ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΣΗΜΕΙΩΣΕΙΣ ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΑΦΟΥΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

ΙΖΑΜΠΩ ΚΑΡΑΛΗ

ΑΘΗΝΑ – 2010

Περιεχόμενο του μαθήματος

- Γενικά για τον αντικειμενοστραφή προγραμματισμό και τις κλάσεις
- Δομή και μεταγλώττιση προγραμμάτων C++ (με χρήση του μεταγλωττιστή g++)
- Διαδικαστικός προγραμματισμός με/σε C++
- Εμβέλεια ονομάτων
- Αφαίρεση στα δεδομένα (σε C++)
- Δημιουργία / καταστροφή αντικειμένων
- Κληρονομικότητα (σε C++)
- Πολυμορφισμός (σε C++)
- Άλλες γλωσσικές δομές της C++, κρίσιμες για τις κλάσεις

Αντί Προλόγου

Από την ηλεκτρονική λίστα του μαθήματος:

Θα μπορούσατε να μας διευκρινήσετε τί ακριβώς εξυπηρετεί ο αντικειμενοστραφής προγραμματισμός; Όχι πώς υλοποιείται (με κλάσεις, αντικείμενα κλπ), αλλά ποια είναι η πρακτική χρησιμότητα και η ουσιαστική διαφορά στις δυνατότητές του από τον διαδικασιακό. Γιατί δηλαδή κάποιος θα χρησιμοποιήσει πχ τη C++ για να φτιάξει ένα πρόγραμμα, αντί για τη C. Αν και παρακολουθώ το μάθημα, δεν έχω καταλάβει την ουσιαστική διαφορά μεταξύ των δύο "ειδών" προγραμματισμού.

Τα δύο πρώτα μαθήματα εστίαζαν [α΄] στην ανάγκη ύπαρξης του αντικειμενοστραφούς προγραμματισμού. Επίσης, στο μάθημα της Παρασκευής [β΄] έγινε υπενθύμιση της ανάγκης ορισμού νέων τύπων, της αναπαράστασής τους σαν κλάσεις στο αντικειμενοστραφές μοντέλο και η υλοποίηση των κλάσεων στη C++ όπου κατόπιν εστιάσαμε.

Σε κάθε περίπτωση, όμως, δε βλάπτει να ξεκαθαρίσουμε και πάλι τα πράγματα μια που το θεωρώ σημαντικό να έχουν διασαφηνιστεί και μια που θυμάμαι κι εγώ, ξεκινώντας από γλώσσες αντικειμενοστραφούς προγραμματισμού κι όχι από εφαρμογές, ότι είχα την ίδια αγωνία. Να ξεκαθαρίσω ότι στα πλαίσια του μαθήματος μιλάμε για αντικειμενοστραφή προγραμματισμό πάνω από διαδικαστικό προγραμματισμό.

Είπαμε λοιπόν ότι το τι είναι κατάλληλο να επιλέξουμε: γλώσσα διαδικαστικού προγραμματισμού (π.χ. C) ή γλώσσα αντικειμενοστραφούς (π.χ. C++) εξαρτάται από τη φύση του προβλήματος που πάμε να αντιμετωπίσουμε. Πάρτε, για παράδειγμα,

---

[α΄]... και εξακολουθούν να εστιάζουν

[β΄]δηλαδή, το εισαγωγικό μάθημα για τους τύπους που ορίζονται από το χρήστη

το εξής πρόβλημα: “Γράψτε ένα πρόγραμμα το οποίο να ταξινομεί N ονόματα”. Εδώ τι έχουμε: η ταξινόμηση αποτελεί μια διαδικασία που βασίζεται σε μια συγκεκριμένη μέθοδο (αλγόριθμο) ταξινόμησης. Αυτή εφαρμόζεται σε κάποια δεδομένα που είναι απλά strings στο παράδειγμα. Επίσης, μπορούμε να πούμε ότι βασίζεται σε μια διαδικασία που πραγματοποιεί σύγκριση μεταξύ δύο strings.

Αντίστοιχο παράδειγμα θα ήταν, π.χ. να επιλυθεί ένα αριθμητικό πρόβλημα με χρήση μιας αριθμητικής μεθόδου.

Στα παραπάνω προβλήματα η έμφαση είναι στην υλοποίηση διαδικασιών οι οποίες εφαρμόζονται πάνω στα δεδομένα. Άρα, χρειαζόμαστε γλώσσες που οι γλωσσικές δομές τους (τα language constructs τους) διευκολύνουν την αναπαράσταση (υλοποίηση) διαδικασιών.

Πάρτε, τώρα το παράδειγμα της υλοποίησης μιας εφαρμογής γραφικών, π.χ. κινούμενα σχέδια. Εδώ τι έχουμε: έχουμε κάποιες οντότητες που τις σχετίζουμε με μια συμπεριφορά. Σαν συμπεριφορά θεωρούμε ένα σύνολο διαδικασιών (στον αντικειμενοστραφή διαδικαστικό προγραμματισμό) που μπορούν να διεξαχθούν αν αυτές οι οντότητες “λάβουν κάποια μηνύματα”. Στο παράδειγμα των κινουμένων σχεδίων έχουμε “σκηνές” που περιέχουν “κινούμενα σχέδια” τα οποία όλα έχουν κάποια γενική συμπεριφορά (π.χ. “μετακίνηση”, “στροφή”, “άλμα” κλπ.) αλλά και κάποια εξειδικευμένη (π.χ. ο “Roadrunner” έχει αυτό το “σπινάρισμα”).

Ανάλογο παράδειγμα είναι οι εφαρμογές προσομοίωσης, π.χ. προσομοίωση διεξαγωγής αγώνων στίβου για πειραματισμούς σε ανάγκες σε βοηθητικό προσωπικό. Δείτε επίσης και τη δεύτερη άσκηση της πρώτης ομάδας [γ΄].

Σε αυτά τα προβλήματα, τώρα, η έμφαση είναι στις οντότητες. Οι διαδικασίες σχετίζονται με αυτές και οφείλουν να τις συνοδεύουν.

[γ΄] Απλοϊκή προσομοίωση πολυκατοικίας.

Άρα, οι γλώσσες που χρειαζόμαστε για τις υλοποιήσεις οφείλουν να προσφέρουν γλωσσικές δομές οι οποίες να επιτρέπουν ορισμό οντοτήτων και συσχετισμό τους με τις διαδικασίες που αντιστοιχούν στη συμπεριφορά τους (π.χ. class στη C++). Επίσης, πρέπει να επιτρέπουν το συσχετισμό μεταξύ οντοτήτων (γενικότερη - εξειδίκευση) μέσω του μηχανισμού της κληρονομικότητας. Οι οντότητες έχουν μια εξωτερική συμπεριφορά αλλά δεν επιβάλλουν συγκεκριμένη εσωτερική αναπαράσταση. Καλό, λοιπόν, είναι ο μηχανισμός ορισμού των οντοτήτων να επιτρέπει διάκριση του τί είναι χρησιμοποιήσιμο από τον έξω κόσμο και του τί αποτελεί απόφαση του προγραμματιστή και άρα είναι λεπτομέρεια υλοποίησης. Το δεύτερο πρέπει να "εγκλωβίζεται" και να μην είναι προσβάσιμο από το υπόλοιπο πρόγραμμα. Ο μηχανισμός του εγκλωβισμού κάνει αυτή τη δουλειά. Έτσι γίνεται δυνατό ανά πάσα στιγμή να αλλάξουμε την υλοποίηση μιας οντότητας χωρίς να αλλάξουμε τίποτα στο υπόλοιπο πρόγραμμα, αρκεί να διατηρήσουμε τη διεπαφή της συμπεριφοράς του.

Εδώ όμως πρέπει να συμπληρώσω το εξής: η χρήση του αντικειμενοστραφούς προγραμματισμού επεκτάθηκε και σε περιοχές προβλημάτων που η αντιμετώπισή τους απαιτεί διάφορες δομές δεδομένων, όπως στοίβες, ουρές κλπ. (π.χ. λειτουργικά συστήματα, επεξεργαστές κειμένου κλπ.). Ακόμα, μπορούμε να πούμε πλέον ότι έχει ξεφύγει κι από τη χρήση του για ορισμό νέων τύπων δεδομένων κι έχει καταλήξει ακόμα και μια ολόκληρη διαδικασία ή ένα σύνολο διαδικασιών να "πακετάρεται" σε μια κλάση χρησιμοποιώντας την σαν μηχανισμό απόκρυψης πληροφορίας (π.χ. η προσέγγιση της Java).

Ελπίζω με τα παραπάνω τα πράγματα να ξεκαθαρίζουν καλύτερα.

Βήματα κατά την Εξέλιξη του (Αντικειμενοστραφούς) Προγραμματισμού

Διαδικαστικός Προγραμματισμός (procedural programming)

- εστιάζει στη σχεδίαση των διαδικασιών
- "αποφάσισε ποιες είναι οι διαδικασίες που χρειάζονται για την υλοποίηση και χρησιμοποίησε τους καλύτερους αλγορίθμους που μπορείς να βρείς"
- Παραδείγματα διαδικασιών:
  - εύρεση της τετραγωνικής ρίζας
  - εύρεση Μ.Κ.Δ.
  - ταξινόμηση αριθμών
- Γλώσσες: Algol60, Algol68, FORTRAN, Pascal, C κ.ά.

Προγραμματισμός με Στοιχεία (modular programming)

- εστιάζει στην οργάνωση των δεδομένων και των διαδικασιών
- "αποφάσισε ποιά είναι τα στοιχεία (modules) που σου χρειάζονται και χώρισε το πρόγραμμά σου έτσι ώστε να κρύψεις δεδομένα και διαδικασίες στα κατάλληλα στοιχεία"

- υπάρχουν κριτήρια για το χωρισμό (βλ. [Meyer])
- Παραδείγματα στοιχείων:
  - στοιχείο διαδικασιών επεξεργασίας προσωπικού
  - στοιχείο διαδικασιών επεξεργασίας εγγράφων
  - Γλώσσες: Modula -2, Modula -3 κ.ά.

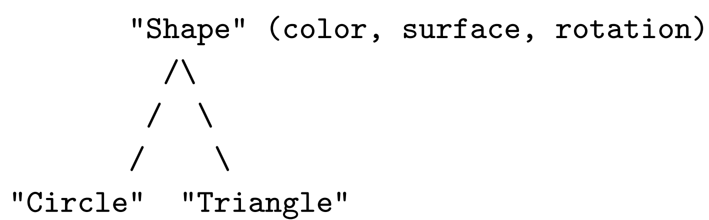
Αφαίρεση Δεδομένων (data abstraction)

- περισσότερη έμφαση στα δεδομένα
- συνδέει τύπους με λειτουργίες μέσω των οποίων χειριζόμαστε τα στοιχεία τους
- "αποφάσισε ποιούς τύπους χρειάζεσαι, δώσε ένα πλήρες σύνολο λειτουργιών για τον κάθε τύπο"
- Παραδείγματα τύπων: stack (top, pop, push)
- Γλώσσες: Ada, C++, Java κ.ά.

Αντικειμενοστραφής Προγραμματισμός (Object oriented programming)

- έκφραση "κοινών" χαρακτηριστικών μεταξύ τύπων μέσω κληρονομικότητας
- "αποφάσισε ποιες κλάσεις θέλεις, δώσε ένα πλήρες σύνολο από λειτουργίες για την κάθε κλάση, έκφρασε ρητά τις κοινές λειτουργίες μέσω κληρονομικότητας"

- Παραδείγματα κλάσεων:



- Γλώσσες: Simula, C++, Java κ.ά.

Γλώσσες με Χαρακτηριστικά Αντικειμενοστραφούς Προγραμματισμού

- ADA
- Smalltalk (1972)
- Modula -3
- C++ (1980)
- Java
- C#
- SIMULA (1967): προσφέρει τη γλωσσική δομή "class" για τον ορισμό κλάσεων, κατόπιν μπορούν να οριστούν αντικείμενα της κλάσης, *ΟΜΩΣ* δεν υπάρχει μηχανισμός προστασίας για τα ονόματα μέσα σε μια κλάση κι έτσι ο χρήστης μπορεί να προσπελάσει οποιοδήποτε μέρος της υλοποίησης

Υποστήριξη Αντικειμενοστραφούς Προγραμματισμού

- Ορισμός κλάσεων, αντικειμένων, κληρονομικότητας
- Δυνατότητα ελέγχου τύπων
  - λειτουργίες τύπου μόνο πάνω στα αντικείμενα του τύπου
  - αντικείμενα του τύπου προσπελαύνονται μόνο μέσω λειτουργιών του τύπου
- Δυνατότητα υποστήριξης εγκλωβισμού (encapsulation)
  - επιτυγχάνει απόκρυψη πληροφορίας (information hiding)
  - συνδέει τον τύπο με τις λειτουργίες του
  - μόνο τα χαρακτηριστικά και οι λειτουργίες γίνονται ορατά έξω από την κλάση
- Δυνατότητα δημιουργίας αντικειμένων (στιγμιοτύπων) και αρχικοποίησης τους
- Κληρονομικότητα (inheritance)

Ιδιότητες

- Στοιχειοποίηση ("διαίρει και βασίλευε")
- Εγκλωβισμός (απόκρυψη πληροφορίας)
- Επαναχρησιμοποίηση (reusability)
- Πολυμορφισμός
  - γενικευμένοι τύποι (`stack_of(X)`)
  - γενικευμένες διαδικασίες (`lenghth_of_list_of(X)`)
  - υπερφόρτωση συμβόλων (Αριθμός + Αριθμός, Λίστα + Λίστα) (overloading)

Γενικευμένοι Τύποι

Παράδειγμα: Στοίβες και Ουρές

- Στοίβες: από τα πιάτα έως τις κλήσεις συναρτήσεων
- Ουρές: από τους ανθρώπους έως τις διεργασίες

Απόκρυψη Πληροφορίας

Παράδειγμα: Έστω ότι οι γενικές οντότητες του προβλήματος είναι:

1. "ποδήλατο"
2. "ποδηλάτης"

Έστω ότι υπάρχει μια λειτουργία για το "ποδήλατο" που είναι "αλλαγή ταχύτητας" που πραγματοποιείται από τη λειτουργία του "ποδήλατου" "αλλαγή γραναζιού". Τη δεύτερη αυτή λειτουργία την κρατάμε κρυφή (την εγκλωβίζουμε): δεν πρέπει να μπορεί να την προσπελαύνει άμεσα ο "ποδηλάτης".

Ονοματολογία για τις Κλάσεις

Δομική (structural)

Δομικοί λίθοι: attributes, data members

Συμπεριφοράς (behavioral)

Διαδικασίες μέσω των οποίων προσπελαύνουμε τα αντικείμενα: member functions, procedures, methods/messages

Κλάσεις και Αφηρημένοι Τύποι Δεδομένων

- "Object Oriented Design is the construction of software systems as structured collections of abstract data type implementations" [Meyer]
- Παραδείγματα:
  - "stack": υλοποιημένο σαν συνδεδεμένη λίστα ή array
  - "student": υλοποιημένο μέσω `struct` ή σαν συνδεδεμένη λίστα
- Αφηρημένοι Τύποι Δεδομένων (ΑΤΔ) : Αφαιρετική περιγραφή τύπων μέσω των ιδιοτήτων των λειτουργιών τους κι όχι λεπτομερειών της αναπαράστασής τους
- Κλάσεις : Συγκεκριμένη μοντελοποίηση και υλοποίηση

C++ Γενικά

- C++ (=increment of C)
- Σχεδιάστηκε να
  - είναι μια καλύτερη C
  - υποστηρίζει αφαίρεση δεδομένων
  - υποστηρίζει αντικειμενοστραφή προγραμματισμό
- Bjarne Stroustrup, 1980, Bell Labs
- "C με κλάσεις" ⟶ C++ το 1983
- Για μεγαλύτερα προγράμματα
- OO χαρακτηριστικά εμπνευσμένα από τη Simula67 (C++ =? C + Simula67)
- Εμβέλεια (scope) ονομάτων-νοημάτων
- Κλάσεις, αντικείμενα, κληρονομικότητα, φιλικές κλάσεις, ιδεατές κλάσεις ...

Το Ελάχιστο C++ Πρόγραμμα

```
int main()
{
    return 0;
}
```

Για να έχουμε και κάποια έξοδο

```
#include <iostream>

using namespace std;

int main()
{
   cout << "Hello world" << endl;
   return 0;
}
```

Η δήλωση `#include <iostream>` δηλώνει στον προεπεξεργαστή να περιλάβει τις δηλώσεις που βρίσκονται στο αρχείο-επικεφαλίδα `iostream`

Μετάφραση και Αρχεία-επικεφαλίδες

- Στα αρχεία-επικεφαλίδες (header files) γράφουμε δηλώσεις (και ορισμούς συναρτήσεων και μεταβλητών με εσωτερική συνδεσιμότητα)
- `#include <iostream>` : με τη χρήση `< >` αναφερόμαστε στον standard include κατάλογο
- `#include "my_header.h"` : αναζήτηση του αρχείου-επικεφαλίδα στον τρέχοντα κατάλογο
- `#include "/home/users/izambo/c++/includes/hed.h"` : δίδουμε όλο το μονοπάτι ρητά
- `g++ -o bla main.cc f.cc`
- `g++ -c main.cc`
- `g++ -c f.cc`
- `g++ -o bla main.o f.o`
- Παράμετροι για τη μεταγλώττιση όπως περιγράφονται στο εγχειρίδιο του `g++`

```
 -c  Compile or assemble the source files, but do not link.
     The  compiler output is an object file corresponding to
     each source file.

.............

-E   Stop after the preprocessing stage; do not run the com-
     piler  proper.  The output is preprocessed source code,
     which is sent to the standard output.

...............
-o file
     Place output in file file.
```

Κάντε δοκιμές μεταγλώττισης χρησιμοποιώντας τα παρακάτω αρχεία

main.cpp

```
#include <iostream>
#include "p1.h"

using namespace std;

int main()
{
   cout << "HELLO!"<< endl;
    f(5);
   cout << "Hi!"<< endl;

   return 0;
}
```

p1.cpp

```
#include <iostream>

using namespace std;

void f(int i)
{
   cout << i << endl;
}
```

p1.h

```
void f(int);
```

Διαδικαστικός Προγραμματισμός με/σε C++

- Δεδομένα
  - Ενσωματωμένοι τύποι
  - Τύποι που ορίζονται από το χρήστη
- Διαδικασίες πάνω στα δεδομένα
  - Δομικά στοιχεία της γλώσσας και εντολές
  - Συναρτήσεις

Ονόματα

- για τις μεταβλητές
- για τις συναρτήσεις
- για τους τύπους
- ...
- Κάθε όνομα προτού να χρησιμοποιηθεί πρέπει να έχει δηλωθεί. Σε ένα πρόγραμμα πρέπει, για κάθε όνομα να έχουμε έναν ακριβώς ορισμό (εκτός αν το όνομα αναφέρεται σε οντότητες διαφορετικού είδους, π.χ. να χρησιμοποιήσουμε το ίδιο όνομα για έναν τύπο και μια μεταβλητή, ή επιτρέπεται από τη γλώσσα ρητά, π.χ. υπερφόρτωση ονομάτων συναρτήσεων)

Δηλώσεις - Ορισμοί - Αναθέσεις αρχικών τιμών

Δηλώσεις (declarations) (ΠΟΛΛΕΣ ΕΜΦΑΝΙΣΕΙΣ)
Σύνδεση ενός ονόματος με έναν τύπο σαν περιγραφή

```
extern int error_number;
```

Ορισμοί (definitions) (ΜΙΑ ΕΜΦΑΝΙΣΗ) Δέσμευση ενός ονόματος με μια συγκεκριμένη οντότητα

```
char ch;
```

Ανάθεση αρχικών τιμών Ανάθεση αρχικών τιμών στην οντότητα στην οποία αναφέρεται ένα όνομα

```
int count = 1;
const double pi = 3.141592;
```

Μεταβλητές και συναρτήσεις: Δηλώσεις - Ορισμοί

- Σε μια δήλωση, συνδέουμε ένα όνομα με το είδος της οντότητας την οποία ονομάζει το όνομα
- Σε έναν ορισμό δημιουργούμε την οντότητα (δεσμεύουμε το χώρο στη μνήμη)
- Μια συνάρτηση μπορεί να δηλωθεί χωρίς να οριστεί δίνοντας μόνο το πρότυπό της
- Ένας ορισμός αποτελεί ταυτόχρονα και δήλωση
- Η ανάθεση αρχικών τιμών σε μια μεταβλητή αποτελεί ταυτόχρονα και ορισμό και δήλωσή της

Παραδείγματα

```
char ch; <---- DEFINITION

string s; <---- DEFINITION

int count = 1;

const double pi = 3.14159265;

extern int error_number; <---- DECLARATION ONLY !!!!!

const char* Name = "TheName";

const char* season[] =
    {"anixi", "kalokeri", "fthinoporo", "ximonas"};

int day(Date* pd) { return pd -> d; } <---- DEFINITION

double sqrt(double); <---- DECLARATION ONLY !!!!!
```

Εμβέλεια (scope)

- Μια δήλωση εισάγει κάποιο όνομα μέσα σε μια εμβέλεια
- Η εμβέλεια ενός ονόματος ξεκινά από τη δήλωση του ονόματος
- Μεταβλητές:
  - καθολικές (global) (παντού)
  - στατικές εξωτερικές (έως το τέλος του αρχείου)
  - στατικές τοπικές σε συνάρτηση (έως το τέλος της εκτέλεσης της συνάρτησης)
  - τοπικές σε συνάρτηση
  - τοπικές σε block
- Χώροι ονομάτων (namespaces)
- Κλάσεις κ.ά.
- Τελεστής `::` (scope resolution operator): για επίλυση εμβέλειας

Συνδυάστε καθένα από τα αρχεία f1.cpp, f2.cpp και f3.cpp με το αρχείο main.cpp, τρέξτε το εκτελέσιμο που προκύπτει κάθε φορά και αιτιολογήστε τα αποτελέσματα της εκτέλεσης.

```
//File: main.cpp
#include <iostream>
using namespace std;

extern int x;
void f();

int main()
{
   cout << x << endl;
   f();
   return 0;
}
```

```
//File: f1.cpp
#include <iostream>
using namespace std;

   int x = 10000;

   void f()
   {
     int x;

     x = 1;

     cout << ::x << endl;

     {
        int x;

        x = 2;

        cout << x << endl;   // prints inner x = 2
     }

     cout << x << endl;  // prints outer x = 1

   }
```

```
// File: f2.cpp
#include <iostream>
using namespace std;

   int x = 20000;

   void f()
     {

        int x = 2;

        cout << x << endl;  // prints local x

        cout << ::x << endl;  // prints global x

     }
```

```
// File: f3.cpp
#include <iostream>
using namespace std;

    int x = 300000;

    void f()
    {
       int y = x;   // assigns global x

       int x = 2;  // defines a new local x

       cout << x << endl ;  // prints new x

       cout << y <<endl ;  // prints y which has the value of global x
    }
```

Συνδυάστε καθένα από τα αρχεία `f1.cpp`, `f2.cpp`, `f3.cpp` και `f4.cpp` με το αρχείο `main.cpp`, τρέξτε το εκτελέσιμο που προκύπτει κάθε φορά και αιτιολογήστε τα αποτελέσματα της εκτέλεσης.

```
// File: main.cpp
#include <iostream>
using namespace std;

extern int x;

int f();


int main()
{
   cout << "                x = " << x << endl;
   cout << "1st output of f()   " << f() << endl;
   cout << "2nd output of f()   " << f() << endl;

   return 0;

}
```

```
// File: f1.cpp

   int x = 10000;

   int f()
   {
     int x = ::x;

     return x++;
   }
```

```
// File: f2.cpp

   int x = 10000;

   int f()
   {
     static int x = ::x;

     return x++;
   }
```

```
// File: f3.cpp

  static int x = 10000; // CAREFUL: x static to current file!


   int f()
   {
     int x = ::x;

     return x++;
   }
```

```
// File: f4.cpp
#include<iostream>

using namespace std;
   int x = 10000;

   int f()
   {
     static int x = ::x;
     {static int x = 8; cout << x++ << endl; }

     cout << x++ << endl;
     return x;
   }
```

Χώροι Ονομάτων (Namespaces)

- Ένα namespace αποτελεί μια εμβέλεια

```
namespace Example{
          const double PI = 3.14159;
          const double E = 2.71828;
          int myInt = 8;
          void printValues();
            }
```

- Ενα namespace "πακετάρει" δηλώσεις και στο "πακέτο" δίνει ένα όνομα
- Μέσα στα namespaces γράφουμε δηλώσεις (και ορισμούς μεταβλητών και συναρτήσεων με εσωτερική συνδεσιμότητα)

Απαλλαγή από τη χρήση του ονόματος του χώρου ονομάτων

- Τα ονόματα ενός namespace γίνονται διαθέσιμα έξω από το namespace με τη χρήση του `using` και πλέον δεν χρειάζεται να χρησιμοποιηθεί το όνομα του namespace με χρήση του τελεστή επίλυσης εμβέλειας (::)

```
double pp = Example::PI;

using namespace Example;

double ee = E;
```

- Η επίδραση του `using` ισχύει έως το τέλος της τρέχουσας εμβέλειας (δηλαδή, τέλος του τρέχοντος block ή τέλος του αρχείου αν το `using` εμφανίζεται έξω από block)
- Δεν πρέπει να χρησιμοποιούμε `using` μέσα σε αρχεία-επικεφαλίδες διότι είναι πολύ επικίνδυνο

- Το `using` μπορεί να αναφέρεται σε ολόκληρο το namespace ή μόνο σε κάποια από τα ονόματα που δηλώνει
- Τα namespaces μπορούν να επαυξάνονται σε άλλα σημεία του κώδικα
- Δεν μπορούμε να επαυξήσουμε το namespace με ένα νέο όνομα απλώς προσδιορίζοντάς το με χρήση του τελεστή επίλυσης εμβέλειας, π.χ.

  ```
  int Example::newInt; // error
  ```

Επιπλέον δυνατότητες για τους χώρους ονομάτων

- Σύνθεση και επιλογή από namespaces
- Alias για namespaces: δίνουμε χαρακτηριστικά ονόματα στα namespaces και τα μετονομάζουμε δίνοντας τους συντομότερα για ευκολία
- Ανώνυμα namespaces: για τη δημιουργία “τοπικών” “κλειστών” χώρων ονομάτων

Παράδειγμα χρήσης namespace

```
#include <iostream>
using namespace std;

int myInt = 100000;


   namespace Example{
              const double PI = 3.14159;
              const double E = 2.71828;
              int myInt = 8;     // Avoid this in the general case!
              void printValues();
            }

int main()
   {
     cout << myInt << endl;

     cout << Example::E << endl;

     cout << "The namespace myInt: " << Example::myInt << endl ;

     Example::printValues();


     using namespace Example;

     cout << "The namespace E from main:  " << E << endl ;
     cout << ::myInt << endl;

     return 0;
   }

   void Example::printValues()
        {
           cout << "Printing from printValues" << endl;

           cout << myInt << endl;

           cout << E << endl;

        }
```

Παράδειγμα χρήσης ανώνυμου namespace

```
// Declarations in the unnamed namespace are COMPLETELY hidden
// outside this file. However the names can be used within the file
// as if they are "imported" by a "using" directive
// Effect: no need to declare this information as "static"


#include <iostream>
using namespace std;


   namespace {
              const double PI = 3.14159;
              const double E = 2.71828;
              int myInt = 8;
              void printValues()
                 {
                     cout << "Printing from printValues" << endl;

                     cout << myInt << endl;

                     cout << E << endl;

                 }
            }



int main()
   {
     cout << myInt << endl;

     cout << E << endl;

     cout << "The namespace myInt: " << myInt << endl ;

     printValues();

     return 0;
   }
```

