Object-oriented programming CS10003

Lecturer: Do Nguyen Kha

Contents

- Object Lifecycle
- Input & Output
- Friend Function
- UML Relationship Notations
- Constructor
- Destructor

Object Lifecycle

```
class Point2D {
private:
  double x, y;
public:
  void set(double, double);
  double getX();
  double getY();
  void move(double, double);
};
```

```
class Circle {
private:
  Point2D center;
  double radius;
public:
  void set(Point2D, double);
  void move(double, double);
  Point2D getCenter();
  double getRadius();
  double getArea();
  double getPerimeter();
};
```

Object Lifecycle

```
int main() {
  Point2D center; // Create Point2D object statically
  c.set(1, 1);
  Circle circle; // Create Circle object
  circle.set(center, 5); // Copy Point2D object
  cout << circle.getArea() << endl;
  return 0; // Destroy all objects when out of scope
}
```

Object Lifecycle

```
int main() {
  Point2D center = new Point2D(); // Create new object dynamically
  c->set(1, 1);
  Circle circle = new Circle(); // Create new object dynamically
  circle->set(*center, 5); // Dereference pointer
  delete center; // Release memory
  cout << circle->getArea() << endl;
  delete circle; // Release memory
  return 0;
}
```

Object Lifecycle

- Using operator `new` to create an object dynamically
- Using operator `delete` to destroy and release memory allocated for an object

Input & Output

- `cin`: is a built-in object with `istream` type
- `cout`: is a built-in object with `ostream` type

Usage:

```
#include <iostream>
using namespace std;
```

Input & Output

```
class Point2D {
private:
  double x, y;
public:
  …
  void input(istream&);
  void output(ostream&);
};
```

```
void Point2D::input(istream& in) {
  in >> x >> y;
}

void Point2D::output(ostream& out) {
  out << x << y;
}
```

Input & Output

```
istream& operator>>(istream& in, Point2D &p) {
  p.input(in);
  return in; // For chaining
}

ostream& operator<<(istream& in, Point2D &p) {
  p.output(out);
  return out; // For chaining
}
```

Friend Function

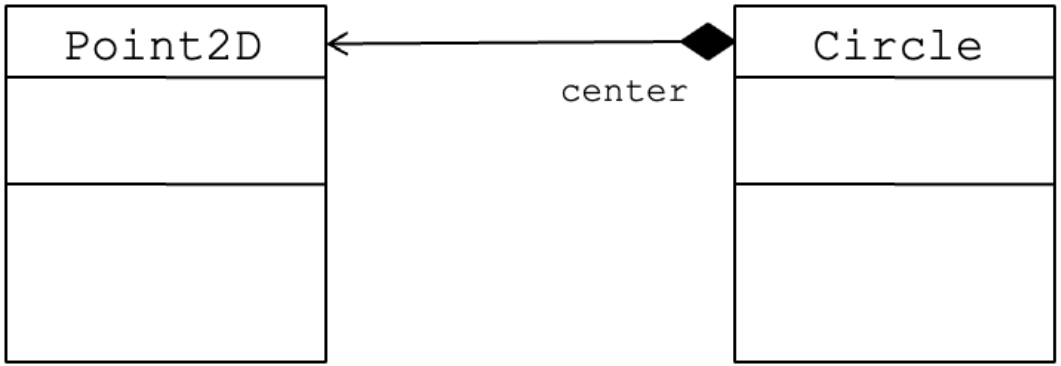
```
istream& operator>>(istream& in, Point2D &p) {
  in >> p.x >> p.y;
  // 'Point2D::x': cannot access private member declared in class 'Point2D'
  return in;
}

ostream& operator<<(istream& in, Point2D &p) {
  out << p.x << p.y;
  return out;
}
```

Friend Function

```
class Point2D {
private:
  double x, y;
public:
  ...
  friend istream& operator>>(istream& in, Point2D &p);
};
```

UML Relationship Notations



Constructor

- A constructor is a member function with the **same name** as its class
- Constructors are used to create, and can initialize, objects of their class type
- You do not specify a return type for a constructor. A return statement in the body of a constructor cannot have a return value

Constructor

```
class Point2D {
private:
  double x, y;
public:
  Point2D(); // Default constructor
  Point2D(double, double); // Custom constructor
  Point2D(const Point2D&); // Copy constructor
};
```

Default Constructor

- A default constructor is a constructor that either has no parameters, or if it has parameters, all the parameters have default values.
- If no user-defined constructor exists for a class `A`, the compiler implicitly declares a default parameterless constructor `A::A()`

Constructor

```
Point2D::Point2D() {
  this->x = this->y = 0;
}

// or

Point2D::Point2D() : x(0), y(0) {}
```

Copy Constructor

- The copy constructor lets you create a new object from an existing one by initialization
- If you do not declare a copy constructor for a class, the compiler will implicitly declare one for you, which will be an inline public member

Copy Constructor

```
Point2D::Point2D(const Point& p) {
  this->x = p.x;
  this->y = p.y;
}

// or

Point2D::Point2D(const Point& p) : x(p.x), y(p.y) {}
```

Copy Constructor

- Why do we need to define a copy constructor of a class?

Custom Constructor

```
Circle::Circle(double x, double y, double r) {
  this->center.set(x, y);
  this->r = r;
}
// or
Circle::Circle(double x, double y, double r) : center(x, y), radius(r) {}
```

Custom Constructor

```
Circle::Circle(Point2D p, double r) {
  this->center = p;
  this->radius = r;
}
// or
Circle::Circle(Point2D p, double r) : Point2D(p), radius(r) {}
```

Destructor

- Destructors are usually used to *deallocate memory* and *do other cleanup* for a class object and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted
- A destructor is a member function with the same name as its class prefixed by a **~**
- A destructor takes no arguments and has no return type
- If no user-defined destructor exists for a class, the compiler implicitly declared a destructor. This implicitly declared destructor is an inline public member of its class.

Destructor

```
class A() {
public:
  A();
  ~A(); // Destructor
};

A::~A() {
  // Clean up
}
```