Why object-oriented programming?

Everyone else is doing it.

It will make me a better programmer.

OO programs are better.

No, really, why object-oriented?

- You are interested in user interfaces.
  - Direct manipulation
  - Current state
  - Polymorphic operations
- You want OO for other applications.
  - Incremental algorithms
  - Protocol interpretation

Ok, so how about C++?

Object-Oriented Systems Programming

- Data abstraction
- Inheritance
- Cheap delivery platform
- Heterogeneity
- Type loopholes (casts)
- Static type-checking
- Machine-level operations (e.g., bitwise and)

Schedule

8:30	Motivation
8:40	Data abstraction
9:40	Break
9:50	Inheritance
10:35	Break
10:45	Hints
11:20	Summary

Data abstraction

Organize programs by types

Define types by function interfaces

Hide representation and implementation

Programs easier to modify

Example: Rectangle

From Xlib.h:

```
typedef struct {
    short x, y;
    unsigned short width, height;
} XRectangle;
```

Right coordinate is *x+width-1*

What if rep is changed to two points?

```
class Rectangle {
public:
    Rectangle();
    ~Rectangle();

    Coord left();
    Coord top();
    Coord right();
    Coord bottom();

    Coord width();
    Coord height();

    void move(Coord, Coord);
    void size(Coord width, Coord height);
private:
     ...
};
```

C++ classes

- Can have functions, or “methods”
- Target is implicit argument to function
- Must specify parameter types
- Can access *public* functions anywhere
- Can access *private* only from own functions
- Procedure is a function that returns *void*

Usage

```
Rectangle r;
Rectangle* rp;

x = r.left();
r.move(x + 1, y + 1);

rp = new Rectangle;
x = rp->right();
rp->size(1, 1);
```

Constructors and destructors

Create an instance => call constructor

Delete an instance <= call destructor

```
Rectangle* someclass::f() {
    Rectangle r;
    /* call Rectangle::Rectangle() */

    Rectangle* rp;
    rp = new Rectangle;
    /* call Rectangle::Rectangle() */

    return rp;
    /* call Rectangle::~Rectangle() on r */
}

void free_rect(Rectangle* rp) {
    delete rp;
    /* call Rectangle::~Rectangle */
}
```

Private section

```
class Rectangle {
public:
    ...
private:
    Coord _left;
    Coord _top;
    Coord _width;
    Coord _height;
};
```

Conventions

- List public before private
- List fields before methods
- Uppercase begins type name
- Lowercase begins method or variable
- Underscore (_) begins field
- Avoid public fields (remember XRectangle)

Function definition

Use “::” for to delimit class scope

Implicit *this*-> for fields and methods

```
void Rectangle::move(Coord left, Coord top) {
// move(Rectangle* this, Coord left, Coord top)

    _left = left;
    // this->_left = left;

    _top = top;
    // this->_top = top;
}
```

Inline functions

Add “inline” as storage class

Put body in header file

```
inline Coord Rectangle::left() {
    return _left;
}
```

Short-hand:

```
class Rectangle {
    ...
    Coord left() { return _left; }
    ...
};
```

Overloading by target

```
class Circle {
    ...
    Coord left();
    ...
};
```

Distinguish by usage

```
Rectangle r;
Circle c;

if (c.left() < r.left()) {
    ...
}
```

Overloading by parameter type

```
class Rectangle {
    ...
    Coord left() { return _left; }
    void left(Coord c) { _left = c; }
    ...
};
```

Overloading can be ambiguous

```
class BigTrouble {
public:
    void doit(A*);
    void doit(B*);
};

BigTrouble b;

b.doit(0);
```

Operators

Can overload infix operators

```
class Rectangle {
    ...
    Rectangle operator +(Rectangle);
    ...
};

r = r1 + r2;
```

Conversion operators

```
class A {
    void f(T2);
};

T1 x;

a.f(x);
```

Look for T1::operator T2()

Look for T2::T2(T1)

Using conversion operators

```
class String {
public:
    String(char*);

    operator char*();
    String operator =(String);
     ...
};

String s;

s = “this is a test”;
```

Conversion operators can be ambiguous

```
class Coord {
public:
   Coord(int);

   operator int();

   Coord operator +(Coord);
};

Coord x0, x1;

x0 = 1 + x1;
```

Conversion operators can be confusing

```
class Semaphore {
public:
   Semaphore(int);

   void wait();
   void signal();
};

Semaphore mutex(1);

mutex = 1;
```

