Quick applicatio

Conclusions 000

1/36

Object Oriented Programming in C

Radu Grigoras radu.grigoras10@gmail.com

University of Hamburg Faculty of Mathematics, Informatics and Natural Sciences Department of Informatics

Seminar "Effiziente Programmierung in C", December, 2012

Introduction and motivation •00000	OOP features and concepts	Quick application	Conclusions
Can it be done?			

• What is OOP?

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
● OO OOO			
Can it be done?			

- What is OOP?
 - programming paradigm

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
00000			
Can it be done?			

- What is OOP?
 - programming paradigm
 - sets of common features (attributes)

Introduction and motivation •00000	OOP features and concepts	Quick application	Conclusions
Can it be done?			

- What is OOP?
 - programming paradigm
 - sets of common features (attributes) =>classes

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
00000			
Can it be done?			

- What is OOP?
 - programming paradigm
 - sets of common features (attributes) =>classes
 - particular instances of classes

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
00000			
Can it be done?			

- What is OOP?
 - programming paradigm
 - sets of common features (attributes) =>classes
 - particular instances of classes => objects

Introduction and motivation •00000	OOP features and concepts	Quick application	Conclusions
Can it be done?			

- What is OOP?
 - programming paradigm
 - sets of common features (attributes) =>classes
 - particular instances of classes => objects
 - manipulating objects

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
• 00 000			
Can it be done?			

- What is OOP?
 - programming paradigm
 - sets of common features (attributes) =>classes
 - particular instances of classes => objects
 - manipulating objects => methods

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
00000			
Can it be done?			

• Can OOP be achieved using only ANSI-C code?

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
00000	0000000000		
Can it be done?			

• Can OOP be achieved using only ANSI-C code?

- yes.

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
00000			
Can it be done?			

- Can OOP be achieved using only ANSI-C code?
 - yes.
 - paradigm vs. language feature

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
0.0000	0000000000		000
Can it be done?			

- Can OOP be achieved using only ANSI-C code?
 - yes.
 - paradigm vs. language feature
 - OO-languages (C++, Java, Python etc.)

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
00000			
Can it be done?			

- Can OOP be achieved using only ANSI-C code?
 - yes.
 - paradigm vs. language feature
 - OO-languages (C++, Java, Python etc.) offer syntactic sugar to achieve OO-code

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
00000			
Can it be done?			

- Can OOP be achieved using only ANSI-C code?
 - yes.
 - paradigm vs. language feature
 - OO-languages (C++, Java, Python etc.) offer syntactic sugar to achieve OO-code

イロト 不得下 イヨト イヨト 二日

3/36

C++ code

object->method(some_args);

Introduction and motivation	OOP features and concepts	Quick application	Conclusions 000
Can it be done?			

- Can OOP be achieved using only ANSI-C code?
 - yes.
 - paradigm vs. language feature
 - OO-languages (C++, Java, Python etc.) offer syntactic sugar to achieve OO-code

C++ code

object->method(some_args);

C code

method(object, some_args);

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
Can it be done?			

• How can it be done?

<□ > < □ > < □ > < ⊇ > < ⊇ > < ⊇ > 三 の Q (~ 4/36

Introduction and motivation	OOP features and concepts	Quick application	Conclusions 000
Can it be done?			

- How can it be done?
 - with structs,

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
00.000	0000000000		000
Can it be done?			

- How can it be done?
 - with structs, pointers

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
000000			
Can it be done?			

- How can it be done?
 - with structs, pointers and other wonderful things

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
Why do it?			

• What is OOP good for?

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
000000			
Why do it?			

- What is OOP good for?
 - data representation and functionality separated from usage

Introduction and motivation	OOP features and concepts	Quick application	Conclusions 000
Why do it?			

- What is OOP good for?
 - data representation and functionality separated from usage
 - divide and conquer

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
000000	0000000000		
Why do it?			

- What is OOP good for?
 - data representation and functionality separated from usage
 - divide and conquer
 - re-use of code

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
Why do it?			

