HY352 : ΤΕΧΝΟΛΟΓΙΑ ΛΟΓΙΣΜΙΚΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΔΑΣΚΩΝ
Αντώνιος Σαββίδης

CSD

ΕΝΟΤΗΤΑ 4

ΣΤΟΙΧΕΙΑ ΟΝΤΟΚΕΝΤΡΙΚΟΥ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

Αριθμός διαλέξεων 7, Διάλεξη 4η

CSD

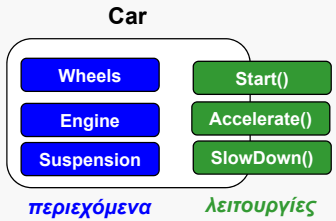
Περιεχόμενα

- *Αφαιρετικές κλάσεις*
- Δυναμική αντιστοίχηση
  - Late / dynamic binding
- Πολυμορφισμός
- Δείκτες στιγμιότυπων και μετατροπές τύπων
  - type casting

CSD

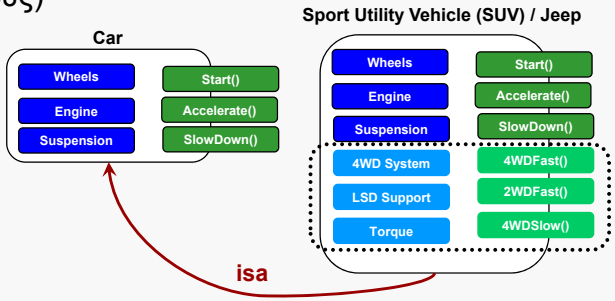
Αφαιρετικές κλάσεις (1/12)

- Αφαίρεση είναι ο μηχανισμός μέσω του οποίου οι ομοιότητες μεταξύ διαφορετικών οντοτήτων εντοπίζονται και τυποποιούνται



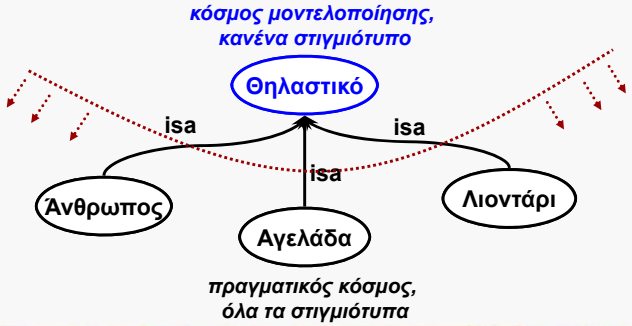
Αφαιρετικές κλάσεις (2/12)

- Μέσα από το πρίσμα των αυθεντικών οντοτήτων, η αφαίρεση συνιστά ουσιαστικά μείωση των επιμέρους χαρακτηριστικών και λειτουργιών (δηλ. *δεδομένων* και *συναρτήσεων*, όσον αφορά την προγραμματιστική εικόνα τους)



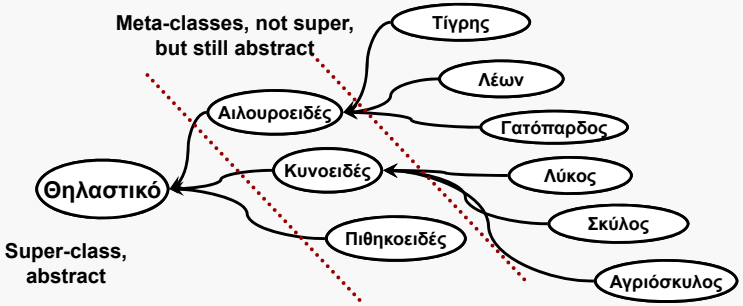
Αφαιρετικές κλάσεις (3/12)

- Στην πράξη, μια αφαιρετική οντότητα ***δεν συνιστά ποτέ μια υπαρκτή ή βιώσιμη οντότητα***, αλλά κυρίως αντιπροσωπεύει την *τομή* των χαρακτηριστικών όλων των αντίστοιχων στιγμιότυπων (δηλ. των πραγματικών οντοτήτων)



Αφαιρετικές κλάσεις (4/12)

