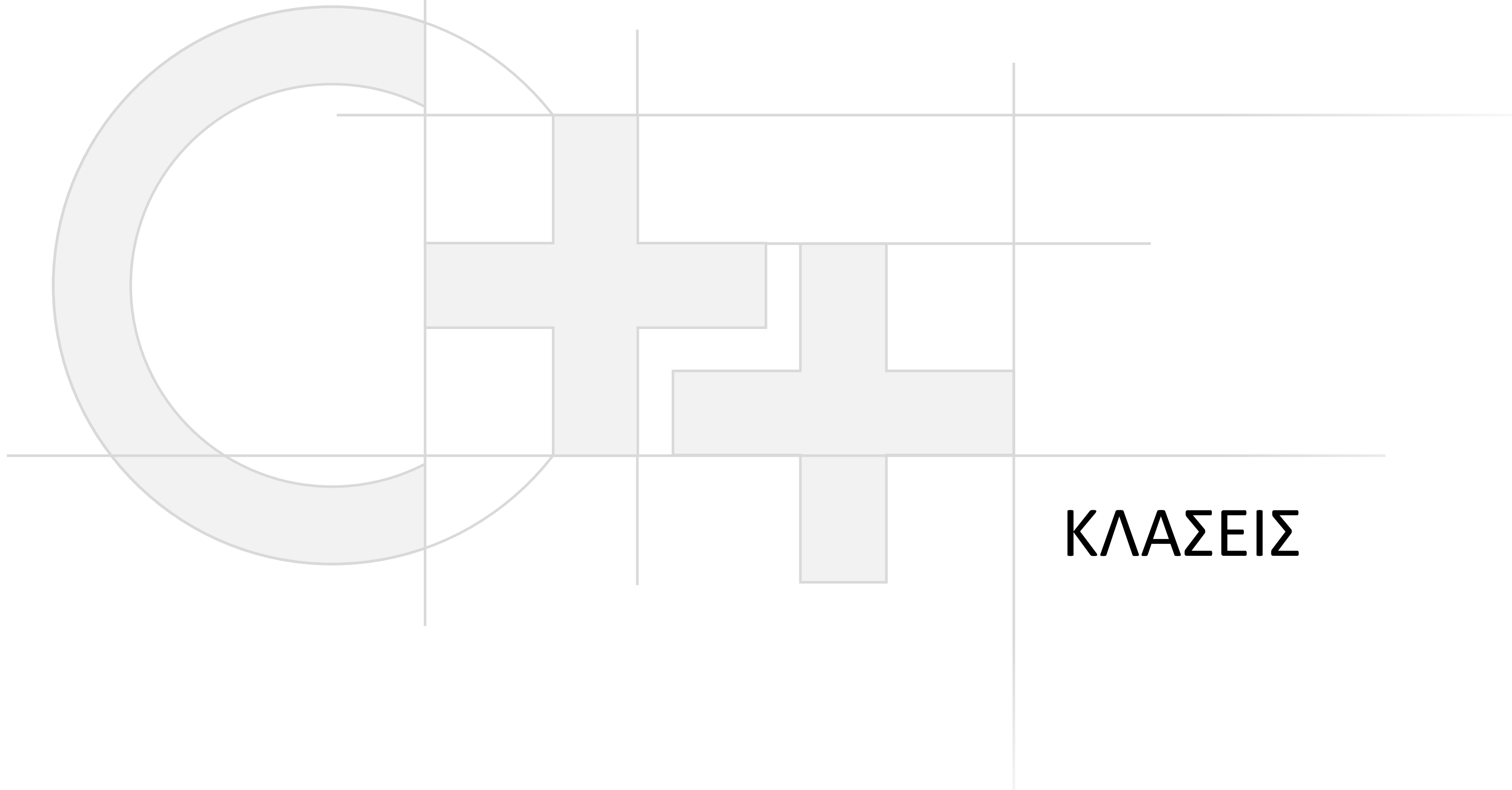
Προγραμματισμός Υπολογιστών με C++



ΚΛΑΣΕΙΣ

Γεώργιος Παπαϊωάννου (2013-20)
gepap@aueb.gr

Περιεχόμενο Παρουσίασης

- Περιγραφή:
  - Οι κλάσεις στη C++
  - Δήλωση και ορισμός μεθόδων
  - Προσδιοριστές πρόσβασης: δημόσια, ιδιωτικά και προστατευμένα μέλη
  - Αρχεία δήλωσης και ορισμού κλάσεων και μεθόδων
  - Κατασκευαστές
  - Καταστροφείς
  - Στατικές Μέθοδοι
- Τελευταία ενημέρωση: Νοέμβριος 2020

Κλάσεις στη C++

- Στη C++ (όχι στη C) έχουμε όπως και στη Java την έννοια της κλάσης (ή τάξης – class).
- Εννοιολογική, ιεραρχική οργάνωση των δεδομένων μας και των πράξεων πάνω σε αυτά
- Η δήλωση μιας απλής κλάσης, χωρίς κληρονομικότητα έχει τη μορφή:

```
class [Όνομα κλάσης] {
   public/protected/private:
      [Τύπος πεδίου] [Όνομα πεδίου];
       …
   public/protected/private:
      [static][Τύπος επιστροφής] [Όνομα μεθόδου](ορίσματα μεθόδου);
      …
};
```

