HY-150a Programming Recitation 2
Επαναληπτικό μάθημα 2/2

Class- Object- Constructor - ***Destructor***

Ενθυλάκωση-Αφαίρεση

Κληρονομικότητα

Πολυμορφισμός

HY-150a Programming Recitation 2

Αντικειμενοστρεφή προγραμματισμό (object-oriented programming), ή OOP, ονομάζουμε ένα προγραμματιστικό υπόδειγμα το οποίο εμφανίστηκε στα τέλη της δεκαετίας του 1960 και καθιερώθηκε κατά τη δεκαετία του 1990, αντικαθιστώντας σε μεγάλο βαθμό το παραδοσιακό υπόδειγμα του δομημένου προγραμματισμού (πχ. C).

- Πρόκειται για μία μεθοδολογία ανάπτυξης προγραμμάτων,υποστηριζόμενη από κατάλληλες γλώσσες προγραμματισμού.

HY-150a Programming Recitation 2

Τα σχετιζόμενα δεδομένα και των διαδικασίες τους που επενεργούν σε αυτά γίνεται από κοινού, μέσω μίας δομής δεδομένων το *αντικείμενο*.

Κεντρική ιδέα στον αντικειμενοστραφή προγραμματισμό (OOP) είναι η **κλάση** (class).

- Μία αυτοτελής και αφαιρετική αναπαράσταση κάποιας κατηγορίας αντικειμένων.
- Είτε φυσικών αντικειμένων του πραγματικού κόσμου είτε νοητών εννοιολογικών αντικειμένων, σε ένα περιβάλλον προγραμματισμού.

HY-150a Programming Recitation 2

- Ένας τύπος δεδομένων, ή αλλιώς το προσχέδιο μίας δομής δεδομένων με δικά της περιεχόμενα, τόσο μεταβλητές όσο και διαδικασίες.
- Τα περιεχόμενα αυτά δηλώνονται είτε ως *δημόσια* (public) είτε ως *ιδιωτικά* (private), με τα ιδιωτικά να μην είναι προσπελάσιμα από κώδικα εκτός της κλάσης.
- Οι διαδικασίες των κλάσεων συνήθως καλούνται *μέθοδοι* (methods) και οι μεταβλητές τους *γνωρίσματα* (attributes) ή *πεδία* (fields).

HY-150a Programming Recitation 2

- Ιδανικά να είναι εννοιολογικά αυτοτελής
- Να περιέχει δηλαδή μόνο πεδία τα οποία περιγράφουν μία κατηγορία αντικειμένων
- Δημόσιες μεθόδους οι οποίες επενεργούν σε αυτά όταν καλούνται από το εξωτερικό πρόγραμμα, χωρίς να εξαρτώνται από άλλα δεδομένα ή κώδικα εκτός της κλάσης.
- Επαναχρησιμοποιήσιμη, ένα μαύρο κουτί δυνάμενο να λειτουργήσει χωρίς τροποποιήσεις ως τμήμα διαφορετικών προγραμμάτων

HY-150a Programming Recitation 2

Αντικείμενο (object) είναι το στιγμιότυπο μίας κλάσης, ή η δομή δεδομένων (με αποκλειστικά δεσμευμένο χώρο στη μνήμη) βασισμένη στο «καλούπι» που προσφέρει η κλάση. Επομένως μόνο τα αντικείμενα καταλαμβάνουν χώρο στη μνήμη του υπολογιστή.

Για παράδειγμα μια κλάση BankAccount, η οποία αναπαριστά έναν τραπεζικό λογαριασμό, και να δηλώσουμε ένα αντικείμενο της με όνομα MyAccount.

Το MyAccount θα έχει δεσμεύσει χώρο στη μνήμη με βάση τις μεταβλητές και τις μεθόδους που περιγράφονται στην κλάση.

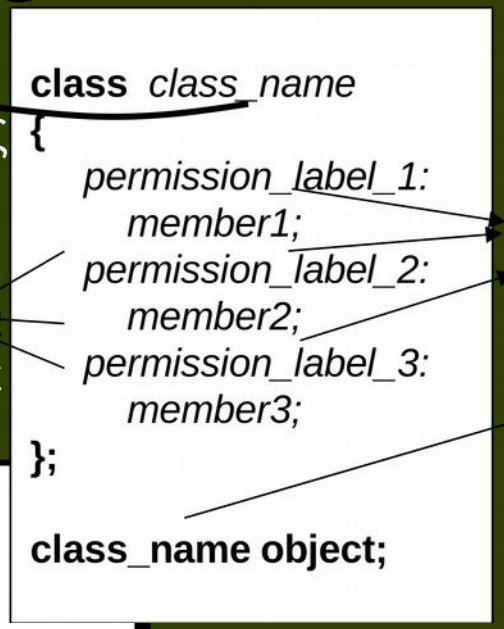
HY-150a Programming Recitation 2

- Το MyAccount θα μπορούσε να περιέχει ένα γνώρισμα (attributes), Balance (=υπόλοιπο))

και μία μέθοδος GetBalance (=επίστρεψε το υπόλοιπο).

- Ακολούθως θα μπορούσαμε να δημιουργήσουμε ακόμα ένα ή περισσότερα αντικείμενα της ίδιας κλάσης τα οποία θα είναι διαφορετικές δομές δεδομένων (διαφορετικοί τραπεζικοί λογαριασμοί στο παράδειγμα).
- Τα αντικείμενα μίας κλάσης μπορούν να προσπελάσουν τα ιδιωτικά περιεχόμενα άλλων αντικειμένων της ίδιας κλάσης.

HY-150a Programming Recitation 2




```
#include <stdio.h>
class Point {
    int x, y;        // by default private
public:
    Point(int _x, int _y) : x(_x), y(_y) {}
    //same as Point(int _x, int _y) { x = _x; y = _y; }
    int GetX(void) { return x; }
    int GetY(void) { return y; }
    void Move(int dx, int dy);
};
void Point::Move(int dx, int dy) { x += dx, y += dy; }

int main(void) {
    Point p(10, 20);
    printf("p =(%d, %d)\n", p.GetX(), p.GetY());
    p.x = 10;    ///< compile error private member
}
```


HY-150a Programming Recitation 2

Ενθυλάκωση δεδομένων (data encapsulation)

- Ή ιδιότητα που προσφέρουν οι κλάσεις να «κρύβουν» τα ιδιωτικά δεδομένα τους από το υπόλοιπο πρόγραμμα και να εξασφαλίζουν πως μόνο μέσω των δημόσιων μεθόδων τους θα μπορούν αυτά να προσπελαστούν.
- Αυτή η τακτική παρουσιάζει μόνο οφέλη καθώς εξαναγκάζει κάθε εξωτερικό πρόγραμμα να φιλτράρει το χειρισμό που επιθυμεί να κάνει στα πεδία μίας κλάσης μέσω των ελέγχων που μπορούν να περιέχονται στις δημόσιες μεθόδους της κλάσης.

