ΠΡΟΓΡΑΜΜΑΤΙΣΤΙΚΕΣ ΤΕΧΝΙΚΕΣ 2024-2025

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

Διδάσκοντες:

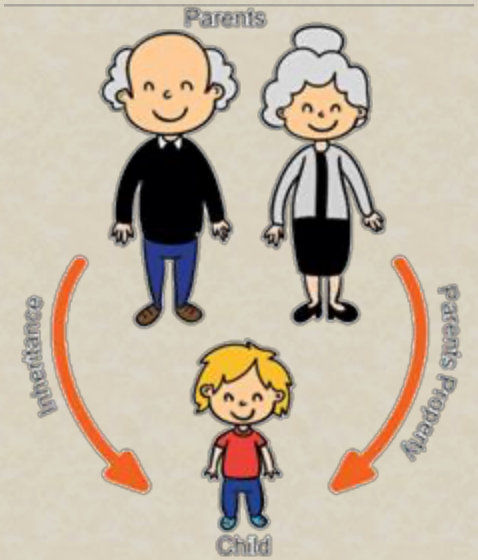
Βασίλης Βεσκούκης, Καθ.
Νίκος Λεονάρδος, Επ.Καθ.
Άρης Παγουρτζής, Καθ.
Νίκος Παπασπύρου, Καθ.
Σωτήρης Κοκόσης, ΕΔΙΠ
Πέτρος Ποτίκας, ΕΔΙΠ
Γιώργος Σιόλας, ΕΔΙΠ

progtech@courses.softlab.ntua.gr

7/3/2025*

Διαφάνειες παρουσιάσεων

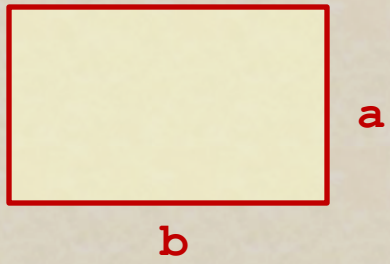
- ✓ Η γλώσσα προγραμματισμού C++
- ✓ Αρχές αντικειμενοστρεφούς προγραμματισμού
- ✓ Δομές δεδομένων

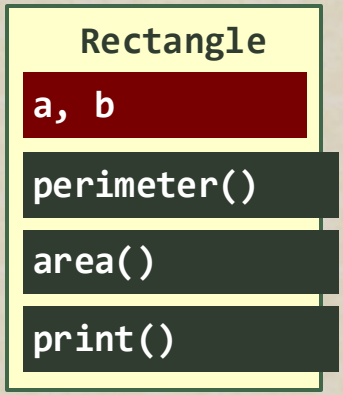


Κληρονομικότητα (συνέχεια)

Πολλαπλή κληρονομικότητα

- Η κλάση Rectangle

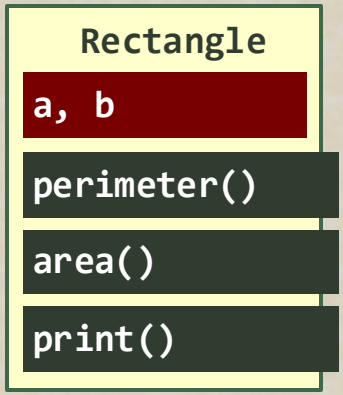





```
class Rectangle {
  protected:
    double a, b;
  public:
    Rectangle;
    Rectangle(double A, double B);

    double A() const;
    double B() const;
    void setA(double A);
    void setB(double B);

    double perimeter() const;
    double area() const;
    void print(ostream &out) const;

    friend ostream& operator<<(ostream &out, const Rectangle &r);
};
```

Πολλαπλή κληρονομικότητα

```
class Rectangle {      // The parent class
 protected:
  double a, b;
 public:
  Rectangle() : a(0), b(0) {}
  Rectangle(double A, double B) : a(A), b(B) {}
  double A() const { return a;}
  double B() const { return b;}
  void setA(double A) { a = A; }
  void setB(double B) { b = B; }
  double perimeter() const { return 2*(a+b); }
  double area() const { return a*b; }
  void print(ostream &out) const {
    out << *this << endl;
  }
  friend ostream& operator<<(ostream &out, const Rectangle &r) {
    out << "[a=" << r.a << " b=" << r.b << " a="
        << r.area() << " p=" << r.perimeter() << "]";
    return out;
  }
};
```

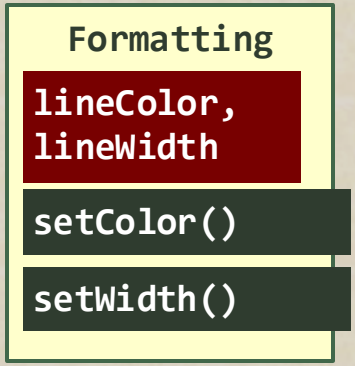



```
int main (){
  Rectangle r(3, 10);
  r.print(cout);
  r.setA(15);
  r.print(cout);
}
```

```
[a=3 b=10 a=30 p=26]
[a=15 b=10 a=150 p=50]
```

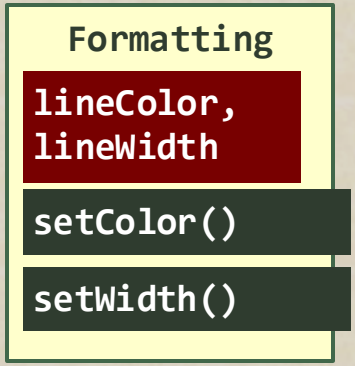
Πολλαπλή κληρονομικότητα

- Η κλάση Formatting




```
class Formatting {
  protected:
    int lineWidth;
    string lineColor;
  public:
    Formatting();
    Formatting(int W, string C);

    void setColor(string c);
    void setWidth(int w);
    friend ostream& operator<<(ostream &out, const Formatting &d);
};
```

Πολλαπλή κληρονομικότητα



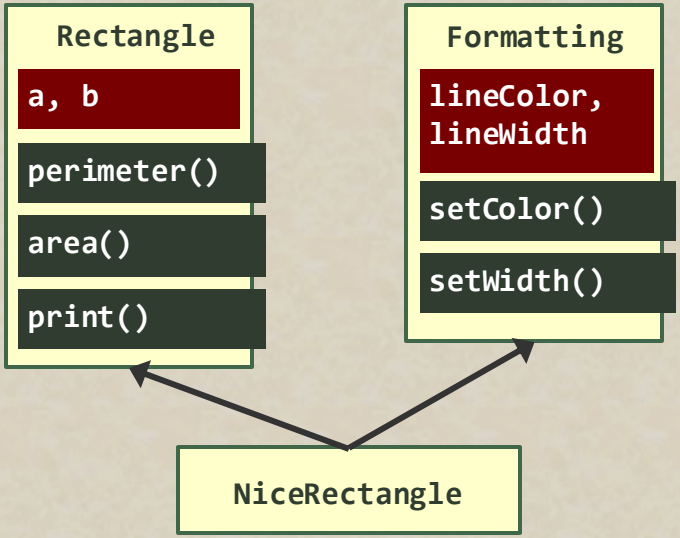

```
class Formatting {
 protected:
  int lineWidth;
  string lineColor;
 public:
  Formatting() : lineWidth(0), lineColor("transparent") {}
  Formatting(int W, string C) : lineWidth(W), lineColor(C) {}
  void setColor(string c) { lineColor = c; }
  void setWidth(int w) { lineWidth = w; }
  friend ostream& operator<<(ostream &out, const Formatting &d) {
    out << "[color=" << d.lineColor
        << " width=" << d.lineWidth << "]";
    return out;
  }
};
```

```
int main (){
  Formatting f1(1,"blue");
  cout << f1 << endl;
  f1.setColor("red");
  cout << f1; }
```

```
[color=blue width=1]
[color=red width=1]
```

