δηλώσεις (και βέβαια και τους ορισμούς, σε πρώτη φάση) των συναρτήσεων `printStudent()` και `printBook()` στο εσωτερικό του ορισμού των αντίστοιχων κλάσεων· έχουν απλοποιηθεί τα ονόματά τους (χωρίς να υπάρχει σύγκρουση καθώς ανήκουν σε διαφορετικές κλάσεις) και έχουν απαλειφθεί τα ορίσματά τους (δε χρειάζονται καθώς μπορούν να δράσουν σε συγκεκριμένο αντικείμενο της κλάσης στην οποία ανήκουν και μόνο). Θυμηθείτε ότι στα μέλη ενός αντικειμένου μίας **struct** έχουμε πρόσβαση με τον τελεστή (.). Το ίδιο ακριβώς ισχύει και για τις κλάσεις και για τις συναρτήσεις-μέλη αυτών. Έτσι, η κλήση της `print()` για μια ποσότητα s τύπου `Student` γίνεται ως εξής:

```
Student s;
 // .... give values to s
s.print();
```

Η πρόσβαση στα μέλη του αντικειμένου s μέσα στο σώμα της συνάρτησης `print()` μπορεί να γίνει απευθείας με το όνομά τους. Προσέξτε ότι στον ορισμό της κλάσης, η συγκεκριμένη συνάρτηση δηλώνεται μετά τη ετικέτα **public:**· αυτή και όσα άλλα μέλη την ακολουθούν μπορούν να κληθούν (αν πρόκειται για συναρτήσεις) ή να προσπελαστούν (αν πρόκειται για μεταβλητές) από οποιοδήποτε τμήμα του κώδικά μας έξω από την κλάση. Αντίθετα, τα μέλη που ορίζονται αμέσως μετά το αρχικό { της κλάσης ή μετά από την ετικέτα **private:** μπορούν να χρησιμοποιηθούν μόνο από συναρτήσεις-μέλη της ίδιας κλάσης.

Σύμφωνα με τα παραπάνω, τα μέλη που περιγράφουν ιδιότητες των δομών είναι πλέον απροσπέλαστα έξω από τις κλάσεις· έτσι, έχουμε προσθέσει συναρτήσεις που ελέγχουν την πρόσβαση και την απόδοση τιμής σε αυτά. Η υλοποίηση των συναρτήσεων αυτών είναι τετριμμένη, π.χ.

```
class Student {
    std::string name;
    int AM;
// ......
public:
    std::string getName() {return name;}
    void setAM(int i) { AM = i;}
// ......
};
```

6.1.3 Κληρονομικότητα – Πολυμορφισμός

Ας υποθέσουμε ότι επιθυμούμε σε δεύτερη φάση να επεκτείνουμε το πρόγραμμα και για άλλα μέλη της πανεπιστημιακής κοινότητας· οποιοσδήποτε ανήκει στο προσωπικό του Πανεπιστημίου μπορεί να δανειστεί βιβλία, με διαφορετι-

κές προϋποθέσεις, ανάλογα με το αν είναι φοιτητής, καθηγητής, ή εργαζόμενος. Θα μπορούσαμε να ορίσουμε κατάλληλες κλάσεις για την 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.

Οι γλώσσες που υποστηρίζουν *προγραμματισμό προσανατολισμένο σε αντικείμενα* μας επιτρέπουν 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", έχουν όλες τις ιδιότητές της (αυτό λέγεται *κληρονομικότητα, inheritance*) και μπορούν να χρησιμοποιηθούν όπου μπορεί εμφανιστεί αυτή. 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), και μπορεί να συμπεριφερθεί σαν καθεμία από αυτές τις έννοιες. 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 παρουσιάζουμε, 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.

Στη συνέχεια του κεφαλαίου θα παρουσιαστούν με λεπτομέρειες οι μηχανισμοί που παρέχει η C++ για την υλοποίηση όσων αναπτύχθηκαν παραπάνω.

6.2 Ορισμός κλάσης

Από την ανάλυση του προβλήματος που κάναμε παραπάνω, είδαμε 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, §2.5.4, των μεταβλητών: 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.

- τη λεπτομερή περιγραφή του τρόπου *καταστροφής* ενός αντικειμένου.

- 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.

Ας επαναλάβουμε, συμπληρωμένο και κατάλληλα τροποποιημένο, τον ορισμό της κλάσης με το όνομα Student· θα τον αναλύσουμε λεπτομερώς αυτή τη φορά.

```
class Student {
public:
    Student(char const * onoma, int am, int y)
        : name(onoma), AM(am), Year(y) {}

    ~Student() {}
    Student(Student const & other);

    Student & operator=(Student const & other);

    int print() const {
        std::cout << name << '␣' << AM << '␣' << Year << '\n';
        return 0;
    }
    std::string const & getName() const {return name;}
    int getAM() const {return AM;}
    int getYear() const {return Year;}

    void setName(char const * s) {name = s;}
    void setAM(int am) {AM = am;}
    void setYear(int y) {Year = y;}
private:
    std::string name;
    int AM;
    int Year;
};
```

6.2.1 Εσωτερική αναπαράσταση – συναρτήσεις πρόσβασης

Ο ορισμός της κλάσης (του νέου τύπου, δηλαδή) εισάγεται με μία από τις λέξεις **class** ή **struct**, ακολουθούμενης από το όνομα του τύπου που δημιουργούμε και, μέσα σε άγκιστρα {}, από τα μέλη του (που υλοποιούν χαρακτηριστικά και ιδιότητές του). Προσέξτε το απαραίτητο καταληκτικό (;) που ολοκληρώνει τον ορισμό.

Η συγκεκριμένη κλάση έχει τρία μέλη, τις ποσότητες name, AM, Year, στα οποία η πρόσβαση επιτρέπεται μόνο από άλλα μέλη της κλάσης. Ο περιορισμός αυτός υποδηλώνεται με την ετικέτα **private:** που προηγείται των δηλώσεών τους. Επιπλέον, η κλάση περιλαμβάνει αρκετά μέλη, τις συναρτήσεις print(), getName(), getAM(), getYear(), setName(), setAM(), setYear() μεταξύ άλλων, η πρόσβαση και χρήση των οποίων είναι επιτρεπτή από οπουδήποτε καθώς ο ορισμός τους έπεται της ετικέτας **public:**. 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

[2]Την τρίτη ετικέτα που επιτρέπεται, την **protected:**, θα τη συναντήσουμε αργότερα.