HY-150a Programming Recitation 2

- Η λειτουργικότητα αυτή λέγεται συνήθως Application Programming Interface – API.
- Για να έχουμε πρόσβαση στα private members παρέχουμε επιπλέον member functions που τα λέμε accessors.
- Το keyword ***this*** είναι ένας pointer στο στιγμιότυπο της κλάσης που μας βοηθάει να προσπελάσουμε members που είναι υποσκιασμένες shadowed από τοπικές μεταβλητές.

```
class Vector {
public:
   Vector Add(Vector v);
   Vector DotProduct(Vector v);
   Vector CrossProduct(Vector v);
   bool   IsEqual(Vector v);

   float GetX(void) {return x;}
   void SetX(float x)
      { this->x = x; }
   void SetY(float y)
      { Vector::y = y; }
   // z...
private:
   float x, y, z;
};
```

HY-150a Programming Recitation 2

```
1  class Vehicle {
2  protected:
3      string license;
4      int year;
6  public:
7      Vehicle(const string &myLicense, const int myYear)
8          : license(myLicense), year(myYear) {}
9      const string getDesc() const
10         {return license + " from " + stringify(year);}
11     const string &getLicense() const {return license;}
12     const int getYear() const {return year;}
13 };
```

Constructor

<<Get-ers>> for be able to
call protected, private members
But without changing var. values

HY-150a Programming Recitation 2

Αφαίρεση δεδομένων καλείται η ιδιότητα των κλάσεων να αναπαριστούν αφαιρετικά πολύπλοκες οντότητες στο προγραμματιστικό περιβάλλον.

- Μία κλάση αποτελεί ένα αφαιρετικό μοντέλο κάποιας κατηγορίας αντικειμένων.
- Επίσης οι κλάσεις προσφέρουν και αφαίρεση ως προς τον υπολογιστή, εφόσον η καθεμία μπορεί να θεωρηθεί ένας μικρός και αυτάρκης υπολογιστής (με δική του κατάσταση, μεθόδους και μεταβλητές).

HY-150a Programming Recitation 2

Υπερφόρτωση μεθόδου (method overloading) είναι η κατάσταση κατά την οποία υπάρχουν, στην ίδια ή σε διαφορετικές κλάσεις, μέθοδοι με το ίδιο όνομα και πιθανώς διαφορετικά ορίσματα.

- Αν πρόκειται για μεθόδους της ίδιας κλάσης διαφοροποιούνται μόνο από τις διαφορές τους στα ορίσματα και στον τύπο επιστροφής.
- Μπορούμε να κάνουμε overload οποιαδήποτε συνάρτηση. Πρέπει όμως οι overloaded συναρτήσεις να έχουμε διαφορετική υπογραφή και μάλιστα να μη διαφέρουν μόνο στον επιστρεφόμενο τύπο.
- Στη C++ μπορούν να υπερφορτωθούν όλοι οι τελεστές, εκτός από τους παρακάτω τέσσερις:

. .* :: ? :

HY-150a Programming Recitation 2

```
class printData {
public:
void print(int i) { cout << "Printing int: " << i << endl; }
void print(double f) {
cout << "Printing float: " << f << endl;
}
void print(char* c) {
cout << "Printing character: " << c << endl;
}
};
int main(void) {
printData pd;
pd.print(5);
pd.print(500.263);
pd.print("Hello C++");
return 0;
}
```

HY-150a Programming Recitation 2

```
// Different signatures for all overloads, so everything ok
int    square(int x)    { return x*x; }
float  square(float x)  { return x*x; }
double square(double x) { return x*x; }

// Overloaded function differs only in return value
float  square(float x)  { return x*x; }
double square(float x)  { return x*x; }  //compile error here

// The following will not compile!
void foo (int x);
void foo (int x, int y=0);
foo(3, 5); // ok, calls second version
foo(3);    // ambiguous call: foo(3) or foo(3,0)?

// Neither  will this!
void foo (int x);
void foo (int x=0, int y=0);
foo();     // ok, calls second version
foo(3);    // ambiguous call: foo(3) or foo(3,0)?
```

HY-150a Programming Recitation 2

```
class Point {
   int x, y;
public:
   Point(int x = 0, int y = 0) : x(x), y(y) {}
   const Point operator=(const Point& p) {  // a=b=c επιτρέπεται
      x = p.x, y=p.y;
      return *this;
   }
   const Point operator+(const Point& p) { return Point(x + p.x, y + p.y); }
};

Point p1(10,20), p2, p3;
p3 = p2 = p1;       // Αυτόματη κλήση του υπερφορτωμένου τελεστή =
p2.operator=(p1);  // Ως συνάρτηση, ισχύει και αυτό το συντακτικό !

class Plus {      // Classes που υπερφορτώνουν το () λέγονται functors
   public: int operator()(int a, int b) { return a + b; } // Υπερφόρτωση τελεστή ()
}

Plus plus;                      // Δήλωση στιγμιότυπου
int a = plus(10,20);            // Αυτόματη κλήση υπερφορτωμένου τελεστή ()
a = plus.operator()(10,20);  // ...ή με το εναλλακτικό μη συνηθισμένο συντακτικό
```

HY-150a Programming Recitation 2

Κληρονομικότητα ονομάζεται η ιδιότητα των κλάσεων να επεκτείνονται σε νέες κλάσεις, ρητά δηλωμένες ως κληρονόμους (*υποκλάσεις* ή 'θυγατρικές κλάσεις'), οι οποίες μπορούν να επαναχρησιμοποιήσουν τις μεταβιβάσιμες μεθόδους και ιδιότητες της γονικής τους κλάσης αλλά και να προσθέσουν δικές τους.

Στιγμιότυπα των θυγατρικών κλάσεων μπορούν να χρησιμοποιηθούν αντί των γονικών αφού η θυγατρική είναι κατά κάποιον τρόπο μία πιο εξειδικευμένη εκδοχή της γονικής αλλά το αντίστροφο δεν ισχύει.

HY-150a Programming Recitation 2

Παράδειγμα κληρονομικότητας είναι μία γονική κλάση Vehicle (=Όχημα) και η υποκλάση της Car.

Πολλαπλή κληρονομικότητα είναι η δυνατότητα που προσφέρουν ορισμένες γλώσσες προγραμματισμού μία κλάση να κληρονομεί ταυτόχρονα από περισσότερες από μία γονικές.

HY-150a Programming Recitation 2

Inheritance example : (Vehicle-Car):

```
 6  public:
 7      Vehicle(const string &myLicense, const int myYear)
 8          : license(myLicense), year(myYear) {}
 9      const string getDesc() const
10          {return license + " from " + stringify(year);}
11      const string &getLicense() const {return license;}
12      const int getYear() const {return year;}
13  };
```

```
1  class Car : public Vehicle { // Makes Car inherit from Vehicle
2      string style;
3
4  public:
5      Car(const string &myLicense, const int myYear, const string
           &myStyle)
6          : Vehicle(myLicense, myYear), style(myStyle) {}
7      const string &getStyle() {return style;}
8  };
```

HY-150a Programming Recitation 2

Inheritance example : (Animal-Cat-Dog):

```
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);
};
```

HY-150a Programming Recitation 2

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 Χ στην οποία ορίζεται, αλλά μπορεί να χρησιμοποιηθεί σε μία κλάση που είναι κληρονόμος της Χ.

HY-150a Programming Recitation 2

- 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.