Πολλαπλή κληρονομικότητα

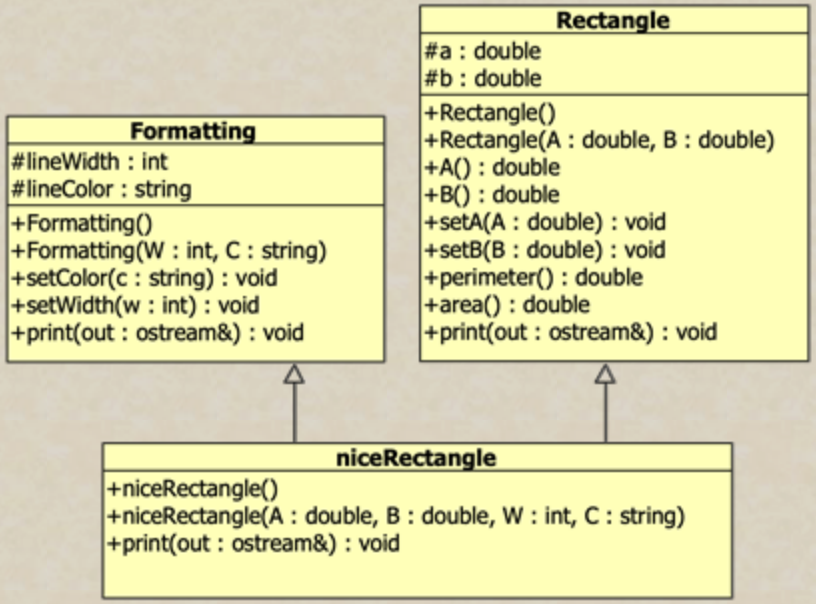
- Μια κλάση κληρονομεί από περισσότερες




```
class NiceRectangle : public Formatting, public Rectangle {
 public:
  NiceRectangle();
  NiceRectangle(double, double, int, string);
  void print(ostream &out) const;
  friend ostream& operator <<(ostream&, const NiceRectangle&);
};
```

Πολλαπλή κληρονομικότητα

- Μια κλάση κληρονομεί από περισσότερες




```
class NiceRectangle : public Formatting, public Rectangle {
 public:
  NiceRectangle();
  NiceRectangle(double, double, int, string);
  void print(ostream &out) const;
  friend ostream& operator <<(ostream&, const NiceRectangle&);
};
```

Πολλαπλή κληρονομικότητα

```
class NiceRectangle : public Formatting, public Rectangle {
 public:
  NiceRectangle(): Rectangle(), Formatting() {}

  NiceRectangle(double A, double B, int W, string C):
                Rectangle(A, B), Formatting(W, C) {}
    void print(ostream &out) const {
      out << "(nr) " << a << " by " << b << ": ar="
          << area() << ", per=" << perimeter()
          << ", linecol=" << lineColor << ", linewidth="
          << lineWidth << endl;
      }

    friend ostream& operator<<(ostream &out,
                               const NiceRectangle &nr) {
      nr.print(out); return out;
    }
};
```

Κλήση των constructors των κλάσεων - γονέων

Από Rectangle

Από Formatting

Πολλαπλή κληρονομικότητα

```
NiceRectangle nr;
nr.print(cout);
```
```
(nr) 0 by 0: ar=0, per=0, linecol=transparent, linewidth=0
```

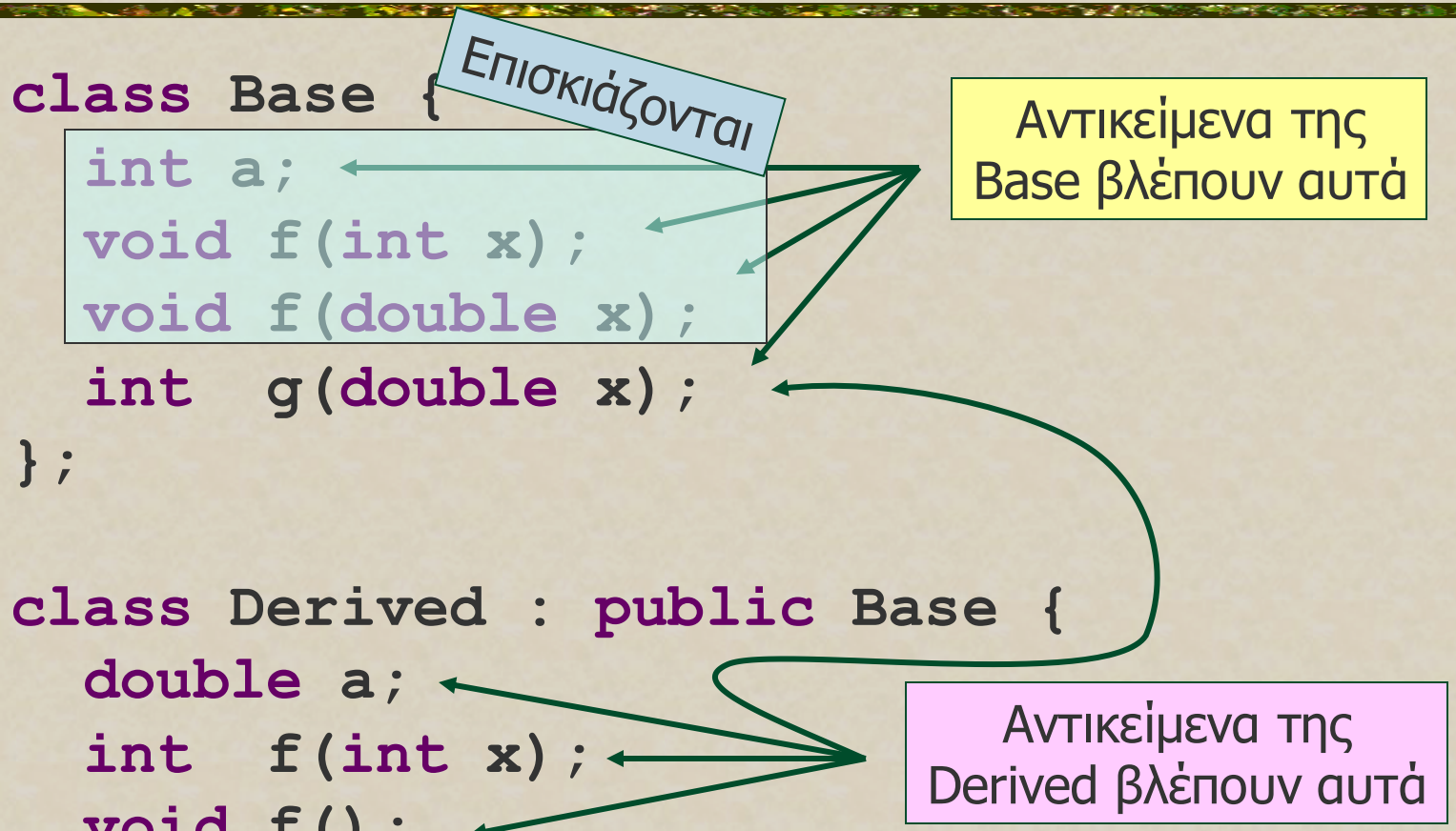
```
NiceRectangle nr2(2, 3, 1, "blue");
nr2.print(cout);
```
```
(nr) 2 by 3: ar=6, per=10, linecol=blue, linewidth=1
```

```
cout << "(nr2) area= " << nr2.area() << endl;
nr2.setA(3);
cout << "(nr2) per=" << nr2.perimeter() << endl;
```
```
(nr2) area= 6
(nr2) per=12
```

```
Rectangle *rr = &nr2;  // ?
rr->print();
```
```
[a=3 b=3 ar=9 per=12]
```

```
Formatting *ff = &nr2; // ?
ff->print();
```
```
[color=blue width=1]
```

Επισκίαση μεθόδων




```
class Base {
  int a;
  void f(int x);
  void f(double x);
  int  g(double x);
};

class Derived : public Base {
  double a;
  int  f(int x);
  void f();
};
```

Επισκίαση μεθόδων

```
class Person {
 protected:
  string name;
 public:
  ...
  void print() const {
    cout << "person " << name << " sleeps" << endl;
  }
};

class Student : public Person {
 private:
  string enrolledIn;
 public:
  void print() const {
    cout << "student " << name
         << " studies " << enrolledIn << "\n";
  }
};
```

