

ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΑΦΗΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

CS-409, Χειμερινό εξάμηνο 2011
Πανεπιστήμιο Ιωαννίνων, Τμήμα Πληροφορικής

Παναγιώτης Τσαπάρας

Συστάσεις

- Ποιός είμαι:
 - Email: tsap@cs.uoi.gr
 - Γραφείο: B.3
 - Προτιμώμενες ώρες γραφείου: Τρίτη/Πέμπτη 2-4 π.μ.
- Ποιοί είσαστε:
 - Συμπληρώστε τη φόρμα με τα στοιχεία σας για την email λίστα του μαθήματος.

Γενικές πληροφορίες για το μάθημα

- Διαλέξεις: Τεταρτη 3-6 π.μ.
- Εργαστήρια: Πέμπτη 12-2 π.μ.
- Web: <http://www.cs.uoi.gr/~tsap/teaching/cs-409/>

Βαθμολογία

- Ασκήσεις: 40%
 - Οι ασκήσεις είναι υποχρεωτικές για όλους όσους παίρνουν το μάθημα, ανεξαρτήτως έτους.
- Τελική εξέταση: 60%
- Προϋπόθεση επιτυχίας:
 - Ασκήσεις ≥ 4.0 και Τελική εξέταση ≥ 4.0
 - Βαθμός = $0.4 * \text{Ασκήσεις} + 0.6 * \text{Τελική} \geq 5.0$

«Προαπαιτούμενα»

- CS-106: Εισαγωγή στον προγραμματισμό
- CS-208: Προγραμματισμός σε C

«Συνεργίες»

- CS-304: Αρχές Γλωσσών Προγραμματισμού (3ο εξαμ.)
- CS-302: Δομές Δεδομένων (3ο εξαμ.)

Στόχοι του μαθήματος

- Να κατανοήσετε τις βασικές αρχές του Αντικειμενοστραφή Προγραμματισμού (Object Oriented Programming) και να αποκτήσετε πρακτική εξάσκηση με τις γλώσσες προγραμματισμού C++ και Java.

Θέματα που θα καλύψουμε

- Διαδικασιακός vs. Αντικειμενοστραφής προγραμματισμός
- Διαφορές και ομοιότητες μεταξύ C και C++
- Δημιουργία και Καταστροφή αντικειμένων.
- Κληρονομικότητα και πολυμορφισμός
- Καθιερωμένη Βιβλιοθήκη Προτύπων (Standard Template Library)
- Εισαγωγή στη γλώσσα προγραμματισμού Java.

Βιβλιογραφία

- *Αντικειμενοστραφής Προγραμματισμός στη C++*, R. Lafore, Εκδόσεις Κλειδάριθμος, 2005, ISBN 960-209-904-4
- *Η Γλώσσα Προγραμματισμού C++*, B. Stroustrup, Εκδόσεις Κλειδάριθμος, 2003 ISBN 960-332-142-7
- *Java με UML*, E. Lervik , V. Havdal, Εκδόσεις Κλειδάριθμος, 2004, ISBN 960-209-802-3.
- *Thinking in Java*, Bruce Eckel,
<http://www.mindview.net/Books/TIJ/>
- *Building Java Programs: A Back to Basics Approach, 2nd edition*, Stuart Reges and Marty Stepp

Οι διαφάνειες του μαθήματος στηρίζονται στα παραπάνω βιβλία και στις διαφάνειες των κ. Χ. Τζώρτζη και κ. Α. Ζάρρα.

ΕΙΣΑΓΩΓΗ

Γενικές ιδέες αντικειμενοστραφή
προγραμματισμού

Εισαγωγή

- Ως σχεδιαστές / προγραμματιστές λογισμικού σκοπό έχουμε να υλοποιούμε λογισμικό για την επίλυση προβλημάτων με βάση κάποιες δοσμένες απαιτήσεις.

Βασικά στάδια ανάπτυξης λογισμικού

- **Ανάλυση απαιτήσεων**
 - Τι ζητάει το πρόβλημα
- **Σχεδίαση**
 - Από ποια βασικά δομικά στοιχεία θα αποτελείται το λογισμικό.
 - Πχ. Δομές δεδομένων, συναρτήσεις, κλάσεις
- **Υλοποίηση / Κατασκευή**
 - Κωδικοποίηση σε C, C++, Java, Fortran ...
- **Έλεγχος**
 - Αποσφαλμάτωση (testing/debugging)
- **Εγκατάσταση**
- **Συντήρηση (maintenance)**
 - Συντήρηση του υπάρχοντος κώδικα
 - Επεκτάσεις και διορθώσεις με βάση νέες ανάγκες

Δημιουργία μεγάλων συστημάτων

- **Μεγάλης κλίμακας λογισμικό** αναπτύσσεται από ομάδες προγραμματιστών που μοιράζονται τις αρμοδιότητες για τα διάφορα στάδια της διαδικασίας.
 - Οι ομάδες συχνά ανανεώνονται με νέο κόσμο.
- Σε τέτοιες περιπτώσεις το μείζον πρόβλημα δεν είναι πλέον η υλοποίηση ενός μόνο αλγορίθμου ή η μετατροπή ενός αλγορίθμου σε κώδικα.
- Το μείζον πρόβλημα είναι η **βέλτιστη σχεδίαση και υλοποίηση του κώδικα** με στόχους:
 - Επιτάχυνση της φάσης της υλοποίησης
 - Διευκόλυνση της φάσης του ελέγχου
 - Διευκόλυνση της **συντήρησης**.

Συντήρηση

- Τι περιλαμβάνει η συντήρηση?
 - **Επιδιόρθωση** σφαλμάτων που προκύπτουν μετά την εγκατάσταση και χρήση του λογισμικού
 - **Προσθήκη** νέων λειτουργιών λόγω της μεταβολής των αρχικών απαιτήσεων των χρηστών του λογισμικού

Προγραμματιστικές απαιτήσεις

- Για να εξασφαλίσουμε εύκολη συντήρηση, θέλουμε κώδικα ο οποίος:
 - να είναι **εύκολο να διαβαστεί** και να κατανοηθεί από κάποιον που τον βλέπει για πρώτη φορά (αναγνωσιμότητα -- readability)
 - να **ανταποκρίνεται** στο μοντέλο που έχουμε για την **πραγματικότητα** (intuitiveness).
 - να είναι **εύκολο να επαναχρησιμοποιηθεί** (reusability).
- Χρειαζόμαστε ένα προγραμματιστικό μοντέλο (programming paradigm) που να μας εξασφαλίζει τα παραπάνω.