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**.

Παρατηρήστε τις συναρτήσεις–μέλη setName(), setAM(), setYear(), οι οποίες τροποποιούν άλλα μέλη του αντικειμένου για το οποίο καλούνται. Ο τρόπος ορισμού τους δε διαφέρει από τις κοινές συναρτήσεις. Όπως αναφέρθηκε, η πρόσβαση στα μέλη γίνεται χρησιμοποιώντας απευθείας τα ονόματά τους. Προσέξτε ότι μπορούν να αναφέρονται σε μέλη που ορίζονται αργότερα στην κλάση καθώς οποιαδήποτε δήλωση οπουδήποτε εντός κλάσης θεωρείται γνωστή για τα υπόλοιπα μέλη. Οι συγκεκριμένες συναρτήσεις, όπως είναι φυσικό, δεν μπορούν να κληθούν για αντικείμενα που θα δηλωθούν ως **const** καθώς αυτά δεν μπορούν να τροποποιηθούν. Αντίθετα, στις συναρτήσεις–μέλη getName(), getAM(), getYear(), οι οποίες δε μεταβάλουν 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 και για σταθερά αντικείμενα. Επομένως, αν το s είναι Student, στον παρακάτω κώδικα

```
Student & s1 = s;
Student const & s2 = s;

s1.setYear(2000);  // correct
s2.setYear(2000);  // error

int y = s1.getYear();  // correct
int z = 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**.

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.

Όπως καταλαβαίνετε, ο χρήστης μιας κλάσης, σε αντίθεση με τον προγραμματιστή της, ενδιαφέρεται μόνο για τις δηλώσεις των μελών (και, ειδικότερα, των **public**) και όχι για τους ορισμούς ή την υλοποίησή τους. Είναι επιθυμητό για λόγους ευκρίνειας, να μπορούμε να μετακινούμε τους ορισμούς

των συναρτήσεων-μελών εκτός του σώματος μιας κλάσης. Για παράδειγμα, στη Student μπορούμε να ορίσουμε τα μέλη print() και setYear() έξω από το κυρίως σώμα της ως εξής:

```
class Student {
public:
// ............
    int print() const;
    void setYear(int y);
// ............
private:
    std::string name;
    int AM;
    int Year;
};


int
Student::print() const {
    std::cout << name << '␣' << AM << '␣' << Year << '\n';
    return 0;
}

inline
void
Student::setYear(int y) {
    Year = y;
}
```

Παρατηρείστε 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**, §4.8· 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 και ορισμοί των μελών—μας επιτρέπει να συγκεντρώνουμε σε αρχείο header (π.χ. Student.h) το πρώτο τμήμα (*και όσες συναρτήσεις-μέλη είναι* ***inline*** *ή* ***template***, δείτε τις §4.8 και §4.7) και σε άλλο αρχείο (π.χ. Student.cc) 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.

6.2.2 Συναρτήσεις Δημιουργίας (Constructors—Copy constructor)

Παρατηρήστε ότι στην κλάση περιλαμβάνονται, μεταξύ άλλων, ως μέλη, κάποιες ειδικού τύπου συναρτήσεις:

- η `Student(char const * onoma, int am, int y)`,
- η `Student()`
- και η `Student(Student const & other)`.

Αυτές περιγράφουν τον τρόπο *δημιουργίας* ενός αντικειμένου της κλάσης Student από τρεις ανεξάρτητες ποσότητες (η πρώτη), από τίποτε (η δεύτερη) ή από ένα άλλο αντικείμενο τύπου `Student` (η τρίτη). Καλούνται αυτόματα κατά τον ορισμό ενός αντικειμένου, ανάλογα με τα ορίσματα που θα δώσουμε. Οι δύο πρώτες χαρακτηρίζονται ως (κάποιοι από τους) *constructors* ("κατασκευαστές") της κλάσης ενώ η τρίτη είναι ο μοναδικός επιτρεπτός *copy constructor* ("κατασκευαστής με αντιγραφή") .

Κατασκευαστής (constructor)

Ας επικεντρωθούμε στην πρώτη· προσέξτε τον ιδιαίτερο τρόπο ορισμού της:

```
Student(char const * onoma, int am, int y)
: name(onoma), AM(am), Year(y) {}
```

Το όνομά της είναι υποχρεωτικά το ίδιο με το όνομα της κλάσης, *δεν επιστρέφει τίποτε* και έχει μια ιδιότυπη απόδοση αρχικών τιμών στα μέλη που αποτελούν την υλοποίηση του αντικειμένου. Τυχαίνει να μην υπάρχουν εντολές στο σώμα της συνάρτησης.

Όλοι οι constructors πρέπει να αποδίδουν τιμές στα μέλη που αποτελούν την αναπαράσταση της κλάσης. Ο συγκεκριμένος δέχεται τρία ορίσματα που αντιστοιχούν στα τρία "εσωτερικά" μέλη. Καλείται *αυτόματα* όταν δώσουμε μια δήλωση της μορφής

```
Student s("George␣Thomson", 123, 2000);
```

όταν, δηλαδή, θελήσουμε να δημιουργήσουμε ένα Student με όνομα s, παραθέτοντας τρεις ποσότητες με τύπους **char const** *, **int**, **int** (ή κάποιους που μπορούν να μετατραπούν αυτόματα σε αυτούς). Με την κλήση του δημιουργεί ένα αντικείμενο τύπου Student, με όνομα s, και δίνει αρχικές τιμές στα name, AM, year. Ο ορισμός του συγκεκριμένου constructor, δηλαδή, δημιουργεί το αντικείμενο εκτελώντας τις εντολές

```
std::string name(onoma);
int AM(am);
int Year(y);
```

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 αναφέρονται στον constructor.

Θα μπορούσε κανείς να θεωρήσει ότι έπρεπε να γράψουμε

```
Student(char const * onoma, int am, int y) {
 name = onoma;
 AM = am;
 Year = y;
}
```

Τυπικά δεν είναι λάθος· δημιουργείται το αντικείμενο σαν να εκτελούνται οι εντολές

