ΗΥ352 : ΕΡΓΑΣΤΗΡΙΟ ΤΕΧΝΟΛΟΓΙΑΣ ΛΟΓΙΣΜΙΚΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

3ο ΦΡΟΝΤΙΣΤΗΡΙΟ

ΔΙΔΑΣΚΟΝΤΑΣ
Αντώνιος Σαββίδης

Περιεχόμενα

- Friend keyword
- Κληρονομικότητα
- Πολυμορφισμός
  - Virtual συναρτήσεις
  - Abstract classes
  - Late binding
- Πολλαπλή Κληρονομικότητα

Friend Keyword

- Στη C++ μπορούμε να δώσουμε πρόσβαση στα private και protected δεδομένα μιας κλάσης στους «φίλους» της
- Φίλοι μιας κλάσης μπορούν να οριστούν συναρτήσεις και άλλες κλάσεις
- Μια κλάση ορίζει του φίλους της μέσα στο σώμα της δήλωσής της με χρήση του keyword ***friend***

```
class X {
   int x;
public:
   X(int x) : x(x) {}
   friend class Y;
   friend void print(X *x);
};

class Y {
   int y;
public:
   Y(X *x) : y(x->x) {}
   // Y is a friend of X so
   // it can access x->x
};

void print(X *x) {
   printf("%d", x->x);
   // print is a friend of X so
   // it can access x->x
}
```

Κληρονομικότητα (1/6)

- Η Κληρονομικότητα είναι η βασική αρχή του Οντοκεντρικού Προγραμματισμού
- Είναι ισχυρό εργαλείο για επαναχρησιμοποίηση κώδικα
- Οι κλάσεις που κληρονομούν από μια βασική κλάση επαναχρησιμοποιούν το interface της
- Ο στόχος είναι συνήθως να επεκτείνουμε ή να αλλάζουμε τη συμπεριφορά μιας κλάσης

```
class Animal {
public:
   Animal(char* name) :
        name(name) {}
   void Walk(void);
private:
   char* name;
};


class Cat : public Animal {
public:
  Cat(char* name):Animal(name){}
  void Climb(void);
};


class Dog : public Animal {
public:
  Dog(char* name):Animal(name){}
   void Bark(void);
};
```

Κληρονομικότητα (2/6)

- Τα μέλη μίας κλάσης μπορούν να έχουν ένα από τους παρακάτω χαρακτηρισμούς πρόσβασης:
  - **private** ή **public**
  - ή **protected,** ο οποίος χρησιμοποιείται αποκλειστικά στην κληρονομικότητα
    - Ένα protected μέλος είναι ουσιαστικά private εκτός της κλάσης ***X*** στην οποία ορίζεται, αλλά μπορεί να χρησιμοποιηθεί σε μία κλάση που είναι κληρονόμος της ***X***
- Η κληρονομικότητα πρέπει να χαρακτηρίζεται με έναν από τους τρεις διαφορετικούς χαρακτηρισμούς (δηλ. public, private, protected)
  - Αυτού του είδους ο χαρακτηρισμός ορίζει *τον τρόπο με τον οποίο οι αυθεντικοί χαρακτηρισμοί πρόσβασης των κληρονομημένων μελών τροποποιούνται μέσα στο χώρο της κληρονόμου κλάσης*
  - Αν δεν χαρακτηριστεί, θεωρείται private όταν κληρονομούμε από κλάσεις και public όταν κληρονομούμε από structs

Κληρονομικότητα (3/6)

- Στον παρακάτω πίνακα βλέπουμε τον χαρακτηρισμό πρόσβασης που έχουν τα μέλη μιας κληρονόμου κλάσης μέσα από την κληρονομούμενη κλάση αναλόγως με τον χαρακτηρισμό που θα έχει η κληρονομικότητα

Χαρακτηρισμός μέλους κλάσης / Χαρακτηρισμός Κληρονομικότητας	**public**	**protected**	**private**
public	public	protected	private
protected	protected	protected	private
private	private	private	private

