ΠΡΟΓΡΑΜΜΑΤΙΣΤΙΚΕΣ ΤΕΧΝΙΚΕΣ 2025-2026

https://helios.ntua.gr/course/view.php?id=869

27/2/2026

Διδάσκοντες:

Νίκος Παπασπύρου, Καθ.
Βασίλης Βεσκούκης, Καθ.
Νίκος Λεονάρδος, Επ. Καθ.
Πέτρος Ποτίκας, ΕΔΙΠ
Γιώργος Σιόλας, ΕΔΙΠ
Σωτήρης Κοκόσης, ΕΔΙΠ

progtech@cslab.ece.ntua.gr

Διαφάνειες παρουσιάσεων

- Η γλώσσα προγραμματισμού C++
- Αρχές αντικειμενοστρεφούς προγραμματισμού
- Δομές δεδομένων

Object-oriented ~~programming~~ thinking & programming

Δομές (struct)

- Ενα struct είναι μια δομή δεδομένων που ορίζεται από τον προγραμματιστή και μέσα του περιέχει επιμέρους συστατικά στοιχεία (πεδία)
  - το όνομα του συμπεριφέρεται ως όνομα **τύπου,** δηλαδή μπορούμε να ορίζουμε μεταβλητές του τύπου αυτού
  - περιέχει ατομικές μεταβλητές ή και άλλα struct
  - η δήλωσή του ολοκληρώνεται με ";"

Παράδειγμα

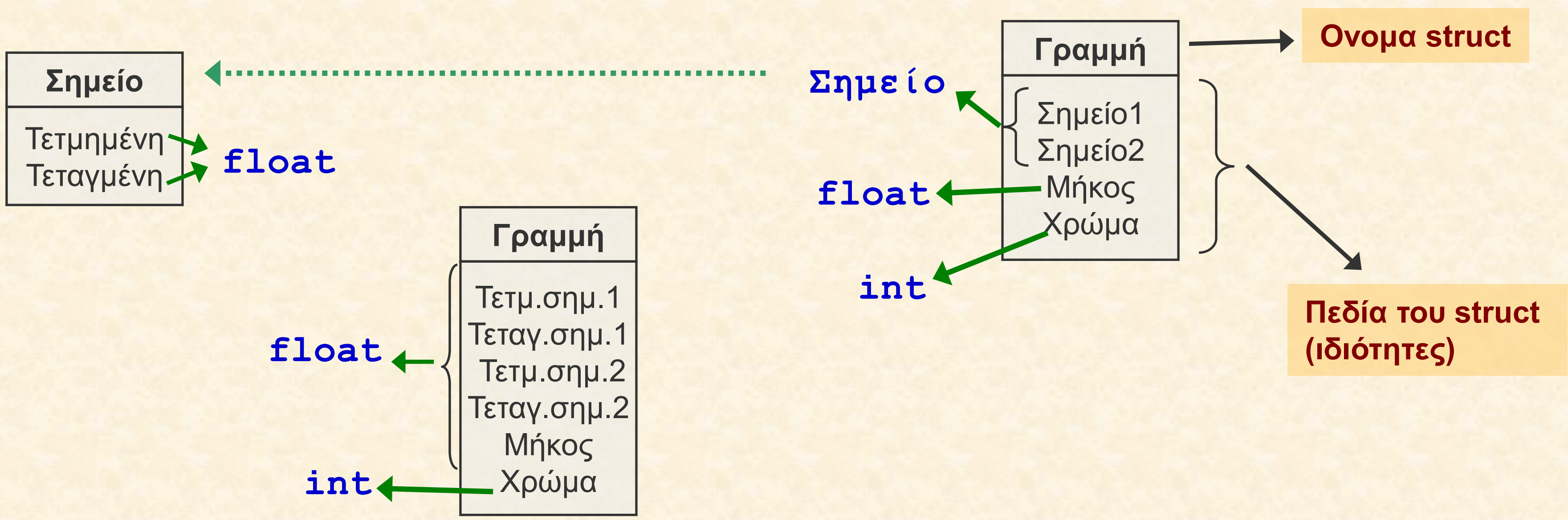
```
// ορισμός
struct myLine {
      float x1, y1, x2, y2;
      };           // προσοχή εδώ! βάζουμε και ";"



// Χρήση
myLine L1, *L2;   // όπως "int i, j;"
...
L1.x1=0; L1.y1=0;        // Χωρίς μτβλ δείκτη
L1.x2=1; L1.y2=1;

L2->x1=0; L2->y1=0;      //  Με μτβλ δείκτη
L2->x2=1; L2->y2=1;
```

Παραδείγματα

```
struct Person {
  char[20] firstName, lastName;
  int birthDate, birthMonth, birthYear;
  char[10] height;
  float weight;
  char[20] colorOfEyes;
};


struct Polygon {
  char[30] description;
  int koryfes;
  float x[max_koryfes];
  float y[max_koryfes];
  float area;
};
```

Το struct ως εργαλείο μοντελοποίησης

- Σύνθετες οντότητες (έννοιες) του πραγματικού κόσμου δεν παριστάνονται απλά και κατανοητά με ατομικές μεταβλητές μνήμης



Τι είναι οι μιγαδικοί αριθμοί;

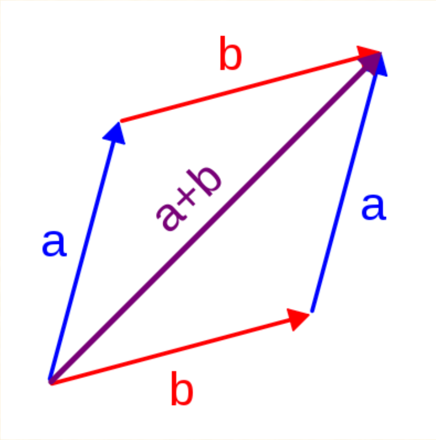
- Wikipedia: Στα μαθηματικά, οι μιγαδικοί αριθμοί είναι μία επέκταση του συνόλου των πραγματικών αριθμών με την προσθήκη του στοιχείου $\boldsymbol{i}$, που λέγεται φανταστική μονάδα, και έχει την ιδιότητα:

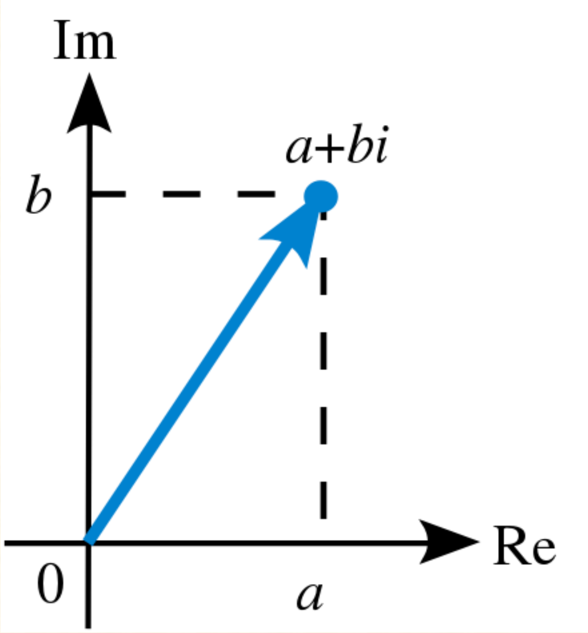
$$\boldsymbol{i}^2 = -1$$

- Κάθε μιγαδικός αριθμός μπορεί να γραφτεί στη μορφή a + $\boldsymbol{i}$ b, όπου a, b $\in$ **R**