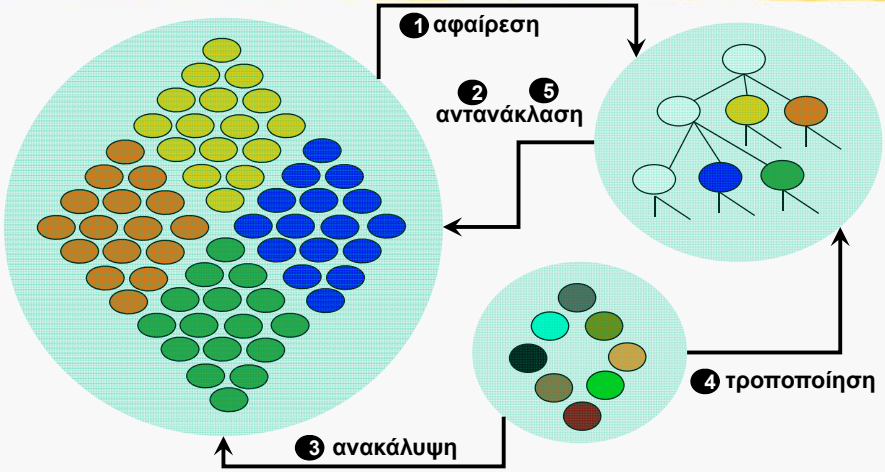
- Αφαιρέσεις μπορούν να οριστούν και σε πολλαπλά επίπεδα. Καθώς κινούμαστε χαμηλότερα στην ιεραρχία, οι αφαιρέσεις αναπαριστούν εξειδικεύσεις της κληροδότη κλάσης, με την εισαγωγή επιπλέον χαρακτηριστικών και λειτουργιών



Αφαιρετικές κλάσεις (5/12)

- *Η επιστήμη και τέχνη της αφαίρεσης,*
  - ***...στον πραγματικό κόσμο***
    - Η αφαίρεση είναι κυρίως μία καλά-ορισμένη σχολαστική και συστηματική διαδικασία, στην οποία εφαρμόζεται εκ βαθέων ανάλυση και κατηγοριοποίηση *υπαρχόντων οντοτήτων* σε κατάλληλες συνομοταξίες (αφαιρέσεις)
    - Σε ορισμένες περιπτώσεις νέες οντότητες, άγνωστες μέχρι πρότινος, εισάγονται, οδηγώντας κάποιες φορές σε τροποποίηση των συνομοταξιών. Αυτός είναι ο μηχανισμός της *ανακάλυψης*.
    - Η βασική αξία της αφαίρεσης είναι περισσότερο για την κατανόηση, παρά για την δημιουργία (αυτό κατά τη θρησκεία είναι έργο Θεού)

Αφαιρετικές κλάσεις (6/12)



Αφαιρετικές κλάσεις (7/12)

- *Η επιστήμη και τέχνη της αφαίρεσης,*
  - ***...στον κόσμο του λογισμικού***
    - Η αφαίρεση σημαίνει ότι οραματιζόμαστε, προβλέπουμε, αναλύουμε, και πιστοποιούμε τα χαρακτηριστικά, ιδιότητες, περιπτώσεις χρήσης, μετεξέλιξη, και ανάπτυξη οντοτήτων οι οποίες *ακόμη δεν υφίστανται*.
    - Μερικές φορές εμφανίζεται η ανάγκη για νέες οντότητες, για τις οποίες θα πρέπει να αλλάξουμε την υπάρχουσα ιεραρχία. Αυτός είναι ο μηχανισμός της *καταστροφής της σχεδίασης*.
    - Η κύρια αξία της αφαίρεσης είναι για την κατασκευή οντοτήτων οι οποίες θα ενσωματωθούν στον κόσμο του λογισμικού (αυτό κατά την επιστήμη μας είθισται να είναι έργο του προγραμματιστή)

Αφαιρετικές κλάσεις (8/12)

- *Ως προγραμματιστικό μοντέλο (1/2)*
  - Κλάσεις (συνομοταξίες) ως τύποι δεδομένων
  - Υποστήριξη κληρονομικότητας σε πολλαπλά επίπεδα
  - Οι αφηρημένες κλάσεις ονομάζονται και μετακλάσεις
    - Ένα υπάρχει μία αφηρημένη κλάση πάνω από όλες, αυτή συνήθως λέγεται υπερκλάση (superclass)
  - ***Αναμένουμε λιγότερα δεδομένα και περισσότερες συναρτήσεις***
    - ➔Μπορούμε να θεωρούμε τα δεδομένα ως φυσικά χαρακτηριστικά και τις συναρτήσεις ως συμπεριφορές. Οι συμπεριφορές είναι πιο αφηρημένες από τα χαρακτηριστικά: ***όλα τα όντα έχουν κοινά βασικά ένστικτα, αλλά όχι κοινά φυσικά χαρακτηριστικά***
    - ➔Σε αρκετές γλώσσες αυτή η ιδιότητα οδηγεί σε ένα ειδικό είδος αφαίρεσης για τον ορισμό ομοιογένειας διαφορετικών κλάσεων μέσω ***interfaces***

Αφαιρετικές κλάσεις (9/12)