Απλή Κλάση - Παράδειγμα

```
class Person {
public:
    string name;
    unsigned short year;
    unsigned phone;
};   // Προσοχή, χρειάζεται «;».

int main() {
    Person p1, p2;
    p1.name = "Γιάννης";   p2.name = "Μαρία";
    p1.year = 1968;   p2.year = 1970;
    p1.phone = 2108203571;   p2.phone = 2108203572;
    cout << p1.name << ": " << p1.year << ", " << p1.phone << endl
            << p2.name << ": " << p2.year << ", " << p2.phone << endl;
}
```

Hint: Ο προσδιοριστής πρόσβασης (access specifier) στη C++ δε χρειάζεται να επαναλαμβάνεται για κάθε πεδίο ή μέθοδο (εδώ το public ισχύει για όλα τα πεδία που ακολουθούν)

Δήλωση και Ορισμός Μεθόδων

- Η δήλωση των μεθόδων μιας κλάσης γίνεται πάντα μέσα στο σώμα της
- Ο ορισμός (υλοποίηση) των μεθόδων μπορεί να γίνει τόσο μέσα στο σώμα της κλάσης, όσο και έξω από αυτή

Δήλωση και Ορισμός Μεθόδων - Παράδειγμα

```
class Person {
public:
    string name;
    unsigned short year;
    unsigned phone;
    void greeting() {
        cout << "Το όνομά μου είναι " << name << endl;
    }
    void tellAge();
};

void Person::tellAge() {
     cout << "Είμαι " << year;
     cout << year==1? "χρόνου": "χρονών" << endl;
}
```

Υλοποίηση μεθόδου εντός κλάσης. Συνήθως το κάνουμε μόνο για πολύ μικρές μεθόδους.

Υλοποίηση μεθόδου εκτός κλάσης. Χρειάζεται το όνομα της κλάσης ακολουθούμενο από :: πριν το όνομα της μεθόδου για να προσδιορίζει σε ποια κλάση ανήκει η συνάρτηση

Δήλωση και Ορισμός Μεθόδων – Καλή Πρακτική

- Είναι καλή πρακτική να κρατάμε τις υλοποιήσεις των μεθόδων χωριστά από τις δηλώσεις (σε αρχεία .cpp και .h) αντίστοιχα, διότι:
  – Αν κρατάμε την υλοποίηση μέσα στο αρχείο τη δήλωσης της κλάσης, κάθε φορά που θα αλλάζουμε τον κώδικα μιας μεθόδου (αλλά όχι την υπογραφή της) θα επηρεάζονται όλα τα αρχεία που την κάνουν include και θα πρέπει να ξαναμεταγλωττιστούν. Σε αντίθετη περίπτωση, όχι.
- Με αυτό τον τρόπο, έχουμε πλήρη διαχωρισμό της δήλωσης (υπογραφής/interface) μιας κλάσης και της υλοποίησής της
  – Μπορούμε να έχουμε πιο εύκολα και ευανάγνωστα διαφορετικές υλοποιήσεις μεθόδων στο/στα cpp αρχεία είτε κάνοντας επιλεκτική μεταγλώττιση άλλων cpp αρχείων για το ίδιο header είτε μέσα στο ίδιο cpp αρχείο με #ifdef/#else/#endif δηλώσεις

Δημόσια, Προστατευμένα και Ιδιωτικά Μέλη

- Ισχύει ο προσδιορισμός προσβασιμότητας σε μέλη όπως και στη Java
  - Δεν υπάρχει η έννοια του «package protected» μέλους αλλά
  - Υπάρχει η έννοια της «φίλιας» κλάσης και μεθόδου (friend) που παρέχει μεγαλύτερη ευελιξία (θα τα δούμε στη συνέχεια)
- Δημόσια (public): Πρόσβαση από όλους, εντός κι εκτός της κλάσης
- Ιδιωτικά (private): Πρόσβαση μόνο από άλλα μέλη της ίδιας κλάσης
- Προστατευμένα (protected): Πρόσβαση από μέλη της ίδιας κλάσης ή των απογόνων της (βλ. κληρονομικότητα)

Έλεγχος Πρόσβασης

- Είναι καλή πρακτική να «κλείνουμε» την πρόσβαση στα εσωτερικά δεδομένα της κλάσης:
- Κάνουμε public ΜΟΝΟ μεθόδους που χρειάζεται ο προγραμματιστής για να αλληλεπιδράσει με την κλάση (*), ποτέ πεδία
- Δηλώνουμε και υλοποιούμε accessor / modifier (get/set) μεθόδους για να διαβάσουμε και αλλάξουμε ΜΟΝΟ όσα πεδία χρειάζεται
  - Πολλά πεδία μιας κλάσης αφορούν μόνο την εσωτερική της λειτουργία. Γι αυτά δεν επιτρέπουμε καθόλου την πρόσβαση εκτός κλάσης
  - Κατά το σχεδιασμό μιας κλάσης, σκεφτόμαστε πρώτα ποια πεδία μας θα είναι αποκλειστικά δεδομένα της κλάσης αυτής (private) ή απαιτούνται από απογόνους για τη λειτουργία τους (protected)

(*) Εκτός κι αν μια κλάση είναι τετριμμένη και έχει μόνο λίγα, απλά πεδία που δε χρειάζονται έλεγχο τιμών κατά τις αναθέσεις και χρήση αυτών

