# Topic 13: C++ Object-Oriented Programming

School of Electrical and Computer Engineering  
Cornell University  
revision: 2017-10-23-09-35

<table>
<thead>
<tr>
<th></th>
<th>Points, Lines, Triangles</th>
<th></th>
<th>Groups, Canvases, Drawings</th>
<th></th>
<th>Lists</th>
<th></th>
<th>Vectors</th>
<th></th>
<th>Revisiting Strings and Standard Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Procedural programming

- Programming model organized around procedures
- Procedures take input data, process it, produce output data
- Focus is on the logic to process data

Object-oriented programming

- Programming model organized around objects
- Objects contain data and actions to perform on data
- Classes are the “types” of objects, objects are instances of classes
- **Classes** are nouns, **methods** are verbs/actions
- Classes are organized according to various relationships
  - **composition** relationship ("Class X has a Y")
  - **generalization** relationship ("Class X is a Y")
  - **association** relationship ("Class X acts on Y")

Example class diagram for animals
Example class diagram for shapes and drawings
Pseudocode for shapes and drawings

Adding shapes to a drawing and displaying on screen

1. set drawing to new Drawing
2. add point0 to drawing
3. add line0 to drawing
4. add triangle0 to drawing

5. set group to new Group
6. add triangle1 to group
7. add triangle2 to group
8. add group to drawing

10. display drawing on screen

Pseudocode for display drawing

1. set canvas to new Canvas
2. for shape in drawing’s internal group
   3. draw shape on canvas
4. display canvas on screen
1. Points, Lines, Triangles

- Perfectly possible to use object-oriented programming in C

```c
typedef struct
{
    double x;
    double y;
}
point_t;

void point_translate( point_t* point_p,
    double x_offset, double y_offset )
{
    point_p->x += x_offset; point_p->y += y_offset;
}

void point_scale( point_t* point_p,
    double scale )
{
    point_p->x *= scale; point_p->y *= scale;
}

void point_rotate( point_t* point_p,
    double angle )
{
    const double pi = 3.14159265358979323846;
    double s = std::sin((angle*pi)/180);
    double c = std::cos((angle*pi)/180);

    double x_new = (c * point_p->x) - (s * point_p->y);
    double y_new = (s * point_p->x) + (c * point_p->y);

    point_p->x = x_new; point_p->y = y_new;
}

void point_print( point_t* point_p )
{
    std::printf("(%.2f,%.2f)", point_p->x, point_p->y);
}
```
```c
int main( void )
{
    point_t pt;
    pt.x = 1;
    pt.y = 2;

    point_translate( &pt, 1, 0 );
    point_scale( &pt, 2 );
    return 0;
}
```
• C++ allows functions to be defined *within* the struct namespace
• C++ struct has both member fields and member functions

```cpp
struct Point
{
    double x; // member fields
    double y; //

    // (static) member functions

static void translate( Point* point_p, double x_offset, double y_offset )
{
    point_p->x += x_offset; point_p->y += y_offset;
}

static void scale( Point* point_p, double scale )
{
    point_p->x *= scale; point_p->y *= scale;
}

static void rotate( Point* point_p, double angle )
{
    const double pi = 3.14159265358979323846;
    double s = std::sin((angle*pi)/180);
    double c = std::cos((angle*pi)/180);

double x_new = (c * point_p->x) - (s * point_p->y);
double y_new = (s * point_p->x) + (c * point_p->y);

    point_p->x = x_new; point_p->y = y_new;
}

static void print( Point* point )
{
    std::printf("(%.2f,%.2f)" , point_p->x, point_p->y );
}
};
```
int main( void )
{
    Point pt;
    pt.x = 1;
    pt.y = 2;

    Point::translate( &pt, 1, 0 );
    Point::scale( &pt, 2 );
    return 0;
}
1. Points, Lines, Triangles

- Non-static member functions have an implicit this pointer
- The this pointer serves same purpose as point_p
- Non-static member functions which do not modify fields are const
- Non-static member functions are accessed using the dot (.) operator in the same way we access fields

```cpp
struct Point
{
    double x; // member fields
    double y; //

    // (non-static) member functions

    void translate( double x_offset, double y_offset )
    {
        this->x += x_offset; this->y += y_offset;
    }

    void scale( double scale )
    {
        this->x *= scale; this->y *= scale;
    }

    void rotate( double angle )
    {
        ...
        double x_new = (c * this->x) - (s * this->y);
        double y_new = (s * this->x) + (c * this->y);
        this->x = x_new; this->y = y_new;
    }

    void print() const
    {
        std::printf("(%.2f,%.2f)", this->x, this->y);
    }
};
```
```c
int main( void )
{
    Point pt;
    pt.x = 1;
    pt.y = 2;

    pt.translate( 1, 0 );
    pt.scale( 2 );
    return 0;
}
```
• Member fields are in scope within every non-static member function
• No need to explicitly use this pointer

```cpp
struct Point
{
    double x; // member fields
    double y; //

    // (non-static) member functions

    void translate( double x_offset, double y_offset )
    {
        x += x_offset; y += y_offset;
    }

    void scale( double scale )
    {
        x *= scale; y *= scale;
    }

    void rotate( double angle )
    {
        ...
        double x_new = (c * x) - (s * y);
        double y_new = (s * x) + (c * y);
        x = x_new; y = y_new;
    }

    void print() const
    {
        std::printf( "(%.2f,%.2f)", x, y );
    }
};
```
```c
int main( void )
{
    Point pt;
    pt.x = 1;
    pt.y = 2;

    pt.translate( 1, 0 );
    pt.scale( 2 );
    return 0;
}
```

- Static member fields/functions are associated with the struct
- Non-static member fields/functions are associated with the object
- An object is just an instance of a struct with member functions