Friends

public is all functions

private is only class functions

friend restricts to a class or function

```
class Rectangle {
    ...
private:
    friend Rectangle operator +(Rectangle, Rectangle);
    friend class RectangleList;
};
```

```
//
// Sample session:
// a 1 2 a 3 4 p
// polygon vertices: (1,2) (3,4)
// r 1 2 p
// polygon vertices: (3,4)
// r 3 4 p
// polygon vertices:
//

#include <stream.h>
#include <stdlib.h>

typedef int Coord;
```

```
class Vertex {
public:
    Vertex(Coord x, Coord y);

    Coord x() { return _x; }
    Coord y() { return _y; }
    Vertex* next() { return _next; }
    Vertex* prev() { return _prev; }

    void link(Vertex*);
    void unlink();
private:
    Coord _x;
    Coord _y;
    Vertex* _next;
    Vertex* _prev;
};
```

```
class Polygon {
public:
    Polygon();
    ~Polygon();

    void append(Coord x, Coord y);
    void remove(Coord x, Coord y);
    void print();

    Vertex* head() { return _dummy->next(); }
    Vertex* tail() { return _dummy->prev(); }
    Vertex* end() { return _dummy; }
private:
    Vertex* _dummy;
};
```

```
//
// Read an integer from standard input.
// If the input is not an integer,
// print an error message and exit the program.
//

int readint() {
    int n;
    if (!(cin >> n)) {
        cerr << "readint failed\n";
        exit(1);
    }
    return n;
}
```

```
//
// The main program reads commands from input
// to create or print * the polygon.
//

int main() {
    Polygon p;
    char c;
    Coord x, y;
    while (cin >> c) {
        switch (c) {
        case 'a':
            x = readint(); y = readint();
            p.append(x, y);
            break;
        case 'r':
            x = readint(); y = readint();
            p.remove(x, y);
            break;
        case 'p':
            p.print();
            break;
        case ' ': case '\n': case '\t':
            // ignore white space
            break;
        default:
            cerr << "unexpected character " <<
                c << "\n";
            exit(1);
        }
    }
}
```

```
Polygon::Polygon() {
    _dummy = new Vertex(0, 0);
}

Polygon::~Polygon() {
    Vertex* v;
    Vertex* vnext;
    for (v = head(); v != end(); v = vnext) {
        vnext = v->next();
        delete v;
    }
    delete _dummy;
}
```

```
void Polygon::append(Coord x, Coord y) {
    tail()->link(new Vertex(x, y));
}

void Polygon::remove(Coord x, Coord y) {
    for (Vertex* v = head(); v != end(); v = v->next()) {
        if (v->x() == x && v->y() == y) {
            v->unlink();
            delete v;
            break;
        }
    }
}

void Polygon::print() {
    cout << "polygon vertices:";
    Vertex* v;
    for (v = head(); v != end(); v = v->next()) {
        cout << " (" << v->x() << "," << v->y() << ")";
    }
    cout << "\n";
}
```

```
Vertex::Vertex(Coord x, Coord y) {
    _x = x;
    _y = y;
    _next = this;
    _prev = this;
}

void Vertex::link(Vertex* v) {
    v->_next = _next;
    v->_prev = this;
    _next->_prev = v;
    _next = v;
}

void Vertex::unlink() {
    _next->_prev = _prev;
    _prev->_next = _next;
    _next = this;
    _prev = this;
}
```

Power of data abstraction

1. Can use without knowing or understanding the implementation

2. Can change implementation without affecting the usage

C++ classes provide a notation for defining abstract data types.

Inheritance

Other side of story

Distinguishes object-oriented programming

Share interfaces, vary implementations

Increases code-sharing

Example: User interfaces

Direct manipulation

Do something to this object on the screen

Something different for different objects

Rectangles, circles, polygons

Group of objects

Draw a list of objects (Glyphs)

```
void draw_list::(GlyphList* gl) {
    Glyph* g;
    for (g = gl->head(); g != gl->end(); g = g->next()) {
        g->draw();
    }
}
```

Data abstraction alone

All objects must be of type Glyph.

Case statement to handle draw individually.

Hard to add new kinds of glyphs (e.g., splines)

Inheritance

Define Glyph class with “virtual” draw function.

Make Rectangle, Circle, and Polygon subclasses of Glyph.

Each subclass implements its own draw.