HY-150a Programming Recitation 2

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:

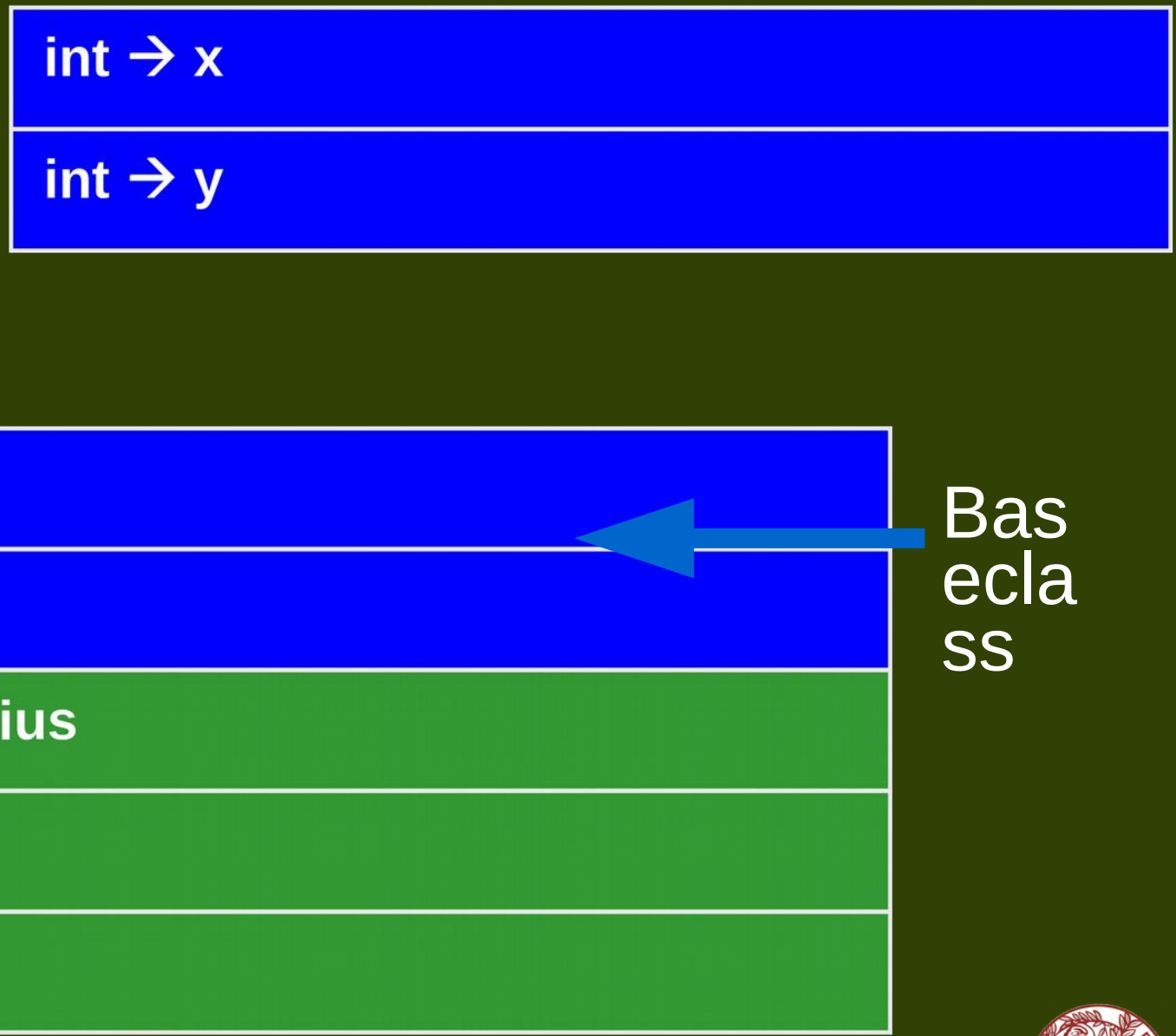
Χαρακτηρισμός μέλους κλάσης / Χαρακτηρισμός Κληρονομικότητας	**public**	**protected**	**private**
public	public	protected	private
protected	protected	protected	private
private	private	private	private

HY-150a Programming Recitation 2

Class definition body

```
class P {
protected:
  int x, y;
};
class C : public P {
    int R;
    P p;
};
```

Run-time memory model



HY-150a Programming Recitation 2

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.

HY-150a Programming Recitation 2

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;
    }
};
```

HY-150a Programming Recitation 2

Constructor δημιουργεί το class object

- Έχει το ίδιο όνομα με την κλάση
- Μια κλάση μπορεί να έχει πολλούς overloaded constructors
- Χρησιμοποιείται ειδικό συντακτικό για την αρχικοποίηση των members

Destructor καταστρέφει το object (όταν βγει εκτός scope είτε από delete)

- Έχει το ίδιο όνομα με την κλάση με επιπλέον το ~
- Μια κλάση μπορεί να έχει μόνο ένα destructor και μάλιστα χωρίς ορίσματα
- Αν δεν οριστεί καθόλου constructor ή destructor, ο compiler δημιουργεί τους default που δεν παίρνουν ορίσματα.

HY-150a Programming Recitation 2

```
int main(void) {

Vector* vec1 = new Vector; // default constructor is called
Vector vec2(2,4,1);  // parametrized constructor is called
vec1->Add(vec2);        // vec1 + vec2

if (vec1->IsEqual(vec2))      // vec1 == vec2

printf("vec = (%f,%f,%f)\n", vec1->GetX(),
        vec1->GetY(), vec1->GetZ());
delete vec1;              // destructor of vec1 object called here
    return 0;
}                        // destructor of vec2 object called here
```

HY-150a Programming Recitation 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; }
```

HY-150a Programming Recitation 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()

HY-150a Programming Recitation 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.

HY-150a Programming Recitation 2

Πολυμορφισμός σημαίνει "πολλές μορφές". Αναφέρεται στην ικανότητα ενός αντικειμένου να έχει πολλούς τύπους. Αν έχουμε μια function που αναμένει ένα object Vehicle , μπορούμε να το περάσου με ασφάλεια ένα Car object, επειδή κάθε Car είναι επίσης ένα Vehicle.

- 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.

HY-150a Programming Recitation 2

Polymorphism example:

```
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();
```

HY-150a Programming Recitation 2

Polymorphism example:

```
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();
```

Problem!!!
prints base but
intended derived

HY-150a Programming Recitation 2

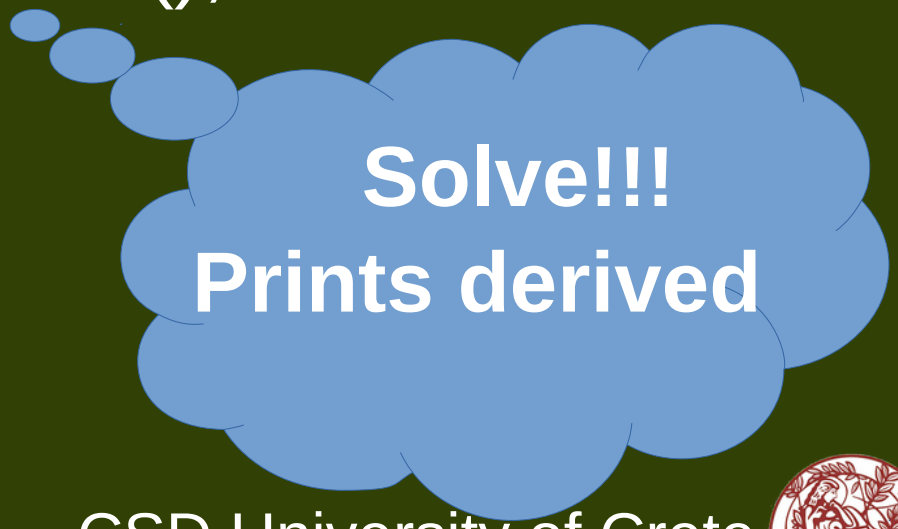
Τις συναρτήσεις αυτές τις δηλώνουμε με το keyword virtual:

- Ο compiler δεν κάνει στατικά link τον κώδικα της συνάρτησης αλλά να παράγει κώδικα που θα αποφασίσει για το ποια συνάρτηση πρέπει να κληθεί κατά το runtime ή late (ή dynamic) binding
- 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.

