CSE 303
Lecture 22

Advanced Classes and Objects in C++

slides created by Marty Stepp
http://www.cs.washington.edu/303/
Arrays of objects

- array of objects

```cpp
Point spointarray[5]; // stack
Point* hpointarray = new Point[5]; // heap
cout << spointarray[0].getX(); // 0
```

- immediately constructs each object with () constructor
  - if no () constructor exists, a compiler error

- aoeu
Arrays of pointers

- array of pointers to objects (more common)

```cpp
Point* spointarray[5]; // stack
Point** hpointarray = new Point*[5]; // heap

for (int i = 0; i < 4; i++) {
    spointarray[i] = new Point(i, 2 * i);
    cout << spointarray[i]->getX(); // i
}
```

- each element object must be created/freed manually
Operator overloading

• **operator overloading**: Redefining the meaning of a C++ operator in particular contexts.
  - example: the `string` class overloads `+` to do concatenation
  - example: the stream classes overload `<<` and `>>` to do I/O

• it is legal to redefine almost all C++ operators
  - `() [] ^ % ! | & << >> = == != < >` and many others
  - intended to be used when that operator "makes sense" for your type
    - example: a `Matrix` class's `*` operator would do matrix multiplication
    - allows your classes to be "first class citizens" like primitives
  - cannot redefine operators between built-in types (`int + int`)

• a useful, but very easy to abuse, feature of C++ (not in C or Java)
Overloading syntax

public: // declare in .h

$returntype$ operator $op(parameters)$;

$returntype$ $classname::operator op(parameters)$ {
    statements;
    // define in .cpp
}

- most overloaded operators are placed inside a class
  - example: overriding Point + Point

- some overloaded operators don't go inside your class
  - example: overriding int + Point
Overloaded comparison ops

- Override `==` to make objects comparable like Java's `equals`
  - comparison operators like `==` return type `bool`
  - by default `==` does not work on objects (what about `Point*`?)
  - if you override `==`, you must also override `!=`

```cpp
// Point.h
bool Point::operator==(const Point& p);

// Point.cpp
bool Point::operator==(const Point& p) {
    return x == p.getX() && y == p.getY();
}
```

- Override `<`, `>`, etc. to make comparable like Java's `compareTo`
  - even if you override `<` and `==`, you must still manually override `<=`
Overriding `<<`

- Override `<<` to make your objects printable like Java's `toString`
  - note that the operator `<<` goes *outside* your class (not a member)
  - `<<` accepts a reference to the stream and to your object
  - returns a reference to the same stream passed in (why?)

```cpp
// Point.h (outside class)
std::ostream& operator<<(std::ostream& out, const Point& p);

// Point.cpp
std::ostream& operator<<(std::ostream& out, const Point& p) {
    out << "(" << p.getX() << ", " << p.getY() << ")";
    return out;
}
```

- similarly, you can override `>>` on an istream to read in an object
Suppose we want to design a class `LineSegment`, where each object represents a 2D line segment between two points.

We should be able to:
- create a segment between two pairs of coordinates,
- ask a segment for its endpoint coordinates,
- ask a segment for its length,
- ask a segment for its slope, and
- translate (shift) a line segment's position.

How should we design this class?
LineSegment.h

#ifndef _LINESEGMENT_H
#define _LINESEGMENT_H

#include "Point.h"

class LineSegment {
  private:
    Point* p1;    // endpoints of line
    Point* p2;

  public:
    LineSegment(int x1, int y1, int x2, int y2);
    double getX1() const;
    double getY1() const;
    double getX2() const;
    double getY2() const;
    double length() const;
    double slope() const;
    void translate(int dx, int dy);
  
};

#endif
#include "LineSegment.h"

LineSegment::LineSegment(int x1, int y1, int x2, int y2) {
    p1 = new Point(x1, y1);
    p2 = new Point(x2, y2);
}

double LineSegment::length() const {
    return p1->distance(*p2);
}

double LineSegment::slope() const {
    int dy = p2->getY() - p1->getY();
    int dx = p2->getX() - p1->getX();
    return (double) dy / dx;
}

void LineSegment::translate(int dx, int dy) {
    p1->setLocation(p1->getX() + dx, p1->getY() + dy);
    p2->setLocation(p2->getX() + dx, p2->getY() + dy);
}

...
Problem: memory leaks

- if we create LineSegment objects, we'll leak memory:

  ```cpp
  LineSegment* line = new LineSegment(1, 2, 5, 4);
  ...
  delete line;
  ```
  
  ▪ what memory is leaked, and why?