Επισκίαση μεθόδων

- Ολόκληρη η κλάση Person

```
class Person {
  public:
    Person() : name("John Doe") {}                          // constructors
    Person(const string &newname) : name(newname) {}

    string Name() const { return name; }                          // getter

    void setName(const string &newname) { name = newname; } // setter

    void print() const {                                        // behaviour
      cout << "person " << name << " sleeps" << endl;
    }

  protected:
    string name;
};
```

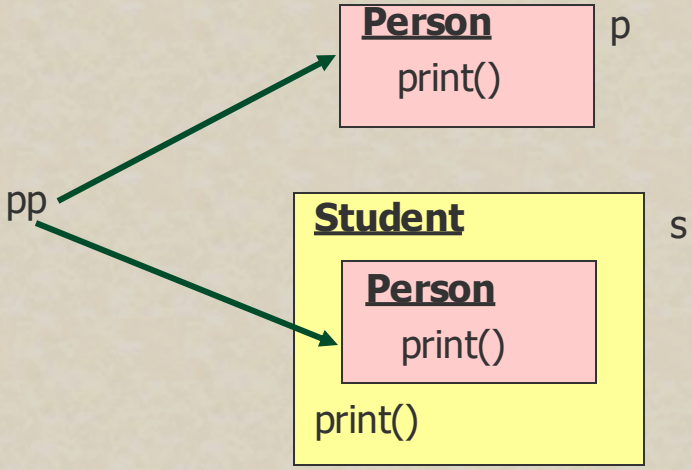
Επισκίαση μεθόδων

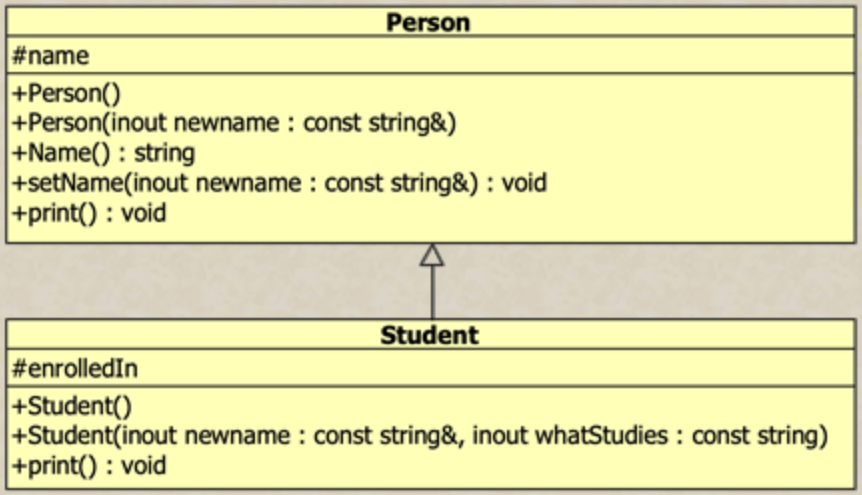
- Ολόκληρη η κλάση Student

```
class Student : public Person {
  public:
    Student() : Person(), enrolledIn("engineering") {}

    Student(const string &newname, const string whatStudies)
      : Person(newname), enrolledIn(whatStudies) {}

    void print() const {
      cout << "student " << name
           << " studies " << enrolledIn << endl;
    }

  protected:
    string enrolledIn;
};
```

Επισκίαση μεθόδων

```
Person p, *pp;
Student s;
p.print();    // person John Doe sleeps
s.print();    // student John Doe studies engineering

pp = &p;
pp->print(); // person John Doe sleeps
pp = &s;
pp->print(); // person John Doe sleeps  (!!)
```



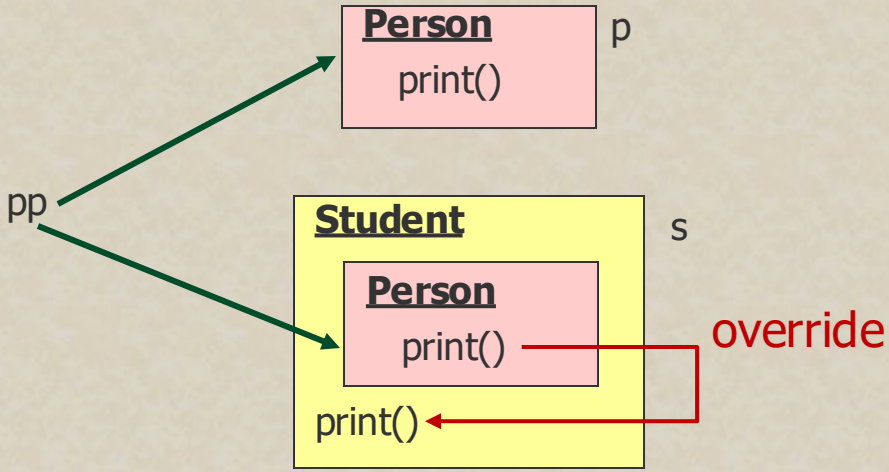


Εικονικές μέθοδοι

```
class Person {
  protected:
    string name;
  public:
    virtual void print() const {
      cout << "person " << name << " sleeps" << endl;
    }
};

class Student : public Person {
  private:
    string enrolledIn;
  public:
    void print() const override {
      cout << "student " << name
           << " studies " << enrolledIn << endl;
    }
};
```

Εικονικές μέθοδοι

```
Person p, *pp;
Student s;
p.print();    // person John Doe sleeps
s.print();    // student John Doe studies engineering

pp = &p;
pp->print(); // person John Doe sleeps
pp = &s;
pp->print(); // student John Doe studies engineering
```



Πολυμορφισμός

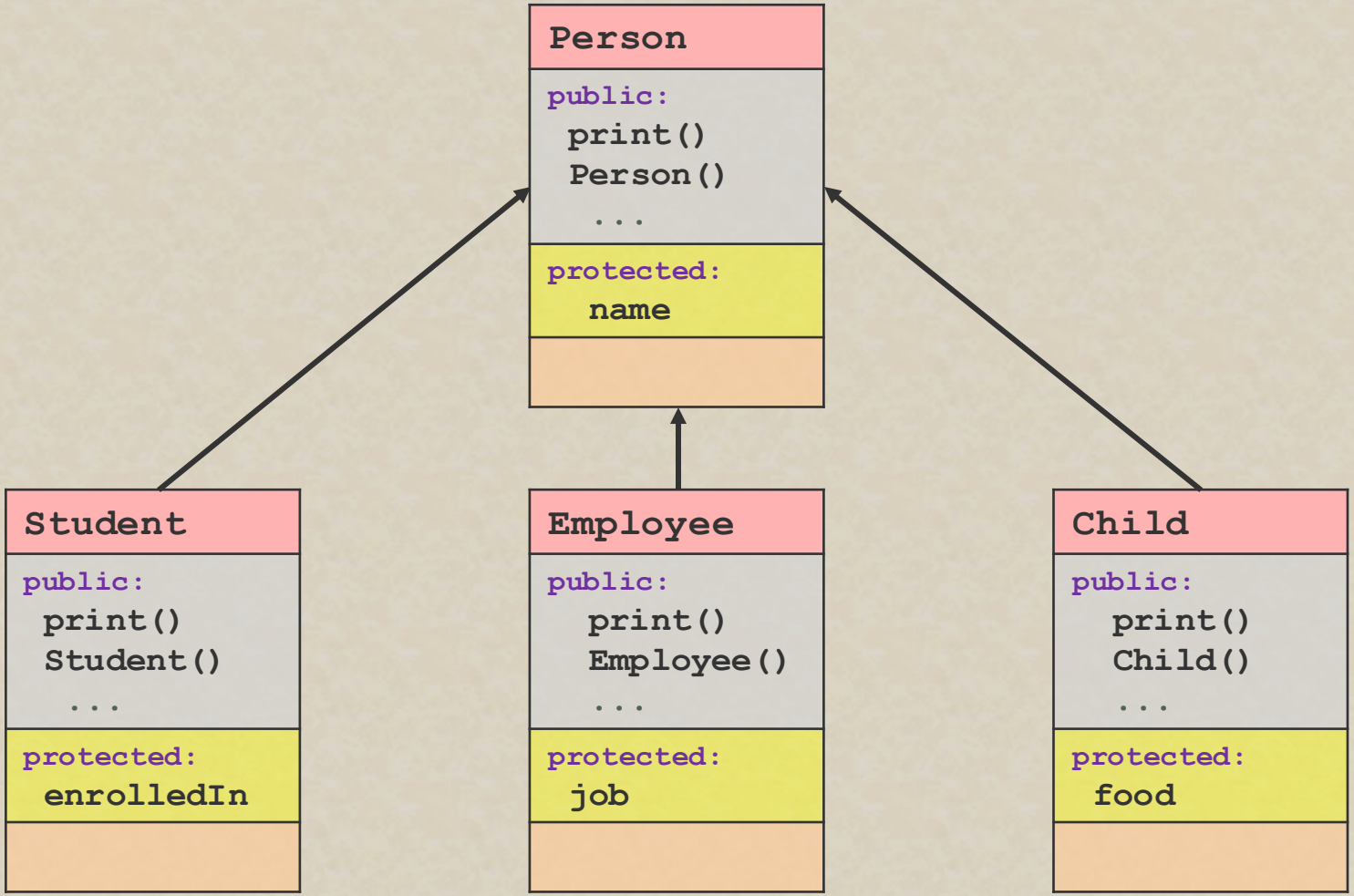
- Η ίδια μέθοδος υλοποιείται διαφορετικά σε διαφορετικές εξειδικεύσεις των αντικειμένων
- Υλοποιείται με χρήση:
  - εικονικών μεθόδων
  - κλήση αυτών μέσω δεικτών ή αναφορών