Spaghetti κώδικας

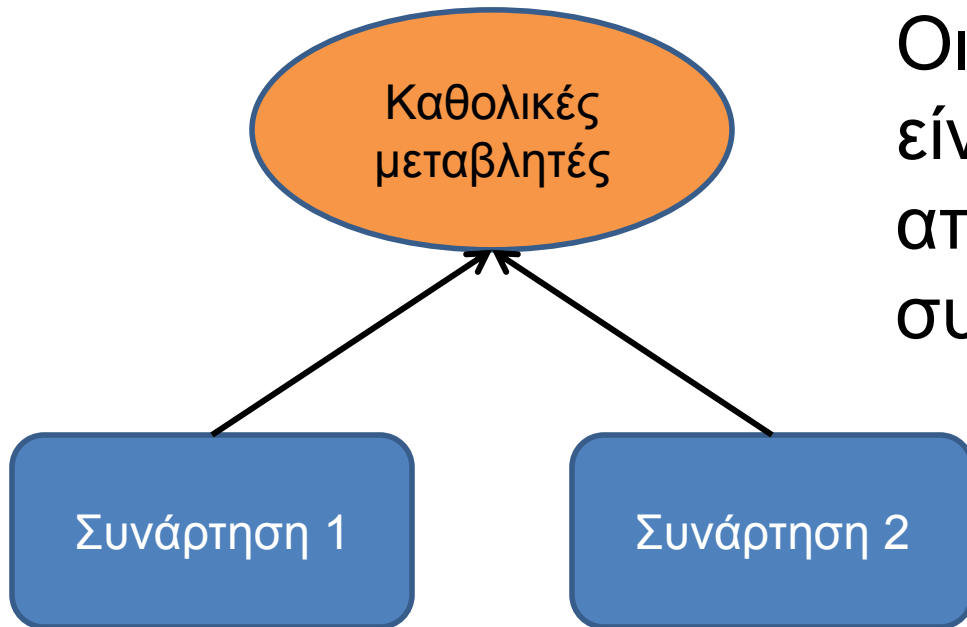
- Όλος ο κώδικας μια σειρά εντολών μέσα στη συνάρτηση `main()`.
 - Αδύνατον να διαβαστεί όταν γίνει πάνω από μερικές εκατοντάδες γραμμές.
 - Πολύ δύσκολη η αποσφαλμάτωση και η αλλαγή του κώδικα γιατί απαιτεί αλλαγές σε πολλά διαφορετικά σημεία.

Διαδικασιακός προγραμματισμός

- Δημιουργία λογικών μονάδων κώδικα στη μορφή **συναρτήσεων** (functions).
 - Ο κώδικας γίνεται μια σειρά από κλήσεις σε συναρτήσεις. Οι συναρτήσεις μπορούν να οργανωθούν ιεραρχικά όπου η μία καλεί την άλλη.
 - Η αναγνωσιμότητα του κώδικα βελτιώνεται αισθητά.
 - Η επαναχρησιμοποίηση του κώδικα γίνεται πιο εύκολα, και η αποσφαλμάτωση περιορίζεται σε λιγότερα σημεία.
- Το μοντέλο αυτό χρησιμοποιήθηκε επιτυχώς για πολλές δεκαετίες.

Μειονεκτήματα

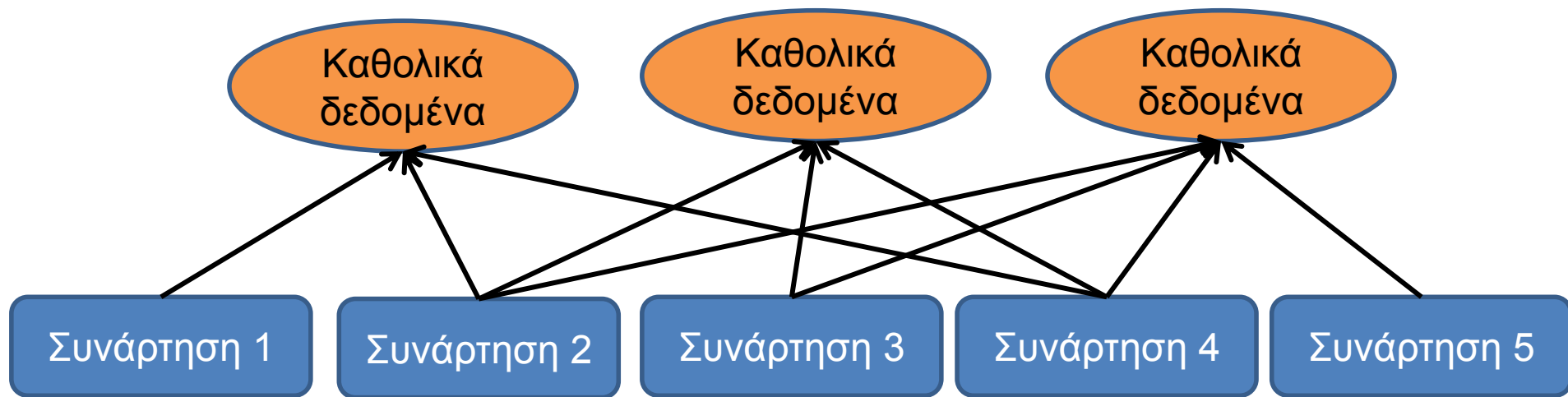
- Όταν πολλές συναρτήσεις πρέπει να χρησιμοποιήσουν τα ίδια δεδομένα, τότε αυτά πρέπει να είναι **καθολικά** (global)



Οι καθολικές μεταβλητές είναι προσπελάσιμες από οποιαδήποτε συνάρτηση.