- *Ως προγραμματιστικό μοντέλο (2/2)*
  - Οι αφηρημένες κλάσεις δεν υποστηρίζουν δημιουργία στιγμιότυπων (*δεν μπορούμε να δημιουργήσουμε αφηρημένα αντικείμενα*), αλλά
  - υποστηρίζουν την προγραμματιστική διαχείριση στιγμιότυπων κληρονόμων κλάσεων μέσω τύπων αναφορών η δεικτών των αφηρημένων κλάσεων
    - Αυτό επιτρέπει τα διαφορετικά στιγμιότυπα διαφορετικών κληρονόμων κλάσεων *D1,...,Dn*, οι οποίες κληρονομούν από μία κοινή αφηρημένη κλάση *A*
    - να χρησιμοποιούνται στο πρόγραμμά μας μέσω του κοινού interface που προσφέρει η αφηρημένη κλάση *A*

Αφαιρετικές κλάσεις (10/12)

```
class Shape;
class Displayer {
   public:
   static void Add (const Shape& shape);
   static void Display (void);
};
class Shape {
   public:
   virtual void Display(void) const = 0;   ← Pure virtual function
   virtual void Move(int dx, int dy) {     ← Normal virtual function
      x += dx, y += dy;
      Displayer::Add(*this); // If not already inside
   }
};
void Displayer::Display (void) {    ← Πολυμορφική συνάρτηση
   for each const Shape& x in display list do
      x.Display();                          ← Καλείται η derived Display
   clear display list;
}
```


Αφαιρετικές κλάσεις (11/12)

- *Αφηρημένες συναρτήσεις*
  - Δυναμική αντιστοίχιση εφαρμόζεται μόνο σε συναρτήσεις με τον χαρακτηρισμό **virtual**. Ειδάλλως, η αυθεντική αντίστοιχη συνάρτηση στην κλάση του στιγμιότυπου, με το οποίο πραγματοποιείται η κλήση, καλείται.
  - Ο χαρακτηρισμός **=0**, δηλ. αφηρημένη συνάρτηση, επιτρέπεται μόνο σε **virtual** συναρτήσεις. *Για τις αφηρημένες συναρτήσεις, ορισμός (δηλ. η υλοποίησή τους) δεν επιτρέπεται*.
  - Εάν μία κλάση περιέχει αφηρημένες συναρτήσεις, δεν επιτρέπεται η δημιουργία στιγμιότυπων (δηλ. είναι αφηρημένη κλάση).
  - Εάν σε αφηρημένη κληρονομημένη συνάρτηση, η κληρονόμος κλάση δεν παρέχει επίσης ορισμό, τότε αυτομάτως και αυτή καθίσταται αφηρημένη.
  - Ο χαρακτηρισμός virtual δεν επιτρέπεται για static συναρτήσεις μίας κλάσης.
  - Μία virtual συνάρτηση είναι πάντα virtual για τις κληρονόμους κλάσεις, είτε γίνει refined ή όχι, είτε ορισθεί ξανά ως virtual ή όχι.

Αφαιρετικές κλάσεις (12/12)

παραδείγματα αφηρημένων συναρτήσεων

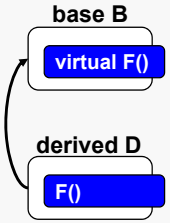
```
class Base {
   public:
   void F(void)=0;           // Error, μη virtual αφηρημένη συνάρτηση
   virtual void F (void)=0;  // Έτσι είναι το σωστό.
   virtual void G(void)=0;   // Ok, virtual αφηρημένη συνάρτηση
   virtual void G(char*)=0;  // Ok, υπερφόρτωση αφηρημένης συνάρτησης
   void F(int a);            // Ok, απλή συνάρτηση, υπερφόρτωση αφηρημένης
};

class Derived : public Base {
   public:
   virtual void G(void) override {}  // Υλοποίηση της αφηρημένης 'void Base::G(void)=0'
   void G(char*){}           // Το ίδιο για την 'void Base::G(char*)=0'
};

Base b;                      // Error, η 'Base' είναι αφηρημένη κλάση
Derived d;                   // Error, 'void Derived::F(void)=0' undefined
```


Περιεχόμενα

- Αφηρημένες κλάσεις
- *Δυναμική αντιστοίχηση*
  - Late / dynamic binding
- Πολυμορφισμός
- Δείκτες στιγμιότυπων και μετατροπές τύπων
  - type casting

Δυναμική αντιστοίχηση (1/7)

- Η χρήση αφηρημένων κλάσεων και κληρονομικότητας είναι στενά συνδεδεμένη με την έννοια της δυναμικής αντιστοίχησης (late binding)
  - *Στην πραγματικότητα, χωρίς την υποστήριξη δυναμικής αντιστοίχησης, η προγραμματιστική αξία των αφηρημένων κλάσεων είναι μηδαμινή*



1. Έστω **B::F()** δήλωση μίας virtual συνάρτησης.
2. Η **B::F()** μπορεί να έχει προαιρετικά και ορισμό.
3. Ας είναι **d** στιγμιότυπο της κλάσης **D**.
4. Ας είναι **b** αναφορά στιγμιότυπου της κλάσης **B**, με αρχική τιμή **b = d**.