Αφηρημένες μέθοδοι και κλάσεις

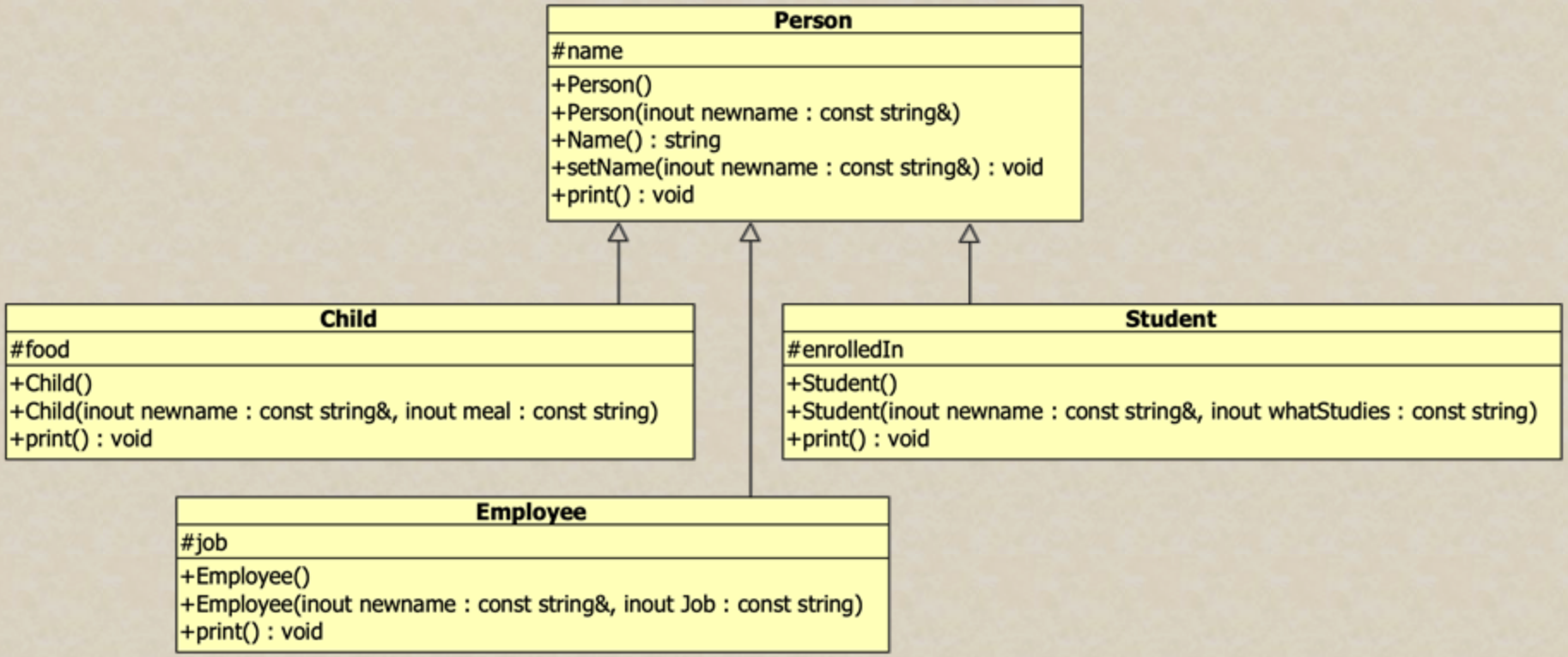
```
class Person {
   ...
   virtual void freeTime() const = 0;
};
```

- Εικονικές (αφηρημένες) μέθοδοι:
  δεν υλοποιούνται στην κλάση Person αλλά οι κλάσεις που την κληρονομούν αναλαμβάνουν την υποχρέωση να τις κάνουν override
- Εικονικές (αφηρημένες) κλάσεις:
  κλάσεις που περιέχουν αφηρημένες μεθόδους

Πολυμορφισμός



Πολυμορφισμός



Πολυμορφισμός

```
class Person {
  virtual void print() const {
    cout << "person " << name << " sleeps" << endl;
  }
};
class Student : public Person {
  void print() const override {
    cout << "student " << name << " studies " << enrolledIn << endl;
  }
};
class Child : public Person {
  void print() const override {
    cout << "child " << name << " eats " << food << endl;
  }
};
class Employee : public Person {
  void print() const override {
    cout << "employee " << name << " works as a ” << job << endl;
  }
};
```

Πολυμορφισμός

```
void allInfo(int n, Person *p[]) {
  for (int i = 0; i < n; i++)
    p[i] -> print();
}

int main() {
  Person *p[] = { new Person, new Student,
                  new Child,  new Employee };
  allInfo(4, p);
}
```

```
person John Doe sleeps
student John Doe studies engineering
child John Doe eats an apple
employee John Doe works as a C++ programmer
```