HY-150a Programming Recitation 2

```
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();
```



HY-150a Programming Recitation 2

Pure virtual συναρτήσεις λέγονται οι virtual συναρτήσεις που δεν έχουν υλοποίηση.

- Δηλώνονται ως virtual συναρτήσεις με ένα ***'=0'*** στο τέλος. 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, αν το derived class δεν υλοποιεί τις pure virtual συναρτήσεις ενός base, τότε είναι και αυτό *abstract.*

Μπορούμε να έχουμε pointers και references σε abstract classes, όχι όμως αντικείμενα

Μία *abstract* κλάση χωρις γνωρίσματα και οι μέθοδοί της είναι *abstract* και δημόσιες καλείται **διασύνδεση** (interface).

HY-150a Programming Recitation 2

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 επόμενο παράδειγμα, αν ο destructor της Shape δεν ήταν virtual, κατά το *delete shape* θα καλούνταν μόνο ο destructor της Base (Shape) και θα είχαμε memory leak αν είχαμε κάνει new ένα *Object* της Derived κλάσης Circle.

HY-150a Programming Recitation 2

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

Στο παράδειγμα, 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 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
```

HY-150a Programming Recitation 2

```
#include <stdio.h>
class Shape {
  public:
  virtual ~Shape();
  virtual void draw() = 0;
};
class Circle : public Shape {
public:
   ~Circle();

  virtual void draw();
};

Shape::~Shape() { printf("shape
destructor\n"); }
```

Virtual Destructor

Pure virtual

default Destructor

```
// void Shape::draw() {
//   printf("Shape::draw\n");}
Circle::~Circle()
{ printf("circle
destructor\n"); }
void Circle::draw() {
printf("Circle::draw\n"); }

int main() {
  Shape *shape = new
Circle;
  shape->draw();
  delete shape;
  return 0;}
```

HY-150a Programming Recitation 2

Στο παράδειγμα, αν ο destructor της Shape δεν ήταν virtual, κατά το *delete shape* θα καλούνταν μόνο ο destructor της Base (Shape) και θα είχαμε memory leak αν είχαμε κάνει new ένα *Object* της Circle.

- Only inherit from something if you have a virtual destructor!
- If you don’t, you’re almost certainly going to have memory leaks.
- Whether or not the destructor is virtual is a clear indication of whether or not a type is intended to be inherited from.
- Don’t override functions that aren’t virtual! It’s not illegal, it’s just bad practice.

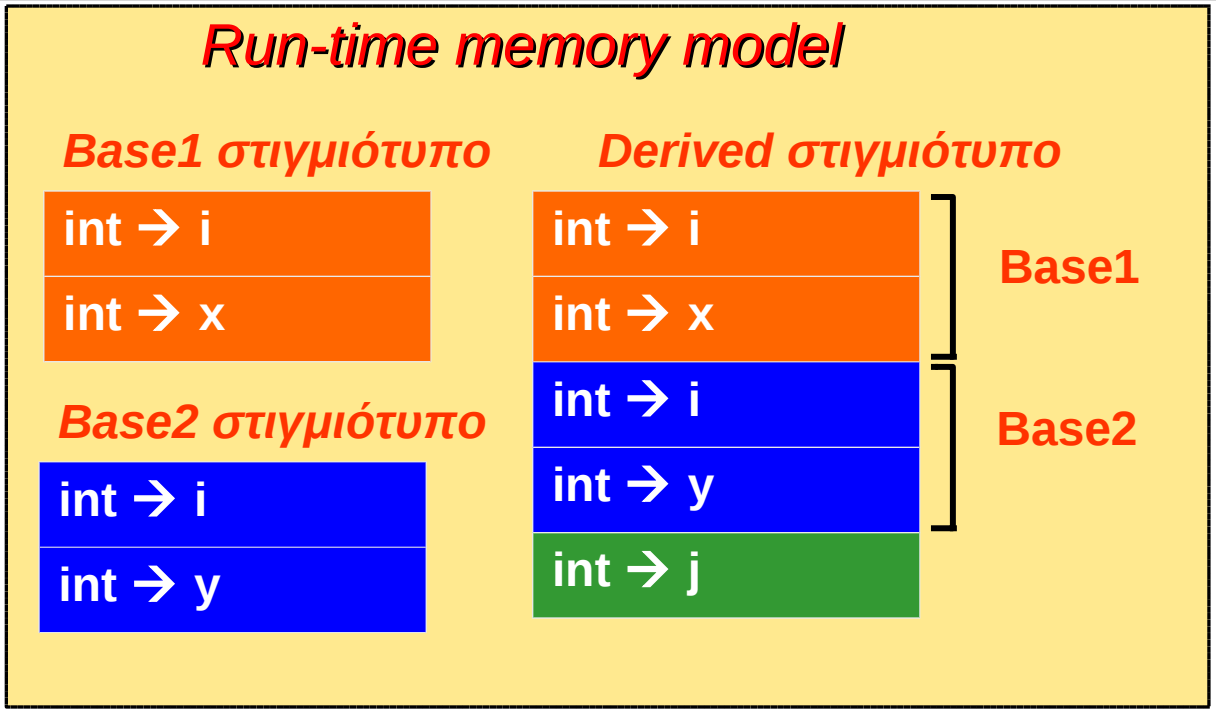
HY-150a Programming Recitation 2

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 **::**

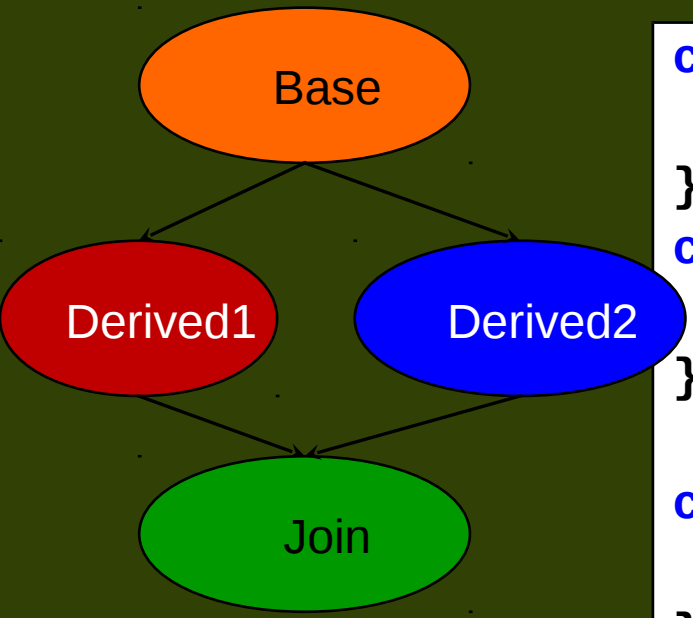
HY-150a Programming Recitation 2