Μιγαδικοί αριθμοί

- a: πραγματικό μέρος
- b: φανταστικό μέρος
- Αριθμητικές πράξεις, π.χ. πρόσθεση

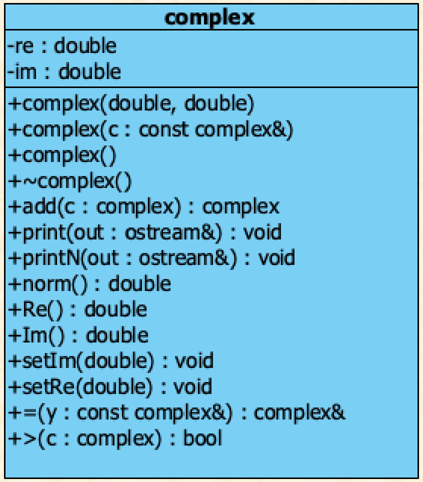




- Για περισσότερα, *ρωτήστε το μαθηματικό σας!*

Μιγαδικοί αριθμοί

- Ζητούμενο: να κατασκευάσουμε πρόγραμμα που χειρίζεται μιγαδικούς αριθμούς, δηλαδή:
  - Παριστά
  - Δημιουργεί
  - Προσθέτει κλπ (γενικά: "χειρίζεται")
  - Εμφανίζει στην οθόνη



"Χειρίζεται":
παριστάνει, δημιουργεί, μεταβάλλει, εμφανίζει, διαγράφει, και γενικά "κάνει ό,τι μπορεί να γίνει" με τους μιγαδικούς

Η κλάση "έχει την ευθύνη" των δεδομένων που περιέχει

Εισαγωγή στο object-orientation

- Μιγαδικοί αριθμοί, με struct
  (όχι object-oriented)

```
struct complex { double re, im; };

complex c_make(double r, double i){
  complex result;
  result.re = r; result.im = i;
  return result;
}

complex c_add(complex c1, complex c2){
  return c_make(c1.re + c2.re,
                c1.im + c2.im);
}
```

παράσταση

δημιουργία

πρόσθεση

Εισαγωγή στο object-orientation

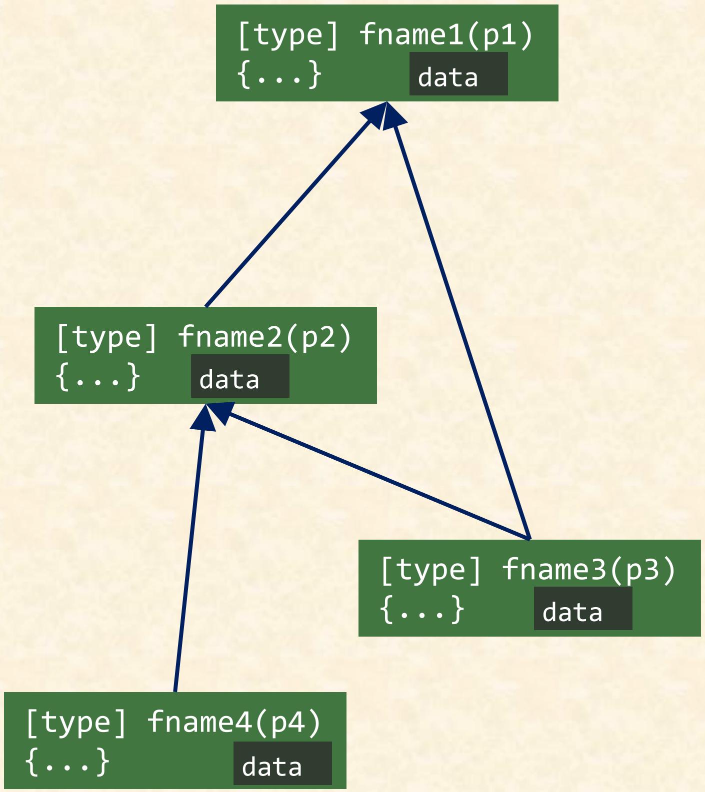
- Μιγαδικοί αριθμοί με struct (συνέχεια)

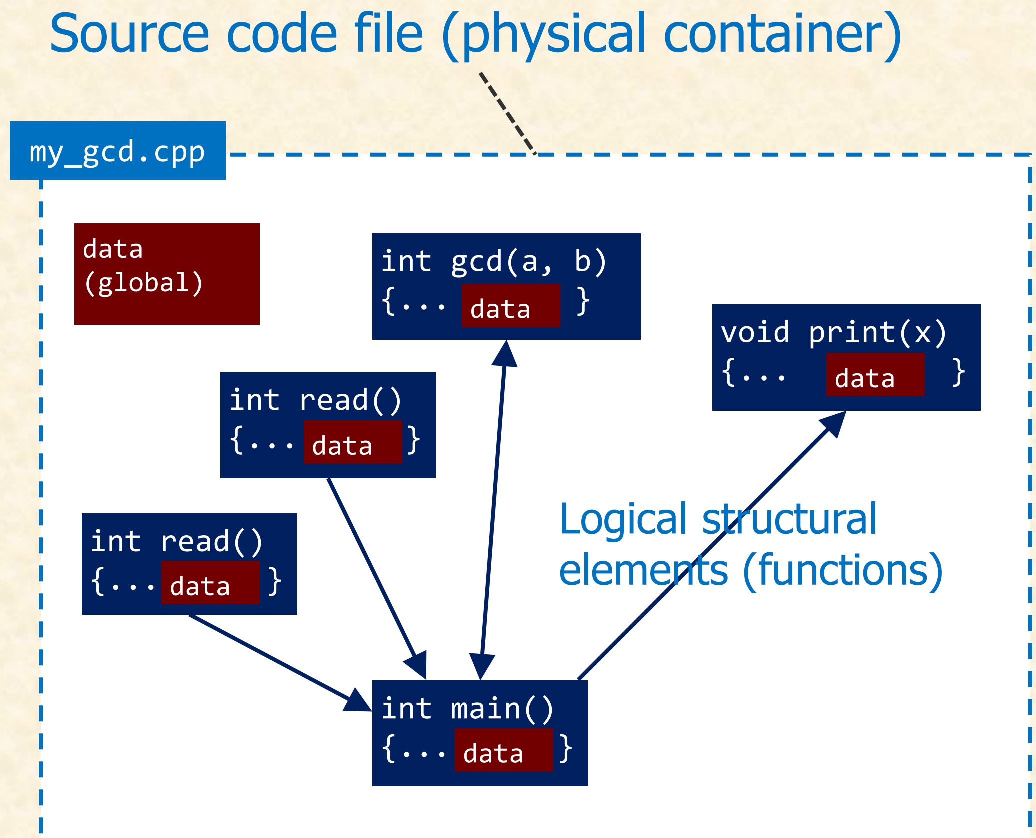
```
void c_print(ostream &out, complex c) {
  out << c.re << "+" << c.im << "i" << endl;
}
```

