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
What is OOP?
What is OOP?
- programming paradigm
What is OOP?
- programming paradigm
- sets of common features (attributes)
● What is OOP?
  - programming paradigm
  - sets of common features (attributes) => classes
What is OOP?
- programming paradigm
- sets of common features (attributes) ⇒ classes
- particular instances of classes
What is OOP?
- programming paradigm
- sets of common features (attributes) $\Rightarrow$ classes
- particular instances of classes $\Rightarrow$ objects
What is OOP?
- programming paradigm
- sets of common features (attributes) \( \Rightarrow \) classes
- particular instances of classes \( \Rightarrow \) objects
- manipulating objects
What is OOP?
- programming paradigm
- sets of common features (attributes) => classes
- particular instances of classes => objects
- manipulating objects => methods
Can OOP be achieved using only ANSI-C code?
Can OOP be achieved using only ANSI-C code?
- yes.
Can OOP be achieved using only ANSI-C code?
- yes.
- paradigm vs. language feature
Can OOP be achieved using only ANSI-C code?

- yes.
- paradigm vs. language feature
- OO-languages (C++, Java, Python etc.)
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
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
C++ code

object->method(some_args);
```
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);
```
How can it be done?
How can it be done?
- with structs,
How can it be done?
- with structs, pointers
How can it be done?
- with structs, pointers and other wonderful things
What is OOP good for?
What is OOP good for?
- data representation and functionality separated from usage
What is OOP good for?
- data representation and functionality separated from usage
- divide and conquer
What is OOP good for?
- data representation and functionality separated from usage
- divide and conquer
- re-use of code
What is OOP good for?
- data representation and functionality separated from usage
- divide and conquer
- re-use of code
- enhanced code readability
Why do it?

- Why should I use C in my program?
Why should I use C in my program?
- mostly because it’s faster
Why should I use C in my program?
- mostly because it’s faster
- environment where C++ or other compilers not available
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.
Are there any disadvantages with this approach?
Are there any disadvantages with this approach?
- rather complex code
Are there any disadvantages with this approach?
- rather complex code
- possible loss of type safety
Are there any disadvantages with this approach?
- rather complex code
- possible loss of type safety
- programmer time
Are there any disadvantages with this approach?
- rather complex code
- possible loss of type safety
- programmer time
- error prone
Are there any disadvantages with this approach?
- rather complex code
- possible loss of type safety
- programmer time
- error prone
- manual memory management
### Listing 1: C++ class example

```cpp
// includes and stuff

class Rectangle {
private:
    int x, y;
    int width;
    int height;
public:
    // getters, setters, if needed
    void draw();
};

void Rectangle::draw() {
    std::cout << "Just drew a nice " << width << " by " << height << " rectangle at position (" << x << ", ", " << y << ")!";
}
```

### Listing 2: C class example

```c
// includes and stuff
typedef struct Rectangle {
    int x, y;
    int width;
    int height;
} Rectangle;

void draw(Rectangle* obj) {
    printf("I just drew a nice %d by %d rectangle at position (%d, %d)!", obj->width, obj->height, obj->x, obj->y);
}
```
Listing 3: C++ object example

1 //pretend that everything we already wrote is here
2 //and create an object where needed
3 Rectangle* r = new Rectangle();

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);
Classes, objects, methods, constructors and destructors

Listing 5: C++ methods example

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

Listing 6: C methods example

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

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

Listing 8: C constructor example

```c
// previously defined stuff is here, as usual
Rectangle* Rectangle_init(int initx, int inity, int initw, int inith) {
    struct Rectangle* obj = malloc(sizeof *obj);
    obj->x = initx;
    obj->y = inity;
    obj->width = initw;
    obj->height = inith;
    return obj;
}
// and this is how you would use it
Rectangle* r = Rectangle_init(1,2,3,4);
```
About destructors in C++

∼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++

~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
Encapsulation

- object data is contained and hidden inside of the object
- access to data is restricted to members of that class or other particular classes
- organizing 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
About encapsulation in C++
- offers some syntactic 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++
- offers some syntactic 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 syntactic 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
- 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
Listing 9: C++ inheritance example

```cpp
class Shape {
  /* Shape class members */
};
class Rectangle : public Shape {
  /* Rectangle class members */
};
```

