Classes and Objects

Chantilly Robotics - 612

Classes

Lectures, too

What are classes?

- User-defined types
- Contain data (fields) and ways to act on that data (methods)
- Essential to Object-Oriented Programming
- Model data and actions that form a logical unit
  - Command
  - Controller
  - Motor

Defining classes

- Use the `class` keyword
- Class body is made up of members
  - Field and method declarations
- Class body must end with a semicolon
  - If you forget, an error will appear on the next line of code

```
class ClassName {
...body...
};
```

Member access levels

- `private` (default):
  - Accessible only within the class
  - Use for data, fields, implementation details
- `protected`:
  - Accessible within and to subclasses
  - Use to help for making subclasses
- `public`:
  - Accessible everywhere
  - Use for the “public API”, methods

Example class

```
class Rectangle {
public:
    Rectangle();
    Rectangle(int w, int h);
    int Width();
    int Height();
    void setWidth(int w);
    void setHeight(int h);
    int Perimeter();
    int Area();
private:
    int width;
    int height;
};
```

Methods

- Functions that act on objects of the class
- Implicit access to `this` pointer (the current object)
  - `this->height`
- Methods may access object properties
  or change object state

```
class Rectangle {
    ...
    int Width();
    int Height();
    void setWidth(int w);
    void setHeight(int h);
    int Perimeter();
    int Area();
    ...
};
```

Defining methods

- Generally done outside the class body
- Use the class name with the scope operator

```
int Rectangle::Perimeter() {
    return 2 * width + 2 * height;
}

int Rectangle::Area() {
    return width * height;
}
```

Methods vs. functions

- Methods belong to objects
- Implicitly access the current object
- Use when the behavior logically belongs to an object
  - `void Car::Drive(int miles)`

- Functions do not belong to objects
- Must be passed an object explicitly
- Use when the behavior does not logically belong to the object
  - `Car Newer(Car ls, Car rs)`

Constructors

- Use to initialize class data
- Named the same as the class name
- No return type
- Constructor overloading is allowed

```
class Rectangle {
    ...
    Rectangle();
    Rectangle(int w, int h);
    ...
};
```

Defining constructors

- Defined much like methods
- However, also may use initializer lists
  - Initializes fields in textual order
  - Simple initialization

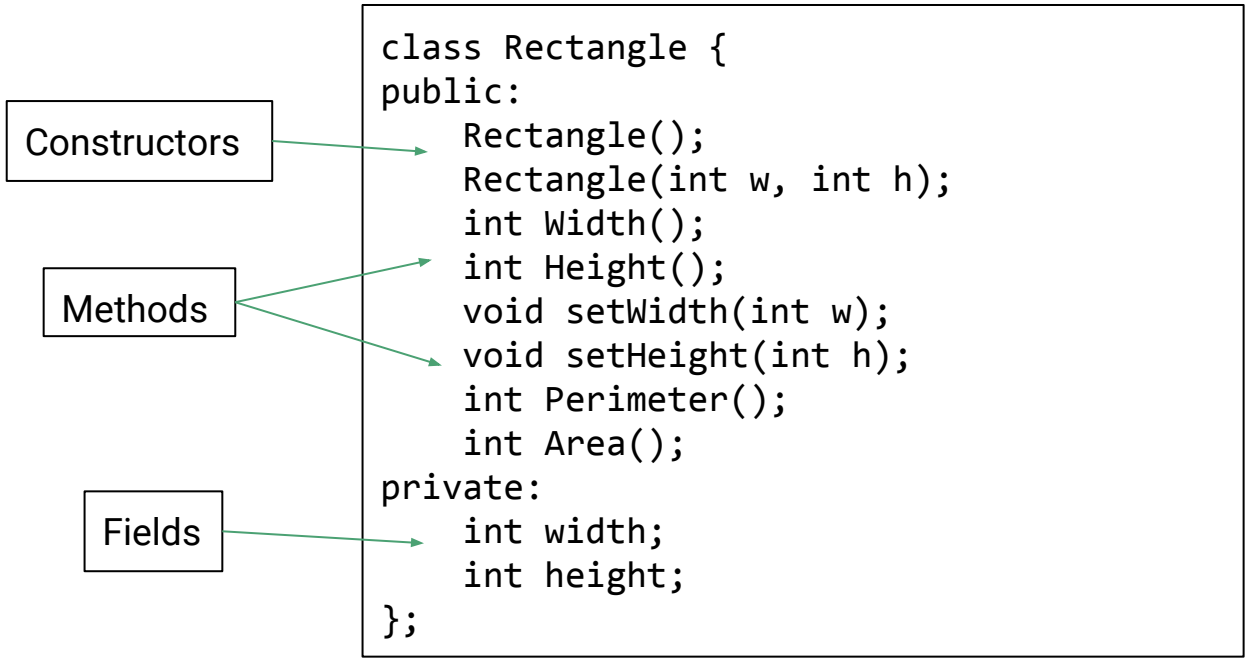
```
Rectangle::Rectangle(int w, int h) : width(w), height(h) { }

Rectangle::Rectangle() : width(1), height(1) { }
```