Παράδειγμα οργάνωσης προγράμματος που χρησιμοποιεί namespace

```
// File main.cpp
#include <iostream>

#include "namespace.h"

using namespace std;

int myInt = 100000000;


int main()
   {
     cout << myInt << endl;

     cout << Example::E << endl;

     cout << "The namespace myInt: " << Example::myInt << endl ;

     Example::printValues();


     using namespace Example;

     cout << "The namespace E from main:  " << E << endl ;
     cout << ::myInt << endl;

     return 0;
   }
```

```
// File namespace.cpp
#include <iostream>

#include "namespace.h"

using namespace std;


   void Example::printValues()
        {
           cout << "Printing from printValues" << endl;

           cout << myInt << endl;

           cout << E << endl;

        }
```

```
// File: namespace.h

   namespace Example{
              const double PI = 3.14159;
              const double E = 2.71828;
              int myInt = 8;    // The problem with this becomes obvious now!
              void printValues();
            }
```

Ο χώρος ονομάτων της καθιερωμένης βιβλιοθήκης

- Σύμφωνα με το standard της C++ [Standard], καταργείται η παλαιότερη σύμβαση τα αρχεία-επικεφαλίδες της standard βιβλιοθήκης να έχουν κατάληξη `.h` και όλα τα ονόματα που ορίζονται ή δηλώνονται από αυτή βρίσκονται μέσα σε ένα namespace, το namespace `std`
- Προκειμένου να γίνουν άμεσα προσβάσιμα από το πρόγραμμα (χωρίς χρήση του τελεστή επίλυσης εμβέλειας), πρέπει να κάνουμε using το `std`
- Παράδειγμα

```
#include <iostream>

int main()
{ std::cout << "Printing!\n";}
```

Η αλλιώς:

```
#include <iostream>
using namespace std;

int main()
{cout << "Printing!\n";}
```

Τύποι

- Τύπος boolean (`bool`) - τιμές: `true`, `false` -
- Τύποι χαρακτήρων (π.χ. `char`)
- Τύποι ακεραίων (π.χ. `int`)
- Τύποι πραγματικών (π.χ. `double`)
- Τύποι απαρίθμησης (π.χ. `enum`)
- απουσία πληροφορίας : `void`
- Τύποι δεικτών (π.χ. `int*`)
- Arrays (π.χ. `char[]`)
- Τύποι αναφορών (π.χ. `int&`)
- Δομές δεδομένων και κλάσεις

Ενσωματωμένοι βασικοί τύποι της C++

- Για ακεραίους διαφόρων μεγεθών:
  `char`, `short int`, `int`, `long int`
- Για πραγματικούς:
  `float`, `double`, `long double`
- Για την ερμηνεία του πρώτου bit:
  `signed/unsigned char`
  `signed/unsigned short int`
  `signed/unsigned int`
  `signed/unsigned long int`

Εκτελώντας το ακόλουθο πρόγραμμα, θα ενημερωθείτε για τα μεγέθη των ενσωματωμένων βασικών τύπων στον υπολογιστή σας

```
//: C03:Specify.cpp
// From Thinking in C++, 2nd Edition
// Available at http://www.BruceEckel.com
// (c) Bruce Eckel 2000
// Copyright notice in Copyright.txt
// Demonstrates the use of specifiers
#include <iostream>
using namespace std;

int main() {
  char c;
  unsigned char cu;
  int i;
  unsigned int iu;
  short int is;
  short iis; // Same as short int
  unsigned short int isu;
  unsigned short iisu;
  long int il;
  long iil;  // Same as long int
  unsigned long int ilu;
  unsigned long iilu;
  float f;
  double d;
  long double ld;
  cout
    << "\n char= " << sizeof(c)
    << "\n unsigned char = " << sizeof(cu)
    << "\n int = " << sizeof(i)
    << "\n unsigned int = " << sizeof(iu)
    << "\n short = " << sizeof(is)
    << "\n unsigned short = " << sizeof(isu)
    << "\n long = " << sizeof(il)
    << "\n unsigned long = " << sizeof(ilu)
    << "\n float = " << sizeof(f)
    << "\n double = " << sizeof(d)
    << "\n long double = " << sizeof(ld)
    << endl;
} ///:~
```

Παραγόμενοι Τύποι

- `*` δηλώνει δείκτη (prefix)
- `&` δηλώνει αναφορά (prefix)
- `[]` δηλώνει array (postfix)
- `()` δηλώνει συνάρτηση (postfix)
- Παραδείγματα:

```
int* a;
float v[10];
char* p[20];
void f(int);
```

void, δείκτες, arrays

- αριθμητική σε δείκτες όπως στη C
- `void`: συντακτικά συμπεριφέρεται σαν τύπος
  - να δηλώσει ότι μια συνάρτηση δεν επιστρέφει τιμή
  - βασικός τύπος για δείκτες σε αντικείμενα που δεν ξέρουμε τον τύπο τους

Άλλα θέματα σχετικά με τους τύπους

- Μετατροπές τύπων
- `struct`

```
struct address {
        char* street;
        long number;
        char* city;
        int postcode;
      };
```

- Για μετονομασία τύπων, `typedef`

```
typedef char* Pchar;
Pchar p1;
```

  Προσοχή: Η χρήση του `typedef` στη C++ είναι πολύ περιορισμένη. Χρησιμοποιείται κυρίως:

  - για ονομασία παραγόμενων τύπων, π.χ. `char*`
  - σε περιπτώσεις που δεν έχουμε ακόμα αποφασίσει τον τύπο κάποιων μεταβλητών και ενδέχεται να τον αλλάξουμε αργότερα

- `union`
- `enum` (απαριθμήσεις)
- Στους τύπους θα επανέλθουμε συζητώντας για τις κλάσεις

ΕΚΦΡΑΣΕΙΣ και ΕΝΤΟΛΕΣ

```
statement
---------
            expression(opt)
            declaration
          { statement-list(opt) }
            ...
```

Τελεστές

- π.χ. `+`, `-`, `*`, `/`, `%`, ...
- Είναι πλούσια σε τελεστές, όπως η C
- Μερικοί έχουν και νέα σημασία, π.χ. `<<`
- Οι τελεστές

  `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`

  συμπτύσσονται με τον `=`

  π.χ. `a = a + 5;` $\longrightarrow$ `a += 5;`
- Γενικά E1 op= E2 ισοδύναμο με E1 = E1 op (E2) (με E1 να υπολογίζεται μια φορά)
- Γενικά οι τελεστές "ομαδοποιούνται" από αριστερά εκτός από τους μοναδιαίους τελεστές και τους τελεστές ανάθεσης που ομαδοποιούνται από δεξιά π.χ.

```
a = b = c      ---->   a = ( b = c )
a + b + c      ---->   ( a + b ) + c
*p++   ---->  * (p++)  NOT  (*p)++ (Which means what? Guess!)
```

- Υπάρχει καλά ορισμένη προτεραιότητα τελεστών, π.χ. οι τελεστές εμβέλειας, `::`, έχουν τη μεγαλύτερη και το κόμμα, `,` , σαν ένδειξη ακολουθίας έχει τη μικρότερη:

  `a + b * c` $\longrightarrow$ `a + ( b * c )`

  (ο τελεστής `*` έχει μεγαλύτερη προτεραιότητα)

  `a + b - c` $\longrightarrow$ `( a + b ) - c`

  (ίδια προτεραιότητα αλλά αριστερή "ομαδοποίηση")
- Καλό είναι να χρησιμοποιούμε παρενθέσεις, `(` και `)`, για να κάνουμε σαφή τη σειρά της αποτίμησης
- Γενικά η σειρά αποτίμησης δεν είναι εγγυημένη

- Οι τελεστές `,` , `&&`, `||` εγγυώνται ότι το αριστερό μέλος τους θα αποτιμηθεί πρώτα, π.χ.

  `b = ( a = 2, a + 1 )` $\longrightarrow$ `b` $= 3$ (κι όχι `b` $= 2$)

- Αύξηση και Μείωση (όπως στη C) με τους τελεστές `++` και `--`

  – Η έκφραση ++lvalue είναι ισοδύναμη με την έκφραση lvalue += 1, ισοδύναμα lvalue = lvalue + 1, (θεωρώντας ότι το lvalue δεν έχει παρενέργειες).

    `y = ++x` $\longrightarrow$ `y = (x+=1)`

    `y = x++` $\longrightarrow$ `y = (t=x,x+=1,t)`

  – Κατά την αύξηση/μείωση των δεικτών που αντιστοιχούν σε array, το αποτέλεσμα αναφέρεται στα στοιχεία του array (κι όχι, π.χ. στις διευθύνσεις). Για παράδειγμα,

    ```
    int p[] = {1,2,3,4};
    int* q = p; cout << *q << endl;
    q++; cout << *q << endl;
    ```

    μετά την αύξηση, ο `q` δείχνει στον επόμενο ακέραιο (του array) κι όχι στην επόμενη θέση μνήμης

Δείκτες και τελεστές: εκτελέστε και μελετήστε το παρακάτω πρόγραμμα

```
/////////////////////////////////////////////////////////////////////
#include <iostream>
using namespace std;

int main()
{
   int p[] = {10,20,30,40};
   int *q = p;
   int x;

///////////////////////////////////////////////////////////////////////

   x = *q;              // x equals to the value of the integer that
                        // q points to (x=10)
   cout << x << endl;   // number 10 is printed

///////////////////////////////////////////////////////////////////////

   x = *q++;         // x equals to the value of the integer that
                     // q points to (x=10),
                     // then q points to the next integer, i.e. p[1]
   cout << x << endl;   // number 10 is printed

///////////////////////////////////////////////////////////////////////

   x = *(q++);       // x equals to the value of the integer that
                     // q points to (x=20),
                     // then q points to the next integer, i.e. p[2]
   cout << x << endl;   //number 20 is printed

///////////////////////////////////////////////////////////////////////

   x = (*q)++;       // x equals to the value of the integer
                     // that q points to (x=30),
                     // then the value of p[2] is incremented by 1.
                     // No change for q
   cout << x << endl; // number 30 is printed

///////////////////////////////////////////////////////////////////////
```

```
   x = *q;            // x equals to the value of the integer that
                      // q points to (x=31)
   cout << x << endl; // number 31 is printed

////////////////////////////////////////////////////////////////////////

   x = *++q;         // first increment q to point to the next integer,
                     // i.e. p[3], and then
                     // assign to x its value (x=40)
   cout << x << endl; // number 40 is printed

////////////////////////////////////////////////////////////////////////

   x = ++*q;         // first increment the value of the integer that
                     // q points to and
                     // then assign this value to x (x=41)
   cout << x << endl;  // number 41 is printed

   return 0;
}
////////////////////////////////////////////////////////////////////////
```

Μερικές Παρατηρήσεις

- Δεν υπάρχουν εντολές ανάθεσης και εντολές κλήσης συνάρτησης: η ανάθεση και η κλήση συνάρτησης υπάγονται στην κατηγορία των εκφράσεων.
- Οι λογικοί τελεστές && και || δεν αποτιμούν το δεύτερο όρισμά τους εκτός αν είναι απαραίτητο.
- Όπου γίνονται έλεγχοι τιμών αλήθειας είναι πιο ευανάγνωστο να γράφουμε ρητά τις λογικές εκφράσεις, με χρήση των λογικών τελεστών, παρά να υπονοείται ο έλεγχος με βάση τις μη μηδενικές τιμές (μην ακολουθούμε την προσέγγιση της C να βασιζόμαστε στην μηδενική ή μη μηδενική τιμή της έκφρασης).
- Προσοχή στις προτεραιότητες! Δείτε το παρακάτω πρόγραμμα και αιτιολογείστε τι εκτυπώνει:

```
// Inspired by an example in Eckel's book
#include <iostream>
using namespace std;

int main()
{
  int a,b;

  b=1;
  cout << ((a = --b) ? b : (b = -99 ))
       << '\n' << " a is  " << a << '\n' <<
       " b is " << b << endl;

  b=1;
  cout << (a = --b ? b : (b = -99 ))
       << '\n' << " a is  " << a << '\n' <<
       " b is " << b << endl;

  return 0;
}
```

ΕΝΤΟΛΕΣ (à la C)

- Εντολές Επιλογής (selection/choice):
  - `if (` *expression* `)` *statement*
  - `if (` *expression* `)` *statement* `else` *statement*
  - `switch (` *expression* `)` *statement*
- Εντολές Επανάληψης (iteration):
  - `while (` *expression* `)` *statement*
  - `do` *statement* `while (` *expression* `);`
  - `for (` *init-statement* ; *expression(opt)* ; *expression* `)` *statement*
- Εντολές Μεταφοράς Ελέγχου (control):
  - `case` *constant-expression* : *statement*
  - `default:` *statement*
  - `break;`
  - `continue;`
  - `return` *expression(opt)* ;
  - `goto` *identifier* ;
- Ονομασία Εντολής: *identifier* : *statement*

Εντολές Επιλογής (selection/choice)

- Επιλογή *if-then*, συνήθης μορφή:

```
if (expression)
    statement
```

- Επιλογή *if-then-else*, συνήθης μορφή:

```
if (expression)
    statement
else
    statement
```

  – Η τιμή της έκφρασης `expression` μπορεί να είναι `true` ή `false`
  – Το `statement` μπορεί να αντιστοιχεί σε μια μόνο εντολή ή σε ένα block εντολών

- Πολλαπλή επιλογή *switch*, συνήθης μορφή:

```
switch (selector)
  {
     case int1 : statement ; break ;
     case int2 : statement ; break ;
       ....
     default : statement ;
  }
```

  – Η έκφραση `selector` παράγει μια τιμή ακεραίου
  – Ανάλογα με την παραγόμενη τιμή, ο έλεγχος πηγαίνει στο `statement` μετά το αντίστοιχο `case`
  – Το `break` χρειάζεται για βγει ο έλεγχος από το `switch`
  – το `default` υποδεικνύει τι πρέπει να εκτελεστεί στην γενική περίπτωση όπου κανένα από τα παραπάνω `case` δε μπορεί να εφαρμοστεί

Εντολές Επανάληψης (iteration)

- Επανάληψη *while*, συνήθης μορφή:

```
while (expression)
      statement
```

  - Η έκφραση `expression` αποτιμάται πριν από κάθε επανάληψη του `statement`
  - Η τιμή της έκφρασης `expression` μπορεί να είναι `true` ή `false`
  - Το `statement` μπορεί να αντιστοιχεί σε μια μόνο εντολή ή σε ένα `block` εντολών

– Παράδειγμα χρήσης *while* και *switch - case*

```
//: C03:Menu2.cpp
// From Thinking in C++, 2nd Edition
// Available at http://www.BruceEckel.com
// (c) Bruce Eckel 2000
// Copyright notice in Copyright.txt
// A menu using a switch statement
#include <iostream>
using namespace std;

int main() {
  bool quit = false;  // Flag for quitting
  while(quit == false) {
    cout << "Select a, b, c or q to quit: ";
    char response;
    cin >> response;
    switch(response) {
      case 'a' : cout << "you chose 'a'" << endl;
                 break;
      case 'b' : cout << "you chose 'b'" << endl;
                 break;
      case 'c' : cout << "you chose 'c'" << endl;
                 break;
      case 'q' : cout << "quitting menu" << endl;
                 quit = true;
                 break;
      default  : cout << "Please use a,b,c or q!"
                 << endl;
    }
  }
} ///:~
```

- Επανάληψη *for*, συνήθης μορφή:

```
for(initialization; conditional; statementL)
     statementB
```

  - Όταν μπαίνει ο έλεγχος στο `for` για πρώτη φορά, εκτελείται το `initialization` άπαξ
  - Πριν από κάθε επανάληψη, ελέγχεται αν ισχύει το `conditional`
  - Αν είναι το `conditional` αληθές, εκτελείται το `statementB`
  - Στο τέλος κάθε επανάληψης (άμα έχει ολοκληρωθεί η εκτέλεση του `statementB`) εκτελείται το `statementL`

- Επανάληψη *do-while*, συνήθης μορφή:

```
do
  statement
while (expression)
```

  - Το `statement` εκτελείται τουλάχιστον μια φορά, όταν ο έλεγχος μπει στο `do` για πρώτη φορά
  - Η έκφραση `expression` αποτιμάται μετά την εκτέλεση του `statement`
  - Μια πιθανή χρήση του είναι στην περίπτωση που η έκφραση `expression` εμπλέκει τιμές που διαβάζονται/υπολογίζονται στο `statement`. Για παράδειγμα, δείτε το παρακάτω πρόγραμμα:

```
//: C03:Guess2.cpp
// From Thinking in C++, 2nd Edition
// Available at http://www.BruceEckel.com
// (c) Bruce Eckel 2000
// Copyright notice in Copyright.txt
// The guess program using do-while
#include <iostream>
using namespace std;

int main() {
  int secret = 15;
  int guess; // No initialization needed here
  do {
    cout << "guess the number: ";
    cin >> guess; // Initialization happens
  }   while(guess != secret);
  cout << "You got it!" << endl;
} ///:~
```

ΣΥΝΑΡΤΗΣΕΙΣ

- Ο τρόπος για να "κάνουμε κάτι".
- Μια συνάρτηση δεν μπορεί να κληθεί αν δεν έχει δηλωθεί.
- Μια δήλωση συνάρτησης περιλαμβάνει:
  - το όνομα της συνάρτησης
  - τον τύπο της τιμής που επιστρέφει
  - το πλήθος και τους τύπους των ορισμάτων της
  - π.χ.

```
int sum(int i, int j);
void swap(int*, int*);
```

- Κατά την κλήση μιας συνάρτησης, η αντιμετώπιση είναι ίδια με την *αρχικοποίηση* των ορισμάτων (κι όχι με την ανάθεση τιμών).
- Μια συνάρτηση πρέπει να ορίζεται μόνο μια φορά μέσα σε ένα πρόγραμμα.
- Όταν καλείται μια συνάρτηση, οι χώροι που έχουν φυλαχθεί για τις τυπικές παραμέτρους της αρχικοποιούνται με τα αντίστοιχα ορίσματα.
- Επιστροφή τιμών: ίδια διαδικασία με την αρχικοποίηση μεταβλητών (μια εντολή `return` λειτουργεί σαν να *αρχικοποιεί* μια μεταβλητή του τύπου που επιστρέφει η συνάρτηση, στην οποία δίνεται η τιμή της έκφρασης της `return`, πραγματοποιώντας τις ανάλογες μετατροπές τύπων).

Υπερφόρτωση ονομάτων συναρτήσεων

- "υπερφόρτωση" (ή "επιφόρτιση"): overloading.
- Χρήση του ίδιου ονόματος για διαφορετικές συναρτήσεις.
- Διάκριση από το πλήθος/τύπο των ορισμάτων τους.
- Ο τύπος που επιστρέφει η συνάρτηση δεν παίζει ρόλο για τη διάκριση.
- Υπάρχουν κανόνες που υποδεικνύουν ποιο είναι το καλύτερο ταίριασμα τύπων κατά την κλήση μιας συνάρτησης.
- π.χ.

```
void print(int);            // for int
void print(const char*);    // for strings
void print(double);  ...
```

- Παράδειγμα:

```
#include <iostream>
using namespace std;

void print(int i)
{
   cout << "Printing an integer value: " << i << endl;
}

void print(double d)
{
   cout << "Printing a double: " << d << endl;
}

void print(const char* str)
{
   cout << "Printing a string: " << str << endl;
}


int main()
{
   int i = 10; double d = 3.14; char* str = "This is a string";
   print(i);
   print(d);
   print(str);

   return 0;
}
```

Default τιμές ορισμάτων

- Για την περίπτωση που η γενική μορφή της κλήσης μιας συνάρτησης εμπλέκει περισσότερα ορίσματα από τη μορφή των συχνότερων κλήσεών της. Π.χ.

```
void print(int value, int base=10);
```

  τότε

```
print(31)       --------->    31
print(31,10)    --------->    31
print(31,16)    --------->    1f
print(31,2)     --------->    11111
```

- Με χρήση εναλλακτικών ορισμών - επιτρεπτό, λόγω υπερφόρτωσης - πετυχαίνουμε το ίδιο αποτέλεσμα. Π.χ.

```
void print(int value, int base);
void print(int value);
```

  // με ορισμό για εκτύπωση στο 10δικό

- Ο έλεγχος τύπων των default ορισμάτων γίνεται στο επίπεδο της δήλωσης ενώ η αποτίμηση γίνεται κατά την κλήση της συνάρτησης.

- Δεν επιτρέπονται default τιμές σε ενδιάμεσα ορίσματα. Π.χ.

  ΟΧΙ `int g(int, int=0, int);`

  ΟΥΤΕ `int g(int=0, int, int);`

Για παράδειγμα, αν θέλαμε κάτι τέτοιο:

```
#include <iostream>
using namespace std;
int main()
{
 int i = 16;
 cout << "i in dec = " << i << endl;
 cout << hex << "i in hex = " << i << endl;
 cout << oct << "i in oct = " << i << endl;
 return 0;
}
```

με χρήση συνάρτησης με default τιμές ορισμάτων μπορούμε να γράψουμε το εξής:

```
#include <iostream>
using namespace std;

// the function void print(int,int)
// uses default value into its second argument

 void print(int i, int base = 10) // if not default value,
                                  // interchange with comment line
//void print(int i, int base)
{
     switch(base) {
       case (10) : cout << "i in dec = " << i << endl; break;
       case (16) : cout << hex << "i in hex = " << i << endl; break;
       case (8)  : cout << oct << "i in oct = " << i << endl; break;
       default   : cout << "No proper base " << endl;
     }
}

int main()
{
 int i = 16;
 print(i); // if not default value,
           // interchange with comment line
// print(i,10);
 print(i,16);
 print(i,8);
 return 0;
}
```

Συναρτήσεις inline

- Μια συνάρτηση μπορεί να οριστεί ως `inline` δίνοντας μια συμβουλή στο μεταγλωττιστή, να βάλει τον κώδικα της συνάρτησης στις κλήσεις της συνάρτησης, αντί να χρησιμοποιήσει τον κλασικό μηχανισμό κλήσης συναρτήσεων και επιστροφής αποτελεσμάτων. Π.χ.

  ```
  inline int fac(int n)
  {return (n<2?1:n*fac(n-1));}
  ```

  τότε, η κλήση `fac(6)`, ανάλογα με την "εξυπνάδα" του μεταγλωττιστή, μπορεί να γίνει `6*5*4*3*2*1` ή `720` ή `6*fac(5)` ή ακόμα και να μείνει ανέπαφη.

- Δεν είναι σίγουρο ότι κάθε `inline` συνάρτηση θα αντιμετωπιστεί `inline` (π.χ. αμοιβαία αναδρομικές συναρτήσεις, συναρτήσεις που καλούν τους εαυτούς τους ανάλογα με την είσοδό τους κλπ.)

- Η υπόδειξη `inline` δεν παραβιάζει την κλασική σημασία της κλήσης συνάρτησης. Άρα, ΔΕΝ ΕΧΟΥΜΕ ΤΙΣ ΠΑΡΕΝΕΡΓΕΙΕΣ ΠΟΥ ΔΗΜΙΟΥΡΓΟΥΝΤΑΙ ΑΠΟ `#define` της C.

- Οι συναρτήσεις `inline` και οι δηλώσεις `const` περιορίζουν τη χρήση `#define` στη C++.

Παράδειγμα για παρενέργειες της `#define` - Ασφαλής χρήση με `inline` συναρτήσεις:

```
#include <iostream>
using namespace std;

#define  MAX1(i,j) (((i)>(j))?(i):(j))

inline int max2(int i, int j)
       { return ( i > j? i : j) ;}

int main()
{
  int m1, m2, i = 5, j = 10;

  m2 = max2(i++,j++) ;

  cout << "m2 = " << m2 << '\n';

  cout << "i =" << i << endl;
  cout << "j =" << j << endl;


  m1 = MAX1(i++, j++);

  cout << "m1 = " << m1 << '\n';
  cout << "i = " << i << '\n';
  cout << "j = " << j << '\n';

  return 0;
}
```

αιτιολογήστε τα αποτελέσματα της εκτέλεσης του παραπάνω προγράμματος.

Υπενθύμιση από τη C: Ορίσματα στη συνάρτηση `main`.