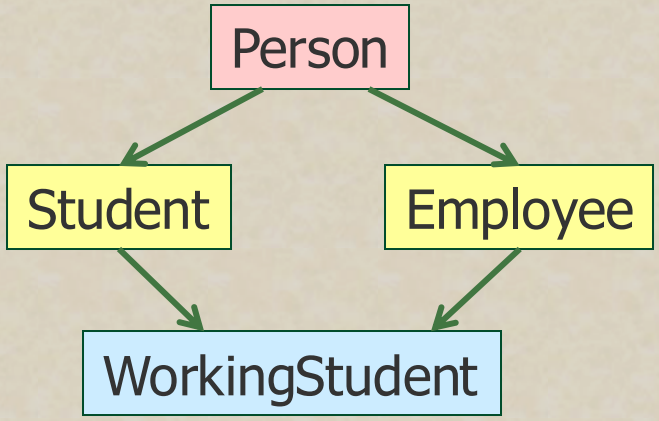
Πολυμορφισμός

- Ολόκληρη η κλάση employee

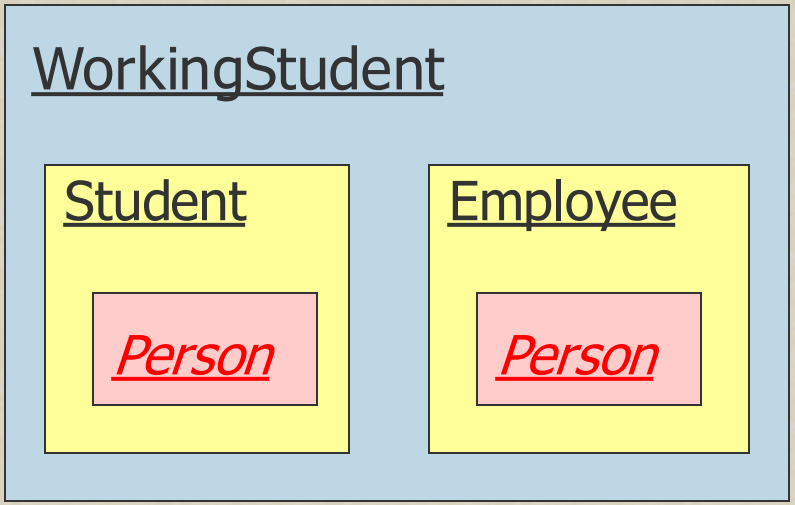
```
class Employee : public Person {
  public:
    Employee() : Person(), job("C++ programmer") {}
    Employee(const string &newname, const string &newjob)
      : Person(newname), job(newjob) {}

    void print() const override {
      cout << "employee " << name << " works as a "
           << job << endl;
    }

  protected:
    string job;
};
```

Εικονικές κλάσεις

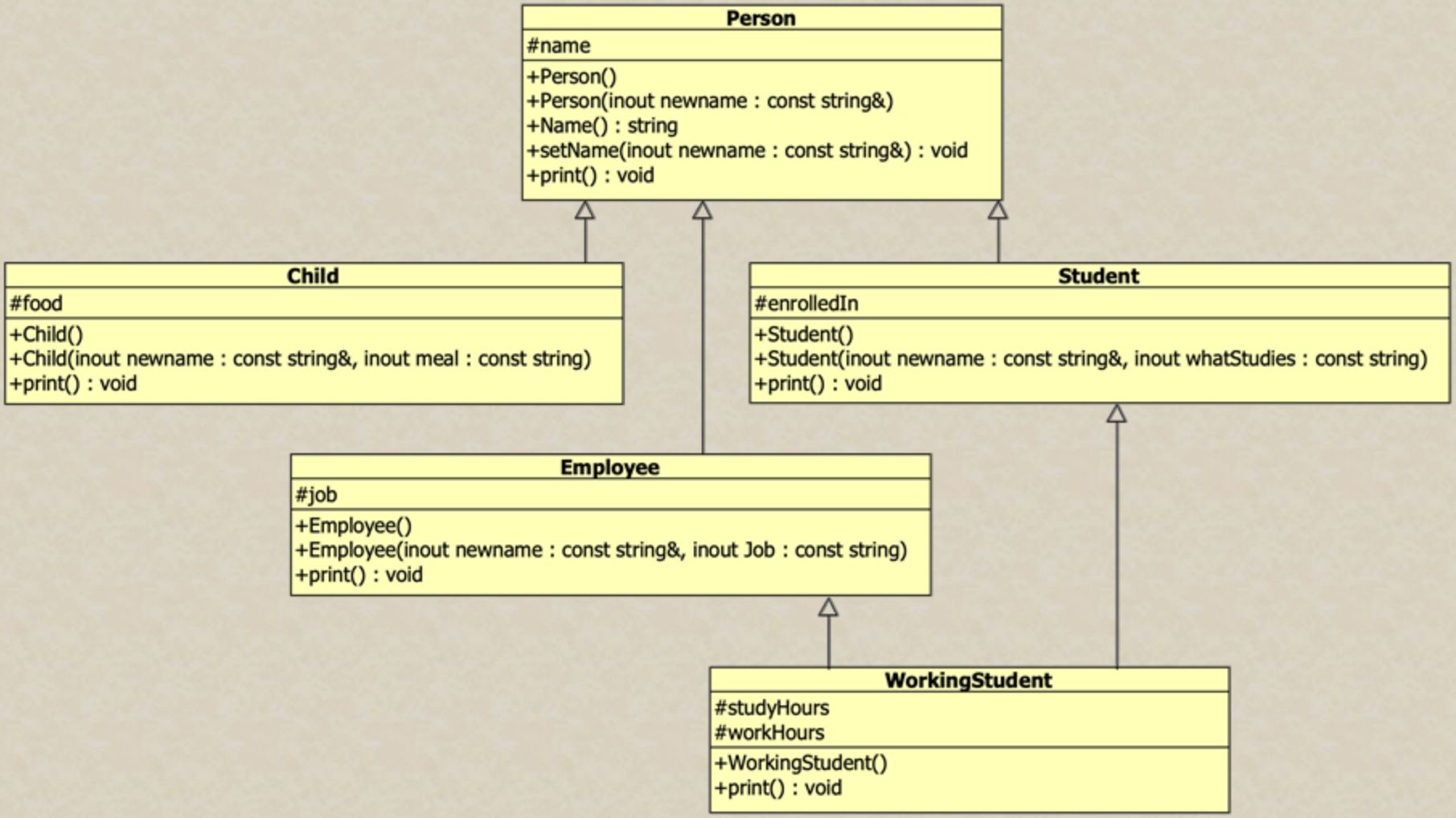
```
class Person { … }

class Student :
  public Person { … }

class Employee :
  public Person { … }

class WorkingStudent :
  public Student,
  public Employee { … }
```




```
Non-static member 'name' found in multiple base-class subobjects of type 'Person':
class WorkingStudent -> class Student -> class Person
class WorkingStudent -> class Employee -> class Person
member found by ambiguous name lookup

Declared in: main.cpp

protected:
basic_string<char, char_traits<char>, allocator<char>> Person::name
```

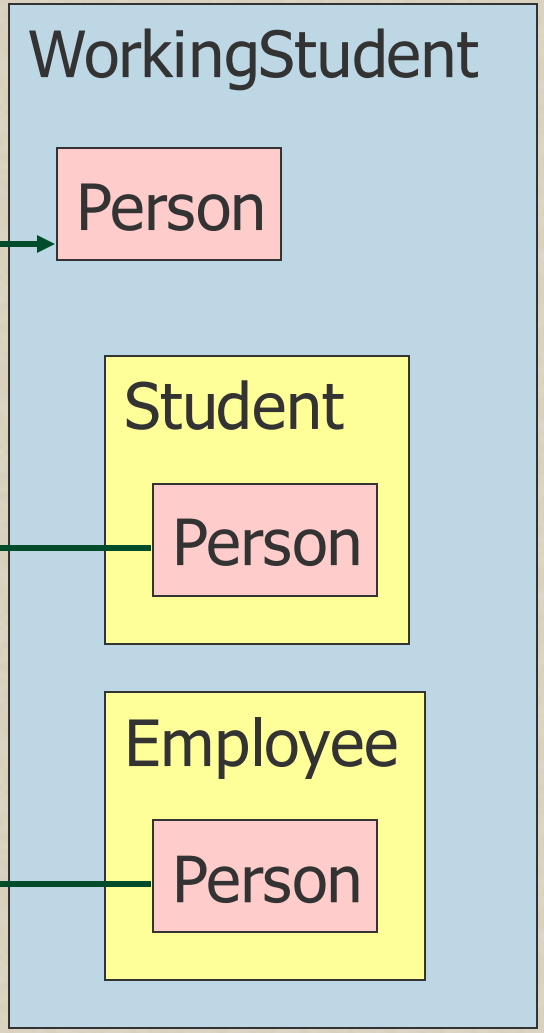


Εικονικές κλάσεις



Εικονικές κλάσεις

```
class Person { … }

class Student :
  virtual public Person { … }

class Employee :
  virtual public Person { … }

class WorkingStudent :
  public Student,
  public Employee { … }
```



virtual: δεν θα δημιουργηθεί στιγμιότυπο μελών δηλωμένων ως virtual

Εικονικές κλάσεις

- Ολόκληρη η κλάση WorkingStudent

```
class WorkingStudent : public Student, public Employee {
  protected:
    int studyHours, workHours;

  public:
    WorkingStudent() : Student(), Employee(),
                       studyHours(4), workHours(6) {}

    void print() const override {
      cout << " working student " << name
           << " studies " << enrolledIn
           << " " << studyHours << " hours per day "
           << " and works as a " << job
           << " " << workHours << " hours per day" << endl;
    }
};
```

