Εισαγωγή στη C++

©2004 Κωνσταντίνος Μαργαρίτης, markos@debian.org

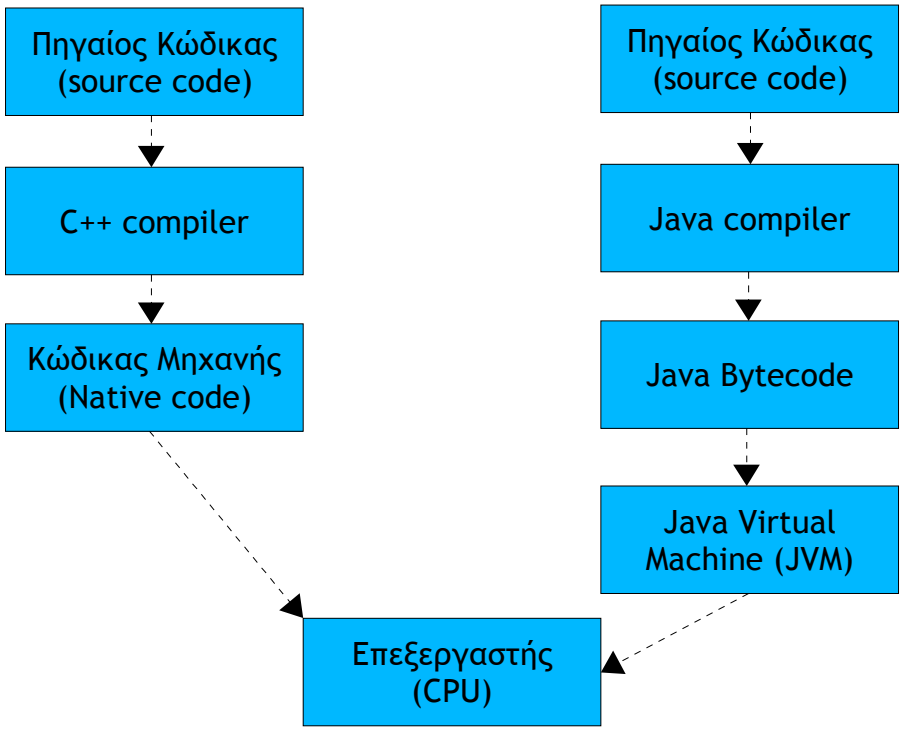
Γιάννης Τσιομπίκας, nuclear@siggraph.org

1.Σχετικά με τη C++

Μια συνηθισμένη γλώσσα προγραμματισμού (π.χ. C, C++, PASCAL) μεταγλωττίζει τον πηγαίο κώδικα του προγράμματος σε εκτελέσιμη μορφή που να καταλαβαίνει ο επεξεργαστής. Η μορφή αυτή είναι η γλώσσα μηχανής, και διαφέρει για κάθε επεξεργαστή, αρχιτεκτονική και λειτουργικό σύστημα. Η εκτελέσιμη μορφή ενός προγράμματος για μια συγκεκριμένη αρχιτεκτονική, δεν είναι δυνατό να χρησιμοποιηθεί σε διαφορετική αρχιτεκτονική, δηλαδή δε μπορούμε να χρησιμοποιήσουμε το εκτελέσιμο πρόγραμμα για Windows σε έναν υπολογιστή Macintosh. Έτσι, είναι απαραίτητη η εκ νέου μεταγλώττιση του πηγαίου κώδικα (source code) για κάθε αρχιτεκτονική στην οποία θέλουμε να τρέξουμε το πρόγραμμα.

Μεταγλώττιση (compile) & Σύνδεση (link)

Στο παρακάτω σχήμα απεικονίζεται η διαδικασία της μεταγλώττισης ενός προγράμματος C++ και ενός προγράμματος Java.



Διαδικαστικός ή Δομημένος Προγραμματισμός (Procedural ή Structured Programming)

Οι παλαιότερες γλώσσες προγραμματισμού όπως οι C, PASCAL, FORTRAN έδιναν έμφαση

στην διαδικασία και στα στάδια που ακολουθούνται για την επίτευξη κάποιου στόχου. Το αντικείμενο ήταν ο κώδικας (code-centric γλώσσες προγραμματισμού). Ο προγραμματισμός γινόταν ορίζοντας τη ροή εκτέλεσης (από το στάδιο Α, στο στάδιο Β) και τις αντίστοιχες υπορουτίνες.

Αντικειμενοστραφής προγραμματισμός (Object-Oriented Programming)

Οι αντικειμενοστραφείς γλώσσες προγραμματισμού (Java, Eiffel, Smalltalk και φυσικά C++) δίνουν έμφαση στα δεδομένα παρά στον κώδικα. Το πρόγραμμα αναπτύσσεται γύρω από τα δεδομένα (data-centric) τα οποία ορίζουν από μόνα τους τον τρόπο με τον οποίο μπορουμε να τα διαχειριστούμε.

Ο φυσικός και ο τεχνητός κόσμος που ζούμε είναι πιο κοντά στη φιλοσοφία του Αντικειμενοστραφή προγραμματισμού παρά του Δομημένου προγραμματισμού.

Ένα απλό παράδειγμα που μπορούμε να χρησιμοποιήσουμε για την κατανόηση της φιλοσοφίας του αντικειμενοστραφούς προγραμματισμού είναι το αυτοκίνητο.

Κάθε αυτοκίνητο είναι ένα αντικείμενο που ανήκει σε μια κλάση που ορίζει τα βασικά χαρακτηριστικά του αυτοκινήτου. Αυτά μπορεί να διαφέρουν ανάμεσα στους κατασκευαστές αλλά όλα θα παρέχουν τα βασικά χαρακτηριστικά που ορίζει η κλάση “αυτοκίνητο” για τη χρήση του. Δηλαδή τιμόνι, γκάζι, φρένο, συμπλέκτης και ταχύτητες. Αυτά είναι τα δεδομένα. Κάθε ένα από αυτά ορίζει τον τρόπο χρήσης του. Το τιμόνι στρίβει αριστερά/δεξιά, τα πεντάλ πιέζονται ή αφήνονται και οι ταχύτητες αλλάζουν διακριτά έχοντας μηχανισμό ασφαλείας – δε μπορούμε να αλλάξουμε ταχύτητα σε όπισθεν ενώ το αυτοκίνητο κινείται με ταχύτητα.

Η υλοποίηση καθενός από αυτούς τους μηχανισμούς διαφέρει σε κάθε κατασκευαστή, αλλά η χρήση τους είναι η ίδια για όλους. Δηλαδή η χρήση του τιμονιού και των ταχυτήτων γίνεται με τον ίδιο τρόπο ανεξαρτήτως κατηγορίας, κατασκευαστή και μοντέλου του αυτοκινήτου. Επίσης, ο μηχανισμός με τον οποίο γίνεται η χρήση των δεδομένων αυτών είναι κρυμμένος από τον χρήστη (δηλαδή τον οδηγό). Ο χρήστης δεν ενδιαφέρεται για τον τρόπο μετάδοσης της κίνησης από το τιμόνι στους τροχούς ώστε το αυτοκίνητο να στρίψει. Επίσης, η χρήση ενός αυτοκινήτου δεν αλλάζει με την επέκτασή του ή αλλαγή ορισμένων χαρακτηριστικών του (όπως π.χ. νέα μηχανή, λάστιχα, κλπ). Αλλάζει η συμπεριφορά του αλλά όχι η χρήση του.

Με αυτόν τον τρόπο, περιγράψαμε με ένα απλό παράδειγμα τα πιο σημαντικά χαρακτηριστικά του αντικειμενοστραφούς προγραμματισμού:

- Encapsulation ()

  Η διαδικασίες κρύβονται από το χρήστη και τα ίδια τα δεδομένα προσδιορίζουν τους τρόπους διαχείρισης τους.

- Polymorphism (πολυμορφισμός)

  Αντικείμενα που ανήκουν σε παρόμοιες κλάσεις μπορούν να έχουν κοινό τρόπο προσπέλασης, με αποτέλεσμα ο χρήστης να μπορεί να τα χειριστεί με τον ίδιο τρόπο χωρίς να χρειάζεται να μάθει νέες διαδικασίες.

- Inheritance (κληρονομικότητα)

  Μπορούμε να δημιουργήσουμε ένα νέο αντικείμενο παίρνοντας ως βάση ένα άλλο ήδη υπάρχον. Το νέο αντικείμενο θα έχει τα χαρακτηριστικά του παλιού ενώ θα μπορέι να τα τροποποιήσει, να τα επεκτείνει και να προσθέσει καινούρια για να καλύψει συγκεκριμένες ανάγκες.

2.Δομή της C++

Όπως και κάθε γλώσσα προγραμματισμού, έτσι και η C++ ακολουθεί κάποιους κανόνες στη σύνταξη και υλοποίηση ενός προγράμματος. Οι κανόνες αυτοί αφορούν τη δήλωση μεταβλητών, τους τύπους δεδομένων, τους ορισμούς κλάσεων κλπ.

Μεταβλητές

Η έννοια της μεταβλητής είναι ίσως η πιο θεμελιώδης στον προγραμματισμό γενικά. Η μεταβλητή δεν είναι άλλο από μια ονομασμένη θέση μνήμης που μπορεί να περιέχει δεδομένα οποιουδήποτε είδους. Το περιεχόμενο της μεταβλητής μπορεί να μεταβάλλεται -εξ' ού και το όνομά της- με δεδομένα παρόμοιου είδους.

Οι μεταβλητές ορίζονται ως εξής:

```
type var;
```

όπου `type` το είδος της μεταβλητής και `var` το όνομά της. Ως είδος μπορούμε να διαλέξουμε έναν από τους διαθέσιμους τύπους δεδομένων της C++ ή το όνομα μιας κλάσης αντικειμένων (σε επόμενη παράγραφο).

Έυρος μεταβλητών

Μια μεταβλητή θεωρείται έγκυρη, δηλαδή μια αναφορά σε αυτήν έχει νόημα μόνο μέσα στο πλάισιο στο οποίο ορίζεται. Το πλαίσιο μπορεί να είναι μια κλάση, μια συνάρτηση ή μέθοδος, το πεδίο ισχύος ενός loop `for/while/do` ή των εντολών `if/else/switch`.

Παράδειγμα:

```
type var1;
if (συνθήκη) {
      // Επεξεργασία της var1 (σωστό).
      // Επεξεργασία της var2 (λάθος).
      type var2;
      // Επεξεργασία της var2 (σωστό)
}
// Επεξεργασία της var1 (σωστό)
// Επεξεργασία της var2 (λάθος)
```

Τύποι δεδομένων

Οι βασικοί τύποι δεδομένων στη C++ είναι σχεδόν οι ίδιοι με αυτούς της C.

Ακολουθεί πίνακας με τους διαθέσιμους τύπους δεδομένων της C++ και το μέγεθός τους σε bytes, για ένα συνηθισμένο PC (32-bit αρχιτεκτονική):

Όνομα	*Μέγεθος (σε bytes)*	*Όρια*
`char`	1	-128 έως 127
`short`	2	-32,768 έως 32,767
`int/long`	4	-2,147,483,648 έως -2,147,483,647
`long long`	8	-9,223,372,036,854,775,808 έως -9,223,372,036,854,775,807
`float`	4	$1.4 * 10^{-45}$ έως $3.4 * 10^{38}$

Όνομα	*Μέγεθος (σε bytes)*	*Όρια*
`double`	8	$4.9 * 10^{-324}$ έως $1.8 * 10^{308}$
`bool`	1	true / false
`wchar`	2	-
`string`	μεταβλητό	-

Οι τύποι δεδομένων `char`, `short`, `int`, `long`, `long long` προορίζονται για χρήση με ακέραιους αριθμούς, ενώ οι τύποι `float`, `double` για χρήση αριθμών κινητής υποδιαστολής (δηλ. δεκαδικών αριθμών). Ο τύπος `bool` μπορεί να λαμβάνει μόνο δύο τιμές `true` και `false`. Ο τύπος `wchar` έχει μέγεθος 2 bytes γιατί έχει σχεδιαστεί να περιέχει χαρακτήρες Unicode (UTF-16 για την ακρίβεια).

Παραδείγματα δηλώσεων και αρχικοποιήσεων μεταβλητών:

```
int an_integer;
an_integer = 10;
long a_long = an_integer *1000;
double verysmallnumber = 0.000000000003;
bool am_i_hungry = false;
char alpha = 'a';
string text = "this is a text";
```

unsigned μεταβλητές

Στη C++ όπως και στη C, αλλά αντίθετα με τη Java, μπορούμε να διαμορφώσουμε το εύρος των μεταβλητών `char`, `short`, `int`, `long`, `long long` ώστε να περιέχουν μόνο θετικούς αριθμούς. Στην πραγματικότητα ορίζουμε το bit του προσήμου (sign) να χρησιμοποιείται και αυτό για αρίθμηση, διπλασιάζοντας έτσι το εύρος των θετικών αριθμών του κάθε τύπου. Οι τύποι αυτοί διαχωρίζονται με τους απλούς με τη λέξη `unsigned` πρίν από το όνομα του τύπου. Το εύρος αυτών των τύπων φαίνεται στον παρακάτω πίνακα:

Όνομα	*Μέγεθος (σε bytes)*	*Όρια*
`unsigned char`	1	0 έως 255
`unsigned short`	2	0 έως 65,535
`unsigned int/` `unsigned long`	4	0 έως 4,294,967,295
`unsigned long long`	8	0 έως 18,446,744,073,709,551,615

Σταθερές (constants)

Αν και οι μεταβλητές έχουν σχεδιαστεί να μεταβάλλονται, υπάρχουν περιπτώσεις που χρειαζόμαστε κάποιες σταθερές, όπως στα Μαθηματικά. Μπορούμε να δηλώσουμε μια μεταβλητή ως σταθερά (δηλαδή που να μη μεταβάλλεται) με τη λέξη `const`. Δηλαδή,

```
const double pi = 3.1415;
```

Τελεστές (operators)

Μπορούμε να εκτελέσουμε διάφορες πράξεις μεταξύ των μεταβλητών με τη χρήση των τελεστών. Οι τελεστές είναι σύμβολα που αντιστοιχούν σε αριθμητικές ή λογικές πράξεις μεταξύ των αντικειμένων. Υπάρχουν 4 είδη τελεστών στη C++ (θα αναφέρουμε τα τρία πιο σημαντικά, και θα αφήσουμε τους δυαδικούς τελεστές προς το παρόν).

Αριθμητικοί τελεστές

Σύμβολο	*Είδος*
+	Πρόσθεση
-	Αφαίρεση
*	Πολλαπλασιασμός
/	Διαίρεση
%	Υπόλοιπο Διαίρεσης
++	Αυξητικός τελεστής
--	Αφαιρετικός τελεστής

Ειδικά για τους αυξητικούς και αφαιρετικούς τελεστές, πρέπει να αναφέρουμε ότι αυτοί αυξάνουν ή αφαιρούν την τιμή της μεταβλητής κατά μία μονάδα (κατά συνέπεια χρησιμοποιούνται μόνο σε ακέραιες μεταβλητές, όχι σε δεκαδικές). Παράδειγμα:

```
int x = 10;
x++; // τώρα η τιμή x έχει την τιμή 11
x--; // και πάλι την τιμή 10
```

Επιπλέον, οι τελεστές αυτοί μπορούν να εκτελεστούν πρίν ή μετά τον υπολογισμό της παράστασης στην οποία ανήκουν, για παράδειγμα:

```
int x = 10;       // το x έχει την τιμή 10
int y = x++;      // y έχει την τιμή 10, αλλά το x την τιμή 11
x = 10;           // το x έχει πάλι την τιμή 10
int z = ++x;      // το z έχει την τιμή 11, όπως και το x
```

Σχεσιακοί Τελεστές

Οι σχεσιακοί τελεστές αναφέρονται στην σχέση μεταξύ δύο αντικειμένων.

