ECE 2400 Computer Systems Programming  
Fall 2018  

Topic 13: Object-Oriented Programming  

School of Electrical and Computer Engineering  
Cornell University  

revision: 2018-10-24-09-07  

1 C++ Classes  

1.1. C++ Member Functions ........................................... 7  
1.2. C++ Constructors and Destructors ............................... 13  
1.3. C++ Operator Overloading ........................................ 16  
1.4. C++ Data Encapsulation .......................................... 19  

Copyright © 2018 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.
Object-oriented programming

• Programming is organized around defining, instantiating, and manipulating objects which contain data (i.e., fields, attributes) and code (i.e., methods)
• 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 chess pieces
Example class diagram for shapes and drawings
1. C++ Classes

• Perfectly possible to use object-oriented programming in C

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

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

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

void point_rotate( point_t* this, 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 * this->x) - (s * this->y);
    double y_new = (s * this->x) + (c * this->y);
    this->x = x_new; this->y = y_new;
}

void point_print( point_t* this )
{
    std::printf("(%.2f,%.2f)", this->x, this->y );
}
```
int main( void )
{
    point_t pt;
    pt.x = 1;
    pt.y = 2;
    point_translate( &pt, 1, 0 );
    point_scale ( &pt, 2 );
    return 0;
}
1. C++ Member Functions

- 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* this_,
                           double x_offset, double y_offset )
    {
        this_->x += x_offset; this_->y += y_offset;
    }

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

    static void rotate( Point* this_, 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;
    }

    static void print( Point* this_ )
    {
        std::printf("(%.2f,%.2f)", this_->x, this_->y);
    }
};
```
int main( void )
{
    Point pt;
    pt.x = 1;
    pt.y = 2;
    Point::translate( &pt, 1, 0 );
    Point::scale( &pt, 2 );
    return 0;
}
• Notice the subtle change from this to this_
• this is actually a keyword in C++! (special syntax/semantics)
• Non-static member functions have an implicit this pointer
• 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 );
    }
};
```
```cpp
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);
    }
};
```
```cpp
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
1.2. C++ Constructors and Destructors

• How do we construct and destruct an object?
• In C, we used `foo_construct` and `foo_destruct`
• In C++, we could add `construct` and `destruct` member functions

```c
int main( void )
{
    Point pt;
    pt.construct();
    pt.x = 1;
    pt.y = 2;

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

• What if we call `translate` before `construct`?
• What if we call `translate` after `destruct`?
• What if `translate` throws an exception?

• C++ adds support for language-level constructors and destructors
• Involves new syntax and semantics
```cpp
struct Point {
    double x;
    double y;

    // constructor
    Point() {
        x = 0.0;
        y = 0.0;
    }

    // destructor
    ~Point() { }

    try {
        Point point0; // constructor called
        point0.x = 1;
        point0.y = 2;

        Point point1; // constructor called
        point1.x = 1;
        point1.y = 2;

        // destructor called if
        // exception is thrown
        throw "example exception";

        point1.translate(0, 1);
        point1.scale(0.5);
    }

    catch (const char* e) {
        std::printf("ERROR: %s\n", e);
    }

    pt0.translate(1, 0);
    pt0.scale(2);
    return 0; // destructor called
};
```

https://repl.it/@cbatten/ece2400-T12-ex1

- Constructors automatically called with new
- Destructors automatically called with delete

```cpp
Point* pt0_p = new Point; // constructor called
delete pt0_p; // destructor called

Point* pt1_p = new Point[4]; // constructor called 4 times
delete[] pt1_p; // destructor called 4 times
```
• Constructors can take arguments to initialize members

• Copy constructors used for copying object
  – Used in initialization statements
  – Used when object is passed by value to a function

• Initialization lists initialize members before body of constructor
  – Avoids creating a temporary default object
  – Required for initializing reference members
  – Prefer initialization lists when possible

```cpp
struct Point
{
  double x; double y;

  // default constructor
  Point() { x = 0.0; y = 0.0; }

  // non-default constructor
  Point( int x_, int y_ )
  { x = x_; y = y_; }

  // copy constructor
  Point( const Point& pt )
  { x = pt.x; y = pt.y; }

  ...;
};

int main( void )
{
  Point pt0; // default constructor called
  Point pt1( 1, 2 ); // non-default constructor called
  Point pt2 = pt1; // copy constructor called
  return 0;
}
```
1.3. C++ Operator Overloading

- C++ operator overloading enables using built-in operators (e.g., +, -, *, /) with user-defined types

- Applying an operator to a user-defined type essentially calls an overloaded function (either a member function or a free function)

```cpp
Point operator+( const Point& pt0,
const Point& pt1 )
{
    // calls copy constructor
    Point tmp = pt0;
    tmp.translate( pt1.x, pt1.y );
    return tmp;
}
```

```cpp
int main( void )
{
    Point pt0(1,2);
    Point pt1(3,4);
    Point pt2 = pt0 + pt1;
    return 0;
}
```
Point operator*( const Point& pt, double scale )
{
    Point tmp = pt;
    tmp.scale( scale );
    return tmp;
}

Point operator*( double scale, const Point& pt )
{
    Point tmp = pt;
    tmp.scale( scale );
    return tmp;
}

Point operator%( const Point& pt, double angle )
{
    Point tmp = pt;
    tmp.rotate( angle );
    return tmp;
}

Point operator%( double angle, const Point& pt )
{
    Point tmp = pt;
    tmp.rotate( angle );
    return tmp;
}

• Operator overloading enables elegant syntax for user-defined types

Point pt0(1,2);
pt0.translate(5,3);
pt0.rotate(45);
pt0.scale(1.5);
Point pt1 = pt0;

Point pt0(1,2);
Point pt1 = 1.5 * ( ( pt0 + Point(5,3) ) % 45 );
1.4. C++ Data Encapsulation

- Recall the importance of separating interface from implementation
- This is an example of abstraction
- In this context, also called information hiding, data encapsulation
  - Hides implementation complexity
  - Can change implementation without impacting users

- So far, we have relied on a policy to enforce data encapsulation
  - Users of a struct could still directly access member fields

```cpp
int main( void )
{
    Point pt(1,2);
    pt.x = 13; // direct access to member fields
    return 0;
}
```

- In C++, we can enforce data encapsulation at compile time
  - By default all member fields and functions of a struct are public
  - Member fields and functions can be explicitly labeled as public or private
  - Externally accessing an internal private field causes a compile time error

```cpp
struct Point
{
    private:
        double x; double y;

    public:
        // default constructor
        Point() { x = 0.0; y = 0.0; }

        // non-default constructor
        Point( int x_, int y_ ) { x = x_; y = y_; }

    };
```
• In C++, we usually use `class` instead of `struct`
  – By default all member fields and functions of a `struct` are `public`
  – By default all member fields and functions of a `class` are `private`
  – We should almost always use `class` and explicitly use `public` and `private`

```cpp
class Point // almost always use class instead of struct
{
    public: // always explicitly use public ...
    private: // ... or private
};
```

• We are free to change how we store the point
• We could even use the heap?