Πολυμορφισμός

```
void allInfo(int n, Person *p[]) {
  for (int i = 0; i < n; i++)
    p[i] -> print();
}

int main() {
  Person * p[] = { new Person, new Student, new Child,
                   new Employee, new WorkingStudent };
  allInfo(5, p);
}
```

```
person John Doe sleeps
student John Doe studies engineering
child John Doe eats an apple
employee John Doe works as a C++ programmer
working student John Doe studies engineering 4 hours per
day and works as a C++ programmer 6 hours per day
```

Κλάσεις: τα απολύτως αναγκαία

- Δηλώνω όλα τα πεδία ως private (πάντα!)
- Ορίζω (public) getters και setters, έναν για κάθε πεδίο
- Ορίζω (public) τουλάχιστον έναν constructor που εκτελείται αυτόματα
- Ορίζω (public) όλες τις μεθόδους που θέλω να εκτελούνται εκτός της κλάσης (π.χ. πράξεις, print, κ.λπ.)
- Ορίζω (private) μεθόδους που θα καλούνται μόνο μέσα από την κλάση

Κλάσεις: τα απολύτως αναγκαία

- Ορίζω (public) constructors με παραμέτρους, που με διευκολύνουν στον ορισμό αντικειμένων
- Δεν ορίζω destructors, εκτός αν είμαι σίγουρος για ποιο λόγο είναι αναγκαίοι
- Ορίζω (public) τελεστές που θα χρησιμοποιεί η κλάση. Αν έχω έτοιμη την υλοποίησή τους από κάπου αλλού, ή αν έχει νόημα να τους χρησιμοποιήσω και εκτός της κλάσης, τότε τους δηλώνω friend
- Ορίζω (public) τον τελεστή **<<** για κομψότητα στο χειρισμό εξόδου (προσοχή: friend!)

Κλάσεις: τα απολύτως αναγκαία

- Μεθόδους που δεν θέλω να μεταβάλλουν τίποτε τις δηλώνω const
- Μεθόδους που απλά κάνουν έναν υπολογισμό χωρίς να διαβάζουν/γράφουν πεδία της κλάσης (δηλαδή δέχονται τα απαιτούμενα δεδομένα ως παραμέτρους) τις δηλώνω static
- Αν (και μόνο αν) υπάρχουν πεδία που έχει νόημα να είναι κοινά για όλα τα αντικείμενα της κλάσης, τότε αυτά τα δηλώνω static και ορίζω και static μεθόδους που τα χειρίζονται

Κλάσεις: τα απολύτως αναγκαία

- Χρησιμοποιώ κληρονομικότητα μόνο όταν (μου) είναι φανερό ότι αυτό έχει νόημα. Τα private πεδία και μεθόδους που θέλω να κληρονομηθούν, τα δηλώνω protected
- Χρησιμοποιώ μόνο public κληρονομικότητα, εκτός αν ξέρω γιατί να πράξω διαφορετικά
- Χρησιμοποιώ πολλαπλή κληρονομικότητα μόνο όταν αντιλαμβάνομαι το όφελος. Στην περίπτωση αυτή, χρησιμοποιώ virtual κληρονομικότητα για να επιλύονται διφορούμενα (αν υπάρχουν)

Προσπαθώ να εκτελέσω και να καταλάβω όλα τα παραδείγματα των διαλέξεων. Κατόπιν (και μόνο αφού έχω δουλέψει) ζητάω βοήθεια στο εργαστήριο και στο μάθημα!

Εξαιρέσεις

Χειρισμός εξαιρέσεων

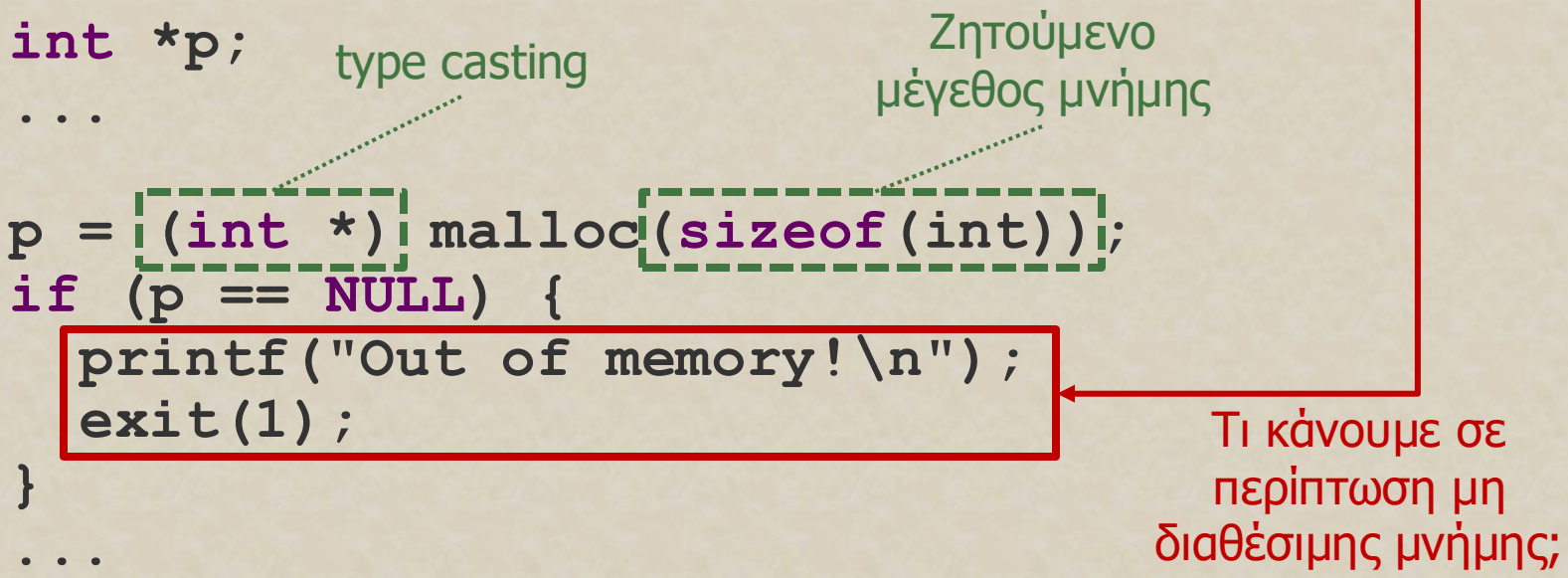
- Τι είναι εξαιρέσεις (exceptions)
  - Κάθε είδους ανωμαλίες ή σφάλματα που προκύπτουν κατά την εκτέλεση του προγράμματος και πρέπει να ξεπεραστούν με ειδικό τρόπο
- Εξαιρέσεις προκαλούνται
  - Από το σύστημα, π.χ. εξάντληση μνήμης
  - Όπως αποφασίζουμε εμείς, σύμφωνα με τη λογική του προγράμματός μας

Χειρισμός εξαιρέσεων

- Χειρισμός εξαιρέσεων στη C++
  - Τι θέλουμε να γίνει όταν προκύψει μια εξαίρεση:
    - τερματισμός του προγράμματος;
    - μεταβολή που αναιρεί την εξαίρεση;
    - άλλο;
- Μηχανισμός: **throw / try - catch**
  - η εξαίρεση προκαλείται με **throw** είτε αυτόματα, είτε ρητά στο πρόγραμμα
  - ανιχνεύεται μέσα σε block εντολών **try**
  - αντιμετωπίζεται μέσα σε block εντολών **catch**

Χειρισμός εξαιρέσεων: τυπικό παράδειγμα

- Αιτούμαστε μνήμη: αν υπάρχει παίρνουμε ένα δείκτη, διαφορετικά παίρνουμε NULL
- Δυναμική δέσμευση σε C (και C++): **malloc**