Παράδειγμα:

```
#include <iostream>
using namespace std;

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

  cout << "argc = " << argc << endl;

  for( int i=0 ; i < argc ; i++ )
      cout << "argv[" << i << "] = " << argv[i] << endl;

  return 0;
}
```

ΣΤΑΘΕΡΕΣ

- Η λέξη-κλειδί `const` μπορεί να συμπεριληφθεί σε μια δήλωση προσδιοριστή (identifier) για να τον μετατρέψει σε σταθερά αντί για μεταβλητή. Π.χ.

```
const int a = 5;
```

```
   +-------+
 a |// 5 //|
   +-------+
```

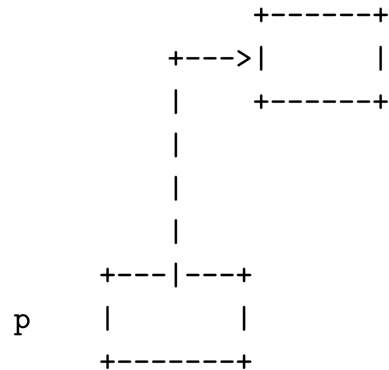
  το περιεχόμενο του `a` δεν μπορεί να αλλάξει

- Ακριβώς επειδή το περιεχόμενο δεν μπορεί να αλλάξει, ένας προσδιοριστής που δηλώνεται με `const` πρέπει να αρχικοποιείται

- Τελικά κι ο μεταγλωττιστής μπορεί να εκμεταλλευτεί μια δήλωση `const` (με διάφορους τρόπους, ανάλογα με το πόσο έξυπνος είναι)

Σταθερές και Δείκτες

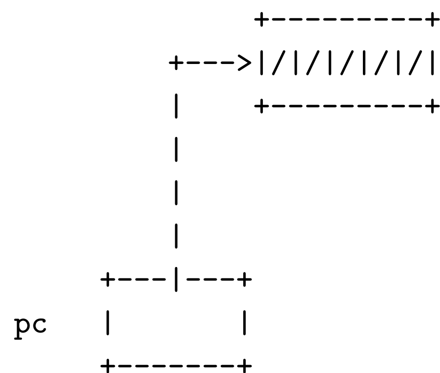
Ένας δείκτης p που δείχνει κάπου στη μνήμη, σχηματικά:



Άρα, έχουμε διάφορα ενδεχόμενα:

- Δήλωση σταθεράς για το αντικείμενο στο οποίο δείχνει ο δείκτης

```
const char* pc = "abcd";
```



Τότε,

```
pc[3] = 'a'; // error
```

- Δήλωση σταθεράς για το δείκτη

```
char *const cp = "abcd";
```

```
                  +---------+
              +--->| | | | | |
              |    +---------+
              |
              |
              |
          +---|---+
    cp    |///////|
          +-------+
```

Τότε,

```
cp = "efgh"; // error
```

- Δήλωση σταθεράς και για τα δύο

```
const char *const cpc = "abcd";
```

```
                  +---------+
              +--->|/|/|/|/|/|
              |    +---------+
              |
              |
              |
          +---|---+
    cpc   |///////|
          +-------+
```

Τότε,

```
cpc[3] = 'a'; // error
cpc = "efgh"; // error
```

Για να πειραματιστείτε, εκτελέστε το παρακάτω πρόγραμμα:

```
#include <iostream>
using namespace std;

int main()
{
const int a = 10;
// a = 1000;   // error
// a++;        // error

const int arr[] = {1,2,3};
// arr[1] = 2000;   // error

///////////////////////////////////////////////////////////////////

char abc[] = "abcdef";

const char* pc = abc;      // pointer to constant

// pc[3] = 'a';                  // error
 abc[3] = 'a';
pc = "efghijkl";                  // ok



char *const cp = abc;      // constant pointer

cp[3] = 'a';                  // ok
// cp = "efgh";                  // error


const char *const cpc = abc;      // constant pointer to constant
// cpc[3] = 'a';                        // error
// cpc = "efgh";                        // error

return 0;
}
```

Ένα άλλο παράδειγμα με συμβολοσειρές:

```
#include <iostream>
using namespace std;

int main()
{
 char a[] = "string";

 cout << a << endl;

 a[0] = 'd';

 cout << a << endl;

 char* sa = "litteral string";

 cout << sa << endl;

  sa++;

 cout << sa << endl;

//  sa[0] = 'd';   // Causes segmentation fault

 return 0;
}
```

Παρατηρήσεις για τη χρήση του const σε συνδυασμό με δείκτες

- Ένα αντικείμενο το οποίο είναι σταθερά όταν προσπελαύνεται μέσω ενός δείκτη μπορεί να είναι μεταβλητή όταν προσπελαύνεται αλλιώς (π.χ. δείκτης σε σταθερά σαν όρισμα συνάρτησης στο οποίο περνά μη σταθερά: το σώμα της συνάρτησης δεν μεταβάλλει το αντικείμενο, το περιβάλλον από το οποίο κλήθηκε η συνάρτηση και στο οποίο θα επιστρέψει ο έλεγχος, όμως, μπορεί)
- χρήσιμο για ορίσματα συναρτήσεων: δηλώνοντας ένα όρισμα που είναι τύπου δείκτη σαν `const`, η συνάρτηση απαγορεύεται να αλλάξει το αντικείμενο στο οποίο δείχνει. Για παράδειγμα,

  ```
  char* strcpy(char* p, const char* q)
  ```

  η συνάρτηση δεν μπορεί να αλλάξει το αντικείμενο στο οποίο δείχνει το `q`

Πέρασμα παραμέτρων σε συναρτήσεις που αλλάζουν το "εξωτερικό" αντικείμενο

- Πέρασμα παραμέτρων à la C:

```
void swap(int* p, int* q)
{
     int t = *p ;
        *p = *q ;
        *q = t  ;
}
```

Κλήση:

```
int i, j;
swap(&i, &j);
```

- Πέρασμα παραμέτρων à la C++

```
void swap(int& r1, int& r2)
{
     int t = r1 ;
        r1 = r2 ;
        r2 = t  ;
}
```

Κλήση:

```
int i, j;
swap(i, j);
```

Τιμές - Δείκτες ... και ΑΝΑΦΟΡΕΣ (references)

- Συγκρίνετε τα παρακάτω κομμάτια κώδικα και το αποτέλεσμα τους στην μνήμη

```
        int i, j;                                  +-------+
                                               i   |   3   |
        i = 3;                                     +-------+

        j = i;                                     +-------+
                                               j   |   3   |
                                                   +-------+

                                               +---------+   j
        int i;                                 |+-------+|   |
                                             i ||   3   ||<--+
        i = 3;                                 |+-------+|
                                               +---------+

        int &j = i;
```

Στη δεύτερη περίπτωση, το `j` είναι "συνώνυμο" του `i`

- Οι αναφορές συμπεριφέρονται σαν σταθεροί δείκτες προς το αντικείμενο με το οποίο αρχικοποιούνται (άρα πάντα πρέπει να αρχικοποιούνται) και δεν χρειάζονται dereferencing

- Προσοχή, ο τελεστής & εξακολουθεί να χρησιμοποιείται και για να εκφράσουμε τη διεύθυνση μιας μεταβλητής.
  Συγκρίνετε τα προηγούμενα με το:

```
                                +---------+
int i;                    i  +->|         |
                             |  +---------+
int* p;                      |
                             |
p = &i;                  +---|---+
                      p  |       |
                         +-------+
```

Άλλες παρατηρήσεις για τις αναφορές

- Στη γενική περίπτωση, τα ορίσματα συναρτήσεων περνάνε μέσω τιμών. Π.χ.

```
void f(int val)                                   main   |  f
                                                ---------+-------
                                                         |
 ...                                                     |
                                                  +---+  | +---+
int i=5;                                        i | 5 |  | | 5 | val
f(i);                                             +---+  | +---+
                                                         |
```

  Μπορούμε να περάσουμε δείκτη: σε αυτήν την περίπτωση έχουμε πρόσβαση στο "εξωτερικό" αντικείμενο από τη συνάρτηση, αλλά ο κώδικας γίνεται πολύπλοκος.

- Πετυχαίνουμε την ίδια πρόσβαση, αν περάσουμε αναφορά στο "εξωτερικό"αντικείμενο. Π.χ.

```
void f(int& val)                                  main   |  f
                                                ---------+-------
                                                         |
 ...                                                     |
                                                  +---+  | +---+
int i=5;                                        i | 5 |<------+ | val
f(i);                                             +---+  | +---+
                                                         |
```

- Μεγάλα αντικείμενα καλό είναι να τα περνάμε μέσω αναφοράς, για λόγους αποδοτικότητας.

- Ακόμα κι αν δεν έχουμε πρόθεση να αλλάξουμε την τιμή τους, το παραπάνω ισχύει. Σε αυτήν την περίπτωση όμως, καλό είναι να περιορίζουμε την αναφορά με `const`. Π.χ.

```
void f(const tree& t)
   {...}
```

  οπότε, στο σώμα της `f` απαγορεύεται να αλλάξει η τιμή του `t`.

- Τα arrays περνάνε μέσω αναφορών.

Μεταβλητές: απλές, αναφορές και δείκτες

Πειραματιστείτε με τα παρακάτω προγράμματα:

```
#include <iostream>
using namespace std;

int main()
{
  int i,j;
  i=5;
  j=i;
  cout << j << '\n' << i << '\n';
  cout << & j << '\n' << & i << '\n';

  int ii;
  cout <<  & ii << '\n';
  ii = 3;
  int & jj = ii;
  cout << jj << '\n' << ii << '\n';
  cout << & jj << '\n' << & ii << '\n';

  int* p;
  int* q;
  p = & ii;
  q = & jj;

  cout << p << '\n' << q << '\n';
  cout << &p << '\n' << &q << '\n';

  return 0;
}
```

Παρατηρήστε ότι ακόμα και όταν μιλάμε για τη διεύθυνση της αναφοράς, τελικά προσπελαύνουμε τη διεύθυνση της μεταβλητής με την οποία δεσμεύσαμε την αναφορά.

```
#include <iostream>
using namespace std;

int main()
{
    int i;
    int &j = i;

    i = 3;

//    int* p = &i; // interchange with the following line
    int* p = &j;

    cout << i << '\n' << j  << '\n' << *p <<  endl ;
    cout << p << '\n' << &j  << '\n' << &i << endl ;
    cout << &p << endl;

    return 0;
}
```

Πέρασμα παραμέτρων σε συνάρτηση: παραδείγματα

- Με τον τρόπο της C

```
//: C03:PassAddress.cpp
// From Thinking in C++, 2nd Edition
// Available at http://www.BruceEckel.com
// (c) Bruce Eckel 2000
// Copyright notice in Copyright.txt
#include <iostream>
using namespace std;

void f(int* p) {
  cout << "p = " << p << endl;
  cout << "*p = " << *p << endl;
  *p = 5;
  cout << "p = " << p << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(&x);
  cout << "x = " << x << endl;
} ///:~
```

- Με τον τρόπο της C++

```
//: C03:PassReference.cpp
// From Thinking in C++, 2nd Edition
// Available at http://www.BruceEckel.com
// (c) Bruce Eckel 2000
// Copyright notice in Copyright.txt
#include <iostream>
using namespace std;

void f(int& r) {
  cout << "r = " << r << endl;
  cout << "&r = " << &r << endl;
  r = 5;
  cout << "r = " << r << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(x); // Looks like pass-by-value,
        // is actually pass by reference
  cout << "x = " << x << endl;
} ///:~
```

ΟΙ ΤΕΛΕΣΤΕΣ `new` ΚΑΙ `delete`

Δέσμευση χώρου στη μνήμη:

static μεταβλητές Ο χώρος τους ανατίθεται όταν αρχίζει η εκτέλεση του προγράμματος και διατηρείται καθ' όλη την εκτέλεσή του.

automatic μεταβλητές Ο χώρος τους ανατίθεται κάθε φορά που ο έλεγχος της εκτέλεσης φτάνει στον ορισμό τους και διατηρείται έως ότου ο έλεγχος βγει από το block μέσα στο οποίο υπάρχει ο ορισμός.

δυναμική δέσμευση Κατά την εκτέλεση ενός προγράμματος, για να δημιουργήσουμε αντικείμενα ρητά τα οποία καταλαμβάνουν κάποιο χώρο έως ότου ρητά να τον αποδεσμεύσουν, χρησιμοποιούμε τους τελεστές `new` και `delete`.

- με τον τελεστή `new` δημιουργούμε τέτοια αντικείμενα
- με τον τελεστή `delete` τα καταστρέφουμε
- δεν υπάρχει garbage collection για τέτοια αντικείμενα κι έτσι πρέπει να προσέχουμε να αποδεσμεύουμε το χώρο
- τους χρησιμοποιούμε σε περιπτώσεις που, π.χ. θέλουμε να δημιουργούμε "δομικά στοιχεία" μιας μεγαλύτερης δομής που δε ξέρουμε το μέγεθός της κατά τη φάση της μεταγλώττισης. Π.χ. κόμβοι δένδρων, στοιχεία λίστας

Μεταβλητές στον στατικό (static) και στο μεταβαλλόμενο χώρο (stack): Πειραματιστείτε εκτελώντας τα παρακάτω προγράμματα και αιτιολογείστε τα αποτελέσματα ("σχεδιάστε" τους χώρους της μνήμης).

- Παράδειγμα

```
int i = 1000;   // global scope, allocated in static space

void g()
{
  int i = 5;    // scope within g, automatic variable
}

void f()
{
  int i = 50;   // scope within f, automatic variable
  static  int j = 500;   // scope within f, allocated in static space

  g();
}

int main()
{
  int i = 5000;  // scope within main, automatic variable

  f();

  return 0;
}
```

- Παράδειγμα

```
//File: vars.cc
#include <iostream>
using namespace std;

int g = 1000;
```

```
void p(int i)
{
  int j = 30;

  cout << "i in p is : " << i << endl;
  cout << "j in p is : " << j << "\n\n\n" << endl;
}

void f(int i)
{
	int j = 20;
	static int k = 100;

	cout << "i in f is : " << i << endl;
	cout << "j in f is : " << j << endl;

	cout << "k in f is : " << k << endl;

	cout << "g in f is : " << g << "\n\n" << endl;

	i++;
	j++;
	k++;

	g++;

	p(i);
}

int main()
{
  int i = 5;
	cout << "i in main is :" << i << '\n' << endl;
  f(i);
	cout << "i in main is :" << i << '\n' << endl;
  f(i++);
	cout << "i in main is :" << i << '\n' << endl;
  f(++i);
	cout << "i in main is :" << i <<  '\n' << endl;

	cout << "g in main is :" << g <<  '\n' << endl;

  return 0;
}
```

Δέσμευση στον ελεύθερο χώρο (heap):

Πειραματιστείτε εκτελώντας τα παρακάτω προγράμματα και αιτιολογείστε τα αποτελέσματα ("σχεδιάστε" τους χώρους της μνήμης).

- Παράδειγμα (τροποποίηση του προηγούμενου ώστε να γίνεται δυναμική δέσμευση χώρου στο σώμα της συνάρτησης p).

```
// Similar to vars.cc but space allocated within the body of function p
#include <iostream>
using namespace std;

int g = 1000;

void p(int i)
{
  int* j = new int(30);   // now j points to heap


  cout << "i in p is : " << i << endl;
  cout << "j in p points to : " << *j << "\n\n\n" << endl;

  delete j;
}

void f(int i)
{
       int j = 20;

       static int k = 100;

       cout << "i in f is : " << i << endl;
       cout << "j in f is : " << j << endl;

       cout << "k in f is : " << k << endl;

       cout << "g in f is : " << g << "\n\n" << endl;

       i++; j++; k++; g++;

       p(i);
}
```

```
int main()
{
  int i = 5;

       cout << "i in main is :" << i << '\n' << endl;

  f(i);

       cout << "i in main is :" << i << '\n' << endl;

  f(i++);


       cout << "i in main is :" << i << '\n' << endl;

  f(++i);

       cout << "i in main is :" << i <<  '\n' << endl;

       cout << "g in main is :" << g <<  '\n' << endl;

  return 0;
}
```

- Παράδειγμα: lifetime και εμβέλειας (χώρος που δεσμεύτηκε δυναμικά αποδεσμεύεται αλλά ο δείκτης που αρχικοποιήθηκε να δείχνει σε αυτόν παραμένει στην εμβέλεια).

```
#include <iostream>
using namespace std;

void p()
{
   int k = 300;
   int* kk = &k;
   cout << "kk points to :"  << *kk << "\n\n\n" << endl;


   int* j = new int(30);
   cout << "j in p points to : " << *j << "\n\n\n" << endl;

   delete j;   // deallocate storage where j pointed to;

   j = kk;     // j is NOT out of scope, now points where kk points to

   cout << "kk points to :"  << *kk << "\n\n\n" << endl;
   cout << "j after deletion points to :"  << *j << "\n\n\n" << endl;
}

int main()
{
 p();

 p();

 return 0;
}
```

Άλλες παρατηρήσεις πάνω στη δυναμική δέσμευση και αποδέσμευση χώρου

- Ο τελεστής `new` επιστρέφει δείκτη σε αντικείμενο του τύπου που προσδιορίζουμε. Π.χ.

  ```
  node* n = new node;
  ```

  Σημείωση: αν το `node` έχει οριστεί σαν κλάση, κατά τη δέσμευση του χώρου στον heap καλείται ο constructor της κλάσης (βλ. Κλάσεις)

- Ο τελεστής `delete` πρέπει να εφαρμόζεται σε δείκτες που επεστράφησαν από εφαρμογή του τελεστή `new`, π.χ. στο δείκτη που ορίστηκε στο παραπάνω

  ```
  delete n;
  ```

  Σημείωση: αν το `node` έχει οριστεί σαν κλάση, καλείται ο destructor της κλάσης (βλ. Κλάσεις)

- Μπορούμε να δημιουργήσουμε/καταστρέψουμε arrays από αντικείμενα χρησιμοποιώντας τους τελεστές `new` και `delete`. Π.χ.

  ```
  char* s = new char[10];
  delete[] s;
  ```

  Σημείωση: αν τα στοιχεία του array είναι αντικείμενα κλάσης, για καθένα από αυτά, καλείται ο default constructor της κλάσης (βλ. Κλάσεις)

- Αρχικοποίηση κατά τη δυναμική δημιουργία:

  ```
  int* pi = new int(1024);
  ```

  Προσοχή:

  – Αν έχουμε: `int* pi = new int(1024);`

`pi` δείκτης σε ακέραιο ο οποίος ακέραιος αρχικοποιείται με την τιμή 1024. Για να αποδεσμεύσουμε το χώρο αρκεί να κάνουμε: `delete pi;`

– Ενώ, αν έχουμε: `int* pia = new int[1024];` δημιουργία ενός array ακεραίων το οποίο έχει μέγεθος 1024. Για να αποδεσμεύσουμε το χώρο πρέπει να κάνουμε: `delete[] pia;`

Ένα απλό παράδειγμα διαδικαστικού προγραμματισμού με C++: διαγραφή χαρακτήρα από μια ακολουθία χαρακτήρων

- με επιστροφή του αποτελέσματος στην αρχική ακολουθία

```
#include <iostream>
using namespace std;

void remove(char*,char);

int main()
{
  char string1[] = "abagaadagaga";
  char string2[] = "abagaadagaga";
  char string3[] = "ccabbbb";


  remove(string1, 'a');
  cout << string1 <<endl;

  remove(string2, 'g');
  cout << string2 <<endl;

  remove(string3, 'a');
  cout << string3 <<endl;

  return 0;
}

void remove(char* str, char ch)
{
     int i=0, j=0;
     while (str[j] != '\0')
         {
              while (str[j] != ch && str[j] != '\0')
                    str[i++] = str[j++];
              while (str[j] == ch)
                    j++;
         }
     str[i] = '\0';
}
```

- με δημιουργία νέας ακολουθίας

```
#include <iostream>
using namespace std;

int length(const char* str)
{
  int length = 0;

  for(int i = 0; str[i] != '\0'; i++)
         length = i;

  return length;
}

char* remove(const char*,char);

int main()
{
  char* string1 = "abagaadagaga";

  cout << remove(string1, 'a') <<endl;
  cout << remove(string1, 'g') <<endl;

  return 0;
}

char* remove(const char* str, char ch)
{
  int l = length(str);

  char* res = new char[l+1];

  int j = 0;
  for( int i = 0; i <= l; i++ )

       if ( str[i] != ch )
          {
             res[j] = str[i] ;
             j++;
          }
  res[j] = '\0';
  return res;
}
```

- Προβληματική υλοποίηση: εσφαλμένη αναφορά σε μνήμη

```
int main()
{
  char* string1 = "abagaadagaga";

  cout << remove(string1, 'a') <<endl;
  cout << remove(string1, 'g') <<endl;

  return 0;
}

char* remove(const char* str, char ch)
{
  int l = length(str);

  char res[l+1];

  int j = 0;
  for( int i =0; i <= l; i++ )

       if ( str[i] != ch )
          {
             res[j] = str[i] ;
             j++;
          }

  res[j] = '\0';
  return res;  //error: when returning from remove, res out of scope
               //and its space is "discarded" from the stack
}
```

Γενικές Παρατηρήσεις περί Μεταγλώττισης και Συνδεσιμότητας Προγραμμάτων C++

- Ένα C++ πρόγραμμα είναι ένα σύνολο συναρτήσεων με τουλάχιστο μια συνάρτηση, τη συνάρτηση `main`
- Οι συναρτήσεις εφαρμόζονται πάνω σε δεδομένα-μεταβλητές που ανήκουν σε (built-in ή user-defined) τύπους
- Η εκτέλεση ενός προγράμματος C++ ξεκινά καλώντας τη συνάρτηση `main`. Η κλήση αυτή εισάγεται αυτόματα κατά την κατασκευή του εκτελέσιμου αρχείου του προγράμματος.
- Το πηγαίο C++ πρόγραμμα μπορεί να είναι γραμμένο σε ένα ή περισσότερα αρχεία
- Η μονάδα μεταγλώττισης είναι ένα αρχείο (δεν μπορούμε να μεταγλωττίσουμε κάτι λιγότερο, π.χ. μια συνάρτηση: αν θέλουμε να μεταγλωττίσουμε μόνο αυτήν, ας τη γράψουμε σε ένα αρχείο)
- Τα αρχεία τα οποία συμμετέχουν στην εντολή μεταγλώττισης περιέχουν κώδικα συναρτήσεων και ορισμούς μεταβλητών και είναι τα αρχεία κώδικα (κατάληξη `.cc`, `.cpp`)
- Τα αρχεία κώδικα μπορεί να κάνουν `include` αρχεία -επικεφαλίδες (header files)
- Στα αρχεία-επικεφαλίδες γράφουμε δηλώσεις (όχι κώδικα συναρτήσεων ή ορισμούς μεταβλητών) εκτός από ειδικές περιπτώσεις (`const` μεταβλητές, `inline` συναρτήσεις) και ορισμούς τύπων και χώρων ονομάτων
- Κάθε όνομα (μεταβλητής, συνάρτησης, τύπου κλπ.) πρέπει να έχει δηλωθεί πριν να χρησιμοποιηθεί μέσα σε ένα αρχείο (όχι απαραίτητα να οριστεί)

- Τόσο οι μεταβλητές ενός προγράμματος όσο και τα σώματα των ορισμών των συναρτήσεων καταλαμβάνουν χώρο στη μνήμη
- Ο ορισμός μιας μεταβλητής, δηλαδή η έκφραση εκείνη που οδηγεί σε δέσμευση χώρου στη μνήμη για τη μεταβλητή αυτή, πρέπει να υπάρχει ακριβώς μια φορά σε όλο το πρόγραμμα (εκτός αν πρόκειται για μεταβλητές που ανήκουν σε διαφορετικές εμβέλειες, άρα άλλες μεταβλητές)
- Ο ορισμός μιας συνάρτησης γίνεται με το να οριστεί και το σώμα της συνάρτησης, δηλαδή ο κώδικάς της
- Ο ορισμός μιας συνάρτησης πρέπει να υπάρχει ακριβώς μια φορά σε όλο το πρόγραμμα (εκτός αν πρόκειται για υπερφορτωμένους ορισμούς)

Συνδεσιμότητα - εμβέλεια μεταβλητών

Καθολικές μεταβλητές : Μια καθολική μεταβλητή

- ορίζεται σε ένα αρχείο, έξω από οποιαδήποτε συνάρτηση
- έχει εξωτερική συνδεσιμότητα, δηλαδή μπορεί να χρησιμοποιηθεί από οπουδήποτε στο πρόγραμμα

```
                   ---------------------------------
                  |                                 |
                 \|/  f1.cpp               f2.cpp   |
             +----------+             +--------------+
             |          |             |              |
             |int i;    |             |extern int i; |
             |          |             |              |
             | ......   |             | ......       |
             |          |             |              |
             +----------+             +--------------+
```

Στο αρχείο `f1.cpp` γίνεται ο ορισμός της μεταβλητής `i`. Στο αρχείο `f2.cpp` υπάρχει απλώς μια δήλωση για την μεταβλητή `i`.

Στατικές μεταβλητές : Μια στατική μεταβλητή

- ορίζεται σε ένα αρχείο, έξω από οποιαδήποτε συνάρτηση και προσημαίνεται με τη λέξη-κλειδί `static`
- έχει εσωτερική συνδεσιμότητα, δηλαδή μπορεί να χρησιμοποιηθεί μόνο μέσα στο αρχείο

```
                      f1.cpp               f2.cpp
             +--------------+             +--------------+
             |              |             |              |
             |static int i; |             |extern int i; |
             |              |             |              |
             | ......       |             | ......       |
             |              |             |              |
             +--------------+             +--------------+
```

Στο αρχείο `f1.cpp` γίνεται ο ορισμός της μεταβλητής `i`. Στο αρχείο `f2.cpp` υπάρχει μια δήλωση για μια μεταβλητή `i` όμως για την `i` αυτή δεν υπάρχει πουθενά ορισμός.

Μεταβλητές const : Μια μεταβλητή const

- αν οριστεί έξω από συνάρτηση, έχει εσωτερική συνδεσιμότητα (δε μπορεί να προσπελαθεί από άλλα αρχεία)
- για να αποκτήσει εξωτερική συνδεσιμότητα πρέπει στην δήλωσή της (και ορισμό της) να προσημανθεί και με τη λέξη-κλειδί `extern`

Στατικές μεταβλητές τοπικές σε συνάρτηση : Μια στατική μεταβλητή τοπική σε συνάρτηση

- ορίζεται μέσα στο σώμα μιας συνάρτησης, προσημαίνεται με τη λέξη -κλειδί `static` και οφείλει να αρχικοποιείται
- υφίσταται καθόλη τη διάρκεια των κλήσεων της συνάρτησης και διατηρεί την τιμή της ανάμεσα στις κλήσεις

Αυτόματες (τοπικές) μεταβλητές : Μια αυτόματη μεταβλητή

- ορίζεται μέσα στο σώμα μιας συνάρτησης, ενδεχόμενα και σε εσωτερικό block
- δημιουργείται προσωρινά, μέσα στη στοίβα (stack) και καταστρέφεται μόλις ο έλεγχος της εκτέλεσης βγει από τη συνάρτηση (ή το block) όπου έγινε ο ορισμός της
- δεν έχει καθόλου συνδεσιμότητα, ο linker δεν ασχολείται με τις τοπικές μεταβλητές

Δυναμική δημιουργία και καταστροφή μεταβλητών κατά την εκτέλεση ενός προγράμματος:

- δημιουργούνται με τη χρήση του τελεστή `new`
- ο χώρος που τους ανατίθεται βρίσκεται στο σωρό (heap)

- πρέπει να καταστραφούν ρητά με χρήση του τελεστή `delete` αλλιώς δεσμεύουν το χώρο έως το τέλος εκτέλεσης του προγράμματος

Παρατηρήσεις:

- Τα παραπάνω δεν κάνουν αναφορά σε μεταβλητές εμβέλειας κλάσης ούτε και χώρου ονομάτων.
- Τα στατικά μέλη -δεδομένα των κλάσεων δεσμεύονται στον στατικό χώρο και έχουν εμβέλεια κλάσης.

Συνδεσιμότητα - εμβέλεια συναρτήσεων

- Οι συναρτήσεις έχουν εξωτερική συνδεσιμότητα και μπορούν να χρησιμοποιηθούν από κώδικα που βρίσκεται σε οποιοδήποτε αρχείο του προγράμματος
- Για να αποκτήσει μια συνάρτηση εσωτερική συνδεσιμότητα και να μην μπορεί να χρησιμοποιηθεί έξω από το αρχείο στο οποίο ορίζεται πρέπει να προσημανθεί με τη λέξη -κλειδί `static`
- Οι συναρτήσεις `inline` έχουν εσωτερική συνδεσιμότητα.

Παρατήρηση: Τα παραπάνω δεν κάνουν αναφορά σε συναρτήσεις εμβέλειας κλάσης ούτε και χώρου ονομάτων.

ΚΛΑΣΕΙΣ

- Ορισμός από το χρήστη νέων τύπων, η πρόσβαση στα στοιχεία των οποίων γίνεται μέσω συγκεκριμένων συναρτήσεων.
- Ένας τύπος είναι η φυσική αναπαράσταση μιας ιδεατής οντότητας. Π.χ. ο τύπος `float` με `+`,`-`,`*` ...είναι η αναπαράσταση του πραγματικού αριθμού των μαθηματικών.
- Ένα πρόγραμμα που χρησιμοποιεί τύπους που προσομοιώνουν τις οντότητες της εφαρμογής γίνεται ευκολότερα κατανοητό.
- Διαχωρισμός των λεπτομερειών υλοποίησης του τύπου από τις ιδιότητες/λειτουργίες του τύπου που είναι απαραίτητες για τη σωστή χρήση του.

Κλάσεις στη C++

- Η θεμελιώδης έννοια του τύπου που ορίζεται από το χρήστη στη C++ είναι αυτή της κλάσης (class).
- Η πρόσβαση στα αντικείμενα μιας κλάσης γίνεται μέσω των συναρτήσεων-μελών (member functions) της κλάσης. (υπάρχει κι άλλη μια δυνατότητα, μέσω φιλικών συναρτήσεων και φιλικών κλάσεων, friends).
- Κατά τη δημιουργία αντικειμένων μιας κλάσης γίνεται κλήση συγκεκριμένων συναρτήσεων που δηλώνονται στη κλάση και λέγονται συναρτήσεις κατασκευής (constructors).
- Κατ' αναλογία, ορίζονται συναρτήσεις καταστροφής (destructors) που καλούνται όταν πρόκειται να γίνει καταστροφή των αντικειμένων των κλάσεων (απελευθέρωση του χώρου που τους έχει ανατεθεί).
- Ένα αντικείμενο μιας κλάσης μπορεί να δημιουργηθεί είτε σαν static είτε σαν automatic είτε μέσω `new`.

Τύποι δεδομένων ορισμένοι από το χρήστη - Συναρτήσεις -μέλη

Στη C θα μπορούσαμε να είχαμε, για παράδειγμα:

```
struct date { int day, month, year; };
date today;
void set_date(date*, int, int, int);
// ...
```

Αυτό, στη C++, μπορεί να γραφτεί σαν

```
struct date {
    int day, month, year;
    void set_date(int, int, int);
// ...
};
```

Σε αυτή την περίπτωση τη συνάρτηση set_date την εκφράσαμε σαν *συνάρτηση-μέλος* του τύπου date και εννοείται ότι, κατά τη χρήση της, εφαρμόζεται απαραίτητα σε ένα αντικείμενο του τύπου.