Έλεγχος Πρόσβασης – Παράδειγμα (1)

```
class Person {
private:                     // Μόνο οι μέθοδοι της Person έχουν πρόσβαση σε αυτά:
   string name;
   unsigned short year;
   void test();
public:                      // Σε αυτά έχουν πρόσβαση όλοι:
   void setData(const string& n, unsigned short y);
   void print();
};

void Person::setData(const string& n, unsigned short y) {
   name = n;                    // OK. Οι μέθοδοι της Person έχουν πρόσβαση
   year = y;    }               // στις ιδιωτικές μεταβλητές της Person.

void Person:: print() {
   cout << name << ": " << year << endl;
   test();                      // OK. Οι μέθοδοι της Person έχουν πρόσβαση στις
}                               // ιδιωτικές μεθόδους της Person.
```

Έλεγχος Πρόσβασης – Παράδειγμα (2)

```
void Person::test() {
   cout << "δοκιμή" << endl;
}

int main() {
   Person p1, p2;
   p1.name = "Γιάννης";        // Λάθος. Το name είναι ιδιωτικό μέλος.
   p1.set("Γιάννης", 1968);    // ΟΚ.  Καλώ δημόσιες μεθόδους
   p2.set("Μαρία", 1970);      // της Person.
   p1.print();                 // Τυπώνει «Γιάννης: 1968 δοκιμή».
   p2.print();                 // Τυπώνει «Μαρία: 1970 δοκιμή».
   cout << p1.name;            // Λάθος. Το name είναι ιδιωτικό μέλος.
   p1.test();                  // Λάθος. Η test() είναι ιδιωτικό μέλος.
}
```

Έλεγχος Πρόσβασης – Γιατί;

- Η ελεγχόμενη πρόσβαση και ανάθεση τιμών σε πεδία μέσω ενός ρεπερτορίου public μεθόδων και μόνο:
  - Μας επιτρέπει να κρατάμε τα δεδομένα μας συνεπή, ειδικά αν κάποια πεδία της κλάσης εξαρτώνται από άλλα
  - Κάνει μονοσήμαντη και ευκολότερη τη σωστή δέσμευση/αποδεύσμευση μνήμης και μας προφυλάσσει από πρόσβαση σε μη αρχικοποιημένη μνήμη
  - Καθιστά δυνατό τον έλεγχο τιμών και τη διόρθωση παραμέτρων κατά το πέρασμά τους (range/type checking)
  - Κρύβει στο χρήστη της κλάσης την εσωτερική υλοποίηση των δομών της και μας επιτρέπει απρόσκοπτη αλλαγή του κώδικά της, χωρίς να επηρεάζεται η σχέση της κλάσης με τον υπόλοιπο κώδικα

Σκοπιμότητα Ελέγχου Πρόσβασης – Παράδειγμα (1)

```
class Vector {
public:
    float x, y;
    inline float length() { return sqrt(x*x+y*y); }   // ακριβή πράξη
};

int main()
{
    Vector v1, v2;
    v1.x = 1.f; v1.y = 3.f;
    v2.x = -2.f; v2.y = 5.f;
    if (v1.length()>0.0f)                        // ο υπολογισμός καλείται
        cout << v2.length() / v1.length();       // πολλές φορές για τα ίδια
}                                                // δεδομένα → σπατάλη

// … ας ψάξουμε για μια διαφορετική υλοποίηση…
```

Και οι μέθοδοι, όπως και οι συναρτήσεις μπορούν να είναι inline.

Σκοπιμότητα Ελέγχου Πρόσβασης – Παράδειγμα (2)

```
class Vector {
private:
    float m_length;
public:
    float x, y;
    inline void updateLength() { m_length = sqrt(x*x+y*y);}
    inline float length() { return m_length;}
};
int main() {
    Vector v1, v2;
    v1.x = 0.f; v1.y = 3.f;
    v2.x = -2.f; v2.y = 5.f;
    v1.updateLength(); v2.updateLength(); // α) απαιτείται να ενημερώσουμε
                                          //    μόνοι μας τα αντικείμενα!
    v1.y = 0.0f;                          // β) πλέον το v1.length δεν είναι
    if (v1.length()>0.0f)                 //    σωστό!
        cout << v2.length() / v1.length();
}
// … ας ψάξουμε για μια διαφορετική υλοποίηση…
```

Σκοπιμότητα Ελέγχου Πρόσβασης – Παράδειγμα (3)

```
class Vector {
private:
    float m_x, m_y;
    float m_length;
    inline void updateLength() { m_length = sqrt( m_x*m_x + m_y*m_y );}
public:
    void setCoordinates(float x, float y);
    inline float getX() {return m_x;}
    inline float getY() {return m_y;}
    inline float length() { return length;}
};

void Vector::setCoordinates(float x, float y) {
    if (m_x==x && m_y==y) // Έλεγχος παραμέτρων
        return;           // lazy evaluation. Ενημέρωση μόνο αν αλλάξει κάτι
    m_x = x;
    m_y = y;
    updateLength();       // πλέον γίνεται αυτόματα ενημέρωση των
}                         // συσχετισμένων πεδίων
```

Σκοπιμότητα Ελέγχου Πρόσβασης – Παράδειγμα (4)

```
int main()
{
    Vector v1, v2;
    v1.setCoordinates(0, 3);
    v2.setCoordinates(-2, 5);
    v1.setCoordinates(0,0); // ή προσθέτουμε και μια setX(), set(Y) για
                            // ευκολία
    if (v1.length()>0.0f)                    // πλέον ελέγχεται ορθά και
       cout << v2.length() / v1.length();   // δεν εκτελείται
}
```