```
std::string name;
name = onoma;

int AM;
AM = am;

int Year;
Year = y;
```

Προσέξτε τη διαφορά: ο εναλλακτικός ορισμός δημιουργεί τις μεταβλητές name, AM, Year, δίνοντας αρχική τιμή στο name το "" ενώ οι ακέραιες ποσότητες είναι απροσδιόριστες. Κατόπιν, αποκτούν τις επιθυμητές τιμές. 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.

Όπως αναφέρθηκε, μπορούμε να έχουμε περισσότερους του ενός constructors, αρκεί να έχουν διαφορετικά ορίσματα (στο πλήθος ή τον τύπο), §4.6. Στο συγκεκριμένο παράδειγμα έχουμε επιπλέον τη συνάρτηση `Student(){}`. Ο συγκεκριμένος constructor δεν δέχεται ορίσματα και έχει κενό σώμα. Στο συγκεκριμένο παράδειγμα, όταν κληθεί για κάποιο αντικείμενο τύπου Student, δίνει στο μέλος name την τιμή "" ενώ οι ακέραιες ποσότητες είναι απροσδιόριστες. Καλείται αυτόματα όποτε έχουμε ορισμό ενός Student χωρίς να προσδιορίζουμε ορίσματα:

```
Student s;   // calls constructor Student()
```

Προσέξτε ότι δεν γράφουμε

```
Student s(); //function declaration, does not call Student()
```

Η παραπάνω γραμμή κώδικα αποτελεί *δήλωση συνάρτησης* με όνομα s που δεν δέχεται ορίσματα και επιστρέφει ένα Student.

Αν σε μια κλάση δεν ορίσουμε ρητά *κανένα* constructor και προσπαθήσουμε να ορίσουμε ένα αντικείμενό της, ο μεταγλωττιστής δημιουργεί αυτόματα έναν constructor που δεν δέχεται ορίσματα και έχει κενό σώμα, δηλαδή για την κλάση X δημιουργείται αυτόματα και καλείται ο `X(){}`.

Κατασκευαστής αντίγραφου (copy constructor)

Η συνάρτηση-μέλος `Student(Student const & other)` ονομάζεται copy constructor και καλείται όποτε έχουμε δημιουργία ενός αντικειμένου της κλάσης Student από άλλο αντικείμενο της ίδιας κλάσης. Μια τέτοια κλήση έχουμε κατά τον ακόλουθο ορισμό (δημιουργία) του t1 ή του ισοδύναμου ορισμού του t2:

```
Student s("George␣Thomson", 123, 2000);

Student t1(s);
Student t2 = s;
```

Επίσης, ο copy constructor μιας κλάσης χρησιμοποιείται όποτε δίνουμε ως όρισμα συνάρτησης ένα αντικείμενο αυτής της κλάσης. Έτσι, αν έχουμε τη συνάρτηση

```
void f(Student t);
```

η κλήση της παρακάτω καλεί αυτόματα τον copy constructor της Student:

```
Student s("George␣Thomson", 123, 2000);

f(s);
```

Όταν δίνουμε ως όρισμα το Student s στην f, δημιουργούμε το Student t (την παράμετρό της) με αρχική τιμή το s· ουσιαστικά εκτελείται το

```
Student t(s);
```

προτού χρησιμοποιηθεί το t στο σώμα της συνάρτησης.

Προσέξτε ότι αν το όρισμα της συνάρτησης είναι αναφορά, §4.2, 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, καθώς δεν έχει οριστεί πώς γίνεται η αντιγραφή του ορίσματος πριν ολοκληρωθεί το σώμα του copy constructor.

Ακόμα μία περίπτωση που καλείται ο copy constructor είναι κατά την επιστροφή "τιμής" από συνάρτηση. Το αντικείμενο που επιστρέφεται, *δημιουργείται με αντιγραφή* από το αντικείμενο που προσδιορίζεται στην εντολή **return**[3].

Αν σε μία κλάση X δεν ορίσουμε ρητά ένα copy constructor, ο μεταγλωττιστής, όταν προσπαθήσουμε να δημιουργήσουμε ένα αντικείμενο από άλλο της

[3]και το οποίο μπορεί να είναι προσωρινό και να έχει προκύψει με μετατροπή από την ποσότητα που προσδιορίζεται στο **return**.

ίδιας κλάσης, ορίζει αυτόματα έναν copy constructor, (`X(X const & other)`), που δημιουργεί ένα αντικείμενο αντιγράφοντας κάθε μέλος από το όρισμα στο νέο αντικείμενο. Αυτή η αντιγραφή είναι συνήθως αποδεκτή· όταν όμως ως μέλος του τύπου περιλαμβάνεται ένας δείκτης, §4.3, ή ποσότητα για την οποία δεν έχει νόημα (ή δεν έχει το αναμενόμενο νόημα) η αντιγραφή (π.χ. `ifstream`, `ofstream`), πρέπει να ορίσει ρητά ο προγραμματιστής της κλάσης τι επιθυμεί να κάνει ο copy constructor.

6.2.3 Καταστροφέας (Destructor)

Η συνάρτηση `~Student(){}` ονομάζεται destructor (καταστροφέας) και καλείται *αυτόματα* όποτε χρειαστεί η καταστροφή αντικειμένου της κλάσης Student. Αυτό συμβαίνει για αντικείμενο που δεν έχει οριστεί ως **static** (§4.9), όποτε η ροή του κώδικα συναντά το τέλος της εμβέλειάς του ή όταν υπάρχει επιστροφή από τη συνάρτηση στην οποία ορίστηκε[4].

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 τον ορίζει αυτόματα με τη μορφή `~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 που είναι δείκτης, και η περιοχή μνήμης ή η ποσότητα στην οποία δείχνει χρειάζεται "ειδική" καταστροφή.

6.2.4 Τελεστής ανάθεσης (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 ανάθεση 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);`.

[4] από **return** ή exception.

Η συνάρτηση-μέλος `Student & operator=(Student const & other);` είναι αυτή που προσδιορίζει τι ακριβώς γίνεται στην ανάθεση. Αν δεν την ορίσουμε ρητά, παράγεται αυτόματα από το μεταγλωττιστή. Ο αυτόματος ορισμός της αντιγράφει τα μέλη του δεξιού μέρους της ανάθεσης, 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 να μεριμνά για να αποφύγει καταστροφή του αντικειμένου κατά την "αυτο-ανάθεση". 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 ανάθεσης, 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.

6.2.5 Λοιποί τελεστές

Ο τελεστής ανάθεσης (=) 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⊗. Το ⊗ μπορεί να είναι ένας από τους τελεστές του Πίνακα 2.4 (και ελάχιστοι ακόμα που δεν αναφέραμε) εκτός από τον τελεστή επιλογής μέλους κλάσης (.) και τον τελεστή για τον υπολογισμό του μεγέθους αντικειμένου (**sizeof**).

6.3 Κλάση template

Μια κλάση μπορεί να παραμετροποιείται με ένα ή περισσότερους *τύπους* ποσοτήτων ή ακέραιες ποσότητες. Έτσι, αν επιθυμούμε να έχουμε στην κλάση X 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 {
  double v[n];
};
```