⇒ *Τότε η κλήση **b.F()** καλεί την **D::F()**.*

Δυναμική αντιστοίχηση (2/7)

- *Συνοψίζοντας, λοιπόν, είναι ο μηχανισμός αυτός ο οποίος προγραμματιστικά μας επιτρέπει*:
  - να χρησιμοποιούμε δείκτες της base κλάσης για αναφορά σε στιγμιότυπα derived κλάσεων, στη θέση των διαφορετικών τύπων δεικτών των διαφορετικών derived κλάσεων
  - να καλούμε virtual συναρτήσεις της base κλάσης
  - και να εξασφαλίζεται ότι οι αναθεωρημένες εκδόσεις των συναρτήσεων, της εκάστοτε derived κλάσης του κάθε στιγμιότυπου, θα καλούνται πάντα κατά την εκτέλεση

Δυναμική αντιστοίχηση (3/7)

παράδειγμα

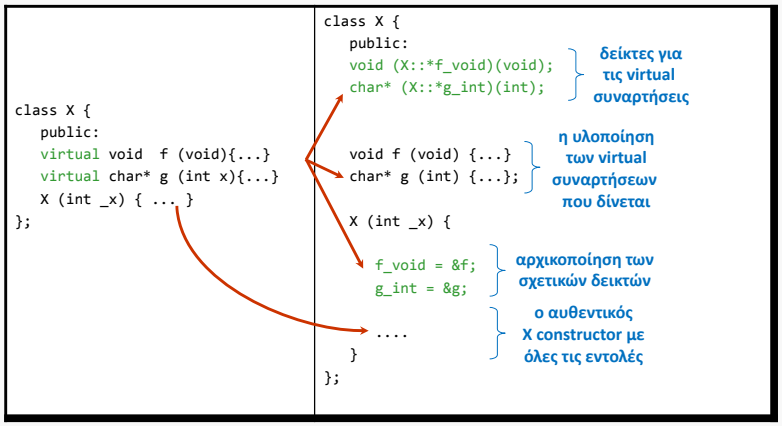
```
class Base {
   public:
   virtual void VirtualDisplay(void) { printf("BaseVirtual\n"); }
   void         Display(void)        { printf("BaseNormal\n"); }
};

class Derived : public Base {
   public:
   virtual void VirtualDisplay(void) { printf("DerivedVirtual\n"); }
   void         Display(void)        { printf("DerivedNormal\n"); }
};

Derived derived;
((Base*) &derived)->Display();         // Χρήση ως base-class pointer, μέσω
((Base*) &derived)->VirtualDisplay();  // up-casting

Ο παραπάνω κώδικας θα εκτυπώσει:
BaseNormal
DerivedVirtual
```


Δυναμική αντιστοίχηση (4/7)

- *Εσωτερικά μυστικά (1/4)*
  - Θα δείξουμε πως ουσιαστικά υλοποιείται ο μηχανισμός της δυναμικής αντιστοίχησης,
    - θα παράγουμε αντίστοιχο κώδικα σε C++ που δεν θα περιέχει καθόλου virtual συναρτήσεις, αλλά θα συμπεριφέρεται ακριβώς όπως ο αυθεντικός κώδικας με virtual συναρτήσεις
    - Παρόμοια τακτική υιοθετείται (με επιπλέον προσθήκες – χρήση πίνακα που λέγεται virtual table / vtable) για την παραγωγή κώδικα «μηχανής» στην πλειονότητα των compilers οντοκεντρικών γλωσσών προγραμματισμού
      - *Προσοχή στο ιδιόρρυθμο αλλά απαραίτητο συντακτικό των δεικτών σε συναρτήσεις – μέλη*

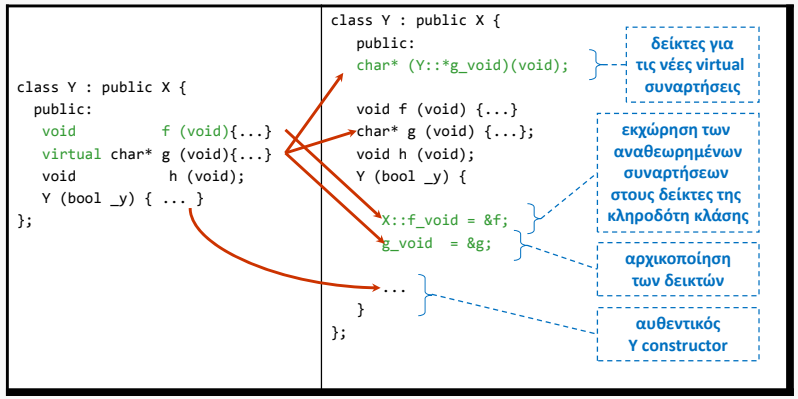
Δυναμική αντιστοίχηση (5/7)