Private/Protected Πεδία – Απομόνωση Δεδομένων

```
class Text {
  private:
    int num_lines;
    char * text_buffer[80];
  public:
    void addLine(const char * line);
    void addLine(const string line);
  …
};
```

A

```
class Text {
  private:
    int num_lines;
    string text_buffer[80];
  public:
    void addLine(const char * line);
    void addLine(const string line);
  …
};
```

B

Μπορώ να αλλάξω την εσωτερική αναπαράσταση των δεδομένων μου και να μην επηρεαστεί το Interface της κλάσης, άρα δε χρειάζεται καμία αλλαγή ο κώδικας που αλληλεπιδρά μαζί της

Private/Protected Πεδία – Απομόνωση Δεδομένων (2)

A

```
void Text::addLine(const char * line)
{
    if (num_lines<80)
    {
        int len = strlen(line);
        text_buffer [num_lines] = new char[len+1];
        strcpy(text_buffer[num_lines++], line);
    }
}
```

B

```
void Text::addLine(const char * line)
{
    if (num_lines<80)
    {
        text_buffer [num_lines++] = line;
    }
}
```

Εμβόλιμη Παρατήρηση

- Στον παραπάνω κώδικα είδαμε ότι την ίδια δουλειά την κάναμε αντικαθιστώντας 3 γραμμές κώδικα με 1. Όμως:
  - Αυτό δε σημαίνει ότι η 1 αυτή συνάρτηση (τελεστής =) δεν κρύβει μεγάλη εσωτερική πολυπλοκότητα!
  - Για παράδειγμα, η χρήση της βιβλιοθήκης STL, ειδικά για I/O, μπορεί να είναι έως και 2 τάξεις μεγέθους πιο αργή από τις αντίστοιχες συναρτήσεις C
- Συμπέρασμα: Αν θέλουμε υψηλή απόδοση:
  - Ελέγχουμε 2-3 εναλλακτικές υλοποιήσεις
  - Προτιμάμε χαμηλού επιπέδου συναρτήσεις, αν είμαστε διατεθειμένοι να γράψουμε λίγο κώδικα παραπάνω
  - Προτιμάμε να έχουμε πλήρη επίγνωση και ακριβή έλεγχο των πράξεων και της δέσμευσης/αποδέσμευσης μνήμης που συντελείται

Προσδιοριστές Πρόσβασης – Default Συμπεριφορά;

- Αν δε δηλώσουμε προσδιοριστή πρόσβασης σε κάποια από τα μέλη μιας κλάσης, αυτά είναι private
- Πλέον, ένα struct (βλ. βασικούς τύπους) στη C++ συμπεριφέρεται κι αυτό σαν κλάση με τη διαφορά ότι ο default προσδιοριστής πρόσβασης είναι public

Τελεστές Πρόσβασης

- Είδαμε ότι για πρόσβαση στα πεδία και τις μεθόδους μιας κλάσης χρησιμοποιούμε τον τελεστή "." :

```
Person p1; p1.print();
```

- Αν η μεταβλητή μας είναι δείκτης σε αντικείμενο (π.χ. Person * p1), τότε αντί να γράψουμε (*p1).print(), μπορούμε να γράψουμε κατευθείαν:

```
p1->print();
```

Μέθοδοι με Ορίσματα Ίδιας Κλάσης

```
class Point {
    float x, y;
public:
    Point(float xx = 0, float yy = 0) { x = xx; y = yy; }
    void print() const { cout << x << "," << y << endl; }
    void add(const Point& p) { x += p.x; y+= p.y; }
};

int main() {
    Point p1(1, 2), p2(3, 4);
    p1.add(p2);
    p1.print();   // Τυπώνει «4,6».
    p2.print();   // Τυπώνει «3,4».
}
```

Αρχεία Κεφαλίδων και Κλάσεις (1)

Person.h

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

class Person {
private:
   string name;
   unsigned short year;
public:
   void set(const string& n, unsigned short y) {
      name = n; year = y}
   void print();
};
```

Αρχεία Κεφαλίδων και Κλάσεις (2)

Person.cpp

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

void Person::print() {   // Δεν είναι inline.
   cout << name << ": " << year << endl;
}
```

Αρχεία Κεφαλίδων και Κλάσεις (3)

MyApp.cpp

```
#include "person.h"

int main() {
   Person p1, p2;
   p1.set("Γιάννης", 1968);
   p2.set("Μαρία", 1970);
   p1.print();    // Τυπώνει «Γιάννης: 1968»
   p2.print();    // Τυπώνει «Μαρία: 1970»
}
```

Προβλήματα με Αρχεία Κεφαλίδων

- Όπως και με τις απλές συναρτήσεις, έτσι και με τις δηλώσεις των κλάσεων έχουμε το πρόβλημα της πολλαπλής έμμεσης εισαγωγής αρχείου κεφαλίδας και άρα της πολλαπλής δήλωσης

αρχείο mylib1.h:

```
#include "person.h"

void f(Person p);
```

αρχείο mylib2.h:

```
#include "person.h"

void g(Person p);
```

αρχείο mynewapp.cpp:

```
#include "person.h"     // Το person.h περιλαμβάνεται 3 φορές και
#include "mylib1.h"     // ο ορισμός της τάξης person δίνεται
#include "mylib2.h"     // 3 φορές στο mynewapp.cpp. Λάθος!
void h(Person p) {
   f(p);
   g(p);
}
```