εμφάνιση

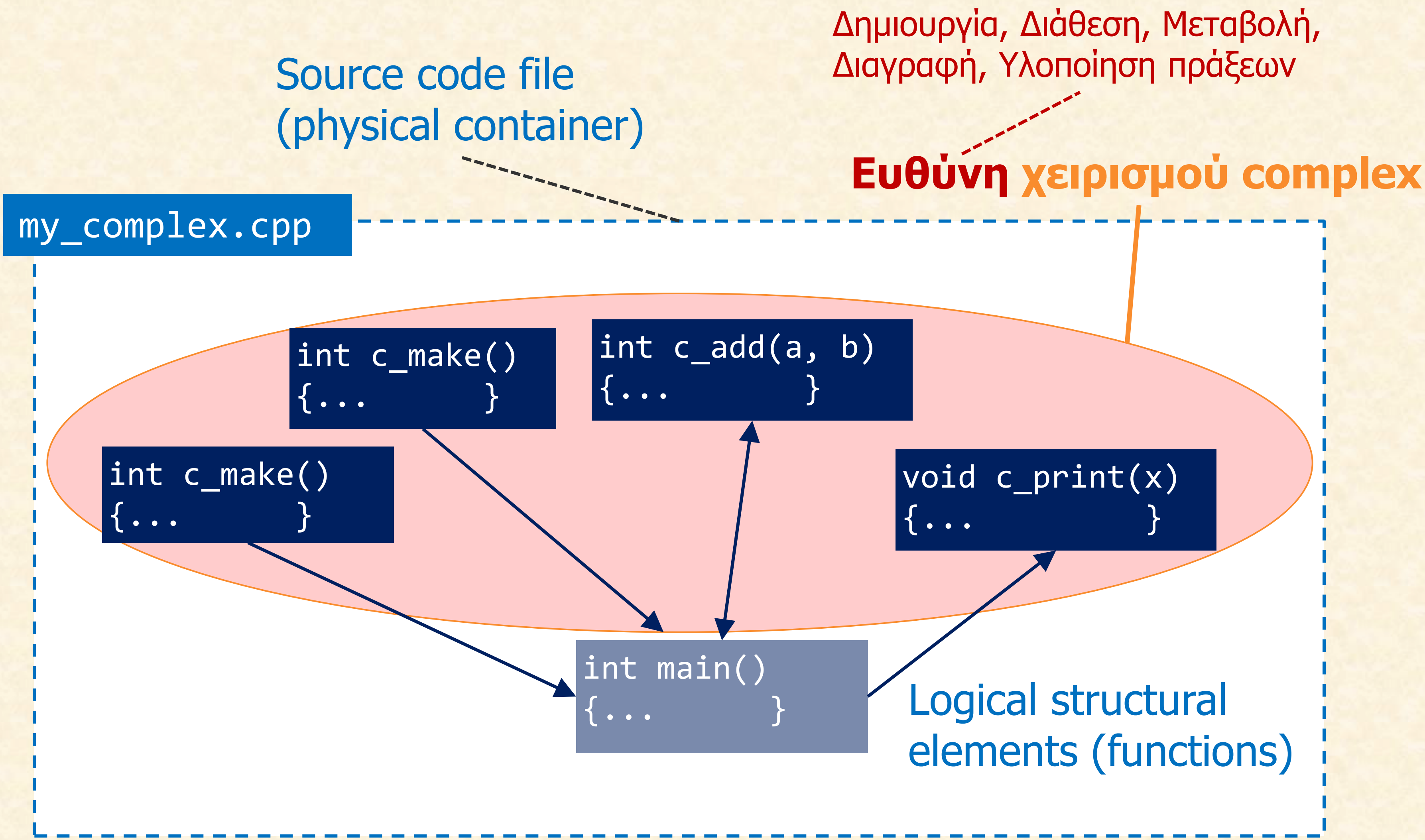
```
int main() {
  complex c1 = c_make(3, 4);
  complex c2 = c_make(1, 2);
  complex c = c_add(c1, c2);
  c_print(cout, c);
  c_print(cerr, c);
}
```

Δομή προγράμματος (επανάληψη)





Εισαγωγή στο object-orientation



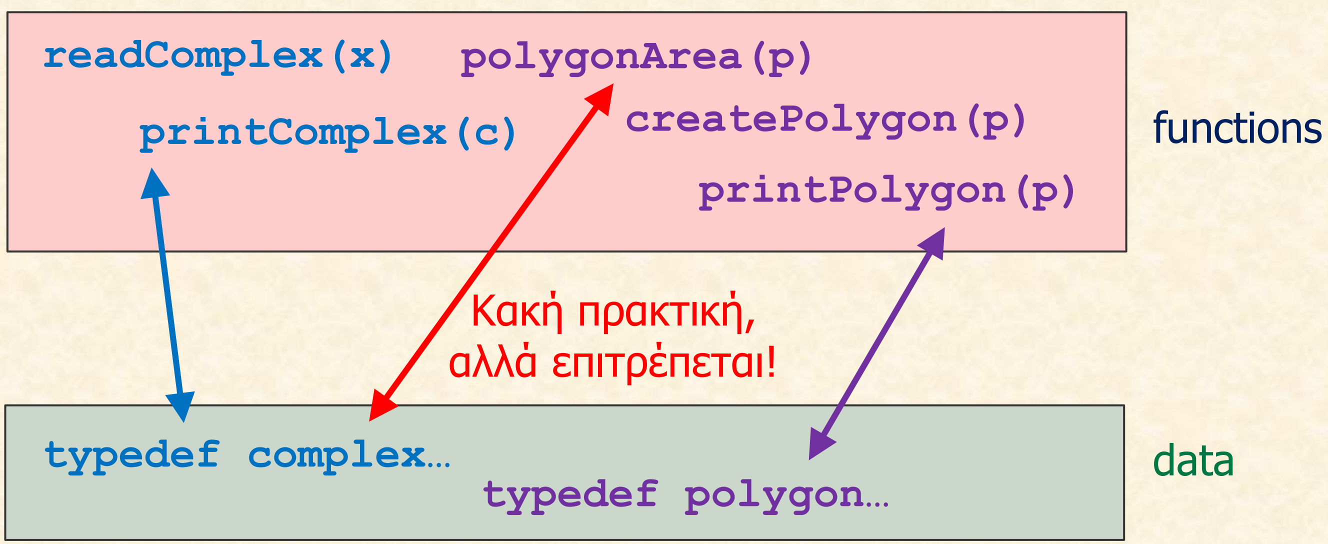
Εισαγωγή στο object-orientation

- Κλάσεις: οριζόμενες από τον χρήστη δομές προγράμματος, οι οποίες περιέχουν
  - Ιδιότητες (ορολογία: πεδία, fields, attributes)
  - Συμπεριφορά (ορολογία: μεθόδους, methods, member functions)

- Οι κλάσεις είναι:
  - Προγραμματιστικό εργαλείο
  - Εργαλείο μοντελοποίησης δεδομένων και της συμπεριφοράς αυτών

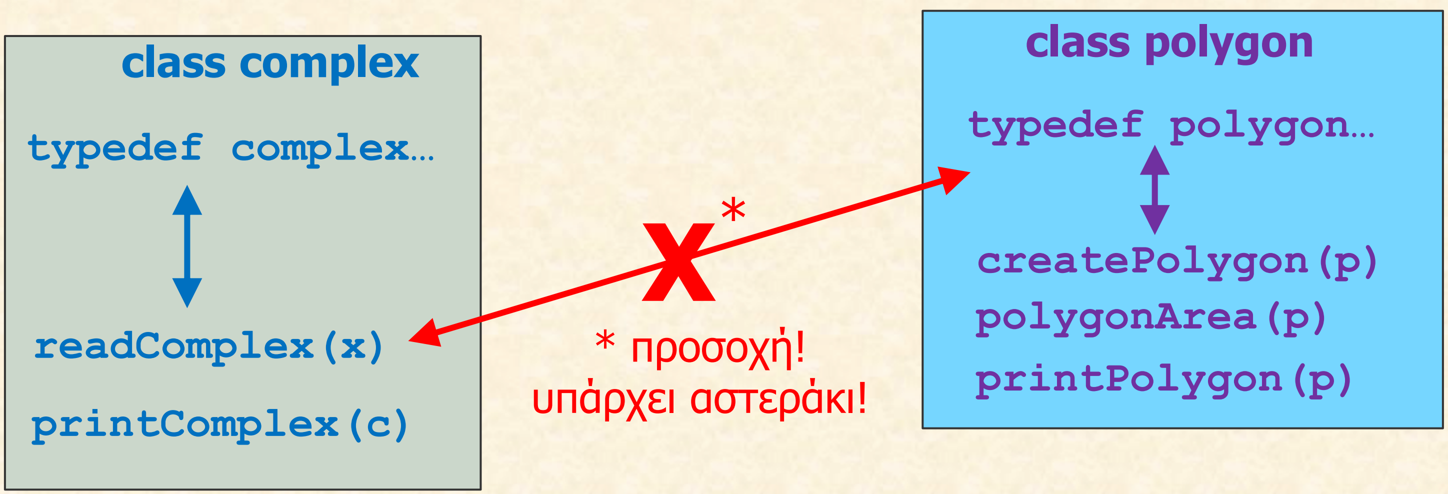
Shift of paradigm: χωρίς χρήση κλάσεων

- Ορισμός των δεδομένων από το πεδίο του προβλήματος ("εκφώνηση!")
- Ορισμός συναρτήσεων που δέχονται ως παραμέτρους τα δεδομένα που ορίστηκαν
- Χρήση των συναρτήσεων