```
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) {}
};
```



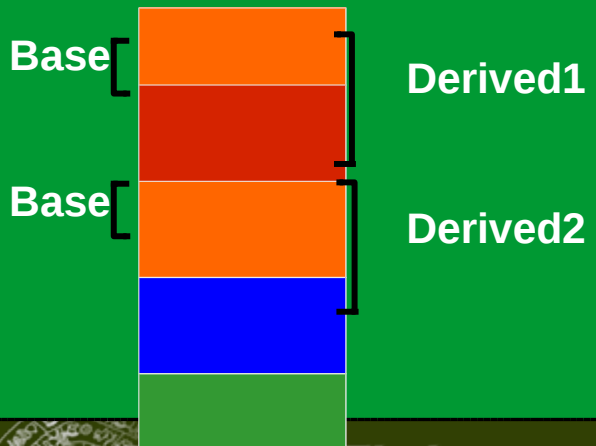

```
class Derived : public Base1, public Base2 {
    int j;
public:
       Derived(int i1, int i2) : Base1(i1), Base2(i2) {}
       void function(void) {
              foo(5);          // compile error: ambiguous call to foo
              i = 3;           // compile error: ambiguous access to i
    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

HY-150a Programming Recitation 2

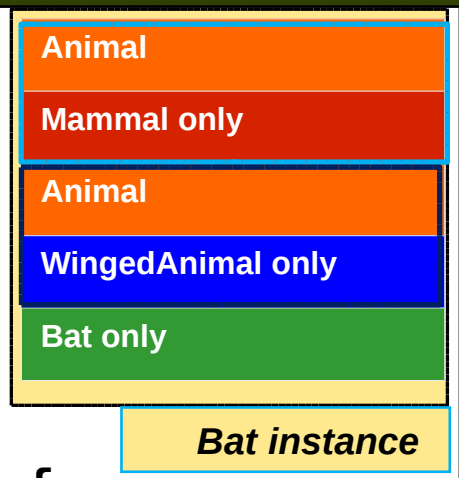


Run-time memory model

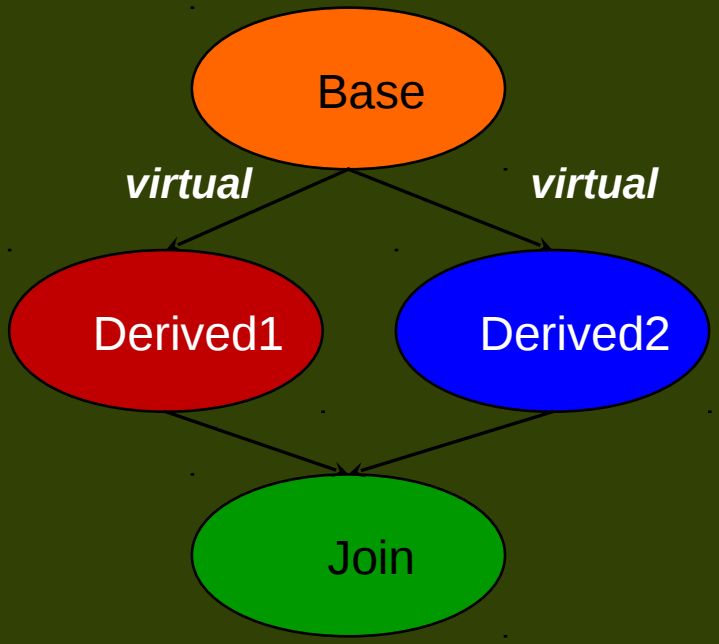
***Join* στιγμιότυπο**




```
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
```

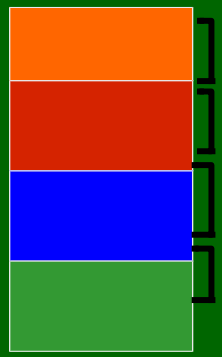


HY-150a Programming Recitation 2



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
```

HY-150a Programming Recitation 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 την 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.

HY-150a Programming Recitation 2

Upcasting and downcasting are an important part of C++. Upcasting and downcasting gives a possibility to build complicated programs with a simple syntax. It can be achieved by using Polymorphism.
C++ allows that a derived class pointer (or reference) to be treated as base class pointer. This is upcasting.
Downcasting is an opposite process, which consists in converting base class pointer (or reference) to derived class pointer.
Upcasting and downcasting should not be understood as a simple casting of different data types. It can lead to a great confusion.

HY-150a Programming Recitation 2

Both upcasting and downcasting do not change object by itself. When you use upcasting or downcasting you just "label" an object in different ways. Downcasting is not safe as upcasting (in single inheritance). A derived class object can be always treated as base class (in single inheritance) object, the opposite is not right.

Use of derived type instead of a base type:

```
Derived *d = new Derived();
Base *b = new Derived();
```

dynamic_cast:

Performs a check, at runtime, that the cast is valid. If it isn’t, returns nullptr.

The check doesn’t work in all situations (sidecasts)

Only works on polymorphic types (types with virtual functions).

HY-150a Programming Recitation 2

static_cast:
- No runtime check
- Bad cast = undefined behavior

Going backwards:

```
Base *b = new Derived();
Derived *d1 = dynamic_cast<Derived *>(b);
Derived *d2 = static_cast<Derived *>(b);
```

Side note: All of these are legal

```
int a = (int)b;
int a = int(b);
int a = static_cast<int>(b);
```

HY-150a Programming Recitation 2

Accessing Overridden Functions

```
class Base {
virtual void foo();
};
class Derived : public Base {
void foo() override;
};
Derived *d = new Derived();
d->Base::foo();
```

HY-150a Programming Recitation 2

Question: Create a class Person and two derived classes Employee, and Student, inherited from class Person. Now create a class Manager which is derived from two base classes Employee and Student. Show the use of the virtual base class.

HY-150a Programming Recitation 2

```
#include<iostream>
using namespace std;
class Person{
private:
char name[20];
char emp_id[10];

public:
void get_data()
{cout<<"Enter Employee ID: ";
cin>>emp_id;
Popular Posts}
void show_data()
{cout<<"\nEmplyee ID: "<<emp_id;}
};
```

```
class Employee :virtual public Person
{
private:
char emp_id[10];

public:
void get_data(){
     cout<<"Enter Employee ID: ";
     cin>>emp_id; }
void show_data(){
cout<<"\nEmplyee ID: "<<emp_id;}
};
```

HY-150a Programming Recitation 2

The keyword 'virtual' makes only one copy of ger_per_data() and show_per_data().

Which avoids ambiguity.
If we do not supply keyword 'virtual' the compiler cannot decides which class' get_per_data() and show_per_data() to be called.
Because they are the member functions of classes Employee , Student, and Manager as well.

HY-150a Programming Recitation 2

```
class Student : virtual public
Person{
private:
char roll_no[10];

public:
void get_data()
{cout<<"Enter Roll No: ";
cin>>roll_no;}
void show_data()
{cout<<"\nRoll No: "<<roll_no;}
};
```

```
class Manager: public Employee,
public Student
{
private:
unsigned int salary;
public:
void get_data() {
Employee::get_data(); /**calls the
get_data() of Employee**/
Student::get_data();
cout<<"Enter Salary: ";
cin>>salary;}

void show_data(){
Employee::show_data();
Student::show_data();
cout<<"\nSalary: "<<salary;}
};
```

HY-150a Programming Recitation 2

```
int main() {
Manager m;
cout<<"Enter data for manager: ";
m.get_per_data(); /**calls single copy of
get_per_data()**/
m.get_data();

cout<<"\nThe data on manager is:  ";

m.show_per_data();
m.show_data();
return 0;
}
```

HY-150a Programming Recitation 2

Question: Create a polymorphic class Vehicle and create other derived classes Bus, Car from Vehicle. With this program illustrate RTTI (Run-Time Type Information) by the use of dynamic_cast and typeid operators.