Σύμβολο	*Είδος*
==	Ισότητα
!=	Ανισότητα
>	Μεγαλύτερο από
<	Μικρότερο από
>=	Μεγαλύτερο από ή ίσο με
<=	Μικρότερο από ή ίσο με

Λογικοί Τελεστές

Οι λογικοί τελεστές αναφέρονται στους τρόπους συνδυασμών αληθών και ψευδών προτάσεων.

Σύμβολο	*Είδος*
&&	Short-circuit AND
||	Short-circuit OR
!	NOT (άρνηση)

Αν με τον υπολογισμό της πρώτης παράστασης ο συνδυασμός βγαίνει αληθής ή ψευδής τότε αποφεύγεται ο υπολογισμός της δεύτερης.

Για παράδειγμα, αν έχουμε την παράσταση

```
(alpha == true) && (beta == 1)
```

το πρόγραμμα θα επιστρέψει αμέσως ψευδή τιμή (false) αν η πρώτη είναι ψευδής, και θα προχωρήσει στον υπολογισμό της δεύτερης μόνο αν η πρώτη είναι αληθής.

Στην πραγματικότητα είναι θέμα εξοικονόμησης χρόνου.

Τελεστές καταχώρησης

Οι τελεστές καταχώρησης χρησιμοποιούνται για να μεταβάλουν τα περιεχόμενα μιας μεταβλητής. Μπορούν να συνδυαστούν και με άλλους τελεστές για συντόμευση κάποιων πράξεων. Παραδείγματα τελεστών καταχώρησης:

```
int x = 4;
x = 10;
x += 20; (είναι το ίδιο με την εντολή x = x + 20, τελικό αποτέλεσμα 30)
x /= 10; (το ίδιο με x = x / 10, αποτέλεσμα 3).
```

Σύμβολο	*Είδος*
`x += y`	το ίδιο με `x = x + y`
`x -= y`	το ίδιο με `x = x - y`
`x *= y`	το ίδιο με `x = x * y`
`x /= y`	το ίδιο με `x = x / y`
`x %= y`	το ίδιο με `x = x % y`
`x &= y`	το ίδιο με `x = x & y`
`x |= y`	το ίδιο με `x = x | y`
`^=`	το ίδιο με `x = x ^ y`

Παραστάσεις (Expressions)

Οποιοσδήποτε έγκυρος συνδυασμός μεταβλητών, σταθερών, αριθμών και τελεστών καλείται μια παράσταση. Το αποτέλεσμα της παράστασης μπορεί να είναι κάποιος από τους τύπους δεδομένων της C++ (`int`, `long`, `double`, `bool`, κλπ) ή κάποιο αντικείμενο

(π.χ. `string`).

Παραδείγματα:

```
int b = 10, i;
i = 2*b*b; (παράδειγμα παράστασης)
if (b * b <= 100 && i > 0)
      cout << "The expression is true" << endl;
string a = "ena, dyo";
string b = a + string(", testing");
```

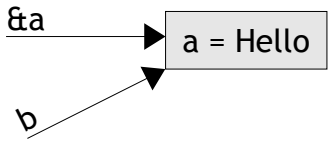
Pointers & References

Στη C και στη C++ δίνεται η δυνατότητα άμεσης προσπέλλασης στα περιεχόμενα της περιοχής της μνήμης που περιέχει κάποια μεταβλητή. Αυτό ίσως φαίνεται κάπως επικίνδυνο, αλλά προσφέρει μεγάλες δυνατότητες στον προγραμματιστή να εισχωρήσει βαθιά στη δομή του προγράμματος. Αυτή η πρόσβαση πραγματοποιείται με δύο τρόπους, με pointers και references. Τα references φαινομενικά είναι ένας πιο κομψός τρόπος χρήσης των pointers αλλά στην ουσία είναι κάτι παραπάνω καθώς προσφέρουν κάποιες επιπλέον δυνατότητες ασφάλειας των δεδομένων. Οι pointers χρησιμοποιούνται κατά κόρον στη δυναμική δημιουργία αντικειμένων με τη `new` (σε επόμενη ενότητα).

Αν έχουμε μια μεταβλητή `a` τύπου `string`, μπορούμε να έχουμε άμεσα την αναφορά (reference) στην διεύθυνση μνήμης που περιέχει το αντικείμενο string πολύ εύκολα με τον τελεστή `&` πρίν το όνομα της μεταβλητής. Δηλαδή το `&a` δείχνει στη διεύθυνση μνήμης της `a`. Αντίστροφα, αν μια μεταβλητή `b` είναι τύπου pointer ενός `string`, τότε έχουμε πρόσβαση στο περιεχόμενο της με τη χρήση του τελεστή `*`, ώστε το `*b` να επιστρέφει το ίδιο το `string`, ενώ το `b` μόνο του να επιστρέφει τη διεύθυνση μνήμης του `string`.

```
string a("Hello"), *b;
b = &a;
cout << a << endl;
cout << *b << endl;
```

Για καλύτερη κατανόηση ακολουθεί μια σχηματική αναπαράσταση της μνήμης που καταναλώνουν τα αντικείμενα αυτά.



3.Ελεγχόμενη ροή προγράμματος

Σε κάθε πρόγραμμα είναι απαραίτητο να ελέγχουμε τη ροή του αναλόγως κάποιες συνθήκες και να την ανακατευθύνουμε κατάλληλα. Υπάρχουν πολλοί τρόποι να κατευθύνουμε τη ροή του προγράμματος και ο κάθε ένας εξυπηρετεί συγκεκριμένο σκοπό.

Η εντολή if

Η εντολή `if` χρησιμοποιείται όταν θέλουμε να εκτελέσουμε κάποιες εντολές μόνο όταν ικανοποιείται κάποια συνθήκη:

```
if (συνθήκη)
{
      εντολές;
}
else
{
      εντολές;
}
```

Μπορούν να συνδυαστούν πολλές εντολές `if` μεταξύ τους, όπως στο ακόλουθο παράδειγμα:

```
if (x == 1) {
      cout << "x is one." << endl;
} else if (x == 2) {
      cout << "x is two." << endl;
} else if (x == 3) {
      cout << "x is three." << endl;
} else if (x == 4) {
      cout << "x is four." << endl;
} else {
      cout << "x is not between 1-4." << endl;
}
```

Η εντολή switch

Η εντολή `switch` χρησιμοποιείται όταν έχουμε πολλαπλές επιλογές ή τιμές για μια μεταβλητή και θέλουμε να εκτελεστούν διαφορετικές εντολές για κάθε τιμή. Με την `switch`, το προηγούμενο παράδειγμα θα πάρει τη μορφή:

```
switch (x) {
      case 1;
            cout << "x is one." << endl;
            break;
      case 2;
            cout << "x is two." << endl;
            break;
      case 3:
            cout << "x is three." << endl;
            break;
      case 4:
            cout << "x is four." << endl;
            break;
      default:
            cout << "x is not between 1-4." << endl;
            break;
      }
```

Προσέξατε την χρήση της `break`. Χωρίς την `break` η εκτέλεση του προγράμματος θα συνεχίσει μέχρι την επόμενη `break` ή μέχρι το τέλος της εντολής `switch`. Αυτό μπορούμε να το εκμεταλευτούμε αν θέλουμε κοινή αντιμετώπιση ορισμένων περιπτώσεων.

```
switch (x) {
      case 1;
      case 2;
      case 3;
      case 4;
            cout << "x is between one and four." << endl;
            break;
      case 5;
      case 6;
      case 7;
      case 8;
            cout << "x is between five and eight." << endl;
            break;
      default:
            cout << "x is not between one and eight." << endl;
            break;
      }
```

Η εντολή for

Η εντολή `for` χρησιμοποιείται για επανάληψη (loop) υπό συνθήκη κάποιων εντολών. Θεωρείται ίσως η πιο δυνατή και ευέλικτη εντολή τύπου `for`, σε οποιαδήποτε γλώσσα (έχει ακριβώς την ίδια λειτουργία με την αντίστοιχη της C).

Η σύνταξή της είναι η εξής:

```
for (εντολή αρχικοποίησης; συνθήκη; εντολή επανάληψης) {
      εντολές;
}
```

Η εντολή αρχικοποίησης (initialization) εκτελείται στην αρχή του loop και στη συνέχεια ελέγχεται αν είναι αληθής ή ψευδής η συνθήκη (condition). Αν είναι αληθής, εκτελούνται οι εντολές και στο τέλος εκτελείται η εντολή επανάληψης (iteration) και ελέγχεται και πάλι αν είναι αληθής η συνθήκη. Αν είναι αληθής εκτελούνται οι εντολές, κ.ο.κ.

Παράδειγμα:

```
for (int i = 1; i < 20; i += 3)  {
      cout << i << endl;
}
```

Το αποτέλεσμα θα είναι το εξής:

```
1
4
7
10
13
16
19
```

Κάθε μία από τις εντολές αρχικοποίησης, επανάληψης και η συνθήκη μπορεί να παραλειφθεί αλλάζοντας τη συμπεριφορά της εντολής `for`. Σημειώνουμε ότι χωρίζονται μεταξύ τους με το ερωτηματικό ';'. Παραδείγματα:

```
int x = 10;
for (; x < 5; x++) {
      cout << x << endl;
}

double y = 20000; // (y = pi)
for (; y >= 10.0;) {
      // υπολογίζει την τετραγωνική του y και τοποθετεί
      // το αποτέλεσμα πάλι στο y.
      // Το for loop θα εκτελείται όσο το y είναι μεγαλύτερο από 10.0
      cout << y);
      y = sqrt(y);
}

for (;;) { // infinite loop
      wait_for_signal();
}
```

Η εντολή while

Η εντολή `while` χρησιμοποιείται αντί της `for` όταν δε μπορούμε να προβλέψουμε εύκολα πόσες φορές θέλουμε να εκτελεστούν οι εντολές, ή όταν δεν έχει σημασία ο αριθμός των επαναλήψεων αλλά η ικανοποιήση ή όχι της συνθήκης:

```
while (συνθήκη) {
      εντολές;
}
```

Παράδειγμα:

```
bool exit_from_loop = false;
while (exit_from_loop == false) {
      // υποθετική ρουτίνα που διαβάζει δεδομένα από ένα αρχείο
      read_bytes(file1);
      // άλλη ρουτίνα που γράφει δεδομένα σε ένα άλλο αρχείο
      write_bytes(file2);
      if (file_empty(file1) == true)
            exit_from_loop = true;
}
```

Η εντολή do/while

Η εντολή `do/while` είναι παρόμοια με την `while`, με μία διαφορά, οι εντολές εντός του loop θα εκτελεστούν τουλάχιστον μια φορά ανεξαρτήτως αν είναι αληθής η συνθήκη ή όχι.

```
do {
      εντολές;
} while (συνθήκη);
```

Παράδειγμα:

```
int x = 10;
do {
      cout << x << endl;
      x++;
} while (x <10);
```

Το αποτέλεσμα θα είναι:

```
10
```

Οι εντολές μέσα στο loop θα εκτελεστούν την πρώτη φορά ακόμη και αν δεν ισχύει η συνθήκη (x < 10).

Οι εντολές break/continue

Πολλές φορές θέλουμε να διακόψουμε την εκτέλεση των εντολών σε ένα loop πρίν από τη προκαθορισμένη στιγμή. Π.χ. να φύγουμε από ένα infinite loop, ή όταν το πρόγραμμα λάβει κάποιο σήμα (signal) για έξοδο (π.χ. Control-C).

```
for (int i = 1; i < 20; i += 3)  {
      if ( i*i > 100)
            break;
      cout << i << endl;
}
```

Το αποτέλεσμα θα είναι:

```
1
4
7
10
```

Η εντολή `break` χρησιμοποιείται σε όλα τα loops, `for`, `while`, `do/while`.

Αντίστοιχη με την `break` είναι η `continue`, η οποία όμως διακόπτει την εκτέλεση μόνο του τρέχονος iteration και όχι ολόκληρου του loop. Συνεχίζει από το επόμενο iteration.

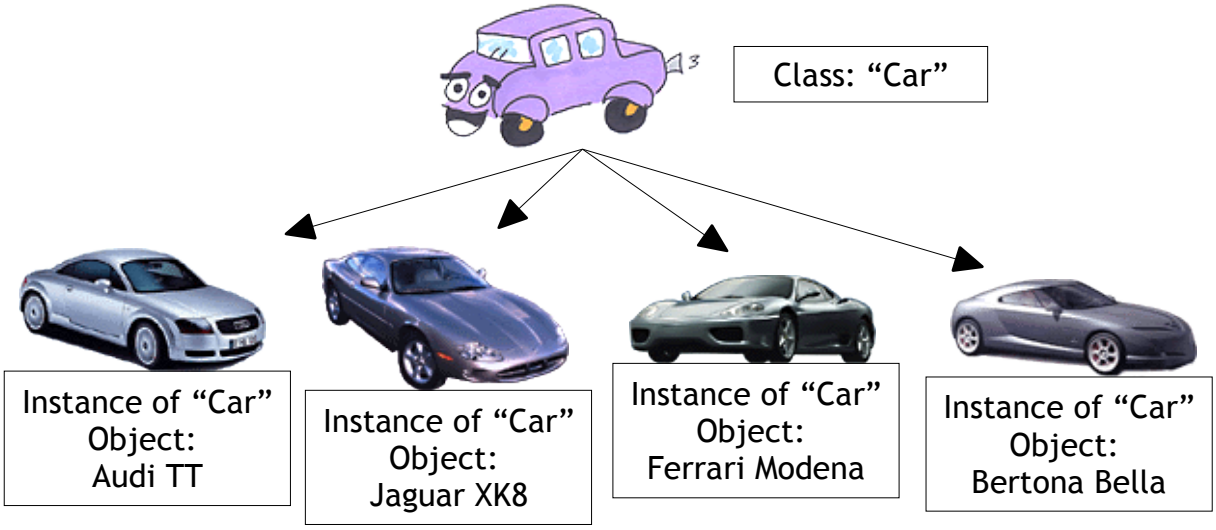
Στο προηγούμενο παράδειγμα:

```
for (int i = 1; i < 20; i++)  {
      // να μην τυπωθούν οι άρτιοι(ζυγοί) αριθμοί
      if ( i % 2 == 0)
            continue;
      cout << i << endl;
}
```

Το αποτέλεσμα θα είναι:

```
1
7
13
19
```

4.Classes



Η αντικειμενοστραφής πλευρά της C++ φαίνεται με την χρήση των αντικειμένων και των κλάσεων στις οποίες ανήκουν τα αντικείμενα. Όλα τα αντικείμενα που έχουν κοινά χαρακτηριστικά ανήκουν στην ίδια κλάση και κάθε ένα από αυτά λέγεται ότι είναι “στιγμιότυπο” (instance) της κλάσης. Για παράδειγμα, η κλάση “αυτοκίνητο” θεωρεί μια γενική εικόνα του αυτοκινήτου με κάποια χαρακτηριστικά που υπάρχουν σε όλα τα στιγμιότυπα. Συγκεκριμένα, η κλάση αυτοκίνητο πιθανώς να ορίζει ότι είναι τετράτροχο, έχει τιμόνι, ταχύτητες, πεντάλ και καθίσματα για τους οδηγούς αλλά δεν ορίζει με σαφήνεια το σχήμα ή τους μηχανισμούς όλων των εξαρτημάτων, ούτε τα χαρακτηριστικά της μηχανής (κυβικά, ίπποι, κύλινδροι, κλπ). Αυτά είναι χαρακτηριστικά που αφορούν το κάθε αντικείμενο ξεχωριστά (ή κάποια υποκατηγορία/υποκλάση της κλάσης “αυτοκίνητο”). Για καλύτερη κατανόηση, δείτε το παραπάνω σχήμα.