- *Εσωτερικά μυστικά (2/4)*



Δυναμική αντιστοίχηση (6/7)

- *Εσωτερικά μυστικά (3/4)*



Δυναμική αντιστοίχηση (7/7)

- *Εσωτερικά μυστικά (4/4)*

```
X x;                    X x;
x.f();                  x.f();    // Κλήση μέσω instance variable.
x.g(10);                x.g(10);  // Το ίδιο (δεν συνάγει late binding).

Y y;                    Y y;
y.h();                  y.h();                    // Κλήση απλής συνάρτησης
X* xp = (X*) &y;        X* xp = (X*) &y;
xp->f();                (xp->*(xp->f_void))();    // Κλήση virtual συνάρτησης
xp->g();                (xp->*(xp->g_int))();     // Κλήση virtual συνάρτησης
```

Μετατροπή του κώδικα κλήσης, μέσω στιγμιότυπων, των συναρτήσεων

Ένθετο – δείκτες σε συναρτήσεις

- *Διαχείριση δεικτών σε συναρτήσεις μέλη (pointers to member functions) παράγοντας C κώδικα*

```
class X {
  public:
  void f (int a);
};

1: void (X::*pmf)(int);
2: pmf = &X::f;
3: X x;
4: (x.*pmf)(10);
5: X* xPtr = &x;
6: (xPtr->*pmf)(20);
```

```
struct X {};
void X_f_int (X* this, int a);



1: void (*pmf)(X*, int);
2: pmf = &X_f_int;
3: X x;
4: (*pmf)(&x, 10);
5: X* xPtr = &x;
6: (*pfm)(xPtr, 20);
```

Οι pointers σε member functions γίνονται pointers σε functions (πάντα global) στη γλώσσα C

Ένθετο – δείκτες σε μεταβλητές

- *Διαχείριση δεικτών σε μεταβλητές μέλη (pointers to member variables) παράγοντας C κώδικα*

```
class X {                    struct X { int a, b; };
  public:                    X.a: field at offset 0
  int a, b;                  X.b: field at offset 4
};
X x1,x2;                     X x1,x2;
1: int X::* p;               1: unsigned p;     ← member points are offsets
2: p = &X::a;                2: p = 0;          ← carry a offset in X layout
3: x1.*p = 7;                3: *((int*) (((unsigned char*) &x1) + p)) = 7;   → value =0
4: x2.*p = 20;               4: *((int*) (((unsigned char*) &x2) + p)) = 20;
5: p = &X::b;                5: p = 4;                                        → value =4
6: x1.*p = 12;               6: *((int*) (((unsigned char*) &x1) + p)) = 7;
```

Οι pointers σε member variables γίνονται offsets (unsigned) στη γλώσσα C

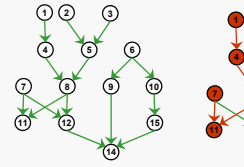
Περιεχόμενα

- Αφηρημένες κλάσεις
- Δυναμική αντιστοίχηση
  - Late / dynamic binding
- *Πολυμορφισμός*
- Δείκτες στιγμιότυπων και μετατροπές τύπων
  - type casting

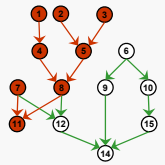
Πολυμορφισμός (1/4)

- *Προγραμματιστικός πολυμορφισμός είναι η δυνατότητα κατασκευής λειτουργικών τμημάτων που μπορούν να εφαρμόζονται σε ανοικτό σύνολο από αντικείμενα*
  - εφόσον αυτά υποστηρίζουν συντακτικά και σημασιολογικά ένα κοινό API (interface)
- Οι αφηρημένες κλάσεις και ο μηχανισμός της δυναμικής αντιστοίχησης είναι ένα καλό υπόβαθρο για την υποστήριξη πολυμορφισμού
- Οι κλάσεις που περιέχουν virtual συναρτήσεις ονομάζονται *πολυμορφικές κλάσεις*
- Αντικείμενα μίας derived κλάσης ονομάζονται πολυμορφικά

Πολυμορφισμός (2/4)