Δήλωση συναρτήσεων -μελών ενός νέου τύπου. Π.χ.

```
struct date {
    int day, month, year;
    void set(int, int, int);
    void get(int*, int*, int*);
};
```

Κλήση συναρτήσεων -μελών για μια μεταβλητή του νέου τύπου

```
date today;
int d,m,y;
today.set(3,5,2004);
today.get(&m,&d,&y);
```

```
    cout << "day " << d
         << ", month " << m
         << ", year " << y << endl ;
```

Ορισμός συναρτήσεων -μελών

```
void date::set(int i, int j, int k)
   {
     month = i;
     day = j;
     year = k;
   }

void date::get(int* i, int* j, int* k)
   {
     *i = month;
     *j = day ;
     *k = year;
   }
```

ΠΡΟΣΟΧΗ: μια συνάρτηση -μέλος χρησιμοποιεί τα ονόματα άλλων μελών χωρίς να αναφέρει ρητά κάποιο αντικείμενο: υπονοείται το αντικείμενο για το οποίο κλήθηκε η συνάρτηση -μέλος.

Ολοκληρωμένος ορισμός του νέου τύπου date,
Ορισμός Συναρτήσεων-μελών του και Χρήσης του

```
#include <iostream>
using namespace std;

 struct date {
        void set(int, int, int);
        void get(int&, int&, int&);
 private :
        int day, month, year;
    };


int main()
{
    date today;
    int d,m,y;

    today.set(3,18,2005);

//    cout << today.month << endl ;
// if the above is uncommented, an error occurs

    today.get(m,d,y);

    cout << "day " << d
         << ", month " << m
         << ", year " << y << endl ;

    return 0;
}
```

```
void date::set(int i, int j, int k)
   {
     month = i;
     day = j;
     year = k;

   }

void date::get(int& i, int& j, int& k)
   {
     i = month;
     j = day ;
     k = year;
   }
```

Αφαίρεση στα δεδομένα - Κλάσεις

Με τη χρήση της `struct` όπως κάναμε προηγούμενα, δεν εκφράσαμε ότι τα αντικείμενα θα είναι (άμεσα) προσπελάσιμα ΜΟΝΟ από τις συναρτήσεις -μέλη. Αν αυτό είναι το επιθυμητό, αυτό, συνήθως, επιτυγχάνεται με τη χρήση `class` αντί για `struct`.

Γενική δομή κλάσης:

```
class< όνομα κλάσης >
{
      <δηλώσεις>

public:
      <δηλώσεις>
};
```

Ή αλλιώς:

```
class< όνομα κλάσης >
{
public:
      <δηλώσεις>

private:
      <δηλώσεις>
};
```

Το παράδειγμα του νέου τύπου date, ορισμένο σαν κλάση

```
#include <iostream>
using namespace std;

   class date {
     public:
        void set(int, int, int);
        void get(int&, int&, int&);
     private:
        int day, month, year;
    };

int main()
{
    date today;
    date* ptoday = &today;

    int d,m,y;

    today.set(3,18,2005);
    today.get(m,d,y);
    cout << "day " << d << ", month " << m << ", year " << y << endl ;

    ptoday->get(m,d,y);
    cout << "day " << d << ", month " << m << ", year " << y << endl ;

    return 0;
}

void date::set(int i, int j, int k)
   {
     month = i;
     day = j;
     year = k;
   }

void date::get(int& i, int& j, int& k)
   {
     i = month;
     j = day ;
     k = year;
   }
```

Ορατότητα και προσπέλαση στα μέλη μιας κλάσης

- Τα ονόματα του ιδιωτικού (private) μέρους της κλάσης μπορούν να χρησιμοποιηθούν ΜΟΝΟ από τις συναρτήσεις -μέλη της κλάσης (γενικότερα, μόνο από την εμβέλεια της κλάσης).

- Το δημόσιο (public) μέρος μιας κλάσης αποτελεί τη διεπαφή (interface) της κλάσης μέσω της οποίας τα αντικείμενα της κλάσης είναι προσπελάσιμα έξω από την κλάση.

- Σημείωση: μια `struct` είναι μια `class` της οποίας τα μέλη (by default) είναι δημόσια ενώ μια `class` είναι μια `struct` της οποίας τα μέλη (by default) είναι ιδιωτικά.

- Οι συναρτήσεις -μέλη καθώς και τα μέλη δεδομένα δηλώνονται, καλούνται και ορίζονται ακριβώς όπως και στη `struct`.

- Η προσπέλαση στα μέλη μιας `class` γίνεται όπως και σε `struct` (με τη χρήση των τελεστών `.` και `->`, ανάλογα αν έχουμε μεταβλητή του τύπου ή δείκτη σε μεταβλητή του τύπου)

Ορίζοντας μια κλάση, συνήθως μιλάμε για

- τα δομικά στοιχεία που κτίζουν το αντικείμενο της κλάσης (building blocks)
- μια σειρά από συναρτήσεις -μέλη που αναθέτουν τιμές στα δομικά στοιχεία (mutators) και
- μια σειρά από συναρτήσεις -μέλη που ανακτούν τις τιμές αυτές (accessors)

Παράδειγμα:

```
class Student
{
 private:
    char* name;                   //  +
    int no;                       //  |     Building blocks
    int year_of_studies;          //  +
 public:
    void set_name(const char*);   //  +
    void set_no(int);             //  |     Mutators
    void set_year_of_studies(int); //  +
    char* get_name();             //  +
    int get_no();                 //  |     Accessors
    int get_year_of_studies();    //  +
};
```

Χρήση του τύπου `Student`

```
int main(){
   Student s;
   int year, no;

   s.set_name("FName LName");
   s.set_no(555);
   s.set_year_of_studies(1);

   year = s.get_year_of_studies();
   no = s.get_no();

   return 0;
}
```

Ο νέος τύπος `Student`: ορισμός - χρήση - δομή αρχείων προγράμματος:

Αρχεία:

- Ορισμός νέου τύπου:
  - `student.h`: ορισμός του νέου τύπου (αρχείο -επικεφαλίδα)
  - `student.cpp`: ορισμός των συναρτήσεων -μελών του νέου τύπου (αρχείο κώδικα)

```
//File: student.h
class Student {
private:
        char* name;
        int no;
        int year_of_studies;
public:
        void set_name(const char*);
        void set_no(int);
        void set_year_of_studies(int);
        char* get_name();
        int get_no();
        int get_year_of_studies();
};
```

```
//File: student.cpp
#include <cstring>
#include <iostream>
#include "student.h"
using namespace std;

void Student::set_name(const char* nam)
{
  name = new char[strlen(nam)+1];
  strcpy(name,nam);
}

void Student::set_no(int n){
        no = n;
}
```

```
void Student::set_year_of_studies(int y){
        year_of_studies = y;
}


char* Student::get_name(){
        return name;
}

int Student::get_no(){
        return no;
}

int Student::get_year_of_studies(){
        return year_of_studies;
}
```

- `classtest.cpp`: χρήση του νέου τύπου στον ορισμό μεταβλητών

```
//File: classtest.cpp
#include <iostream>
#include "student.h"
using namespace std;

int main()
{
        Student s;
        int year, no;

// s.no = 13;
        s.set_name("First Last");
        s.set_no(100);
        s.set_year_of_studies(1);
        s.set_name("NFirst Last");

        year = s.get_year_of_studies();
        no = s.get_no();
        cout << "\nYear of studies is " << year <<
                " for student with no " << no <<
                " and name " << s.get_name() << endl ;

        return 0;
}
```

Άλλες παρατηρήσεις σχετικά με τις κλάσεις

Ο δείκτης this

- Ουσιαστικά, οι συναρτήσεις -μέλη δουλεύουν σαν να έχουν κρυμμένο ένα όρισμα που αναφέρεται σε δείκτη στο αντικείμενο για το οποίο καλούνται. Δηλαδή, για μια κλάση X, ο δείκτης είναι `X *const this` (το `this` υποδηλώνει το δείκτη στο αντικείμενο για το οποίο καλείται η συνάρτηση).
- Η χρήση του `this` έχει νόημα όταν μια συνάρτηση -μέλος παίρνει σαν όρισμα μεταβλητές της ίδιας της κλάσης (π.χ. `void append(list*);` για συνένωση λιστών) και/ή υπάρχει ανάγκη για πρόσβαση στο δείκτη και χειρισμού του μέσα από το σώμα της συνάρτησης και/ή επιστροφής του αντικειμένου από τη συνάρτηση.
- Σε περίπτωση που μια συνάρτηση -μέλος δεν πρόκειται να τροποποιήσει το αντικείμενο για το οποίο καλείται, τότε στη δήλωσή της συμπεριλαμβάνουμε τη λέξη -κλειδί `const`, π.χ. `char* get_name() const;`. Σ' αυτήν την περίπτωση, ο τύπος του `this` μπορεί να θεωρηθεί ότι είναι για μια κλάση X , `const X *const this`

Η χρήση του δείκτη this στο παράδειγμα του τύπου date

```
#include <iostream>
using namespace std;

   class date {
     private:
        int day, month, year;
     public:
        void set(int, int, int);
        void get(int&, int&, int&);
    };


int main()
{
    date today;
    int d,m,y;
    today.set(3,18,2005);
    today.get(m,d,y);
    cout << "day " << d << ", month " << m << ", year " << y << endl ;
    return 0;
}

void date::set(int i, int j, int k)
   {
     this->month = i;
     this->day = j;
     this->year = k;
   }

void date::get(int& i, int& j, int& k)
   {
     i = this->month;
     j = this->day ;
     k = this->year;
   }
```

inline συναρτήσεις -μέλη

Προγραμματίζοντας με κλάσεις, συχνά χρησιμοποιούμε πολύ μικρές συναρτήσεις. Το κόστος κλήσης μιας συνάρτησης είναι μεγάλο. Μια συνάρτηση -μέλος που ορίζεται -κι όχι απλώς δηλώνεται- στον ορισμό μιας κλάσης θεωρείται ότι είναι inline. Π.χ.

```
class Student
{
  char* name;
    // ...
public:
    // ...
  char* get_name() { return name; }   // INLINE
    // ...
};
```

Παράδειγμα:

```
#include <iostream>
using namespace std;

   class date {
     private:
        int day, month, year;
     public:
     // all member functions are declared inline
        date(){ cout << "Creating a date!" << endl;
                month = 0; day = 0; year = 0; }
        ~date(){ cout << "Destroying a date!" << endl; }
        void set(int i, int j, int k)
             { month = i; day = j; year = k;}
        void get(int& i, int& j, int& k)
             { i = month; j = day ; k = year; }
    };
```

Αρχικοποίηση στιγμιοτύπων

- Δίνεται η δυνατότητα στον προγραμματιστή να δηλώνει κάποια(ες) συναρτήσεις που έχουν σαν στόχο να αρχικοποιούν σωστά τα αντικείμενα (δηλαδή, υπάρχει η δυνατότητα ταυτόχρονης αρχικοποίησης κατά τη στιγμή του ορισμού).
- Οι συναρτήσεις αυτές λέγονται *συναρτήσεις κατασκευής* (constructors) και προσδιορίζονται από το ότι έχουν το ίδιο όνομα με την κλάση. Π.χ.

```
    class date{
    // ...
    public:
            date(int, int, int);
            date(int, int);          // this year
            date(int);               // this year and this month
            date();                  // today
    // ...
    };

int main()
    {
    // ...
         date day(14);
         date day_month(14,3);
         date today;
    // ...
    }
```

- Μια κλάση που δεν έχει συναρτήσεις κατασκευής μπορεί να αρχικοποιήσει αντικείμενά της με αντιγραφή άλλων αντικειμένων της (π.χ. `date d = today;` αρχικοποίηση της `d` με αντιγραφή της `today`). Αυτό ισχύει ακόμα και για περιπτώσεις που η κλάση έχει συναρτήσεις κατασκευής.
- Γίνεται επιμέρους αντιγραφή τιμών (bitwise copy), σε περίπτωση που δεν έχει δοθεί κάποιος άλλος ορισμός.

- Ο άλλος ορισμός ουσιαστικά αναφέρεται στο να οριστεί μια συνάρτηση, για μια κλάση `X` , `X::X(const X&)`.
- Αυτή είναι η *συνάρτηση κατασκευής αντιγράφων* (copy constructor).
- Είναι ενδιαφέρον να ξεκαθαριστούν τα παρακάτω:
  1. Κατασκευή στιγμιότυπου κλάσης με αρχικοποίηση (με παραμέτρους άλλου τύπου) (με χρήση συνάρτησης κατασκευής)
  2. Κατασκευή στιγμιότυπου κλάσης με αντιγραφή (με χρήση συνάρτησης κατασκευής αντιγράφων)
  3. Ανάθεση στιγμιοτύπων (με υπερφόρτωση του τελεστή ανάθεσης, =)

  ΠΡΟΣΟΧΗ: η χρήση του τελεστή ανάθεσης στον ορισμό μεταβλητής κλάσης ΔΕΝ είναι έκφραση ανάθεσης. Υπονοεί χρήση του copy constructor.

Καταστροφή στιγμιοτύπων

- Η αντίστροφη διαδικασία της κατασκευής αντικειμένων είναι εκείνη της καταστροφής αντικειμένων.
- Πολλοί τύποι χρειάζονται να οριστεί μια *συνάρτηση καταστροφής* (destructor) αντικειμένων για να διασφαλιστεί η ορθή απελευθέρωση του χώρου που καταλαμβάνει ένα αντικείμενό τους.
- Γενική μορφή, για μια κλάση `X` , `~X()` (~: συμπλήρωμα)
- Όταν ένα αντικείμενο βγαίνει από την εμβέλεια, καλείται η συνάρτηση καταστροφής της κλάσης του. ΠΡΟΣΟΧΗ: στη C++, βγαίνοντας από το σώμα της `main`, δεν σταματά η εκτέλεση: καλούνται και οι συναρτήσεις καταστροφής όλων των αντικειμένων που δημιουργήθηκαν (εκτός εκείνων στο σωρό).

Το παράδειγμα της κλάσης student επαυξημένο με συναρτήσεις κατασκευής-καταστροφής

```
//File: student_constr.h
class Student {
private:
        char* name;
        int no;
        int year_of_studies;
public:
        Student(const char*);
        ~Student();

        void set_name(const char*);
        void set_no(int);
        void set_year_of_studies(int);

        char* get_name();
        int get_no();
        int get_year_of_studies();
};
```

```
//File: student_constr.cpp
#include <cstring>
#include <iostream>
#include "student_constr.h"

using namespace std;
//////////////////////////////////////////////////////////

void Student::set_name(const char* nam)
{
  delete[] name;
  name = new char[strlen(nam)+1];
  strcpy(name,nam);
}

void Student::set_no(int n)
{
        no = n;
}
```

```
void Student::set_year_of_studies(int y)
{
        year_of_studies = y;
}

/////////////////////////////////////////////////////////

char* Student::get_name()
{
        return name;
}

int Student::get_no()
{
        return no;
}

int Student::get_year_of_studies()
{
        return year_of_studies;
}

/////////////////////////////////////////////////////////

Student::~Student()
{
   cout << "Deleting student with name " << name << endl ;
   delete[] name;
}

/////////////////////////////////////////////////////////

Student::Student(const char* nam)
{
   name = new char[strlen(nam)+1];
   strcpy(name,nam);
   cout << "I just constructed a student " << endl;
}
```

```
// File: classtest_constr.cpp
#include <iostream>
#include "student_constr.h"

using namespace std;

int main()
{
        Student s("First Last");
//        Student s1(s); // default copy constructor used
        cout << "This is the first output of the main function " << endl;


        s.set_no(100);
        s.set_year_of_studies(1);
        s.set_name("NFirst Last");

        int year, no;

        year = s.get_year_of_studies();
        no = s.get_no();
        cout << "\nYear of studies is " << year <<
                " for student with no " << no <<
                " and name " << s.get_name() << endl ;

//////////////////////////

         cout << "Just exiting the main function ...." << endl;

         return 0;
}
```

Το παράδειγμα της κλάσης student επαυξημένο με συναρτήσεις κατασκευής-καταστροφής καθώς και συνάρτηση κατασκευής αντιγράφων

```
//File: student_constr.h
class Student {
private:
        char* name;
        int no;
        int year_of_studies;
public:
        Student(const char*);

        Student(const Student&);

        ~Student();

        void set_name(const char*);
        void set_no(int);
        void set_year_of_studies(int);

        char* get_name();
        int get_no();
        int get_year_of_studies();
};
```

```
//File: student_constr.cpp
#include <cstring>
#include <iostream>
#include "student_constr.h"
using namespace std;

/////////////////////////////////////////////////////////

void Student::set_name(const char* nam)
{
  delete[] name;
  name = new char[strlen(nam)+1];
  strcpy(name,nam);
}
```

```
void Student::set_no(int n)
{
        no = n;
}

void Student::set_year_of_studies(int y)
{
        year_of_studies = y;
}

//////////////////////////////////////////////////////////

char* Student::get_name()
{
        return name;
}

int Student::get_no()
{
        return no;
}

int Student::get_year_of_studies()
{
        return year_of_studies;
}

//////////////////////////////////////////////////////////

Student::~Student()
{
   cout << "Deleting student with name " << name << endl ;
   delete[] name;
}

//////////////////////////////////////////////////////////

Student::Student(const char* nam)
{
   name = new char[strlen(nam)+1];
   strcpy(name,nam);
   cout << "I just constructed a student " << endl;
}
```

```
Student::Student(const Student& s)
{
   name = new char[strlen(s.name)+1];
   strcpy(name,s.name);
//   name = s.name;     Careful: this is the result of default copy
   no = s.no;
   year_of_studies = s.year_of_studies;
   cout << "I just created a student by copying ... " << endl;
}
```

```
// File: classtest_constr.cpp
#include <iostream>
#include "student_constr.h"
using namespace std;


int main()
{
        Student s("First Last");
        cout << "This is the first output of the main function " << endl;


        s.set_no(100);
        s.set_year_of_studies(1);
        s.set_name("NFirst Last");

        int year, no;

        year = s.get_year_of_studies();
        no = s.get_no();
        cout << "\nYear of studies is " << year <<
                " for student with no " << no <<
                " and name " << s.get_name() << endl ;

//////////////////////////
//         Student scopied(s); // Student creation
//         Student sassgncopied = s; // Student creation
//
//
         cout << "Just exiting the main function ...." << endl << endl;

         return 0;

}
```

Το παρακάτω παράδειγμα σας επιτρέπει να πειραματιστείτε με τις εκτελέσεις των συναρτήσεων κατασκευής και καταστροφής μιας κλάσης καθώς και με τον επαναορισμό του τελεστή ανάθεσης σαν μέλος μιας κλάσης. (Υπενθύμιση: η χρήση του τελεστή ανάθεσης κατά τον ορισμό μιας μεταβλητής τύπου κλάσης είναι ισοδύναμη με χρήση του copy constructor και παρέχεται από τη γλώσσα για λόγους συμβατότητας με την αρχικοποίηση των ενσωματωμένων τύπων)

Πειραματιστείτε αφαιρώντας σταδιακά τα σχόλια. Προσπαθήστε να αιτιολογήσετε τα αποτελέσματα της εκτέλεσης.

Αρχείο -επικεφαλίδα `student_constr.h`

```
//File: student_constr.h
class Student {
private:
        char* name;
        int no;
        int year_of_studies;
public:
        Student(const char*);

        Student(const Student&);

        ~Student();

        Student& operator=(const Student& s);

        void set_name(const char*);
        void set_no(int);
        void set_year_of_studies(int);

        char* get_name();
        int get_no();
        int get_year_of_studies();
};
```

Αρχείο -κώδικα student_constr.cpp

```
//File: student_constr.cpp
#include <cstring>
#include <iostream>
#include "student_constr.h"
using namespace std;

///////////////////////////////////////////////////////////

void Student::set_name(const char* nam)
{
  delete[] name;
  name = new char[strlen(nam)+1];
  strcpy(name,nam);
}

void Student::set_no(int n)
{
        no = n;
}

void Student::set_year_of_studies(int y)
{
        year_of_studies = y;
}

///////////////////////////////////////////////////////////

char* Student::get_name()
{
        return name;
}

int Student::get_no()
{
        return no;
}

int Student::get_year_of_studies()
{
        return year_of_studies;
}
```

```
/////////////////////////////////////////////////////////

Student::~Student()
{
   cout << "Deleting student with name " << name << endl ;
   delete[] name;
}

/////////////////////////////////////////////////////////

Student::Student(const char* nam)
{
   name = new char[strlen(nam)+1];
   strcpy(name,nam);
   cout << "I just constructed a student " << endl;
}

Student::Student(const Student& s)
{
   name = new char[strlen(s.name)+1];
   strcpy(name,s.name);
//   name = s.name;     Careful: this is the result of default copy
   no = s.no;
   year_of_studies = s.year_of_studies;
   cout << "I just created a student by copying ... " << endl;
}

/////////////////////////////////////////////////////////

Student& Student::operator=(const Student& s)
{
   //  name = s.name;      Careful: this is the result of default assignment
   if (this != &s) // Careful not to assign to itself, especially
                   // if some storage reclaim was made within the body
      {
        delete[] name;
        name = new char[strlen(s.name)+1];
        strcpy(name,s.name);
        no = s.no;
        year_of_studies = s.year_of_studies;
        cout << "I just performed a student ASSIGNMENT ... " << endl;
      }
   return *this;
}
```

Χρήση της κλάσης Student

```
//File: classtest_constr.cpp
#include <iostream>
#include "student_constr.h"
using namespace std;

int main()
{
        Student s("First Last");
        cout << "This is the first output of the main function " << endl;


        s.set_no(100);
        s.set_year_of_studies(1);
//        s.set_name("NFirst Last");

        int year, no;

        year = s.get_year_of_studies();
        no = s.get_no();
        cout << "\nYear of studies is " << year <<
                " for student with no " << no <<
                " and name " << s.get_name() << endl ;

//////////////////////////
//         Student scopied(s); // Student creation
//         Student sassgncopied = s; // Student creation
//
//         scopied = sassgncopied;   // Student assignment
//
         cout << "Just exiting the main function ...." << endl << endl;

         return 0;
}
```

- Συναρτήσεις κατασκευής - καταστροφής: λειτουργία κατά το πέρασμα παραμέτρων συναρτήσεων και επιστροφής τιμής τους κατά τιμή.

  Πειραματιστείτε αφαιρώντας σταδιακά τα σχόλια. Προσπαθήστε να αιτιολογήσετε τα αποτελέσματα της εκτέλεσης. (Υπενθύμιση: Το πέρασμα παραμέτρων σε συνάρτηση καθώς και η επιστροφή αποτελέσματος είναι αντίστοιχο με αρχικοποίηση κι όχι ανάθεση).

```
//File: classtest_constr.cpp
#include <iostream>
#include "student_constr.h"
using namespace std;

void bla(Student s)
{
   cout << "Body of the bla function" << endl;
}


Student blaS(Student s)
{
   cout << "Body of the blaS function" << endl;
   return s;
}


int main()
{
        Student s("First Last");
        cout << "This is the first output of the main function " << endl;

        s.set_no(100);
        s.set_year_of_studies(1);
//        s.set_name("NFirst Last");

        int year, no;

        year = s.get_year_of_studies();
        no = s.get_no();
```

```
        cout << "\nYear of studies is " << year <<
                " for student with no " << no <<
                " and name " << s.get_name() << endl ;

/////////////////////////
//          Student scopied(s); // Student creation
//          Student sassgncopied = s; // Student creation
//
//          scopied = sassgncopied;   // Student assignment
//
//          bla(s);
//          blaS(s);
//
//          scopied = blaS(s);

         cout << "Just exiting the main function ...." << endl << endl;

         return 0;

}
```

- Πειραματιστείτε και για την περίπτωση που το πέρασμα παραμέτρων ή/και η επιστροφή συνάρτησης γίνεται με αναφορά. Αντικαταστήστε το αρχείο classtest_constr.cpp με το αρχείο refclass_test.cpp που ακολουθεί.

```
//File: refclass_test.cpp
#include <iostream>
#include "student_constr.h"
using namespace std;

void bla(Student& s)
{
   cout << "Body of the bla function" << endl;
}

Student& blaS(Student& s) // interchange with the following line
//Student blaS(Student& s)
{
   cout << "Body of the blaS function" << endl;
   return s;
}
```

```
int main()
{
        Student s("First Last");
//        Student & rs = s; // references work
        cout << "This is the first output of the main function " << endl;
        int year, no;

        s.set_no(100);
        s.set_year_of_studies(1);
//        s.set_name("NFirst Last");

        year = s.get_year_of_studies();
        no = s.get_no();
        cout << "\nYear of studies is " << year <<
                " for student with no " << no <<
                " and name " << s.get_name() << endl ;

//////////////////////////
//        Student scopied(s);
//
//
//         bla(s);
//         blaS(s);
//
//         scopied = blaS(s);

         cout << "Just exiting the main function ...." << endl << endl;

         return 0;
}
```

Προτερήματα της αφαίρεσης στα δεδομένα:
Πειραματιστείτε με τα παρακάτω. Ορισμός ενός νέου τύπου `Time` με δύο διαφορετικές αναπαραστάσεις:

1. Οι ώρες και τα λεπτά φυλάσσονται χωριστά (αρχεία `time1.h` - `time1.cc`)

```
//File: time1.h
class Time
{

      int hours;
      int mins;

 public:

      Time() { hours = 0; mins = 0; };
      ~Time() {};
      void set(int,int);
      void print();
      Time add(Time);

};



//File: time1.cc
#include <iostream>
#include "time1.h"

using namespace std;

void Time::print()
{
     cout << hours <<" hours  and "<< mins <<" mins " << endl;
}

void Time::set(int h, int m)
{
     hours = h;
     mins = m;
```

```
     print();

//   this->print();
}

Time Time::add(Time t)
{
    Time result;
    result.hours = 0;
    result.mins = mins + t.mins;

    if (result.mins >= 60)
       {
           result.mins -= 60;
           result.hours += 1;
       }

    result.hours += hours + t.hours;

    return result;
}
```

2. Ο χρόνος αναπαριστάνεται μόνο από τα λεπτά (αρχεία `time2.h` - `time2.cc`)

```
//File: time2.h
class Time
{
      int tmins;

 public:

      Time() {tmins = 0;}
      ~Time() {};

      void set(int,int);
      void print();
      Time add(Time);
};
```

```
//File: time2.cc
#include <iostream>
#include "time2.h"

using namespace std;

void Time::print()
{
     cout << tmins / 60 <<" hours  and "<< tmins % 60 <<" mins " << endl;
}

void Time::set(int h, int m)
{
     tmins = h * 60 + m;

     print();

//   this->print();
}

Time Time::add(Time t)
{
    Time result;
    result.tmins = tmins + t.tmins;

    return result;
}
```

Χρήση του νέου τύπου `Time` σαν συμμιγούς (αρχείο `timetest.cc`): η χρήση είναι ανεξάρτητη από την αναπαράσταση και την υλοποίηση του τύπου.

```
#include<iostream>
#include "time1.h"
//#include "time2.h"

using namespace std;

int main()
    {
      Time t;

      t.print();
      t.set(2,59);

//      t.hours; //  Not visible (even if time1.h is included)

      Time tt;
      tt.set(1,59);

      Time ttt;
      ttt = t.add(tt);

      ttt.print();

      return 0;
    }
```

Κάνοντας `include` το αντίστοιχο αρχείο -επικεφαλίδα, χρησιμοποιείται από το μεταγλωττιστή ο αντίστοιχος ορισμός της `Time`. Ποιος εναλλακτικός ορισμός είναι καταλληλότερος, εξαρτάται από την εφαρμογή στην οποία χρησιμοποιείται.