```
#include<iostream>
#include<typeinfo>
using namespace std;
class Vehicle{
public:
        virtual void show(){}
};
class Bus: public Vehicle{};
class Car: public Vehicle{};
```

```
int main(){
Vehicle *pveh, veh;
Bus *pbs;// declare pointer to bus to use
it in dynamic_cast.
Bus bs; //static class instance (object) to
//use it as reference to initialize pveh
and then dynamic_cast to pbs, is on the
stack and not in heap when using new
Car *pcr; Car  cr;//class instance (object)
pveh = &bs;// reference
pbs=dynamic_cast<Bus*>(pveh);
if(pbs) //check if cast is ok
        cout<<"The Cast to derived pointer
pbs is Sucessful"<<endl;
else
    cout<<"The Cast to derived pointer pbs
is Failed"<<endl;
```

HY-150a Programming Recitation 2

Here the cast from base class Vehicle pointer pveh to derived class Bus pointer pbs works well because pveh is pointing to derived class Bus' Object.

HY-150a Programming Recitation 2
General examples

1A.In this example you will implement a class representing an array of Points(PointArray).
It will allow dynamically resizing the array, and it will track its own length so that if you
Were to pass it to a function, you would not need to pass its length separately.
Create the class with two private members, a pointer to the start of an array of Points and an int that stores the size (length)of the array. This is geometry.h example header file

```
1  class PointArray {
2      int size;
3      Point *points;
4
5      void resize(int size);
6
7  public:
8      PointArray();
9      PointArray(const Point pts[], const int size);
10     PointArray(const PointArray &pv);
11     ~PointArray();
12
13     void clear();
14     int getSize() const { return size;}
15     void push_back(const Point &p);
16     void insert(const int pos, const Point &p);
17     void remove(const int pos);
18     Point *get(const int pos);
19     const Point *get(const int pos) const;
20 };
```

HY-150a Programming Recitation 2
General examples

1B.Now Implement the default constructor(a constructor with no arguments) with the following signature. It should create an array with size0. Implement a constructor that takes a Pointarray called points and an int called size as its arguments. It should initialize a PointArray with the specified size, copying the values from points. You will need to dynamically allocate the PointArray's internal array to the specified size.

PointArray::PointArray(const Point points[], const int size)

Finally, implement a constructor that creates a copy of a given PointArray (a copy constructor).
PointArray::PointArray(const PointArray& pv)

(Hint: Make sure that the two PointArrays do not end up using the same memory for their internal arrays. Also make sure that the contents of the original array are copied, as well.)

Define a destructor that deletes the internal array of the PointArray.
PointArray::~PointArray()

HY-150a Programming Recitation 2
General examples

```
#include "geometry.h"

PointArray::PointArray() {
    size = 0;
    points = new Point[0]; // Allows deleting later
}

PointArray::PointArray(const Point ptsToCopy[], const int toCopySize
    ) {
    size = toCopySize;
    points = new Point[toCopySize];
    for(int i = 0; i < toCopySize; ++i) {
        points[i] = ptsToCopy[i];
    }
}
```

HY-150a Programming Recitation 2
General examples

```
PointArray::PointArray(const PointArray &other) {
    // Any code in the PointArray class has access to
    // private variables like size and points
    size = other.size;
    points = new Point[size];
    for (int i = 0; i < size; i++) {
        points[i] = other.points[i];
    }
}

PointArray::~PointArray() {
    delete[] points;
}
```

HY-150a Programming Recitation 2
General examples

1.C Since we will allow modifications to our array, you'll find that the internal array grows and shrinks quite often.
A simple (though very inefficient) way to deal with this without repetitively writing similar code is to write a member function PointArray::resize(int n) that allocates a new array of size n, copies the first min (previous array size, n) existing elements into it, and deallocates the old array. If doing so has increased the size, it's fine for resize to leave the new spaces uninitialized; whatever member function calls it will be responsible for filling those spaces in. Then every time the array size changes at all (including clear), you can call this function. In some cases, after you call this function, you will have to subsequently shift some of the contents of the array right or left in order to make room for a new value or get rid of an old one. This is of course inefficient; for the purposes of this exercise, however, we won't be worrying about efficiency. If you wanted to do this the "right" way, you'd remember both how long your array is and how much of it is filled, and only reallocate when you reach your current limit or when how much is filled dips below some threshhold.
Add the PointArray::resize(int n) function as specified above to your PointArray class. Give it an appropriate access modifier, keeping in mind that this is meant for use only by internal functions; the public interface is specified below.

HY-150a Programming Recitation 2
General examples

Member Functions Implement public functions to perform the following operations:

Add a Point to the end of the array
void PointArray::pushback(const Point &p)

Insert a Point at some arbitrary position (subscript) of the array, shifting the elements past position to the right
void PointArray::insert(const int position, const Point &p)

Remove the Point at some arbitrary position (subscript) of the array, shifting the remaining elements to the left void
PointArray::remove(const int pos)

Get the size of the array
const int PointArray::getSize() const

Remove everything from the array and sets its size to 0
void PointArray::clear()

HY-150a Programming Recitation 2
General examples

Get a pointer to the element at some arbitrary position in the array,
where positions start at 0 as with arrays
Point *PointArray::get(const int position)
const Point *PointArray::get(const int position) const

If get is called with an index larger than the array size, there is no Point you can return a pointer to, so your function should return a null pointer. Be sure your member functions all behave correctly in the case where you have a 0-length array (i.e., when your PointArray contains no points, such as after the default constructor is called).

This part continues the previous 1A, 1B code, and parts 1B and 1C form the geometry.cpp file.

HY-150a Programming Recitation 2
General examples

```
void PointArray::resize(int newSize) {
    Point *pts = new Point[newSize];
    int minSize = (newSize > size ? size : newSize);
    for (int i = 0; i < minSize; i++)
        pts[i] = points[i];
    delete[] points;
    size = newSize;
    points = pts;
}

void PointArray::clear() {
    resize(0);
}
```

HY-150a Programming Recitation 2
General examples

```
void PointArray::remove(const int pos) {
    if(pos >= 0 && pos < size) { // pos < size implies size > 0
        // Shift everything over to the left
        for(int i = pos; i < size - 2; i++) {
            points[i] = points[i + 1];
        }
        resize(size - 1);
    }
}

Point *PointArray::get(const int pos) {
    return pos >= 0 && pos < size ? points + pos : NULL;
}

const Point *PointArray::get(const int pos) const {
    return pos >= 0 && pos < size ? points + pos : NULL;
}
```

HY-150a Programming Recitation 2
General examples

```
void PointArray::remove(const int pos) {
    if(pos >= 0 && pos < size) { // pos < size implies size > 0
        // Shift everything over to the left
        for(int i = pos; i < size - 2; i++) {
            points[i] = points[i + 1];
        }
        resize(size - 1);
    }
}