Μειονεκτήματα

- Όταν το πρόγραμμα μεγαλώσει, η πρόσβαση στα καθολικά δεδομένα γίνεται δύσκολο να ελεγχθεί



- Αλλαγές στα καθολικά δεδομένα έχουν ως αποτέλεσμα να πρέπει να τσεκάρουμε όλες τις συναρτήσεις για να δούμε ποιες επηρεάζονται.
- Είναι εύκολο μια συνάρτηση να αλλάξει κατά λάθος κάτι που δεν πρέπει.
- Η ανάγνωση και η αποσφαλμάτωση (debugging) του κώδικα γίνονται πιο δύσκολα.

Μειονεκτήματα

- Ο διαδικασιακός προγραμματισμός δεν μοντελοποιεί σωστά τον πραγματικό κόσμο.
- Στον φυσικό κόσμο έχουμε οντότητες (αντικείμενα, ανθρώπους) που έχουν **χαρακτηριστικά**, και μπορούν να εκτελέσουν κάποιες **δράσεις**.
- Ο διαδικασιακός προγραμματισμός δεν ομαδοποιεί μαζί τα δεδομένα και τις δράσεις για να ορίσει την αντίστοιχη οντότητα.

Αντικειμενοστραφής προγραμματισμός

- Στον αντικειμενοστραφή προγραμματισμό οι **μεταβλητές** και οι **συναρτήσεις** που επενεργούν πάνω σε αυτές τις μεταβλητές ομαδοποιούνται σε μια αφαίρεση που ονομάζουμε **κλάση**.
- Τα **αντικείμενα** είναι στιγμιότυπα (instances) των κλάσεων. Αποθηκεύουν **δεδομένα**, και εκτελούν **συναρτήσεις** πάνω σε αυτά τα δεδομένα.
- Ένα φυσικό αντικείμενο μπορεί να μοντελοποιηθεί ως ένα αντικείμενο στον ΑΣΠ.

Γενική ιδέα

- ... σε πρώτη φάση, φανταστείτε μια **κλάση** σαν ένα **struct X**
 - το οποίο πέρα από τα **πεδία** του ορίζει και
 - ένα σύνολο από **συναρτήσεις** μέσω των οποίων το υπόλοιπο λογισμικό επεξεργάζεται τα δεδομένα που αποθηκεύονται στα πεδία μεταβλητών τύπου **X**

Παράδειγμα – Procedural

```
struct Human {
    int height;
    int age;
};
void isborn(struct Human *aHuman);
void ages(struct Human *aHuman);
void grows(struct Human *aHuman, int inc)

main(){
    struct Human peter;
    isborn(&peter);
    ages(&peter);
    grows(&peter, 10);
}

void isborn(struct Human *aHuman){
    aHuman->height = 40;
    aHuman->age = 0;
}
void ages(struct Human *aHuman){
    aHuman->age += 1;
}
void grows(struct Human *aHuman, int inc){
    aHuman->height += inc;
}
```

Παράδειγμα – OOP

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

class Human {
private:
    int height;
    int age;

public:
    void Ages();
    void IsBorn();
    void Grows(int inc);
};

void Human::Ages() {
    age += 1; // απλούστερος κώδικας !!!
}

void Human::IsBorn() {
    height = 40;
    age = 0;
}

void Human::Grows(int inc) {
    height += inc;
}
```

```
int main() {
    Human peter;
    peter.IsBorn();
    peter.Ages();
    peter.Grows(10);
    // απλούστερος κώδικας !!!
}
```

Λίγη ορολογία

- Τα πεδία **height, age** τα λέμε **χαρακτηριστικά** της κλάσης **Human**
- Οι συναρτήσεις **IsBorn, Ages, Grows** ονομάζονται **μέθοδοι** της κλάσης **Human**
- Οι μεταβλητή **peter** τύπου **Human** είναι ένα **αντικείμενο** της κλάσης **Human**
- Οι τιμές των πεδίων ενός αντικειμένου κατά τη διάρκεια εκτέλεσης του λογισμικού ορίζουν την **κατάσταση** του αντικειμένου
- Οι μέθοδοι της κλάσης ορίζουν τη **συμπεριφορά** των αντικειμένων της
 - Δηλ. τον τρόπο με τον οποίο μεταβάλλεται η κατάσταση των αντικειμένων της κλάσης όταν κληθεί μια μέθοδος....

Ενθυλάκωση (encapsulation)

- Τα χαρακτηριστικά και οι μέθοδοι χωρίζονται σε **public** και **private**.
 - **Private χαρακτηριστικά**: η επεξεργασία των τιμών τους για ένα αντικείμενο γίνεται μόνο μέσω των μεθόδων της κλάσης (απόκρυψη δεδομένων – data hiding).
 - **Public μέθοδοι**: μπορούν να χρησιμοποιηθούν οπουδήποτε στον κώδικα για την αλλαγή της κατάστασης ενός αντικειμένου της κλάσης
 - ...τα σφάλματα δημιουργούνται σε σημεία του κώδικα στα οποία γίνεται επεξεργασία δεδομένων (π.χ. ανάθεση τιμών)
 -αν η επεξεργασία γίνεται μέσω συγκεκριμένων μεθόδων τα σφάλματα βρίσκονται εκεί και όχι διάσπαρτα σε όλο τον κώδικα !
- Ο συνδυασμός δεδομένων και συναρτήσεων μέσα στα αντικείμενα, και η «προστασία» των private δεδομένων μέσω των public συναρτήσεων λέγεται **ενθυλάκωση** (encapsulation)

Μια αναλογία

- Τα αντικείμενα ως οργανισμοί μέσα σε μια εταιρεία:
 - Κάθε οργανισμός (μισθοδοσία, πωλήσεις, προσωπικό) έχει κάποια δεδομένα.
 - Η πρόσβαση στα δεδομένα γίνεται μόνο μέσα από τις υπηρεσίες που προσφέρει ο οργανισμός.
 - Κανείς άλλος δεν έχει πρόσβαση στα δεδομένα.

Μια άλλη αναλογία

- Το iPod ως αντικείμενο:
 - Έχει κάποια χαρακτηριστικά (φυσικά χαρακτηριστικά, τραγούδια, On/Off, κλπ) και μια κατάσταση.
 - Μπορεί να εκτελέσει συγκεκριμένες δράσεις (TurnOn, TurnOff, Play, Pause, κλπ).
- Στον πραγματικό κόσμο, δεν υπάρχει μια συνάρτηση `Play(iPod,song)` η οποία να παίρνει ως ορίσματα το iPod και ένα τραγούδι. Αντιθέτως ζητάμε να εκτελέσουμε τη **δράση** `iPod.Play(song)`.
- Οι κλάσεις μας προσφέρουν αφαίρεση:
 - Δεν έχει σημασία πως δουλεύει το iPod, αρκεί να εκτελεί τις πράξεις που θέλουμε.
- Στις Δομές Δεδομένων, οι κλάσεις αντιστοιχούν σε **Αφηρημένους Τύπους Δεδομένων**.

Τι μπορούμε να παραστήσουμε ως αντικείμενα

- Φυσικά αντικείμενα
- Ανθρώπους
- Στοιχεία από user interfaces.
- Δομές Δεδομένων
- Συλλογές Δεδομένων
- Νέους τύπους δεδομένων.
- Στοιχεία από παιχνίδια.
- Κλπ...

Συνάθροιση & Σύνθεση Αντικειμένων

Μπορούμε να κατασκευάσουμε αντικείμενα που αποτελούνται από άλλα επιμέρους αντικείμενα – επαναχρησιμοποιώντας κώδικα.....