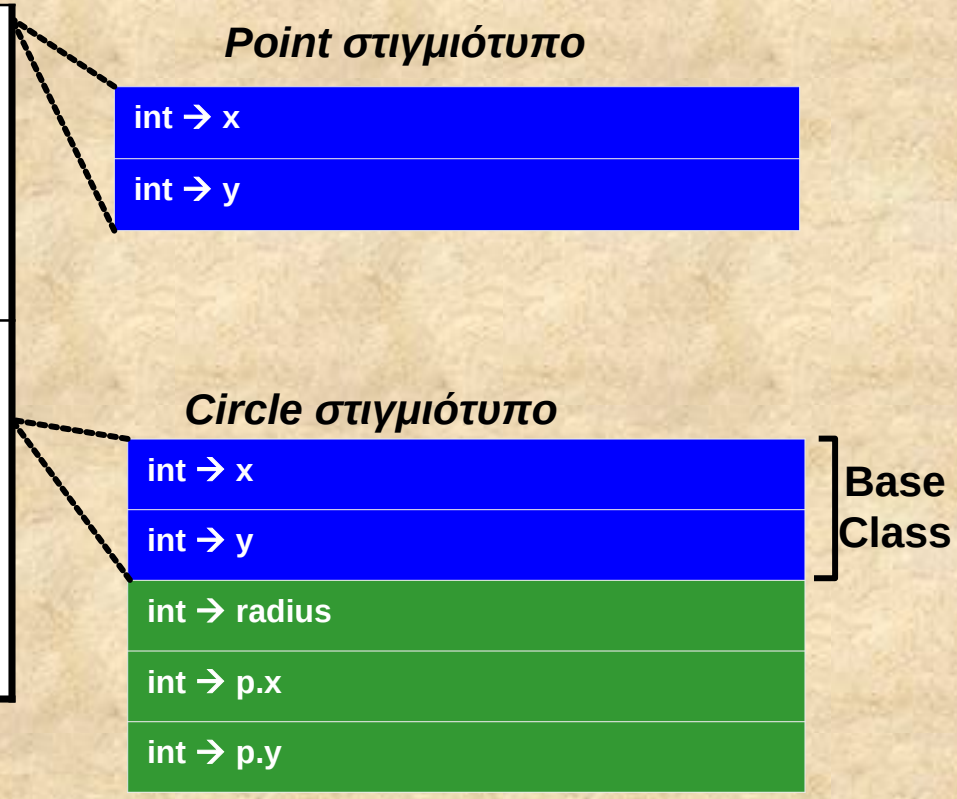
Κληρονομικότητα (4/6)

Class definition body

```
class Point {
protected:
    int x, y;
};
```

```
class Circle : public Point {
    int radius;
    Point p;
};
```

Run-time memory model



Κληρονομικότητα (5/6)

- Στην κληρονομικότητα τα derived classes κληρονομούν από το base class όλα τα member data και όλα τα member functions εκτός από:
  - Constructors (όλα τα overloaded versions)
  - Destructor
  - Operator = (όλα τα overloaded versions)
  - Friends
- Παρόλο που δεν κληρονομούνται οι constructors και ο destructor, ο default constructor και destructor πάντα καλούνται όταν δημιουργείται ή καταστρέφεται το derived στιγμιότυπο
  - Αν δεν υπάρχει default constructor ή θέλουμε να καλέσουμε κάποιον overloaded χρησιμοποιούμε το constructor initializer list

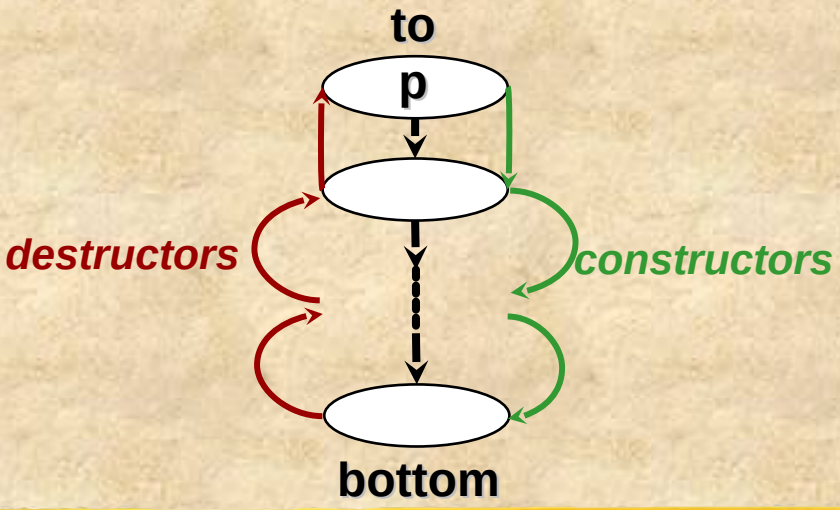
Κληρονομικότητα (6/6)