Στη C++ η κλάση δηλώνεται με τη λέξη `class`. Στο παράδειγμα του αυτοκινήτου, έστω ότι θέλουμε να δηλώσουμε μια κλάση με το όνομα Car:

```
class Car
{
      (δεδομένα/μεταβλητές)
      (συναρτήσεις/μέθοδοι)
};
```

(προσέξτε το ερωτηματικό στο τέλος του ορισμού της κλάσης!)

member variables

Τα χαρακτηριστικά του αντικειμένου είναι τα δεδομένα γύρω από τα οποία πρέπει να αναπτύξουμε το πρόγραμμά μας. Στο παράδειγμα με το αυτοκίνητο, τα δεδομένα αυτά είναι η γωνία στρέψης του τιμονιού, η πίεση που ασκούμε στα πεντάλ, η θέση του μοχλού ταχυτήτων και άλλα. Στον προγραμματισμό, αυτά τα δεδομένα θα πρέπει να μοντελοποιηθούν, δηλαδή να γίνει η αντιστοιχία τους σε μεταβλητές τις οποίες μπορούμε να επεξεργαστούμε στο πρόγραμμά μας.

Για την κλάση Car, μπορούμε να έχουμε:

```
class Car
{
      // Γωνία στρέψης του τιμονιού
      float steering_angle;

      // Ποσοστό πατήματος του γκαζιού (0 = καθόλου, 100 = τερματισμένο!)
      float gas_pedal;

      // Ποσοστό πατήματος του φρένου (0 = καθόλου, 100 = τερματισμένο!)
      float break_pedal;

      // Ποσοστό πατήματος του συμπλέκτη (0 = καθόλου,
      // 100 = τερματισμένο!)
      float clutch;

      // θέση της τρέχουσας ταχύτητα (πιθανές τιμές: 0, 1,2,3,4,5,
      // 0 = νεκρό, -1 = όπισθεν)
      int gear;

      // μεταβλητές που καθορίζουν την επιτάχυνση, την ταχύτητα του
      // αυτοκινήτου και τις στροφές του κινητήρα
      float acceleration, speed, rpm;
}
```

Εδώ πρέπει να σημειωθεί ότι οι μεταβλητές της κλάσης έχουν διαφορετικές τιμές για κάθε αντικείμενο, και κάθε αντικείμενο έχει πρόσβαση μόνο στις δικές του μεταβλητές. Επίσης, ο τύπος δεδομένων που επιλέχθηκε για κάθε μεταβλητή εξαρτάται από το είδος της πληροφορίας που θα κρατάει αυτή η μεταβλητή. Για παράδειγμα, εφόσον η γωνία στρέψης του τιμονιού θα είναι σε μοίρες, είναι λογικό να χρησιμοποιήσουμε ένα τύπο που θα μπορεί να κρατήσει δεκαδικούς αριθμούς, όπως ο `float`.

Αυτά, λοιπόν, είναι τα δεδομένα μας ομαδοποιημένα υπό την έννοια της κλάσης “Car”. Αυτή τη στιγμή δεν είναι τίποτε παραπάνω από μια ομάδα μεταβλητών χωρίς να έχουμε ορίσει τον τρόπο επεξεργασίας τους. Γι' αυτό είναι απαραίτητη η ύπαρξη του κατάλληλου interface των δεδομένων με τον χρήστη. Το interface επιτυγχάνεται με την ύπαρξη των μεθόδων της κλάσης (member methods).

member methods

Οι μέθοδοι (methods) σε μια κλάση, δεν είναι παρά συναρτήσεις (υπορουτίνες) που προσφέρουν πρόσβαση στα δεδομένα του εκάστοτε αντικειμένου της κλάσης. Η αλληλεπίδραση του χρήστη με κάθε αντικείμενο γίνεται μέσω των μεθόδων της κλάσης, οι οποίες καθορίζουν και τον τρόπο χειρισμού των μεταβλητών του αντικειμένου. Στο παράδειγμά μας, οι μέθοδοι θα καθορίζουν με ποιον τρόπο θα στρίβουμε το τιμόνι, θα αυξάνουμε τη το γκάζι ή το φρένο, θα αλλάζουμε ταχύτητες αλλά και πώς θα μπορούμε να γνωρίζουμε την ταχύτητα του αυτοκινήτου, τις στροφές του κινητήρα δηλαδή πληροφορίες που δε μπορούμε να ελέγξουμε άμεσα.

Πιθανές μέθοδοι για την κλάση “Car” θα μπορούσαν να είναι οι εξής:

```
// Αλλαγή της γωνία στρέψης του τιμονιού, <relative_angle> μοίρες
// σε σχέση με την τρέχουσα γωνία.
void turn_wheel(float relative_angle);

// Πάτημα πεντάλ γκαζιού
void press_gas_pedal(float amount);

// Πάτημα πεντάλ φρένου
void press_break_pedal(float amount);

// Πάτημα πεντάλ συμπλέκτη
void press_clutch_pedal(float amount);

// Αλλαγή της ταχύτητας. Επιστρέφει true αν η αλλαγή ήταν επιτυχής
// ή false αν ήταν ανεπιτυχής (π.χ. από 5 σε όπισθεν).
bool change_gear(int new_gear);

// προβολή της τρέχουσας ταχύτητας, επιτάχυνσης και στροφών του
// κινητήρα
float get_acceleration();
float get_speed();
float get_rpm();
```

Οι παραπάνω μέθοδοι, όπου κρίνεται αναγκαίο επιστρέφουν κάποιο αποτέλεσμα, είτε ως δεκαδικό αριθμό (`float`) στην περίπτωση των μεθόδων `get_*()`, ή ως `bool` στην `change_gear()`. Ο τύπος `void` είναι ειδική περίπτωση που δεν επιστρέφει κάποιο αποτέλεσμα. Χρησιμοποιείται όταν μας ενδιαφέρει η εκτέλεση κάποιας διαδικασίας χωρίς όμως να είναι απαραίτητο να γνωρίζουμε το αποτέλεσμά της, ή μπορεί να μην επιστρέφει κάποιο αποτέλεσμα εξ' αρχής.

Υλοποίηση μιας μεθόδου

Οι παραπάνω είναι απλώς οι δηλώσεις των μεθόδων, δηλαδή δεν περιέχουν καθόλου κώδικα και πρέπει να τις συμπεριλαμβάνουμε στον ορισμό της κλάσης. Για να είναι ολοκληρωμένος ο ορισμός μιας μεθόδου θα πρέπει να συνοδεύεται και από την υλοποίησή της (implementation). Η υλοποίηση συνιστάται να γίνεται σε ξεχωριστό αρχείο (το implementation αρχείο, με κατάληξη `.cpp`) , για παράδειγμα η υλοποίηση της `turn_wheel()` θα μπορούσε να είναι στο αρχείο `car.cpp`:

```
void Car::turn_wheel(float relative_angle)
{
      steering_angle += relative_angle;
      if (steering_angle <= -720.0)
            steering_angle = -720.0;
      if (steering_angle >= 720.0)
            steering_angle = 720.0;
}
```

Όπως βλέπουμε στην υλοποίηση της μεθόδου προσθέσαμε και επιπλέον κώδικα ασφαλείας, ο οποίος απαγορεύει στο τιμόνι να κάνει περισσότερες από 2 στροφές αριστερά ή δεξιά. Παρόμοιοι περιορισμοί σε ένα κανονικό αυτοκίνητο γίνονται με μηχανικό τρόπο, αλλά σε ένα πρόγραμμα (π.χ. εξομοιωτή οδήγησης), το κάθε αντικείμενο πρέπει να θέσει τους δικούς του περιορισμούς και δικλείδες ασφαλείας με τη μορφή κώδικα στις μεθόδους.

Στη C++, συνήθως η δήλωση μιας κλάσης, μεταβλητής ή συνάρτησης γίνεται σε ξεχωριστό αρχείο, το αρχείο κεφαλίδας του οποίου το όνομα λήγει σε `.h` ή `.hpp`, ενώ το αρχείο ορισμού -δηλαδή το αρχείο που περιέχει τον πηγαίο κώδικα ορισμού της αντίστοιχης κλάσης, μεταβλητής ή συνάρτησης- λήγει σε `.cpp` ή `.cxx`. Δηλαδή για την κλάση `Car` θα έχουμε δύο αρχεία, το αρχείο κεφαλίδας `Car.h` και το αρχείο ορισμού `Car.cpp`. Αυτός ο διαχωρισμός εξυπηρετεί τη γρήγορη χρήση της κλάσης μέσα σε άλλες κλάσεις ή συναρτήσεις, φορτώνοντας το αρχείο κεφαλίδας μόνο και διαβάζοντας έτσι μόνο τις απαραίτητες πληροφορίες για την χρήση της κλάσης, δηλαδή τις δηλώσεις των μεταβλητών και των μεθόδων που περιέχει.

Δημιουργία αντικειμένων

Έχοντας ορίσει την κλάση μας, θα πρέπει να δημιουργήσουμε τα αντικείμενα/ στιγμιότυπα (instances) της κλάσης. Στη C++, μπορούμε να δημιουργήσουμε αντικείμενα με δύο τρόπους, στατικά και δυναμικά. Η στατική δημιουργία γίνεται όπως και ο ορισμός κάποιας μεταβλητής ενώ η δυναμική γίνεται με τη χρήση της εντολής `new` της C++.

Η `new` δημιουργεί μια φυσική αναπαράσταση της κλάσης, ένα μοναδικό στιγμιότυπο, και αν είναι επιτυχής επιστρέφει ένα δείκτη (pointer) σε αυτό, διαφορετικά επιστρέφει μηδέν (0). Με αυτό το δείκτη μπορούμε να προσπελάσουμε το αντικείμενο με οποίον τρόπο θέλουμε (και εφόσον το επιτρέπει το ίδιο το αντικείμενο). Έτσι για την κλάση `Car` η δημιουργία ενός αντικειμένου στατικά και δυναμικά γίνεται ως εξής (το αντικείμενο `acar` δημιουργείται στατικά, ενώ το `anothercar` δημιουργείται δυναμικά):

```
Car acar();
Car *anothercar = new Car();
```

Αν ο constructor δεν παίρνει παραμέτρους μπορούμε να αποφύγουμε τις παρενθέσεις, δηλαδή το ακόλουθο είναι ισοδύναμο:

```
Car acar;
Car *anothercar = new Car;
```

Όσον αφορά το αντικείμενο `anothercar`, η δημιουργία του δεν είναι αναγκαίο να γίνει κατά την δήλωσή του. Το ίδιο αποτέλεσμα μπορούμε να έχουμε και με τις εντολές:

```
Car *anothercar;
anothercar = new Car;
```

Τα δεδομένα του κάθε αντικειμένου (οι μεταβλητές) αλλά και οι μέθοδοι της κλάσης που αντιστοιχούν στο αντικείμενο, μπορούν να προσπελλαστούν ως εξής:

```
// Η γωνία στρέψης του τιμονιού του acar
acar.steering_angle

// Η γωνία στρέψης του τιμονιού του anothercar
anothercar->steering_angle

// Εντολή στο acar να στρίψει δεξιά 13.4 μοίρες.
acar.turn(13.4);

// Επιστρέφει την τρέχουσα ταχύτητα του acar
float speed = acar.get_speed();

// Εντολή στο anothercar να στρίψει αριστερά 32 μοίρες
anothercar->turn(-32.0);
```

```
// Εντολή στο anothercar να βάλει όπισθεν
bool result = anothercar->ghange_gear(-1);
```

Αν η προσπέλαση κάποιου αντικειμένου γίνεται μέσω pointer τότε χρησιμοποιούμε τον τελεστή '->' (όπως π.χ. κάνουμε με τα δυναμικά αντικείμενα), διαφορετικά χρησιμοποιούμε τον τελεστή '.'.

Όπως βλέπουμε ορισμένοι μέθοδοι δέχονται κάποιες παραμέτρους (εντός παρενθέσεων). Οι παράμετροι μπορεί να είναι μεταβλητές οποιουδήποτε αποδεκτού τύπου στη C++ ή ακόμη και άλλα αντικείμενα. Αν έχουμε περισσότερες από μία παραμέτρους τις διαχωρίζουμε με κόμμα ','.

Constructors

Όταν δημιουργείται ένα αντικείμενο (στατικά ή δυναμικά με την εντολή `new`), στην πραγματικότητα η C++, αφού δεσμεύσει την απαραίτητη μνήμη για το αντικείμενο, εκτελεί μια συγκεκριμένη μέθοδο της κλάσης, τον δημιουργό (constructor). Ο δημιουργός πραγματοποιεί τις απαραίτητες ενέργειες για να καταστεί κάποιο αντικείμενο έτοιμο προς χρήση. Αυτές μπορεί να είναι κάτι απλό όπως η ρύθμιση κάποιων μεταβλητών με αρχικές τιμές, ή πιο περίπλοκο όπως η δημιουργία σύνδεσης με μια βάση δεδομένων, η αρχικοποίηση των πινάκων SQL, η δέσμευση κάποιων δικτυακών θυρών (sockets) για κάποιο server ή ακόμη και το άνοιγμα κάποιου παραθύρου για εμφάνιση γραφικής πληροφορίας.

Ο δημιουργός έχει τη μορφή μιας μεθόδου με το όνομα της κλάσης και χωρίς τύπο (δηλαδή δεν δηλώνεται ο τύπος δεδομένων που επιστρέφει, το οποίο είναι διαφορετικό από το να δηλωθεί ως void).

Για παράδειγμα, στην κλάση “Car”, ένας πιθανός δημιουργός μπορεί να είναι:

```
Car::Car()
{
	steering_wheel = 0.0;
	gas_pedal = 0.0;
	break_pedal = 0.0;
	float clutch = 0.0;
	int gear = 0;
	acceleration = 0.0;
	speed = 0.0;
	rpm = 0.0;
}
```

Δηλαδή, ο δημιουργός πραγματοποιεί την αρχικοποίηση (initialization) των μεταβλητών του αντικειμένου ώστε αυτό να είναι έτοιμο προς χρήση.

Μπορούμε να έχουμε περισσότερους από έναν δημιουργούς, οι οποίοι να δέχονται διαφορετικές παραμέτρους ο καθένας.

Για παράδειγμα, αν υποθέσουμε ότι μπορούσαμε να ορίσουμε χαρακτηριστικά του κινητήρα (`engine_cc`: κυβικά, `engine_hp`: ίπποι) στη δημιουργία του αντικειμένου, θα μπορούσαμε να έχουμε έναν επιπλέον δημιουργό:

```
Car::Car(int cc, int hp)
{
	engine_cc = cc;
	engine_hp = hp;
	// Ακολουθούν οι υπόλοιπες εντολές αρχικοποίησης του αντικειμένου
}
```

επιπλέον του αρχικού δημιουργού.

Η δημιουργία περισσότερων από έναν δημιουργούς καλείται constructor overloading και είναι υποπερίπτωση του χαρακτηριστικού της C++, method overloading (σε επόμενη παράγραφο).

Destructors

Στη C++, ό,τι αντικείμενο δημιουργούμε δυναμικά πρέπει να το καταστρέφουμε (συνήθως με την εντολή `delete`) ελευθερώνοντας όλους τους πόρους που δεσμεύσαμε κατά την ύπαρξή του (κλείσιμο αρχείων, αποσύνδεση από βάσεις δεδομένων, τερματισμός threads, αποδέσμευση μνήμης, κλπ). Η καταστροφή του αντικειμένου γίνεται καλώντας μια μέθοδο που καλείται καταστροφέας (destructor). Το όνομα του destructor είναι το ίδιο με της κλάσης προπορευόμενο από τον τελεστή ~. Δηλαδή για την κλάση `Car`, το όνομα του καταστροφέα είναι `~Car()`.

Ενώ ο δημιουργός είναι απαραίτητο να οριστεί τις περισσότερες φορές, δεν είναι πάντα απαραίτητος ο ορισμός του καταστροφέα. Κάτι τέτοιο έχει νόημα μόνο όταν στον δημιουργό πραγματοποιούμε κάποια δυναμική λειτουργία (άνοιγμα δικτυακής σύνδεσης, αρχείου, κλπ), και την οποία πρέπει να τερματίσουμε κατά την καταστροφή του αντικειμένου (π.χ. κλείσιμο αρχείου, τερματισμός σύνδεσης, κλπ).