Virtual functions

```
class Glyph {
public:
    virtual void draw();
};

class Rectangle : public Glyph {
public:
     ...
    virtual void draw();
     ...
};
```

Draw implementation

```
void Rectangle::draw() {
    point(left(), top());
    point(right(), top());
    point(right(), bottom());
    point(left(), bottom());
    closepath();
}

void Polygon::draw() {
    ...
}
```

Abstract classes

- No instances, only subclasses
- Define protocols
- Subclasses can access protected functions
- Glyph is abstract

Example: Expression tree evaluation

```
class Expr {
public:
    virtual int eval();
    virtual ~Expr();
protected:
    Expr();
};
```

Subclasses

```
class BinaryExpr : public Expr {
public:
   BinaryExpr(ExprOp, Expr*, Expr*);
   ~BinaryExpr();

   virtual int eval();
protected:
   ExprOp _op;
   Expr* _left;
   Expr* _right;
};

class IntLeaf : public Expr {
public:
   IntLeaf(int);

   virtual int eval();
private:
   int _value;
};
```

Top-level command loop

```
Expr* e = readexpr();
printf("%d\n", e->eval());
```

Readexpr cases

```
case 1:
   return new BinaryExpr(op_add, e1, e2);
case 2:
   return new IntLeaf(i);
 ...
```

Eval operations:

```
int BinaryExpr::eval() {
    int e;
    int e1 = _left->eval();
    int e2 = _right->eval();
    switch (_op) {
    case op_add:
        e = e1 + e2;
        break;
     ...
    }
    return e;
}

int IntLeaf::eval() {
    return _value;
}
```

Constructors

Create an object for a subclass

Call base class constructor BEFORE

```
Rectangle r;
// call Glyph(), then Rectangle()
```

Destructors

Delete an object of a subclass

Destructor must be virtual

Call base class destructor AFTER

```
Rectangle* r;
delete r;
// call ~Rectangle(), then ~Glyph()
```

Protected access

Suppose all glyphs have a bounding box.

Subclasses need to set box, others should not.

```
class Glyph {
public:
    ...
protected:
    void bounding(Rectangle&);
};
```

Power of inheritance

Share interfaces, vary implementations

Share code using interfaces (e.g., Box::draw)

Inheritance is unique to object-oriented programming.

Hints++

Everything is an object

It takes two to communicate

Optimize no code before its time

Young programming environments

Everything is an object

- No global functions
- No static variables
- No static functions
- Few global objects

Example: scanner

```
class InputFile {
public:
    InputFile();
    ~InputFile();

    static InputFile* open(Filename);

    Filename name() { return _name; }
    int lineno() { return _lineno; }

    boolean getline(char* dst, int maxchars);
private:
    Filename _name;
    FILE* _fp;
    int _lineno;
};
```

```
class Scanner {
public:
    Scanner(CharMap*);
    ~Scanner();

    char get();
    char peek();
    boolean push(Filename);
    void pop();
    int scan(LexSet);
    void skipwhite();
private:
    InputFile* _cur;
    InputFile** _stack;
    InputFile** _sp;
    CharMap* _map;
    char* _curline;
    char* _curchar;

    boolean getline();
};
```

It takes two to communicate

Think about protocols

Consider both caller and implementor

Keep it simple

Stable, shared protocols

InterViews subject-view protocol

- Subject represents data
- View represents user interface
- View attaches to subject
- Subject notifies views when it changes
- View detaches from subject
- Subject notifies views from its destructor

Optimize no code before its time

- C++ about as fast as C
- Use inlines sparingly, if at all
- Use virtual functions heavily
- Profile and tune when the need arises

Example:

```
int k = 0;
for (int i = 1; i ¡= w; i++) {
    for (int j = 1; j ¡= h; j++) {
        color[k++] = new Color(
            i * iinc, j * jinc, limit - k * kinc
        );
    }
}
```

Should Color::Color(int, int, int) be inline?

No, it sends an X call anyway, which costs about 10 function calls.

Young programming environments

- AT&T cfront is a toddler
- g++ is chaotic
- Slow progress on browsing/debugging

Example: Reference counting

Try #1: Base destructor calls virtual Delete

Try #2: Base::operator delete

Try #3: Base::Unref operation

Current: Global Unref operation

Concluding remarks

- C++ is much better than C
- Notation for abstraction and inheritance
- Classes, virtual functions
- Design communication between objects
- Share protocols, vary implementations
- Simplicity is important
- Write a C++ program today!