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) {
 ...
}
```

6.4 Ασκήσεις

1. Συμπληρώστε τον τύπο Book που αναφέρθηκε παραπάνω.

2. 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"

int
main() {
    date a;              // a = January 1, 2000
    date b(1996);        // b = January 1, 1996
    date c(5,2003);      // c = May 1, 2003
    date d(14,5,2005);   // d = May 14, 2005
    date e(d);           // e = d  (May 14, 2005)
    date const f(d+3);   // f = May 17, 2005
    date g(d+30);        // g = Jun 13, 2005
    date h(d-30);        // h = April 14, 2005
    date i(32,5,2000);   // i = June 1, 2000
    date j(12,13,2000);  // j = January 12, 2001

    std::cout << "h␣day␣is␣"   << h.day()
              << "h␣month␣is␣" << h.month()
              << "h␣year␣is␣"  << h.year();

    h.add_year(2);       // h = April 14, 2007
    h.add_month(2);       // h = June 14, 2007
    h.add_day(2);       // h = June 16, 2007

    std::cout << "f␣day␣is␣"   << f.day()
              << "f␣month␣is␣" << f.month()
```

```
                << "f␣year␣is␣"  << f.year();

    std::cout << j.to_string();
    // prints to std::cout : January 12, 2001

    int k = d-c;  // k = 366 + 365 + 13 = 744

    ++a;  // a = January 2, 2000
    --c;  // c = April 30, 2003

    if (c > a)
        std::cout << c.to_string() << "is␣after␣"
                  << a.to_string() << '\n';

    if (c == a)
        std::cout << c.to_string() << "coincides␣with␣"
                  << a.to_string()<<'\n';

    if (c < a)
        std::cout << c.to_string() << "is␣before␣"
                  << a.to_string() << '\n';


    d += 4;  // d = May 18, 2005
    g -= 3;  // g = Jun 10, 2005
    i = g;   // i = Jun 10, 2005
}
```

3. Δημιουργήστε ένα κατάλληλο τύπο οποίος να αναπαριστά το διάνυσμα στις 3 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.

4. Δημιουργήστε κατάλληλα την κλάση που αντιπροσωπεύει τους μιγαδικούς αριθμούς με όλες τις ιδιότητες που έχουν (δηλαδή, υλοποιήστε την κλάση **template**<**typename** T> complex<T> χωρίς να χρησιμοποιήσετε την κλάση std::complex της STL.

Παράρτημα Α΄

Παραδείγματα προς . . . αποφυγή !

Ας παραθέσουμε απλώς, χωρίς σχόλια, λίγα παραδείγματα κώδικα σωστής C που δείχνουν την κακομεταχείριση των κανόνων και των δυνατοτήτων των 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).
- του Michael Savastio το 1995, Κώδικας Α.2. Υπολογίζει *ακριβώς* το παραγοντικό ακεραίων μέχρι το 429539.
- του Ken Huffman το 1996, Κώδικας Α.3. Μετατρέπει ένα κείμενο σε κώδικα Braille και αντίστροφα.

```
#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

```
#include <stdio.h>

#define l111 0xFFFF
#define ll1 for
#define llll1 if
#define l111 unsigned
#define l1111 struct
#define lllll short
#define llll1 long
#define llll putchar
#define lllll(l) l=malloc(sizeof(l111 llll1));l->llll1=1-1;l->llll1=1-1;
#define l1111 *llll1++=l111%10000;l111/=10000;
#define l1111 llll1(!l1->llll1){lllll(l1->llll1);l1->llll1->llll1=l1;}\
llll1=(l1=l1->llll1)->lll;ll=1-1;
#define llll 1000



                                                l111 llll1 {
                                                l111 llll1 *
      llll1,*llll1        ;l111                 llll1 lll [
      llll];};main      (){l111 llll1           *l111,*l11,*
      l1, *llll, *    malloc ( ) ; llll         l1111 llll ;
      l1111 l11,ll  ,l;l111 llll1 *llll,*       llll1; lll(l
      =1-1 ;l< 14; llll1("\t\"8)>l\"9!.)>vl"    [l]^'L'),++l
      );scanf("%d",&l);l1111(lll) l1111(l111    ) (l1=lll)->
      lll[lll->lll[1-1]     =1]=l111;ll1(l11    =1+1;l11<=l;
      ++l11){l1=llll;         lll1 = (llll=(    l111=l11))->
      lll; llll1 =(            l1l=l1)->lll;    ll=(l111=1-1
      );ll1(;llll->             llll1||l111!=   *llll;){llll
      +=l11**llll++             ;l1111 l1111    (++ll>llll){
      l1111 llll=(              llll =llll->    llll1)->lll;
      }}lll(;llll;              ){l1111 ll111   (++ll>=llll)
      { l1111} } *              llll1=l111;}
      lll(l=(ll=1-              l);(l<llll)&&
      (l1->lll[ l]              !=l111);++l);   lll (;l1;l1=
      l1->l1111,l=              llll){lll(--l   ;l>=1-1;--l,
      ++ll)printf(              (ll)?((ll%19)   ?"%04d":(ll=
      19,"\n%04d")              ):"%4d",l1->    lll[l] ) ; }
                                                l1111(10); }
```

Κώδικας Α.2: savastio.c

```
#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 A.3: huffman.c

Παράρτημα Β΄

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).