Point *PointArray::get(const int pos) {
    return pos >= 0 && pos < size ? points + pos : NULL;
}

const Point *PointArray::get(const int pos) const {
    return pos >= 0 && pos < size ? points + pos : NULL;
}
```

HY-150a Programming Recitation 2
General examples

```
void PointArray::push_back(const Point &p) {
    resize(size + 1);
    points[size - 1] = p;
    // Could also just use insert(size, p);
}

void PointArray::insert(const int pos, const Point &p) {
    resize(size + 1);

    for (int i = size - 1; i > pos; i--) {
        points[i] = points[i-1];
    }

    points[pos] = p;
}
```

HY-150a Programming Recitation 2
General examples

Why do we need const and non-const versions of get ? (Think about what would happen if we only had one or the other, in particular what would happen if we had a const PointArray object.)

We need the `const` versions so that we can return read-only pointers for `const PointArray` objects. (If the `PointArray` object is read-only, we don't want to allow someone to modify a `Point` it contains just by using these functions.) However, many times we will have a non-`const PointArray` object, for which we may want to allow modifying the contained `Point` objects. If we had only `const` accessor functions, then even in such a case we would be returning a `const` pointer. To allow returning a non-`const` pointer in situations where we might want one, we need non-`const` versions of these

HY-150a Programming Recitation 2
General examples

5 Polygon

In this section you will implement a class for a convex polygon called `Polygon`. A *convex polygon* is a simple polygon whose interior is a convex set; that is, if for every pair of points within the object, every point on the straight line segment that joins them is also within the object.

`Polygon` will be an *abstract class* – that is, it will be a placeholder in the class hierarchy, but only its subclasses may be instantiated. `Polygon` will be an immutable type – that is, once you create the `Polygon`, you will not be able to change it.

Throughout this problem, remember to use the `const` modifier where appropriate.

5.1 Foundation

Create the class with two protected members: a `PointArray` and a `static int` to keep track of the number of `Polygon` instances currently in existence.

HY-150a Programming Recitation 2
General examples

5.2 Constructors/Destructors

Implement a constructor that creates a `Polygon` from two arguments: an array of `Points` and the length of that array. Use member initializer syntax to initialize the internal `PointArray` object of the `Polygon`, passing the `Polygon` constructor arguments to the `PointArray` constructor. You should need just one line of code in the actual constructor body.

Implement a constructor that creates a polygon using the points in an existing `PointArray` that is passed as an argument. (For the purposes of this problem, you may assume that the order of the points in the `PointArray` traces out a convex polygon.) You should make sure your constructor avoids the unnecessary work of copying the entire existing `PointArray` each time it is called.

Will the default "memberwise" copy constructor work here? Explain what happens to the `PointArray` field if we try to copy a `Polygon` and don't define our own copy constructor.

Make sure that your constructors and destructors are set up so that they correctly update the `static int` that tracks the number of `Polygon` instances.

HY-150a Programming Recitation 2
General examples

5.3 Member Functions

Implement the following public functions according to the descriptions:

- `area`: Calculates the area of the `Polygon` as a `double`. Make this function pure virtual, so that the subclasses must define it in order to be instantiated. (This makes the class abstract.)
- `getNumPolygons`: Returns the number of `Polygons` currently in existence, and can be called even without referencing a `Polygon` instance. (*Hint:* Use the `static int`.)
- `getNumSides`: Returns the number of sides of the `Polygon`.
- `getPoints`: Returns an unmodifiable pointer to the `PointArray` of the `Polygon`.

HY-150a Programming Recitation 2
General examples

5.4 Rectangle

Write a subclass of `Polygon` called `Rectangle` that models a rectangle. Your code should

- Allow constructing a `Rectangle` from two `Points` (the lower left coordinate and the upper right coordinate)
- Allow construct a `Rectangle` from four `ints`
- Override the `Polygon::area`'s behavior such that the rectangle's area is calculated by multiplying its length by its width, but still return the area as a double.

Both of your constructors should use member initializer syntax to call the base-class constructor, and should have nothing else in their bodies. C++ unfortunately does not allow us to define arrays on the fly to pass to base-class constructors. To allow using member initializer syntax, we can implement a little trick where we have a global array that we update each time we want to make a new array of `Points` for constructing a `Polygon`. You may include the following code snippet in your `geometry.cpp` file:

HY-150a Programming Recitation 2
General examples

```
1 Point constructorPoints[4];
2
3 Point *updateConstructorPoints(const Point &p1, const Point &p2,
    const Point &p3, const Point &p4 = Point(0,0)) {
4     constructorPoints[0] = p1;
5     constructorPoints[1] = p2;
6     constructorPoints[2] = p3;
7     constructorPoints[3] = p4;
8     return constructorPoints;
9 }
```

You can then pass the return value of `updateConstructorPoints(...)` (you'll need to fill in the arguments) as the `Point` array argument of the `Polygon` constructor. (Remember, the name of an array of `T`s is just a `T` pointer.)

HY-150a Programming Recitation 2
General examples

```
1 Point constructorPoints[4];
2
3 Point *updateConstructorPoints(const Point &p1, const Point &p2,
    const Point &p3, const Point &p4 = Point(0,0)) {
4     constructorPoints[0] = p1;
5     constructorPoints[1] = p2;
6     constructorPoints[2] = p3;
7     constructorPoints[3] = p4;
8     return constructorPoints;
9 }
```

You can then pass the return value of `updateConstructorPoints(...)` (you'll need to fill in the arguments) as the `Point` array argument of the `Polygon` constructor. (Remember, the name of an array of `T`s is just a `T` pointer.)

HY-150a Programming Recitation 2
General examples

5.5 Triangle

Write a subclass of `Polygon` called `Triangle` that models a triangle. Your code should

- Construct a `Triangle` from three `Points`
- Override the area function such that it calculates the area using Heron's formula:

$$K = \sqrt{s(s-a)(s-b)(s-c)}$$

where a, b, and c are the side lengths of the triangle and $s = \frac{a+b+c}{2}$.

Use the same trick as above for calling the appropriate base-class constructor. You should not need to include any code in the actual function body.

HY-150a Programming Recitation 2
General examples

5.7 Putting it All Together

Write a small function with signature `void printAttributes(Polygon *)` that prints the area of the polygon and prints the (x, y) coordinates of all of its points.

Finally, write a small program (a `main` function) that does the following:

- Prompts the user for the lower-left and upper-right positions of a `Rectangle` and creates a `Rectangle` object accordingly
- Prompts the user for the point positions of a `Triangle` and creates a `Triangle` object accordingly
- Calls `printAttributes` on the `Rectangle` and `Triangle`, printing an appropriate message first.

HY-150a Programming Recitation 2
General examples

5.1.1 geometry.h

```
1  class Polygon {
2  protected:
3      static int numPolygons;
4      PointArray points;
5
6  public:
7      Polygon(const PointArray &pa);
8      Polygon(const Point points[], const int numPoints);
9      virtual double area() const = 0;
10     static int getNumPolygons() {return numPolygons;}
11     int getNumSides() const {return points.getSize();}
12     const PointArray *getPoints() const {return &points;}
13     ~Polygon() {--numPolygons;}
14 };
```

HY-150a Programming Recitation 2
General examples

5.1.2 geometry.cpp