Shift of paradigm: με χρήση κλάσεων

- **Data abstraction**:
  Ορισμός κάθε τύπου δεδομένων στο πεδίο του προβλήματος ως μία κλάση, που περιέχει
  - την περιγραφή των δεδομένων [πεδία]
  - τις συναρτήσεις που τα χειρίζονται [μέθοδοι]
- Δημιουργία και χρήση αντικειμένων



Κλάσεις - ορολογία

- Μη-αντικειμενοστρεφείς τύποι: **τύπος -> μεταβλητή**
  - **`int i`** [Μεταβλητή τύπου **`int`**]
  - **`Point p1`** [μεταβλητή τύπου **`Point`** (struct που ορίστηκε από τον προγραμματιστή και είναι νέος τύπος)]
- Κλάσεις: **Κλάση** -> **αντικείμενο (object)**
  - Όπως ορίζουμε μεταβλητές ενός τύπου, δημιουργούμε **αντικείμενα** μιας **κλάσης** που περιέχουν **δεδομένα (πεδία, κατάσταση)** και **συναρτήσεις (μεθόδους, συμπεριφορά)**
- Object-oriented programming = αντικειμενοστρ**ε**φής προγραμματισμός

Εισαγωγή στο object-orientation

Source code file (physical container)

my_oo_program.cpp

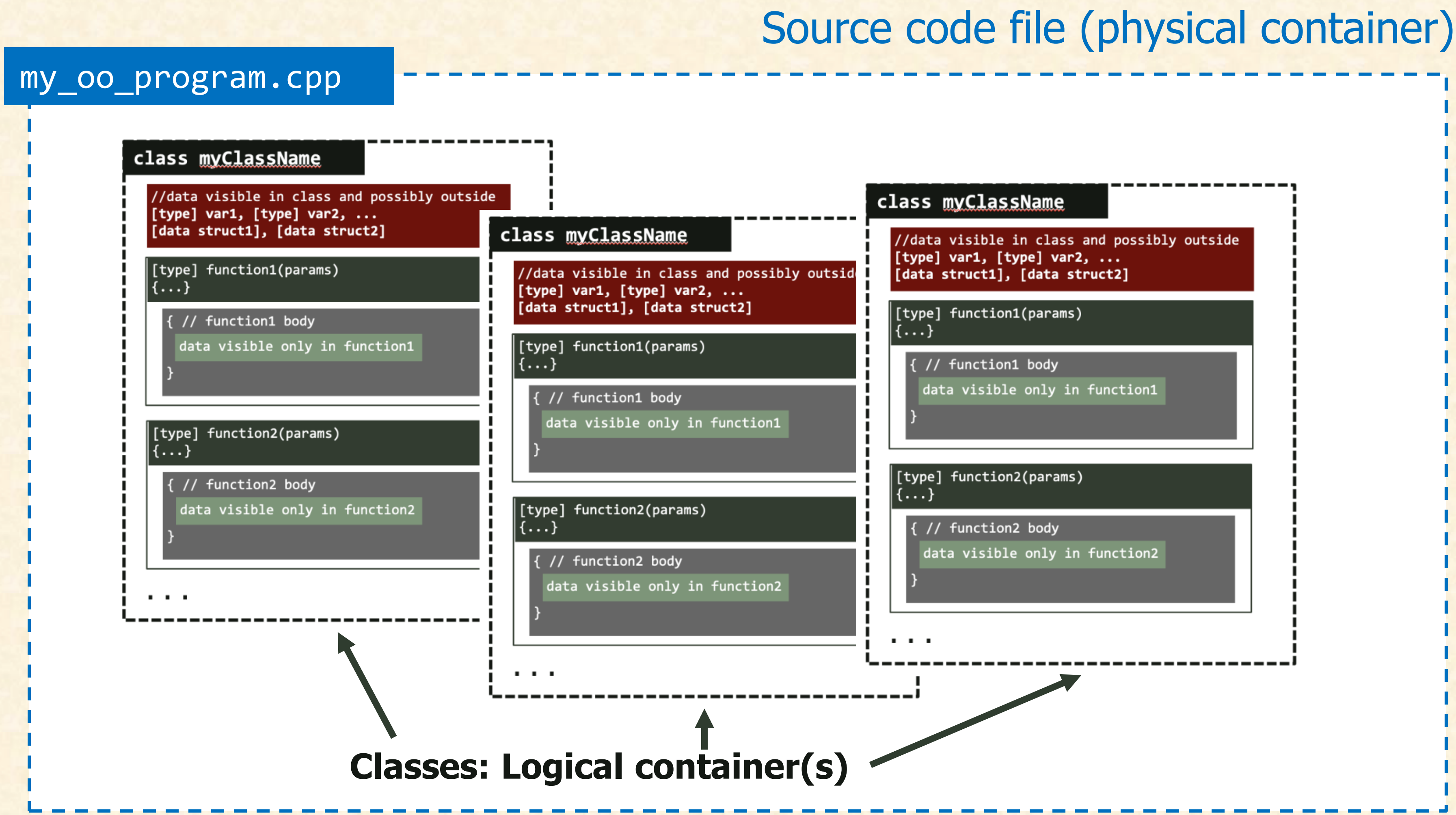
```
class myClassName
  //data visible in class and possibly outside
  [type] var1, [type] var2, ...
  [data struct1], [data struct2]

  [type] function1(params)
  {...}
    { // function1 body
      data visible only in function1
    }

  [type] function2(params)
  {...}
    { // function2 body
      data visible only in function2
    }

  ...
```

Logical container

Data within the responsibility of the logical container (fields)

Logical structural elements that do something: (~~functions~~ methods)

Εισαγωγή στο object-orientation



Κλάσεις και προγράμματα

- Μέχρι σήμερα: ένα πρόγραμμα "με μία έννοια" είναι μια κλάση (υπό προϋποθέσεις), διότι:
  - Περιέχει **μεταβλητές μνήμης** (**πεδία**)
  - Περιέχει **συναρτήσεις** (**μεθόδους**)
  - Η εκτέλεση ξεκινάει αυτόματα (main)

Σημείωση:

Το αν ένα "πρόγραμμα" (όπως το ξέρουμε μέχρι τώρα) μπορεί να θεωρηθεί παρόμοιο με κλάση, εξαρτάται από το αν τα δεδομένα και οι λειτουργίες του αντιστοιχούν σε μία οντότητα ενός "πεδίου προβλήματος" που διαχειρίζεται

Κλάσεις...

- Μοντελοποιούν κάποια δεδομένα και τη συμπεριφορά τους
- Οποιοδήποτε πρόγραμμα μπορεί να θεωρηθεί ως περιγραφή (υλοποίηση) της συμπεριφοράς των δεδομένων του
- Προσοχή:
  - Μπορούμε να γράψουμε ένα "καλό" πρόγραμμα που να έχει όλα τα ποιοτικά χαρακτηριστικά της έννοιας "κλάση"
  - Μπορούμε να γράψουμε μια "κακή" κλάση που να περιέχει "ό,τι να' ναι".