- What is OOP good for?
 - data representation and functionality separated from usage
 - divide and conquer
 - re-use of code
 - enhanced code readibility

Introduction and motivation	OOP features and concepts	Quick application	Conclusions 000
Why do it?			

• Why should I use C in my program?

Introduction and motivation	OOP features and concepts	Quick application	Conclusions 000
Why do it?			

・ロト ・ 日 ・ ・ ヨ ・ ・ ヨ ・

э

6/36

- Why should I use C in my program?
 - mostly because it's faster

Introduction and motivation	OOP features and concepts	Quick application	Conclusions 000
Why do it?			

- Why should I use C in my program?
 - mostly because it's faster
 - environment where C++ or other compilers not available

Introduction and motivation	OOP features and concepts	Quick application	Conclusions 000
Why do it?			

- Why should I use C in my program?
 - mostly because it's faster
 - environment where C++ or other compilers not available
 - you just like it.

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
00000			
Why do it?			

• Are there any disadvantages with this approach?

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
00006	00000000000		000
Why do it?			

• Are there any disadvantages with this approach? - rather complex code

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
000000			
Why do it?			

- Are there any disadvantages with this approach?
 - rather complex code
 - possible loss of type safety

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
000000			
Why do it?			

- Are there any disadvantages with this approach?
 - rather complex code
 - possible loss of type safety
 - programmer time

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
00000			
Why do it?			

- Are there any disadvantages with this approach?
 - rather complex code
 - possible loss of type safety
 - programmer time
 - error prone

Introduction and motivation ○○○○○●	OOP features and concepts	Quick application	Conclusions
Why do it?			

- Are there any disadvantages with this approach?
 - rather complex code
 - possible loss of type safety
 - programmer time
 - error prone
 - manual memory management

ntroduction	and	motivation	0
			•

Quick applicati

Conclusions 000

Classes, objects, methods, constructors and destructors

Listing 1: C++ class example

```
//includes and stuff
2
  class Rectangle {
3
  private:
4
     int x, y;
5
   int width;
6
   int height;
7
  public:
8
   //getters, setters, if
         needed
     void draw();
9
10 };
  void Rectangle::draw() {
11
12
     std::cout << "Just drew a</pre>
         nice " << width << " by</pre>
         " << height << "
         rectangle at position
         (" << x << ", " << y <<
         ")!".
13 }
```

Listing 2: C class example

```
//includes and stuff
  typedef struct Rectangle {
3
    int x,y;
4
    int width;
5
    int height;
 } Rectangle;
6
  void draw(Rectangle* obj) {
7
    printf("I just drew a nice
8
        %d by %d rectangle at
        position (%d, %d)!",
        obj->width,
        obj->height, obj->x,
        obj->v);
9 }
```

イロト イポト イヨト イヨト

- 3

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
000000	000000000		000
Classes all'este methode constructions on	d destructions		



Listing 4: C object example

1 //pretend that everything we already wrote is here 2 //and create an object where needed 3 Rectangle* r = (Rectangle*)malloc(sizeof Rectangle); Introduction and motivation 000000 Quick application

Conclusions 000

Classes, objects, methods, constructors and destructors

```
Listing 5: C++ methods example
 //again, previously defined
      stuff is here, even if
      you cannot see it!
  void Rectangle::draw() {
2
3
    std::cout << "Just drew a</pre>
        nice " << width << " by</pre>
        " << height << "
        rectangle at position
        (" << x << ", " << y <<
        ")!":
4 }
5 //create a Rectangle object
6 Rectangle* r = new
      Rectangle();
7 //then just call one if its
      methods
8 r->draw():
```

Listing 6: C methods example

```
//again, previously defined
      stuff is here, even if
      you cannot see it!
  void draw(Rectangle* obj) {
    printf("I just drew a nice
3
        %d by %d rectangle at
        position (%d, %d)!",
        obj->width,
        obj->height, obj->x,
        obj->y);
4
5 //create a Rectangle object
6
  Rectangle* r =
      (Rectangle*)malloc(sizeof
      *Rectangle);
  //then just call a function
7
      that receives it as a
      param
8
  draw(r);
```