```
class Door {  
    int code; ...  
    .....  
};
```

```
class Room {  
    Door x;  
    .....  
};
```

```
class Building {  
    Room parts[10];  
    Door main_door;  
};  
  
main () {  
    Building xxx;  
}
```

Με τις κλάσεις ορίζουμε
νέους τύπους δεδομένων

Κληρονομικότητα (Inheritance)

- Φανταστείτε ένα πρόβλημα που έχει ως αντικείμενα πελάτες, υπαλλήλους και άλλα συναφή
- Τόσο οι πελάτες όσο και οι υπάλληλοι έχουν κοινά χαρακτηριστικά και συμπεριφορά (μεθόδους)
 - Θα πρέπει να φτιάξω 2 κλάσεις.
 - π.χ. Ύψος, βάρος κλπ.
 - Θα χω πολλές φορές τον ίδιο κώδικα.

Κληρονομικότητα

- Για να το αποφύγουμε και να έχουμε καλύτερη επαναχρησιμοποίηση δεδομένων και λειτουργιών χρησιμοποιούμε τον **μηχανισμό της κληρονομικότητας**.
 - Ορίζουμε μια **βασική κλάση** (π.χ., Άνθρωπος)
 - Αυτή θα ορίζει τα κοινά χαρακτηριστικά / μεθόδους
 - Η βασική κλάση επαναχρησιμοποιείται για τον ορισμό νέων **παράγωγων** κλάσεων (π.χ. πελάτης, υπάλληλος).
 - Οι παράγωγες κλάσεις κληρονομούν τα χαρακτηριστικά της βασικής κλάσης αλλά έχουν και επιπλέον δικά τους χαρακτηριστικά και μεθόδους.

Σύνοψη

- Ο αντικειμενοστραφής προγραμματισμός προσφέρει ένα πιο κατανοητό μοντέλο προγραμματισμού, στηριγμένο πάνω στις κλάσεις και τα αντικείμενα.
- Το μοντέλο αυτό έχει επιπλέον πλεονεκτήματα
 - Ενθυλάκωση (Encapsulation)
 - Ορισμός νέων τύπων δεδομένων
 - Επαναχρησιμοποίηση κωδικα
 - Κληρονομικότητα
- Σε μεγάλες (και μικρότερες) εταιρείες όλα τα αρκετά μεγάλα συστήματα λογισμικού γίνονται σε γλώσσες αντικειμενοστραφή προγραμματισμού (C++, Java, C#,...).
- Γλώσσες που δεν είναι κλασσικά αντικειμενοστραφής (π.χ., scripting γλώσσες όπως Python), προσφέρουν δυνατότητες αντικειμενοστραφή προγραμματισμού.
- Οι βασικές αρχές είναι κοινές.

ΔΙΑΔΙΚΑΣΙΑΚΟΣ VS. ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΑΦΗΣ

Ένα απλό παράδειγμα

- Έστω ένα απλό πρόγραμμα του χειρίζεται πληροφορία για ένα σύνολο ανθρώπων
 - Κάθε άνθρωπος χαρακτηρίζεται από
 - όνομα
 - ηλικία
 -

Spaghetti υλοποίηση

```
# include <stdio.h>  
# include <stdlib.h>
```

```
struct Person {  
    char * name;  
    int age;  
};
```

```
int main() {  
  
    struct Person x, y;  
  
    x.name = (char *) malloc (10* sizeof(char));  
    x.age = 0;  
  
    /* .... */  
    strcpy (x.name, "Name1");  
    printf ("%s", x.name);  
  
    y.name = (char *) malloc (10* sizeof(char));  
    y.age = 0;  
  
    /* .... */  
    strcpy (y.name, "Name2");  
    printf ("%s", y.name);  
  
    return 0;  
}
```

Προβλήματα

- Ο κώδικας είναι δύσκολο να διαβαστεί και να συντηρηθεί
- Π.χ., έστω ότι κατά την εκτέλεση του προγράμματος προκύπτει πρόβλημα λόγω της απουσίας ελέγχου για το αν έχει επιτύχει η δυναμική δέσμευση μνήμης.