Ορατότητα μελών κλάσης

Θα επανέλθουμε, στη συζήτηση περί κληρονομικότητας

```
class [όνομα] {
  public:
    Πεδία/μέθοδοι ορατά εντός και εκτός της κλάσης

  private: // default!
    Πεδία/μέθοδοι ορατά μόνο μέσα στην κλάση

  protected:
    Πεδία/μέθοδοι ορατά μέσα στην κλάση και στις
   κλάσεις-παιδιά

};
```

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

Κλάσεις… (iii)

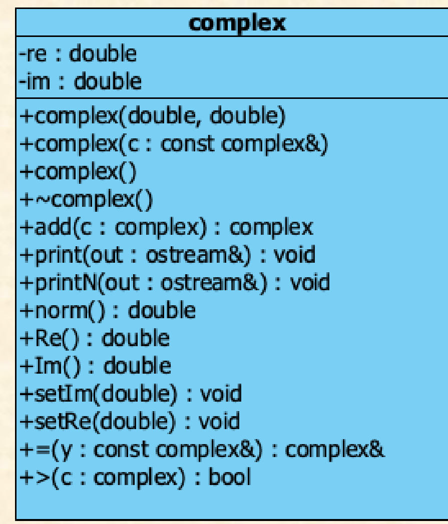
- Οι κλάσεις σχεδιάζονται με διαφορετική φιλοσοφία (object-oriented)
  - αφαίρεση δεδομένων / απόκρυψη πληροφοριών
  - ενθυλάκωση (encapsulation) = πεδία και μέθοδοι μέσα στα αντικείμενα
  - περισσότερα στην πορεία

- Ακολουθεί παράδειγμα
  - Μαντέψτε: μιγαδικοί αριθμοί

Εισαγωγή στο object-orientation

- Μιγαδικοί αριθμοί, με class (object-oriented)

```
class complex {
public:
   complex(double r = 0.0, double i = 0.0);
   complex add(complex c);
   void print(ostream &out);
private:
   double re, im;
};
```



Δημόσιο και ιδιωτικό μέρος
Πεδία και μέθοδοι, κατασκευαστής

Ορισμός μεθόδων **εκτός** κλάσης

```
class complex {
public:
   complex(double r = 0.0, double i = 0.0);
   double norm();
   ...
};
```

complex.hpp

```
complex::complex(double r, double i) {
   re = r; im = i;
}

double complex::norm() {
   return sqrt(re * re + im * im);
}
```

complex.cpp

Ίσως και σε ξεχωριστά αρχεία!

Ορισμός μεθόδων **εντός** κλάσης

```
class complex {
public:
   complex(double r = 0.0, double i = 0.0) {
      re = r; im = i;
   }
   double norm() {
      return sqrt(re * re + im * im);
   }
   ...
};
```

complex.hpp

Αναγκαστικά στο ίδιο αρχείο!

- Υλοποίηση: μέθοδοι **inline**

Εισαγωγή στο object-orientation

```
complex::complex(double r, double i) {
  re = r; im = i;
}

complex complex::add(complex c) {
  return complex(re + c.re, im + c.im);
}

void complex::print(ostream &out) {
  out << re << "+" << im << "i" << endl;
}

int main() {
  complex c1(3, 4), c2(1, 2);
  complex c = c1.add(c2);
  c.print(cout);
}
```

Εισαγωγή στο object-orientation

- Πώς σκεφτήκαμε;;;

Κάθε μιγαδικός αριθμός μπορεί να γραφτεί στη μορφή a + $\boldsymbol{i}$ b, όπου a, b $\in \mathbf{R}$

```
class complex {
public:
   ...
private:
   double re, im;
};
```

Θέλουμε να ορίζουμε μιγαδικούς

```
class complex {
public:
   complex(double r, double i);
   ...
};
```

Θέλουμε να προσθέτουμε μιγαδικούς

Θέλουμε να εμφανίζουμε μιγαδικούς

```
class complex {
public:
   ...
   complex add(complex c);

   void print(ostream &out);
...
};
```

Εισαγωγή στο object-orientation

- Εργαλεία object-oriented υλοποίησης (part I)
  - Προσδιοριστές ορατότητας
  - Κατασκευαστές (constructors)
  - Καταστροφείς (destructors)
  - Μέθοδοι getter / setter
  - Φίλες συναρτήσεις
  - Υπερφόρτωση τελεστών (overloading)
  - Μέθοδοι const
  - Στατικά πεδία και μέθοδοι

Εισαγωγή στο object-orientation

- Εργαλεία object-oriented υλοποίησης (part I)
  - Προσδιοριστές ορατότητας
  - **Κατασκευαστές (constructors)**
  - **Καταστροφείς (destructors)**
  - Μέθοδοι getter / setter
  - Φίλες συναρτήσεις
  - Υπερφόρτωση τελεστών (overloading)
  - Μέθοδοι const
  - Στατικά πεδία και μέθοδοι

Κατασκευαστές (constructors)

- Εκτελούνται αυτόματα με τη δήλωση ενός αντικειμένου
- Εχουν πάντα το όνομα της κλάσης
- ΔΕΝ έχουν τύπο
- Μπορεί να είναι περισσότεροι του ενός, αλλά πρέπει να έχουν διαφορετική λίστα παραμέτρων (signature)

Καταστροφείς (destructors) (i)

- Καλούνται όταν καταστρέφονται τα αντικείμενα (αποδεσμεύεται η μνήμη τους)
- Κάθε κλάση έχει μόνο έναν destructor

```
class complex {
public:
   ...
   ~complex() {
     cout << "a complex number dies... :-("
          << endl;
   }
   ...
};
```

Καταστροφείς (destructors) (ii)

```
int main() {
  complex c1(3, 4), c2(1, 2);
  complex c = c1 + c2;
  cout << c << endl;
}
```

- Ο καταστροφέας θα κληθεί τρεις φορές για τα αντικείμενα `c1`, `c2` και `c`, όταν τελειώσει η `main`.
- Το ίδιο και εδώ (γιατί;)

```
int main() {
  complex c1(3, 4), c2(1, 2);
  cout << c1 + c2 << endl;
}
```

Εισαγωγή στο object-orientation

- Εργαλεία object-oriented υλοποίησης (part I)
  - Προσδιοριστές ορατότητας
  - Κατασκευαστές (constructors)
  - Καταστροφείς (destructors)
  - **Μέθοδοι getter / setter**
  - Φίλες συναρτήσεις
  - Υπερφόρτωση τελεστών (overloading)
  - Μέθοδοι const
  - Στατικά πεδία και μέθοδοι

Good practice

Getters/Setters

Καλή πρακτική: μέθοδοι getters / setters

- Η ορατότητα των μελών μιας κλάσης από έξω πρέπει να είναι η εντελώς απαραίτητη:
  - Ορισμός αντικειμένων (constructors)
  - Απόδοση/ανάγνωση τιμών σε/από πεδία (setters/getters)
  - Υπηρεσίες της κλάσης προς τα έξω

- Με βάση αυτά (γενική "γραμμή"):
  - Όλα τα μέλη μιας κλάσης δηλώνονται private
  - Μέθοδοι getters, setters και υπηρεσίες δηλώνονται public

Καλή πρακτική: getters

- Μια μέθοδος "getter" επιστρέφει:
  - Την τιμή ενός πεδίου ή ένα αντικείμενο του τύπου του πεδίου
  - Το αποτέλεσμα ενός υπολογισμού (private μεθόδου της κλάσης) που εκτελείται εντός

- Καλή ιδέα:
  μια μέθοδος getter ΔΕΝ πρέπει να εμφανίζει output στην οθόνη

Καλή πρακτική: setters

- Μια μέθοδος "setter":
  - Συνήθως είναι τύπου void, εκτός αν πρέπει να επιστρέψει το αποτέλεσμα της εκτέλεσής της
  - Δίνει τις τιμές που δέχεται ως παράμετρους, στα αντίστοιχα πεδία της κλάσης
  - Προκαλεί την εκτέλεση όλων των μεθόδων που υπολογίζουν τιμές εξαρτημένων πεδίων (αν υπάρχουν), ώστε όλα τα πεδία να έχουν σωστές (=συνεπείς) τιμές

Παράδειγμα getters / setters

```
class line
    float x1,y1,x2,y2,len
    line(float X1,Y1,X2,Y2)
    float calclen()
```

public

```
line myline(0,0,1,1);

myline.calclen();

cout << myline.len;   // 1.4142

myline.x1 = -1;       // move the
myline.y1 = -1;       // one end
cout << myline.len;   // 1.4142
                      //(ERROR!)

myline.calclen();
cout << myline.len;   // 2.8284
```

Η ευθύνη της συνέπειας των τιμών των πεδίων του αντικειμένου, ανήκει στον χρήστη του: κακή ιδέα!

Παράδειγμα getters / setters

```
class line

  float x1,y1,x2,y2,len

  line(float X1,Y1,X2,Y2) {
    x1 = X1; y1 = Y1;
    x2 = X2; y2 = Y2;
    calclen();
  }

  float setX1(float X1) {
    x1 = X1;
    calclen(); }

  float calclen()
```

setter

private public

```
line myline(0,0,1,1);

cout << myline.len;  // 1.4142

myline.setX1(-1);
myline.setY1(-1);
cout << myline.len;  // 2.8284

cout << myline.x1 << "\n";
cout << myline.y1 << "\n";
```

Η ευθύνη της συνέπειας των τιμών των πεδίων του αντικειμένου, ανήκει στην κλάση!