Ο δείκτης αναφοράς this

Μέσα σε μια μέθοδο, μπορούμε να χρησιμοποιήσουμε μια μεταβλητή της κλάσης απλώς με το όνομά της, αναφερόμενοι φυσικά στην τιμή που έχει η μεταβλητή για το συγκεκριμένο αντικείμενο. Τις περισσότερες φορές κάτι τέτοιο είναι αρκετό, υπάρχουν όμως και περιπτώσεις που χρειάζεται πιο ρητή αναφορά στο συγκεκριμένο αντικείμενο. Για το σκοπό αυτό μπορούμε να χρησιμοποιήσουμε τη μεταβλητή `this` που επιστρέφει πάντα ένα δείκτη στο τρέχον αντικείμενο (δηλαδή αυτό που καλεί την μέθοδο). Με το δείκτη `this`, η μέθοδος `turn_wheel()` που είδαμε παραπάνω μετασχηματίζεται ως εξής:

```
void turn_wheel(float relative_angle)
{
      this->steering_angle += relative_angle;
      if (this->steering_angle <= -720.0)
            this->steering_angle = -720.0;
      if (this->steering_angle >= 720.0)
            this->steering_angle = 720.0;
}
```

Ο δείκτης `this` είναι ιδιαίτερα χρήσιμος ειδικά σε περιπτώσεις που μεταχείριζομαστε περισσότερα από ένα όμοια αντικείμενα στην ίδια μέθοδο, για αποφυγή σύγχυσης.

Method Overloading

Σε ένα πρόγραμμα, υπάρχει περίπτωση να χρειαστεί να εκτελέσουμε την ίδια διαδικασία με ελαφρώς διαφορετικά δεδομένα. Αυτό συνήθως σημαίνει ότι θα χρειαστεί να έχουμε διαφορετικές συναρτήσεις/μεθόδους για κάθε διαφορετική παράμετρο που δίνουμε. Για παράδειγμα, ας υποθέσουμε ότι ο χρήστης καλεί την μέθοδο `turn_wheel()` με παράμετρο ένα `int` αντί για `float`. Σε μια γλώσσα όπως η C θα έπρεπε να έχουμε δύο συναρτήσεις/μεθόδους, μια για κάθε διαφορετική παράμετρο και μάλιστα με διαφορετικό όνομα:

```
void Car::turn_wheel_int(int relative_angle)
{
```

```
	this->steering_angle += (float) relative_angle;
	if (this->steering_angle <= -720.0)
		this->steering_angle = -720.0;
	if (this->steering_angle >= 720.0)
		this->steering_angle = 720.0;
}

void Car::turn_wheel_float(float relative_angle)
{
	steering_angle += relative_angle;
	if (steering_angle <= -720.0)
		steering_angle = -720.0;
	if (steering_angle >= 720.0)
		steering_angle = 720.0;
}
```

Φυσικά, το συγκεκριμένο παράδειγμα είναι αρκετά απλό και δεν αποτελεί πρόβλημα. Μπορούμε μάλιστα να τροποποιήσουμε την `turn_wheel_int()` ως εξής:

```
void Car::turn_wheel_int(int relative_angle)
{
	turn_wheel_float((float) relative_angle);
}
```

Στην πραγματικότητα, αν έχουμε κάποιο πιο περίπλοκο πρόγραμμα με μερικές δεκάδες παραλλαγές μεθόδων και κάποιες χιλιάδες γραμμές κώδικα, η παραπάνω τεχνική δυσκολεύει το debugging, προκαλεί διπλασιασμό του κώδικα και δυσχεραίνει τη συντήρηση του προγράμματος. Στην περίπτωση αυτή επιβάλλεται η χρήση μιας Object-Oriented γλώσσας όπως η Java ή η C++ που παρέχουν δυνατότητες Method Overloading. Η τεχνική του method overloading, επιτρέπει τον ορισμό διάφορων παραλλαγών μιας μεθόδου αναλόγως τις ζητούμενες παραμέτρους που δέχεται αυτή. Το παραπάνω παράδειγμα, θα μετασχηματιστεί στην εξής μορφή:

```
void Car::turn_wheel(float relative_angle)
{
	steering_angle += relative_angle;
	if (steering_angle <= -720.0)
		steering_angle = -720.0;
	if (steering_angle >= 720.0)
		steering_angle = 720.0;
}

void Car::turn_wheel(int relative_angle)
{
	turn_wheel((float) relative_angle);
}
```

Το όνομα της μεθόδου παραμένει `turn_wheel()` και μπορούμε να έχουμε όσες παραλλαγές θέλουμε όσον αφορά το είδος και το πλήθος των παραμέτρων που δέχεται. Παρ' όλ' αυτά, ο τύπος δεδομένων που επιστρέφει η μέθοδος (ή `void` αν δεν επιστρέφει κάτι) θα πρέπει να είναι ο ίδιος σε κάθε παραλλαγή της μεθόδου.

5.Επιπλέον Τύποι Δεδομένων

Πίνακες

Αν και έχουμε ήδη αναφέρει τους σημαντικότερους τύπους δεδομένων στη C++ σε προηγούμενη παράγραφο, αφήσαμε σκόπιμα την αναφορά στους πίνακες. Ένας λόγος είναι ότι στη C++, οι πίνακες υλοποιούνται είτε ως παραδοσιακοί πίνακες (όπως στη C) είτε ως αντικείμενα.

Οι πίνακες γενικά χρησιμοποιούνται για την οργάνωση και καταχώρηση όμοιων αντικειμένων. Μπορούν να είναι μίας ή πολλών διαστάσεων και ο τρόπος προσπέλασης των στοιχείων είναι ο ίδιος με αυτόν της C.

Στη C++ μπορούμε να ορίσουμε ένα μονοδιάστατο πίνακα πολύ απλά ως εξής:

```
type table[size];
```

Ως `type` θεωρούμε τον τύπο δεδομένων των αντικειμένων του πίνακα και μπορεί να είναι είτε ένας από τους απλούς τύπους (`bool`, `char`, `short`, `int`, `long`, `float`, `double`, `char`) είτε το όνομα μιας κλάσης αντικειμένων. Το `size` απεικονίζει το μέγεθος του πίνακα `table`.

Όπως και στη C, μπορούμε να προσπελλάσουμε τα στοιχεία του πίνακα με την σύνταξη `table[i]`, όπου `i` η θέση του στοιχείου που μας ενδιαφέρει. Εδώ πρέπει να σημειωθεί ότι σε αντιστοιχία με τη C η C++ πραγματοποιεί την αρίθμηση των πινάκων από το μηδέν (`0`) ως το `size-1`. Δηλαδή αν έχουμε έναν πίνακα `A` με 10 στοιχεία το πρώτο στοιχείο είναι το `A[0]` και το τελευταίο το `A[9]`.

Σημειώνεται ότι ενώ στη Java οι πίνακες είναι πραγματικά αντικείμενα και το μέγεθος ενός πίνακα `A` δίνεται από τη μεταβλητή `A.length`, στη C++ δεν υπάρχει τέτοια δυνατότητα και το μέγεθος το λαμβάνουμε με τη χρήση της συνάρτησης `sizeof()`. Η ίδια όμως επιστρέφει το μέγεθος σε bytes και όχι σε στοιχεία του πίνακα. Ακολουθεί ένα παράδειγμα για καλύτερη κατανόηση:

```
int data[10];
int datasize = sizeof(data) / sizeof(int);
int i;
cout << "Size of array data: " << sizeof(data) / sizeof(int) << endl;
for (i = 0; i < datasize; i++) {
      data[i] = i*i;
      cout << "data[" << i << "] = " << data[i] << endl;
}
```

Το αποτέλεσμα του κώδικα αυτού θα είναι:

```
Size of array data: 10
data[0] = 0
data[1] = 1
data[2] = 4
data[3] = 9
data[4] = 16
data[5] = 25
data[6] = 36
data[7] = 49
data[8] = 64
data[9] = 81
```

Σε περίπτωση που θέλουμε να αρχικοποιήσουμε ένα πίνακα, δηλαδή να ορίσουμε αρχικές τιμές για τα στοιχεία του, αυτό μπορούμε να το πετύχουμε με την εξής

συντακτική δομή:

```
int dataset[] = { 22, 3, 54, 43, 199, 20, 20, 67, 7, 80 };
```

Πολυδιάστατοι πίνακες

Δε θα ήταν πλήρης η υποστήριξη των πινάκων σε μια γλώσσα προγραμματισμού όπως η C++, αν αυτή δεν υποστήριζε πολυδιάστατους πίνακες.

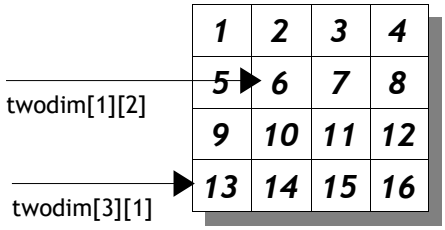
Για παράδειγμα, ένας δισδιάστατος πίνακας μπορεί να δηλωθεί ως εξής:

```
int twodim[4][4];
int arraysize = 4;
int i, j, counter = 1;
for (i = 0; i < arraysize; i++) {
      for (j = 0; j < arraysize; j++) {
            twodim[i][j] = counter;
            counter++;
            cout << twodim[i][j] << " ";
      }
      cout << endl;
}
```

Το αποτέλεσμα αυτού του κώδικα θα είναι:

```
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
```

Δημιουργήσαμε έτσι έναν δισδιάστατο πίνακα, του οποίου τα στοιχεία τα προσπελλάζουμε όπως δείχνει το σχήμα:



Περισσότερα για τα Strings

Τα strings τα έχουμε ήδη αναφέρει και τα χρησιμοποιήσαμε μερικές φορές στην εντολή `cout`. Αντίθετα με τα strings σε άλλες γλώσσες προγραμματισμού (C, PASCAL) που είναι απλώς πίνακες χαρακτήρων, στη C++ τα strings είναι κανονικά αντικείμενα, που υλοποιούνται με την κλάση `string` (στο namespace `std`). Φυσικά, για λόγους συμβατότητας με τη C, υπάρχει πλήρης υποστήριξη των strings υπό την μορφή πινάκων χαρακτήρων (`char *` ή `char []`).

Πέρα από την απευθείας χρήση τους που είδαμε στην `cout`, μπορούμε να δημιουργήσουμε αντικείμενα `string`, με τον ίδιο τρόπο όπως και με κάθε άλλο αντικείμενο, δηλαδή στατικά ή δυναμικά (με τη χρήση της `new`).

```
std::string str("Hello");
std::string str2 = " there";
std::string *str3 = new std::string("Hello there");
```

```
cout << str << str2 << endl;
cout << *str2 << endl;
```

Η κλάση `string` παρέχει ορισμένες μεθόδους, οι οποίες είναι αρκετά χρήσιμες για επεξεργασία του κειμένου μέσα στο `string`. Παραθέτουμε τις σημαντικότερες από αυτές στον ακόλουθο πίνακα:

Όνομα	*Λειτουργία*
`bool empty()`	Επιστρέφει `true` αν το αντικείμενο `string` είναι κενό.
`int length()`	Επιστρέφει το μήκος (σε χαρακτήρες) του `string`.
`reference operator[](int index)`	Επιστρέφει ένα δείκτη αναφοράς (reference) στον χαρακτήρα που βρίσκεται στη θέση `index` του `string`.
`int compare(string &str)`	Συγκρίνει δύο αντικείμενα `string`. Αν το καλόν αντικείμενο (δηλ. αυτό που καλεί την `compare()`) είναι μικρότερο από το `str`, τότε επιστρέφει αρνητικό αποτέλεσμα, μηδέν αν έχουν το ίδιο περιεχόμενο, ή θετικό αποτέλεσμα αν το καλόν `sttring` είναι μεγαλύτερο από το `str`.
`int find(string &str)`	Αναζητά το `str` μέσα στο καλόν αντικείμενο `string`. Αν βρεθεί επιστρέφει τη θέση της πρώτης εμφάνισής του, αλλιώς -1.
`int find_last_of(string str)`	Αναζητά το `str` μέσα στο καλόν αντικείμενο `string`. Αν βρεθεί επιστρέφει τη θέση της τελευταίας εμφάνισής του, αλλιώς -1.

Ακολουθεί ένα παράδειγμα χρήσης των strings:

```
string str1("Hello there, from C++!");
string str2 = "One two three four";
string str3 = "C++ strings are cool!";
string *str4 = new string(str3);
int index, result;

cout << "str1 is " << str1.length() << " characters long.");
for (int i=0; i < str1.length(); i++)
      cout << str1[i] << "|";
cout << endl;

if (str3 == *str4)
      cout << "str3 == str4" << endl;
else
      cout << "str3 != str4" << endl;

if (str3 == str2)
      cout << "str3 == str2" << endl;
else
      cout << "str3 != str2" << endl;

result = str3.compare(str1);
if (result < 0)
      cout << "str3 < str1" << endl;
else if (result == 0)
      cout << "str3 == str1" << endl;
```

```
else
      cout << "str3 > str1" << endl;

index = str1.find("C++");
if (index != -1)
      cout << "'C++' exists in str1 in position " << index << endl;
else
      cout << "'C++' does not exist in str1" << endl;

index = str2.find("C++");
if (index != -1)
      cout << "'C++' exists in str2 in position " << index << endl;
else
      cout << "'C++' does not exist in str2" << endl;

index = str3.find("C++");
if (index != -1)
      cout << "'C++' exists in str3 in position " << index << endl;
else
      cout << "'C++' does not exist in str3" << endl;
```

Το αποτέλεσμα του κώδικα αυτού θα είναι:

```
str1 is 22 characters long.
H|e|l|l|o| |t|h|e|r|e|,| |f|r|o|m| |C|+|+|!|
str3 == str4
str3 != str2
str3 < str1
'C++' exists in str1 in position 18
'C++' does not exist in str2
'C++' exists in str3 in position 0
```

6.Κατασκευάζοντας ένα πρόγραμμα

Στο σημείο αυτό έχουμε ορίσει τα περισσότερα από τα βασικά εργαλεία για να γράψουμε ένα πλήρες πρόγραμμα σε C++. Δεν έχουμε αναφέρει όμως τη δομή ενός προγράμματος, τον τύπο αρχείων που θα χρησιμοποιηθούν ή πως θα τα μεταγλωττίσουμε και εκτελέσουμε.

Η ρουτίνα main()

Στη C και στη C++, κάθε πρόγραμμα ξεκινά την εκτέλεσή του από την ρουτίνα `main()`. Η ρουτίνα αυτή μπορεί να βρίσκεται σε οποιοδήποτε αρχείο πηγαίου κώδικα του προγράμματός μας, αλλά μπορεί να είναι μόνο μία για κάθε εκτελέσιμο πρόγραμμα. Σε αντίθεση με τη Java, δεν αποτελεί μέρος μιας κλάσης αλλά είναι αυτόνομη ρουτίνα. Το ακόλουθο είναι το πιο απλό παράδειγμα προγράμματος C++:

```
#include <iostream>

int main(int argc, char *argv[]) {
      std::cout << "hello everyone" << std::endl;
}
```

Με την εντολή `#include` εισάγουμε την κεφαλίδα που δηλώνεται η χρήση των βασικών stream εισόδου και εξόδου της C++, των `cin`, `cout` και `cerr` αντίστοιχα (περισσότερα σε επόμενη ενότητα). Η κεφαλίδα αυτή είναι η `iostream`.