Αντιμετώπιση Προβλημάτων Αρχείων Κεφαλίδων

- Όπως και με τις απλές συναρτήσεις:

Person.h

```
#ifndef  _PERSON_H_    // Αν δεν έχει δημιουργηθεί η σταθερά
                       // _PERSON_H_ του προεπεξεργαστή...
#define _PERSON_H_     // Δημιουργία της σταθεράς.
#include <string>
using namespace std;
class Person {
private:
   string name;   unsigned short year;
public:
   void set(const string& n, unsigned short y) {
      name = n; year = y}
   void print();
};
#endif
```

Κυκλική Εξάρτηση Δηλώσεων

αρχείο car.h:

```
#pragma once
#include "driver.h"

class Car
{
    protected:
        Driver * drivers;
};
```

αρχείο driver.h:

```
#pragma once
#include "car.h"

class Driver
{
    protected:
        Car * cars;
};
```

Ένα από τα δύο δε θα μπορέσει να μεταγλωττιστεί σωστά!

Κυκλική Εξάρτηση Δηλώσεων - Λύση

αρχείο car.h:

```
#once
#include "driver.h"

class Car
{
    protected:
        class Driver * drivers;
};
```

αρχείο driver.h:

```
#once
#include "car.h"

class Driver
{
    protected:
        class Car * cars;
};
```

Ο compiler περιμένει να συναντήσει τη δήλωση αργότερα

αρχείο car.cpp:

```
#include "car.h"
#include "driver.h"

Car::Car()
{
    drivers = new Driver[10];
}
```

αρχείο driver.cpp:

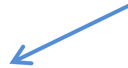
```
#include "driver.h"
#include "car.h"

Driver::Driver()
{
    cars = new Car[3];
}
```

Κατασκευαστές

- Γιατί είναι αναγκαίοι;
  - Αρχικοποίηση τιμών πεδίων
  - Κλήση αντίστοιχων κατασκευαστών περιεχόμενων αντικειμένων
  - Στη C++ τα πεδία δεν αρχικοποιούνται σε κάποια default τιμή, οπότε πάντα πρέπει να τους προσδίδουμε μια τιμή αρχικοποίησης

Αναγκαιότητα Κατασκευαστών - Παράδειγμα

Στη C++11 επιτρέπονται και τέτοιες αρχικοποιήσεις.



```
class Person {
    string name;
    unsigned short year;
    unsigned short year = 1980;
public:
    void setName(const string& n) { name = n; }
    void setYear(unsigned short y) { year = y; }
    void print() { cout << name << ": " << year << endl; }
};

int main() {
    Person p1;
    // Το p1.name δεν έχει αρχικοποιηθεί.
    p1.print();   // Τυπώνει «σκουπίδια» για το name(τυχαίες τιμές).
}
```

Κατασκευαστές – Τρόπος Δήλωσης

```
class Person {
    string name;
    unsigned short year;
public:
    void setName(const string& n) { name = n; }
    void setYear(unsigned short y) { year = y; }
    void print() { cout << name << ": " << year << endl; }
    Person(const string& n, unsigned short y);   // Κατασκευαστής
    Person() {name= ""; year=0;}         // default κατασκευαστής
};
Person::Person(const string& n, unsigned short y) {
    name = n; year = y; }

int main() {
    Person p1; // καλείται ο default constructor αυτόματα
    Person p2 = Person("Γιώργος", 1975);
    p2.print();        // Τυπώνει «Γιώργος: 1975».
}
```

Προεπιλεγμένοι (Default) Κατασκευαστές

- Αν δε δηλώσουμε κάποιον κατασκευαστή, δημιουργούνται αυτόματα από το μεταγλωττιστή 2 τυπικοί κατασκευαστές:
  - Ο προεπιλεγμένος κατασκευαστής (default constructor): Δεν παίρνει κανένα όρισμα
  - Ο κατασκευαστής αντιγραφής (copy constructor) : Παίρνει ως όρισμα ένα άλλο στιγμιότυπο της ίδιας κλάσης και αντιγράφει τα πεδία του
- Αν δε δώσουμε εμείς δικό μας default constructor, αυτός που δημιουργείται, δεν κάνει τίποτα!
- Αν δώσουμε οποιονδήποτε άλλο κατασκευαστή με ορίσματα και δε δηλώσουμε default constructor, τότε ΔΕ δημιουργείται ο πρώτος

Προεπιλεγμένοι Κατασκευαστές – Παράδειγμα (1)

```
class Person {
   string name;
   unsigned short year;
};                     // Δημιουργείται αυτόματα ο default constructor

int main() {
   Person p1;             // καλείται ο default constructor αυτόματα
   Person p2 = Person(); // καλούμε ρητά τον default constructor
}
```

Προεπιλεγμένοι Κατασκευαστές – Παράδειγμα (2)

```
class Person {
    string name;
    unsigned short year;
    Person (string n, unsigned int y) {name=n; year=y;}
};                              // Δε δημιουργείται default constructor
                                // αφού έχουμε δηλώσει άλλον constructor

int main() {
    Person p1;                  // Σφάλμα! Δεν υπάρχει default constructor
    Person p2 = Person(); // Ομοίως
    Person p3 = Person("Γκόλφω", 23);
    Person p4("Γκόλφω", 23);              // Ισοδύναμη αρχικοποίηση

}
```