◆□ > ◆□ > ◆豆 > ◆豆 > ・豆 - のへの

Introduction	and	motivation

Quick application

Conclusions 000

Classes, objects, methods, constructors and destructors

```
Listing 7: C++ constructor example
```

```
1 //previously defined stuff is
      here. as usual
 Rectangle::Rectangle (int
2
      initx, int inity, int
      initw, int inith) {
3
      x = initx;
4
      y = inity;
5
      width = initw:
6
      height = inith;
7
  }
8 //and this is how you would
      use it
9 Rectangle* r = new
      Rectangle(1,2,3,4);
```

Listing 8: C constructor example

```
//previously defined stuff is
      here, as usual
  Rectangle* Rectangle_init(int
       initx, int inity, int
       initw. int inith) {
3
    struct Rectangle* obj =
         malloc(sizeof *obi):
4
    obj ->x = initx;
5
    obj -> y = inity;
    obj->width = initw;
6
7
    obj->height = inith;
8
9
    return obj;
10 }
11 //and this is how you would
      use it
12 Rectangle* r =
       Rectangle_init(1,2,3,4);
```

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
	000000000		
	d destaurations		

About destructors in C++

 \sim Rectangle();

-implicitly defined and called when the object is no longer needed

-can be defined explicitly and manually called if needed

Quick application

Conclusions 000

Classes, objects, methods, constructors and destructors

About destructors in C++

 \sim Rectangle();

-implicitly defined and called when the object is no longer needed

-can be defined explicitly and manually called if needed

About destructors in C

there is no automatic memory management in C
you should use free() or wrap your own function around it
manually call it when needed

Introduction and motivation	OOP features and concepts	Quick application	Conclusions
Encapsulation			

- object data is contained and hidden inside of the object
- acces to data is restricted to members of that class or other particular classes
- organising code so that operations on an object type are close to the definition of that type
- lowers the possibility of a user messing up
- reduces the amount of details needed to know when trying to use a type
- provides decoupling: usage is separated from implementation

Encapsulation

OOP features and concepts ○○○○○●○○○○

Quick application

Conclusions 000

About encapsulation in C++

- offers some syntatic sugar to help achieve encapsulation

- public, protected, private

- this is checked by the compiler (at compile time)

- you can stab the compiler in the back and do what you want to the code at run time anyway Introduction and motivation 000000 OOP features and concepts ○○○○○●○○○○

Quick application

Conclusions 000

Encapsulation

About encapsulation in C++

- offers some syntatic sugar to help achieve encapsulation

- public, protected, private

- this is checked by the compiler (at compile time)

- you can stab the compiler in the back and do what you want to the code at run time anyway

About encapsulation in C

- C does not offer the same syntatic sugar
- use naming conventions to help associate types with their methods
- integrate functions into structs using function pointers
- private variables vs. private methods

- also keep in mind that pointers to structs can be used without knowledge of the struct declaration Inheritance

- captures the "is-a" relationship

- a pointer to a derived class is type-compatible with a pointer to its base class

- i.e. Rectangle "is-a" Shape
- Rectangle inherits properties from Shape
- this allows code re-use and a better structure for your program

Introduction	and	

Quick applicatio

Conclusions 000

Inheritance

```
Listing 9: C++ inheritance exam-
                                       Listing 10: C inheritance example
  ple
                                       struct Shape {
  class Shape {
                                     2
                                           /* base class members */
2
    /* Shape class members */
                                     3
                                       };
3
  };
                                     4
                                       struct Rectangle {
  class Rectangle : public
4
                                     5
                                           struct Shape super;
      Shape {
                                           /* derived class members
                                     6
5
    /* Rectangle class members
                                                */
        */
                                     7
                                       };
6 };
```

Introduction and motivation 000000	OOP features and concepts	Quick application	Conclusions 000
Polymorphism			

- allows values of different data types to be handled using an uniform interface
- a polymorphic function can be evaluated or applied to values of different types
- polymoprhism takes advantage of inheritance