- Δε μπορούμε να αρχικοποιήσουμε private μεταβλητές μιας base class από τον constructor της derived class γιατί δεν έχουμε πρόσβαση σε αυτή.
- Χρησιμοποιούμε το συντακτικό αρχικοποίησης για constructors για να καλέσουμε τον constructor του base class

```
class Base {
    int i;
public:
    Base(int i) : i(i) {}
};

class Derived : public Base {
   int value;
public:
   Derived(int i, int val) :
     Base(i), value(val) {}
   Derived(int _i, int val) {
      i=_i; //error: i is private
      value = val;
   }
};
```

Σειρά κλήσης constructor / destructor (1/2)

- Σειρά κλήσης Constructor & Destructor
  - Οι constructors στην ιεραρχία κληρονομικότητας καλούνται από πάνω προς τα κάτω - *downwards*
  - Οι destructors στην ιεραρχία κληρονομικότητας καλούνται από κάτω προς τα πάνω - *upwards*



Σειρά κλήσης constructor / destructor (2/2)

```
class X {
public: X(void) { printf("X()\n"); }
       ~X() { printf("~X()\n"); }
};

class Y : public X {
public: Y(void) { printf("Y()\n"); }
       ~Y() { printf("~Y()\n"); }
};

class Z : public Y {
public: Z(void) { printf("Z()\n"); }
       ~Z() { printf("~Z()\n"); }
};

int main (int, char**) { Z z; return 0; }
```

Εκτυπώνεται κατά την εκτέλεση:

```
X()
Y()
Z()
~Z()
~Y()
~X()
```

Πολυμορφισμός (1/5)

- Το πιο δυνατό στοιχείο του Object Oriented Programming
- Θέλουμε να αλλάξουμε (εξειδικεύσουμε) τη συμπεριφορά της base class κάνοντας override τη λειτουργικότητα των μεθόδων της
- Οι συναρτήσεις όμως γίνονται linked στατικά κατά το compile και δεν υπάρχει τρόπος για τον pointer b να γνωρίζει at run-time τον τύπο του object στο οποίο δείχνει.

```
struct Base {
void foo(void) { printf("base");}
};
struct Derived : public Base {
void foo(void) {printf("derived");}
};

Derived derived;
Base base, *b = new Derived;

base.foo();     //prints base
derived.foo(); //prints derived
b->foo();       //prints base but
                //intended derived
```

Πολυμορφισμός (2/5)

- Τις συναρτήσεις αυτές τις δηλώνουμε με το keyword ***virtual***
- Λέμε έτσι στον compiler να μην κάνει στατικά link τον κώδικα της συνάρτησης αλλά να παράγει κώδικα που θα αποφασίσει για το ποια συνάρτηση πρέπει να κληθεί κατά το runtime
  - Λέγεται ***late (ή dynamic) binding***
  - Χρησιμοποιούνται πίνακες με τις κατάλληλες virtual συναρτήσεις για κάθε κλάση, γνωστοί ως *virtual function tables* (ή αλλιώς *vtables*)
  - Έχει κάποιο runtime overhead
- Οι static member functions δε μπορούν να δηλωθούν ως virtual
- Μια συνάρτηση που δηλώνεται virtual σε ένα class είναι πάντα virtual σε όλα τα derived classes

```
struct Base {
   virtual void foo(void) {
           printf("base");
   }
};
struct Derived : public Base{
   void foo(void) {
           printf("derived");
   }
};
Derived derived;
Base base, *b = new Derived;

base.foo();    //prints base
derived.foo(); //prints derived
b->foo();      //prints derived
```

Πολυμορφισμός (3/5)