Όσον αφορά την `int` αναφέρεται στο ότι η ρουτίνα `main()` επιστρέφει στο λειτουργικό σύστημα έναν κωδικό επιτυχίας ή αποτυχίας εκτέλεσης του προγράμματος. Σε όλα τα λειτουργικά συστήματα και για λόγους συμβατότητας, ο κωδικός αυτός είναι μηδέν (0) για επιτυχή εκτέλεση του προγράμματος και μή μηδενικό (5, 10, ή άλλο) σε περίπτωση ελεγχόμενου τερματισμού του προγράμματος π.χ γιατί δεν είχε αρκετή μνήμη ή δικαιώματα εγγραφής σε κάποια αρχεία, ενώ ο κωδικός αυτός αποκτά μεγάλη τιμή όταν το πρόγραμμα τερματίσει απότομα λόγω σφάλματος (crash).

Οι παράμετροι `argc`, `argv` παίζουν το ρόλο της παραμέτρου `args[]` στη Java. Δηλαδή περιέχουν τις παραμέτρους με τις οποίες καλείται το πρόγραμμα από την γραμμή εντολών. Ο λόγος που έχουμε δύο παραμέτρους είναι ότι ένας πίνακας στη C/C++ δεν παρέχει κάποιο εύκολο τρόπο πληροφόρησης του μεγέθους του. Έτσι ενώ ο πίνακας `argv[]` περιέχει τις παραμέτρους σε strings της C (ακολουθίες χαρακτήρων `char` και *όχι* αντικείμενα `string`), δεν είναι δυνατό να γνωρίζουμε το πλήθος αυτών των παραμέτρων χωρίς την `argc`. Σε επόμενη παράγραφο ακολουθεί παράδειγμα χρήσης των `argc`, `argv`.

Το `cout` το έχουμε ήδη δεί αρκετές φορές σε προηγούμενα παραδείγματα, είναι το stream που αντιστοιχεί στη πρότυπη έξοδο (standard output) όπου τυπώνονται τα μηνύματα (την ίδια λειτουργία σε άλλες γλώσσες έχουν εντολές όπως `print`, `printf`, `write`, κλπ). Περισσότερα για τα streams στη C++ θα δούμε σε επόμενη ενότητα. Το `endl` συμβολίζει το χαρακτήρα νέας γραμμής (new line) και έχει ακριβώς την ίδια λειτουργία με την εκτύπωση του χαρακτήρα "`\n`" (για την ακρίβεια κάνει και flush το stream εξόδου).

Θα παρατηρήσατε ίσως ότι το `cout` και το `endl` έχουν το πρόθεμα `std` με διπλές άνω και κάτω τελείες "`::`". Το πρόθεμα αυτό συμβολίζει το namespace στο οποίο ανήκει το αντικείμενο stream `cout` και το `endl`, δηλαδή στην ουσία το πεδίο ισχύος τους. Αν ο κώδικάς μας ανήκε στο ίδιο namespace `std`, δε θα ήταν αναγκαία η χρήση του προθέματος "`std::`", αλλά το συγκεκριμένο namespace είναι ήδη καθορισμένο και δεν συνιστάται η τροποποίηση ή προσθήκη άλλων κλάσεων ή αντικειμένων. Οποιαδήποτε

χρήση αντικειμένου εκτός κάποιου namespace απαιτεί το πρόθεμα του ονόματος του namespace (στην προκειμένη περίπτωση το `std`) ακολουθούμενο από '`::`'. Εναλλακτικά, θα μπορούσαμε να χρησιμοποιήσουμε στην αρχή του προγράμματος (εκτός της ρουτίνας `main()`) την εντολή

```
using namespace std;
```

ώστε να μετασχηματιστεί το πρόγραμμά μας στο εξής:

```
using namespace std;

#include <iostream>

int main(int argc, char *argv[]) {
      cout << "hello everyone" << endl;
}
```

Τα δύο προγράμματα είναι απολύτως ισοδύναμα.

Οι παράμετροι της main()

Όπως αναφέραμε σε προηγούμενη ενότητα, η `main()` δέχεται κάποιες παραμέτρους `argc` και `argv[]`. Δηλαδή, επιτρέπει στο πρόγραμμά μας να παραμετροποιεί την δράση του με βάση κάποιες επιλογές του χρήστη στη γραμμή εντολών (DOS/Command Prompt στα Windows, shell στο Linux).

Ακολουθεί ένα απλό παράδειγμα που απλώς τυπώνει τις παραμέτρους που περνάμε στο πρόγραμμα:

```
#include <iostream>

using namespace std;

int main(int argc, char *argv[]) {
      for (int i=0; i < argc; i++)
            cout << argv[i] << endl;
}
```

Αυτό το αποθηκεύουμε ως `ArgsExample.cpp` και πολύ απλά το μεταγλωττίζουμε. Το αποτέλεσμα θα είναι ένα εκτελέσιμο αρχείο `ArgsExample`, το οποίο εκτελούμε δίνοντας του κάποιες παραμέτρους:

```
C:\> ArgsExample hello mate "what's up?"
ArgsExample
hello
mate
what's up?
```

Το πρόγραμμα δέχτηκε τέσσερις παραμέτρους (`ArgsExample`, `hello`, `mate` και `"what's up?"`) τις οποίες τυπώνει μία σε κάθε γραμμή. Σε αντίθεση με τη Java, στη C/C++ η πρώτη παράμετρος (`argv[0]`) είναι πάντα το όνομα του προγράμματος που εκτελείται.

Εδώ σημειώνουμε ότι οι παράμετροι γενικά χωρίζονται με κενούς χαρακτήρες (space, tab) εκτός αν περικλείονται ανάμεσα σε εισαγωγικά "", οπότε θεωρούνται και οι κενοί χαρακτήρες μέρος της παραμέτρου (όπως και με τη παράμετρο `"what's up?"` πιο πάνω).

7.Προχωρημένα θέματα

Encapsulation

Ας ορίσουμε μια κλάση Person η οποία θα περιέχει ορισμένες πληροφορίες για ένα άτομο, όπως π.χ. το όνομά του, την ηλικία του, το τηλέφωνό του και τη διεύθυνση email του. Μια τέτοια κλάση μπορεί να χρησιμοποιηθεί π.χ. σε ένα πρόγραμμα ατζέντας ή ακόμη και ως βάση σε πρόγραμμα πελατολογίου, ασθενών, κλπ.

```
class Person
{
public:
      // οι μεταβλητές της κλάσης
      string Firstname_, Lastname_;
      int Age_;
      string Telephone_;
      string Email_;

      // ο constructor
      Person(string fname, string lname, int age, string tel,
                  string email)
      {
            Firstname_ = fname;
            Lastname_ = lname;
            Age_ = age;
            Telephone_ = tel;
            Email_ = email;
      }
}
```

Ίσως παρατηρήσατε ότι τα ονόματα των μεταβλητών της κλάσης λήγουν σε "_". Φυσικά, κάτι τέτοιο δεν είναι αναγκαίο, είναι όμως μια συνηθισμένη πρακτική και βοηθάει στην αναγνώριση και διαχωρισμό των απλών μεταβλητών από τις μεταβλητές μέλη μιας κλάσης.

Αφού ορίσαμε την κλάση, μπορούμε να προχωρήσουμε στην δημιουργία κάποιων αντικειμένων της κλάσης:

```
Person bilbo("Bilbo", "Baggins", 111, "+306970123456",
               "bilbobaggins@theshire.net");
```

Με αυτόν τον τρόπο, δημιουργήσαμε το αντικείμενο `bilbo` που αντιστοιχεί στο άτομο Bilbo Baggins, 111 ετών με τηλ. 306970123456 και email bilbobaggins@theshire.net.

Όπως είναι οι πληροφορίες που περιγράφουν το άτομο είναι προσβάσιμες σε όλους. Αυτό σημαίνει ότι αν με κάποιον τρόπο αποκτήσουμε πρόσβαση στο αντικείμενο `bilbo`, θα μπορούμε να αλλάξουμε οποιαδήποτε πληροφορία θελήσουμε και με οποιονδήποτε τρόπο. Δηλαδή, να δώσουμε μια μη έγκυρη διεύθυνση email στο πεδίο `email_` ή μια άσχετη πληροφορία στο πεδίο `telephone_`. Και μάλιστα με πολύ απλό τρόπο:

```
bilbo.Firstname_ = "μπίλμπο";
bilbo.Lastname_  = "μπαγκινσόπουλος";
bilbo.Age_       = 3;
bilbo.Email_     = "this is definitely not a valid email address";
bilbo.Telephone_ = "yeah, try to call this";
```

Πρόκειται για τρανή παραβίαση των προσωπικών δεδομένων!!!

Πώς μπορούμε να αποφύγουμε τέτοιου είδους μη προβλεπόμενη μετατροπή των

δεδομένων ενός αντικειμένου;

Η C++ καθώς και οι περισσότερες γλώσσες προγραμματισμού που έχουν σχεδιαστεί γύρω από το μοντέλο του αντικειμενοστραφούς προγραμματισμού, προβλέπουν τον περιορισμό της πρόσβασης των δεδομένων σε επίπεδα. Ένα από τα επίπεδα πρόσβασης είναι η πρόσβαση χωρίς περιορισμό σε όλους, που ορίζεται με τη λέξη `public`.

Ακριβέστερα, η `public` ορίζει μια περιοχή δηλώσεως μεθόδων ή μεταβλητών (γενικότερα μέλη της κλάσης). Κάθε μέλος που βρίσκεται στην περιοχή public θα είναι προσβάσιμο από οπουδήποτε μέσα στην ίδια κλάση ή εκτός της κλάσης. Το αντίστροφο, δηλαδή ο περιορισμός της πρόσβασης γίνεται με τη χρήση της λέξης `private`. Η `private` περιορίζει την πρόσβαση των μεταβλητών ή των μεθόδων που βρίσκονται στην αντίστοιχη περιοχή, μόνο στην συγκεκριμένη κλάση (και φυσικά σε αντικείμενα αυτής). Οποιαδήποτε πρόσβαση εκτός, θα αποτρέπεται στο επίπεδο της μεταγλώττισης ακόμη. Για παράδειγμα η παραπάνω κλάση `Person`, με τη χρήση της `private`, θα μετασχηματιστεί ως εξής:

```
class Person
{
private:
      // οι μεταβλητές της κλάσης
      private string Firstname_, Lastname_;
      private int Age_;
      private string Telephone_;
      private string Email_;
public:
      ...
```

Αυτό όμως σημαίνει ότι δεν θα είναι πλέον δυνατή η πρόσβαση σε οποιαδήποτε πληροφορία του ατόμου ακόμη και για απλή ανάγνωση! Κάτι τέτοιο δεν είναι επιθυμητό και πρέπει να βρεθεί τρόπος να επιτραπεί έστω και ελεγχόμενη πρόσβαση στα δεδομένα.

Ακριβώς, αυτό επιτυγχάνουμε με την χρήση των μεθόδων της κλάσης. Ελεγχόμενη πρόσβαση για ανάγνωση αλλά και μετατροπή των δεδομένων. Συνήθως και για τις περισσότερες περιπτώσεις κλάσεων, αρκεί ο ορισμός ενός ζεύγους μεθόδων για κάθε μεταβλητή της κλάσης, μία για ανάγνωση και μια για μετατροπή της τιμής της μεταβλητής (ένα ζεύγος getter/setter όπως λέγονται συχνά).

Για παράδειγμα, για την κλάση `Person` παραθέτουμε πιθανές μεθόδους get/set για ορισμένες από τις μεταβλητές (`Age_` και `Email_`):

```
// Return the age
int Person::getAge()
{
      return Age_;
}

// return the Email address
string Person::getEmail()
{
      return Email_;
}

// method to set the age of the person
bool Person::setAge(int Age)
{
      // check if given Age is non-negative (> 0)
      if (Age > 0)
      {
```

```
            Age_ = Age;
            return true;
      } else
            return false;
}

// method to set the email address
bool Person::setEmail(string Email)
{
      // call a helper method to check the validity of the email
      // address (if it's in the form x@y.z).
      // Ideally, the email address should be a class on its own.
      if (isValid(Email) == true)
      {
            Email_ = Email;
            return true;
      } else
            return false;
}
```

Βλέπουμε πώς ελέγχεται πλέον η πρόσβαση στις μεταβλητές. Η μεταβλητή που κρατά τη διεύθυνση email του ατόμου, για παράδειγμα, αλλάζει μόνο αν έχουμε δώσει μια έγκυρη διεύθυνση email (της μορφής x@y.z).

Με τον ίδιο τρόπο που περιορίζουμε την πρόσβαση σε μεταβλητές μπορούμε να περιορίσουμε την πρόσβαση και σε μεθόδους. Θα μπορούσαμε π.χ. να έχουμε μια μέθοδο που να ελέγχει αν ο αριθμός τηλεφώνου του ατόμου είναι έγκυρος, πραγματοποιώντας αναζήτηση σε κάποια βάση δεδομένων. Οπωσδήποτε, μια τέτοια μέθοδος δεν θα θέλαμε να είναι προσβάσιμη από οποιονδήποτε εκτός της κλάσης, παρά μόνο σε άλλες μεθόδους της ίδιας της κλάσης (π.χ. στην μέθοδο `setTelephone()`).

Inheritance

Η κληρονομικότητα είναι ένα ακόμη χαρακτηριστικό του αντικειμενοστραφούς προγραμματισμού. Πρακτικά, μια κλάση κληρονομεί τα χαρακτηριστικά μιας υπάρχουσας κλάσης και προσθέτει καινούρια ή τροποποιεί τα ήδη υπάρχοντα.

Για καλύτερη κατανόηση της έννοιας, θα χρησιμοποιήσουμε το παράδειγμα της κλάσης `Person` της προηγούμενης ενότητας. Η κλάση `Person` ορίζει χαρακτηριστικά που περιγράφουν ένα άτομο αλλά δεν προβλέπει επιπλέον πληροφορίες, όπως π.χ. το φύλο, τί δουλειά κάνει το άτομο και τη διεύθυνση εργασίας του, αν έχει κινητό τηλέφωνο, αν είναι παντρεμένος/η με παιδιά κλπ. Το πρόβλημα είναι ότι δεν μπορούμε να προβλέψουμε όλες τις πιθανές πληροφορίες και να τις εισάγουμε στην κλάση `Person` γιατί οι περισσότερες θα έμεναν αναπάντητες και κάτι τέτοιο θα οδηγούσε σε σπατάλη χώρου (αφού θα έπρεπε να καταχωρήσουμε όλες τις πληροφορίες που θα ήταν κενές). Επιπλέον, θα ήταν εκνευριστικό και κουραστικό για το χρήστη να πρέπει να εισάγει άχρηστες πληροφορίες. Αν κάποιος δεν εργάζεται δεν έχει νοήμα να του ζητάμε το όνομα της ειδικότητάς του και διεύθυνση/τηλέφωνο της εργασίας του, σωστά;

Κάτι πολύ πιο χρήσιμο είναι να έχουμε μια κοινή βάση και να κρατάμε επιπλέον πληροφορίες μόνο όταν τις χρειαζόμαστε. Έστω ότι η κοινή βάση είναι η κλάση `Person` και θέλουμε να μελετήσουμε τις περιπτώσεις να είναι κάποιος υπάλληλος (Clerk) ή δάσκαλος (Teacher). Και οι δύο κατηγορίες ατόμων μοιράζονται κοινά χαρακτηριστικά που θεωρούμε ότι περιέχονται στην κλάση `Person`. Μπορούμε δηλαδή να ορίσουμε δύο νέες κλάσεις που θα κληρονομούν την κλάση `Person`. Η δήλωση της κληρονομικότητας μιας κλάσης γίνεται ως εξής (ορίζουμε ταυτόχρονα και την πρόσβαση στις μεταβλητές