Β΄.1 Κώδικας σε C

Στην περίπτωση που επιθυμούμε να χρησιμοποιήσουμε κώδικα σε C επιδιώκουμε, καταρχήν, 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 4.

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 (ήδη μεταγλωττισμένος κώδικας) ακολουθούμε την εξής

[1] για παράδειγμα, ο Fortran compiler της NAG όταν η ρουτίνα προς μεταγλώττιση έχει ποσότητα τύπου CHARACTER ως όρισμα.

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 ρουτίνες ακολουθούν το πρότυπο διασύνδεσης της C συμπληρώνοντας τις δηλώσεις τους με το **extern** "C".

Παράδειγμα:

Έστω ότι δύο ρουτίνες 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 linking πρέπει να προσδιορίσουμε 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.cc και οι "ξένες" ρουτίνες είναι στο fg.c, δίνουμε τις εντολές

```
gcc -c fg.c
g++ prog.cc 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.

Β΄.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 ρουτινών σε Fortran, ελεύθερα διαθέσιμες, είναι η BLAS και η LAPACK. Η πρώτη παρέχει ταχύτατες ρουτίνες για στοιχειώδη χειρισμό πινάκων, μονοδιάστατων ή διδιάστατων (πολλαπλασιασμός σταθεράς με πίνακα, πρόσθεση πινάκων, κλπ.). Η δεύτερη χρησιμοποιεί αυτές για να υλοποιήσει αλγόριθμους επίλυσης γραμμικών συστημάτων, εύρεσης ιδιοτιμών και ιδιοδιανυσμάτων κλπ. Είναι ιδιαίτερα χρήσιμες σε υπολογιστικούς κώδικες. Μια από τις βοηθητικές ρουτίνες είναι η `dlasrt()` η οποία ταξινομεί μονοδιάστατο πίνακα πραγματικών αριθμών διπλής ακρίβειας. Η περιγραφή της από την τεκμηρίωση της LAPACK[2] είναι η εξής

```
SUBROUTINE DLASRT( ID, N, D, INFO )

    CHARACTER          ID
    INTEGER            INFO, N
    DOUBLE PRECISION D( * )
```

Η επεξήγηση των ορισμάτων 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.

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):

[2] σε συστήματα UNIX η εντολή man dlasrt παρουσιάζει την περιγραφή της.

Fortran	C++
INTEGER	**int**
REAL	**float**
DOUBLE PRECISION	**double**
LOGICAL	**bool**
CHARACTER	**char**
COMPLEX	`std::complex<`**float**`>`

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 *μονοδιάστατος* πίνακας σε column-major order (η αποθήκευση γίνεται κατά στήλες). Συμβουλευτείτε τη §2.6.1 για το πώς γίνεται ο ορισμός και η πρόσβαση σε τέτοιο πίνακα. Στην περίπτωση που χρησιμοποιήσουμε `std::vector<>` για την υλοποίησή του θα πρέπει να "περάσουμε" στη συνάρτηση *τη διεύθυνση του πρώτου στοιχείου*, δηλαδή, αν *υ* είναι ένα `std::vector` το όρισμα θα υποκατασταθεί με το `&v[0]`.

Όπως βλέπουμε στην περιγραφή της συνάρτησης τα δύο πρώτα ορίσματα, id, 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 μια ρουτίνα της Fortran μπορούν να τροποποιηθούν από αυτή. Αντίστοιχη συμπεριφορά επιτυγχάνεται στη C++ δηλώνοντας τα ως αναφορές (§4.2) (εκτός από τα ορίσματα πινάκων ή άλλων δεικτών, για τα οποία δε χρειάζεται). 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 GNU compilers 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. Βάσει των παραπάνω η δήλωση για αυτούς τους compilers γίνεται

```
extern "C"
void dlasrt_(char const & id, int const & n, double d[],
             int & info);
```

Με την παραπάνω δήλωση, η μεταγλώττιση έχει όλη την πληροφορία που χρειάζεται για να προχωρήσει. Η σύνδεση των ρουτινών (που για τις LAPACK και BLAS έρχονται σε library) γίνεται (σε συστήματα UNIX) με την ακόλουθη εντολή

```
g++ prog.cc -llapack -lblas
```

Χωρίς να μπούμε σε λεπτομέρειες, αναφέρουμε ότι η Fortran 2003 προσδιορίζει κανόνες για τη διασύνδεση ρουτινών της σε κώδικα C (συνεπώς και C++).

Σε κάθε περίπτωση πρέπει να συμβουλευόμαστε την τεκμηρίωση που συνοδεύει τους compilers που θα χρησιμοποιήσουμε.

Παράρτημα Γ΄

Λύσεις επιλεγμένων ασκήσεων

Παρακάτω παρουσιάζονται συνοπτικά ενδεικτικές λύσεις κάποιων από τις ασκήσεις.

Κεφάλαιο 3

3.1. Παραγοντικό (n!) ενός ακεραίου.

```
#include <iostream>

// factorial
int
main()
{
    int n;
    std::cout << "Give␣non␣negative␣integer␣number␣";
    std::cin  >> n;

    double fac = 1.0;
    for (int i=1; i<=n; ++i)
        fac *= i;

    std::cout << "The␣factorial␣of␣" << n
              << "␣is␣" << fac << '\n';
}
```

3.2. *N* πρώτοι όροι της ακολουθίας Fibonacci.

```
#include <iostream>

// Fibonacci
int
main()
{
    int n;
    std::cout << "How␣many␣Fibonacci␣numbers?␣";
    std::cin  >> n;

    int f0 = 0;
    int f1 = 1;

    if (n > 0)
```

```
        std::cout << f0 << '\n';
    if (n > 1)
        std::cout << f1 << '\n';

    for (int i = 2; i < n; ++i) {
        int const f2 = f0 + f1;
        std::cout << f2 << '\n';

        f0 = f1;
        f1 = f2;
    }
}
```

3.3. Μέγιστος κοινός διαιρέτης δύο ακεραίων αριθμών.