Παράδειγμα getters / setters

```
class line

    float x1,y1,x2,y2,len

    line(float X1,Y1,X2,Y2) {
      x1 = X1; y1 = Y1;
      x2 = X2; y2 = Y2;
      calclen();
    }

    void setX1(float X1) {
      x1 = X1;
      calclen(); }
    // same for setY1...
```

setter

```
    float X1() {
      return x1; }
    float Y1() {
      return y1; }

    // same for X2,Y2,len...
```

getter

```
    float calclen()
```

```
line myline(0,0,1,1);

cout << myline.LEN();  // 1.4142

myline.setX1(-1);
myline.setY1(-1);
cout << myline.LEN();  // 2.8284

cout << myline.X1() << "\n";
cout << myline.Y1() << "\n";
```

Η ευθύνη της συνέπειας των τιμών των πεδίων του αντικειμένου ανήκει στην κλάση!

Εισαγωγή στο object-orientation

- Εργαλεία object-oriented υλοποίησης (part I)
  - Προσδιοριστές ορατότητας
  - Κατασκευαστές (constructors)
  - Καταστροφείς (destructors)
  - Μέθοδοι getter / setter
  - Φίλες συναρτήσεις
  - Υπερφόρτωση τελεστών (overloading)
  - Μέθοδοι const
  - Στατικά πεδία και μέθοδοι

Φίλες συναρτήσεις (friend functions)

- Μια συνάρτηση ή κλάση B, που δηλώνεται ως "φίλη" της κλάσης A, έχει πρόσβαση σε όλα τα μέλη της A, ανεξάρτητα από την ορατότητά τους

```
class Line {
private:    // private is the default!
    float x1, y1, x2, y2, length;
public:
    Line(float X1, float Y1, float X2, float Y2) {
        x1=X1; y1=Y1; x2=X2; y2=Y2;
        length=sqrt(pow(x1-x2,2)+pow(y1-y2,2));
    }
    void print() { cout << length << endl; }
};

float hypot(Line L) {
    return sqrt(pow(L.x1-L.x2,2)+pow(L.y1-L.y2,2));  // ERROR!
}
```

Φίλες συναρτήσεις (friend functions)

```
class Line {
private:    // private is the default!
    float x1, y1, x2, y2, length;
public:
    Line(float X1, float Y1, float X2, float Y2) {
        x1=X1; y1=Y1; x2=X2; y2=Y2;
        length=sqrt(pow(x1-x2,2)+pow(y1-y2,2));
    }
    Line(float X1, float Y1, float X2, float Y2) :
        x1(X1), y1(Y1), x2(X2), y2(Y2)
        { length=sqrt(pow(x1-x2,2)+pow(y1-y2,2));
    }
    void print() { cout << length << endl; }
    friend float hypot(Line);
};

float hypot(Line L) {
    return sqrt(pow(L.x1-L.x2,2)+pow(L.y1-L.y2,2));   // OK!
}
```

Μέθοδοι και φίλες συναρτήσεις (i)

- Οι τελεστές με δύο τελούμενα μπορούν να υλοποιηθούν με δύο τρόπους
- #1: ως μέθοδοι της κλάσης

```
class complex {
public:
   ...
   complex operator+(const complex &y) {
      return complex(re + y.re, im + y.im);
   }
   ...
};
```

Μέθοδοι και φίλες συναρτήσεις (ii)

- Οι τελεστές με δύο τελούμενα μπορούν να υλοποιηθούν με δύο τρόπους
- #2: ως φίλες συναρτήσεις

```
class complex {
public:
   ...
   friend complex operator+(const complex &x,
                            const complex &y) {
     return complex(x.re + y.re, x.im + y.im);
   }
   ...
};
```

Εισαγωγή στο object-orientation

- Εργαλεία object-oriented υλοποίησης (part I)
  - Προσδιοριστές ορατότητας
  - Κατασκευαστές (constructors)
  - Καταστροφείς (destructors)
  - Μέθοδοι getter / setter
  - Φίλες συναρτήσεις
  - Υπερφόρτωση τελεστών (overloading)
  - Μέθοδοι const
  - Στατικά πεδία και μέθοδοι

Υπερφόρτωση τελεστών

- Operator overloading
  - Απόδοση της δικής μας επιθυμητής συμπεριφοράς σε τελεστές (+, -, /, ...) εντός ή εκτός κλάσεων
  - Οι τελεστές μπορούν να προγραμματίζονται να λειτουργούν σε αντικείμενα και να κάνουν αυτό που θέλουμε εμείς
- Προσοχή στους συνειρμούς με τα μαθηματικά!

Overloadable operators
`+ - * / = < > += -= *= /= << >> <<= >>= == != <= >= ++ -- % & ^ ! | ~ &= ^= |= && || %= [] () , ->* -> new delete new[] delete[]`

Υπερφόρτωση τελεστών

```
type operator sign (parameters)
  { /* ο κώδικας πηγαίνει εδώ...*/ }
```

ο τελεστής

υποχρεωτική δεσμευμένη λέξη

τύπος τελουμένων

τύπος που θα επιστρέφει ο τελεστής

```
complex operator + (myvector v)
  { }
```

Υπερφόρτωση τελεστών

- Αν ένα τελεστής ορίζεται εκτός κλάσης
  - Οι παράμετροι είναι τα τελούμενα
  - Απαιτείται τουλάχιστον μία παράμετρος
  - `complex operator+(complex a, int b)`

- Αν ένας τελεστής ορίζεται μέσα σε μία κλάση
  - Το πρώτο τελούμενο είναι υποχρεωτικά το αντικείμενο της κλάσης `(*this)`
  - Δεν απαιτείται παράμετρος στον ορισμό (γιατί;)
  - `complex operator+(int b)`

Εισαγωγή στο object-orientation

- Υπερφόρτωση (overloading) τελεστών: **+** και **<<**

```
class complex {
public:
  complex(double r = 0.0, double i = 0.0);
  friend complex operator+(complex c1,
                           complex c2);
  friend ostream& operator<<(ostream &out,
                             complex c);
private:
  double re, im;
};
```

Εισαγωγή στο object-orientation

```
complex operator+(complex c1, complex c2) {
  return complex(c1.re + c2.re,
                 c1.im + c2.im);
}

ostream& operator<<(ostream &out,
                    complex c) {
  out << c.re << "+" << c.im << "i";
  return out;
}

int main() {
  complex c1(3, 4), c2(1, 2);
  complex c = c1 + c2;
  cout << c << endl;
}
```

Μέθοδοι και φίλες συναρτήσεις (iii)

- Και στις δύο περιπτώσεις η χρήση είναι ίδια: `c1 + c2`
- Στην περίπτωση #1 το αντικείμενο `c1` είναι αυτό για το οποίο καλείται η μέθοδος και το `c2` είναι η παράμετρος
- Στην περίπτωση #2 και τα δύο αντικείμενα είναι παράμετροι (πρόκειται για συνάρτηση, όχι για μέθοδο)

Μέθοδοι και φίλες συναρτήσεις (iv)

- Ο τελεστής εκτύπωσης όμως μπορεί να υλοποιηθεί μόνο ως φίλη συνάρτηση (γιατί;)

```
class complex {
public:
  ...
  friend ostream& operator<<(ostream &out,
                             const complex &x) {
    out << x.re << " + " << x.im << "i";
    return out;
  }
  ...
};
```

Υπερφόρτωση (overloading) (i)

```
int  f() { ... }                      // 1
int  f(int x) { ... }                 // 2
void f(bool x, int y) { ... }         // 3
int  f(const complex &c) { ... }      // 4
```

- Διαφέρουν στο πλήθος και τους τύπους των παραμέτρων

αν δεν υπήρχε η 2;

```
f();                    // 1          // 1
f(42);                  // 2          // 4
f(true, 17);            // 3          // 3
complex i(0, 1);
f(i);                   // 4          // 4
f(42.0);                // 2  γιατί;  // 4  γιατί;
```