Introduction and motivation	OOP features and concepts	Quick application	Conclusions 000
Polymorphicm			

Listing 11: C++ polymorphism example

```
class Shape { //abstract interface
  public:
 2
 3
   virtual void draw() = 0; // pure virtual function
 4
  };
 5 class Rectangle : public Shape { //inheritance
6 //other stuff here too of course
 7
  public:
8
   virtual void draw(); //implement this along the way
9 };
10 //some function that handles a shape polymorphically
11 void handleShape(Shape* s) {
12 s->draw(); //then do something to the shape
13 }
14 //usage
15 Shape* shape;
16 shape = new Rectangle();
17 handleShape(s);
```

Introduction and motivation	OOP features and concepts	Quick application	Conclusions

Listing 12: C++ OOP full example

```
1 #include <iostream>
2
3 /* Shape abstract interface */
4
5 class Shape {
   public:
7   virtual void draw() = 0;
8   virtual void moveTo(int newx, int newy) = 0;
9 };
```

Introduction and motivation	OOP features and concepts	Quick application	Conclusions

```
10
  /* Rectangle class */
11
12 class Rectangle : public Shape {
13 private:
14
    int x, y;
15
  int width;
16
   int height;
17 public:
18
    Rectangle (int initx, int inity, int initw, int inith);
19
    int getX() { return this->x; }
20
    int getY() { return this->y; }
    int getWidth() { return this->width; }
21
22
    int getHeight() { return this->height; }
    void setX(int newx) { this->x = newx: }
23
24
    void setY(int newy) { this->y = newy; }
    void setWidth(int neww) { this->width = neww; }
25
26
    void setHeight(int newh) { this->height = newh; }
27
   virtual void draw():
28
    virtual void moveTo(int newx, int newy);
29 };
```

Introduction and motivation	OOP features and concepts	Quick application	Conclusions

```
30 Rectangle::Rectangle(int initx, int inity, int initw, int
      inith) {
31
    x = initx:
32
   y = inity;
33
   width = initw:
34
    height = inith;
35 }
36
37
  void Rectangle::draw() {
38
     std::cout << "Just drew a nice " << width</pre>
39
       << " by " << height
40
       << " rectangle at position (" << x
       << ", " << y << ")!" << std::endl;
41
42 }
43
44 void Rectangle::moveTo(int newx, int newy) {
45
     x = newx;
46
    y = newy;
47
   std::cout << "Moving your rectangle to (" << x</pre>
     << ", " << y << ")!" << std::endl;
48
49 }
```

Introduction and motivation 000000	OOP features and concepts	Quick application	Conclusions

```
50 /* Circle class */
51 class Circle : public Shape {
52 private:
53
  int x,y;
54
  int radius;
55 public:
56
  Circle(int initx, int inity, int initr);
  virtual void draw();
57
58
   virtual void moveTo(int newx, int newy);
59
   int getX() { return this->x; }
   int getY() { return this->y; }
60
    int getRadius() { return this->radius; }
61
62
   void setX(int newx) { this->x = newx; }
  void setY(int newy) { this->y = newy; }
63
64
    void setRadius(int newr) { this->radius = newr; }
65 };
```

Introduction and motivation	OOP features and concepts	Quick application	Conclusions 000

```
66 Circle::Circle(int initx, int inity, int initr) {
67
   x = initx:
   y = inity;
68
   radius = initr;
69
70 }
71
72 void Circle::draw() {
73
  std::cout << "Just drew a perfect circle of radius "</pre>
74
      << radius << " at position ("
75
     << x << ", " << y << ")!" << std::endl;
76 }
77
78 void Circle::moveTo(int newx, int newy) {
79
   x = newx:
80
    y = newy;
81
  std::cout << "Moving your circle to (" << x</pre>
82
     << ", " << y << ")!" << std::endl;
83 }
```

Introduction and motivation	OOP features and concepts	Quick application	Conclusions 000

```
84 /* A function that uses a Shape polymorphically */
85
86 void handleShape(Shape* s) {
87 std::cout << "Bad shape! Go to the corner!" << std::endl;
88 s->moveTo(0,0);
89 }
```