Παράδειγμα χρήσης των γλωσσικών δομών της C++ class, struct και typedef.

```
#include<iostream>
#include<cstring>
#include "time1.h"
//#include "time2.h"

using namespace std;

typedef int FligthNoType;

struct Flight
{
       FligthNoType FNo;
       char* DCity;
       char* ACity;
       Time DTime;
       Time Duration;
};


int main()
    {
      Flight AthensParis;

      AthensParis.FNo = 314;
      AthensParis.DCity = "Athens";
      AthensParis.ACity = "Paris";
      AthensParis.DTime.set(10,30);
      AthensParis.Duration.set(3,15);

      Time ATime;

      ATime = AthensParis.DTime.add(AthensParis.Duration);

       cout << AthensParis.DCity << " to " << AthensParis.ACity << endl;
       cout << "Arriving ...at  " ;
       ATime.print();

       return 0;
    }
```

ΕΠΑΝΑΧΡΗΣΙΜΟΠΟΙΗΣΗ (reusability)

- μέσω σύνθεσης (composition) **has a**
- μέσω κληρονομικότητας (inheritance) **is a**

Σύνθεση

- Μια κλάση έχει σαν μέλη αντικείμενα άλλης κλάσης.
- Αρχικοποίηση αντικειμένων: Πρώτα αρχικοποιούνται τα αντικείμενα που είναι μέλη της κλάσης (με τη σειρά που δηλώνονται στον ορισμό της κλάσης κι όχι με τη σειρά που εμφανίζονται στη συνάρτηση κατασκευής της κλάσης) κι έπειτα αρχικοποιείται το αντικείμενο της κλάσης.
- Καταστροφή αντικειμένων: Ακολουθείται η αντίστροφη σειρά από εκείνη της δημιουργίας, δηλαδή, πρώτα εκτελείται το σώμα της συνάρτησης καταστροφής της κλάσης και μετά εκείνες των μελών της (με την αντίστροφη σειρά της δήλωσης).

Παράδειγμα μέλους δεδομένου που ανήκει σε κλάση

Ο τύπος date:

```
//File: day_class.h
#include <iostream>

   class date {
     private:
        int day, month, year;
     public:
        date(){ std::cout << "Creating a date!" << std::endl;
                month = 0; day = 0; year = 0; }
        ~date(){ std::cout << "Destroying a date!" << std::endl; }
        void set(int i, int j, int k)
             { month = i; day = j; year = k;}
        void get(int& i, int& j, int& k)
             { i = month; j = day ; k = year; } };
```

Ο τύπος student:

```
//File: student_constr.h
#include"day_class.h"

class Student {
private:
        char* name;
        int no;
        int year_of_studies;
        date RegistrationDate;
public:
        Student(const char*);
        ~Student();


        void set_name(const char*);
        void set_no(int);
        void set_year_of_studies(int);
        void set_date(int,int,int);

        char* get_name();
```

```
        int get_no();
        int get_year_of_studies();
        void get_date(int&,int&,int&); };



//File: student_constr.cpp
#include <cstring>
#include <iostream>
#include "student_constr.h"
using namespace std;

//////////////////////////////////////////////////////////
void Student::set_name(const char* nam)
{
  delete[] name;
  name = new char[strlen(nam)+1];
  strcpy(name,nam);
}

void Student::set_no(int n)
{
        no = n;
}

void Student::set_year_of_studies(int y)
{
        year_of_studies = y;
}

void Student::set_date(int m, int d, int y)
{
       RegistrationDate.set(m,d,y);
}



//////////////////////////////////////////////////////////
char* Student::get_name()
{
        return name;
}
```

```
int Student::get_no()
{
        return no;
}

int Student::get_year_of_studies()
{
        return year_of_studies;
}

void Student::get_date(int& m, int& d, int& y)
{
       RegistrationDate.get(m,d,y);
}

//////////////////////////////////////////////////////////
Student::~Student()
{
   cout << "Deleting student with name " << name << endl ;
   delete[] name;
}

//////////////////////////////////////////////////////////
Student::Student(const char* nam)
{
   name = new char[strlen(nam)+1];
   strcpy(name,nam);
   cout << "I just constructed a student " << endl;
}
```

Χρήση του student:

```
#include <iostream>
#include "student_constr.h"
using namespace std;

int main()
{
        Student s("First Last");
        cout << "This is the first output of the main function " << endl;


        s.set_no(100);
        s.set_year_of_studies(1);
        s.set_name("NFirst Last");
        s.set_date(11,20,2005);

        int year, no;

        year = s.get_year_of_studies();
        no = s.get_no();
        cout << "\nYear of studies is " << year <<
                " for student with no " << no <<
                " and name " << s.get_name() << endl ;

        int rday, rmonth, ryear;
        s.get_date(rmonth,rday,ryear);
        cout << "and the registration date is:" << rday << "   "
             << rmonth << "   " << ryear << endl;

//////////////////////////

         cout << "Just exiting the main function ...." << endl << endl;

         return 0;

}
```

Παράδειγμα για:

- τον ορισμό πολλαπλών συναρτήσεων κατασκευής (με χρήση default τιμών στα ορίσματα)
- τη χρήση initializer list για non -class μέλη -δεδομένα (αντί για ανάθεση στο σώμα της συνάρτησης κατασκευής) (Σημείωση: αυτή η συντακτική δυνατότητα παρέχεται για λόγους συμβατότητας με τα class μέλη -δεδομένα όπου στην περίπτωση αυτή, έτσι περνάμε παραμέτρους στις συναρτήσεις κατασκευής τους)
- τον ορισμό κλάσης με μέλη -δεδομένα που ανήκουν σε άλλες κλάσεις
- την κλήση (non -trivial synthesized) συναρτήσεων κατασκευής/καταστροφής για τα αντικείμενα της σύνθετης κλάσης

Πειραματιστείτε βγάζοντας τα σχόλια των ορισμών των μεταβλητών στο σώμα της `main` του ακόλουθου προγράμματος.

```
#include <iostream>
using namespace std;

//////////////////////////////////////////////////////////////////////
   class Date {

        int day;
        int month;
        int year;

      public:

        Date(int d = 1, int m = 1, int y = 2006)
         : day(d), month(m), year(y) // non-class types,
                                     // i.e. initializer list
                                     // alternative to assignment in the body
            {
//              day = d; month = m; year = y;

              cout << "I just created a Date with value: "
                   << day << ' ' << month << ' ' << year << endl;
             }

        ~Date() {
                   cout << '\n' << "I am destroying a Date with value: "
                   << day << ' ' << month << ' ' << year << endl;
                 }

        void print()
                  { cout << day << ' ' << month << ' ' << year << endl; }

    };


//////////////////////////////////////////////////////////////////////
class Time {

      int hours;
      int mins;

 public:

      Time(int h = 0, int m = 0 )
        : hours(h), mins(m)
```

```
          {
//             hours = h; mins = m;
            cout << "I just created a Time with value: "
                 << hours << ' ' << mins << endl;
          }

      ~Time() {
               cout << '\n' << "I am destroying a Time with value: "
                    << hours << ' ' << mins << endl;
              }

      void print()
               { cout << hours <<" hours  and "<< mins <<" mins " << endl; }
};


/////////////////////////////////////////////////////////////////////
class DetailedDate{

       Date date;
       Time time;

      public:

        DetailedDate()
// Default constructor is synthesized, using the default constructors of
// the data members.
          { cout << "DetailedDate was just created with value:" << '\n' << endl;
// NO VISIBILITY in private part
//          cout << date.day << ' ' << date.month << ' ' << date.year << endl;
//          cout << time.hours <<" hours  and "<< time.mins <<" mins " << endl;

            date.print();
            time.print();
          }

//         DetailedDate(int d, int mo, int y, int h, int mi):
         DetailedDate(int d, int mo=0, int y=0, int h=0, int mi=0):
                     date(d,mo,y), time(h,mi)
// Initializer list is required to pass the values to the constructors
// of data members.
            { cout << "DetailedDate was just created with value:"
                   << '\n' << endl;
```

```
              date.print();
              time.print();
            }

        ~DetailedDate()
            {
              cout << '\n'
                   << "I am destroying a DetailedDate with value: " << endl;
              date.print();
              time.print();
            }
};

/////////////////////////////////////////////////////////////////////
int main()
{
//   Date d1;
//   Date d2(3);
//   Date d3(3,4);
//   Date d4(3,4,2005);

//   Time t1;
//   Time t2(5);
//   Time t3(2,35);

// dd1, dd2 and dd3 are instances/objects of DetailedDate

   DetailedDate dd1;
//   DetailedDate dd2(10);
//  DetailedDate dd3(25,10,2005,3,45);

   return 0;
}
```

Μέλη-δεδομένα που είναι δείκτες σε αντικείμενα κλάσεων:

- Προβληματική αντιμετώπιση:
  1. προβληματική αρχικοποίηση (μη ασφαλής δέσμευση και αρχικοποίηση)
  2. κακή προσπέλαση μέσω δεικτών

```
#include <iostream>
using namespace std;

/////////////////////////////////////////////////////////////////////
   class Date {
        int day;
        int month;
        int year;

     public:

        Date(int d = 1, int m = 1, int y = 2004)
            { day = d; month = m; year = y;
              cout << "I just created a Date with value: "
                   << day << ' ' << month << ' ' << year << endl; }

        ~Date()
             { cout << '\n' << "I am destroying a Date with value: "
                    << day << ' ' << month << ' ' << year << endl; }

        void print()
                  {cout << day << ' ' << month << ' ' << year << endl;}
    };

/////////////////////////////////////////////////////////////////////
class Time {
      int hours;
      int mins;

 public:

      Time(int h = 0, int m = 0 )
          {  hours = h; mins = m;
```

```
            cout << "I just created a Time with value: "
                 << hours << ' ' << mins << endl; }

      ~Time()
           { cout << '\n' << "I am destroying a Time with value: "
                 << hours << ' ' << mins << endl; }

      void print()
               {cout << hours <<" hours  and "<< mins <<" mins " << endl;}
};

/////////////////////////////////////////////////////////////////////
class DetailedDate{

       Date* date;
       Time* time;

      public:
        DetailedDate()
      { cout << "DetailedDate was just created with value:" << '\n' << endl;
//      CAREFUL: no space has been allocated or initialized for Date and Time

              date->print();   // error
              time->print();   // error
              }

        ~DetailedDate()
             { cout << '\n'
                    << "I am destroying a DetailedDate with value: " << endl;
               date->print();
               time->print(); }
};

/////////////////////////////////////////////////////////////////////
int main()
{
   DetailedDate dd1;

   return 0;
}
```

- Ολοκληρωμένη αρχικοποίηση/αποδέσμευση μνήμης: πειραματιστείτε βγάζοντας τα σχόλια από τους ορισμούς των μεταβλητών της main.

```
#include <iostream>
using namespace std;

////////////////////////////////////////////////////////////////////
   class Date {

        int day;
        int month;
        int year;

      public:

        Date(int d = 1, int m = 1, int y = 2005)
            {
              day = d; month = m; year = y;

              cout << "I just created a Date with value: "
                   << day << ' ' << month << ' ' << year << endl;
             }

        ~Date() {
                   cout << '\n' << "I am destroying a Date with value: "
                   << day << ' ' << month << ' ' << year << endl;
                }

        void print()
                  {cout << day << ' ' << month << ' ' << year << endl;}

    };

////////////////////////////////////////////////////////////////////
class Time {

      int hours;
      int mins;

 public:

      Time(int h = 0, int m = 0 )
```

```
        {
           hours = h; mins = m;
           cout << "I just created a Time with value: "
                << hours << ' ' << mins << endl;
        }

    ~Time() {
              cout << '\n' << "I am destroying a Time with value: "
                   << hours << ' ' << mins << endl; }

    void print() {
               cout << hours <<" hours  and "<< mins <<" mins " << endl;}

};


//////////////////////////////////////////////////////////////////////
class DetailedPDate{

       Date* date;
       Time* time;

    public:
      DetailedPDate()
          { cout << "DetailedPDate was just created with value:"
                 << '\n' << endl;

//              date = new Date;
//              time = new Time;
// Interchange the above with:
                time = new Time;
                date = new Date;

              date->print();     // actually this->date->print()
              time->print();
          }

       DetailedPDate(int d, int mo, int y, int h, int mi)
           {
             cout << "DetailedPDate was just created with value:"
                  << '\n' << endl;

             date = new Date(d,mo,y);
             time = new Time(h,mi);
```

```
            date->print();
            time->print();
         }

      ~DetailedPDate()
          {
            cout << '\n'
                 << "I am destroying a DetailedPDate with value: "
                 << endl;

            date->print();
            time->print();

            delete date;
            delete time;
          }
};

/////////////////////////////////////////////////////////////////////
int main()
{
   DetailedPDate dd1;
//   DetailedPDate dd2(25,10,2005,3,45);

   return 0;
}
```

ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑ (inheritance)

```
                      VEHICLE:
               (move, position, wheels)

                     /    |     \
                    /     |      \
                   /      |       \
                  +       +        +

               CAR:   BICYCLE:   MOTORBIKE:
         (4_wheels,    (2_wheels,    (2_wheels,
          engine)       no_engine)    engine)

      /   |  \   \
     /    |   \   \
    +     |    \   +
          |     \
CABRIO:   |      \   4_WHEEL_DRIVE:
(no_roof) |       \  (drive_in_4_wheels)
          |        +
          |
          |       LIMOUSINE:
          +       (no_back_door)

     HATCHBACK:
     (back_door)
```

- Ένα "όχημα" έχει κάποια "θέση" από την οποία μπορεί να "μετακινηθεί" με χρήση "τροχών".
- Ένα "αυτοκίνητο" είναι ένα "όχημα" με "τέσσερις τροχούς" και "μηχανή".
- Ένα "cabrio" είναι ένα "αυτοκίνητο" που "δεν έχει οροφή".
- ...

- Πιο γενικευμένες οντότητες και εξειδικεύσεις τους
- Διαβάζουμε το παραπάνω σχήμα από πάνω προς τα κάτω. Σχεδιάζουμε ανάποδα: Βρίσκουμε κοινά χαρακτηριστικά στις οντότητες και φτιάχνουμε νέες, πιο γενικευμένες, οντότητες με αυτά.
- Ορολογία:

  Γενικεύσεις: *Βασική κλάση* (base class), *υπερκλάση* (superclass)

  Εξειδικεύσεις: *Παραγόμενη κλάση* (derived class), *υποκλάση* (subclass)
- Άλλο ζήτημα στην κληρονομικότητα: *πολλαπλή κληρονομικότητα* (multiple inheritance)

Κληρονομικότητα στη C++

- Νέες κλάσεις μπορούν να οριστούν από ήδη υπάρχουσες ενσωματώνοντας τα χαρακτηριστικά και τη συμπεριφορά τους ή επαναορίζοντάς τα όπου είναι απαραίτητο.
- Μια παραγόμενη κλάση μπορεί να προσθέσει μέλη -δεδομένα και συναρτήσεις -μέλη επαυξάνοντας τον ορισμό της (εξειδικεύοντας όμως τα στοιχεία της).
- Ο τελεστής επίλυσης εμβέλειας μπορεί να χρησιμοποιηθεί για να χρησιμοποιηθεί η εκδοχή που ορίζεται σε μια βασική κλάση αντί για την επαναορισθείσα στην παραγόμενη.

Παράδειγμα: Ένας "ωρομίσθιος εργαζόμενος" σαν υποπερίπτωση "εργαζομένου"

Ορισμός των κλάσεων:

```
class Employee
   {
    public:
       Employee(const char*, const char*);
       void print() const;
       ~Employee();

    private:
       char* firstname;
       char* lastname;
   };

class HourlyWorker : public Employee
   {
    public:
      HourlyWorker(const char*, const char*,
                   double, double);
      double getPay() const;
      void print() const;

    private:
      double wage;
      double hours;
   };
```

Ορισμός των συναρτήσεων -μελών της HourlyWorker:

```
HourlyWorker::HourlyWorker(const char* first,
                           const char* last,
                           double initHours,
                           double initWage)
             : Employee(first,last)
 {
      hours = initHours;
      wage = initWage;
 }


 double HourlyWorker::getPay() const
        { return wage * hours; }


 void HourlyWorker::print() const
      {
          Employee::print();
          cout << "Payment:" << getPay() << endl;
      }
```

Χρήση της HourlyWorker:

```
int main()
   {
      HourlyWorker hw("FN", "LN", 0.0, 100.0);
      hw.print();
      return 0;
   }
```

Διαβαθμίσεις στον έλεγχο της πρόσβασης στα μέλη μιας κλάσης

Ένα μέλος μιας κλάσης μπορεί να είναι:

private: το όνομά του μπορεί να χρησιμοποιηθεί μόνο από τις συναρτήσεις -μέλη της κλάσης και φιλικές συναρτήσεις και φιλικές κλάσεις της κλάσης.

protected: το όνομά του μπορεί να χρησιμοποιηθεί από τις συναρτήσεις -μέλη της κλάσης και φιλικές συναρτήσεις και φιλικές κλάσεις της κλάσης καθώς και από συναρτήσεις -μέλη παραγόμενων κλάσεων και φιλικές συναρτήσεις και φιλικές κλάσεις παραγόμενων κλάσεων.

public: το όνομά του μπορεί να χρησιμοποιηθεί από οποιαδήποτε συνάρτηση.

```
-----------------   +
|               |   |
|    PUBLIC     |   |  PUBLIC PART
+---------------+   +
| / PROTECTED / |   |
+---------------+   |  PRIVATE PART
|////PRIVATE////|   |
|///////////////|   |
-----------------   +
```

Ζητήματα προς προβληματισμό στην κληρονομικότητα

Αναπαράσταση αντικειμένων της παραγόμενης κλάσης: κτίζονται από τη συνένωση των μελών -δεδομένων της βασικής κλάσης και των δικών της.

Θέματα εμβέλειας:

- Πώς (μέσω ποιων ονομάτων) χειριζόμαστε τα αντικείμενα της παραγόμενης κλάσης από τον υπόλοιπο κώδικα;
  Στην περίπτωση της is -a κληρονομικότητας (`public` inheritance), πέρα από τα ονόματα που είναι δηλωμένα στο δημόσιο τμήμα της παραγόμενης κλάσης, μπορούμε να χρησιμοποιήσουμε και κάθε όνομα που είναι στο δημόσιο τμήμα της βασικής κλάσης. Στην περίπτωση που μια συνάρτηση -μέλος έχει επαναοριστεί στην παραγόμενη, ο ορισμός που υπάρχει στη βασική δεν είναι (άμεσα) ορατός.
- Πώς (μέσω ποιων ονομάτων) χειριζόμαστε τα αντικείμενα της παραγόμενης κλάσης από το περιβάλλον της κλάσης αυτής; (π.χ. στα σώματα των ορισμών των συναρτήσεων -μελών της παραγόμενης κλάσης)
  Όσον αφορά κληρονομούμενα ονόματα από τη βασική κλάση, μπορούμε να χρησιμοποιήσουμε τα ονόματα που είναι δηλωμένα είτε στο `public` είτε στο `protected` τμήμα της.
- Πώς (μέσω ποιων ονομάτων) χειριζόμαστε τα αντικείμενα της βασικής κλάσης από το περιβάλλον της παραγόμενης;
  Όπως ακριβώς θα τα χειριζόμασταν από οποιοδήποτε άλλο μέρος του κώδικα, δηλαδή μέσω του δημόσιου

τμήματός τους.

Αρχικοποίηση - καταστροφή των αντικειμένων της παραγόμενης κλάσης:

- Όταν δημιουργείται ένα αντικείμενο μιας παραγόμενης κλάσης, μια συνάρτηση κατασκευής της βασικής κλάσης πρέπει να κληθεί για να αρχικοποιήσει εκείνα τα μέλη του στιγμιοτύπου που προέρχονται από τη βασική κλάση.
- Στις συναρτήσεις κατασκευής της παραγόμενης κλάσης πρέπει να προσδιορίζεται η συνάρτηση κατασκευής της βασικής που είναι να χρησιμοποιηθεί και να περαστούν οι τιμές στα ορίσματά της (αυτό γίνεται με τη χρήση initializer list). Διαφορετικά καλείται η default συνάρτηση κατασκευής της βασικής κλάσης.
- Οι συναρτήσεις κατασκευής δε κληρονομούνται.
- Κατά την κατασκευή ενός αντικειμένου, πρώτα καλείται η συνάρτηση κατασκευής της βασικής κλάσης και μετά της παραγόμενής της.
- Κατά την καταστροφή η σειρά είναι ανάποδα.

Αντιγραφή (Αρχικοποίηση μέσω αντιγραφής μεταξύ αντικειμένων της παραγόμενης κλάσης): με σειρά αντίστοιχη με τη σειρά κλήσης των συναρτήσεων κατασκευής, γίνεται bitwise copy είτε καλούνται οι (επανα)ορισμένες συναρτήσεις από το χρήστη αν υπάρχουν.

Κληροδότηση κληρονομιάς: Τι κληρονομείται στους απογόνους μιας παραγόμενης κλάσης από αυτά που κληρονόμησε από τη βασική της; (βλ. δηλώσεις κληρονομικότητας).

Τα παραπάνω ισχύουν αναδρομικά για κάθε βάθος ιεραρχίας.

Πειραματιστείτε σε θέματα ορατότητας και αναπαράστασης με το παρακάτω:

```
//: C14:Inheritance.cpp + Useful.h
// From Thinking in C++, 2nd Edition
// Available at http://www.BruceEckel.com
// (c) Bruce Eckel 2000
// Copyright notice in Copyright.txt
// Simple inheritance
#include <iostream>
using namespace std;
class X {
  int i;
public:
  X() { i = 0; }
  void set(int ii) { i = ii; }
  int read() const { return i; }
  int permute() { return i = i * 47; }
};

class Y : public X {
  int i; // Different from X's i
public:
  Y() { i = 0; }
  int change() {
    i = permute(); // Different name call
    return i;  }
  void set(int ii) {
    i = ii;
    X::set(ii); }  // Same-name function call
};

int main() {
  cout << "sizeof(X) = " << sizeof(X) << endl;
  cout << "sizeof(Y) = "
       << sizeof(Y) << endl;
  Y D;
  D.change();
  // X function interface comes through:
  D.read();
  D.permute();
  // Redefined functions hide base versions:
  D.set(12);
} ///:~
```

Επίσης, πειραματιστείτε βγάζοντας τα σχόλια από το παρακάτω:

```
#include <iostream>
using namespace std;

class X {
// protected:
  int i;

 public:
  X() { i = 10; }

  void set(int ii) {
     cout << "This is X::set" << endl;
     i = ii; }

  int read() const { return i; }

  int permute() {
         cout << "Printing from X::permute: " << i << endl;
         return i = i * 47; }
};


class Y : public X {
  int i;

public:
  Y() { i = 0; }

  int change() {
    i = permute(); // Different name call
    return i;
  }

  void set(int ii) {
    i = ii;
    X::set(ii); // Same-name function call
    cout << "This is Y::set" << endl;
  }
```

```
  void print() const {
//        cout << "X::i " << X::i << endl;
//        cout << "X::i " << read() << endl;
//        cout << "X::i " << X::read() << endl;
         cout << "Y::i " << i << endl; }
};



int main() {
/*
  cout << "sizeof(X) = " << sizeof(X) << endl;
  cout << "sizeof(Y) = "
       << sizeof(Y) << endl;
*/
  Y D;

/*
  D.change();
  D.print();
*/

/*
  D.permute();
  D.print();
*/

/*
  D.X::set(12);
  D.set(12);
  D.print();
*/
  return 0;
}
```

Ένα απλό παράδειγμα κληρονομικότητας: "Μεταπτυχιακός Φοιτητής" σαν υποκλάση του "Φοιτητή" της σελίδας 113. Πειραματιστείτε με τη συμπεριφορά των συναρτήσεων κατασκευής, βγάζοντας τα σχόλια από τη συνάρτηση main.

```
#include <iostream>
#include "/home/users/cpp/public_html/mathima13/assgn_student/student_constr.h"

using namespace std;

//////////////////////////////////////////////////////////////////////
class PGStudent : public Student {

      int gyear;

    public :

      PGStudent(const char* nam, int gy=0) : Student(nam), gyear(gy)
                { cout << "The Graduation year was :" << gyear << endl; }

      int get_year() { return gyear; }
};

int main()
{
   PGStudent pgs1("FN1");

//   PGStudent pgs2("FN2",2004);
//   PGStudent pgs3(pgs2);  // copy constructor invoked:
                            //  - copy constructor of Student is used
                            //  to copy the Student subobject
                            //  - bitwise copy for the rest

     cout << "Printing from Main:" << pgs1.get_year() << endl;
//     cout << "Printing from Main:" << pgs2.get_year() << endl;
//     cout << "Printing from Main:" << pgs3.get_year() << endl;

// Inherited functions:
     pgs1.set_no(123);
     cout << pgs1.get_no() << endl;

     return 0;
}
```

Παράδειγμα (συνέχεια): Τροποποίηση του "Μεταπτυχιακού Φοιτητή" ώστε να πειραματιστείτε με παραγόμενη κλάση που έχει μέλη δεδομένα που ανήκουν σε κλάση (κατασκευή -καταστροφή, ορατότητα).

```
#include <iostream>
#include "/home/users/cpp/public_html/mathima13/assgn_student/student_constr.h"

using namespace std;

   class Date {

        int day;
        int month;
        int year;

     public:

        Date(int d = 1, int m = 1, int y = 2005)
            {

              day = d; month = m; year = y;

              cout << "I just created a Date with value: "
                   << day << ' ' << month << ' ' << year << endl;

             }


        ~Date() {

                   cout << '\n' << "I am destroying a Date with value: "
                   << day << ' ' << month << ' ' << year << endl;
                 }
        void print() {
                   cout << day << ' ' << month << ' ' << year << endl;}

    };



/////////////////////////////////////////////////////////////////////
```

```
class PGStudent : public Student {

      Date gdate;

    public :

      PGStudent(const char* nam) : Student(nam) {}

      PGStudent(const char* nam, int gd, int gm)
               : Student(nam), gdate(gd,gm)
                 { cout << "The Graduation date was :" << endl;
                    this -> gdate.print(); }

//        Date get_date() { return gdate; }
        Date& get_date() { return gdate; }
};

int main()
{
   PGStudent pgs1("FN");

//   PGStudent pgs2(pgs1);
//   PGStudent pgs3("FN", 22, 5);

   pgs1.get_date().print(); // Careful:
                            // invokes copy-contructor and
                            // destructor of Date for the
                            // temporary return object of get_date


  cout << "Ending main" << endl;

  return 0;
}
```

Σειρά κατασκευής - καταστροφής: γενική μορφή

Κατασκευή:

```
bmcs
bc
dmcs
dc
```

Καταστροφή:

```
dd
dmds
bd
bmds
```

Όπου:

`bmcs`: constructors of base class data members (με τη σειρά που εμφανίζονται στον ορισμό της κλάσης)

`bc`: constructor of base class

`dmcs`: constructors of derived class data members (με τη σειρά που εμφανίζονται στον ορισμό της κλάσης)

`dc`: constructor of derived class

`dd`: destructor of derived class

`dmds`: destructors of derived class data members (με την αντίστροφη σειρά που εμφανίζονται στον ορισμό της κλάσης)

`bd`: destructor of base class

`bmds`: destructors of base class data members (με την αντίστροφη σειρά που εμφανίζονται στον ορισμό της κλάσης)

