Topic 13: Object-Oriented Programming

1 C++ Classes

1.1. C++ Member Functions .................................................. 6
1.2. C++ Constructors ..................................................... 10
1.3. C++ Operator Overloading .......................................... 12
1.4. C++ Rule of Three ...................................................... 15
1.5. C++ Data Encapsulation .............................................. 23
1.6. C++ I/O Streams ....................................................... 24

2 C++ Inheritance .......................................................... 27

3 Object-Oriented Data Structures ....................................... 46

Sections marked with a star (★) are not covered in lecture but are instead covered in the online lecture notes. Students are responsible for all material covered in lecture and in the online lecture notes. Material from the online lecture notes will definitely be assessed in the prelim and final exam.

Copyright © 2019 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 shapes and drawings
1. C++ Classes

- Perfectly possible to use object-oriented programming in C

```cpp
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);
}
```
```cpp
int main( void )
{
    point_t pt;
    pt.x = 1;
    pt.y = 2;
    point_translate( &pt, 1, 0 );
    point_scale ( &pt, 2 );
    return 0;
}
```
1.1. C++ Member Functions

- C++ allows functions to be defined within the struct namespace
- C++ struct has both member fields and member functions
- Member functions have an implicit this pointer
- Member functions which do not modify fields are const

```c++
struct Point
{
    double x; // member fields
    double y; //

    // member functions

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

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

    void point_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 point_print() const
    {
        std::printf("(%.2f,%.2f)", this->x, this->y);
    }
};
```
• Non-static member functions are accessed using the dot (.) operator in the same way we access fields

```
int main( void )
{
    Point pt;
    pt.x = 1;
    pt.y = 2;
    pt.point_translate( 1, 0 );
    pt.point_scale( 2 );
    return 0;
}
```

• Recall object-oriented prog in C

```
int main( void )
{
    point_t pt;
    pt.x = 1;
    pt.y = 2;
    point_translate( &pt, 1, 0 );
    point_scale   ( &pt, 2 );
    return 0;
}
```

point_translate( &pt, 1, 0 ) ↔ pt.point_translate( 1, 0 )
foo( &bar, ... ) ↔ bar.foo( ... )
- Member functions are in struct namespace
- No need to use the `point_` prefix
- Member fields are in scope within every member function
- No need to explicitly use this pointer

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

  // 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("(%0.2f,%0.2f)", x, y);
  }
};
```
```cpp
int main( void )
{
    Point pt;
    pt.x = 1;
    pt.y = 2;
    pt.translate( 1, 0 );
    pt.scale( 2 );
    return 0;
}
```

- A class is just a struct with member functions
- An object is just an instance of a struct with member functions
1.2. C++ Constructors

- We want to avoid the user from directly accessing member fields
- We want to ensure an object is always initialized to a known state
- In C, we used `foo_construct`
- In C++, we could add a `construct` member function

```cpp
int main( void )
{
    Point pt;
    pt.construct( 1, 2 );
    pt.translate( 1, 0 );
    pt.scale( 2 );
    return 0;
}
```

- What if we call `translate` before `construct`?
- What if we call `construct` multiple times?
- We want a way to specify a special “constructor” member function
  - *always* called when you create an object
  - cannot be called directly, can *only* be called during object creation
- C++ adds support for language-level constructors
  (i.e., special member functions)
  - no return type
  - same name as the class
- Can use function overloading to have many different constructors
struct Point
{
    double x;
    double y;
}

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

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

...

int main( void )
{
    Point pt0;
    Point pt1( 1, 2 );
    pt0 = pt1;
    pt0.translate( 1, 0 );
    return 0;
}
1. C++ Classes

- Constructors automatically called with `new`

```cpp
Point* pt0_p = new Point;  // constructor called
Point* pt1_p = new Point[4];  // constructor called 4 times
```

- 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(double x_, double y_) { x = x_; y = y_; }
};
```

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)
Point operator+( const Point& pt0, const Point& pt1 )
{
    Point tmp = pt0;
    tmp.translate( pt1.x, pt1.y );
    return tmp;
}

int main( void )
{
    Point ptA(1,2);
    Point ptB(3,4);
    Point ptC = ptA + ptB;
    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++ Rule of Three

- What if point coordinates are allocated on the heap?

```cpp
struct DPoint {
    double* x_p;
    double* y_p;

    DPoint() {
        x_p = new double;
        y_p = new double;
        *x_p = 0.0;
        *y_p = 0.0;
    }

    void translate(double x_offset, double y_offset) {
        *x_p += x_offset;
        *y_p += y_offset;
    }

    ...}

int main(void) {
    DPoint pt0;
    pt0.translate(1, 0);
    return 0;
}
```

C++ Destructors

- Destructors are special member functions for destroying an object

```cpp
struct DPoint {
    double* x_p;
    double* y_p;

    DPoint() {
        x_p = new double;
        y_p = new double;
        *x_p = 0.0;
        *y_p = 0.0;
    }

    ~DPoint() {
        delete x_p;
        delete y_p;
    }

    ...}

int main( void )
{
    DPoint pt0;
    pt0.translate( 1, 0 );
    return 0;
}
```
• What if we copy an object with dynamically allocated memory?

```cpp
struct DPoint {
    double* x_p;
    double* y_p;

    DPoint() {
        x_p = new double;
        y_p = new double;
        *x_p = 0.0;
        *y_p = 0.0;
    }

    ~DPoint() {
        delete x_p; delete y_p;
    }

    // ... }

    int main( void )
    { }

    DPoint pt0;
    DPoint pt1 = pt0;
    pt0.translate( 1, 0 );
    return 0;
}
```
C++ Copy Constructors