```
int main() {  
  
    struct Person x, y;  
  
    x.name = (char *) malloc (10* sizeof(char));  
    x.age = 0;  
  
    /* .... */  
    strcpy (x.name, "Name1");  
    printf ("%s", x.name);  
  
    y.name = (char *) malloc (10* sizeof(char));  
    y.age = 0;  
  
    /* .... */  
    strcpy (y.name, "Name2"); /* BOOM!!! */  
    printf ("%s", y.name);  
  
    return 0;  
}
```

Debugging

- Για να το επιλύσει πρέπει να αναζητηθούν όλα τα σημεία του κώδικα στα οποία γίνεται δέσμευση μνήμης για το πεδίο name των εκάστοτε μεταβλητών τύπου Person.....
- αυτός που κάνει την αναζήτηση δεν είναι απαραίτητα αυτός που κατασκεύασε το πρόγραμμα
 - ομάδα ελέγχου
 - ομάδα συντήρησης

```
int main() {  
  
    struct Person x, y;  
  
    x.name = (char *) malloc (10* sizeof(char));  
    x.age = 0;  
  
    /* .... */  
    strcpy (x.name, "Name1");  
    printf ("%s", x.name);  
  
    y.name = (char *) malloc (10* sizeof(char));  
    y.age = 0;  
  
    /* .... */  
    strcpy (y.name, "Name2"); /* BOOM!!! */  
    printf ("%s", y.name);  
  
    return 0;  
}
```

Διαδικασιακή λύση I

- Προσθήκη ελέγχου σε **κάθε σημείο**.....

```
int main(){
    struct Person x, y;

    x.name = (char *) malloc (10* sizeof(char));
    if(x.name == NULL) {printf("...."); return 1;}
    x.age = 0;
    /* .... */
    strcpy (x.name, "Name1");
    printf ("%s", x.name);

    y.name = (char *) malloc (10* sizeof(char));
    if(y.name == NULL) {printf("...."); return 1;}
    y.age = 0;

    /* .... */
    strcpy (y.name, "Name2");
    printf ("%s", y.name);

    return 0;
}
```

Διαδικασιακή λύση II

- Καλύτερος σχεδιασμός του προγράμματος
 - κατασκευή μιας συνάρτησης για την αρχικοποίηση μεταβλητών τύπου Person
 - ο κώδικας που δεσμεύει δυναμικά μνήμη θα είναι συγκεντρωμένος σε μια συνάρτηση και όχι διάσπαρτος σε όλο το πρόγραμμα....

Συνάρτηση αρχικοποίησης

```
# include <stdio.h>
```

```
struct Person {  
    char * name;  
    int age;  
};
```

```
void init(struct Person *p){  
    p->name = (char *) malloc (10* sizeof(char));  
    p->age = 0;  
}
```

```
int main() {  
  
    struct Person x, y;  
  
    init(&x);  
    /* .... */  
    strcpy (x.name, "Name1");  
    printf ("%s", x.name);  
  
    init(&y);  
    /* .... */  
    strcpy (y.name, "Name2");  
    printf ("%s", y.name);  
  
    return 0;  
}
```

Debugging

- Για επιλύσουμε το πρόβλημα ελέγχου για δέσμευση μνήμης πρέπει να αναζητηθεί η συνάρτηση στην οποία γίνεται δέσμευση μνήμης για το πεδίο name των εκάστοτε μεταβλητών τύπου Person.....
 - οι συνθήκες αναζήτησης έχουν βελτιωθεί σε σχέση με το spaghetti μοντέλο
 - μόλις βρούμε την συνάρτηση έχουμε τελειώσει.....
 - μία αλλαγή αντί για πολλές
- ...όμως, αυτός που κάνει την αναζήτηση δεν είναι απαραίτητα αυτός που κατασκεύασε το πρόγραμμα
 - Πιθανότατα πρέπει να ελεγχθούν πολλές συναρτήσεις για να βρούμε ποια πρέπει να διορθωθεί

```
# include <stdio.h>
```

```
struct Person {  
    char * name;  
    int age;  
};
```

```
void init(struct Person *p){  
    p->name = (char *) malloc (10* sizeof(char));  
    if (p -> name == NULL) {printf("...."); exit(1);}  
    p->age = 0;  
}
```

```
int main() {  
  
    struct Person x, y;  
  
    init(&x);  
    /* .... */  
    strcpy (x.name, "Name1");  
    printf ("%s", x.name);  
  
    init(&y);  
    /* .... */  
    strcpy (y.name, "Name2");  
    printf ("%s", y.name);  
  
    f();  
  
    return 0;  
}
```

Αντικειμενοστραφής λύση

- Ομαδοποίηση των δεδομένων `Person` και του κώδικα που χρησιμοποιείται για την επεξεργασία αυτών των δεδομένων
 - π.χ. της συνάρτησης `init()`
- ορισμός της `κλάσης` `Person`

Ορισμός κλάσης

```
# include <cstdlib>
# include <cstring>

// αυτό είναι σχόλιο !!

class Person {
    public:
    // τι είναι αυτό το public ??

    char * name; // πεδία - χαρακτηριστικά κλάσης
    int age;

    void init(); // μέθοδοι κλάσης - δήλωση (declaration)
    void set_name(char *n);
    void set_age(int a);
    char *get_name();
    int get_age();

}; // προσοχή στο ερωτηματικό - είναι υποχρεωτικό
```

```
// ορισμός μεθόδων (definition)
```

```
void Person::init() {  
    name = (char *) malloc (10* sizeof(char));  
    // τι είναι αυτό το name ???  
    age = 0;  
}
```

```
void Person::set_name(char *n) {  
    strcpy(name, n);  
}
```

```
void Person::set_age(int a) {  
    age = a;  
    // τι είναι αυτό το age ???  
}
```

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

```
int Person::get_age() {  
    return age;  
}
```

```
int main() {  
  
    Person x, y; // δήλωση αντικειμένου  
  
    x.init(); // κλήση μεθόδου στο αντικείμενο  
  
    /* ..... */  
    x.set_name("Name1");  
    printf ("%s", x.get_name());  
  
    y.init();  
  
    /* ..... */  
    y.set_name("Name2");  
    printf ("%s", y.get_name());  
  
    return 0;  
}
```

Πρόβλημα

- Έστω πάλι πως κατά την εκτέλεση του προγράμματος προκύπτει **πρόβλημα λόγω της απουσίας ελέγχου** για το αν έχει επιτύχει η δυναμική δέσμευση μνήμης.