και τον δημιουργό της κλάσης):

```
class Clerk : public Person
{
private:
      string JobTitle_;
      string CompanyName_;
      string JobAddress_;
      string JobEmail_;
      string JobTel_;
      string JobFax_;
      string JobDescription_;
public:
      Clerk(string fname, string lname, int age, string tel,
                        string email, string jobtitle, string companyname,
                        string jobaddress, string jobemail,
                        string jobtel, string jobfax,
                        string jobdescription)
      {
            Firstname_ = fname;
            Lastname_ = lname;
            Age_ = age;
            Telephone_ = tel;
            Email_ = email;
            JobTitle_ = jobtitle;
            CompanyName_ = companyname;
            JobAddress_ = jobaddress;
            JobEmail_ = jobemail;
            JobTel_ = jobtel;
            JobFax = jobfax;
            JobDescription_ = jobdescription;
      }
      // ακολουθούν οι μέθοδοι get/set για κάθε μεταβλητή με τους
      // απαραίτητους ελέγχους...
      ...

      // η ακόλουθη μέθοδος δίνει συνοπτικές πληροφορίες για τον
      // υπάλληλο.
      string getInfo() {
            return (getFirstname()+" "+getLastname()
                              +" works at "+CompanyName_
                              +", at "+JobAddress_
                              +".\n Email: "+getEmail()+"\n"
                              +"Tel: "+JobTel_);
      }
}
```

Αντίστοιχα, ορίζουμε την κλάση `Teacher`:

```
class Teacher : public Person
{
private:
      string Title_;
      string School_;
      string SchoolAddress_;
      string SchoolTel_;
      string CourseName_;
      string CourseDescription_;
public:
      Teacher(string fname, string lname, int age, string tel,
                        string email, string title, string school,
                        string schooladdress, string schooltel,
```

```
                        string coursename, string coursedescription)
      {
            Firstname_ = fname;
            Lastname_ = lname;
            Age_ = age;
            Telephone_ = tel;
            Email_ = email;
            Title_ = title;
            School_ = school;
            SchoolAddress_ = schooladdress;
            SchoolTel_ = jobtel;
            CourseName_ = coursename;
            CourseDescription_ = coursedescription;
      }
      // ακολουθούν οι μέθοδοι get/set για κάθε μεταβλητή με τους
      // απαραίτητους ελέγχους...
      ...

      // Η ακόλουθη μέθοδος δίνει συνοπτικές πληροφορίες για τον
      // καθηγητή.
      string getInfo() {
            return (getFirstname()+" "+getLastname()
                        +" teaches "+CourseName_+" at "+School_
                        +", "+SchoolAddress_+".\n"
                        +"Email: "+getEmail()+"\n"
                        +"Tel: "+SchoolTel_);
      }
}
```

Προσέξτε ότι χρησιμοποιήσαμε τις μεθόδους `get()` της κλάσης `Person` για την πρόσβαση στις μεταβλητές της (εφόσον είναι δηλωμένες `private`). Θυμίζουμε ότι αν μια μεταβλητή ή μέθοδος είναι δηλωμένη `private`, είναι προσβάσιμη *μόνο* από μεθόδους της ίδιας της κλάσης. Δεν είναι προσβάσιμη από τις μεθόδους μιας υποκλάσης αυτής.

Πώς όμως φαίνεται η χρησιμότητα αυτής της κατασκευής; Δείτε το ακόλουθο παράδειγμα κώδικα όπου χρησιμοποιούμε και τις τρεις κλάσεις:

```
Person bilbo("Bilbo", "Baggins", 111, "+306970123456",
                  "bilbobaggins@theshire.net");
Clerk sam( "Samwise", "Gamgee", 33, "+30697654321",
                  "samgamgee@theshire.net",
                  "Gardener", "Baggins Inc.",
                  "Bag End, Hobbiton, The Shire",
                  "gardener@baggins.com",
                  "+302103456789", "+302103456780",
                  "Garden Dept. Director");
Teacher pippin( "Peregrin", "Took", 27, "+30690090090",
                  "pippin@theshire.net", "Dr.",
                  "King's College", "Hobbiton",
                  "+30210000001", "Philosophy",
                  "Deal with the important matters of life, eg. what do we
eat?");
```

Μπορούμε πολύ εύκολα να χρησιμοποιήσουμε για κάθε ένα από αυτά τα αντικείμενα, πέρα από τις μεθόδους της κλάσης στην οποία ανήκει, και τις μεθόδους της γονικής κλάσης (δηλαδή την κλάση της οποίας τα χαρακτηριστικά κληρονομεί):

```
cout << "bilbo has email address: " << bilbo.getEmail()) << endl;
```

αυτή η εντολή θα τυπώσει:

```
bilbo has email address: bilbobaggins@shire.net
```

Ενώ η εντολή:

```
cout  << "sam works as a " << sam.getJobTitle() << " at "
      << sam.getCompanyName()) << endl;
```

θα τυπώσει:

```
sam works as a Gardener at Baggins Inc.
```

Παράλληλα, η εντολή:

```
cout  << "pippin teaches " << pippin.getCourseName() << " at "
      << pippin.getSchool()) << endl;
```

θα τυπώσει:

```
pippin teaches Philosophy at King's College
```

Τέλος, οι εντολές:

```
cout << "sam's private telephone is " << sam.getTel() << endl;
cout << "pippin is " << pippin.getAge() << " years old" << endl;
```

θα τυπώσουν:

```
sam's private telephone is +30697654321
pippin is 27 years old
```

Καλέσαμε δηλαδή μεθόδους της κλάσης `Person` και από τα τρία αντικείμενα!!! Αυτή είναι η ουσία της κληρονομικότητας των κλάσεων! Κάτι τέτοιο μας επιτρέπει να επαναχρησιμοποιούμε κώδικα που έχουμε γράψει παλαιότερα, απλώς επεκτείνοντάς τον όπως μας βολεύει κάθε φορά (*code reusability*). Έχοντας δηλαδή μερικές κλάσεις με ορισμένα μόνο τα βασικά χαρακτηριστικά, μπορούμε αναλόγως το πρόγραμμα που πρέπει να υλοποιήσουμε να προσθέσουμε ή να τροποποιήσουμε χαρακτηριστικά κατά βούληση, ώστε να επιτύχουμε το επιθυμητό αποτέλεσμα.

Polymorphism και Virtual Methods

Τί γίνεται όμως αν θέλουμε να αλλάξουμε τη λειτουργία μιας μεθόδου στην νέα κλάση που υπάρχει και στην παλιά; Στο προηγούμενο παράδειγμα, η νέες κλάσεις ορίζουν μια μέθοδο την `getInfo()`, η οποία επιστρέφει πληροφορίες για τον υπάλληλο ή τον δάσκαλο αντίστοιχα. Όπως είναι δηλωμένη, η κάθε κλάση προσφέρει τη δική της εκδοχή αλλά η αρχική `Person`, δεν έχει ορισμένη μια τέτοια μέθοδο `getInfo()`.

Αν η `getInfo()` οριστεί και για την κλάση `Person`, π.χ. ως εξής:

```
string Person::getInfo() {
      return (getFirstname()+" "+getLastname()
                  +"is "+getAge()+" years old");
}
```

τότε δημιουργείται το εξής ερώτημα: ποια εκδοχή της `getInfo()` θα καλείται για κάθε αντικείμενο;

Στη συγκεκριμένη περίπτωση θα είναι αδύνατη η μεταγλώττιση των κλάσεων `Clerk` και `Teacher`. Η C++ (αντίθετα με τη Java), δεν επιτρέπει το override των μεθόδων χωρίς ειδική άδεια. Η άδεια αυτή δίνεται με τη λέξη `virtual`, δηλαδή η μέθοδος ορίζεται ως δυνητική, και ο ορισμός της θα μετατραπεί ως εξής:

```
virtual string getInfo() {
      return (getFirstname()+" "+getLastname()
                  +"is "+getAge()+" years old");
}
```

Με την αλλαγή αυτή, θα καλείται κάθε φορά η `getInfo()` της κλάσης που ανήκει το αντικείμενο από το οποίο καλούμε την μέθοδο.

Για παράδειγμα, ο ακόλουθος κώδικας:

```
cout << bilbo.getInfo() << endl;
cout << sam.getInfo() << endl;
cout << pippin.getInfo() << endl;
```

θα παράγει

```
Bilbo Baggins is 111 years old
Samwise Gamgee works at Baggins Inc., at Bag End, Hobbiton, The Shire.
Email: gardener@baggins.com
Tel: +302103456789
Peregrin Took teaches Philosophy at King's College, Hobbiton.
Email: pippin@theshire.net
Tel: +30210000001
```

Τι ακριβώς έγινε εδώ; Καλέσαμε την ίδια μέθοδο και για τα τρία αντικείμενα (την `getInfo()`) η οποία όμως ορίζεται και στις τρεις κλάσεις. Η γονική κλάση `Person` ορίζει την `getInfo()` με έναν απλό τρόπο ("`Bilbo Baggins is 111 years old`") και η ίδια θα χρησιμοποιούνταν στις κλάσεις `Clerk` και `Teacher` αν δεν ορίζονταν και εκεί. Δηλαδή, αν δεν ορίζαμε την `getInfo()` στην κλάση `Clerk`, η `getInfo()` για το αντικείμενο `sam` θα επέστρεφε "`Samwise Gamgee is 33 year old`". Επειδή όμως η κλάση `Clerk` περιέχει περισσότερες πληροφορίες που πρέπει να απεικονιστούν με την `getInfo()`, η τελευταία επαναορίστηκε, αποκτώντας νέα λειτουργία.

Η τεχνική αυτή καλείται *Method Overriding* και είναι το θεμελιώδες χαρακτηριστικό του πολυμορφισμού. Η κλάσεις στις οποίες ορίζουμε virtual methods λέγονται virtual (virtual classes). Σημειώνουμε και πάλι ότι είναι απαραίτητη η χρήση της λέξης virtual στον ορισμό της μεθόδου.

Ο πολυμορφισμός μας δίνει τη δυνατότητα δημιουργίας αφηρημένων εννοιών/κλάσεων/μεθόδων στις οποίες θα κολλάμε κάθε φορά τη σωστή υλοποίηση αναλόγως τις ανάγκες μας.

Το ακόλουθο παράδειγμα έχει σκοπό να δείξει πώς ακριβώς χρησιμοποιούμε τον πολυμορφισμό. Θεωρούμε τα αντικείμενα που έχουμε ορίσει πριν (`bilbo`, `sam`, `pippin`).

```
Person* who[3];
who[0] = &bilbo;
who[1] = &sam;
who[2] = &pippin;
for (int i=0; i < who.length; i++)
      cout << who[i]->getInfo() << endl;
```

Το παραπάνω παράδειγμα θα τυπώσει ό,τι και το προηγούμενο μόνο που τώρα χρησιμοποιήσαμε έναν πίνακα αντικειμένων `Person` για να οργανώσουμε την πληροφορία. Αλλά το αντικείμενο `sam` είναι τύπου `Clerk` και το `pippin` ανήκει στην κλάση `Teacher`. Πώς τα καταχωρήσαμε στον πίνακα `who`; Η απάντηση βρίσκεται στην σχέση που έχουν οι κλάσεις `Person`, `Clerk` και `Teacher` μεταξύ τους. Ένα αντικείμενο `Clerk` είναι ταυτόχρονα και αντικείμενο `Person`, όπως και ένα αντικείμενο `Teacher` είναι επίσης `Person`. Ένας `Clerk` δεν είναι όμως ταυτόχρονα `Teacher`. Έτσι μπορούμε να τα

αντιμετωπίζουμε κατά βούληση ως `Clerk` και `Teacher` ή `Person`, αναλόγως τις ανάγκες μας. Ακόμη όμως και αν τα προσπελάσουμε ως `Person`, η `getInfo()` του καθενός θα είναι αυτή που ορίζει η κλάση του.

Τί γίνεται όμως αν χρειάζεται να καλέσουμε και την γονική μέθοδο (δηλαδή αυτή που επαναορίζουμε στη νέα κλάση); Στην περίπτωση αυτή χρησιμοποιούμε την αναφορά στη γονική κλάση καλώντας τον δημιουργό της. Ένα παράδειγμα είναι αναγκαίο για καλύτερη κατανόηση. Μετασχηματίζουμε το δημιουργό και τη μέθοδο `getInfo()` της κλάσης `Teacher` στα εξής:

```
Teacher(string fname, string lname, int age, string tel,
                  string email, string title, string school,
                  string schooladdress, string schooltel,
                  string coursename, string coursedescription)
{
      Person(fname, lname, age, tel, email);
      Title_ = title;
      School_ = school;
      SchoolAddress_ = schooladdress;
      SchoolTel_ = jobtel;
      CourseName_ = coursename;
      CourseDescription_ = coursedescription;
}
string getInfo() {
      return (Person.getInfo()
                  +" and teaches "+CourseName_+" at "+School_
                  +", "+SchoolAddress_+".\n"
                  +"Email: "+getEmail()+"\n"
                  +"Tel: "+SchoolTel_);
}
```

Με αυτόν τον τρόπο στον δημιουργό αρκεί πλέον να αρχικοποιούμε (initialize) μόνο τα χαρακτηριστικά που είναι νέα στην κλάση χωρίς να κάνουμε διπλό κόπο που έχει ήδη γίνει στην γονική κλάση. Επίσης, στην `getInfo()`, χρησιμοποιούμε την πληροφορία που επιστρέφεται από τη γονική κλάση και συμπληρώνουμε με τα νέα στοιχεία. Αν π.χ. αλλάζαμε το κείμενο που επιστρέφει η `getInfo()` της κλάσης `Person`, θα άλλαζε αυτόματα και το κείμενο που επέστρεφε η `getInfo()` της κλάσης `Teacher`, εφόσον χρησιμοποιούμε την αναφορά στη κλάση `Person`..

Σημειώνουμε ότι method overriding έχουμε μόνο όταν η δήλωση της μεθόδου είναι η ίδια στη γονική και στην θυγατρική κλάση. Δηλαδή ίδιο όνομα, επιστρεφόμενα δεδομένα και παραμέτρους. Αν ένα από αυτά είναι διαφορετικά (εκτός από το όνομα φυσικά) τότε έχουμε υπερφόρτωση μεθόδου (method overloading) που το έχουμε ήδη αναλύσει σε προηγούμενη ενότητα.

Pure Virtual Classes

Μερικές φορές μπορεί να θέλουμε να ορίσουμε απλώς μια γενική δομή κάποιων κλάσεων, χωρίς όμως να προσφέρουμε μια ακριβή υλοποίηση. Δηλαδή, να ορίσουμε μια "αφηρημένη" κλάση, που να παρέχει απλώς ένα πλαίσιο που θα συμπληρώνουν οι θυγατρικές κλάσεις μέσω της τεχνικής method overriding. Αυτό σημαίνει ότι στον ορισμό της κλάσης απλώς θα δηλώνονται ορισμένες μέθοδοι ως "αφηρημένες" αλλά δεν θα παρέχεται κάποιος ορισμός γι' αυτές. Στη C++ οι κλάσεις αυτές καλούνται pure virtual ενώ στη Java καλούνται abstract.

Ένα κλασσικό παράδειγμα αφηρημένης κλάσης είναι η κλάση του αυτοκινήτου. Ο μηχανισμός που αλλάζει ταχύτητες σε κάθε αυτοκίνητο είναι διαφορετικός και διαφέρει

ανάμεσα στους κατασκευαστές και στα διάφορα μοντέλα. Όμως ο τρόπος αλληλεπίδρασης (δηλαδή μέσω του μοχλού ταχυτήτων) είναι καλά ορισμένος έστω και με κάποια περιθώρια παραλλαγών. Ο τρόπος ορισμού της κλάσης γίνεται ως εξής:

```
class Car {
      ...
      // Αλλαγή της ταχύτητας. Επιστρέφει true αν η αλλαγή ήταν επιτυχής
      // ή false αν ήταν ανεπιτυχής (π.χ. από 5 σε όπισθεν).
      virtual bool change_gear(int new_gear) = 0;
      ...
}

class FerrariModena extends Car {
      ...
      bool change_gear(int new_gear) {
            // Η συγκεκριμένη υλοποίηση βρίσκεται εδώ
      }
      ...
}
```

8.Αρχεία και Streams

Τα αρχεία σε κάθε υπολογιστή δεν είναι παρά ακολουθίες χαρακτήρων. Αυτό που αλλάζει στον προγραμματισμό είναι ο τρόπος χειρισμού τους και αυτός με τη σειρά του εξαρτάται από τη γλώσσα προγραμματισμού, οι βιβλιοθήκες που χρησιμοποιούμε καθώς και από το ίδιο το λειτουργικό σύστημα. Π.χ. στο BeOS τα αρχεία θεωρούνται αντικείμενα που τα ίδια προσφέρουν τις μεθόδους με τις οποίες θα τα προσπελάσουμε.