Πειραματιστείτε στις συναρτήσεις κατασκευής αντιγράφων κλάσεων κάτω από κληρονομικότητα και σύνθεση, σχολιάζοντας και αποσχολιάζοντας γραμμές στο παρακάτω παράδειγμα.

```
#include<iostream>
using namespace std;

// Add comments gradually to the copy constructor definitions
// of the following classes starting from the derived class
// to see copy construction initialization under inheritance
// and composition.

class A1 {
        public :
          A1() { cout << "Constructing from A1" << endl; }
          A1(const A1& a1) { cout << "Copy Constructing from A1" << endl; }
          ~A1(){ cout << "Destructing from A1" << endl; }
};

class A2 {
        public :
          A2() { cout << "Constructing from A2" << endl; }
          A2(const A2& a2) { cout << "Copy Constructing from A2" << endl; }
          ~A2(){ cout << "Destructing from A2" << endl; }
};

class A {
         A1 a1;
         A2 a2;
        public :
          A() { cout << "Constructing from A" << endl; }
//          A(const A& a) : a1(a.a1), a2(a.a2)
//             { cout << "Copy Constructing from A" << endl; }
// Interchange with the following copy constructor
          A(const A& a) { cout << "Copy Constructing from A" << endl; }
          ~A(){ cout << "Destructing from A" << endl; }
};
```

```
class B1 {
        public :
          B1() { cout << "Constructing from B1" << endl; }
          B1(const B1&  b1) { cout << "Copy Constructing from B1" << endl; }
          ~B1(){ cout << "Destructing from B1" << endl; }
};

class B : public A {
         B1 b1;
        public :
          B() { cout << "Constructing from B" << endl; }
//          B(const B& b)
// Uncomment the following lines:
//          : A(b)
//              { cout << "Copy Constructing from B" << endl; }
          ~B(){ cout << "Destructing from B" << endl; }
};

int main()
{
   B b1;
   B b2(b1);

/*
  A a1;
  A a2(a1);
*/
  return 0;
}
```

Κληρονομικότητα - Σύνθεση - Πέρασμα παραμέτρων - Επιστροφή τιμής: Κατασκευή - καταστροφή. Αιτιολογήστε τα αποτελέσματα της εκτέλεσης του παρακάτω προγράμματος.

```
#include <iostream>
using namespace std;

class A1 {
public:
        A1() { cout << "constructing an A1" << endl; }
        A1(int i) { cout << "constructing an A1 with :"
                         << i << endl; }
        ~A1() {cout << "destroying an A1" << endl; } };

class A2 {
public:
        A2() { cout << "constructing an A2" << endl; }
        A2(int i) { cout << "constructing an A2 with :"
                        << i << endl; }
        ~A2() {cout << "destroying an A2" << endl; } };

class B1 {
public:
        B1() { cout << "constructing a B1" << endl; }
        ~B1() {cout << "destroying a B1" << endl; } };

class A {
      A1 a1;
      A2 a2;
public:
        A() { cout << "constructing an A" << endl; }
        A(int i, int j) : a2(i), a1(j)
            { cout << "constructing an A" << endl; }
        ~A() {cout << "destroying an A" << endl; } };

class B : public A {
      B1 b1;
public:
        B() { cout << "constructing a B" << endl; }
        B(int i, int j) : A(i,j)
            { cout << "constructing a B" << endl; }
        ~B() {cout << "destroying a B" << endl; } };
```

```
/////////////////////////////////////////////////////////////////////
void bloo(B* b)
{       cout << "Printing in bloo" << endl; }


void bla(B b)
{       cout << "Printing in bla" << endl;
        bloo(&b); }

////////////////////////////////////////////////////////////////////
int main()
{
   B object1;
   B object2(5,15);
   cout << "First printing in main " << endl;

   bla(object1);

   bloo(&object2);

   cout << "Last printing in main " << endl;

   return 0;
 }
```

Αλήθεια, τι να σημαίνει κενή κλάση; Εκτελέστε το παρακάτω:

```
#include <iostream>
using namespace std;

class A {};

int main()
{
  A a;
  cout << sizeof(a) << endl;

  A b;
  cout << &a << '\n' << &b << endl;   // different addresses

  return 0;
}
```

Δείκτες σε αντικείμενα ιεραρχίας κλάσεων

- Δεν επιτρέπεται η ανάθεση στιγμιοτύπου (κλπ...) της βασικής κλάσης σε στιγμιότυπο (κλπ...) της παραγόμενης κλάσης διότι παραμένουν μη ορισμένα τα υπόλοιπα μέλη της παραγόμενης κλάσης.
- Αυτό μπορεί να παρακαμφθεί είτε επαναορίζοντας τον τελεστή ανάθεσης είτε/και ορίζοντας συνάρτηση μετατροπής.
- ΠΡΟΣΟΧΗ : Η χρήση δεικτών στη βασική κλάση (έστω κι αν σε αυτόν έχει ανατεθεί ένας δείκτης σε παραγόμενη κλάση) παραπέμπει στη πρόσβαση των συναρτήσεων-μελών της βασικής κλάσης. Αυτό, όμως, πολλές φορές δεν είναι επιθυμητό ($\Rightarrow$ χρήση συναρτήσεων `virtual`).
- Με ρητό `cast` μπορούμε να μετατρέψουμε έναν δείκτη σε βασική κλάση σε δείκτη σε παραγόμενη. Τότε μπορούμε να του κάνουμε dereference μόνο όταν δείχνει πράγματι σε ένα αντικείμενο της παραγόμενης κλάσης.

Τροποποίηση του "Μεταπτυχιακού Φοιτητή" ώστε να πειραματιστείτε με αναθέσεις ανάμεσα σε (δείκτες σε) αντικείμενα κλάσεων διαφορετικού επιπέδου μιας ιεραρχίας. Βγάλετε ένα -ένα τα σχόλια από τις αναθέσεις της συνάρτησης `main`. (Διευκρίνιση: το τμήμα `VARIOUS` αναφέρεται στη συμπεριφορά συναρτήσεων κατασκευής -καταστροφής μεμονωμένης κλάσης κι όχι ιεραρχίας. Απλώς, δεν είχε συζητηθεί νωρίτερα.)

```
#include <iostream>
#include "/home/users/cpp/public_html/mathima13/assgn_student/student_constr.h"

using namespace std;

//////////////////////////////////////////////////////////////////////
class PGStudent : public Student {

      int gyear;
      char* first_degree;   // We also include the
                            // first degree of PGStudent

    public :
      PGStudent(const char* nam, const char* fd, int gy=0)
                : Student(nam), gyear(gy)
                {
                  first_degree = new char[strlen(fd)+1];
                  strcpy(first_degree,fd);
                  cout << "I just constructed a PGstudent " << endl; }

     ~PGStudent()
               {
                  cout << "Deleting postgraduate student with name "
                       << get_name() << endl ;
                  delete[] first_degree;
               }

      int get_year() { return gyear; }

// Many member functions are missing but are not required
// for testing the functionality of copying
};
```

```
int main()
{
   PGStudent pgs("FN1","FDEGR");


     cout << "Printing from Main:" << endl;

//////////////////////////////////////////////////////
// VARIOUS:
//////////////////////////////////////////////////////

// Pointer to Student:
//    Student* ps = new Student("TheNew");
//    delete ps;  // otherwise the destructor of Student is not called

// Array of Student:
//   Student ss[100];   // Problem: no default constructor is
                        // provided for Student
//    Student* pss = new Student[100];   // Same as above

///////////////////////////////////////////////////////

Student s("FirstLast");

 Student* ps;
 PGStudent* ppgs;

// pgs = s;  // error
// s = pgs;  // Student redefinition of operator = is called

// ppgs = ps; // error
// ps = ppgs; // ok

return 0;
}
```

Πολυμορφισμός και συναρτήσεις virtual

- Ο πολυμορφισμός στη C++ γίνεται εμφανής μέσω της δυνατότητας να προσπελαύνουμε αντικείμενα παραγόμενων κλάσεων μέσω δεικτών (ή αναφορών) σε υπερκλάσεις.
- Μια λύση στην πρόσβαση στο σωστό ορισμό συνάρτησης -μέλους αν έχουμε δείκτες (ή αναφορές) σε υπερκλάση:
  - χρήση πεδίου τύπου στην υπερκλάση και
  - περιπτωσιολογία στις συναρτήσεις για όλες τις τιμές των πεδίων τύπου
- Η λύση στη C++: χρήση συναρτήσεων `virtual`.
- Μας βοηθούν να σχεδιάζουμε και να υλοποιούμε συστήματα που είναι εύκολα επεκτάσιμα.
- Η δήλωση μιας συνάρτησης ως `virtual` στη βασική κλάση επιτρέπει να επαναορίζουμε τη συνάρτηση στις παραγόμενες κλάσεις και να επιτυγχάνουμε την πρόσβαση στη "σωστή" συνάρτηση (ορισμό) στην κάθε περίπτωση.
- Παράδειγμα:

```
virtual void draw() const;
virtual void draw() const = 0;
```
← pure virtual

- Άπαξ και μια συνάρτηση δηλωθεί `virtual`, παραμένει `virtual` για όλη την ιεραρχία των κλάσεων που βρίσκονται κάτω από την κλάση της συνάρτησης.
- Μπορεί και να μην επαναορίζεται σε κάποια παραγόμενη κλάση.
- Αρκεί η παρουσία μιας συνάρτησης `virtual` σε μια κλάση για να διαπιστωθεί ότι τα αντικείμενα της ιεραρχίας από τη

κλάση αυτή και κάτω χρειάζονται επιπλέον πληροφορίες κατά την αναπαράστασή τους για να χρησιμοποιηθούν, αν χρειαστεί, στη φάση εκτέλεσης.

- Όταν η κλήση σε μια συνάρτηση `virtual` γίνεται μέσω του ονόματος του αντικειμένου και του τελεστή `.`, τότε η αναφορά επιλύεται σε φάση μεταγλώττισης και η συνάρτηση (ορισμός) που καλείται είναι αυτή που ορίζεται (ή κληρονομείται) στην κλάση του αντικειμένου.
- Δε μπορούμε να έχουμε `virtual` συναρτήσεις κατασκευής αντικειμένων (μπορούμε να έχουμε `virtual` συναρτήσεις καταστροφής).
- STL: δε χρησιμοποιούν `virtual` συναρτήσεις.
- ΠΡΟΣΟΧΗ: οι default τιμές συναρτήσεων `virtual` ανατίθενται σε φάση μεταγλώττισης οπότε μπορεί να μην είναι οι επιθυμητές.

- **Παραδείγματα:**

  Ένα απλό παράδειγμα χρήσης συναρτήσεων virtual. Συγκρίνετε τα αποτελέσματα της εκτέλεσης των παρακάτω δύο προγραμμάτων.

  χωρίς χρήση συναρτήσεων virtual:

```
//: C15:Instrument2.cpp
// From Thinking in C++, 2nd Edition
// Available at http://www.BruceEckel.com
// (c) Bruce Eckel 2000
// Copyright notice in Copyright.txt
// Inheritance & upcasting
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {
public:
  void play(note) const {
    cout << "Instrument::play" << endl;
  }
};
// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
  // Redefine interface function:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```

με χρήση συναρτήσεων virtual:

```
//: C15:Instrument3.cpp
// From Thinking in C++, 2nd Edition
// Available at http://www.BruceEckel.com
// (c) Bruce Eckel 2000
// Copyright notice in Copyright.txt
// Late binding with the virtual keyword
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
};
// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
  // Override interface function:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```

Άλλο παράδειγμα:

```
#include <iostream>
using namespace std;

class I{
public:
  virtual void play() const { // interchange with next header
//  void play() const {
    cout << "I::play" << endl;
  }
};

class W: public I{
public:
  void play() const {
    cout << "W::play" << endl;
  }
  void bla() {};
};


int main() {

  W flute;
  flute.play();

  I* pflute = &flute;
  pflute->play();
  pflute->bla();

  return 0;
} ///:~
```

Ας θυμηθούμε τον "Μεταπτυχιακό Φοιτητή" της σελίδας 161.

```
#include <iostream>
#include "/home/users/cpp/public_html/mathima13/assgn_student/student_constr.h"

using namespace std;

//////////////////////////////////////////////////////////////////////
class PGStudent : public Student {

      int gyear;
      char* first_degree;   // We also include the
                            // first degree of PGStudent

    public :
      PGStudent(const char* nam, const char* fd, int gy=0)
                : Student(nam), gyear(gy)
                {
                  first_degree = new char[strlen(fd)+1];
                  strcpy(first_degree,fd);
                  cout << "I just constructed a PGstudent " << endl; }

     ~PGStudent()
               {
                  cout << "Deleting postgraduate student with name "
                       << get_name() << endl ;
                  delete[] first_degree;
               }

      int get_year() { return gyear; }

// Many member functions are missing but are not required
// for testing the functionality of copying
};
```

Ας αναθεωρήσουμε τώρα τον ορισμό της κλάσης `Student` της σελίδας 113 προσθέτοντας και μια συνάρτηση `print`.

student_constr.h:

```
class Student {
private:
        char* name;
        int no;
        int year_of_studies;
public:
        Student(const char*);
        Student(const Student&);
       ~Student();

        Student& operator=(const Student& s);

        void set_name(const char*);
        void set_no(int);
        void set_year_of_studies(int);

        char* get_name();
        int get_no();
        int get_year_of_studies();

        void print();
//        virtual void print();
};
```

student_constr.cc:

```
#include <iostream>
#include "student_constr.h"
using namespace std;

void Student::print()
{
  cout << "Student's name is: " << name << endl;
}
```

```
/////////////////////////////////////////////////////////
void Student::set_name(const char* nam)
{
  delete[] name;
  name = new char[strlen(nam)+1];
  strcpy(name,nam);
}

void Student::set_no(int n)
{
        no = n;
}

void Student::set_year_of_studies(int y)
{
        year_of_studies = y;
}



/////////////////////////////////////////////////////////
char* Student::get_name()
{
        return name;
}

int Student::get_no()
{
        return no;
}

int Student::get_year_of_studies()
{
        return year_of_studies;
}
/////////////////////////////////////////////////////////

Student::~Student()
{
   cout << "Deleting student with name " << name << endl ;
   delete[] name;
}

/////////////////////////////////////////////////////////
```

```
Student::Student(const char* nam)
{
   name = new char[strlen(nam)+1];
   strcpy(name,nam);
   cout << "I just constructed a student " << endl;
}


Student::Student(const Student& s)
{
   name = new char[strlen(s.name)+1];
   strcpy(name,s.name);
//   name = s.name;     Careful: this is the result of default copy
   no = s.no;
   year_of_studies = s.year_of_studies;
   cout << "I just created a student by copying ... " << endl;
}


//////////////////////////////////////////////////////////

Student& Student::operator=(const Student& s)
{

   //  name = s.name; Careful: this is the result of default assignment

   if (this != &s) // Careful not to assign to itself, especially
                   // if some storage reclaim was made within the body
      {
        delete[] name;
        name = new char[strlen(s.name)+1];
        strcpy(name,s.name);
        no = s.no;
        year_of_studies = s.year_of_studies;
        cout << "I just performed a student ASSIGNMENT ... " << endl;
      }
   return *this;

}
```

Ας επαυξήσουμε τον ορισμό της κλάσης PGStudent ώστε να επανορίζει τη συνάρτηση -μέλος print. Πειραματιστείτε βγάζοντας τα σχόλια από τη συνάρτηση main του παρακάτω προγράμματος, βγάζοντας ταυτόχρονα και τα σχόλια από τον ορισμό της κλάσης Student. Σε κάθε περίπτωση, ο destructor της κλάσης Student πρέπει να έχει δηλωθεί virtual για να έχουμε σωστή αποδέσμευση κατά την καταστροφή του δείκτη σε Student που στην πραγματικότητα δείχνει σε αντικείμενο της PGStudent.

```
#include <iostream>
#include "student_constr.h"
using namespace std;

/////////////////////////////////////////////////////////////////////
class PGStudent : public Student {

      int gyear;
      char* first_degree;

    public :
      PGStudent(const char* nam, const char* fd, int gy=0)
                : Student(nam), gyear(gy)
                {
                  first_degree = new char[strlen(fd)+1];
                  strcpy(first_degree,fd);
                  cout << "I just constructed a PGstudent " << endl; }
     ~PGStudent()
               {
                  cout << "Deleting postgraduate student with name "
                       << get_name() << endl ;
                  delete[] first_degree;
               }
      int get_year() { return gyear; }
      void print()
               { Student::print();
                 cout << "First degree is: " << first_degree << endl; }
};
```

```
int main()
{
   PGStudent pgs("FN1","FDEGR");

//   Student* ps= new PGStudent("FN2","FDEGR2");
//   delete ps;

     cout << "Printing from Main:" << endl;

//   Student s("FirstLast");


//  s.print();
//  pgs.print();
//  pgs.Student::print();

/////////////////////////////////////////////
//   Student* ps=&s;
//   PGStudent* ppgs=&pgs;


//  ps->print();
//  ppgs->print();
//  ppgs->Student::print();

/////////////////////////////////////////////
// Student* ss[2];

//  ss[0] = &s;
//  ss[1] = &pgs;


//  ss[0]->print();
//  ss[1]->print();

//  ss[0] = new Student("TestedSt");
//  ss[1] = new PGStudent("TestPSt","FDeg");

//  delete ss[0];
//  delete ss[1];


/////////////////////////////////////////////
```

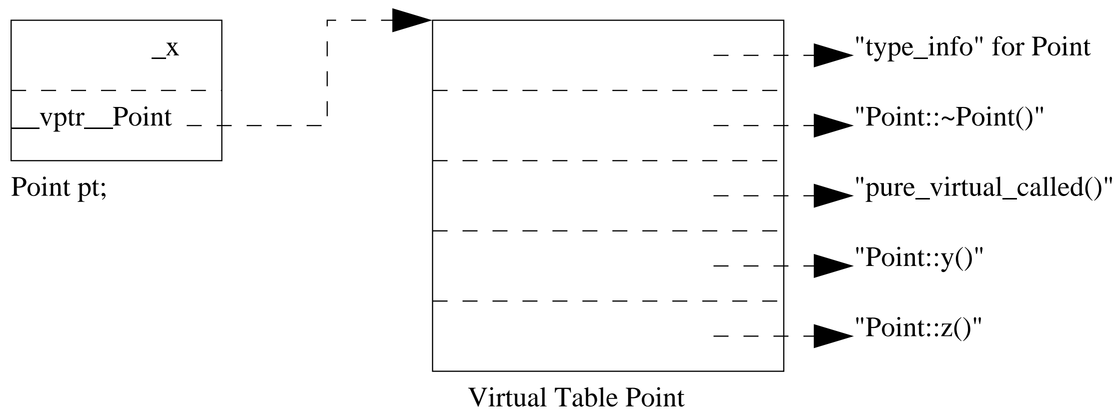
```
// Student* p1 = new Student("TestSt");
// Student* p2 = new PGStudent("TestPSt","FDeg");

// delete p1;
// delete p2;

   return 0;
}
```

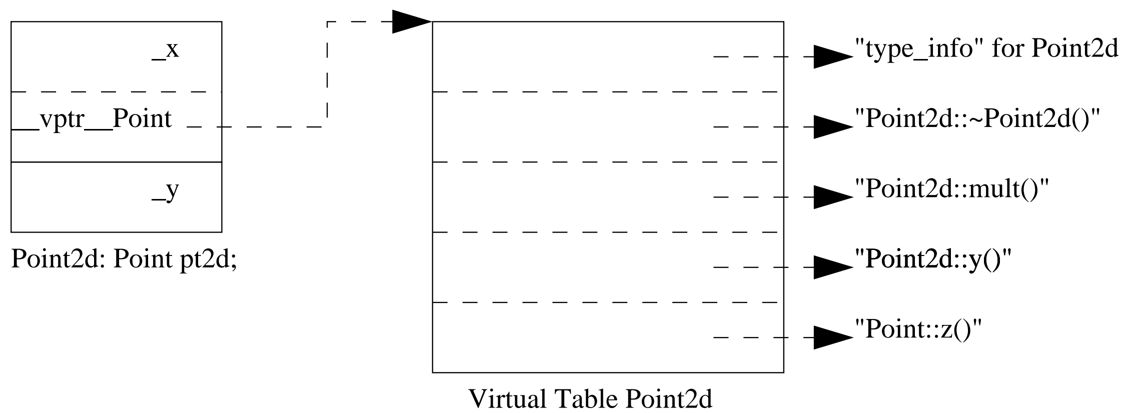
Αναπαράσταση: πίνακας συναρτήσεων virtual

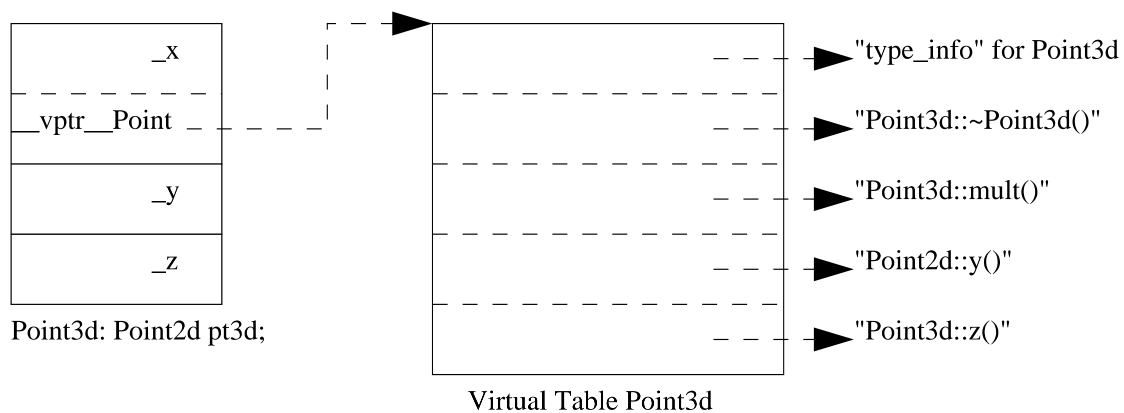
- Όταν μια κλάση έχει συναρτήσεις `virtual` αποκτά, από το μεταγλωττιστή, έναν "πίνακα συναρτήσεων `virtual`" και τα αντικείμενά της αποκτούν ένα δείκτη στον πίνακα αυτό.
- Κάθε κλάση της ιεραρχίας από την πρώτη εμφάνιση συνάρτησης `virtual` και κάτω αποκτά το δικό της πίνακα συναρτήσεων `virtual`.
- Για την κάθε συνάρτηση `virtual` μιας κλάσης, έχει καταχωρηθεί ένας δείκτης πίνακα που αντιστοιχεί στη θέση του πίνακα συναρτήσεων `virtual` που οδηγεί στον κατάλληλο ορισμό της συνάρτησης για την κλάση αυτή.
- Ο δείκτης πίνακα που αντιστοιχεί σε μια συνάρτηση `virtual` παραμένει ο ίδιος για όλη την ιεραρχία.

Αναπαράσταση: παράδειγμα (προσαρμογή από [Lippman2])

```
class Point {
public:
   virtual ~Point();
   virtual Point& mult( float ) = 0;
// ...other operations...
   float x() const { return _x; }
   virtual float y() const { return 0; }
   virtual float z() const { return 0; }
// ...
protected:
   Point( float x = 0.0 );
   float _x;
};

class Point2d : public Point {
public:
    Point2d( float x = 0.0, float y = 0.0 ) :
      Point( x ), _y( y ) {}
    ~Point2d();
// overridden base class virtual functions
    Point2d& mult( float ) ;
    float y() const { return _y; }
// ...other operations...
protected:
   float _y;
};

class Point3d : public Point2d {
public:
    Point3d( float x = 0.0, float y = 0.0,
             float z = 0.0 ) :
      Point2d( x, y ), _z( z ) {}
    ~Point3d();
// overridden base class virtual functions
    Point3d& mult( float ) ;
    float z() const { return _z; }
// ...other operations...
protected:
   float _z;
};
```







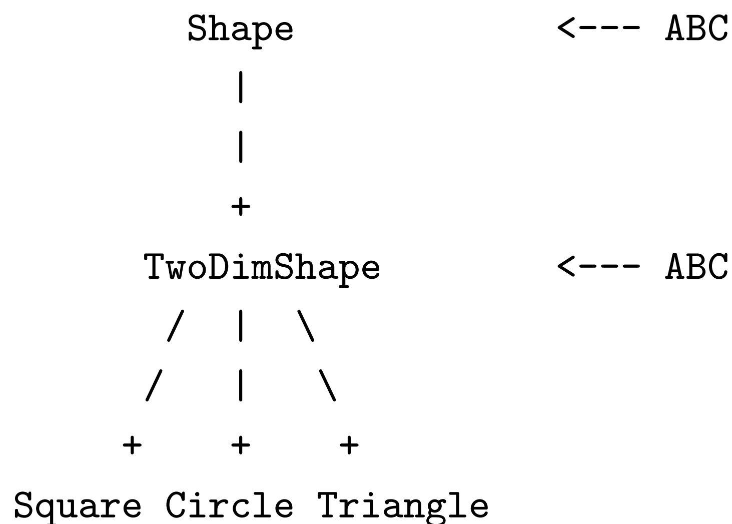

```
Point *pp;
   ........
pp->z(); => (*pp->__vptr__Point[4])(pp);
```

- Default τιμές σε συναρτήσεις `virtual`: προβληματιστείτε με το παρακάτω

```
#include<iostream>
using namespace std;

class Base {
public:
  virtual int foo(int ival =1024)
          { cout << "Base::foo() returns :" << ival << endl;
            return ival;
          }
};

class Derived : public Base {
public:
  virtual int foo(int ival =2048)
          { cout << "Derived::foo() returns :" << ival << endl;
            return ival;
          }
};

int main()
   {
    Base b;
    Base * pb = &b;

    b.foo();
    pb->foo();

    Derived d;
    pb = &d;

    d.foo();
    pb->foo();

    return 0;
   }
```

Αφηρημένες κλάσεις (Abstract base classes) Όταν μια κλάση έχει μια (ή περισσότερες) πλήρως εικονικές (pure virtual) συναρτήσεις, τότε:

- δεν μπορούμε να ορίσουμε αντικείμενα αυτών των κλάσεων
- αυτές οι κλάσεις λέγονται *Αφηρημένες Κλάσεις* (Abstract Base Classes)
- αυτές οι κλάσεις χρησιμεύουν μόνο για κληρονομικότητα υπογραφών (signatures) και υλοποίησης.
- Παράδειγμα



Η κλάση `Shape`:

```
class Shape
{
 // ...
 virtual void print() const = 0;
 // ...
};
```

- Αν μια παραγόμενη κλάση δεν ορίζει μια pure virtual συνάρτηση, τότε και η κλάση αυτή είναι αφηρημένη.
- Παρόλο που δεν μπορούμε να ορίσουμε αντικείμενα μιας αφηρημένης κλάσης μπορούμε να ορίσουμε δείκτες σε αφηρημένες κλάσεις και αναφορές σε αφηρημένες κλάσεις για να πετύχουμε πολυμορφικό χειρισμό σε αντικείμενα παραγόμενων κλάσεων.