```
int main() {  
  
    Person x, y; // δήλωση αντικειμένου  
  
    x.init(); // κλήση μεθόδου στο αντικείμενο  
  
    /* .... */  
    x.set_name("Name1");  
    printf ("%s", x.get_name());  
  
    y.init();  
  
    /* .... */  
    y.set_name("Name2"); /* BOOOOMMM */  
    printf ("%s", y.get_name());  
  
    return 0;  
}
```

Debugging

- Για επιλύσουμε το πρόβλημα ελέγχου για δέσμευση μνήμης πρέπει να αναζητηθεί η συνάρτηση στην οποία γίνεται δέσμευση μνήμης για το πεδίο `name` των εκάστοτε μεταβλητών τύπου `Person`
- οι συνθήκες αναζήτησης έχουν βελτιωθεί ακόμα περισσότερο σε σχέση με το διαδικαστικό μοντέλο.
 - από τις δεκάδες / εκατοντάδες συναρτήσεις/μεθόδους του προγράμματος γνωρίζουμε ποιες χρησιμοποιούνται για την επεξεργασία μεταβλητών τύπου `Person` από τη δήλωση της κλάσης.....

```
# include <cstdlib>
# include <cstring>
```

```
class Person {
public:
```

```
    char * name; // πεδία - χαρακτηριστικά κλάσης
    int age;
```

```
    void init(); // μέθοδοι κλάσης
    void set_name(char *n);
    void set_age(int a);
    char *get_name();
    int get_age();
```

```
};
```

**μόνο αυτές οι μέθοδοι
είναι ύποπτες !!!**

```
void Person::init() {  
    name = (char *) malloc (10* sizeof(char));  
    if (name == NULL) {printf(...); exit(1);}  
    age = 0;  
}
```

```
void Person::set_name(char *n) {  
    strcpy(name, n);  
}
```

```
void Person::set_age(int a) {  
    age = a;  
}
```

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

```
int Person::get_age() {  
    return age;  
}
```

```
int main(){  
  
    Person x, y; // δήλωση αντικειμένου  
  
    x.init(); // κλήση μεθόδου στο αντικείμενο  
  
    /* .... */  
    x.set_name("Name1");  
    printf ("%s", x.get_name());  
  
    y.init();  
  
    /* .... */  
    y.set_name("Name2");  
    printf ("%s", y.get_name());  
  
    return 0;  
}
```

καμιά αλλαγή εδώ!!!

Συντήρηση

- Η χρήση κλάσεων από μόνη της λύνει το πρόβλημα μας ????
 - δηλαδή την εύρεση των σημείων του κώδικα στο οποίο πρέπει να προστεθούν έλεγχοι
- Γενικά όχι, αν πρόκειται για ένα **λογισμικό μεγάλης κλίμακας** ο κώδικας που κατασκευάζεται από έναν προγραμματιστή μπορεί να χρησιμοποιηθεί από έναν άλλο προγραμματιστή
 - τι μπορεί να προκύψει σε αυτή την περίπτωση ???

```
# include <stdio>
# include <stdlib>
# include <string>

class Person {
    public:
        char * name;
        int age;

        void init();
        void set_name(char *n);
        void set_age(int a);
        char *get_name();
        int get_age();
};
```

Πρόβλημα

- Ότι (πεδίο / μέθοδο) δηλώνουμε σαν **public** σε μια κλάση μπορεί να χρησιμοποιηθεί σε όλο τον υπόλοιπο κώδικα
- Άρα ένας **απρόσεκτος προγραμματιστής** που θα χρησιμοποιήσει αντικείμενα της κλάσης μας μπορεί **να αρχικοποιεί απευθείας τα πεδία** αυτών των αντικειμένων, ξεχνώντας παράλληλα να κάνει τους απαραίτητους ελέγχους.....

```
int main() {  
  
    Person x, y;  
  
    x.name = (char *) malloc (10* sizeof(char));  
    x.age = 0;  
  
    /* .... */  
    strcpy (x.name, "Name1");  
    printf ("%s", x.name);  
  
    y.name = (char *) malloc (10* sizeof(char));  
    y.age = 0;  
  
    /* .... */  
    strcpy (y.name, "Name2");  
    printf ("%s", y.name);  
  
    return 0;  
}
```

Encapsulation

- Η λύση στο πρόβλημα
 - ανεξέλεγκτη πρόσβαση σε πεδία και μεθόδους μιας κλάσης από άλλους προγραμματιστές
- είναι η αρχή της **ενθυλάκωσης** (encapsulation)
 - σε κάθε κλάση διαχωρίζει ο προγραμματιστής που την υλοποίησε
 - ποια πεδία και μέθοδοι (**public**) μπορούν να χρησιμοποιηθούν από όλο τον υπόλοιπο κώδικα
 - ποια πεδία και μέθοδοι (**private**) μπορούν να χρησιμοποιηθούν μόνο μέσω κλήσης των public μεθόδων της κλάσης...
 - μια συνήθης πρακτική την οποία **πρέπει να ακολουθείτε** είναι ότι **όλα τα πεδία τα δηλώνουμε private**

```
# include <cstdio>
# include <cstdlib>
# include <cstring>
```

```
class Person {
```

```
private:
```

```
    char * name;
```

```
    int age;
```

```
public:
```

```
    void init();
```

```
    void set_name(char *n);
```

```
    void set_age(int a);
```

```
    char *get_name();
```

```
    int get_age();
```

```
};
```

```
int main() {  
  
    Person x, y;  
  
    x.name = (char *) malloc (10* sizeof(char));  
    x.age = 0;  
  
    /* .... */  
    strcpy (x.name, "Name1");  
    printf ("%s", x.name);  
  
    y.name = (char *) malloc (10* sizeof(char));  
    y.age = 0;  
  