```
#include <iostream>

// greatest common divisor, Euclid's algorithm: gcd(N,M) = gcd(M, N mod M)
int
main()
{
    std::cout << "Give␣two␣positive␣integers:␣";
    int m,n;
    std::cin  >> m >> n;

    while (n != 0) {
        const int temp = m;
        m = n;
        n = temp % n;
    }

    std::cout << "The␣greatest␣common␣divisor␣is␣" << m << '\n';
}
```

3.4. Θεώρημα των τεσσάρων τετραγώνων του Lagrange.

```
#include <iostream>

int
main()
{
    std::cout << "Give␣a␣positive␣integer␣number:␣";

    int N;
    std::cin >> N;

    for (int a = 0; a*a <= N; ++a) {
        for (int b = a; a*a + b*b <= N; ++b) {
            for (int c = b; a*a + b*b + c*c <= N; ++c) {
                for (int d = c; a*a + b*b + c*c + d*d <= N; ++d)
                    if (a*a + b*b + c*c + d*d == N)
                        std::cout << N  << "␣is␣"
                                  << a <<'*' << a << "␣+␣"
                                  << b <<'*' << b << "␣+␣"
                                  << c <<'*' << c << "␣+␣"
                                  << d <<'*' << d << '\n';
            }
        }
    }
}
```

5. Γράψτε κώδικα που να τυπώνει 8 τυχαίους αριθμούς.

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

int
main() {
    std::cout <<"RAND_MAX␣is␣" << RAND_MAX << '\n';

    std::srand(12345U);

    for (int i = 0; i < 8; ++i)
        std::cout << std::rand() << '\n';
}
```

6.1. Δημιουργήστε ένα αρχείο με 1000 τυχαίους ακεραίους στο διάστημα $[-20 : 20]$.

```
#include <fstream>
#include <cstdlib>

int
main()
{
    std::ofstream random("rndnumbers");

    for (int i = 0; i < 1000; ++i)
        random << static_cast<int>(-20.0
                                   + 41.0 * std::rand() / (RAND_MAX+1.0))
               << '\n';
}
```

6.2. Γράψτε ένα πρόγραμμα C++ που να διαβάζει το αρχείο και να τυπώνει στην οθόνη πόσους θετικούς, αρνητικούς και ίσους με το 0 αριθμούς περιέχει.

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

int
main()
{
    std::ifstream random("rndnumbers");

    int r;
    int neg = 0;
    int zer = 0;
    int pos = 0;

    while (random >> r) { // when read fails, random is false
        if (r  < 0) ++neg;
        if (r == 0) ++zer;
        if (r  > 0) ++pos;
    }
    std::cout << "There␣are␣" << neg << "␣negative␣numbers.\n";
    std::cout << "There␣are␣" << zer << "␣zero␣numbers.\n";
    std::cout << "There␣are␣" << pos << "␣positive␣numbers.\n";
}
```

10. Γράψτε κώδικα που να τυπώνει τις ρίζες του πολυωνύμου $ax^2 + bx + c$.

```
#include <iostream>
#include <complex>
#include <cmath>
#include <iomanip>

// roots of quadratic a x^2 + b x + c
int
main()
{
    std::cout << "Give␣coefficients␣a,b,c:␣";

    double a,b,c;
    std::cin >> a >> b >> c;

    if (a == 0.0) {
        std::cerr << "a␣must␣be␣non-zero\n";
        return -1;
    }

    typedef std::complex<double> complex;

    complex const sqrtD = std::sqrt(complex(b*b - 4.0 * a * c));

    complex x1 = (-b + sqrtD) / (2.0 * a);
    complex x2 = (-b - sqrtD) / (2.0 * a);

    double const large = 1e6; // a number much larger than 1

    if ( (b*b > large * 4.0 * a * c) && (c != 0.0) )  // b^2 >> 4ac
        if (b > 0.0)  // then x1 is not accurate
            x1 = 2.0 * c / (-b - sqrtD);
        else        // x2 is not accurate
            x2 = 2.0 * c / (-b + sqrtD);

    std::cout << "The␣roots␣are␣"
              << std::setprecision(12) << x1 << '␣' << x2 << '\n';

    std::cout << "The␣residues␣are␣"
              << a*x1*x1 + b*x1 + c << '␣'
              << a*x2*x2 + b*x2 + c << '\n';
}
```

13. Πολλαπλασιασμός πινάκων με γνωστές διαστάσεις.

```
#include <fstream>

int
main() {
    int const M = 10, N = 20, P = 30;

    double A[M][N], B[M][P], C[M][P];

    std::ifstream matA("matA.dat");
    //read A
    for (int i=0; i < M; ++i) {
        for (int j=0; j < N; ++j) {
            matA >> A[i][j];
        }
    }
```

```
    std::ifstream matB("matB.dat");
    //read B
    for (int i=0; i < N; ++i) {
        for (int j=0; j < P; ++j) {
            matB >> B[i][j];
        }
    }

    // compute C
    for (int i=0; i < M; ++i) {
        for (int j=0; j < P; ++j) {
            C[i][j] = 0.0;
            for (int k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    std::ofstream matC("matC.dat");
    // print C
    for (int i=0; i < M; ++i) {
        for (int j=0; j < P; ++j) {
            matC << C[i][j] << '␣';
        }
    }
}
```

18. Γράψτε ένα πρόγραμμα C++ που να υλοποιεί το Game of Life.

```
#include <fstream>
#include <sstream>
#include <iomanip>


int
main() {
    int const M = 512;
    int const N = 512;

    int const maxgen = 1000; // how many generations
    int const digits = 3; // 1+ log10(maxgen-1);

    int gen[M][N];

    // initial generation; two pairs of diagonal lines
    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < N; ++j) {
            if ( (i==j+1) || (i==j-1) || (i==N-1-j-1) || (i==N-1-j+1) ) {
                gen[i][j] = 1;
            } else {
                gen[i][j] = 0;
            }
        }
    }

    for (int generation = 0; generation < maxgen; ++generation) {
        // save generation
        std::ostringstream filename;
        filename << "life_"
                 << std::setw(digits) << std::setfill('0')
                 << generation
                 << ".ppbm";
```

```
            std::ofstream out(filename.str().c_str());

            out << "P1\n" << N << ' ' << M << '\n'; // P1, width, height
            for (int i = 0; i < M; ++i) {
                for (int j = 0; j < N; ++j) {
                    out << gen[i][j] << '\n';
                }
            }