- ***Pure virtual*** συναρτήσεις λέγονται οι virtual συναρτήσεις που δεν έχουν υλοποίηση
- Δηλώνονται κανονικά ως virtual συναρτήσεις προσθέτοντας ένα ***'=0'*** στο τέλος τους
- Δε μπορούμε να δημιουργήσουμε στιγμιότυπα από κλάσεις που έχουν έστω και μια pure virtual συνάρτηση
- Αυτές οι κλάσεις ονομάζονται ***abstract***
- Χρησιμοποιούνται για να ορίσουν γενική λειτουργικότητα που θα υλοποιηθεί αργότερα από τα derived classes
- Λειτουργικά, μοιάζουν με τα interfaces της Java
- Αν ένα derived class δεν υλοποιεί pure virtual συναρτήσεις ενός base, τότε είναι και αυτό *abstract*
- Μπορούμε να έχουμε pointers και references σε abstract classes, όχι όμως αντικείμενα

Πολυμορφισμός (4/5)

```
struct Shape {
   virtual double Area(void) const = 0;
};
struct Rectangle : public Shape {
   virtual double Area(void) const { return x*y }
   double x, y;
};
struct Box : public Rectangle {
   double Area(void) const { //once virtual always virtual
            assert(x == y);
            return Rectangle::Area();
    }
};
struct Circle : public Shape {
   double Area(void) const { return 3.14 * r * r; }
   double r;
};
Shape* s = new Shape; //error: shape is an abstract class
```

Πολυμορφισμός (5/5)

- Μια generator συνάρτηση γεμίζει έναν πίνακα με pointers τύπου κλάσης Shape
- Παρατηρείστε ότι η συνάρτηση *CreateShape* έχει επιστρεφόμενο τύπο Shape* αλλά τα return statements είναι concrete object pointers (Rectangle, Box, Circle)
  - Αυτό λέγεται *upcasting*. Box *IsA* Shape οπότε ένας Box* ικανοποιεί τις απαιτήσεις του Shape*
  - **ΠΡΟΣΟΧΗ: Το ανάποδο δεν είναι τόσο εύκολο...**
- Έχοντας τον πίνακα με τα shape instances, προσθέτουμε το εμβαδόν κάθε ενός όπως αυτό επιστρέφεται από τη συνάρτηση Area() της κλάσης Shape.
  - Έχουμε Shape* pointers αλλά ο compiler καλεί τις σωστές (αντίστοιχες) συναρτήσεις

```
#define SIZE 100
Shape* shapes[SIZE];

Shape* CreateShape (int shape) {
    switch (shape) {
        case 0: return new Rectangle;
        case 1: return new Box;
        case 2: return new Circle;
    }
    assert(false); return 0;
}

for (int i = 0; i < SIZE; ++i)
    shapes[i] = CreateShape(rand()%3);

int total = 0;
for (int i = 0; i < SIZE; ++i)
    total += shapes[i]->Area();
    //late bound call to Area
```

Virtual Destructors

- Ενδέχεται να έχουμε δεσμεύσει δυναμικά μνήμη για πολυμορφικά αντικείμενα
- Κατά τη διαγραφή τους με delete θέλουμε πάντα να κληθεί ο κατάλληλος (πιο derived) destructor
- Για αυτό το λόγο ορίζουμε το destructor της base class ***virtual***
- Πάντα όταν σχεδιάζουμε μια κλάση που πιθανό να κληροδοτήσει σε κάποια άλλη ορίζουμε τον destructor της ως virtual
- Στο παράδειγμα δίπλα, αν ο destructor της Base δεν ήταν virtual, κατά το *delete b* θα καλούνταν μόνο ο destructor της Base και θα είχαμε memory leak για το *Object o* της κλάσης Derived