Υπερφόρτωση (overloading) (ii)

```
int  f(int x) { ... }                    // 2
```

- Απαγορεύεται να προστεθεί η παρακάτω, που διαφέρει από την 2 μόνο στον τύπο επιστροφής

```
bool f(int y) { ... }
```

- Συμπέρασμα: πολύπλοκος μηχανισμός
  - περιπλέκεται λόγω των αυτόματων μετατροπών (coercions)
  - να χρησιμοποιείται με φειδώ

Πριν προχωρήσουμε

Παράδειγμα

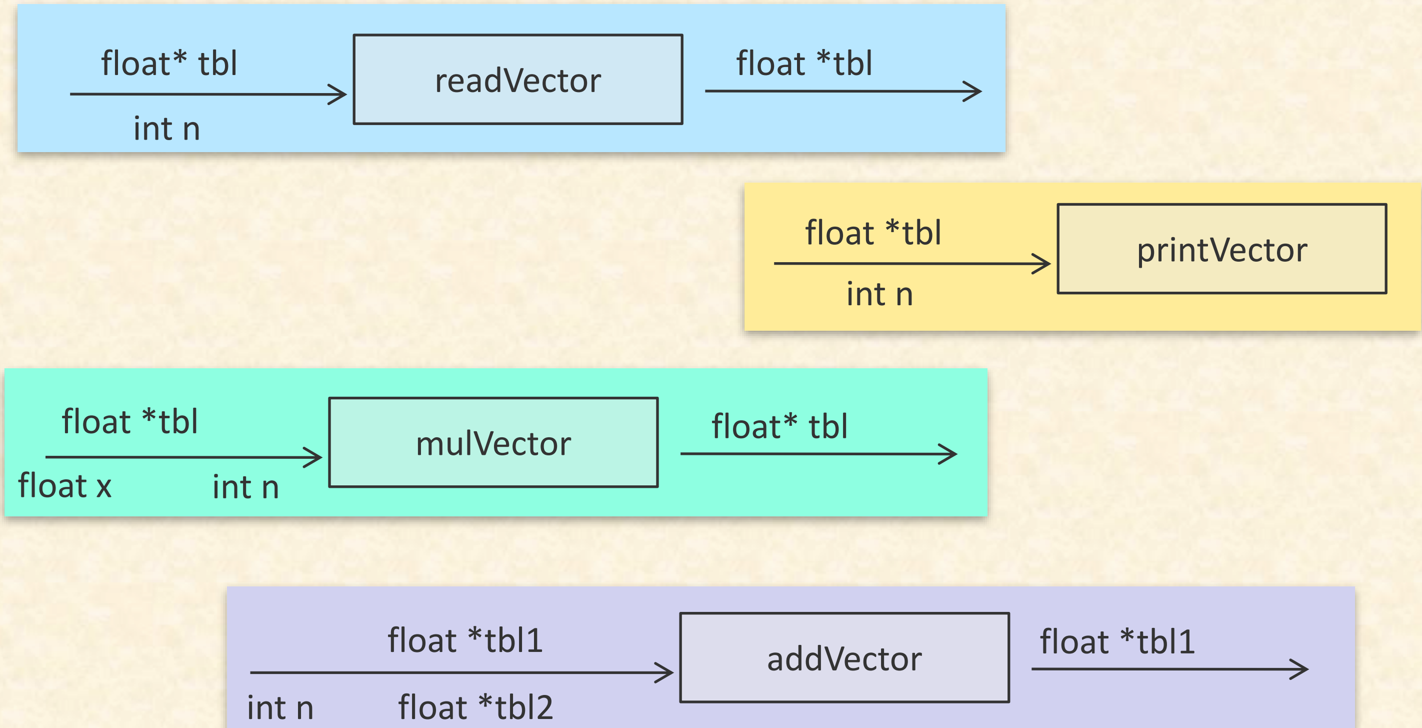
Παράδειγμα: διανύσματα Ν διαστάσεων

- Ενα Ν-διάστατο διάνυσμα παριστάνεται με έναν μονοδιάστατο πίνακα με Ν-στοιχεία
- Ζητείται
  - Να δημιουργούμε διανύσματα
  - Να διαβάζουμε τα στοιχεία τους από το πληκτρολόγιο
  - Να εμφανίζουμε τα στοιχεία τους στην οθόνη
  - Να πολλαπλασιάζουμε διάνυσμα με αριθμό
  - Να προσθέτουμε διανύσματα μεταξύ τους
- Παράδειγμα (Ν=3)
  - **A**=(2, 4, 5), **B**=(3, 1, 0), **A*2**=(4, 8, 10), **A+B**=(5, 5, 5)

Σχεδίαση χωρίς τη χρήση κλάσεων

- Συνάρτηση που δέχεται ως παράμετρο πίνακα με N στοιχεία και τον γεμίζει με δεδομένα από το πληκτρολόγιο [readvector]
- Συνάρτηση που δέχεται ως παράμετρο έναν πίνακα με N στοιχεία και τον εκτυπώνει στην οθόνη [printvector]
- Συνάρτηση που δέχεται ως παραμέτρους έναν μονοδιάστατο πίνακα με N στοιχεία και έναν αριθμό, και πολλαπλασιάζει όλα τα στοιχεία του πίνακα με τον αριθμό αυτό [multvector]
- Συνάρτηση που δέχεται ως παραμέτρους δύο πίνακες με N στοιχεία και προσθέτει ένα προς ένα τα στοιχεία του δεύτερου σε αυτά του πρώτου [addvector]

Σχεδίαση χωρίς τη χρήση κλάσεων




```
main()
float* vector1, a;
readVector(vector1, N); cin>>a;
mulVector(vector1, N, a); printVector(vector1,N);
```

Υλοποίηση χωρίς τη χρήση κλάσεων

```
const int N = 2;

// READ VECTOR
void readVector(float *tbl, int n) {
  for (int i=0; i<n; i++) {
    cout<<"Element "<<i<<": "; cin>>tbl[i];
  }
}
// PRINT VECTOR
void printVector(float *tbl, int n) {
  int i;
  for (i=0; i<N-1; i++) cout<<tbl[i]<<", ";
  cout<<tbl[i];
}
// MULTIPLY VECTOR
void mulVector(float *tbl, int n, float f) {
  for (int i=0; i<N; i++) tbl[i] *=  f;
}
// ADD VECTOR
void addVector(float *tbl1, float *tbl2,
               int n) {
  for (int i=0; i<N; i++) tbl1[i] += tbl2[i];
}
```

```
int main() {

  float *p = new(nothrow) float[N];
  if (!p) {
    exit(1);
  }

  float *p2 = new float[N];
  float a;

  readVector(p, N);
  readVector(p2, N);
  cout<<"A:";cin>>a;
  mulVector(p,N,a);
  printVector(p, N);

  addVector(p, p2, N);
  printVector(p, N);
}
```

Σχεδίαση με χρήση κλάσεων

- Ορισμός μιας κλάσης myvector που περιέχει
  - Έναν μονοδιάστατο πίνακα με N στοιχεία [vector]
  - Μέθοδο ανάγνωσης από το πληκτρολόγιο
  - Μέθοδο εκτύπωσης στην οθόνη
  - Μέθοδο που δέχεται ως παράμετρο έναν αριθμό και πολλαπλασιάζει όλα τα στοιχεία του διανύσματος με αυτόν
  - Μέθοδο που δέχεται ως παράμετρο ένα vector και προσθέτει ένα προς ένα τα στοιχεία του στα στοιχεία του πίνακα της κλάσης

Σχεδίαση με χρήση κλάσεων

class Vector

float elements[N]

read()

print()

multiply(float a)

add(Vector Q)

```
main()
Vector v1(N), v2(N); float a;
v1.read(); cin>>a; v1.multiply(a); v1.print();
v2.read (); v2.read();
v1.add(v2); v1.print();
```

Σχεδίαση με κλάσεις (+Quiz!)

```
class Vector {
    float* elements;
    int E;
public:
    Vector(int);
    void read();
    void print();
    friend void readVector(float*, int);
    friend void printVector(float*, int);
};

Vector::Vector(int n) {
    E = n;
    elements = new float[E];
    for (int i=0;i<E;i++) elements[i]=0;
}
void Vector::read() {
    readVector(this->elements, E);
}
void Vector::print() {
    printVector(this->elements, E);
}
```

Περισσότεροι κατασκευαστές (i)

- Με χρήση υπερφόρτωσης
- Κοινός κατασκευαστής

```
complex(double r, double i) {
  re = r; im = i;
}
```

ή ισοδύναμα:

```
complex(double r, double i):
  re(r), im(i) {}
```

Η τελευταία μορφή διαχωρίζει την αρχικοποίηση των πεδίων του αντικειμένου από άλλες εντολές.