Κατασκευαστές με Προεπιλεγμένες Τιμές

```
class Person {
   string name;
   unsigned short year;
public:
   void setName(const string& n) { name = n; }
   void setYear(unsigned short y) { year = y; }
   void print() { cout << name << ": " << year << endl; }
   // Παίζει και το ρόλο προεπιλεγμένου κατασκευαστή:
   Person(const string& n = "?", unsigned short y = 1980) {
      name = n; year = y; }
};

int main() {
   Person p1; // OK. Χρησιμοποιεί το μοναδικό κατασκευαστή.
   // Person p1 = Person(); // Ισοδύναμο με το προηγούμενο.
   Person p2 = Person("Γιώργος", 1975); // OK.
   p1.print();  p2.print(); // Τυπώνει "?: 1980", "Γιώργος: 1975".
}
```

Λίστα Αρχικοποίησης (Initialization List)

- Είναι το σημείο στο οποίο μπορούμε να δώσουμε αρχικές τιμές σε:
  - Πεδία
  - Κατασκευαστές προγόνων της κλάσης
- Είναι το μοναδικό σημείο στο οποίο μπορούμε να αρχικοποιήσουμε αμετάβλητα (immutable) πεδία:
  - Const πεδία
  - Πεδία που είναι αναφορές
- Η λίστα αρχικοποίησης εκτελείται πριν τον κώδικα του κατασκευαστή

Λίστα Αρχικοποίησης – Που χρησιμεύει;

- Αρχικοποίηση των πεδίων ενός αντικειμένου
- Κλήση του κατασκευαστή της κλάσης προγόνου με συγκεκριμένα ορίσματα

Λίστα Αρχικοποίησης - Δήλωση

```
class Person {
   string name;
   unsigned short year;
public:
   void setName(const string& n) { name = n; }
   void setYear(unsigned short y) { year = y; }
   void print() { cout << name << ": " << year << endl; }
   // Παίζει και το ρόλο προεπιλεγμένου κατασκευαστή:
   Person() :
      name("unspecified"), year(1)
   {}
};
```

Αρχικοποίηση σε μορφή "κατασκευαστή" ακόμα και για τους βασικούς τύπους (θα δούμε γιατί όταν μιλήσουμε για templates)

Αναδιάταξη: Ο μεταγλωττιστής αναδιατάσσει τη σειρά των αναθέσεων τιμών σύμφωνα με τη σειρά δήλωσης των πεδίων στην κλάση, ανεξάρτητα από το πώς τα δίνουμε στη λίστα αρχικοποίησης

Στατικά Πεδία

- Όπως και στη Java, έτσι και στη C++ μπορώ να έχω στατικά πεδία (static)
- Δε συγχέουμε τα const με τα static:
  - const πεδίο: ορίζεται μια φορά για κάθε αντικείμενο, είναι ξεχωριστό για κάθε στιγμιότυπο της κλάσης και δεν αλλάζει
  - static: ορίζεται σε επίπεδο κλάσης, είναι κοινό για όλα τα αντικείμενα μιας κλάσης και μπορεί να αλλαχθεί ανά πάσα στιγμή

Σταθερές

```
class Person {
    string name;
    unsigned short year;
    const unsigned short min_year;
public:
    void setName(const string& n) { name = n; }
    void setYear(unsigned short y) { year = y; }
    void print() { cout << name << ": " << year << endl; }
    // Παίζει και το ρόλο προεπιλεγμένου κατασκευαστή:
    Person() : min_year(1910) {
    ...
    }
    Person(unsigned short min_year) : min_year(min_year) {
    ...
    }

};
```

Μόνο στη λίστα αρχικοποίησης μπορεί να προσδιοριστεί η τιμή ενός const πεδίου

Δεν υπάρχει σύγχυση ονομάτων

Αρχικοποίηση Στατικών Πεδίων

- Η δήλωση του στατικού πεδίου γίνεται στη δήλωση της κλάσης
- Η αρχικοποίηση της τιμής του στατικού πεδίου γίνεται στο αρχείο υλοποίησης, υποχρεωτικά μόνο μια φορά:

Foo.h:

```
class foo {
  private:
    static int i;
};
```

Foo.cpp:

```
int foo::i = 0;
```

Αντικείμενα και Δυναμική Δέσμευση Μνήμης

```
void someFunction() {
    Person p1, p2("Γιάννης", 1975);
    Person* q1 = new Person;
    Person* q2 = new Person("Μαρία", 1984);
    p1.print(); p2.print(); // Τυπώνει «?: 1980», «Γιάννης: 1975».
    (*q1).print();          // Τυπώνει: «?: 1980».
    q1->print();            // Ισοδύναμο με το προηγούμενο.
    (*q2).print();          // Τυπώνει: «Μαρία: 1984».
    q2->print();            // Ισοδύναμο με το προηγούμενο.
    delete q1; delete q2;   // Τα αντικείμενα που έχουν
                            // δημιουργηθεί με new δεν καταστρέφονται
                            // αυτόματα στο τέλος της συνάρτησης.
}
```