Introduction and motivation	OOP features and concepts	Quick application	Conclusions

```
90 int main() {
91
      /* using shapes polymorphically */
92
93
      Shape * shapes[2];
      shapes[0] = new Rectangle(20, 12, 123, 321);
94
95
      shapes[1] = new Circle(21, 12, 2012);
96
97
      for (int i = 0; i < 2; ++i) {
        shapes[i]->draw();
98
99
        handleShape(shapes[i]);
100
      }
101
102
      /* access a specific class function */
103
104
      Rectangle* r = new Rectangle(1, 2, 3, 4);
105
      r \rightarrow setWidth(5):
106
      r \rightarrow draw();
107
108
      return 0:
109 }
```

Introduction and motivation 000000	OOP features and concepts	Quick application	Conclusions 000

Listing 13: C OOP full example

```
#include <stdio.h>
  #include <stdlib.h>
 2
 3
  #include <assert.h>
 4
  /* Shape abstract interface */
  struct Shape {
 6
    struct ShapeFuncTable *funcTable;
 7
  };
8
  struct ShapeFuncTable {
9
   void (*draw) (struct Shape* obj);
  void (*moveTo) (struct Shape* obj, int newx, int newy);
10
   void (*destructor_) (struct Shape *obj);
11
12 };
13 struct Shape *Shape_init() { assert(0); }
14 void Shape_destroy(struct Shape *obj) { }
```

Introduction and motivation	OOP features and concepts	Quick application	Conclusions 000

```
15 /* Rectangle class */
16
17
  struct Rectangle {
18
    struct Shape super;
19
20
   int x, y;
21
  int width;
22
   int height;
23 };
24
25 void Rectangle_draw(struct Shape* obj) {
26
    struct Rectangle* rdata = (struct Rectangle*) obj;
27
28
    printf("I just drew a nice %d by %d rectangle at position
        (%d, %d)! \n",
29
      rdata->width, rdata->height, rdata->x, rdata->y);
30 }
```

Introduction and motivation	OOP features and concepts	Quick application	Conclusions 000

```
31 void Rectangle_moveTo(struct Shape* obj, int newx, int
      newy) {
32
    struct Rectangle * rdata = (struct Rectangle*) obj;
33
34
    rdata - > x = newx;
35
    rdata->y = newy;
36
37
    printf("Moving your rectangle to (%d, %d)\n",
38
       rdata->x, rdata->y);
39 }
40
41
  void Rectangle_setWidth(struct Shape* obj, int neww) {
42
    struct Rectangle * rdata = (struct Rectangle*) obj;
43
44
    rdata->width = neww;
45 }
```

Introduction	and	motivation

Quick application

Conclusions

29 / 36

```
46 void Rectangle_destroy(struct Shape *obj) {
47
    Shape_destroy(obj);
48
    free(obj);
49 }
50 struct RectangleFuncTable {
51
    struct ShapeFuncTable super;
   void (*setWidth) (struct Shape* obj , int neww);
52
53 } rectangleFuncTable = { {
54
      Rectangle_draw,
55
      Rectangle_moveTo,
56
      Rectangle_destroy },
57
   Rectangle_setWidth
58 };
59 struct Shape* Rectangle_init(int initx, int inity, int
      initw. int inith) {
    struct Rectangle* obj = (struct Rectangle*)
60
        malloc(sizeof(struct Rectangle));
61
    obj->super.funcTable = (struct ShapeFuncTable*)
        &rectangleFuncTable;
62
    obj ->x = initx;
63
    obj->y = inity;
64
    obj->width = initw;
65
    obj->height = inith;
66
    return (struct Shape*)obj;
                                           67 }
```

ntroduction and motivation	OOP features and concepts	Quick application	Conclusions 000

```
68 /* Circle class */
69 struct Circle {
70 struct Shape super;
71
   int x, y;
72
   int radius;
73 };
74 void Circle_draw(struct Shape* obj) {
75
  struct Circle* cdata = (struct Circle*) obj;
76 printf("Just drew a perfect circle of radius %d at
         position (%d, %d)! \n",
77
       cdata->radius. cdata->x. cdata->v):
78 }
79
80 void Circle_moveTo(struct Shape* obj, int newx, int newy) {
81
     struct Circle* cdata = (struct Circle*) obj;
82
    cdata - > x = newx:
83
     cdata->y = newy;
84
   printf("Moving your circle to (%d, %d)\n",
85
       cdata->x, cdata->y);
86 }
```