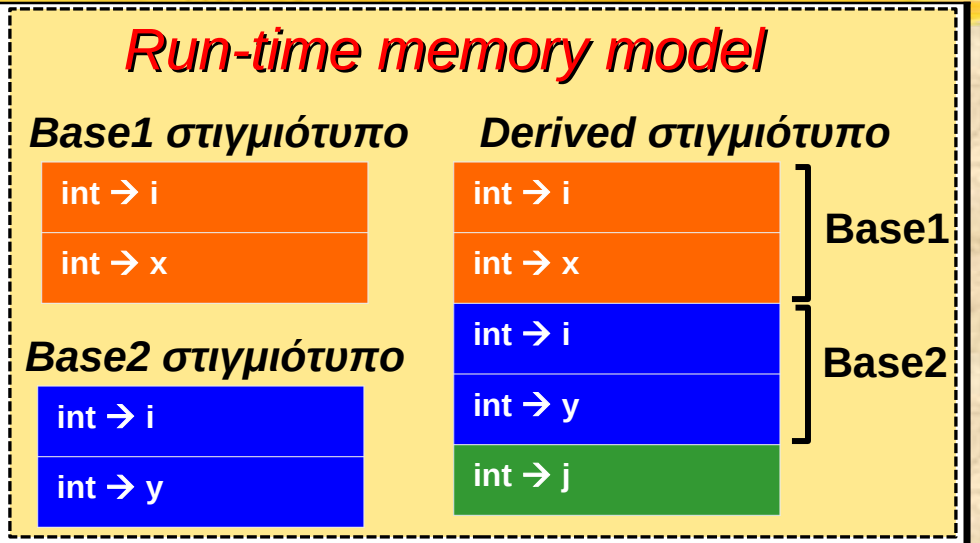
```
struct Base {
   Base(void) { }
   virtual ~Base(void) { foo(); }
   virtual void foo(void)
      { printf("base"); }
};

struct Derived : public Base {
   Derived(void) { o = new
   Object; }
   ~Derived(void){ foo(); delete
   o;}
   void foo(void)
      { printf("derived"); }
private:
   Object* o;
};

Base* b = new Derived;
delete b; //prints derived base
```

Πολλαπλή Κληρονομικότητα (1/2)

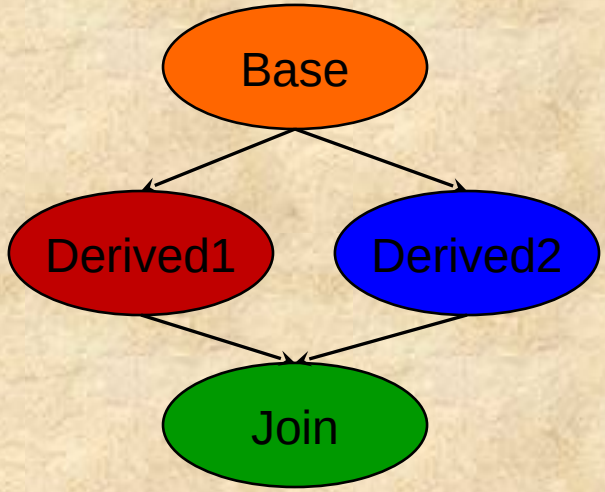
- Στη C++ επιτρέπεται μια κλάση να κληρονομεί από περισσότερες από μια κλάσεις.
- Η derived κλάση κληρονομεί τις μεταβλητές και τις μεθόδους από όλες τις base κλάσεις.
- Πρέπει όμως να διευκρινίζουμε στη derived κλάση ποια μέθοδο χρησιμοποιούμε σε περίπτωση που συμπίπτουν ονόματα από διαφορετικές base κλάσεις
  - Name qualification με τον τελεστή **::**

Πολλαπλή Κληρονομικότητα (2/2)

```
class Base1 {
public:
    int i, x;
    void foo(int x) {}
    Base1(int i = 0) : i(i) {}
};
class Base2 {
public:
    int i, y;
    void foo(int x) {}
    Base2(int i = 0) : i(i) {}
};

class Derived : public Base1, public Base2 {
    int j;
public:
          Derived(int i1, int i2) : Base1(i1), Base2(i2) {}
          void function(void) {
                    foo(5);         // compile error: ambiguous call to foo
                    i = 3;          // compile error: ambiguous access to i
                    Base1::foo(5); // ok, καλείται η foo της Base1
                    Base2::i = 5;  // ok, η ανάθεση γίνεται στο i της Base2
          }
};
```