    /* .... */  
    strcpy (y.name, "Name2");  
    printf ("%s", y.name);  
  
    return 0;  
}
```

προσπάθεια πρόσβασης
σε private πεδία

COMPILE ERRORS!!!

Compiling

- τα προγράμματα C++ τα αποθηκεύουμε σε αρχεία .cpp
- τα κάνουμε compile χρησιμοποιώντας g++ αντί για gcc

Ένα άλλο παράδειγμα

- Έστω μια εφαρμογή διαχείρισης Αριθμών Φορολογικού Μητρώου (ΑΦΜ)....

Ανάλυση απαιτήσεων...

- δύο βασικές αρμοδιότητες
 - καταχώρηση νέων αριθμών φορολογικού μητρώου
 - ΑΦΜ = ακέραιος αριθμός με 5 ψηφία
 - καταγραφή του πλήθους των ΑΦΜ που καταλήγουν σε 0, 1, 2, ..., 9

Σχεδίαση

- καταχώρηση νέων αριθμών
 - `issueNew(...)` : ανάγνωση και αποθήκευση σε ένα πίνακα A ενός νέου ΑΦΜ
 - `checkValidity(...)` : επαλήθευση αριθμών φορολογικού μητρώου που υπάρχουν στον πίνακα A
 - έλεγχος αν είναι 5ψήφιοι.
- καταγραφή του πλήθους των ΑΦΜ που καταλήγουν σε 0, 1, 2, ..., 9
 - `countByLastDigit(...)` : καταγραφή του πλήθους των ΑΦΜ που καταλήγουν σε 0, 1, 2, ..., 9 σε δεύτερο πίνακα B
 - `printNumbers(...)` : εμφάνιση στην οθόνη του πίνακα B

Υλοποίηση

```
#include <stdio.h>
```

```
void issueNew (int A[], int *size);
```

```
void checkValidity(int A[], int size);
```

```
void countByLastDigit(int A[], int size, int B[]);
```

```
void printNumbers(int B[]);
```

Υλοποίηση

```
int main(){  
    int A[100];  
    int currentSize = 0;  
    int B[10];  
  
    issueNew(A, &currentSize);  
    issueNew(A, &currentSize);  
    issueNew(A, &currentSize);  
  
    checkValidity(A, currentSize);  
  
    countByLastDigit(A, currentSize, B);  
    printNumbers(B);  
  
    return 0;  
}
```

Υλοποίηση

```
void issueNew (int A[], int *size){  
    int input;  
  
    printf("Enter new number: ");  
    scanf("%d", &input);  
    A[*size] = input;  
    (*size)++;  
}
```

Υλοποίηση

```
void checkValidity(int A[], int size){
    int i;
    int length;
    int temp;

    for (i=0; i < size; i++){
        length = 1;
        temp = A[i];
        while(temp > 9){
            temp = temp / 10;
            length++;
        }
        if (length != 5) {
            printf("Error with number in position %d\n", i);
            return;
        }
    }
    printf("ALL OK!\nx");
}
```

Υλοποίηση

```
void countByLastDigit(int A[], int size, int B[]){
    int i;
    int lastDigit;

    for(i = 0; i < 10; i++)
        B[i] = 0;

    for(i = 0; i < size; i++){
        lastDigit = A[i] % 10;
        B[lastDigit]++;
    }
}

void printNumbers(int B[]){
    int i;
    printf("Count by last digit:\n");

    for (i = 0; i < 10; i++){
        printf("%d: %d\n", i, B[i]);
    }
}
```

Συντήρηση

- ...το φορολογικό σύστημα αλλάζει και τα ΑΦΜ γίνονται 6ψήφια#\$%#@!!!!
- ποιες συναρτήσεις επηρεάζονται από αυτή την αλλαγή ?

Συντήρηση

```
void issueNew (int A[], int *size){  
    int input;  
  
    printf("Enter new number: ");  
    scanf("%ld", &input);  
    A[*size] = input;  
    (*size)++;  
}
```

Συντήρηση

```
void checkValidity(int A[], int size){
    int i;
    int length;
    int temp;

    for (i=0; i < size; i++){
        length = 1;
        temp = A[i];
        while(temp > 9){
            temp = temp / 10;
            length++;
        }
        if (length != 5) {
            printf("Error with number in position %d\n", i);
            return;
        }
    }
    printf("ALL OK!\nx");
}
```

```
void checkValidity(int A[], int size){
    int i;
    int length;
    int temp;

    for (i=0; i < size; i++){
        length = 1;
        temp = A[i];
        while(temp > 9){
            temp = temp / 10;
            length++;
        }
        if (length != 5) {
            printf("Error with number in position %d\n", i);
            return;
        }
    }
    printf("ALL OK!\nx");
}
```

Συντήρηση

```
void checkValidity(int A[], int size){
    int i;
    int length;
    int temp;

    for (i=0; i < size; i++){
        length = 1;
        temp = A[i];
        while(temp > 9){
            temp = temp / 10;
            length++;
        }
        if (length != 6) {
            printf("Error with number in position %d\n",
i);
            return;
        }
    }
    printf("ALL OK!\nx");
}
```

```
void countByLastDigit(int A[], int size, int B[]){
    int i;
    int lastDigit;

    for(i = 0; i < 10; i++)
        B[i] = 0;

    for(i = 0; i < size; i++){
        lastDigit = A[i] % 10;
        B[lastDigit]++;
    }
}
```

```
void printNumbers(int B[]){
    int i;
    printf("Count by last digit:\n");

    for (i = 0; i < 10; i++){
        printf("%d: %d\n", i, B[i]);
    }
}
```

Σχεδίαση

- Στον αντικειμενοστρεφή προγραμματισμό για κάθε αρμοδιότητα μιας εφαρμογής ορίζουμε μια νέα κλάση
 - ομαδοποιούμε δεδομένα και συναρτήσεις οι οποίες διαχειρίζονται αυτά τα δεδομένα
 - μια κλάση είναι σαν ένα struct στη C το οποίο αποτελείται από
 - ένα σύνολο πεδίων / χαρακτηριστικών
 - ένα σύνολο συναρτήσεων / μεθόδων

Υλοποίηση