Listing 10: C inheritance example

```c
struct Shape {
  /* base class members */
};
struct Rectangle {
  struct Shape super;
  /* derived class members */
};
```
- 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
- polymorphism takes advantage of inheritance
Listing 11: C++ polymorphism example

```cpp
class Shape { // abstract interface
public:
    virtual void draw() = 0; // pure virtual function
};

class Rectangle : public Shape { // inheritance
    // other stuff here too of course
public:
    virtual void draw(); // implement this along the way
};

// some function that handles a shape polymorphically
void handleShape(Shape* s) {
    s->draw(); // then do something to the shape
}

// usage
Shape* shape;
shape = new Rectangle();
handleShape(s);
```
Listing 12: C++ OOP full example

```cpp
#include <iostream>

/* Shape abstract interface */

class Shape {
public:
    virtual void draw() = 0;
    virtual void moveTo(int newx, int newy) = 0;
};
```
/* Rectangle class */

class Rectangle : public Shape {
private:
    int x, y;
    int width;
    int height;
public:
    Rectangle(int initx, int inity, int initw, int inith);
    int getX() { return this->x; }
    int getY() { return this->y; }
    int getWidth() { return this->width; }
    int getHeight() { return this->height; }
    void setX(int newx) { this->x = newx; }
    void setY(int newy) { this->y = newy; }
    void setWidth(int neww) { this->width = neww; }
    void setHeight(int newh) { this->height = newh; }
    virtual void draw();
    virtual void moveTo(int newx, int newy);
};
```cpp
Rectangle::Rectangle(int initx, int inity, int initw, int inith) {
    x = initx;
    y = inity;
    width = initw;
    height = inith;
}

void Rectangle::draw() {
    std::cout << "Just drew a nice " << width
              << " by " << height
              << " rectangle at position (" << x
              << ", " << y << ")!" << std::endl;
}

void Rectangle::moveTo(int newx, int newy) {
    x = newx;
    y = newy;
    std::cout << "Moving your rectangle to (" << x
              << ", " << y << ")!" << std::endl;
}
```
/* Circle class */
class Circle : public Shape {
private:
    int x,y;
    int radius;
public:
    Circle(int initx, int inity, int initr);
    virtual void draw();
    virtual void moveTo(int newx, int newy);
    int getX() { return this->x; }
    int getY() { return this->y; }
    int getRadius() { return this->radius; }
    void setX(int newx) { this->x = newx; }
    void setY(int newy) { this->y = newy; }
    void setRadius(int newr) { this->radius = newr; }
};
Circle::Circle(int initx, int inity, int initr) {
    x = initx;
    y = inity;
    radius = initr;
}

void Circle::draw() {
    std::cout << "Just drew a perfect circle of radius "
              << radius << " at position (" << x << ", " << y << ")!" << std::endl;
}

void Circle::moveTo(int newx, int newy) {
    x = newx;
    y = newy;
    std::cout << "Moving your circle to (" << x
              << ", " << y << ")!" << std::endl;
}
/* A function that uses a Shape polymorphically */

void handleShape(Shape* s) {
    std::cout << "Bad shape! Go to the corner!" << std::endl;
    s->moveTo(0,0);
}


int main() {
    /* using shapes polymorphically */

    Shape * shapes[2];
    shapes[0] = new Rectangle(20, 12, 123, 321);
    shapes[1] = new Circle(21, 12, 2012);

    for (int i = 0; i < 2; ++i) {
        shapes[i]->draw();
        handleShape(shapes[i]);
    }

    /* access a specific class function */

    Rectangle* r = new Rectangle(1, 2, 3, 4);
    r->setWidth(5);
    r->draw();

    return 0;
}
Listing 13: C OOP full example

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

struct Rectangle {
    struct Shape super;

    int x, y;
    int width;
    int height;
};

void Rectangle_draw(struct Shape* obj) {
    struct Rectangle* rdata = (struct Rectangle*) obj;