Το παράδειγμα της σελίδας 167 με χρήση πλήρως εικονικών συναρτήσεων - αφηρημένης κλάσης. Αποσχολιάστε κατάλληλα τον παρακάτω κώδικα ώστε να πειραματιστείτε με τα εναλλακτικά:

- Η συνάρτηση `play()` να είναι πλήρως εικονική συνάρτηση - η κλάση I να είναι πλήρως αφηρημένη.
- Η συνάρτηση `play()` να είναι εικονική συνάρτηση.
- Η συνάρτηση `play()` να είναι απλή συνάρτηση μέλος.

```
#include <iostream>
using namespace std;

class I{
public:
 virtual void play() const = 0;
// virtual void play() const
// void play() const
// {
//    cout << "I::play" << endl;
//  }
};

class W: public I{
public:
  void play() const {
    cout << "W::play" << endl;
  }
};
```

```
int main() {

  I instr;
//instr.play();

  W flute;
  flute.play();

  I* pflute = &flute;
  pflute->play();
//  pflute->W::play();

  return 0;
} ///:~
```

Ας αναθεωρήσουμε τώρα τον ορισμό της κλάσης `Student` της σελίδας 169, προσθέτοντας και μια συνάρτηση `print` που να είναι pure virtual. Κάντε ανάλογους πειραματισμούς, χρησιμοποιώντας τον κώδικα των σελίδων 169 έως 173, αφαιρώντας κάθε φορά τα κατάλληλα σχόλια.

student_constr.h:

```
class Student {
private:
        char* name;
        int no;
        int year_of_studies;
public:
        Student(const char*);
        Student(const Student&);
       ~Student();
//        virtual ~Student();

        Student& operator=(const Student& s);

        void set_name(const char*);
        void set_no(int);
        void set_year_of_studies(int);

        char* get_name();
        int get_no();
        int get_year_of_studies();

        void print();
//        virtual void print();
//        virtual void print() = 0;
};
```

Κληρονομικότητα και εικονικές συναρτήσεις: το παράδειγμα ολοκληρωμένο.

```
// File:  student_constr.h
class Student {
private:
        char* name;
        int no;
        int year_of_studies;
public:
        Student(const char*);
        Student(const Student&);
       ~Student();
//        virtual ~Student();

        Student& operator=(const Student& s);

        void set_name(const char*);
        void set_no(int);
        void set_year_of_studies(int);

        char* get_name();
        int get_no();
        int get_year_of_studies();

        void print();
//        virtual void print();
//        virtual void print() = 0;
};
```

```
// File: student_constr.cc
#include <cstring>
#include <iostream>
#include "student_constr.h"

using namespace std;

void Student::print()
{
  cout << "Student's name is: " << name << endl;
}

//////////////////////////////////////////////////////////
void Student::set_name(const char* nam)
{
  delete[] name;
  name = new char[strlen(nam)+1];
  strcpy(name,nam);
}

void Student::set_no(int n)
{
        no = n;
}

void Student::set_year_of_studies(int y)
{
        year_of_studies = y;
}
```

```
///////////////////////////////////////////////////////////
char* Student::get_name()
{
        return name;
}

int Student::get_no()
{
        return no;
}

int Student::get_year_of_studies()
{
        return year_of_studies;
}

///////////////////////////////////////////////////////////
Student::~Student()
{
   cout << "Deleting student with name " << name << endl ;
   delete[] name;
}


///////////////////////////////////////////////////////////
Student::Student(const char* nam)
{
   name = new char[strlen(nam)+1];
   strcpy(name,nam);
   cout << "I just constructed a student " << endl;
}
```

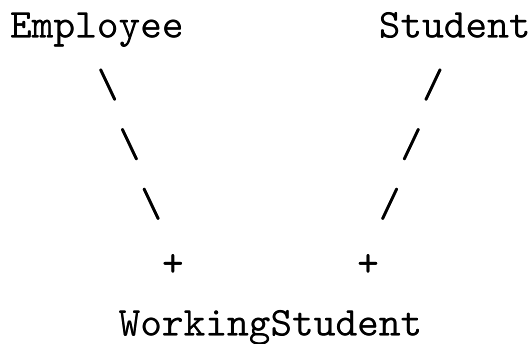
```
Student::Student(const Student& s)
{
   name = new char[strlen(s.name)+1];
   strcpy(name,s.name);
// Careful: the following is the result of default copy
//   name = s.name;
   no = s.no;
   year_of_studies = s.year_of_studies;
   cout << "I just created a student by copying ... " << endl;
}

//////////////////////////////////////////////////////////
Student& Student::operator=(const Student& s)
{
   // Careful: the following is the result of default assignment
   //  name = s.name;

   if (this != &s) // Careful not to assign to itself, especially
              // if some storage reclaim was made within the body
      {
        delete[] name;
        name = new char[strlen(s.name)+1];
        strcpy(name,s.name);
        no = s.no;
        year_of_studies = s.year_of_studies;
        cout << "I just performed a student ASSIGNMENT ... "
             << endl;
      }
   return *this;

}
```

```
#include <iostream>
#include "student_constr.h"

using namespace std;

//////////////////////////////////////////////////////////

class PGStudent : public Student {

      int gyear;
      char* first_degree;

    public :

      PGStudent(const char* nam, const char* fd, int gy=0)
                : Student(nam), gyear(gy)
                {
                  first_degree = new char[strlen(fd)+1];
                  strcpy(first_degree,fd);
                  cout << "I just constructed a PGstudent "
                       << endl; }

     ~PGStudent()
               {
                  cout <<
                       "Deleting postgraduate student with name "
                       << get_name() << endl ;
                  delete[] first_degree;
               }
```

```
      int get_year() { return gyear; }

      void print()
               { Student::print();
                 cout << "First degree is: " << first_degree
                      << endl; }
};

int main()
{
   PGStudent pgs("FN1","FDEGR");

//   Student* ps= new PGStudent("FN2","FDEGR2");
//   delete ps;

     cout << "Printing from Main:" << endl;

//   Student s("FirstLast");


//  s.print();
//  pgs.print();
//  pgs.Student::print();

/////////////////////////////////////////////
//   Student* ps=&s;
//   PGStudent* ppgs=&pgs;


//  ps->print();
//  ppgs->print();
//  ppgs->Student::print();
```

```
/////////////////////////////////////////////
// Student* ss[2];

//  ss[0] = &s;
//  ss[1] = &pgs;


//  ss[0]->print();
//  ss[1]->print();

//  ss[0] = new Student("TestedSt");
//  ss[1] = new PGStudent("TestPSt","FDeg");

//  delete ss[0];
//  delete ss[1];


/////////////////////////////////////////////

// Student* p1 = new Student("TestSt");
// Student* p2 = new PGStudent("TestPSt","FDeg");

// delete p1;
// delete p2;

   return 0;
}
```

Πολλαπλή Κληρονομικότητα




```
class WorkingStudent : public Student, public Employee
{
 // ...
};
```

- Μια κλάση μπορεί να παράγεται από παραπάνω από μια βασικές κλάσεις άμεσα.
- Σ' αυτήν την περίπτωση, κληρονομεί τα μέλη από όλες τις κλάσεις.
- Σε περίπτωση σύγκρουσης ονομάτων, γίνεται χρήση του τελεστή `::`

ΠΡΟΤΥΠΑ (templates)

- Συναρτήσεων
- Κλάσεων

Πρότυπα Συναρτήσεων

- Γενική μορφή:

```
template< class T >
```

- Π.χ. αντί,

```
void printArray(const int* array, const int count);
void printArray(const double* array, const int count);
// ...
```

κάνουμε

```
template < class T >
void printArray(const T* array, const int count)
{
   for (int i=0; i < count; i++)
        cout << array[i] << " ";
   cout << endl;
}
```

- τότε πρέπει να κάνουμε:

```
int main()
   {
      const int aCount=5, bCount=7, cCount=6;
      int a[aCount];
      double b[bCount];
      char c[cCount];
   // ...
      printArray(a, aCount);
      printArray(b, bCount);
      printArray(c, cCount);

      return 0;
   }
```

Παρατηρήσεις στα πρότυπα συναρτήσεων

- Επιτυγχάνουν πολυμορφισμό.
- Τα πρότυπα επιτρέπουν να ορίσουμε ένα σύνολο σχετιζόμενων συναρτήσεων (ή ένα σύνολο από σχετιζόμενες κλάσεις).
- Γράφουμε έναν ορισμό για τη συνάρτηση, περιλαμβάνοντας παραμέτρους τύπων.
- Κατά τη μεταγλώττιση, ανάλογα με τους τύπους που εμφανίζονται στην κλήση μιας τέτοιας συνάρτησης, παράγεται κώδικας για την κάθε μια περίπτωση
- Παράμετροι τύπων μπορούν να εμφανίζονται:
  - στα τυπικά ορίσματα της συνάρτησης
  - στον τύπο που επιστρέφει η συνάρτηση
  - μέσα στο σώμα της συνάρτησης
- Οι παράμετροι τύπων πρέπει να είναι μοναδικοί στα πρότυπα.

Παράδειγμα:

utilities.h :

```
#ifndef UTILITIES_H
#define UTILITIES_H

template <typename T> int smaller(const T&, const T&);

#include "utilities.cc"
#endif
```

utilities.cc :

```
template <typename T> int smaller(const T& v1, const T& v2)
{

  if (v1 < v2) return 1;
  if (v2 < v1) return 2;

  return 0;

}
```

main.cc :

```
#include <iostream>

#include "utilities.h"

using namespace std;

int main()
{

  cout << smaller(2,3) << endl;
// error: in the following,
// no type conversions int to double can be made in this case
//  cout << smaller(2,0.3) << endl;
  cout << smaller(2.5,0.3) << endl;
  cout << smaller('a','a') << endl;

  return 0;
}
```

Πρότυπα Κλάσεων

- Παραμετρικοί τύποι.
- Γενικοί ορισμοί κλάσεων με συγκεκριμενοποίηση (όχι εξειδίκευση) με βάση συγκεκριμένους τύπους.
- Στην επικεφαλίδα μιας κλάσης μπορούν να χρησιμοποιηθούν και παράμετροι που δεν είναι τύποι, π.χ.

```
template<class T, int elements>
class Stack
{
 // ...
}
```

  τότε,

```
Stack<double,100>
```

- Μπορεί να οριστεί μια κλάση για ένα συγκεκριμένο τύπο.
- Ζήτημα: κληρονομικότητα και πρότυπα κλάσεων.

Παράδειγμα: ορισμός ενός προτύπου κλάσης, των συναρτήσεων μελών της και χρήση της

Ορισμός του προτύπου:

```
template < class T >
class Stack
{
 public:
   Stack(int=10);
   ~Stack() { delete [] stackPtr; }
   bool push(const T&);
   bool pop(T&);

 private:
    int size;
    int top;
    T* stackPtr;

    bool isEmpty() const { return top == -1; }
    bool isFull() const { return top == size - 1; }
};
```

Ορισμός συναρτήσεων -μελών:

```
template < class T >
   Stack<T>::Stack(int s)
   {
      size = s>0?s:10;
      top = -1;
      stackPtr = new T[size];
   }

   template < class T >
   bool Stack<T>::push(const T& pushValue)
   {
     if (!isFull())
     {
         stackPtr[++top] = pushValue;
         return true;
     }
     return false;
   }
// ...
```

Χρήση του προτύπου:

```
int main()
   {
      Stack<double> dStack(5); //instantiation
      Stack<int> iStack(20);   //instantiation
      return 0;
   }
```

Πλήρες παράδειγμα:

StackTemplate.h

```
// Simple stack template
#ifndef STACKTEMPLATE_H
#define STACKTEMPLATE_H

template<class T>
class StackTemplate {
  static const int ssize = 100;
  T stack[ssize];
  int top;
public:
  StackTemplate() : top(0) {}
  void push(const T& i) {
    stack[top++] = i;
  }
  T pop() {
    return stack[--top];
  }
  int size() { return top; }
};
#endif // STACKTEMPLATE_H ///:~
```

StackTemplateTest.cc

```
// Adopted from :From Thinking in C++, 2nd Edition
// Test simple stack template
#include "StackTemplate.h"
#include <iostream>
using namespace std;

int main() {

  StackTemplate<int> is;

  for(int i = 0; i < 20; i++)
    is.push(i);
// add - remove comments to the following loop, to check  whether
//  the assembly code for pop() is produced
  for(int k = 0; k < 20; k++)
    cout << is.pop() << endl;

} ///:~
```

Γενικές παρατηρήσεις στα πρότυπα

- Παράγεται κώδικας μόνο για τις συναρτήσεις που καλούνται.
- Εμβέλεια: επίλυση νοήματος στην εμβέλεια του ορισμού ενός προτύπου ή της χρήσης του;

ΕΙΣΟΔΟΣ - ΕΞΟΔΟΣ στη C++

- `<iostream>`: αρχείο-επικεφαλίδα για τις περισσότερες λειτουργίες `I/O` της C++ (διεπαφή με τη βιβλιοθήκη για το χειρισμό ρευμάτων, streams)
- Αντικειμενοστραφής φιλοσοφία I/O στη C++
- Μορφοποιημένη I/O
- I/O χαρακτήρων
- I/O πάνω σε αρχεία
- I/O in RAM

Κριτήρια για το σχεδιασμό του συστήματος I/O της C++

- Στις "παραδοσιακές" γλώσσες προγραμματισμού, το σύστημα I/O είχε να υποστηρίξει ένα πεπερασμένο σύνολο από ενσωματωμένους τύπους.
- Στη C++ το σύνολο των τύπων είναι επεκτάσιμο από τους τύπους που ορίζει ο χρήστης.
- Το σύστημα I/O της C++ είναι εύκολο στη χρήση, πλήρες, αποτελεσματικό ΑΛΛΑ ΚΑΙ ασφαλές στο χειρισμό τύπων και ευέλικτο.

Ρεύματα

- Το σύστημα I/O της C++ λειτουργεί μέσω ρευμάτων (streams).
- Ένα ρεύμα είναι μια λογική συσκευή που είτε παράγει είτε καταναλώνει πληροφορία.
- Ένα ρεύμα συνδέεται με μια φυσική συσκευή, μέσα από το σύστημα I/O της C++.
- Τα ρεύματα, ουσιαστικά, είναι στιγμιότυπα των κλάσεων `istream`, `ostream`, `iostream` -ή, όπως θα δούμε παρακάτω, υποκλάσεών τους -.
- Όταν ένα πρόγραμμα C++ ξεκινά την εκτέλεσή του, "ανοίγουν" τα ενσωματωμένα ρεύματα:

`cin`:	προκαθορισμένη είσοδος (standard input)
`cout`:	προκαθορισμένη έξοδος (standard output)
`cerr,clog`:	προκαθορισμένη έξοδος σφαλμάτων

  Το πρώτο αντιστοιχεί στο πληκτρολόγιο και τα άλλα δύο στην οθόνη εκτός αν έχει γίνει ανακατεύθυνση.

Είσοδος/Εξοδος Μεταβλητών

Έξοδος μεταβλητών : με χρήση του τελεστή <<

- Γενική μορφή: <ΡεύμαΕξόδου> << <Μεταβλητή>
- Π.χ.

```
int i = 10;
cout << i;
```

Είσοδος μεταβλητών : με χρήση του τελεστή >>

- Γενική μορφή: <ΡεύμαΕισόδου> >> <Μεταβλητή>
- Π.χ.

```
float f ;
cin >> f;
```

- Στην έκφραση εισόδου ή εξόδου, δε δίνεται καμιά πληροφορία για τον τύπο της μεταβλητής. Το επιθυμητό αποτέλεσμα επιτυγχάνεται λόγω των υπερφορτωμένων ορισμών των τελεστών << και >>.
- Δεδομένου ότι οι τελεστές << και >> επιστρέφουν ρεύμα εξόδου και ρεύμα εισόδου, αντίστοιχα, μπορούμε να τους χρησιμοποιήσουμε περισσότερες από μια φορά σε μια έκφραση.

Παραδείγματα

```
#include <iostream>
using namespace std;

int main()
{
  int i;
  cin >> i;

  float f;
  cin >> f;

  char c;
  cin >> c;

  char buf[10];
  cin >> buf;

  cout << "i = " << i << endl;
  cout << "f = " << f << endl;
  cout << "c = " << c << endl;
  cout << "buf = " << buf << endl;

  return 0;
}
```

Το ίδιο επιτυγχάνουμε με το παρακάτω:

```
#include <iostream>
using namespace std;

int main()
{
  int i; float f;
  char c; char buf[10];

  cin >> i >> f >> c >> buf;

  cout << "i = " << i << endl;
  cout << "f = " << f << endl;
  cout << "c = " << c << endl;
  cout << "buf = " << buf << endl;

  return 0;
}
```

Μορφοποίηση Εισόδου/Εξόδου Μπορεί να γίνει με δύο τρόπους:

1. Με χρήση συναρτήσεων-μελών της κλάσης `ios` και χειρισμό σημαιών (flags).
2. Με χρήση "διαμορφωτών" (manipulators).

Διαμορφωτές (manipulators)

- Για να γράφουμε στην ίδια γραμμή εισόδου ή εξόδου τις λειτουργίες που θέλουμε να γίνουν.
- Ορισμός συναρτήσεων που λέγονται διαμορφωτές και μπορούν να περιληφθούν στις εκφράσεις I/O.
- Γενική μορφή:
  `ostream&` $< \textit{manip_name} >$(`ostream&` $< \textit{stream} >$)
  `istream&` $< \textit{manip_name} >$(`istream&` $< \textit{stream} >$)
  `ostream&` $< \textit{manip_name} >$(`ostream&` $< \textit{stream} >$,
  $< \textit{type param} >$)
  ...
- Μπορούν να οριστούν και από το χρήστη (λόγω υποστήριξης υπερφόρτωσης)

Ενσωματωμένοι Διαμορφωτές

```
dec: input/output data in decimal (input/output)
hex: input/output data in hexadecimal (input/output)
oct: input/output data in octal (input/output)
endl: output nl and flush the stream (output)
ends: output null character '\0' (output)
flush: flush the stream (output)
ws: skip leading whitespace (input)
resetiosflags(long f): turn off flags specified in "f"
                       (input/output)
setbase(int base): set the number base to "base" (output)
setfill(int ch): set fill character to "ch" (output)
setiosflags(long f): turn on the flags specified in "f"
                    (input/output)
setprecision(int p): set the number of digits of precision
                    (output)
setw(int w): set the field width to "w" (output)
...
```

Παρατηρήσεις:

- Για τους διαμορφωτές που δεν έχουν ορίσματα δεν πρέπει να βάζουμε παρενθέσεις.
- Για να χρησιμοποιήσουμε τους διαμορφωτές που έχουν ορίσματα πρέπει να περιλάβουμε το αρχείο -επικεφαλίδα `<iomanip>`.

Είσοδος/Έξοδος σε Επίπεδο Χαρακτήρων

Είσοδος (μέσω συναρτήσεων-μελών της istream)

- `istream& get(char& ch)`: διαβάζει έναν χαρακτήρα και τον τοποθετεί στο ch. Π.χ. `char ch; cin.get(ch);` Στην περίπτωση που φτάσουμε στο τέλος του αρχείου, η κλήση της get επιστρέφει τιμή (που αντιστοιχεί σε) false.
- `istream& get(char* buf, int num, char delim='\n')`: διαβάζει χαρακτήρες και τους τοποθετεί στον πίνακα στον οποίο δείχνει η buf έως ότου είτε διαβαστούν num-1 χαρακτήρες είτε φτάσουμε σε delim. Ο πίνακας στον οποίο δείχνει η buf θα τερματιστεί με null. Το delim δεν αφαιρείται από το ρεύμα (θα το επεξεργαστεί η επόμενη διαδικασία εισόδου)
- `int get()`: επιστρέφει τον επόμενο χαρακτήρα από το ρεύμα εισόδου (επιστρέφει EOF αν έφτασε στο τέλος του αρχείου)
- `bool eof()`: για έλεγχο αν φτάσαμε στο τέλος του αρχείου
- `istream& getline(char* buf, int num, char delim='\n')`: σαν την αντίστοιχη get εκτός του ότι το delim διαβάζεται και αφαιρείται από το ρεύμα εισόδου.
- `istream& read(unsigned char* buf, int num)`: για να διαβάσουμε ένα μπλοκ num χαρακτήρων και να το βάλουμε στον πίνακα που δείχνει η buf.
- Για ανάγνωση χαρακτήρων αντίστοιχη με την `read()`, μπορεί να χρησιμοποιηθεί η `ignore()`. Σε αυτήν την περίπτωση, οι χαρακτήρες που διαβάστηκαν δεν

φυλάσσονται πουθενά.

Έξοδος (μέσω συναρτήσεων -μελών της ostream)

- ostream& put(char ch): γράφει το χαρακτήρα ch στο ρεύμα εξόδου.
- ostream& write(const unsigned char* buf, int num): γράφει num χαρακτήρες στο ρεύμα εξόδου από τον πίνακα στον οποίο δείχνει η buf.

Έλεγχος προβλημάτων για οποιοδήποτε στιγμιότυπο της iostream: good(), bad(), eof(), fail() (βασίζονται στα eofbit, failbit, badbit που είναι ορισμένα στην ios και ελέγχουν την κατάσταση των ρευμάτων)

Έλεγχος της Κατάστασης Ρεύματος

Ανά πάσα στιγμή, κάθε ρεύμα βρίσκεται σε μια "κατάσταση" (state). Η κατάσταση αυτή μπορεί να ελεγχθεί μέσω των παρακάτω συναρτήσεων-μελών της ios:
bool good() const;: η επόμενη πράξη ενδέχεται να πετύχει
bool eof() const;: η είσοδος έφτασε στο τέλος
bool fail() const;: η επομένη πράξη θα αποτύχει
bool bad() const;: το ρεύμα έχει χαλάσει
Όταν ένα ρεύμα χρησιμοποιείται σε μία συνθήκη, ουσιαστικά, ελέγχεται η κατάστασή του με βάση κλήση στην fail().

Είσοδος/Έξοδος πάνω σε Αρχεία

- `<fstream>`: αρχείο -επικεφαλίδα που πρέπει να περιλάβουμε αν θέλουμε να κάνουμε είσοδο ή έξοδο από και προς αρχεία.
- Ορίζουν διάφορες κλάσεις μεταξύ των οποίων και τις:
  `ifstream`: για είσοδο
  `ofstream`: για έξοδο
  `fstream`: για είσοδο και έξοδο
  σαν υποκλάσεις των κλάσεων `istream` και `ostream`.
- ό,τι μπορούμε να κάνουμε σε `istream` και `ostream` μπορούμε να κάνουμε και σε `ifstream` και `ofstream` και μάλιστα
- ό,τι μπορούμε να κάνουμε και από την `ios` μπορούμε να το εφαρμόσουμε και στα αρχεία.

Ανοιγμα/Κλείσιμο αρχείων

- Ανοίγουμε ένα αρχείο συνδέοντάς το με ένα ρεύμα.
- Τα ρεύματα είναι στιγμιότυπα των κλάσεων `ifstream`, `ofstream`, `fstream`.
- `void open(const char* filename, int mode, int access=filebuf::openprot)`, όπου
  - `filename`: το όνομα του αρχείου που ανοίγουμε
  - `mode`: μια από τις `ios::app`, `ios::ate`, `ios::binary`, `ios::in`, `ios::out`, `ios::trunc` (μπορούν να συνδυαστούν με OR)
  - `access`, αρκετά χαμηλού επιπέδου, σχεδόν πάντα παίρνει τη default τιμή

- Αν αποτύχει η `open`, το ρεύμα παίρνει την τιμή μηδέν (κλασικός τρόπος ελέγχου αν όλα πήγαν καλά με το άνοιγμα).

- Για να κλείσουμε ένα αρχείο χρησιμοποιούμε `close()` , π.χ.

  ```
  datastr.close();
  ```

  όμως, δεν είναι υποχρεωτικό να το κάνουμε ρητά μια που αυτό γίνεται αυτόματα από τη συνάρτηση καταστροφής όταν βγαίνουμε από την εμβέλεια του ρεύματος.

- Επειδή οι κλάσεις `ofstream`, `ifstream`, `fstream` έχουν συναρτήσεις κατασκευής στιγμιοτύπων που καλούν την `open`, τις περισσότερες φορές δεν είναι ανάγκη να ανοίγουμε τα αρχεία ρητά. Αρκεί, π.χ.

  ```
  ifstream datastr("DATAFILE");
  ```

  με αυτό ταυτόχρονα κάναμε:

  – δήλωση για ρεύμα `datastr`

  – σύνδεση με το αρχείο `DATAFILE`

  – άνοιγμα για διάβασμα

- Πρόσθετες δυνατότητες:

  – τυχαία προσπέλαση

  – χειρισμός/φύλαξη τρέχουσας θέσης

  – ...

Παραδείγματα:

- Έξοδος μεταβλητών σε αρχείο

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   ofstream out("test");
   if (!out) {
      cout << "Cannot open file.\n";
      return 1;
   }

   out << 10 << " " << 123.23 << "\n";
   out << "This is a short text file.\n";

   out.close();
   return 0;
}
```

- Είσοδος μεταβλητών από αρχείο

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   char ch; int i; float f; char str[80];

   ifstream in("test");
   if(!in) {
      cout << "Cannot open file.\n";
      return 1;
   }
   in >> i >> f >> ch >> str;

   cout << i << " " << f << " " << ch << endl;
   cout << str << endl;

   in.close();
   return 0;
}
```

- Κατασκευή αντιγράφου περιεχομένων αρχείου

```
#include <fstream>

void error(const char* p, const char* p2 = "")
{
   std::cerr << p << ' ' << p2 << '\n';
   std::exit(1);
}

int main(int argc, char* argv[])
{
   if (argc != 3) error("wrong number of arguments");

   std::ifstream from(argv[1]);                // open input file stream
   if (!from) error("cannot open input file", argv[1]);

   std::ofstream to(argv[2]);                  // open output file stream
   if (!to) error("cannot open output file", argv[2]);

   char ch;
   while (from.get(ch)) to.put(ch);            // copy characters

   if (!from.eof() || !to) error("something strange happened");

   return 0;
}
```

Είσοδος/Έξοδος από Πίνακες

- Αρχείο -επικεφαλίδα `sstream`
- Χρήση της RAM για είσοδο/έξοδο
- Ανάλογη προσέγγιση με τη C (`sprintf()`, `sscanf()`)
- Με χρήση ρευμάτων

Αντικειμενοστραφής Προσέγγιση του Συστήματος I/O

- Κλάση: `ios` (υποκλάση της `ios_base`)
- Η κλάση `ios` προσφέρει λειτουργίες για μορφοποιημένη είσοδο και έξοδο.
- Υποκλάσεις: `istream`, `ostream` (υποκλάση τους: `iostream`)
  - Υποκλάσεις της `istream`: `ifstream`, `istringstream`
  - Υποκλάσεις της `ostream`: `ofstream`, `ostringstream`
  - Υποκλάσεις της `iostream`: `fstream`, `stringstream`
- `cin`: αντικείμενο της `istream`
- `cout`, `cerr`, `clog`: αντικείμενα της `ostream`
- `istream`: κλάση για ρεύματα εισόδου (συνάρτηση -τελεστής: `>>`)
- `ostream`: κλάση για ρεύματα εξόδου (συνάρτηση -τελεστής: `<<`)
- `iostream`: κλάση για ρεύματα εισόδου/εξόδου
- Κλάση `streambuf`: προσφέρει τις βασικές λειτουργίες I/O.