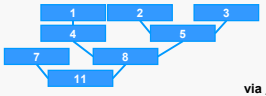


Sample inheritance acyclic graph (we also call it *hierarchy* or *scheme*)

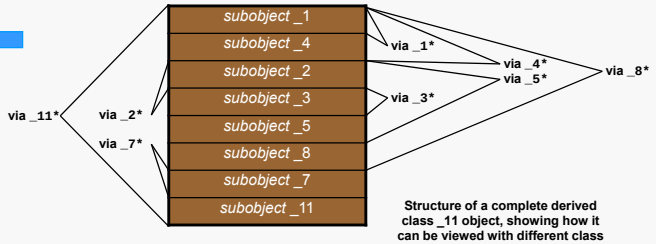


Acyclic inheritance subgraph for class 11

```
struct _1 {};
struct _2 {};
struct _3 {};
struct _5 : public _2, public _3 {};
struct _4 : public _1 {};
struct _8 : public _4, public _5 {};
struct _7 {};
struct _11 : public _7, public _8 {};
```



Composition of a derived class _11 object from its bases



Structure of a complete derived class _11 object, showing how it can be viewed with different class pointers (polymorphic)

CSD

Πολυμορφισμός (3/4)

- Ο πολυμορφισμός είναι ένα πολύ έξυπνο προγραμματιστικό τέχνασμα καθώς
  - επιτρέπει τον προγραμματισμό με σχετική απλότητα αλγορίθμων επαναχρησιμοποιήσιμων «όπως είναι» για ένα δυνητικά απεριόριστο φάσμα κλάσεων
    - αυτών που κληρονομούν από τις κλάσεις που αναφέρονται στον αλγόριθμο
  - επιπλέον οδηγεί σε απλούστερο, ευκρινέστερο και ευκολότερα συντηρήσιμο πηγαίο κώδικα
    - εξαλείφουμε κλήσεις συναρτήσεων σε εξειδικευμένες κλάσεις και ασχολούμαστε μόνο με base class ή ακόμη και abstract classes

CSD

Πολυμορφισμός (4/4)

παράδειγμα

```
const unsigned MAX_SHAPES = 1024;

class Displayer {
 private:
   static Shape* shapes [MAX_SHAPES];
   static unsigned shapeSerial;
 public:
   static bool Add (Shape* shape);
   static void Display (void);
};

class Circle : public Shape {...};
class Triangle : public Shape {...};

Circle circle(10,10,20);
Triangle triangle(0,0,30,30, 40,40);

circle.Move(-4,-4);
triangle.Move(0,10);
Displayer::Display();
```

```
Πολυμορφικοί αλγόριθμοι, οι οποίοι δουλεύουν
σωστά για στιγμιότυπα κλάσεων που
κληρονομούν από την κλάση Shape →

bool Displayer::Add (Shape* shape) {
   if (shapeSerial = =MAX_SHAPES)
      return false;
   else {
      shapes[shapeSerial++] = shape;
      return true;
   }
}

void Displayer::Display (void) {
   while (shapeSerial)
     shapes[--shapeSerial]->Display();
}
```


CSD

Περιεχόμενα

- Αφηρημένες κλάσεις
- Δυναμική αντιστοίχηση
  - Late / dynamic binding
- Πολυμορφισμός
- *Δείκτες στιγμιότυπων και μετατροπές τύπων*
  - type casting

CSD

Δείκτες στιγμιότυπων και μετατροπές τύπων (1/9)

- Η μετατροπή από δείκτη κληρονόμου κλάσης σε δείκτη κληροδότη κλάσης ονομάζεται μετατροπή προβιβασμού - ***up casting***.
  - Στη C++, δεν είναι απαραίτητο να γράψετε type casting για μετατροπή τύπου εάν εκχωρείτε δείκτη κληρονόμου κλάσης σε δείκτη κληροδότη κλάσης
    - συνιστάται όμως να γράφετε πάντα την μετατροπή για αυτό-τεκμηρίωση του κώδικα
- Η μετατροπή αντίθετης κατεύθυνσης ονομάζεται μετατροπή υποβιβασμού - ***down casting***.
  - Στη C++, τέτοιου είδους αυτόματη μετατροπή δεν υποστηρίζεται, και χρειάζεται να γράψετε την επιθυμητή μετατροπή τύπων με type casting