Περισσότεροι κατασκευαστές (ii)

- Default constructor

```
complex() { re = im = 0; }
```

ή ισοδύναμα:

```
complex(): re(0), im(0) {}
```

Κατασκευαστής χωρίς παραμέτρους, καλείται αυτόματα όταν έχουμε π.χ.

```
complex c;
```

Αν δεν οριστεί άλλος κατασκευαστής για μία κλάση, η C++ ορίζει αυτόματα έναν default

Περισσότεροι κατασκευαστές (iii)

- Copy constructor

```
complex(const complex &c) {
  re = c.re; im = c.im;
}
```

ή ισοδύναμα:

```
complex(const complex &c):
  re(c.re), im(c.im) {}
```

Καλείται αυτόματα όταν έχουμε π.χ.

```
complex c1(c);
complex c2 = c;
```

Περισσότεροι κατασκευαστές (iv)

- Copy constructor
  - Αν δεν οριστεί για μία κλάση, η C++ ορίζει αυτόματα έναν copy constructor που αντιγράφει ένα-ένα τα πεδία του αντικειμένου
  - Αυτό δεν είναι πάντα το επιθυμητό *γιατί;* (θα το συζητήσουμε σε λίγο στο παράδειγμα με **`weight`** και **`totalWeight`**)

Περισσότεροι κατασκευαστές (v)

- Κατασκευαστής που επιτρέπει αυτόματη μετατροπή τύπου

```
complex(double r) { re = r; im = 0; }
```

ή ισοδύναμα:

```
complex(double r): re(r), im(0) {}
```

Καλείται αυτόματα όταν έχουμε π.χ.

```
complex c1(5);
complex c2 = 5;
```

```
void f (const complex &c);
f(5);
```

Τελεστής ανάθεσης

```
const complex& operator=(const complex &y) {
  re = y.re;
  im = y.im;
  return *this;  // Αναφορά στο τρέχον αντικείμενο!
}
```

Καλείται αυτόματα όταν έχουμε π.χ.

```
c1 = c2;
```

Αν δεν οριστεί για μία κλάση, η C++ ορίζει αυτόματα έναν τελεστή ανάθεσης που αντιγράφει ένα-ένα τα πεδία του αντικειμένου

Εισαγωγή στο object-orientation

- Εργαλεία object-oriented υλοποίησης (part I)
  - Προσδιοριστές ορατότητας
  - Κατασκευαστές (constructors)
  - Καταστροφείς (destructors)
  - Μέθοδοι getter / setter
  - Φίλες συναρτήσεις
  - Υπερφόρτωση τελεστών (overloading)
  - Μέθοδοι const
  - Στατικά πεδία και μέθοδοι

Μέθοδοι const

- Δεν επιτρέπεται να αλλάξουν τιμές στα πεδία της κλάσης τους
- Καλή πρακτική για αποφυγή σφαλμάτων όταν εξ' αρχής (=κατά τη σχεδίαση της κλάσης) γνωρίζουμε ότι μια μέθοδος δεν θέλουμε να μεταβάλει τιμές πεδίων

- Αντικείμενα const
  - Τιμές πεδίων που δεν αλλάζουν από τις αρχικές
  - Μπορούν να καλέσουν μόνο const μεθόδους

Μέθοδοι const (i)

```
class Person {
public:
  Person(double w): weight(w) {}
  double getWeight() {
    return weight;
  }
  void setWeight(double w) {
    weight = w;
  }
private:
  double weight;
};
```

Μέθοδοι const (ii)

```
int main() {
  Person me(75);
  cout << me.getWeight() << endl;
  me.setWeight(76);
  cout << me.getWeight() << endl;
  const Person constantlyFatPanda(630);
  // Το παρακάτω είναι λάθος:
  constantlyFatPanda.setWeight(730);
  // Αλλά δυστυχώς και αυτό είναι λάθος (γιατί;):
  cout << constantlyFatPanda.getWeight()
       << endl;
}
```

Μέθοδοι const (iii)

```
class Person {
public:
   Person(double w): weight(w) {}
   double getWeight() const {
      return weight;
   }
   void setWeight(double w) {
      weight = w;
   }
private:
   double weight;
};
```

Αυτή η μέθοδος μπορεί να καλείται για σταθερά (**`const`**) αντικείμενα

Αυτή όχι!

Μέθοδοι const (iv)

```
int main() {
  Person me(75);
  cout << me.getWeight() << endl;
  me.setWeight(76);
  cout << me.getWeight() << endl;
  const Person constantlyFatPanda(630);
  // Αυτό εξακολουθεί να είναι λάθος:
  constantlyFatPanda.setWeight(730);
  // Αλλά τώρα αυτό είναι σωστό:
  cout << constantlyFatPanda.getWeight()
       << endl;
}
```

Εισαγωγή στο object-orientation

- Εργαλεία object-oriented υλοποίησης (part I)
  - Προσδιοριστές ορατότητας
  - Κατασκευαστές (constructors)
  - Καταστροφείς (destructors)
  - Μέθοδοι getter / setter
  - Φίλες συναρτήσεις
  - Υπερφόρτωση τελεστών (overloading)
  - Μέθοδοι const
  - Στατικά πεδία και μέθοδοι

Στατικά πεδία και μέθοδοι

- Χωρίς static:
  - Με τη δημιουργία κάθε αντικειμένου αντιγράφονται στη μνήμη όλα τα μέλη του
- Static πεδία:
  - Ολα τα αντικείμενα "μοιράζονται" το ίδιο αντίγραφο του πεδίου
  - **Πρέπει** να αρχικοποιηθούν πριν τη δημιουργία αντικειμένων με χρήση του τελεστή εμβέλειας (::)
  - Μπορεί να χρησιμοποιηθούν και χωρίς αντικείμενα: className::staticMemberName
  - Μόνο static μέθοδοι μπορούν να τα μεταβάλουν

Στατικές μέθοδοι

- Ολα τα αντικείμενα της κλάσης μοιράζονται την ίδια εκδοχή (αντίγραφο) μιας static μεθόδου
  - Μπορεί να καλεί μόνο άλλες static μεδόδους
  - Μπορεί να μεταβάλει τιμές static πεδίων
  - Μπορεί να κληθεί χωρίς δημιουργία αντικειμένου: className::staticMethodName()

Στατικά πεδία και μέθοδοι (i)

```
class Person {
public:
   Person(double w);
  ~Person();
             double getWeight();
   static double getTotalWeight();
private:
             double weight;
   static double totalWeight;
};

double Person::totalWeight = 0.0;
```

Στατικά πεδία και μέθοδοι (ii)

```
Person::Person(double w): weight(w) {
  totalWeight += weight;
}
Person::~Person() {
  totalWeight -= weight;
}
double Person::getWeight() {
  return weight;
}
double Person::getTotalWeight() {
  return totalWeight;
}
```

Στατικά πεδία και μέθοδοι (iii)

```
int main() {
  Person x(65), y(100);
  cout << x.getWeight()                        65
       << endl
       << x.getTotalWeight()                   165
       << endl;
  Person *z = new Person(85);
  cout << Person::getTotalWeight()             250
       << endl;
  delete z;
  cout << y.getTotalWeight()                   165
       << endl;
}
```

γιατί;;;