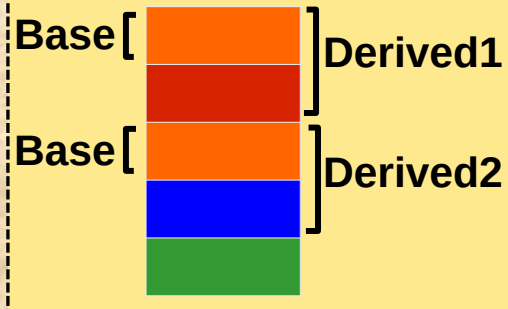


Κληρονομικότητα “Diamond”

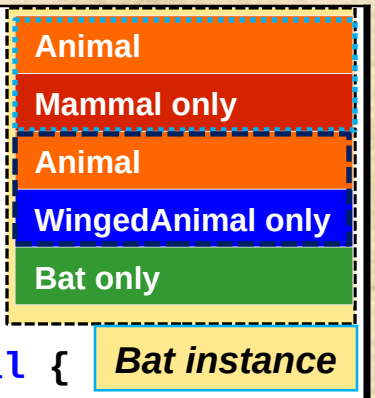


Run-time memory model

***Join** στιγμιότυπο*



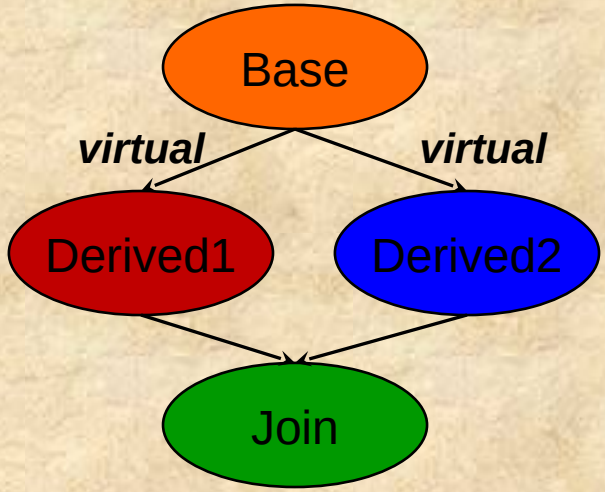

```
class Animal {
    public: virtual void eat();
};
class Mammal : public Animal {
    public: virtual void walk();
};
class WingedAnimal : public Animal {
    public: virtual void fly();
};

// A bat is a winged mammal
class Bat : public Mammal,public WingedAnimal{};

Bat *bat = new Bat;
bat->eat();                 //error, ambiguous call
bat->Mammal::eat();         //ok, calls Mammal::eat
Animal *a=(Animal *)bat; //error, ambiguous cast
```



Virtual Κληρονομικότητα (1/2)

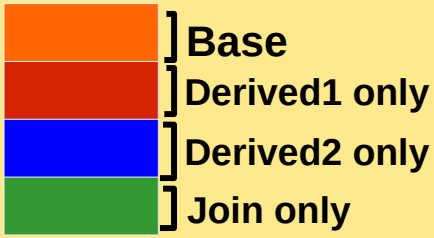
- Το diamond inheritance προκύπτει επειδή κληρονομούμε δύο φορές από την ίδια κλάση
- Για να εξασφαλίσουμε ότι όσες φορές κι αν κληρονομήσουμε από κάποιο base class θα έχουμε πάντα ***μόνο ένα κοινό*** base instance, χρησιμοποιούμε ***virtual κληρονομικότητα***
- Ενδέχεται να κληρονομούμε από κάποιο base class και με virtual τρόπο αλλά και κανονικά
  - Σε αυτή την περίπτωση, έχουμε *ένα κοινό base instance* από τη virtual κληρονομικότητα και *ένα κανονικό* για κάθε επιπλέον κανονική κληρονομικότητα

Virtual Κληρονομικότητα (2/2)



Run-time memory model

Join στιγμιότυπο




```
class Animal {
    public: virtual void eat();
};
class Mammal : public virtual Animal {
    public: virtual void walk();
};
class WingedAnimal : public virtual Animal {
    public: virtual void fly();
};

// A bat is still a winged mammal
class Bat : public Mammal,public WingedAnimal{};

Bat *bat = new Bat;
bat->eat();    //ok, single animal sub instance
bat->Mammal::eat();      //ok, but unnecessary
Animal *a=(Animal *)bat; //ok, inherited once
```