            // In the last generation don't calculate the new one.
            if (generation == maxgen-1) {
                break;
            }

            // new generation
            int newgen[M][N];

            for (int i = 0; i < M; ++i) {
                for (int j = 0; j < N; ++j) {

                    // find number of neighbours
                    int neigh = 0;
                    for (int i1 = i-1; i1 <= i+1; ++i1) {
                        for (int j1 = j-1; j1 <= j+1; ++j1) {
                            if ( (i1 == i) && (j1 == j) ) {
                                continue;
                            }

                            if ((0 <= i1) && (i1 < M) && (0 <= j1) && (j1 < N)) {
                                neigh += gen[i1][j1];
                            }
                        }
                    }

                    // calculate new generation
                    if ((neigh == 3) || ((gen[i][j] == 1) && (neigh == 2))) {
                        newgen[i][j] = 1;
                    } else {
                        newgen[i][j] = 0;
                    }
                }
            }

            // copy new generation to old
            for (int i = 0; i < M; ++i) {
                for (int j = 0; j < N; ++j) {
                    gen[i][j] = newgen[i][j];
                }
            }
        }
    }
```

Κεφάλαιο 4

5. Υπολογίστε την ορίζουσα τετραγωνικού πίνακα A.

```
#include <vector>
```

```
void
copy(std::vector<double> const & A, int N, std::vector<double> & Aij,
     int row, int col);

double
determinant(std::vector<double> const & A, int N) {
    if (N==1) {
        return A[0];
    }

    std::vector<double> Aij((N-1)*(N-1));

    int const col = 0;
    int sign = 1;   // (-1)^(0+col)

    double sum = 0.0;
    for (int i = 0; i < N; ++i) {
        copy(A, N, Aij, i, col);

        sum += sign * A[i+col*N] * determinant(Aij, N-1);

        sign = -sign;
    }

    return sum;
}

void
copy(std::vector<double> const & A, int N, std::vector<double> & Aij,
     int row, int col) {

    int in = 0;
    for (int i = 0; i < N; ++i) {
        if (i == row) {
            continue;
        }

        int jn = 0;
        for (int j = 0; j < N; ++j) {
            if (j == col) {
                continue;
            }

            Aij[in+jn*(N-1)] = A[i+j*N];
            ++jn;
        }
        ++in;
    }
}
```

Αναζήτηση. Αλγόριθμος δυαδικής αναζήτησης.

```
// recursive binary search;  a[] sorted from low values to high values.
template<typename T>
bool
bsearch(T const a[], int beg, int end, T v) {
    if (end-beg == 1) {
        return a[beg] == v;
    }

    int const m = beg + (end-beg)/2;
```

```
    return v < a[m] ? bsearch(a, beg, m, v): bsearch(a, m, end, v);
}



template<typename T>
bool
bsearch_nonrecursive(T const a[], int beg, int end, T v) {
    while (end-beg > 1) {
        int const m = beg + (end-beg)/2;

        if (v < a[m]) {
            end = m;
            } else {
            beg = m;
        }
     }

     return a[beg] == v;
}
```

Ταξινόμηση. Αλγόριθμοι ταξινόμησης.

```
template<typename T>
void
bubbleSort(T a[], int n) {
 for (int k = n-1; k > 0; --k)
        for (int j = 0; j < k; ++j)
            if (a[j] > a[j+1]) {
                T const c = a[j];
                a[j] = a[j+1];
                a[j+1] = c;
            }
}


template<typename T>
void
insertionSort(T a[], int n) {
    for (int k = 1; k < n; ++k) {
        int j = 0;
        while (a[k] > a[j])
            ++j;

        T const temp = a[k];

        for (int i = k; i > j; --i)
            a[i] = a[i-1];

        a[j] = temp;
    }
}

//  Quicksort with auxilliary matrix.

#include <vector>
template <typename T>
void
quickSort(T a[], int n) {
    if (n <= 1)
        return;
    /* if the matrix has one element or does not exist,
```

```
       there is nothing to do. */

    // make b large enough for (n-1) elements above or below the initial.
    std::vector<T> b((n-1) + 1 + (n-1));

    int const middle = n-1;

    int const k = 0; // choose any index in [0, n).

    // copy chosen element to middle position of b.
    b[middle] = a[k];

    // separate low elements of a to upper half of b
    // and high elements of a to lower half of b.
    int jlow = middle;
    int jhig = middle;
    for (int i = 0; i < n; ++i) {
        if (i == k) // skip i if on the chosen element. It has been copied.
            continue;

        if (a[i] <= a[k]) {
            --jlow;
            b[jlow] = a[i];
        } else {
            ++jhig;
            b[jhig] = a[i];
        }
    }

    // b[jlow] .. b[jhig] is the new matrix. Copy it to a:
    for (int i = 0; i < n; ++i)
        a[i] = b[jlow+i];

    // sort "low" matrix

    // between b[jlow] and b[middle-1] there are (middle-1) - jlow + 1 elements
    // The "low" matrix starts at a[0]
    quickSort(a, middle - jlow);

    // sort "high" matrix

    // between b[middle+1] and b[jhig] there are jhig - (middle+1) + 1 elements
    // The "high" matrix starts at a[middle-jlow+1]
    quickSort(a+middle-jlow+1, jhig-middle);
}

template<typename T>
void
mergesort(T a[], int beg, int end) {

    // if there is only one element (or less), the array is already sorted.
    if (end-beg <= 1) {
        return;
    }

    //get middle point
    int const m = beg + (end-beg)/2;

    // sort the two halves
    mergesort(a, beg, m);
    mergesort(a, m, end);
```

```
    std::vector<T> c(end-beg);

    int i1 = beg;
    int i2 = m;

    for (int i = 0; i < end-beg; ++i) {
        bool const cond1 = i1 == m;
        bool const cond2 = i2 == end;

        int j;

        if (cond1) {
            j = i2++;
        }

        if (cond2) {
            j = i1++;
        }

        if (!cond1 && !cond2) {
            j = (a[i1] < a[i2] ? i1++ : i2++);
        }

        c[i] = a[j];
    }

    for (int i = 0; i < end-beg; ++i) {
        a[i+beg] = c[i];
    }
}
```

Κεφάλαιο 5

3. Υλοποιήστε αλγορίθμους της STL.