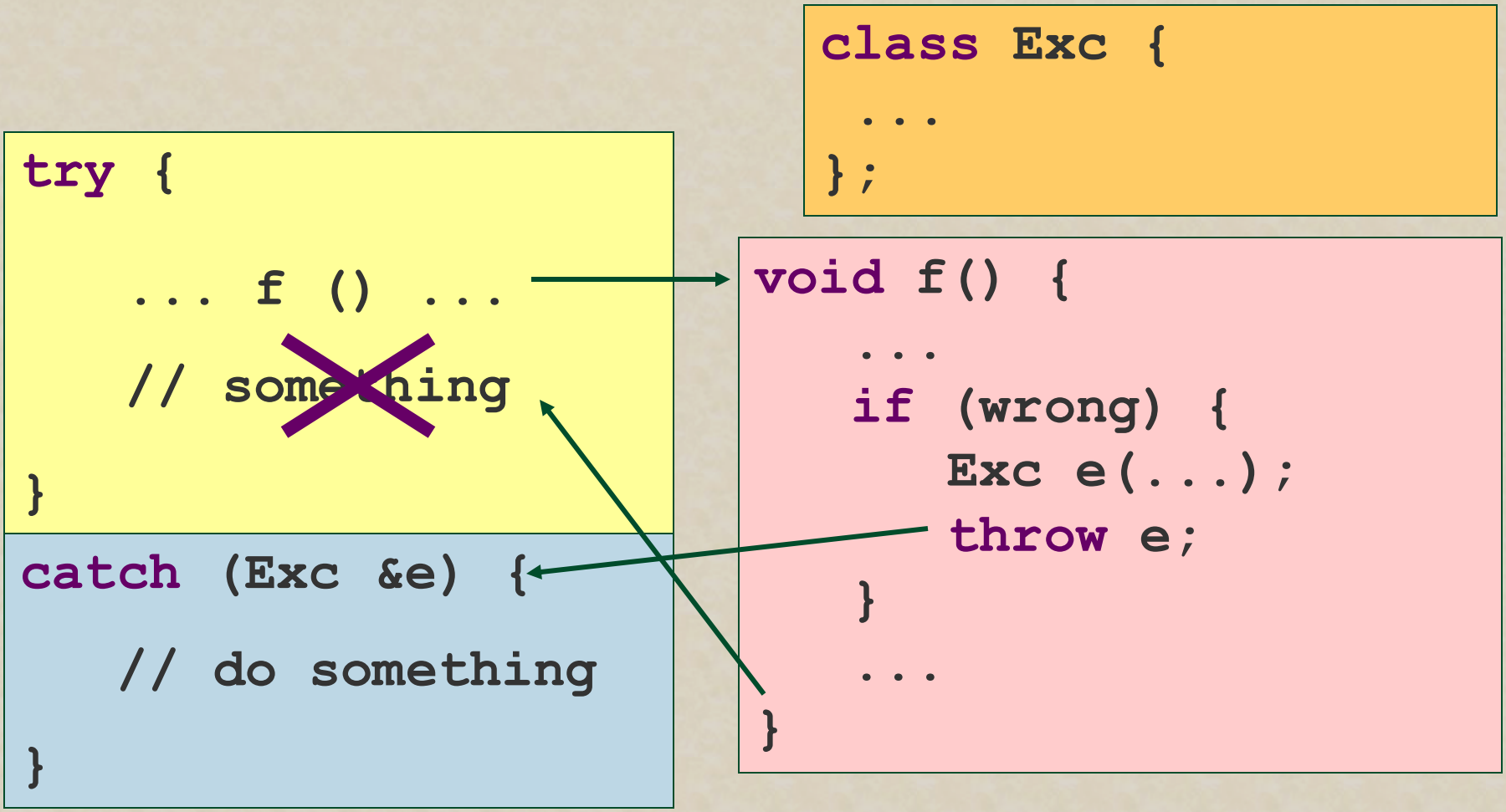
Χειρισμός εξαιρέσεων

- Δυναμική δέσμευση σε C++ (μόνο!)
  - Έλεγχος με ανίχνευση εξαίρεσης (exception) ή με ανίχνευση null, δηλώνοντας nothrow

```
int *p = new(nothrow) int;
if (!p) {
   cout << "Out of memory!";
   exit(1);
}
```

Τι κάνουμε σε περίπτωση μη διαθέσιμης μνήμης;

Χειρισμός εξαιρέσεων



Χειρισμός εξαιρέσεων: παράδειγμα

1. Γνωστή εξαίρεση (ορισμένη στη γλώσσα ή από τον προγραμματιστή)

2. Πρόκληση εξαίρεσης (από τον προγραμματιστή, ενίοτε από τη γλώσσα)

3. Χειρισμός εξαίρεσης (από τον προγραμματιστή)

```
class runtime_error: public exception

double division(double num, double den) {
  if (den) return num/den;
  throw runtime_error("Division by zero!\n");
}

int main() {
  double a = 1, b = 0, c;
  try {
    c = division(a, b);
    cout << a << "/" << b << " = " << c << endl;
  } catch(runtime_error &e) {
    cout << "exception: " << e.what();
  }
}
```

`what()`:
Ορίζεται στην `runtime_error`

Χειρισμός εξαιρέσεων: παράδειγμα

```
class Rectangle {
  private:
    double a, b;
  public:
    Rectangle();
    Rectangle(double, double);
    double A() const;
    double B() const;
    void setA(double);
    void setB(double);
    double perimeter() const;
    double area() const;
    void print(ostream&) const;
};
```

Τι κάνουμε σε περίπτωση αρνητικού μήκους;

Ερώτηση:
Είναι υποχρεωτικό να γίνει με exception η αντιμετώπιση της ανεπιθύμητης αυτής συνθήκης;

Hint: όχι!

Χειρισμός εξαιρέσεων: στρατηγική

- ορισμός κλάσεων εξαιρέσεων
- συγγραφή κώδικα **throw ... try ... catch**

ΠΡΟΚΛΗΣΗ

ΧΕΙΡΙΣΜΟΣ

```
class exc_1: public exception {...}
class exc_2: public exception {...}
...
try {
   ...
   if (error condition 1) throw exc_1()
   if (error condition 2) throw exc_2()
   ...
}
catch(exc_1 &e) { ...do_something 1...}
catch(exc_2 &e) { ...do_something 2...}
```

εκτελείται ο constructor της exc_1

πρόσβαση στο αντικείμενο

Χειρισμός εξαιρέσεων: παραδείγματα

```
class unreal_rectangle : public exception {};

class Rectangle {
  public:
    Rectangle(double A, double B) {
      if (A < 0 || B < 0) throw unreal_rectangle();
      a = A; b = B;
    }
    ...
};

int main() {
  while (true) {
    try {
      int width, height;
      cin >> width >> height;
      Rectangle r(width, height);
      cout << r.area() << endl;
    } catch (unreal_rectangle &e) {
      cout << "wrong width or height!" << endl;
    }
  }
}
```

Πρόκληση εξαίρεσης

Χειρισμός εξαίρεσης

Χειρισμός εξαιρέσεων: παραδείγματα

```
class unreal_rectangle_detailed : public runtime_error {
  public:
    unreal_rectangle_detailed()
      : runtime_error("exception: invalid rectangle") {};
};
```

```
class Rectangle {
  public:
    Rectangle(double A, double B) {
      if (A < 0 || B < 0) throw unreal_rectangle_detailed();
      a = A; b = B;
    }
    ...
};
```

Ορισμός εξαίρεσης από τον προγραμματιστή

```
...
try {
  ...
} catch (unreal_rectangle_detailed &e) {
  cout << e.what() << endl;
}
```

Χειρισμός εξαιρέσεων: παραδείγματα

```
class negative_length_a : public exception {};
class negative_length_b : public exception {};
```

```
class Rectangle {
  public:
    Rectangle(double A, double B) {
      if (A < 0) throw negative_length_a();
      if (B < 0) throw negative_length_b();
      a = A; b = B;
    }
    ...
};
```

