If you think C++ is not overly complicated, just what is a protected abstract virtual base pure virtual private destructor and when was the last time you needed one? — *Tom Cargill*

If C++ has taught me one thing, it's this: Just because the system is consistent doesn't mean it's not the work of Satan. — *Andrew Plotkin*

David Notkin ● Autumn 2009 ● CSE303 Lecture 26

The plan

11/30 C++ intro		**12/2 C++ intro**	**12/4 C++ intro**	
12/7 Social		**12/9 Implications**	**12/11 Final prep, evaluations**	
	12/15 Final			

Constructing objects

- client code creating stack-allocated object:
  `type name(parameters);`
  `Point p1(4, -2);`
- creating heap allocated (pointer to) object:
  `type* name = new type(parameters);`
  `Point* p2 = new Point(5, 17);`

- in Java, all objects are allocated on the heap
- in Java, all variables of object types are references (pointers)

A client program

```
#include <iostream>
#include "Point.h"
using namespace std;

int main() {
    Point p1(1, 2);
    Point p2(4, 6);
    cout << "p1 is: (" << p1.getX() << ", "
         << p1.getY() << ")" << endl;  // p1 is: (1, 2)
    cout << "p2 is: (" << p2.getX() << ", "
         << p2.getY() << ")" << endl;  // p2 is: (4, 6)
    cout << "dist : " << p1.distance(p2) << endl;
    return 0;                          // dist : 5
}
```

Client with pointers

```
#include <iostream>
#include "Point.h"
using namespace std;

int main() {
    Point* p1 = new Point(1, 2);
    Point* p2 = new Point(4, 6);
    cout << "p1 is: (" << p1->getX() << ", "
         << p1->getY() << ")" << endl; // p1 is: (1, 2)
    cout << "p2 is: (" << p2->getX() << ", "
         << p2->getY() << ")" << endl; // p2 is: (4, 6)
    cout << "dist : " << p1->distance(*p2) << endl;
    delete p1;                         // dist : 5
    delete p2;   // free
    return 0;
}
```

Stack vs. heap objects

- which is better, stack or pointers?
  - if it needs to live beyond function call (e.g. returning), use a pointer
  - if allocating a whole bunch of objects, use pointers
- "primitive semantics" can be used on objects
  - stack objects behave use primitive value semantics (like ints)
- new and delete replace malloc and free
  - new does all of the following:
    - allocates memory for a new object
    - calls the class's constructor, using the new object as this
    - returns a pointer to the new object
  - must call delete on any object you create with new, else it leaks

Why doesn't this code change p1?

```
int main() {
    Point p1(1, 2);
    cout << p1.getX() << "," << p1.getY() << endl;
    example(p1);
    cout << p1.getX() << "," << p1.getY() << endl;
    return 0;
}
void example(Point p) {
    p.setLocation(40, 75);
    cout << "ex:" << p.getX() << "," << p.getY() << endl;
}
// 1,2
// ex:40,75
// 1,2
```

Object copying

- a stack-allocated object is copied whenever you:
  - pass it as a parameter **`foo(p1);`**
  - return it return p;
  - assign one object to another **`p1 = p2;`**
- the above rules do not apply to pointers
  - object's state is still (shallowly) copied if you dereference/assign
    `*ptr1 = *ptr2;`
- You can control how objects are copied by redefining the = operator for your class (ugh)

Objects as parameters

- We generally don't pass objects as parameters like this:

```
double Point::distance(Point p) {
    int dx = x - p.getX();
    int dy = y - p.getY();
    return sqrt(dx * dx + dy * dy);
}
```

- on every call, the entire parameter object p will be copied
- this is slow and wastes time/memory
- it also would prevent us from writing a method that modifies p

References to objects

- Instead, we pass a reference or pointer to the object:

```
double Point::distance(Point& p) {
    int dx = x - p.getX();
    int dy = y - p.getY();
    return sqrt(dx * dx + dy * dy);
}
```

- now the parameter object p will be shared, not copied
- are there any potential problems with this code?

const object references

- If the method will not modify its parameter, make it **`const`**

```
double Point::distance(const Point& p) {
    int dx = x - p.getX();
    int dy = y - p.getY();
    return sqrt(dx * dx + dy * dy);
}
```

- the distance method is promising not to modify p
  - if it does, a compiler error occurs
  - clients can pass Points via references without fear that their state will be changed

const methods

- If the method will not modify the object itself, make the *method* const:

```
double Point::distance(const Point& p) const {
    int dx = x - p.getX();
    int dy = y - p.getY();
    return sqrt(dx * dx + dy * dy);
}
```

- a const after the parameter list signifies that the method will not modify the object upon which it is called (this)
  - helps clients know which methods aren't mutators and helps the compiler optimize method calls
- a const reference only allows const methods to be called on it

const and pointers

- `const Point* p`
  - p points to a Point that is const; cannot modify that Point's state
  - can reassign p to point to a different Point (as long as it is const)
- `Point* const p`
  - p is a constant pointer; cannot reassign p to point to a different object
  - can change the Point object's state by calling methods on it
- `const Point* const p`
  - p points to a Point that is const; cannot modify that Point's state
  - p is a constant pointer; cannot reassign p to point to a different object
- (This is not one of the more beloved features of C++.)

Pointer, reference, etc.?

- How do you decide whether to pass a pointer, reference, or object? Some principles:
  - Minimize the use of object pointers as parameters. (C++ introduced references for a reason. They are safer and saner.)
  - Minimize passing objects by value, because it is slow, it has to copy the entire object and put it onto the stack, etc.
  - In other words, pass objects as references as much as possible; but if you *really want* a copy, pass it as a normal object.
  - Objects as fields are usually pointers (why not references?).
  - If you are not going to modify an object, declare it as const.
  - If your method returns a pointer/object field that you don't want the client to modify, declare its return type as const.

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 for 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++

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 == doesn't work on objects (what about `Point*`?)
  - if you override `==` , you must also override `!=`

```
// 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 <=

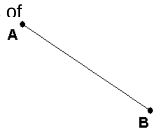
Overriding <<

- Override << to make your objects printable like Java's `toString`
  - << goes outside your class (not a member)
  - << takes a stream reference and your object
  - returns a reference to the same stream passed in

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

Designing a class

- 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 coord
  - ask a segment for its length,
  - ask a segment for its slope, and
  - translate (shift) a line segment's position.



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);
};
```

LineSegment.cpp

```
#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:

```
LineSegment* line = new LineSegment(1, 2, 5, 4);
...
delete line;
```

- 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:
  - `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 constructors

- 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 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());
    p2 = new Point(line.getX2(), line.getY2());
}
```

Assignment bug

- Another problem with assigning `LineSegment` objects:

```
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
- 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.

```
// 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!
- Instead

```
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;
```

- Instead

```
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

`Point p1;`	calls 0-argument constructor
`Point p2(17, 5);`	calls 2-argument constructor
`Point p3 = p2;`	calls copy constructor
`Point p4(p3);`	calls copy constructor
`foo(p4);`	calls copy constructor
`p4 = p1;`	calls operator =

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

Questions?