```
1 int Polygon::n = 0;
2
3 Polygon::Polygon(const PointArray &pa) : points(pa) {
4     ++numPolygons;
5 }
6
7 Polygon::Polygon(const Point pointArr[], const int numPoints) :
    points(pointArr, numPoints) {
8     ++numPolygons;
9 }
```

HY-150a Programming Recitation 2
General examples

5.2 Rectangle

5.2.1 geometry.h

```
1 class Rectangle : public Polygon {
2 public:
3     Rectangle(const Point &a, const Point &b);

4     Rectangle(const int a, const int b, const int c, const int d);
5     virtual double area() const;
6 };
```

HY-150a Programming Recitation 2
General examples

5.2.2 geometry.cpp

```
1
2 Point constructorPoints[4];
3
4 Point *updateConstructorPoints(const Point &p1, const Point &p2,
      const Point &p3, const Point &p4 = Point(0,0)) {
5     constructorPoints[0] = p1;
6     constructorPoints[1] = p2;
7     constructorPoints[2] = p3;
8     constructorPoints[3] = p4;
9     return constructorPoints;
10 }
11
12 Rectangle::Rectangle(const Point &ll, const Point &ur)
13     : Polygon(updateConstructorPoints(ll, Point(ll.getX(), ur.getY()
          ),
14                                       ur, Point(ur.getX(), ll.getY()
```

HY-150a Programming Recitation 2
General examples

```
15
16 Rectangle::Rectangle(const int llx, const int lly, const int urx,
       const int ury)
17     : Polygon(updateConstructorPoints(Point(llx, lly), Point(llx,
           ury),
18                                       Point(urx, ury), Point(urx,
                                             lly)), 4) {}
19
20 double Rectangle::area() const {
21     int length = points.get(1)->getY() - points.get(0)->getY();
22     int width  = points.get(2)->getX() - points.get(1)->getX();
23     return std::abs((double)length * width);
24 }
```

(You'll need to add `#include <cmath>` at the top of your file to use the `abs` function.)

HY-150a Programming Recitation 2
General examples

5.3 Triangle

5.3.1 geometry.h

```
1 class Triangle : public Polygon {
2 public:
3     Triangle(const Point &a, const Point &b, const Point &c);
4     virtual double area() const;
5 };
```

HY-150a Programming Recitation 2
General examples

5.4 geometry.cpp

```
1  Triangle::Triangle(const Point &a, const Point &b, const Point &c)
2      : Polygon(updateConstructorPoints(a, b, c), 3) {}
3
4  double Triangle::area() const {
5      int dx01 = points.get(0)->getX() - points.get(1)->getX(),
6          dx12 = points.get(1)->getX() - points.get(2)->getX(),
7          dx20 = points.get(2)->getX() - points.get(0)->getX();
8      int dy01 = points.get(0)->getY() - points.get(1)->getY(),
9          dy12 = points.get(1)->getY() - points.get(2)->getY(),
10         dy20 = points.get(2)->getY() - points.get(0)->getY();
11
12     double a = std::sqrt(dx01*dx01 + dy01*dy01),
13            b = std::sqrt(dx12*dx12 + dy12*dy12),
14            c = std::sqrt(dx20*dx20 + dy20*dy20);
15
16     double s = (a + b + c) / 2;
17
18     return std::sqrt( s * (s-a) * (s-b) * (s-c) );
19 }
```

HY-150a Programming Recitation 2
General examples

5.5 main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  #include "geometry.h"
5
6  void printAttributes(Polygon *p) {
7      cout << "p's area is " << p->area() << ".\n";
8
9      cout << "p's points are:\n";
10     const PointArray *pa = p->getPoints();
11     for(int i = 0; i < pa->getSize(); ++i) {
12         cout << "(" << pa->get(i)->getX() << ", " << pa->get(i)->
               getY() << ")\n";
13     }
```

HY-150a Programming Recitation 2
General examples

```
14 }
15
16 int main(int argc, char *argv[]) {
17     cout << "Enter lower left and upper right coords of rectangle as
           four space separated integers: ";
18     int llx, lly, urx, ury;
19     cin >> llx >> lly >> urx >> ury;
20     Point ll(llx, lly), ur(urx, ury);
21     Rectangle r(ll, ur);
22     printAttributes(&r);
23
24     cout << "Enter three coords of triangle as six space separated
          integers: ";
25     int x1, y1, x2, y2, x3, y3;
26     cin >> x1 >> y1 >> x2 >> y2 >> x3 >> y3;
27     Point a(x1, y1), b(x2, y2), c(x3, y3);
28     Triangle t(a, b, c);
29     printAttributes(&t);
30
31     return 0;
32 }
```

HY-150a Programming Recitation 2
General examples

```
14 }
15
16 int main(int argc, char *argv[]) {
17     cout << "Enter lower left and upper right coords of rectangle as
           four space separated integers: ";
18     int llx, lly, urx, ury;
19     cin >> llx >> lly >> urx >> ury;
20     Point ll(llx, lly), ur(urx, ury);
21     Rectangle r(ll, ur);
22     printAttributes(&r);
23
24     cout << "Enter three coords of triangle as six space separated
          integers: ";
25     int x1, y1, x2, y2, x3, y3;
26     cin >> x1 >> y1 >> x2 >> y2 >> x3 >> y3;
27     Point a(x1, y1), b(x2, y2), c(x3, y3);
28     Triangle t(a, b, c);
29     printAttributes(&t);
30
31     return 0;
32 }
```

HY-150a Programming Recitation 2
General examples

5.6 Questions

1. In the `Point` class, what would happen if the constructors were private?
2. Describe what happens to the fields of a `Polygon` object when the object is destroyed.
3. Why did we need to make the fields of `Polygon` protected?

For the next question, assume you are writing a function that takes as an argument a `Polygon *` called `polyPtr`.

4. Imagine that we had overridden `getNumSides` in each of `Rectangle` and `Triangle`. Which version of the function would be called if we wrote `polyPtr->getNumSides()`? Why?

HY-150a Programming Recitation 2
General examples

5.6 Questions

1. If the constructors were private, then we would not be able to create any `Point` objects.
2. When a `Polygon` is destroyed, the counter for number of `Polygons` created is decremented, and the `PointArray`'s destructor is implicitly called.
3. We had to make the fields of `Polygon` protected so that they could be accessed from `Rectangle` and `Triangle`, but not by arbitrary outside code.
4. The `getNumSides` from `Polygon` would be called, because the function is not `virtual`.

HY-150a Programming Recitation 2

References

1)HY352,Τεχνολογία Λογισμικού
http://www.csd.uoc.gr/~hy352/

2)Examples Inheritance
http://www.programming-techniques.com/search?q=Inheritance

3) Templates
http://www.programming-techniques.com/search?q=templates

4)C107 Programming Paradigms.
http://web.stanford.edu/class/cs107l/

5)Advanced Inheritance and Virtual Methods.
http://web.stanford.edu/class/cs107l/handouts/07-Virtual-Functions.pdf

6)Introduction to C++ MIT OPEN COURSES.
https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-096-introduction-to-c-january-iap-2011/

7)CS106L, more complete exploration of the C++ language.
http://web.stanford.edu/class/cs106l/lectures.html

8)GotW #5 Solution: Overriding Virtual Functions, Sutter's Mill.
https://herbsutter.com/2013/05/22/gotw-5-solution-overriding-virtual-functions/

9) Nice C++ Links
https://akrzemi1.wordpress.com/