CSD

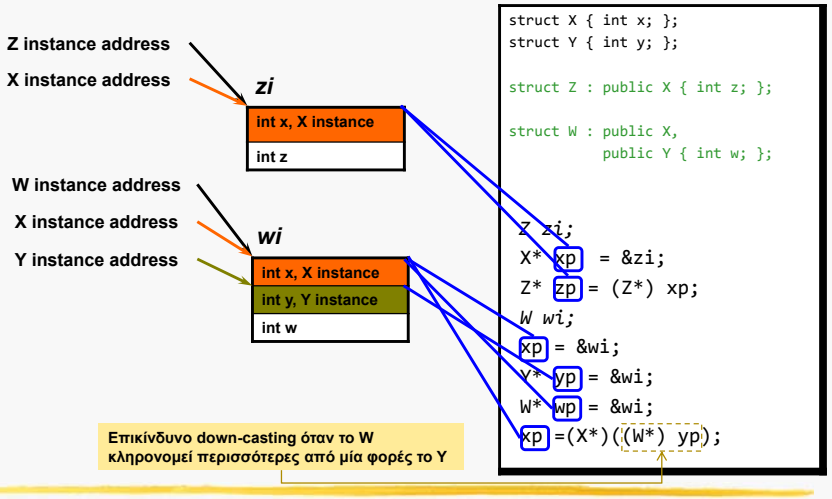
Δείκτες στιγμιότυπων και μετατροπές τύπων (2/9)

παραδείγματα

```
class X {
   public: X(void){} ...
};
class Y : public X {
   public: Y(void){} ...
};

X xInst;
Y yInst;
X* xPtr = &xInst;        // Ιδια  κλάση, δεν χρειάζεται type casting
Y* yPtr;
yPtr = xPtr;             // Error, αυτόματο down casting δεν υποστηρίζεται
yPtr = (Y*) xPtr;        // Ok, down casting, αλλά ο yPtr είναι επικίνδυνος !
xPtr = &yInst;           // Ok, up casting, και χωρίς πρόβλημα
xPtr = yPtr;             // Ok, up casting, σωστό ακόμη και με τον yPtr
```


CSD

Δείκτες στιγμιότυπων και μετατροπές τύπων (3/9)



CSD

Δείκτες στιγμιότυπων και μετατροπές τύπων - ένθετο

```
class Castable {
         virtual void* Downcast (void) = 0;
};

class X : public Castable {
         virtual void* Downcast (void) { return this; }
};

class Y : public Castable {
         virtual void* Downcast (void) { return this; }
};

class W :public X, public Y {
         virtual void* Downcast (void) { return this; }
};

W w;
X* x = (X*) &w;                      ←up casting
Y* yy = (Y*) ((W*) x->Downcast());   ←cross down casting
Y* y = (Y*) &w;                      ←up casting
W* wp = (W*) x->Downcast();          ←down casting
```

Αυτός είναι ο ασφαλής τρόπος για down-casting to bottom class (most-derived) σε multiple / single inheritance

CSD

Δείκτες στιγμιότυπων και μετατροπές τύπων (4/9)

- Η C++ προσφέρει μερικούς εδικούς τελεστές για μετατροπές τύπων δεικτών / αναφορών. Οι πιο κοινοί (παραλείπεται ένας) είναι:
  - *static_cast<T>(e)*, με σημασιολογία ταυτόσημη της κοινής σκόπιμης μετατροπής τύπου *(T) e*
  - *dynamic_cast<T>(e)*, που μπορεί να χρησιμοποιείται και για ασφαλές down casting, και επιτρέπει την μετατροπή του *e* σε *T*, εάν το *e* είναι δείκτης / αναφορά κλάσης *TT*, με *TT* πολυμορφικό κληροδότη του *T*
    - εάν η μετατροπή δεν υφίσταται, επιστρέφεται null
  - *const_cast<T>(e)*, ο οποίος αφαιρεί από ένα αμετάβλητο / σταθερό αντικείμενο *e*, τύπου *const T*, τον χαρακτηρισμό *const*

CSD

Δείκτες στιγμιότυπων και μετατροπές τύπων (5/9)

παραδείγματα

```
class X {
   public: virtual void f (void); ← Πολυμορφική κλάση
   X(void){}
};

class Y : public X {
   public: void f (void);
   Y(void){}
};

X xInst;
Y yInst;
X* xPtr = &xInst;                // Αυτόματη μετατροπή
Y* yPtr;
yPtr = static_cast<Y*>(xPtr);    // Το ίδιο με (Y*) xPtr, επισφαλές
yPtr = dynamic_cast<Y*>(xPtr);   // Εδώ όμως πρέπει να επιστραφεί null
xPtr = dynamic_cast<X*>(&yInst); // Επιτυχημένο up casting
xPtr = dynamic_cast<Y*>(xPtr);   // Επιτυχημένο down casting
```


CSD

Δείκτες στιγμιότυπων και μετατροπές τύπων (6/9)