Destructor

- Use to clean up class data
- Release resources
- Name is the class name preceded by a tilde (~)
- No arguments for destructor
- Only one per class
- Destructors are called when the object goes out of scope

```
class Rectangle {
    ...
    ~Rectangle();
    ...
};
```

Example class



Implicit class members

- Classes implicitly have some members even if not written in the source
- 0-arg constructor
  - If no other constructors declared
  - Does nothing
- Destructor
  - Does nothing
- Copy constructor and assignment
  - Copies object fields
  - Slightly advanced topic, just be aware that these exist in classes you define/use

Accessing class members

- Use dot notation with an object
- Use arrow notation with a pointer
  - Both raw pointers and `std::shared_ptr`

```
Rectangle rect;
Rectangle* p = &rect;
rect.SetWidth(6);
p->SetHeight(7);
```

Static class members

- Declared with the `static` keyword
- Belong to the class itself, not an object
- Accessed using scope operator (`::`) with class name
  - Basically using the class as a namespace
- Useful for functions and variables with tight coupling with the class

```
class Breadfish {
    ...
    static Breadfish* MakeBreadfish();
    ...
    static int NumBreadfish;
};
```

Typedefs

- Define a type alias (another name for an existing type)
  - Shorten long or hard to read type names
- Syntax
  - `typedef` oldName newName`;`
  - Written exactly like a variable declaration
- Alternate syntax
  - `using` newName = oldName`;`

```
typedef BlurredFruitPictureThing
    BlurredFruit;

using Supplier = Breadfish(*)();
```

Objects and Pointers

Topics to be addressed

Objects

- Objects are instances of classes
- Object variables contain the data for an object
- Declaring and initializing an object is called instantiating
- Instantiate objects by calling a constructor
    - Many different ways to do so

```
Rectangle rect1;                    //default constructor
Rectangle rect2(3, 4);              //parameter constructor
Rectangle rect3(rect2);             //copy constructor
Rectangle rect4 = rect2;            //also copy constructor
```

Pointers

- Refer to the location of an object in memory
- Declared with a star (*) after the type name
- May refer to subclasses of declared type
- May also refer to `nullptr`
  - Never dereference a null pointer!
- Address operator (&) returns pointer to variable
- Dereference operator (*) returns object pointed to by pointer
- May also use `new` keyword (dynamic allocation)
  - Don’t use this with raw pointers though!

```
Rectangle rect1;
Rectangle* p1 = &rect1;
Rectangle* p2 = p1;
Rectangle rect2 = *p2;
Rectangle* p3 =
    new Rectangle();
```

Shared pointers

- Instances of the class `std::shared_ptr<T>`
  - T is the class of objects pointed to by the pointer
- Shared pointers can be copied to share the pointed to object
- Automatically manages memory from shared instances (unlike the built in pointers)
- Can only initialize with dynamic allocated object (via `new`)
  - This should be the only time you use `new`

Note: shared pointers are declared in the header `<memory>`

```
std::shared_ptr<Rectangle> p1(new Rectangle());
std::shared_ptr<Rectangle> p2 = p1;
```

`virtual` methods

As opposed to real methods

Inheritance

- Classes may inherit from other classes (zero, one, or many)
  - `public` and `protected` methods and fields are inherited from base classes
- Generally used for IS-A relationships
  - Rectangle is a shape
  - Move is a command
- Pointers may point to derived classes
  - Regular object variables may **not** contain derived class objects

```
class Breadfish : public Meme {
   ...
};
Breadfish fish;
Meme* p = &fish;
std::shared_ptr<Meme>
    p2(new Breadfish());
```

Virtual methods

- Call subclass method on superclass pointer or reference
- Polymorphism
- Derived classes may override a virtual method
- Declared using `virtual` keyword in base classes
- Declared with `override` keyword in derived classes

```
class Shape {
public:
    virtual int NumSides() {
        return -1;
    }
};

class Triangle : public Shape {
public:
    int NumSides() override {
        return 3;
    }
};
```

Virtual methods - example

```
class Shape {
public:
    //can be overridden
    virtual int NumSidesV() {
        return -1;
    }
    //cannot be overridden
    int NumSides() {
        return -1;
    }
};
```

```
class Triangle : public Shape {
public:
    //overrides
    int NumSidesV() override {
        return 3;
    }
    //overloads
    int NumSides() {
        return 3;
    }
};
```

Virtual methods - example

```
Shape s;
Triangle t;
Shape* ps = &s;
Triangle* pt = &t;
Shape* pt2 = &t;
```

The difference is here!

```
s.NumSides();      //-1
s.NumSidesV();     //-1
t.NumSides();      //3
t.NumSidesV();     //3
ps->NumSides();    //-1
ps->NumSidesV();   //-1
pt->NumSides();    //3
pt->NumSidesV();   //3
pt2->NumSides();   //-1 (!)
pt2->NumSidesV(); //3
```