Introduction and motivation	OOP features and concepts	Quick application	Conclusions

```
87
   void Circle_destroy(struct Shape *obj) {
88
     Shape_destroy(obj);
89
     free(obj);
90 }
91 struct CircleFuncTable {
92
     struct ShapeFuncTable super;
93 } circleFuncTable = { {
94
       Circle draw.
   Circle_moveTo,
95
96
       Circle_destroy }
97 };
98 struct Shape* Circle_init(int initx, int inity, int initr) {
99
     struct Circle* obj = (struct Circle*)
         malloc(sizeof(struct Circle)):
100
     obj->super.funcTable = (struct ShapeFuncTable*)
         &circleFuncTable:
101
     obj ->x = initx;
102
     obj ->y = inity;
103
     obj->radius = initr;
     return (struct Shape*)obj;
104
105 }
```

Introduction and motivation	OOP features and concepts	Quick application	Conclusions 000

```
106 #define Shape_DRAW(obj) (((struct
       Shape*)(obj))->funcTable->draw((obj)))
107 #define Shape_MOVETO(obj, newx, newy) \
     (((struct Shape*)(obj))->funcTable->moveTo((obj), (newx),
108
         (newy)))
109
110 #define Rectangle_SETWIDTH(obj, width) \
111
     ((struct RectangleFuncTable*)((struct
         Shape*)(obj))->funcTable)->setWidth( \
112
       (obj), (width))
113
114 #define Shape_DESTROY(obj) (((struct
       Shape*)(obj))->funcTable->destructor_((obj)))
115 /* A function that uses a Shape polymorphically */
116 void handleShape(struct Shape* s) {
117
     Shape_MOVETO(s, 0, 0);
118 }
```

Introduction	and	

Quick application

Conclusions

```
119 int main() {
120
     int i:
121
     struct Shape* shapes[2];
122
     struct Shape* r;
123
     /* using shapes polymorphically */
124
      shapes[0] = Rectangle_init(20,12,123,321);
125
      shapes[1] = Circle_init(21, 12, 2012);
126
     for (i = 0; i < 2; ++i)
127
     Ł
128
        Shape_DRAW(shapes[i]);
129
        handleShape(shapes[i]);
130
     }
131
     /* accessing Rectangle specific data */
132
     r = Rectangle_init(1, 2, 3, 4);
133
     Rectangle_SETWIDTH(r, 5);
134
     Shape_DRAW(r);
135
      Shape_DESTROY(r);
136
137
     for (i = 1; i \ge 0; --i)
138
        Shape_DESTROY(shapes[i]);
139 }
```

Introduction	and	

Quick application

・ロン ・回と ・ヨン ・ヨン

3

34 / 36

Conclusions

Pros

OOC vs. C

- better, more logical structuring of code

- decoupling: separating implementation from usage

- code recycling

Pros

OOC vs. C

- better, more logical structuring of code
- decoupling: separating implementation from usage
- code recycling

Cons

- requires in-depth programming knowledgecode is more complex and
- harder to write
- manual memory management
- (manual *everything* actually...) - no syntatic sugar to help write
- OO-code =>more lines of code
- =>more time

Introduction and motivation	OOP features and concepts	Quick application	Conclusions ○●○
Further reading			

- Object Oriented Programming with ANSI C, by Axel-Tobias Schreiner (free ebook)

- Google is your friend

000000	00000000000	quick application	000
Thank you!			

Thank you for not falling asleep! Any questions?