Ορισμός εξαιρέσεων από τον προγραμματιστή (εναλλακτικό σενάριο)

```
...
try {
  ...
} catch(negative_length_a &e) {
  cout << "wrong width!" << endl;
} catch(negative_length_b &e) {
  cout << "wrong height!" << endl;
}
```

Χειρισμός εξαιρέσεων

- Δυναμική δέσμευση μνήμης σε C++
  - Έλεγχος με ανίχνευση εξαίρεσης (exception)

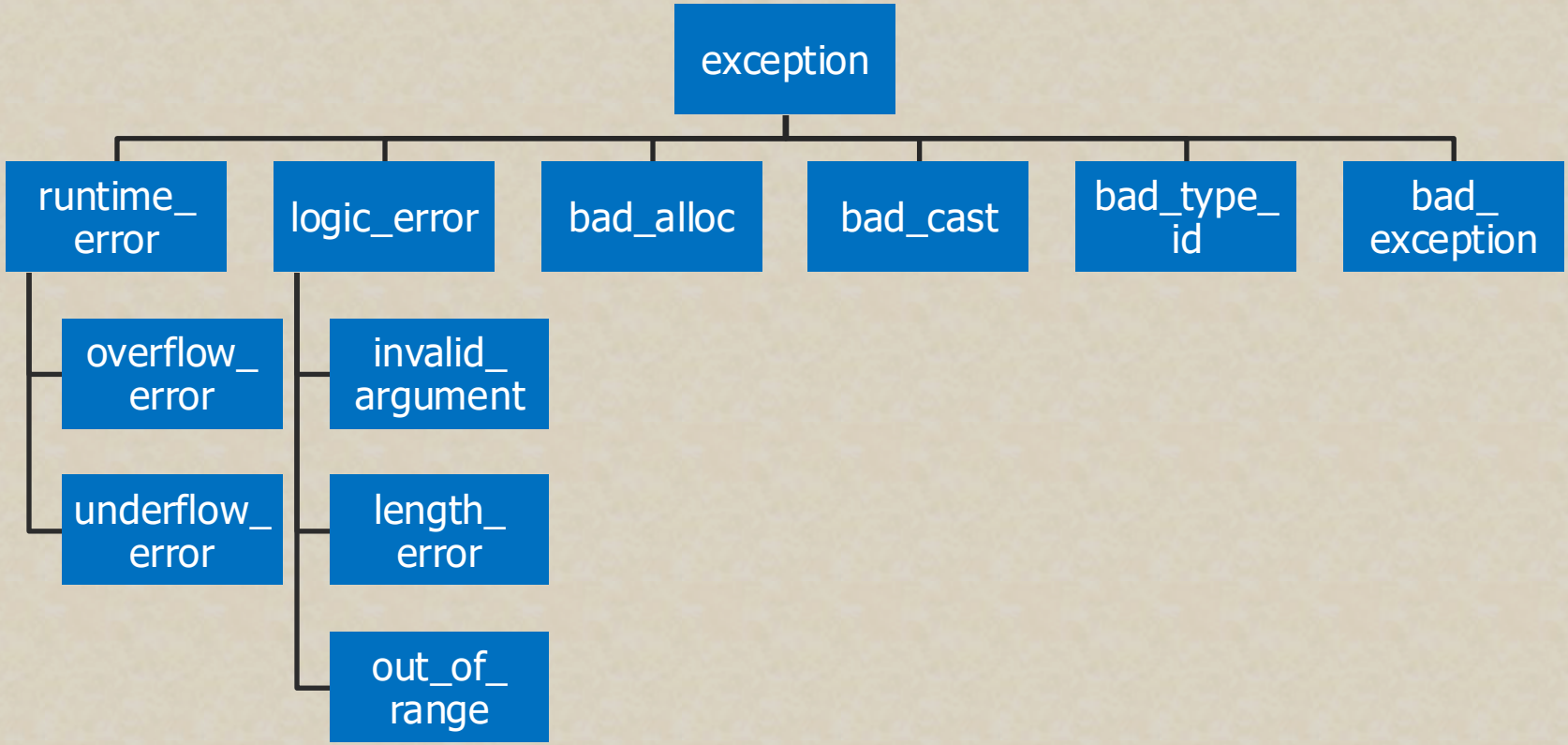
Πρόκληση εξαίρεσης από την γλώσσα

Ορισμός εξαίρεσης (@STL)

```
try {
  int *p = new int[10000000000000000];
} catch(bad_alloc) {
  cout << "Are you serious?!" << endl;
  exit(1);
}
```

Χειρισμός εξαίρεσης

Εξαιρέσεις στη C++



Χειρισμός εξαιρέσεων

```
class stack {
  ...
  void push (int x) {
    a[top++] = x;
  }
  int pop () {
    return a[--top];
  }
  ...
  int mysize;
  int top;
  int *a;
};
```

Αν η στοίβα είναι γεμάτη:
top == mysize

Αν η στοίβα είναι άδεια:
top == 0

Χειρισμός εξαιρέσεων

```
#include <exception>
using namespace std;
class full_stack : public exception {};
class stack {
  ...
  stack(int n) : top(0), mysize(n) {
  try {
    data = new int[mysize];
  } catch(bad_alloc) {
    mysize = 5; data = new int[mysize];
    cerr << "not enough memory, size set to 5"
         << endl;
  }
};
```

το new κάνει throw

Δεν είναι υποχρεωτικό να διακόπτουμε το πρόγραμμα...

Χειρισμός εξαιρέσεων

```
#include <exception>
using namespace std;
class empty_stack : public exception {};
class stack {
  ...
  int pop() {
    if (top > 0)
      return a[--top];
    else
      throw empty_stack();
  }
};
```

Χειρισμός εξαιρέσεων

```
#include <exception>
using namespace std;
class full_stack : public exception {};
class stack {
   ...
  void push(int x) {
    if (top < mysize)
      a[top++] = x;
    else
      throw full_stack();
  }
};
```

Πιθανή αντιμετώπιση:

```
catch (full_stack) {
  myStack.pop();
  myStack.push(data);
}
```

Χειρισμός εξαιρέσεων

```
int main() {
  try {
    ...
    if (...) throw 42;
    if (...) throw "hello";
    if (...) throw out_of_memory();
    ...
  }
  catch (int n)                { ... }
  catch (const char *s)        { ... }
  catch (out_of_memory &e)     { ... }
  catch (exception &e)         { ... }
};
```

Πολλαπλοί handlers, δοκιμάζονται με τη σειρά

Χειρισμός εξαιρέσεων

```
int main() {
  int choice;
  do {
    cin >> choice;
    try {
      if (choice == 1) throw 1.0;
      if (choice == 2) throw 2;
      if (choice == 3) throw bad_alloc();
      if (choice == 0) throw 'e';
    }
    catch (const double s) { cout << "your choice is " << s; }
    catch (const int s) { cout << "hello, your choice is " << s; }
    catch (const char s) { cout << s << " means bye!"; break; }
    catch (bad_alloc &e) { cout << "not really bad_alloc..."; }
  } while (true);
}
```

Κακή πρακτική!

Χειρισμός εξαιρέσεων: αρχεία

```
ifstream inFile;
char c;
try {
  inFile.open("mydata.txt");
  if (inFile.fail()) throw "cannot open 'mydata.txt'";
  inFile >> c;
  while (inFile.good()) {
    cout << c;
    inFile >> c;
    if (inFile.fail()) throw "file error";
  }
} catch(const char *e) {
  cout << "file error: " << e << endl;
  // ...
}
```

Χειρισμός εξαιρέσεων

```
class red_ball: public exception {};
class green_ball: public exception {};

void children_playing() {
  // may throw red_ball or green_ball
  // or all sorts of other things (they're only children)
  ...
}

void little_field() {
  try {
    children_playing();
  } catch (red_ball &e) {
    cout << "Caught the red one!" << endl;
  }
  cout << "Little finished" << endl;
}
```

Χειρισμός εξαιρέσεων

```
void bigger_field() {
  try {
    little_field();
  } catch (green_ball &e) {
    cout << "Caught the green one!" << endl;
  }
  cout << "Bigger finished" << endl;
}
```