Παραδοσιακά, η C προσφέρει ορισμένες δομές δεδομένων `FILE` (struct, ο πρόγονος της κλάσης) και ανεξάρτητες συναρτήσεις που έχουν πρόσβαση στα αρχεία. Η όλη διαδικασία του προγραμματισμού γίνεται με pointers στις δομές `FILE` και είναι αρκετά εύκολο να γίνει κάποιο λάθος κατά τη χρήση τους. Επιπλέον δε προσφέρουν καμία από τις ευκολίες του αντικειμενοστραφούς προγραμματισμού (κληρονομικότητα, πολυμορφισμό, κλπ). Ήταν αναγκαίος κάποιος εναλλακτικός προγραμματισμός αρχείων, χρησιμοποιώνας αντικείμενα.

Για το σκοπό αυτό η C++, υιοθέτησε τη χρήση των streams. Έτσι δεν παρέχεται απευθείας πρόσβαση των δεδομένων των αρχείων, παρά μόνο ως ανταλλαγή bytes από το πρόγραμμα προς το αρχείο και αντίστροφα. Η χρήση των streams ως ροές δεδομένων από και προς το αρχείο, έχει το άμεσο όφελος ότι δεν είναι πλέον σημαντικό το μέσον στο οποίο βρίσκεται το αρχείο αλλά μόνο η εναλλαγή της πληροφορίας από και πρός το πρόγραμμα.

Οι κλάσεις οι οποίες μας ενδιαφέρουν σε αυτό το σημείο είναι η `fstream` και οι πιο συγκεκριμένες `ifstream`, `ofstream`. Όπως ίσως έχετε καταλάβει, η `ifstream` αναφέρεται στο άνοιγμα αρχείου μόνο για ανάγνωση (input file stream), ενώ η `ofstream` χρησιμοποιείται μόνο για εγγραφή στο αρχείο (output file stream). Η `fstream` προσφέρει ταυτόχρονα δυνατότητα για ανάγνωση και εγγραφή.

Η κλάση fstream

Η κλάση `fstream` (που δηλώνεται στο αντίστοιχο αρχείο κεφαλίδας), προσφέρει αρκετές μεθόδους για ανάγνωση, εγγραφή και μετακίνηση στο αρχείο καθώς και αρκετά flags για τροποποίηση των δεδομένων κατά την εγγραφή ή την ανάγνωση. Ακολουθούν μερικές από τις σημαντικότερες μέθοδοι:

Όνομα	*Λειτουργία*
`bool good()` `bool bad()`	Η `good()` επιστρέφει `true` αν το αρχείο είναι σε καλή κατάσταση (αν δεν έχει συμβεί κάποιο σφάλμα εισόδου/εξόδου) ή `false` διαφορετικά. Η `bad()` έχει την αντίθετη συμπεριφορά.
`bool eof()`	Επιστρέφει `true` αν έχουμε φτάσει στο τέλος του αρχείου κατά την ανάγνωση.
`void flush()`	Αποθηκεύει όλες τις αλλαγές που έχουν γίνει στο αρχείο και δεν βρίσκονται στους buffers του συστήματος.
`int get()`	Επιστρέφει τον επόμενο χαρακτήρα του αρχείου.
`istream &getline(char *buf, int num)`	Αντιγράφει από το αρχείο στο πίνακα χαρακτήρων `buf`, το μέγιστο `num` χαρακτήρες εκτός αν φτάσει στο τέλος της γραμμής ή του αρχείου. Επιστρέφει μια αναφορά στο αρχείο.

Όνομα	*Λειτουργία*
`int gcount()`	Επιστρέφει τον αριθμό των χαρακτήρων που αναγνώστηκαν κατά την προηγούμενη εκτέλεση των μεθόδων `get()`, `read()`.
`istream &read(char *buf, int num)`	Αντιγράφει από το αρχείο στο πίνακα χαρακτήρων buf, το μέγιστο `num` χαρακτήρες εκτός αν φτάσει στο τέλος του αρχείου.Επιστρέφει μια αναφορά στο αρχείο.
`ostream &write(char *buf, int num)`	Αντιγράφει από το πίνακα χαρακτήρων `buf` στο αρχείο, το μέγιστο `num` χαρακτήρες. Επιστρέφει μια αναφορά στο αρχείο.
`void seekg(int offset, origin)` `void seekp(int offset, origin)`	Ορίζει τη τρέχουσα θέση του δρομέα για ανάγνωση (`seekg`) ή για εγγραφή (`seekp`). Τα streams στη C++ χρησιμοποιούν δύο διαφορετικούς δείκτες, ένα για ανάγνωση δεδομένων και ένα για εγγραφή.
`int tellg()` `int tellp()`	Επιστρέφει την αντίστοιχη θέση του δρομέα για ανάγνωση (`tellg`) ή για εγγραφή (`tellp`).

Για την καλύτερη κατανόηση των μεθόδων της `fstream` παραθέτουμε ένα απλό παράδειγμα ενός προγράμματος αντιγραφής δύο αρχείων, σε δύο εκδοχές: μία στην οποία η αντιγραφή γίνεται ανά χαρακτήρα (byte) και μία δεύτερη όπου αντιγράφονται blocks των 256KB κάθε φορά.

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

using namespace std;

int main(int argc, char *argv[]) {
      // Έλεγχος για το αν το πρόγραμμα δέχεται 3 παραμέτρους.
      // Σημειώνουμε ότι στη C++ η πρώτη παράμετρος argv[0] είναι
      // πάντα το ίδιο το εκτελέσιμο πρόγραμμα.
      if (argc != 3) {
            cout << "Usage: CopyFile <from> <to>" << endl;
            return 0;
      }

      // Δοκιμάζουμε να ανοίξουμε το αρχείο <from> για ανάγνωση (παράμετρος
      // argv[1]).
      // Η διαδικασία θα αποτύχει αν το αρχείο δεν υπάρχει ή αν δεν έχουμε
      // πρόσβαση σε αυτό.
      ifstream fin(argv[1]);
      if (fin == 0) {
            cout << "Error: Input file cannot be opened for reading!" << endl;
            return 10;
      }

      // Δοκιμάζουμε να ανοίξουμε το αρχείο <to> για εγγραφή (παράμετρος
      // argv[2]).
      // Η διαδικασία θα αποτύχει αν δεν έχουμε πρόσβαση ή αν δεν υπάρχει
      // χώρος στο δίσκο.
      ofstream fout(argv[2]);
      if (fout == 0) {
```

```
            cout << "Error: Output file cannot be opened for writing!" << endl;
            return 10;
      }

      // Για να μάθουμε το τέλος του αρχείου, χρησιμοποιούμε τις μεθόδους
      // seekg() και tellg(). Η seekg() αλλάζει την τρέχουσα θέση του αρχείου,
      // ενώ η tellg() επιστρέφει αυτή τη θέση.
      // Πρακτικά αυτό που κάνουμε είναι να πάμε το δρομέα (cursor) στη θέση που
      // έχει offset 0 από το τέλος του αρχείου (ios::end) και να διαβάσουμε τη
      // θέση που επιστρέφει η tellg(). Αυτό είναι και το μέγεθος του αρχείου.
      fin.seekg(0, ios::end);
      size_t finsize = fin.tellg();
      cout << "Input file size: " << finsize << endl;

      // Επειδή θέλουμε να αντιγράψουμε το αρχείο, επιστρέφουμε τον δρομέα στην
      // αρχή του αρχείου, offset 0 bytes από τη θέση ios::beg).
      fin.seekg(0, ios::beg);

      // Θα αντιγράψουμε τώρα τα περιεχόμενα του αρχείου fin στο fout.
      // Η αντιγραφή θα γίνει byte προς byte και θα τυπώνουμε την πρόοδο

      int c;
      int percent = 0;
      while (fin.eof() == false) {  // έχουμε φτάσει στο τέλος του αρχείου;
            c = fin.get();          // διάβασε ένα byte
            if (fin.eof() == false) // αν δεν έχουμε φτάσει στο τέλος
                  fout.put(c);      // γράψε το byte
            cout << "Copy Completed : " << 100*percent/finsize << "\r";
            percent++;
      }
      fin.close();
      fout.close();
      return 0;
}
```

Η αντιγραφή ενός αρχείο περίπου 40MB διήρκεσε 20 δευτερόλεπτα. Όχι ιδιαίτερα γρήγορο. Με μια μικρή αλλαγή στον κώδικα θα δείξουμε πώς η ίδια διαδικασία μπορεί να διαρκέσει μόλις 4 δευτερόλεπτα στον ίδιο υπολογιστή και για το ίδιο αρχείο! Συγκεκριμένα θα αλλάξουμε τον κωδικά του βρόχου while:

```
      // Θα αντιγράψουμε τώρα τα περιεχόμενα του αρχείου fin στο fout.
      // Η αντιγραφή θα γίνει ανά block και θα τυπώνουμε την πρόοδο

      size_t bufsize = 262144;           // το μέγεθος του block
      char buf[bufsize];                 // ορίζουμε το buffer από bytes
      int count;                         // ο μετρητής των bytes που γράφονται
      int total = 0;
      while (fin.eof() == false) {       // έχουμε φτάσει στο τέλος του αρχείου;
            fin.read(buf, bufsize);      // διάβασε το πολύ bufsize bytes
            count = fin.gcount();        // μέτρα πόσα πραγματικά διαβάστηκαν
            if (count)                   // αν έχει διαβαστεί έστω και 1 byte
                  fout.write(buf,count); // γράψε τα στο αρχείο fout
            cout << "Copy Completed : " << 100*total/finsize << "\r";
            total += count;
      }
```

Οι τελεστές <<, >>

Πέρα από τις μεθόδους, η `fstream` προσφέρει έναν πολύ εύχρηστο τρόπο ανταλλαγής δεδομένων με το αρχείο, μέσω των τελεστών << και >> που δηλώνουν την κατεύθυνση

από (>>) και προς (<<) το αρχείο. Τους τελεστές τους έχουμε ήδη χρησιμοποιήσει αρκετές φορές με την πρότυπη έξοδο (κονσόλα) `cout`. Αυτό ισχύει γιατί και η `cout` είναι ένα προκαθορισμένο αντικείμενο `ostream` (η γονική κλάση της `ofstream`) που αντιστοιχεί στην πρότυπη έξοδο (`stdout`). Αντίστοιχα η πρότυπη είσοδος `cin` είναι ένα `istream` (η γονική κλάση της `ifstream`) που δίνει πρόσβαση στην `stdin`, και η πρότυπη έξοδος λαθών `cerr` που αντιστοιχεί στην `stderr`.

Ο τρόπος χρήσης των τελεστών είναι πολύ απλός. Έστω `fin`, `fout` δύο αντικείμενα `ifstream` και `ofstream` αντίστοιχα:

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

using namespace std;

int main() {
	ifstream fin("data.txt");
	if (fin == 0) {
		cout << "Error: could not open file data.txt" << endl;
		return 10;
	}

	ofstream fout("output.txt");
	if (fout == 0) {
		cout << "Error: could not open file output.txt" << endl;
		return 10;
	}

	// Δήλωση των μεταβλητών που θα χρησιμοποιηθούν.
	string name;
	double age;

	// όσο δεν έχουμε φτάσει στο τέλος του αρχείου
	while (fin.eof() == false) {
		// διάβασε κάθε στήλη στην αντίστοιχη μεταβλητή
		fin >> name >> age;
		if (fin.eof() == false) {
			// τύπωσε τα στην κονσόλα
			cout << "Ο " << name << " είναι " << age << " ετών." << endl;

			// και γράψε το ίδιο κείμενο στο αρχείο output.txt
			fout << "Ο " << name << " είναι " << age << " ετών." << endl;
		}
	}
}
```

Αν το αρχείο `data.txt` έχει π.χ. τα εξής δεδομένα:

```
Κώστας 29.7
Νίκος 34.4
Γιαννάκης 6.5
Μαθουσάλας 803.4
```

Το αποτέλεσμα του προγράμματος θα αποθηκευτεί στο αρχείο `output.txt` και θα είναι:

```
Ο Κώστας είναι 29.7 ετών.
Ο Νίκος είναι 34.4 ετών.
Ο Γιαννάκης είναι 6.5 ετών.
Ο Μαθουσάλας είναι 803.4 ετών.
```

Stringstreams

Στη γλώσσα C, όπως ίσως γνωρίζετε, η μορφοποίηση ενός string μέσω παραμέτρων γίνεται με την εντολή `sprintf`, που λειτουργεί με τον ίδιο τρόπο όπως η `printf` για την πρότυπη έξοδο (`stdout`) ή η `fprintf` για την έξοδο σε αρχείο. Αντίστοιχα, η λειτουργία των streams και η ευκολία των τελεστών `<<` και `>>` παρέχεται στη C++ και για strings. Για το σκοπό αυτό έχει υλοποιηθεί η κλάση `stringstream` (στο αρχείο κεφαλίδας `sstream`), και χρησιμοποιείται όπως ακριβώς ένα `fstream`. Παραθέτουμε το ακόλουθο παράδειγμα για καλύτερη κατανόηση:

```
#include <string>
#include <sstream>
#include <iostream>

using namespace std;

int main() {
	stringstream formatted;

	// Δήλωση των μεταβλητών που θα χρησιμοποιηθούν.
	int data[] = {10, 5, 4, 3, 8, 11};
	string names[] = {"one", "two", "three", "four", "five", "six"};

	for (int i=0; i < 6; i++) {
		formatted << "Name: " << names[i] << ", value: " << data[i] << endl;
	}
	cout << formatted.str();
}
```

Το αποτέλεσμα του προγράμματος αυτού είναι το εξής:

```
Name: one, value: 10
Name: two, value: 5
Name: three, value: 4
Name: four, value: 3
Name: five, value: 8
Name: six, value: 11
```

Σημειώνουμε ότι αυτό τυπώνεται μόνο από την τελευταία εντολή:

```
	cout << formatted.str();
```

Η μέθοδος `str()` του `stringstream` "παγώνει" τα περιεχόμενά του και επιστρέφει ένα αντικείμενο `string` το οποίο μπορούμε να τυπώσουμε. Δεν είναι αποδεκτό να τυπώσουμε ένα αντικείμενο `stringstream` απευθείας.

9.Τα πρότυπα (templates)

Τα πρότυπα (templates) είναι ένα από τα πιο δυνατά χαρακτηριστικά της C++, το οποίο αποτελεί τη βάση της STL(την οποία θα δούμε παρακάτω), καθώς επιτρέπουν τη χρήση πολυμορφισμού κατά τη μεταγλώττιση (compile-time) αντί κατά την εκτέλεση (run-time), γεγονός που αυξάνει κατά πολύ την απόδοσή τους, αφού ο μεταγλωττιστής μπορεί να πραγματοποιήσει αρκετές βελτιστοποιήσεις κατά την μεταγλώττιση.

Πρότυπα μπορούν να οριστούν για συναρτήσεις ή για κλάσσεις. Η λειτουργία είναι ή ίδια, με τη διαφορά ότι αν οριστεί μια κλάση πρότυπο, η προτυποποίηση ισχύει για κάθε μέθοδο/συνάρτηση της κλάσης.