- the two Point objects p1 and p2 inside line are not freed
  ▪ the delete operator is a "shallow" delete operation
  ▪ it doesn't recursively delete/free pointers nested inside the object
    ▪ why not?
Destructors

public:
    ~classname(); // declare in .h

classname::~classname() { // define in .cpp
    statements;
}

• destructor: Code that manages the deallocation of an object.
  ▪ usually not needed if the object has no pointer fields
  ▪ called by delete and when a stack object goes out of scope
  ▪ the default destructor frees the object's memory, but no pointers
  • Java has a very similar feature to destructors, called a finalizer
Destructor example

// LineSegment.h
class LineSegment {
private:
    Point* p1;
    Point* p2;

public:
    LineSegment(int x1, int y1, int x2, int y2);
    double getX1() const;
    ...
    ~LineSegment();
};

// LineSegment.cpp
LineSegment::~LineSegment() {
    delete p1;
    delete p2;
}
Shallow copy bug

• A subtle problem occurs when we copy LineSegment objects:

```cpp
LineSegment line1(0, 0, 10, 20);
LineSegment line2 = line1;
line2.translate(5, 3);
cout << line1.getX2() << endl;  // 15 !!!
```

• When you declare one object using another, its state is copied
  ▪ it is a *shallow copy*; any pointers in the second object will store the same address as in the first object (both point to same place)
  ▪ if you change what's pointed to by one, it affects the other

• even worse: the same p1, p2 above are freed twice!
• **copy constructor**: Copies one object's state to another.
  ▪ called when you assign one object to another at declaration
    LineSegment line2 = line1;
  ▪ can be called explicitly (same behavior as above)
    LineSegment line2(line1);
  ▪ called when an object is passed as a parameter
    foo(line1);  // void foo(LineSegment l)...

• if your class doesn't have a copy constructor,
  ▪ the default one just copies all members of the object
  ▪ if any members are objects, it calls their copy constructors
    • (but not pointers)
Copy constructor syntax

public:
    classname(const classname& rhs);  // declare in .h

classname::classname(const classname& rhs) {
    statements;                      // define in .cpp
}

• in the copy constructor's body, do anything you need to do to properly copy the object's state
Copy constructor example

// LineSegment.h
class LineSegment {
    private:
        Point* p1;
        Point* p2;

    public:
        LineSegment(int x1, int y1, int x2, int y2);
        LineSegment(const LineSegment& line);
        ...

// LineSegment.cpp
LineSegment::LineSegment(const LineSegment& line) {
    p1 = new Point(line.getX1(), line.getY1()); // deep-copy
    p2 = new Point(line.getX2(), line.getY2()); // both points
}
Assignment bug

• Another problem occurs when we assign LineSegment objects:

```cpp
LineSegment line1(0, 0, 10, 20);
LineSegment line2(9, 9, 50, 80);
...
line2 = line1;
line2.translate(5, 3);
cout << line1.getX2() << endl;  // 15 again !!!
```

• When you assign one object to another, its state is copied
  ▪ it is a shallow copy; if you change one, it affects the other
  ▪ assignment with = does NOT call the copy constructor (why not?)

• we wish the = operator behaved differently...
Overloading =

// LineSegment.h
class LineSegment {
    private:
        Point* p1;
        Point* p2;
        void init(int x1, int y1, int x2, int y2);

    public:
        LineSegment(int x1, int y1, int x2, int y2);
        LineSegment(const LineSegment& line);
        ...
        const LineSegment& operator=(const LineSegment& rhs);
    ...
}
Overloading = , cont'd.

```cpp
// LineSegment.cpp
void LineSegment::init(int x1, int y1, int x2, int y2) {
    p1 = new Point(x1, y1);  // common helper init function
    p2 = new Point(x2, y2);
}
LineSegment::LineSegment(int x1, int y1, int x2, int y2) {
    init(x1, y1, x2, y2);
}
LineSegment::LineSegment(const LineSegment& line) {
    init(line.getX1(), line.getY1(), line.getX2(), line.getY2());
}

const LineSegment& LineSegment::operator=(const LineSegment& rhs) {
    init(rhs.getX1(), rhs.getY1(), rhs.getX2(), rhs.getY2());
    return *this;  // always return *this from =
}
An extremely subtle bug

- if your object was storing pointers to two Points p1, p2 but is then assigned to have new state using =, the old pointers will leak!

- the correction:

```cpp
const LineSegment& LineSegment::operator=(const LineSegment& rhs) {
    delete p1;
    delete p2;
    init(rhs.getX1(), rhs.getY1(), rhs.getX2(), rhs.getY2());
    return *this;  // always return *this from =
}
```
Another subtle bug

• if an object is assigned to itself, our = operator will crash!

    LineSegment line1(10, 20, 30, 40);
    ...
    line1 = line1;

• the correction:

    const LineSegment& LineSegment::operator=(const LineSegment& rhs) {
        if (this != &rhs) {
            delete p1;
            delete p2;
            init(rhs.getX1(), rhs.getY1(), rhs.getX2(), rhs.getY2());
        }
        return *this;    // always return *this from =
    }
Recap

<table>
<thead>
<tr>
<th>Point p1;</th>
<th>calls 0-argument constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point p2(17, 5);</td>
<td>calls 2-argument constructor</td>
</tr>
<tr>
<td>Point p3 = p2;</td>
<td>calls copy constructor</td>
</tr>
<tr>
<td>Point p4(p3);</td>
<td>calls copy constructor</td>
</tr>
<tr>
<td>foo(p4);</td>
<td>calls copy constructor</td>
</tr>
<tr>
<td>p4 = p1;</td>
<td>calls operator =</td>
</tr>
</tbody>
</table>

- When writing a class with pointers as fields, you must define:
  - a destructor
  - a copy constructor
  - an overloaded operator =

  conclusion: C++ blows.