```
#include <stdio>
```

```
class ManageAFM{
```

```
private:
```

```
    int A[100];
```

```
    int size;
```

```
public:
```

```
    void init();
```

```
    void issueNew();
```

```
    void checkValidity();
```

```
    int getLastDigitsAndSize(int LastDigits[]);
```

```
    /* επιστρέφει το size και τα τελευταία ψηφία των A[i] */
```

```
};
```

Υλοποίηση

```
void ManageAFM::init() {  
    size = 0;  
}
```

```
void ManageAFM::issueNew () {  
    int input;  
  
    printf("Enter new number: ");  
    scanf("%d", &input);  
    A[size] = input;  
    size++;  
}
```

Υλοποίηση

```
void ManageAFM::checkValidity() {  
    int i;  
    int length;  
    int temp;  
  
    for (i=0; i < size; i++){  
        length = 1;  
        temp = A[i];  
        while(temp > 9){  
            temp = temp / 10;  
            length++;  
        }  
        if (length != 5) {  
            printf("Error with number in position %d\n", i);  
            return;  
        }  
    }  
    printf("ALL OK!\nx");  
}
```

Υλοποίηση

```
int ManageAFM::getLastDigitsAndSize(int LastDigits[]){  
    int i;  
    for(i = 0; i < size; i++){  
        LastDigits[i] = A[i] % 10;  
    }  
    return size;  
}
```

Υλοποίηση

```
class AFMStatistics{  
private:  
    int B[10];  
public:  
    void init();  
    void countByLastDigit(int LastDigits[], int size);  
    void printNumbers();  
};
```

```
void AFMStatistics::init(){  
    int i;  
    for(i = 0; i < 10; i++)  
        B[i] = 0;  
}
```

Υλοποίηση

```
void AFMStatistics::countByLastDigit(int LastDigits[], int size){
    int i;
    int lastDigit;

    for(i = 0; i < size; i++){
        lastDigit = LastDigits[i];
        B[lastDigit]++;
    }
}

void AFMStatistics::printNumbers(){
    int i;
    printf("Count by last digit:\n");

    for (i = 0; i < 10; i++){
        printf("%d: %d\n", i, B[i]);
    }
}
```

```
int main() {
    ManageAFM manager;
    AFMStatistics stats;

    int LastDigits[100];
    int currentSize;

    manager.init();
    manager.issueNew();
    manager.issueNew();
    manager.issueNew();

    manager.checkValidity();

    currentSize = manager.getLastDigitsAndSize(LastDigits);

    stats.init();
    stats.countByLastDigit(LastDigits, currentSize);
    stats.printNumbers();

    return 0;
}
```

Συντήρηση - τι κερδίσαμε ???

- ποιες μέθοδοι εξαρτώνται από τον πίνακα με τα ΑΦΜ (πίνακας A)

Συντήρηση - τι κερδίσαμε ???

```
void ManageAFM::init() {  
    size = 0;  
}
```

```
void ManageAFM::issueNew () {  
    int input;  
  
    printf("Enter new number: ");  
    scanf("%ld", &input);  
    A[size] = input;  
    size++;  
}
```

Συντήρηση - τι κερδίσαμε ???

```
void ManageAFM::checkValidity() {
    int i;
    int length;
    int temp;

    for (i=0; i < size; i++){
        length = 1;
        temp = A[i];
        while(temp > 9){
            temp = temp / 10;
            length++;
        }
        if (length != 5) {
            printf("Error with number in position %d\n", i);
            return;
        }
    }
    printf("ALL OK!\nx");
}
```

Συντήρηση - τι κερδίσαμε ???

```
int ManageAFM::getLastDigitsAndSize (int LastDigits[]){  
    int i;  
    for(i = 0; i < size; i++){  
        LastDigits[i] = A[i] % 10;  
    }  
    return size;  
}
```

Συντήρηση - τι κερδίσαμε ???

```
#include <cstdio>
```

```
class ManageAFM{
```

```
private:
```

```
    int A[100];
```

```
    int size;
```

```
public:
```

```
    void init();
```

```
    void issueNew();
```

```
    void checkValidity();
```

```
    int getLastDigitsAndSize (int LastDigits[]);
```

```
};
```

Συντήρηση - τι κερδίσαμε ???

- ποιες μέθοδοι εξαρτώνται από τον πίνακα με τα στατιστικά (πίνακας B) ??

Συντήρηση - τι κερδίσαμε ???

```
void AFMStatistics::init() {
    int i;
    for(i = 0; i < 10; i++)
        B[i] = 0;
}

void AFMStatistics::countByLastDigit (int LastDigits[], int size) {
    int i;
    int lastDigit;

    for(i = 0; i < size; i++) {
        lastDigit = LastDigits[i];
        B[lastDigit]++;
    }
}

void AFMStatistics::printNumbers() {
    int i;
    printf("Count by last digit:\n");
    for (i = 0; i < 10; i++) {
        printf("%d: %d\n", i, B[i]);
    }
}
```

Συντήρηση - τι κερδίσαμε ???

```
class AFMStatistics{  
private:  
    int B[10];  
public:  
    void init();  
    void countByLastDigit (int LastDigits[], int  
size);  
    void printNumbers ();  
};
```

Συντήρηση - τι κερδίσαμε ???

- οι μέθοδοι που εξαρτώνται από τα εκάστοτε δεδομένα μιας εφαρμογής είναι αυτές που τα χειρίζονται
- ποιες τα χειρίζονται ??
 - μόνο αυτές που ορίζονται στην ίδια κλάση με τα δεδομένα
 - στην κλάση της οποίας η αρμοδιότητα είναι η διαχείριση των δεδομένων

Συντήρηση - τι κερδίσαμε ???

- άρα για να ανακαλύψουμε **ποιον κώδικα πρέπει να ελέγχουμε μετά από ένα σενάριο συντήρησης που αφορά τα δεδομένα μιας κλάσης** αρκεί να εξετάσουμε **ποιες μέθοδοι ορίζονται σε αυτή την κλάση**