Έξοδος Μεταβλητών Στην κλάση `ostream` ορίζεται ο τελεστής `<<` για να χειρίζεται την έξοδο των ενσωματωμένων τύπων της C++.

```
class ostream : public virtual ios
{
   // ...
 public:
      ostream& operator<<(const char*);
      ostream& operator<<(char);
      ostream& operator<<(short);
      ostream& operator<<(int);
      ostream& operator<<(long);
   // ...
 };
```

- Μια συνάρτηση του τελεστή `<<` επιστρέφει αναφορά σε `ostream` για να μπορεί να εφαρμοστεί κι άλλος `<<` στη συνέχεια, πάνω στο αποτέλεσμά της (ομαδοποίηση από αριστερά προς τα δεξιά).
- Για νέους τύπους από το χρήστη, αρκεί ο χρήστης να ορίσει μια

  `ostream& operator<<(ostream& str,` $<new_cl_obj>$)

Είσοδος Μεταβλητών

Κατ' αναλογία με την έξοδο.
Στην κλάση `istream` ορίζεται ο τελεστής `>>` για να χειρίζεται την είσοδο των ενσωματωμένων τύπων της C++.

```
class istream : public virtual ios
{
   // ...
 public:
      istream& operator>>(const char*);
      istream& operator>>(char&);
      istream& operator>>(short&);
      istream& operator>>(int&);
   // ...
 };
```

- Ο τελεστής `>>` διαβάζει κατά τύπο.
- κενό, tab, nl, cr, fd: (whitespaces) διαχωριστικά
- Για νέους τύπους από το χρήστη, αρκεί ο χρήστης να ορίσει μια

  `istream& operator>>(istream& str,` $<\mathit{new_cl_obj}>$)

Σημαίες μορφοποίησης της ios (ένα enumeration της ios)

```
showbase/noshowbase: indicate the numeric base when printing
                     a num value
showpos/noshowpos: show (+) for positive values
uppercase/nouppercase: display uppercase for hex and
                       scientific values
showpoint/noshowpoint: show decimal point and trailing zeroes
                       for floating-point values
skipws/noskipws: skip whitespace on input
left/right/internal: output left/right justified /
                     padding in-between
scientific: use scientific notation
fixed: normal display with precision as specified or 6 (default)
dec/oct/hex: numeric base
```

...μπορούν να συνδυαστούν bitwise

Χειρισμός των σημαίων μορφοποίησης

- `setf` συνάρτηση-μέλος της `ios`,

  `long setf(long`<*flags*>`);`

  π.χ.

  ```
  cout.setf(ios::hex);
  cout.setf(ios::showbase);
  ```

  ή καλύτερα

  ```
  cout.setf(ios::showbase|ios::hex);
  ```

- `long unsetf(long`< *flags*>`);`
- `long setf(long`<*flags1*>, `long`<*flags2*>`);` `reset`< *flags2*> and `set`<*flags1*>
- `long flags();`: έλεγχος σημαιών
- `long flags(long` <*flag*>`);`

Άλλες συναρτήσεις -μέλη της `ios`

- `int width(int`<*width*>`)`: ορίζει το εύρος ενός πεδίου εξόδου
- `int precision(int`<*precision*>`)`: ορίζει την ακρίβεια κατά την έξοδο των πραγματικών
- `char fill(char`<***char***>`)`: ορίζει το χαρακτήρα που "γεμίζει" τα πεδία εξόδου

ΑΛΛΕΣ ΓΛΩΣΣΙΚΕΣ ΔΟΜΕΣ ΤΗΣ C++ (κρίσιμες για τις κλάσεις)

Κλάσεις (περίληψη προηγουμένων)

- Η δομή που εισάγεται με τη δεσμευμένη λέξη `class` (ή `struct`).
- Διαχωρισμός `private`/`public` επιτυγχάνει απόκρυψη πληροφορίας.
- Δημιουργία στιγμιοτύπων (αντικειμένων) που "κτίζονται" από τα μέλη -δεδομένα (data members) της κλάσης στην οποία ανήκουν.
- Αρχικοποίση στιγμιοτύπων μέσω συναρτήσεων κατασκευής.
- Καταστροφή στιγμιοτύπων εκμεταλλευόμενη συναρτήσεις καταστροφής.
- Χειρισμός αντικειμένων μέσω συναρτήσεων -μελών (member functions).
- Πρόσβαση στο αντικείμενο για το οποίο καλείται η συνάρτηση -μέλος με χρήση του δείκτη `this`.
- Αρχικοποίση δια αντιγραφής.
- Χρήση τελεστή ανάθεσης.

Αντικείμενα const, **Συναρτήσεις -μέλη** const, **Μέλη -δεδομένα** const

- Για την εφαρμογή της αρχής του "ελάχιστου απαραίτητου δικαιώματος".
- Κάποια αντικείμενα δεν θέλουμε να μεταβληθούν κατά τη διάρκεια της "ζωής" τους.
- Η χρήση της λέξης -κλειδιού `const` δηλώνει ότι το αντικείμενο δεν θα μεταβληθεί (και κάθε ενέργεια προς το αντίθετο δημιουργεί συντακτικό λάθος).
- Παράδειγμα:

```
class Time
{
public:
  Time(int=0, int=0, int=0);
  void settime(int, int, int);
  // ...
private:
  int hour;
  int minute;
  int second;
};

// ...

 const Time noon(12,0,0);
```

Σ' αυτήν την περίπτωση, οι μόνες συναρτήσεις που μπορούν να εφαρμοστούν σε αυτά τα αντικείμενα είναι εκείνες οι συναρτήσεις -μέλη που έχει δηλωθεί ότι δεν μεταβάλλουν τα αντικείμενα στα οποία εφαρμόζονται.

- Το παραπάνω επιτυγχάνεται με τη χρήση της λέξης -κλειδιού `const` τόσο στη δήλωση όσο και στον ορισμό της συνάρτησης -μέλους (θέτοντάς το μετά τις παραμέτρους της συνάρτησης).

```
   class Time
     {
     public:
     // ...
     int getHour() const;
     int getMinute() const;
     int getSecond() const;
     // ...
     };
// ...
   int Time::getHour() const
        { return hour; }
   int Time::getMinute() const
        { return minute; }
   int Time::getSecond() const
        { return second; }
```

- Μπορώ να έχω υπερφόρτωση για τις `const` συναρτήσεις -μέλη με non -const εκδοχές τους.

- Η επιλογή για τη χρήση μιας `const` ή non -const εκδοχής μιας συνάρτησης -μέλους γίνεται από το μεταγλωττιστή με βάση το εάν εφαρμόζεται σε `const` αντικείμενο ή όχι.

- Μέλη -δεδομένα μπορούν να δηλωθούν σαν `const` επιβάλλοντας η τιμή τους να μην αλλάξει κατά τη διάρκεια ζωής του αντικειμένου. Η τιμή αυτή δίδεται μέσω initializer list από τη συνάρτηση κατασκευής.

Φιλικές Συναρτήσεις και Φιλικές Κλάσεις

- Μια κλάση μπορεί να δηλώσει μια συνάρτηση που δεν είναι συνάρτηση -μέλος της ως φιλική (με τη χρήση της λέξης -κλειδιού `friend`).
- Η συνάρτηση αυτή παρόλο που δεν είναι στην εμβέλεια της κλάσης, έχει δικαίωμα να έχει πρόσβαση στο ιδιωτικό μέρος της κλάσης.
- Αυτό είναι ιδιαίτερα χρήσιμο στην περίπτωση ορισμού τελεστών και συναρτήσεων επανάληψης (iterators).
- Σαν φιλικές μπορούν να δηλωθούν και ολόκληρες κλάσεις (πάλι με τη χρήση της λέξης -κλειδιού `friend`). Π.χ.

```
class NewClass
{
 // ...
 friend class FriendClass;
 // ...
};
```

Παράδειγμα χρήσης φιλικών συναρτήσεων

```
class Count
 {
     friend void setX(Count&, int);       // friend declaration

  public:
     Count() { x= 0; }
     void print() const { cout << x << endl; }

  private:
     int x;
 };

 void setX(Count& c, int val)
 {
   c.x = val;                                        // Access to private part
 }

 int main()
 {
   Count counter;
   counter.print();
   setX(counter,8);
   counter.print();
   return 0;
 }
```

Παρατηρήσεις στη "φιλία"

- Οι φιλικές συναρτήσεις δεν αποτελούν συναρτήσεις -μέλη της κλάσης (απλώς τυχαίνει να παίρνουν ένα αντικείμενο της κλάσης στις παραμέτρους τους).
- Δηλώσεις `friend` μπορούν να εμφανιστούν οπουδήποτε μέσα στον ορισμό μιας κλάσης (δεν επηρεάζει αν είναι μέσα στο private ή στο public μέρος τους)
- Η φιλία δεν είναι ούτε συμμετρική ούτε μεταβατική μεταξύ των κλάσεων, ούτε κληρονομείται.

Στατικά Μέλη Κλάσεων

- Κάθε αντικείμενο μιας κλάσης έχει το δικό του αντίγραφο για κάθε μέλος -δεδομένο της κλάσης ("χτίζεται" από αυτά).
- Υπάρχουν περιπτώσεις που θέλουμε όλα τα αντικείμενα να μοιράζονται το ίδιο αντίγραφο για μια μεταβλητή.
- Αυτό επιτυγχάνεται με τη χρήση της λέξης -κλειδιού `static` κατά τη δήλωση της μεταβλητής στον ορισμό της κλάσης.
- Η εμβέλεια των στατικών μελών μιας κλάσης είναι η κλάση και μπορούν να δηλωθούν ως private ή public (ή protected).
- Τα στατικά μέλη κλάσεων πρέπει να αρχικοποιούνται (και μόνο μια φορά) σε επίπεδο εμβέλειας αρχείου.
- Τα στατικά μέλη κλάσεων μπορούν να δηλωθούν και ως `const`.

Παράδειγμα:

Δήλωση στατικών μελών:

```
// employee.h
class Employee
   {
    public:
      Employee(const char*, const char*, const long);

      ~Employee();

      const char* getFirstName() const;
      const char* getLastName() const;
      const long getDate() const { return date_of_hire; }

      static int getCount();     //   <----------

      static const int MAX_EMPLOYEES = 100;  //   <----------

    private:
      char* firstName;
      char* lastName;

      static int count;          //   <----------
      const long date_of_hire;              //   <----------
   };
```

Ορισμός και αρχικοποίηση:

```
// employee.cc
#include<cstring>
#include "employee.h"
using namespace std;


int Employee::count = 0;        //   <----------


int Employee::getCount() { return count; }
```

```
Employee::Employee(const char* first, const char* last, const long date):
   date_of_hire(date)
{
    if ( count < MAX_EMPLOYEES ){
    firstName = new char[strlen(first) + 1];
    strcpy( firstName, first );

    lastName = new char[strlen(last) + 1];
    strcpy( lastName, last );

//    date_of_hire = date   // error

    ++count; }                  //    <----------
}


Employee::~Employee()
{
    delete [] firstName;
    delete [] lastName;

    --count;                    //    <----------
}


const char* Employee::getFirstName() const
{
   return firstName;
}

const char* Employee::getLastName() const
{
   return lastName;
}
```

Χρήση στατικών μελών:

```
// test.cc
#include <iostream>

#include "employee.h"

using namespace std;
```

```
int main()
{
   cout << Employee::getCount() << endl;         //    <----------

   Employee* e1 = new Employee( "FN1", "LN1", 1122004 );
   Employee* e2 = new Employee( "FN2", "LN2", 1012003 );

   cout << e1->getFirstName() << endl;
   cout << e1->getLastName() << endl;
   cout << e1->getDate() << endl;

   cout<< e1-> MAX_EMPLOYEES << endl;
   cout<< Employee::MAX_EMPLOYEES << endl;

   cout << e1->getCount() << endl;               //    <----------

   delete e1;
   delete e2;

   cout << Employee::getCount() << endl;         //    <----------

   return  0;
}
```

Παρατηρήσεις στα στατικά μέλη κλάσεων

- Μπορούμε να έχουμε πρόσβαση στα public static μέλη μιας κλάσης
  - είτε μέσω ενός αντικειμένου της κλάσης
  - είτε με χρήση του ονόματος της κλάσης και του τελεστή `::`
- Έξω από την κλάση μπορούμε να έχουμε πρόσβαση στα static private (ή protected) μέλη μέσω public συναρτήσεων -μελών ή φιλικών συναρτήσεων.
- Τα στατικά μέλη των κλάσεων έχουν υπόσταση ανεξάρτητα από τη δημιουργία ή όχι αντικειμένων της κλάσης.
- Στην παραπάνω περίπτωση, μπορούμε να έχουμε πρόσβαση, έξω από την κλάση στα public static μέλη της, χρησιμοποιώντας το όνομα της κλάσης και τον τελεστή `::`
- Όταν δεν έχουν δημιουργηθεί αντικείμενα μιας κλάσης, για να έχουμε πρόσβαση σε private (ή protected) static μέλη της κλάσης, έξω από την κλάση, πρέπει να έχει οριστεί μια public static συνάρτηση -μέλος της κλάσης και να κληθεί με το όνομα της κλάσης και χρήση του τελεστή `::`
- Δε μπορούμε να χρησιμοποιήσουμε το δείκτη `this` σε μια static συνάρτηση -μέλος μιας κλάσης.
- Δε μπορούμε να δηλώσουμε μια static συνάρτηση -μέλος σαν `const`.
- Τα στατικά μέλη -δεδομένα και οι στατικές συναρτήσεις -μέλη μιας κλάσης υπάρχουν και μπορούν να χρησιμοποιηθούν ακόμα κι αν δεν έχουν δημιουργηθεί αντικείμενα της κλάσης.

Ένα επαναληπτικό παράδειγμα για:

- Μιά απλή κλάση με δήλωση φιλίας στον υπερφορτωμένο τελεστή εκτύπωσης:

```
// File: date.h
#include<iostream>

class Date {
  int day, month, year;

 public:
  Date(int d, int m, int y) : day(d), month(m), year(y) {}
  int get_year() {return year;}
  int get_month() {return month;}
  int get_day() {return day;}

friend std::ostream& operator<<(std::ostream&, const Date&);
};
```

- class και struct καθώς και λίστες:

```
// File:  person.h
#include"date.h"

enum Sex {M,F};

class Person{
   char* name;
   Date date_of_birth;
   Sex sex;

public:
   Person(int d, int m, int y, Sex sx):
          date_of_birth(d,m,y), sex(sx), name(NULL) {};
   ~Person() { if (name!=NULL)
                 { std::cout << "Deleting " << name << endl;
                   delete[] name;} }

   void set_name(const char*) ;
   char* get_name() {return name;}
   Date get_date_of_birth() {return date_of_birth;}
   Sex get_sex() {return sex;}

friend ostream& operator<<(ostream&, const Person&);
};
```

```
struct PersonN{
   Person* person;
   PersonN* next;
   PersonN(Person* p, PersonN* n) : person(p),next(n){};
   ~PersonN() { delete person; //assuming dynamic creation
                if (next != NULL) delete next; }
};
```

```
// File: person.cc
#include"person.h"

   void Person::set_name(const char* nam)
    {
       if (name != NULL) delete[] name;
       name = new char[strlen(nam)+1];
       strcpy(name,nam);
    }
```

- class και struct

```
// File: family.h
#include"person.h"



struct Car
{
  int licence_no;
  char* model;
  int cc;
  Car(int no, char* mdl, int c):
     licence_no(no), model(mdl), cc(c) {};
  ~Car(){ cout << "Deleting car " << licence_no << endl; }
};
  ostream& operator<<(ostream&, const Car&);
```

```
class Family{
  Person& husband;
  Person& wife;
  Date date_of_wedding;
  PersonN* children;
  Car* owned_car;

  Person* first_child();
  Person* last_child();

 public:
  Family(int d, int m, int y, Person& h, Person& w):
        date_of_wedding(d,m,y), husband(h), wife(w),
        children(NULL), owned_car(NULL) {};
  ~Family() { if (has_children())
                  delete children;  // dynamic creation assumed
              if (has_car())
                  delete owned_car; } // dynamic creation assumed

  Person& birth(int, int, int, Sex);

  bool has_children() { return children != NULL; }
  bool has_car() { return owned_car != NULL; }

  int diff1()
      { if (has_children())
          return first_child()->get_date_of_birth().get_year() -
                 date_of_wedding.get_year(); }

  int diff2()
      { if(has_children())
         return last_child()->get_date_of_birth().get_year() -
                first_child()->get_date_of_birth().get_year(); }

  Car* buy_car(int no, const char* mdl, int c) //we may need to return null
       { owned_car = new Car(no, mdl, c);
         return owned_car;}

friend ostream& operator<<(ostream&, const Family&);
};
```

```
// File: family.cc
#include<iostream>
#include"family.h"

using namespace std;

Person& Family::birth(int d, int m, int y, Sex sx)
{
   Person* pChild = new Person(d,m,y,sx);

   PersonN* pChildN = new PersonN(pChild,children);

   children = pChildN;

   return *pChild;
}


Person* Family::last_child()
{
  if (children != NULL) {return children->person;}
     else {
       cout << "Careful, no children" << endl;
       return NULL; };
}

Person* Family::first_child()
{
  if (children != NULL)  {
        PersonN* tmpp = children;
        while(tmpp->next != NULL) tmpp = tmpp->next;
        return tmpp->person; }
     else {
       cout << "Careful, no children" << endl;
       return NULL; };
}
```

- Υλοποίηση του τελεστή εκτύπωσης για τις προηγούμενες κλάσεις:

```
// File:  io.cc
#include<iostream>
#include "family.h"
using namespace std;

ostream& operator<<(ostream& os, const Date& date)
{
    os << date.day << "  " <<
          date.month << "  " <<
          date.year << endl;
    return os; }

ostream& operator<<(ostream& os, const Person& person)
{
   char* thesex;
   if (person.sex == M)
       thesex = "male";
      else
       thesex = "female";

   os << person.name << "  " << thesex <<
         " born on: " << person.date_of_birth << endl;
   return os; }

ostream& operator<<(ostream& os, const Car& cr)
{
   os << cr.licence_no << "  model " << cr.model << " with "
      << cr.cc << "cc" << endl;
   return os; }

ostream& operator<<(ostream& os, const Family& f)
{
    os << f.husband << "married with: "
       << f.wife << "on " << f.date_of_wedding;

    if (f.children == NULL ) os << "and have no children" << endl;
       else
        {PersonN* tmp = f.children;
         cout << "have children: " << endl;
         while(tmp->next != NULL)
             { os << *(tmp->person); tmp = tmp->next;}
```

```
            os << *(tmp->person) << endl;
        }
    if(f.owned_car != NULL)
       os << "and bought the car :" << *(f.owned_car) << endl;
    return os; }
```

- Μια χρήση των προηγουμένων:

```
// File: story.cc
#include<iostream>
#include<fstream>
#include"family.h"
using namespace std;

int main()
{
 Person h(1,3,1950,M);
 Person w(2,6,1955,F);

 h.set_name("Giorgos Papas");
 w.set_name("Nitsa Labrou");
 cout << h << endl;
 cout << w << endl;

cout << "/////////////////////////////////////////////" << endl;
/////  Set up a family ///////////////////////////
 Family f(1,1,1990,h,w);

/////////////////////////////////////////////
/// Set up a family -by pointer ////////////
// Family* pf = new Family(1,1,1990,h,w);
////// have children -in family by pointer///
// Person& child1 = pf->birth(1,1,1992,M);
// Person& child2 = pf->birth(1,1,1994,F);
// Person& child3 = pf->birth(1,1,1996,F);

/////////////////////////////////////////
//// Buy car /////////////////////////
 cout <<
        *(f.buy_car(1234,"glof",1800))
      << endl;
// cout << *pf->buy_car(1234,"glof",1800) << endl;
```

```
/////////////////////////////////////////
/////// The family gives birth /////////
// f.birth(1,1,1992,M);
// f.birth(1,1,1994,F);
// f.birth(1,1,1996,F);

cout << "///////////////////////////////////////////" << endl;
/////// The family gives birth in "named" //////////
 Person& child1 = f.birth(1,1,1992,M);
 Person& child2 = f.birth(1,1,1994,F);
 Person& child3 = f.birth(1,1,1996,F);

////////////They name their children //////////////

 child1.set_name("nikos");
 child2.set_name("marika");
 child3.set_name("lilika");

// cout << f.diff1() << endl;
// cout << f.diff2() << endl;

/////////////////////////////////////////
//// PRINTING  /////////////////////////
////////////////////////////////////////


///////////////////////////////////////
//// Printing to standard output //////
///////////////////////////////////////
///////////////////////////////////////
// Date d1(1,4,2005);
// cout << d1 << endl;
/////////////////////////////////////////

cout << "///////////////////////////////////////////" << endl;
 cout << f << endl;
// cout << *pf << endl;
```

```
/////////////////////////////////////////
///// Printing to file "outfile" ///////
////////////////////////////////////////
 ofstream out("outfile");
    if (!out) {
       cout << "Cannot open file.\n";
       return 1;
    }

  out << h << endl;
  out << w << endl;
  out << f << endl;

///////////////////////////////////////////////////////////
// Delete pointer family //////////////////////////////////
///////////////////////////////////////////////////////////
// delete pf;
///////////////////////////////////////////////////////////

cout << "This is the last output of the main function" << endl;

return 0;
}
```

ΜΕΤΑΤΡΟΠΕΣ ΤΥΠΩΝ

- Μπορεί να χρειαστεί να μετατρέψουμε ρητά μια τιμή, που είναι κάποιου τύπου, σε κάποιον άλλο τύπο. Π.χ.

  ```
  float r = float(1);
  ```

  μετατρέπει τον ακέραιο `1` στον πραγματικό `1.0f` για να δοθεί σαν τιμή στον πραγματικό `r`.

- Υπάρχουν δύο τρόποι να το συμβολίσουμε:
  1. (à la C): cast, με τη χρήση παρενθέσεων, π.χ.

     ```
     (double) a
     ```

  2. "συναρτησιακή" γραφή:

     ```
     double(a)
     ```

     δεν μπορεί όμως να χρησιμοποιηθεί για τύπους που έχουν σύνθετο συμβολισμό. Π.χ.

     ΟΧΙ `char*(0777);`

     ΜΟΝΟ `(char*) 0777;`

- Στη C++ δεν υπάρχει τόσο έντονη ανάγκη για ρητή μετατροπή τύπων, αντίθετα με τη C.
- Στους δείκτες, ανάθεση από/σε `void*` δεν χρειάζεται ρητή μετατροπή τύπου.
- Αυτόματη μετατροπή τύπων, ορισμένη από τον χρήστη:
  – μέσω ορισμού συναρτήσεων κατασκευής (του *τύπου -προς* με όρισμα του *τύπου -από*)
  – μέσω ορισμού τελεστών (του *τύπου -προς* που εφαρμόζονται στον *τύπο -από*)
- Ρητή μετατροπή τύπων (για να αγνοήσουμε τελείως τις μετατροπές à la C):

`static_cast` : για "ασφαλείς" μετατροπές που θα έκανε ούτως ή άλλως ο μεταγλωττιστής ή λιγότερο ασφαλείς αλλά καλά ορισμένες

`const_cast` : για να παραβιάσουμε δήλωση `const` (ή `volatile`)

`reinterprete_cast` : ΕΠΙΚΙΝΔΥΝΟ: ερμηνεύει το αντικείμενο σαν να ήταν ακολουθία από bits. (Πρέπει να το "επαναφέρουμε" πριν το ξαναχειριστούμε ομαλά)

`dynamic_cast` : "ασφαλής" μετάβαση σε χαμηλότερη κλάση στην ιεραρχία.

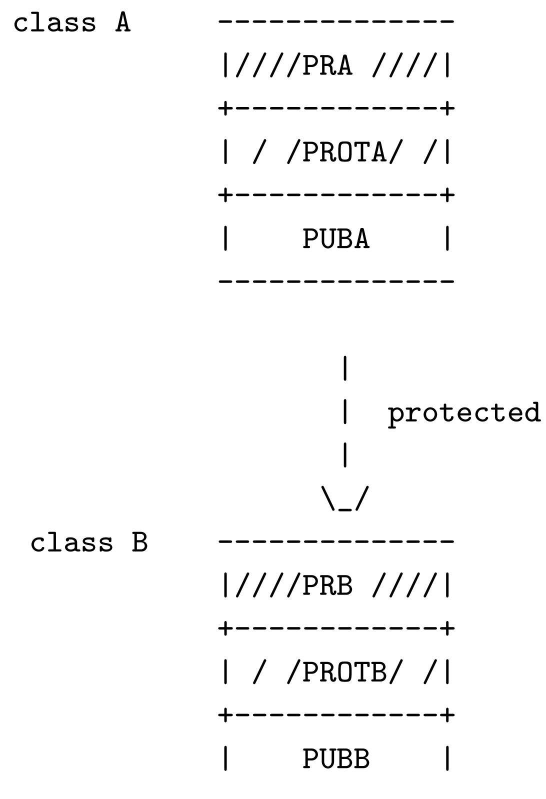
Protected KAI Private ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑ ΣΤΗ C++

PUBLIC inheritance (την οποία έχουμε εξετάσει)

```
class B : public A { ... };

            class A     --------------
                        |////PRA ////|
                        +------------+
                        | / /PROTA/ /|
                        +------------+
                        |    PUBA    |
                        --------------

                              |
                              |  public
                              |
                             \_/
             class B    --------------
                        |////PRB ////|
                        +------------+
                        | / /PROTB/ /|
                        +------------+
                        |    PUBB    |
                        --------------

                                          PUBB = PUBB + PUBA
                                          PROTB = PROTB + PROTA
                                          PRB = PRB
```

PROTECTED inheritance

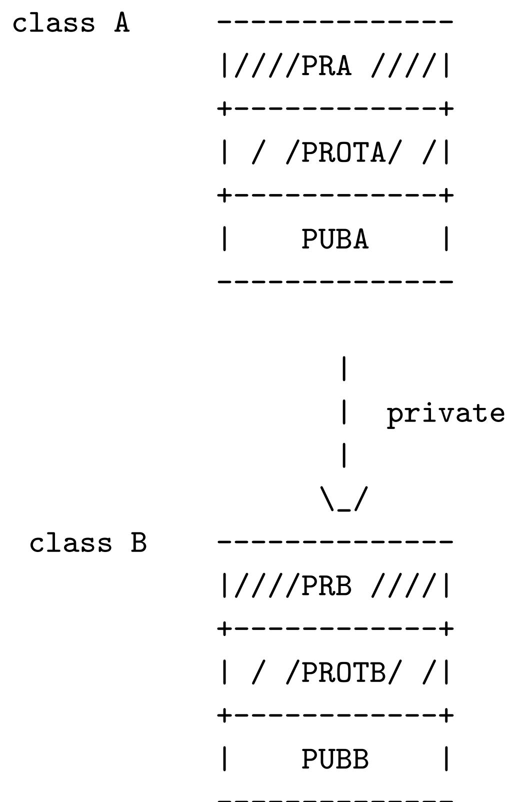
```
class B : protected A { ... };
```




```
PUBB = PUBB
PROTB = PROTB + PROTA + PUBA
PRB = PRB
```

PRIVATE inheritance

```
class B : private A { ... };
```




```
PUBB = PUBB
PROTB = PROTB
PRB = PRB + PROTA + PUBA
```

Οι private και οι protected μορφές κληρονομικότητας δεν είναι καθαρές "is a" σχέσεις.

Ευχαριστίες, εκ βαθέων

Θα ήθελα να ευχαριστήσω ιδιαίτερα τους συνεργάτες, και παλαιότερα φοιτητές, του μαθήματος Νίκο Ποθητό και Μανόλη Πλατάκη τόσο για τις παρατηρήσεις τους στις σημειώσεις όσο και για την αποδοχή των αρχών του μαθήματος και την προσφορά τους σε αυτό.

Επίσης, θερμά ευχαριστώ στον ανεκτίμητο Στέφανο Σταμάτη για την υποστήριξη των δραστηριοτήτων που σχετίζονται με το μάθημα στα πλαίσια όλων των απαιτούμενων από το Τμήμα.

Τέλος, σε όλους όσους αποδέχτηκαν το μάθημα, το αγάπησαν και το στήριξαν με τη συμβολή τους είτε σαν μεταπτυχιακοί συνεργάτες του είτε εθελοντικά με τη συμμετοχή τους στο forum του, πολλά ευχαριστώ.

Βιβλιογραφία

- Bjarne Stroustrup, “Η Γλώσσα Προγραμματισμού C++” (Τρίτη Αμερικάνικη Έκδοση), εκδόσεις “Κλειδάριθμος”, 1999
- Bruce Eckel, “Thinking In C++” (Second Edition), Volume One: Introduction to Standard C++, Prentice Hall, 2000
- ISO International Standard: Programming Languages — C++, 1998
- Stanley B. Lippman, Josee Lajoie, “C++ Primer” (3rd Edition), Addison-Wesley, 1998
- Stanley B. Lippman, “Inside the C++ Object Model” Addison-Wesley, 1996
- Herbert Schildt, “C++: The Complete Reference” (2nd Edition), McGraw-Hill, 1995
- Till Jeske, “C++ Nitty Gritty”, Addison-Wesley, 2000 (Pearson Education, 2002)
- The Standard Template Library: *http://www.sgi.com/tech/stl*
- Bertrand Meyer, “Object -Oriented Software Construction” (2nd Edition), Prentice Hall, 2000
- Russel Winder, Graham Roberts, “Developing Java Software” (2nd Edition), John Wiley and Sons, 2000