    printf("I just drew a nice %d by %d rectangle at position (%d, %d)! \n", 
           rdata->width, rdata->height, rdata->x, rdata->y);
}
```c
31 void Rectangle_moveTo(struct Shape* obj, int newx, int newy) {
    struct Rectangle * rdata = (struct Rectangle*) obj;

    rdata->x = newx;
    rdata->y = newy;

    printf("Moving your rectangle to (%d, %d)\n", rdata->x, rdata->y);
}

41 void Rectangle_setWidth(struct Shape* obj, int neww) {
    struct Rectangle * rdata = (struct Rectangle*) obj;

    rdata->width = neww;
}
```
```c
void Rectangle_destroy(struct Shape *obj) {
    Shape_destroy(obj);
    free(obj);
}

struct RectangleFuncTable {
    struct ShapeFuncTable super;
    void (*setWidth)(struct Shape* obj, int neww);
} rectangleFuncTable = {
    Rectangle_draw,
    Rectangle_moveTo,
    Rectangle_destroy },
    Rectangle_setWidth
};

struct Shape* Rectangle_init(int initx, int inity, int initw, int inith) {
    struct Rectangle* obj = (struct Rectangle*)
        malloc(sizeof(struct Rectangle));
    obj->super.funcTable = (struct ShapeFuncTable*)
        &rectangleFuncTable;
    obj->x = initx;
    obj->y = inity;
    obj->width = initw;
    obj->height = inith;
    return (struct Shape*)obj;
}
```
/* Circle class */
struct Circle {
  struct Shape super;
  int x, y;
  int radius;
};

void Circle_draw(struct Shape* obj) {
  struct Circle* cdata = (struct Circle*) obj;
  printf("Just drew a perfect circle of radius %d at position (%d, %d)\n",
         cdata->radius, cdata->x, cdata->y);
}

void Circle_moveTo(struct Shape* obj, int newx, int newy) {
  struct Circle* cdata = (struct Circle*) obj;
  cdata->x = newx;
  cdata->y = newy;
  printf("Moving your circle to (%d, %d)\n",
         cdata->x, cdata->y);
}
```c
void Circle_destroy(struct Shape *obj) {
    Shape_destroy(obj);
    free(obj);
}

struct CircleFuncTable {
    struct ShapeFuncTable super;
} circleFuncTable = {
    Circle_draw,
    Circle_moveTo,
    Circle_destroy
};

struct Shape* Circle_init(int initx, int inity, int initr) {
    struct Circle* obj = (struct Circle*)malloc(sizeof(struct Circle));
    obj->super.funcTable = (struct ShapeFuncTable*)
        &circleFuncTable;
    obj->x = initx;
    obj->y = inity;
    obj->radius = initr;
    return (struct Shape*)obj;
}
```
# define Shape_DRAW(obj) (((struct Shape*)(obj))->funcTable->draw((obj)))
# define Shape_MOVETO(obj, newx, newy) \\
    (((struct Shape*)(obj))->funcTable->moveTo((obj), (newx), (newy)))

# define Rectangle_SETWIDTH(obj, width) \\
    ((struct RectangleFuncTable*)((struct Shape*)(obj))->funcTable)->setWidth( \\
    (obj), (width))

# define Shape_DESTROY(obj) (((struct Shape*)(obj))->funcTable->destructor_((obj)))

/* A function that uses a Shape polymorphically */
void handleShape(struct Shape* s) {
    Shape_MOVETO(s, 0, 0);
}
int main() {
    int i;
    struct Shape* shapes[2];
    struct Shape* r;
    /* using shapes polymorphically */
    shapes[0] = Rectangle_init(20, 12, 123, 321);
    shapes[1] = Circle_init(21, 12, 2012);
    for (i = 0; i < 2; ++i) {
        Shape_DRAW(shapes[i]);
        handleShape(shapes[i]);
    }
    /* accessing Rectangle specific data */
    r = Rectangle_init(1, 2, 3, 4);
    Rectangle_SETWIDTH(r, 5);
    Shape_DRAW(r);
    Shape_DESTROY(r);
    for (i = 1; i >= 0; --i)
        Shape_DESTROY(shapes[i]);
}
OOC vs. C

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

**Cons**
- requires in-depth programming knowledge
- code is more complex and harder to write
- manual memory management (manual *everything* actually...)
- no syntactic sugar to help write OO-code ➞ more lines of code ➞ more time
- Object Oriented Programming with ANSI C, by Axel-Tobias Schreiner (free ebook)
- Google is your friend
Thank you for not falling asleep!
Any questions?