Τελεστής «A->B» : Αν A είναι δείκτης σε class/struct, ισοδύναμο με(*A).B

Πίνακες Αντικειμένων

```
Person p1[3];              // Χρήση προεπιλεγμένου κατασκευαστή.
Person p2[] = {           // Αυτόματος υπολογισμός αριθμού θέσεων.
        Person("Γιώργος", 1975),
        Person("Μαρία", 1980) };

Person p3[3] = {                    // Χρήση προεπιλεγμένου
        Person("Γιώργος", 1975),  // κατασκευαστή για την 3η θέση.
        Person("Μαρία", 1980) };

Person* q = new Person[2];  // Δυναμική καταχώριση με χρήση
                            // του προεπιλεγμένου κατασκευαστή.
q[0].setName("Γιώργος");   q[0].setYear(1975);   …
```

Μέθοδοι και const

```
const Person p1;    // Το p1 δεν μπορεί να αλλάξει.
p1.print();         // Ο μεταγλωττιστής δεν ξέρει ότι η print()
                    // δεν αλλάζει το p1.

class Person {     // Βελτίωση για να αντιμετωπιστεί το πρόβλημα.
   public:
      void print() const { cout << name << ": " << year << endl; }
   ...
};

p1.print(); // OK τώρα ξέρουμε ότι η print() δεν αλλάζει το p1.
```

- Μια const μέθοδος δεν επιτρέπεται να αλλάζει τις μεταβλητές-μέλη ή να καλεί μη-const μεθόδους του ίδιου αντικειμένου.

Μέθοδοι και const - Προσοχή

Επιστρέφει const string

```
const string ToString();
```

≠

```
string ToString() const;
```

Η μέθοδος είναι const

Καταστροφείς

- Στη C++ υπάρχει η έννοια του destructor (καταστροφέα)
- Είναι μέθοδος που καλείται πριν αποδεσμευθεί ένα αντικείμενο από τη μνήμη
- Σε ιεραρχίες κλάσεων, καλεί και τον καταστροφέα του προγόνου

Γιατί Χρειαζόμαστε Καταστροφείς;

- Στη Java, ο Garbage Collector περιοδικά ελέγχει για unreferenced objects και τα αποδεσμεύει από το χώρο δυναμικής εκχώρησης μνήμης
- Στη C++ είμαστε εμείς υπεύθυνοι γι αυτή τη λειτουργία
- Είδαμε ότι ένα αντικείμενο που δεσμεύω με new, μπορώ να το αποδεσμεύσω με delete
- Τι γίνεται όμως με μνήμη που δεσμεύεται εσωτερικά στις μεθόδους (π.χ. στον κατασκευαστή);
  - Αυτή πρέπει να την καθαρίσω πριν διαγράψω το αντικείμενο από τη μνήμη, αλλιώς θα συνεχίσει να είναι δεσμευμένη, έστω και αν διέγραψα το αντικείμενο!

Αναγκαιότητα Κατασκευαστών - Παράδειγμα

```
class MultiPerson {
   unsigned howManyNames;
   string* names;           // Θα δείχνει στην 1η θέση δυναμικού πίνακα.
public:
   MultiPerson(unsigned num = 1);
   void setName(unsigned i, const string& n) { names[i] = n; }
   string getName(unsigned i) const { return names[i]; }
};

MultiPerson::MultiPerson(unsigned num) {
   howManyNames = num;
   names = new string[num];}   // Δυναμική καταχώριση μνήμης.

void f() {
   MultiPerson p(2);           // Δύο θέσεις ονομάτων.
   p.setName(0, "Γιάννης");
   p.setName(1, "Ιωάννης");
}                  // Καταστρέφεται το p (μαζί και ο δείκτης p.names),
                   // αλλά όχι ο χώρος στον οποίο δείχνει το p.names...
```

Καταστροφείς - Δήλωση

```
class MultiPerson {
   unsigned howManyNames;
   string* names;
public:
   MultiPerson(unsigned num = 1);
   ~MultiPerson();    // Καταστροφέας.
   void setName(unsigned i, const string& n) { names[i] = n; }
   string getName(unsigned i) const { return names[i]; }
};

MultiPerson::~MultiPerson() {
   cout << "deleting" << endl;
   delete []names;
}

void f() {
   MultiPerson p(2);
   p.setName(0, "Γιάννης");  p.setName(1, "Ιωάννης");
}  // OK, τώρα όταν τελειώνει η f καλείται ο καταστροφέας του p.
```

Κλήση Καταστροφέων

- **Δεν τους καλούμε απευθείας** στο πρόγραμμά μας (*)
- Για αντικείμενα που έχουν δημιουργηθεί με **στατική εκχώριση μνήμης** (στη stack), ο καταστροφέας καλείται **αυτόματα** όταν το αντικείμενο παύει να υπάρχει (π.χ. στο τέλος της f στο προηγούμενο παράδειγμα).
- Για αντικείμενα που έχουν δημιουργηθεί με **δυναμική καταχώριση μνήμης** (με new), ο καταστροφέας καλείται **όταν εκτελείται η εντολή delete**

(*) εξαίρεση όταν κάνουμε "placement new", όπου είναι ο μοναδικός τρόπος να κληθεί ο καταστροφέας