- Copy constructors are special member functions for constructing a new object from an old object.

```cpp
struct DPoint {
    double* x_p;
    double* y_p;

dpoint( const DPoint& pt ) {
    x_p = new double;
    y_p = new double;
    *x_p = *pt.x_p;
    *y_p = *pt.y_p;
}

~DPoint() {
    delete x_p; delete y_p;
}
...);

int main( void ) {
    DPoint pt0;
    DPoint pt1 = pt0;
    pt0.translate( 1, 0 );
    return 0;
}
```
• What if we assign an object with dynamically allocated memory?

```cpp
struct DPoint {
    double* x_p;
    double* y_p;

    DPoint() {
        x_p = new double;
        y_p = new double;
        *x_p = 0.0;
        *y_p = 0.0;
    }

    ~DPoint() {
        delete x_p; delete y_p;
    }

    // ...

    int main( void ) { 
        DPoint pt0;
        DPoint pt1;
        pt1 = pt0;
        pt0.translate( 1, 0 );
        return 0;
    }
};
```
C++ Assignment Operators

- An overloaded assignment operator will be called for assignment

```cpp
struct DPoint
{
  double* x_p;
  double* y_p;

  DPoint& operator=( const DPoint& pt )
  {
    *x_p = *pt.x_p;
    *y_p = *pt.y_p;
    return *this;
  }

  ...
};

int main( void )
{
  DPoint pt0;
  DPoint pt1;
  pt1 = pt0;
  pt0.translate( 1, 0 );
  return 0;
}
```
C++ Rule of Three

• Default destructor, copy constructor, and assignment operator will work fine for simple classes
• For a more complex class may need to define one of these ...
• ... and if you define one, then you probably need to define all three!

```
struct DPoint
{
    double* x_p;
    double* y_p;

    DPoint()
    {
        x_p = new double;
        y_p = new double;
        *x_p = 0.0;
        *y_p = 0.0;
    }

    DPoint( double x, double y )
    {
        x_p = new double;
        y_p = new double;
        *x_p = x;
        *y_p = y;
    }

    DPoint( const DPoint& pt )
    {
        x_p = new double;
        y_p = new double;
        *x_p = *pt.x_p;
        *y_p = *pt.y_p;
    }

    ~DPoint()
    {
        delete x_p;
        delete y_p;
    }

    DPoint&
    operator=( const DPoint& pt )
    {
        *x_p = *pt.x_p;
        *y_p = *pt.y_p;
        return *this;
    }

    ... 

};
```
C++ Exceptions and Destructors

- Destructors called automatically for all objects in scope when exception thrown

```cpp
struct DPoint {
    double* x_p;
    double* y_p;

    ~DPoint() {
        delete x_p;
        delete y_p;
    }

    void translate(double x_offset, double y_offset) {
        if ((x_offset > 100) || (y_offset > 100))
            throw 42;
        *x_p += x_offset;
        *y_p += y_offset;
    }

    ...}

int main( void )
{
    try {
        DPoint pt0;
        pt0.translate( 1e9, 0 );
    }
    catch ( int e ) {
        return e;
    }
    return 0;
}
```
1.5. 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( double x_, double 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 change point to store coordinates on the stack or heap
• Statically guaranteed that others cannot access this private implementation

### 1.6. C++ I/O Streams

• printf does not support user-defined types
• Need to encapsulate printing in a member function ...
• ... but this leads to very awkward syntax

```cpp
int main( void )
{
  Point pt0(1,2);
  Point pt1 = pt0;
  pt1.translate(2.0,2.0);

  std::printf("initial point = ");
  pt0.print();
  std::printf("\n");
  std::printf("translate by %.2f,%.2f\n", 2.0, 2.0 );
  std::printf("new point = ");
  pt1.print();
  std::printf("\n");
}
```
C++ Classes

1. C++ I/O Streams

- Clever use of operator overloading can provide cleaner syntax

```cpp
struct ostream
{
    // internal stream state
};

ostream cout;

ostream& operator<<( ostream& os, int i )
{
    std::printf("%d",i); return os;
}

ostream& operator<<( ostream& os, double d )
{
    std::printf("%.2f",d); return os;
}

ostream& operator<<( ostream& os, const char* str )
{
    std::printf("%s",str); return os;
}

ostream& operator<<( ostream& os, const Point& pt )
{
    pt.print(); return os;
}

struct EndOfLine
{
};

EndOfLine endl;

ostream& operator<<( ostream& os, const EndOfLine& endl )
{
    std::printf("\n"); return os;
}
```
int main( void )
{
    Point pt0(1,2);
    Point pt1 = pt0;
    pt1.translate(2.0,2.0);

    cout << "initial point = " << pt0 << endl;
    cout << "translate by " << 2.0 << "," << 2.0 << endl;
    cout << "new point = " << pt1 << endl;
}

• The standard C++ library provides powerful streams
  – iostream: write/read standard I/O as streams
  – fstream: write/read files as streams
  – sstream: write/read strings as streams