Η σωστή χρήση του down casting

```
class Super {…};
class Derived : public Super {…};

void MyClass::foo (Super* s) {
Option-1: Μόνο εάν έχετε φροντίσει να καλείται πάντοτε η foo με Derived objects
assert(dynamic_cast<Derived*>(s));
Derived* d = (Super*) s;

Option-2: Μόνο όταν είναι καλή ορισμένη η λογική του else
Derived* d = dynamic_cast<Derived*>(s);
if (d) // was really a Derived object
  { /* do what needs to be done */ }
else
   what we do in this case?
}

Option-3: Δικός μας τρόπος επιβεβαίωσης κλάσης – έχει νόημα όταν έχουμε εναλλακτικές κλάσεις (πιο γρήγορο)
if (s->GetClassId() == "Derived1") {
   assert(dynamic_cast<Derived1*>(s));
   Derived1* d = (Super*) s;
} else similar ifs for every Derived2…Derivedn
```


CSD

Δείκτες στιγμιότυπων και μετατροπές τύπων (7/9)

- *Μυστικά στις μετατροπές δεικτών (1/3)*
  - Θα δούμε μία απλή υλοποίηση του up casting και down casting ενός επιπέδου με αντικατάσταση των βασικών ειδικών τελεστών
    - Το up casting ουσιαστικά βασίζεται στο απλό type casting
    - Το down casting απαιτεί κάποια τυποποιημένα μέλη σε κάθε κλάση που κατασκευάζετε

Προαπαιτούμενη απλή βοηθητική κλάση:

```
class PointerBag {
   public:
   void Add (void* ptr);
   void Remove (void* ptr);
   bool In (void* ptr) const; // hashing
};
```


CSD

Δείκτες στιγμιότυπων και μετατροπές τύπων (8/9)

- *Μυστικά στις μετατροπές δεικτών (2/3)*

```
class X : public Y, public Z {
   private:
   static PointerBag instsX;
   static PointerBag instsY;
   static PointerBag instsZ;

   void Register (void) {
      instsX.Add(this);
      instsY.Add((Y*) this);
      instsZ.Add((Z*) this);
   }
   void Cancel (void) {
     instsX.Remove(this);
     instsY.Remove((Y*) this);
     instsZ.Remove((Z*) this);
   }
   συνεχίζεται →
```

```
  public:
  static X* DownCast (void* p) {
     return instsX.In(p) ? (X*) p :
            instsY.In(p) ? (X*) ((Y*) p) :
            instsZ.In(p) ? (X*) ((Z*) p) :
            (X*) 0;
  }
  X (...) {
     Register();
     Αυθεντικός κώδικας του constructor
  }
  ~X () {
     Αυθεντικός κώδικας του destructor
     Cancel();
  }
};
```

CSD

Δείκτες στιγμιότυπων και μετατροπές τύπων (9/9)

- *Μυστικά στις μετατροπές δεικτών (3/3)*

```
Δύο βασικά βοηθητικά macros (γίνονται templates ακομη καλύτερα):
#define UP_CAST(_baseclass, inst) \
  (_baseclass*) inst

#define DOWN_CAST(_derivedclass, inst) \
  _derivedclass::DownCast(inst)

Παραδείγματα χρήσης:
X x;
Y y;
Z z;
X* xPtr = DOWN_CAST(X, &z);         // Θα επιστρέψει null
xPtr = DOWN_CAST(X, &y);            // Θα επιστρέψει null
xPtr = &x;                          // Δεν χρειάζεται casting
Y* yPtr = UP_CAST(Y, xPtr);         // Ok, X κληρονόμος της Y
void* p = yPtr;                     // Κάθε pointer μπορεί να εκχωρηθεί σε void*
xPtr = DOWN_CAST(X, p);             // Επιστρέφει σωστά το &x
xPtr = DOWN_CAST(X, yPtr);          // Και αυτό επιστρέφει το ίδιο cons
```


CSD

Ένθετο – οδηγίες στη χρήση δεικτών

- *Όταν χρησιμοποιείτε δείκτες σε κλάσεις οι οποίες εμπλέκονται σε ιεραρχία κληρονομικότητας, φροντίστε να έχετε τρόπους να επικυρώνετε με δομές του προγράμματος σας*
  - Την συμφωνία ενός γενικού δείκτη **void*** ως νόμιμο στιγμιότυπο εκάστοτε κλάσης χρησιμοποιώντας λίστες καταγραφής των στιγμιότυπων κάθε κλάσης
  - Έτσι, θα μπορείτε να πιστοποιείτε κάθε down casting με σιγουριά, χωρίς την χρήση του *dynamic_cast*, ανάγοντας το σε compile-time cast
  - Να ελέγχετε με αυτό τον τρόπο όλα τα down casts και μη βασίζεστε στον έλεγχο με το μάτι ή σε εμπιστοσύνη στο τμήμα που παρέχει τον δείκτη
    - *never trust the origin of a pointer*