Κλήση Καταστροφέα με delete - Παράδειγμα

…

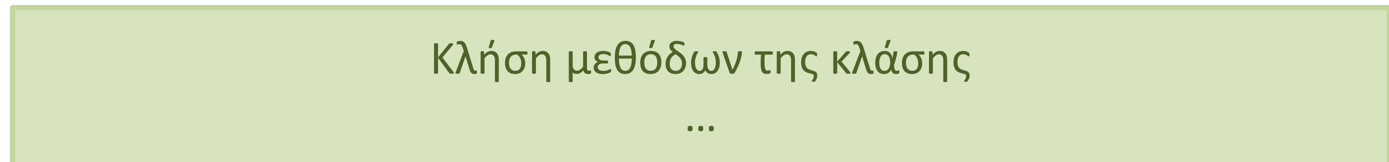
```
void f() {
   MultiPerson* q = new MultiPerson(2);
   q->setName(0, "Γιάννης");  q->setName(1, "Ιωάννης");
   ...
   delete q;
}
```

Καλείται ο καταστροφέας του MultiPerson (αφού υπάρχει), ο οποίος αποδεσμεύει τον πίνακα names. Στη συνέχεια, επιστρέφει ο κατασκευαστής και διαγράφεται το αντικείμενο q.

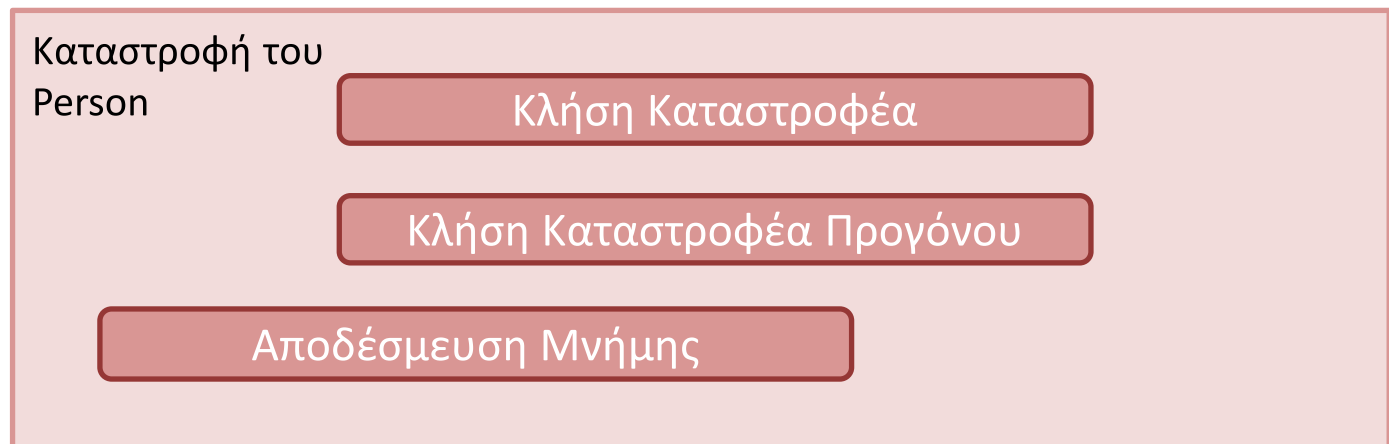
Σύνοψη: Ο Κύκλος Ζωής ενός Αντικειμένου

```
void Foo() {
   Person * p = new Person();
   Person q = Person();
```

```
p->print();
q.print();
```




```
  delete p;
}
```



Στατικές Μέθοδοι

- Στατικές μέθοδοι είναι αυτές που ενώ δηλώνονται στο σώμα μιας κλάσης, ούτε χρησιμοποιούν, ούτε αλλάζουν μη στατικά πεδία της κλάσης αυτής
- Οι στατικές μέθοδοι είναι στην ουσία «ελεύθερες» συναρτήσεις που εννοιολογικά υπάγονται σε μια κλάση
- Δε χρειάζονται στιγμιότυπο της κλάσης (αντικείμενο) για να εκτελεστούν

Στατικές Μέθοδοι - Παράδειγμα

```
class vec3 {
private:
    float x, y, z;
public:
    static vec3 crossProduct (vec3 a, vec3 b) {
        float x,y,z;
        x = a.y*b.z – a.z*b.y;
        y = a.z*b.x – a.x*b.z;
        z = a.x*b.y – a.y*b.x;
        return vec3(x,y,z);
    }
};

...

vec3 v1 = vec3::cross( vec3(0,2,0), vec3(-1,0,1) );
```

Εδώ τα x,y,z είναι οι τοπικές μεταβλητές x,y,z που έχουν προτεραιότητα σε σχέση με τα πεδία.

Unions (1)

- Στη C++ υπάρχει μηχανισμός μια κλάση (ή struct) να χρησιμοποιεί επικαλυπτώμενα πεδία!
- Αυτό γίνεται με την έννοια της «ένωσης» (union)
- Ουσιαστικά τα πεδία αυτά καταλαμβάνουν κοινό (όχι απαραίτητα ίσο) χώρο στη μνήμη και μπορούμε να διαλέγουμε κατά την εκτέλεση σε πιο να αναφερόμαστε
- Συνήθης χρήση: Πολλαπλά ονόματα για πεδία

Unions (2)

- Παράδειγμα Δήλωσης

```
class Vec3
{
  public:
    union { float x, r; };
    union { float y, g; };
    union { float z, b; };
};
```

- Παράδειγμα Χρήσης

```
Vec3 a {0.1f, 0.2f, 0.3f};
// Νέος initializer στη C++11

cout << a.x;    // 0.1
cout << a.r;    // 0.1

Vec3 b;
b.x = a.r;
cout << b.r;  // 0.1
```