```
template<typename iterator, typename T>
T
accumulate(iterator beg, iterator end, T value) {
    T s = value;
    for (iterator it = beg; it != end; ++it) {
        s += *it;
    }

    return s;
}

template<typename iterator, typename T>
T
inner_product(iterator beg1, iterator end1,
              iterator beg2, T value) {
    T s = value;

    iterator it1 = beg1;
    iterator it2 = beg2;

    while (it1 != end1) {
        s += (*it1) * (*it2);
```

```
        ++it1;
        ++it2;
    }

    return s;
}


template<typename iterator, typename functor>
void
generate(iterator beg, iterator end, functor op) {
    for (iterator it=beg; it != end; ++it) {
        *it = op();
    }


}



template<typename iterator, typename T>
iterator
remove(iterator beg, iterator end, T const & x) {
    while (beg!=end) {
        if (*beg != x) {
            ++beg;
            continue;
        }

        for (iterator r=beg; r+1!=end; ++r) {
            iterator q=r+1;

            T const temp = *r;
            *r = *q;
            *q = temp;
        }
        --end;
    }
    return beg;
}


template<typename iterator, typename T>
bool
binary_search(iterator beg, iterator end, T const & v) {
    int n = std::distance(beg,end);

    if (n == 1) {
        return *beg == v;
    }

    iterator it=beg;
    std::advance(it, n/2);

    if (v < *it) {
        return binary_search(beg, it, v);
    }
    else {
        return binary_search(it, end, v);
    }
```

```
}

template<typename iteratorA, typename iteratorB, typename iteratorC>
iteratorC
merge(iteratorA Abeg, iteratorA Aend, iteratorB Bbeg, iteratorB Bend,
      iteratorC Cbeg){
    while ((Bbeg != Bend) || (Abeg != Aend)) {
        if (Bbeg == Bend) {
            *Cbeg++ = *Abeg++;
            continue;
        }

        if (Abeg == Aend) {
            *Cbeg++ = *Bbeg++;
            continue;
        }

        *Cbeg++ = (*Abeg < *Bbeg ? *Abeg++; *Bbeg++);
    }

    return Cbeg;
}
```

Βιβλιογραφία

[1] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison Wesley, Reading, MA, USA, first edition, September 1999.

[2] Herb Sutter. *More Exceptional C++: 40 More Engineering Puzzles, Programming Problems, and Solutions*. C++ in Depth Series. Addison Wesley, Reading, MA, USA, January 2002.

[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.

[4] Andrei Alexandrescu. *Modern C++ Design: Applied Generic and Design Patterns*. C++ In-Depth Series. Addison Wesley, Reading, MA, USA, January 2001.

[5] Matthew H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Professional Computing Series. Addison Wesley, Reading, MA, USA, January 1999.

[6] John J. Barton and Lee R. Nackman. *Engineering and Scientific C++: An Introduction with Advanced Techniques and Examples*. Addison Wesley, Reading, MA, USA, 1994.

[7] Dov Bulka and David Mayhew. *Efficient C++ Performance Programming Techniques*. Addison Wesley, Reading, MA, USA, November 1999.

[8] Bruce Eckel. *Thinking in C++. Introduction to Standard C++*, volume 1. Prentice Hall, second edition, 2000. Also available as electronic book.

[9] Bruce Eckel and Chuck Allison. *Thinking in C++. Practical Programming*, volume 2. Prentice Hall, second edition, 2003. Also available as electronic book.

[10] Francis Glassborow. *You Can Program in C++: A Programmer's Introduction*. John Wiley & Sons, 2006.

[11] John R. Hubbard. *Programming with C++*. Schaum's Outline Series. McGraw–Hill, second edition, June 2000.

[12] John R. Hubbard. *Fundamentals of Computing with C++*. Schaum's Outline Series. McGraw–Hill, May 1998.

[13] Nicolai M. Josuttis. *Object-Oriented Programming in C++*. John Wiley and Sons Ltd, November 2002.

[14] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison Wesley, Reading, MA, USA, second edition, March 2012.

[15] Andrew Koenig and Barbara E. Moo. *Accelerated C++: practical programming by example*. C++ In-Depth Series. Addison Wesley, Reading, MA, USA, 2000.

[16] Stanley B. Lippman. *Essential C++*. C++ In-Depth Series. Addison Wesley, Reading, MA, USA, 2000.

[17] Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo. *C++ Primer*. Addison Wesley, Reading, MA, USA, fifth edition, August 2005.

[18] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Professional Computing Series. Addison Wesley, Reading, MA, USA, third edition, June 2005.

[19] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Professional Computing Series. Addison Wesley, Reading, MA, USA, March 1996.

[20] Trevor Misfeldt, Gregory Bumgardner, and Andrew Gray. *The Elements of C++ Style*. SIGS Reference Library. Cambridge University Press, 2002.

[21] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, Reading, MA, USA, fourth edition, 2013.

[22] Bjarne Stroustrup. *A tour of C++*. Addison Wesley, Reading, MA, USA, 2013.

[23] Bjarne Stroustrup. *Programming - Principles and Practice Using C++*. Addison Wesley, Reading, MA, USA, December 2008.

[24] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, Reading, MA, USA, third edition, 1997.

[25] Bjarne Stroustrup. Learning Standard C++ as a New Language. *The C/C++ Users Journal*, May 1999. Also available in CVU, Vol. 12, No. 1, January 2000.

[26] Herb Sutter. *Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems and Solutions*. C++ In-Depth Series. Addison Wesley, Reading, MA, USA, August 2004.

[27] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: Rules, Guidelines, and Best Practices*. C++ In-Depth Series. Addison Wesley, Reading, MA, USA, November 2004.

[28] Herb Sutter. *Exceptional C++*. C++ In-Depth Series. Addison Wesley, Reading, MA, USA, October 1999.

[29] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison Wesley, Reading, MA, USA, December 2002.

[30] David Vandevoorde. *C++ Solutions: Companion to the C++ Programming Language*. Addison Wesley, Reading, MA, USA, August 1998.

[31] D. Yang. *C++ and Object-Oriented Numeric Computing for Scientists and Engineers*. Springer-Verlag, New York, January 2001.

Κατάλογος Πινάκων

Ευρετήριο