Ποιον ακριβώς σκοπό όμως εξυπηρετούν τα πρότυπα; Αν σε κάποιο πρόγραμμα χρειαζόμαστε την ταξινόμηση δύο διαφορετικών συνόλων δεδομένων (π.χ. ένα πίνακα από `int` και ένα πίνακα από `string`) θα χρειαστούμε δύο συναρτήσεις που θα πραγματοποιήσουν την ταξινόμηση στους πίνακες. Κάτι τέτοιο είναι περιττό, επιρρεπές σε λάθη και αιτία αύξησης του μεγέθους του προγράμματος. Αν το πρόγραμμα εξελιχθεί να υποστηρίζει και άλλους τύπους δεδομένων, ακόμη και κλάσεις, θα χρειαστούμε μια συνάρτηση ταξινόμησης για κάθε τύπο! Οπωσδήποτε πρέπει να υπάρχει μια καλύτερη λύση.

Η λύση είναι να οριστεί ένα πλαίσιο, ή πρότυπο, το οποίο θα περιγράφει τον αλγόριθμο, χρησιμοποιώντας μια αφηρημένη κλάση, ο τύπος της οποίας δε μας ενδιαφέρει παρά μόνο ορισμένα χαρακτηριστικά που είναι απαραίτητα. Π.χ. για τη ταξινόμηση χρειαζόμαστε απλώς μια μέθοδο σύγκρισης ανάμεσα σε δύο τυχαία αντικείμενα.

Η δήλωση ενός τέτοιου πλαισίου γίνεται με την λέξη-κλειδί `template` ακολουθούμενη από το όνομα της αφηρημένης κλάσης κλεισμένης σε <>.

Για την καλύτερη κατανόηση των προτύπων ακολουθεί ένα απλό παράδειγμα μιας κλάσης λίστας αντικειμένων (αδιάφορο τί είδους):

```
#include <iostream>

using namespace std;

// Ορίζουμε την κλάση list ως template που θα χρησιμοποιεί
// την αφηρημένη κλάση data_t (δεν υπάρχει πραγματικά, απλώς
// υποδηλώνει μια οποιαδήποτε κλάση).

template<class data_t> class list {
      // το item θα είναι κάθε φορά τύπου data_t
      data_t item;

      // το next είναι δείκτης στο επόμενο αντικείμενο
      // της λίστας
      list *next;
public:
      // ο δημιουργός (δέχεται ένα αντικείμενο data_t)
      list(data_t d);

      // για να προσθέσουμε ένα αντικείμενο στη λίστα
      // χρησιμοποιούμε την add()
      void add(list *node) {
            node->next = this;
            return;
      }
```

```
	// η get_next() επιστρέφει το επόμενο αντικείμενο στη λίστα
	list *get_next() {
		return next;
	}

	// η get_data επιστρέφει το αντικείμενο item. Ο τύπος θα είναι
	// πάντα σωστός (π.χ. θα επιστρέφει char αν η data_t είναι
	// char, string, κλπ.
	data_t get_data() {
		return item;
	}
};

// Στον ορισμό της κλάσης, απλώς δηλώσαμε τον δημιουργό
// Με τον παρακάτω τρόπο τον ορίζουμε κιόλας. Σημειώστε
// τη χρήση της λέξης template και των brackets <data_t>
// στο όνομα της κλάσης
template<class data_t> list<data_t>::list(data_t d) {
	// ορισμός του item στην παράμετρό d
	item = d;
	// δεν έχουμε επόμενο αντικείμενο (πρώτο στοιχείο
	next = 0;
}

int main() {
	// Ορισμός ενός αρχικού αντικειμένου list.
	// Το ρόλο της αφηρημένης κλάσης data_t παίρνει ο τύπος char
	list<char> start('a');
	// Δηλώνουμε και δύο δείκτες (pointers) της ίδιας κλάσης
	list<char> *p, *last;

	// θα χρησιμοποιήσουμε τον δείκτη last για να μετακινούμαστε
	// στη λίστα, όσο μεγαλώνει
	last = &start;
	for (int i=1; i < 26; i++) {
		// δημιούργησε ένα καινούριο κόμβο της λίστας με
		// αντικείμενο το χαρακτήρα 'a' + i.
		// Η δημιουργία είναι δυναμική (προσέξτε τη χρήση της new.
		p = new list<char>('a' +i);

		// πρόσθεσε το νέο αντικείμενο στον προηγούμενο κόμβο.
		p->add(last);

		// ο προηγούμενος κόμβος δεν είναι πλέον τελευταίος.
		// όρισε τον τρέχοντα κόμβο να φαίνεται τελευταίος.
		last = p;
	}

	// ξεκίνα από την αρχή
	p = &start;
	while (p) {
		// όσο έχουμε κόμβους στη λίστα (δηλαδή όσο η get_next()
		// θα επιστρέφει μη μηδενικό αποτέλεσμα, τύπωσε το αντικείμενο
		// που περιέχεται στον κόμβο.
		cout << p->get_data() << ", ";

		// προχώρα στον επόμενο κόμβο.
		p = p->get_next();
	}
	cout << endl;
```

```
    return 0;
}
```

Το πρόγραμμα αυτό θα τυπώσει το εξής αποτέλεσμα.

```
a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z,
```

Σημειώνουμε και πάλι ότι στην κλάση `list`, επειδή αυτή είναι templated, μπορεί να χρησιμοποιηθεί και οποιαδήποτε άλλη κλάση εκτός της `char`.

Standard Template Library (STL)

Η βιβλιοθήκη STL είναι ένα σύνολο από έτοιμες κλάσεις και συναρτήσεις γενικής χρήσης που γλυτώνουν σημαντικό χρόνο από το προγραμματιστή καθώς προσφέρουν έναν εύκολο τρόπο να αντιμετωπιστούν τα περισσότερα θέματα γενικής φύσεως σε ένα πρόγραμμα. Έτσι ο προγραμματιστής έχει πλέον τη δυνατότητα να ασχοληθεί με την ουσία του προβλήματός του, παρά με το ποια ρουτίνα θα χρησιμοποιήσει για να ταξινομήσει ένα πίνακα από strings.

Γενικά η STL αποτελείται από τρεις κατηγορίες κλάσεων και μεθόδων: τους containers, τους αλγόριθμους και τους iterators. Αυτά συνδέονται μεταξύ τους με τέτοιο τρόπο ώστε να επιτρέπεται η εύκολη αντιμετώπιση μιας πληθώρας προβλημάτων των οποίων η επίλυση είναι απαραίτητη για τον προγραμματιστή αλλά είναι απλώς λεπτομέρειες στην συνολική εικόνα ενός προγράμματος (π.χ. ταξινόμηση στοιχείων).

Οι κλάσεις containers

Οι containers είναι αντικείμενα που χρησιμοποιούνται ως “δοχεία” για άλλα αντικείμενα (οποιουδήποτε είδους) και προσφέρουν συγκεκριμένα χαρακτηριστικά όσον αφορά το τρόπο πρόσβασης των αντικειμένων, τον τρόπο αποθήκευσης των αντικειμένων στη μνήμη, την αντιστοίχιση αντικειμένων σε “κλειδιά” (σχεσιακοί containers), κλπ. Οι βασικοί containers είναι οι `vector`, `list`, `queue`, `stack` και `set`, ενώ ορίζονται και οι σχεσιακοί containers οι `map`, `multimap`.

Οι containers επειδή είναι βασισμένοι στα πρότυπα, μπορούν να περιέχουν αντικείμενα οποιασδήποτε κλάσης.

Οι μέθοδοι algorithms

Οι containers από μόνοι τους δε θα ήταν ιδιαίτερα χρήσιμοι, όχι περισσότερο από την ξεχωριστή υλοποίηση κάποιου προγραμματιστή. Αυτό που κάνει την STL να αποδεικνύεται αξεπέραστο εργαλείο είναι ο συνδυασμός των containers με τους αλγόριθμους (algorithms). Οι αλγόριθμοι είναι απλώς μέθοδοι που έχουν σχεδιαστεί να λειτουργούν πάνω στα αντικείμενα που περιέχονται στους containers, χωρίς όμως να απαιτούν κάποιο συγκεκριμένο τύπο, μόνο κάποια γενικά χαρακτηριστικά. Π.χ. για να ταξινομήσουμε τα αντικείμενα ενός container, δεν είναι ανάγκη να απαιτήσουμε ο container να περιέχει string, απλώς να ορίζεται η πράξη της σύγκρισης για τα αντικείμενα αυτά (δηλαδή οι τελεστές `<`, `==`, `>`).

Η STL προσφέρει έτοιμους αλγόριθμους για πολλές διαδικασίες, όπως ταξινόμηση (`sort`, `stable_sort`), αντιγραφή αντικειμένων από τον container με ή χωρίς συνθήκη που πρέπει να πληροίται (`copy`, `copy_backward`, `unique_copy`, `unique_copy_if`), ορισμός κάποιας τιμής στα αντικείμενα (`fill`, `fill_n`), εύρεση αντικειμένων στο container (`find`, `find_end`, `find_first_of`, `find_if`), καταμέτρηση αντικειμένων με ή χωρίς συνθήκη (`count`, `count_if`), εκτέλεση κάποιας ρουτίνας για συγκεκριμένα αντικείμενα

(for_each, transform), πράξεις συνόλων (set_union, set_intersection), εύρεση μεγίστων και ελαχίστων (max_element, min_element), διαγραφή ή αντικατάσταση αντικειμένων στο container με ή χωρίς συνθήκη και με ή χωρίς αντιγραφή σε νέο container (remove, remove_if, remove_copy, remove_copy_if, replace, replace_if, replace_copy, replace_copy_if), και πολλοί άλλοι ακόμη.

Οι δείκτες iterators

Κάθε αλγόριθμος δέχεται το αρχικό και το τελικό αντικείμενο του container στο οποίο θα πραγματοποιηθεί η εργασία ως παραμέτρους. Τα αντικείμενα αυτά δίνονται ως iterators, το τελικό στοιχείο της STL για να ολοκληρωθεί ως εργαλείο. Πρακτικά οι iterators είναι pointers αλλά με κάποια ιδιαίτερη συμπεριφορά, π.χ. υπάρχουν forward και backward iterators για μετακίνηση στα αντικείμενα ενός container μόνο προς τη μία κατεύθυνση, input και output iterators για χρήση σε streams κλπ. Δε θα αναφερθούμε περισσότερο στην υλοποίηση τους ή σε προχωρημένη χρήση τους, το παράδειγμα που ακολουθεί χρησιμοποιεί τους πιο συχνούς iterators σε κάθε container, που υποδεικνύουν την αρχή και το τέλος του container (begin() και end()).

Ο container vector

Το παρόν κείμενο είναι απλώς εισαγωγικό και δεν αποσκοπεί σε εξαντλητική ανάλυση της STL. Θα παρουσιάσουμε μια από τις ενδεικτικές χρήσεις του container vector.

```
#include <string>
#include <vector>
#include <iostream>
#include <fstream>

// Δε θέλουμε να χρησιμοποιούμε το πρόθεμα std:: συνέχεια...
using namespace std;

// Ορίζουμε την κλάση Person με μόνο δύο μεταβλητές,
// το όνομα και την ηλικία. Λόγω απλότητας, δηλώνουμε τα πάντα
// ως public.

class Person {
public:
      // το όνομα είναι τύπου string, ενώ η ηλικία τύπου age
      string name_;
      int age_;

      // Ο constructor δεν κάνει τίποτε άλλο παρά να αντιγράψει τις
      // τιμές των παραμέτρων στις μεταβλητές του αντικειμένου
      Person(string n, int a) {
            name_ = n;
            age_  = a;
      }
};

// η main() ρουτίνα του προγράμματος
int main() {
      // Ορίζουμε ένα vector που θα περιέχει αντικείμενα Person
      // το οποίο θα παίξει το ρόλο της ατζέντας (addressbook)
      vector<Person> addressbook;

      // ορίζουμε βοηθητικές μεταβλητές input και age.
      // η const linesize είναι "σταθερή" μεταβλητή (constant)
      // και η τιμή της δεν επιτρέπεται να αλλάξει.
```

```
	const int linesize = 100;
	char input[linesize];
	int age;

	// διαβάζουμε 10 ζεύγη (όνομα, ηλικία)
	for (int i=0; i < 5; i++) {
		// η getline() διαβάζει μια ολόκληρη γραμμή κειμένου
		// και την αποθηκεύει στη μεταβλητή που δίνουμε,
		// συγκεκριμένα στην input.
		cout << "Enter name: ";
		cin.getline(input, linesize);

		// Διαβάζουμε τώρα την ηλικία. Δεν είναι ανάγκη η χρήση
		// της getline πάλι, αρκεί ο τελεστής >> του cin.
		cout << "Enter age: ";
		cin >> age;

		// Εδώ τώρα γίνονται δύο πράγματα:
		// * καταρχάς δημιουργούμε το αντικείμενο Person με
		// παραμέτρους τη γραμμή που μόλις διαβάσαμε (input) και
		// την ηλικία age. Η δημιουργία γίνεται με απλή κλήση του
		// constructor Person(input, age).
		// * Το αντικείμενο αυτό το αποθηκεύουμε απευθείας στην τελευταία
		// θέση του vector<Person> addressbook. Για το σκοπό αυτό
		// χρησιμοποιούμε τη μέθοδο push_back().
		addressbook.push_back(Person(input, age));

		// η γραμμή αυτή είναι απαραίτητη για να διαβάσει το χαρακτήρα
		// newline '\n', γιατί το cin >> age διαβάζει απλώς τον αριθμό
		// αγνοώντας την αλλαγή γραμμής. Αυτό έχει ως αποτέλεσμα
		// να μη λειτουργεί σωστά η επόμενη επανάληψη του loop. (Όσοι
		// είσασταν στο συγκεκριμένο μάθημα και δε μπορείτε να κοιμηθείτε
		// τα βράδια επειδή δε δούλεψε το πρόγραμμα μπορείτε να ησυχάσετε
		// πλέον...
		cin.getline(input, linesize);
	}

	// Έχουμε σκοπό να αποθηκεύσουμε τα αποτελέσματα μέσα σε ένα αρχείο
	ofstream fout("address.txt");
	if (fout == 0) {
		cout << "Error! cannot open file address.txt" << endl;
		return 10;
	}

	// Το επόμενο βήμα είναι να αποθηκεύσουμε τα στοιχεία του addressbook
	// στο αρχείο fout. Αυτό γίνεται με ένα απλό for. Η μέθοδος size() του
	// addressbook επιστρέφει (προφανώς) το μέγεθος του vector.
	cout << "Writing results to file address.txt" << endl;
	for (int j=0; j < addressbook.size(); j++) {
		fout << addressbook[j].name_ << " is "
		     << addressbook[j].age_ << " years old" << endl;
	}

	// και κλείνουμε το αρχείο
	fout.close();
	return 0;
}
```

Το πρόγραμμα αυτό αν το καλέσουμε και δώσουμε τις εξής τιμές:

```
Enter name: Kostas
```

```
Enter age: 29
Enter name: Nikos
Enter age: 33
Enter name: Myrsini
Enter age: 27
Enter name: Bilbo
Enter age: 22
Enter name: Gimli
Enter age: 27
```

θα δημιουργήσει το αρχείο `address.txt` που περιέχει τα εξής δεδομένα:

```
Kostas is 29 years old
Nikos is 33 years old
Myrsini is 27 years old
Bilbo is 22 years old
Gimli is 27 years old
```

Ε, ναι λοιπόν, τέλος!

Σόρρυ... κάποια άλλη φορά ίσως;

Βιβλιογραφία

C/C++ Programmer's Reference, 3rd Edition, SCHILDT H., OSBORNE, ISBN 0072227222

Η γλώσσα προγραμματισμού C++, BJARNE STROUSTRUP, Κλειδάριθμος, ISBN 9603321427, 1200 σελ.

C++: a Beginner's Guide, SCHILDT, OSBORNE, ISBN 0072194677

Thinking in C++ Vol 1, Bruce Eckel, PRENTISH HALL, ISBN 0139798099 (online)

Thinking in C++ Vol 2, Bruce Eckel, PRENTISH HALL, ISBN 0130353132 (online)